Piggyback Statistics Collection for Query Optimization: Towards a Self-Maintaining Database Management System

QIANG ZHU[1], BRIAN DUNKEL[2], WING LAU[1], SUYUN CHEN[3] AND BERNI SCHIEFER[3]

[1]*Department of Computer and Information Science, The University of Michigan, Dearborn, MI 48128, USA*
[2]*Department of Electrical Engineering and Computer Science, The University of Michigan, Ann Arbor, MI 48109, USA*
[3]*IBM Toronto Laboratory, Markham, Ontario, L6G 1C7, Canada*
Email: qzhu@umich.edu

A database management system (DBMS) 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 Out-of-date statistics may lead to inefficient query processing in the system. The existing utility method, which collects statistics in batch mode, suffers from drawbacks such as heavy administrative burden, high system load and tardy updates. In this paper, we study approaches to performing statistical analysis on the fly during query execution, taking advantage of data already resident in main memory. We propose a framework for on-the-fly statistics collection, which we term piggybacking, and analyze the tradeoffs of piggybacking various statistics collection techniques on top of query execution plans. We present a multiple-granularity interleaving algorithm to integrate a set of piggyback operations with an execution plan, and show how the algorithm can be incorporated into an existing query optimizer. Our experiments demonstrate that useful statistics can be obtained via the piggyback method with a small overhead.

Received 8 January 2002; revised 28 August 2003

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 [1, 2, 3, 4, 5]. 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 the analysis on query costs [6, 7, 8, 9, 10]. The more accurate the cost analysis is, the more efficient the query processing. However, accurate cost analysis requires up-to-date statistics about the underlying database, which are maintained in the system catalog of a DBMS.

A typical statistics-collection method, which is widely used in many commercial DBMS products including DB2, Oracle, Informix and Sybase, is to invoke a utility that periodically collects and updates statistics about the underlying database [6, 7, 8, 9, 10]. We term this method the utility method. The major disadvantages of this method are:

(i) *Heavy system load*. The utility competes for system resources with other components in a DBMS and, therefore, significantly increases the system load.

(ii) *Out-of-date statistics*. To avoid heavy system load, the utility cannot be invoked very often. As a result, out-of-date statistics may be used by the query optimizer frequently, and this leads to inefficient query processing.

(iii) *Incomplete statistics*. Some statistics, such as the costs of user-defined functions (UDFs) in object-relational DBMSs, cannot be collected by the utility method. Users are required to update manually such statistics in the system catalog.

(iv) *High cost for analyzing large databases*. Unless a user 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.

(v) *Inconvenience for users*. A database administrator (DBA) has to invoke the utility manually 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 a DBA to invoke the utility rarely 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 employed to improve the utility method in some DBMSs [9, 10, 11]. 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 DBAs still need to invoke the utility manually. Furthermore, sampling itself may involve accessing a significant part of the database. Note that sampling techniques have been applied not only to estimate database statistics but also to estimate directly the sizes of query results that are needed by a query optimizer for selecting an efficient execution plan [12, 13]. In addition, people have studied the problem of constructing a random sample from a query result without computing the full result [14]. However, we are only interested in the issues for collecting/estimating database statistics in this paper.

Several on-the-fly approaches to gathering optimization information/statistics have also been studied in the field. Yu and Antoshenkov *et al.* [15, 16, 17, 18] suggested some dynamic (adaptive) query optimization techniques, which can be classified into direct and indirect ones. The direct ones dynamically optimize the current query based on runtime information, while the indirect ones collect dynamic information from the current query to optimize subsequent queries. Greenwald and Gibbons *et al.* [19, 20, 21, 22] have studied techniques of self-scaling histograms. The basic idea of self-scaling histograms is to determine dynamically the parameters (e.g. the bounds and bucket sizes) of a histogram and adjust the histogram using the data being processed during query execution. The refinement to the histogram can be done either on-line or off-line. In fact, some on-the-fly approaches to gathering statistics have been adopted in some commercial systems. For example, both IBM's DB2 [23] and iAnywhere Solutions' SQL Anywhere [24] allow statistics to be collected during the data loading process for a table. Furthermore, SQL Anywhere has also incorporated the ideas of self-scaling histograms in the system to allow histograms to be adjusted dynamically during query processing.

Clearly, 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 DBA's burden of invoking a utility manually. In this paper, we introduce a framework to gather statistics during query processing. The key idea is to 'piggyback' additional side retrievals and statistical analysis tasks on the execution plan of a user query. Although these side retrievals are not directly related to the computation of the user query's result, and may slow query execution to some extent, the statistics collected from the results of the side retrievals can be used by the query optimizer to improve the execution plans of subsequent queries. We term this on-the-fly approach to statistics collection as the piggyback method. The piggyback method meets all the requirements for statistics collection mentioned above.

Our piggybacking framework generalizes and extends various statistics collection techniques in a systematic way. The framework integrates real-time statistics gathering using both on-the-fly collection and random sampling, which can be used to estimate a statistic when it would be infeasible to compute a precise one with a specific execution plan. Moreover, the framework integrates the automatic invocation of the utility method to gather initial statistics and update expensive-to-gather statistics during off-peak hours. By supporting these techniques, the framework permits the user significant degrees of freedom in how to approach statistics gathering systematically for the entire database.

In addition, our framework enables the efficient integration of piggyback operations with query execution plans, generalizing existing on-the-fly approaches and overcoming some of their limitations. For example, our framework constitutes a generic model of statistics gathering that is more general than those approaches that center on only self-scaling histograms (i.e. focusing only on those statistics related to the distribution of column values). Perhaps more importantly, unlike known indirect query optimization techniques, our framework permits the integration of piggybacking operators that can access a larger set of data than that necessary to compute the result of a specific query.

To realize our piggybacking framework, we need to identify (1) useful piggyback side retrievals; (2) obtainable statistics via piggybacking and (3) efficient ways to integrate normal query processing operations and piggyback (statistics collection) operations. These issues will be discussed in detail in this paper. Another related issue is how a query optimizer can make use of statistics gathered/updated via the piggyback method. Since our focus is to study how to collect statistics, we assume that the collected statistics are used by query optimizers in the same way as before, with one exception: that some statistics are used to guide utility-based statistics collection, and not simply to optimize subsequent queries. How to make use of statistics in query optimization, and to what extent statistical accuracy impacts execution plan quality, varies from one system to another and depends on several factors. These factors include the specific implementation of access methods, the accuracy of employed cost models, the use of system parameters such as the buffer pool size and the number of concurrent processes, and the queries themselves. On the other hand, although no systematic study on the correlation between accuracy of statistics and quality of query optimization has been reported in the field, practitioners generally believe that more accurate statistics usually lead to more effective query optimization. Our study is also based on this widely accepted assumption, like other related research in the field [11, 12, 13, 22, 25]. Although providing a systematic validation of this assumption is beyond the scope of this paper, we will give some experimental results to demonstrate that this assumption is indeed true in a real system.

The rest of the paper is organized as follows. Section 2 introduces different types of piggyback operations. Section 3 studies the statistics obtainable from different access methods via piggyback analysis and suggests several piggybacking levels to meet users' different performance requirements. Section 4 discusses issues related to efficient interleaving of query processing operations and piggyback operations. Section 5 discusses several related issues when the piggyback method is incorporated into a DBMS. Section 6 outlines the architecture of a piggybacking prototype and presents some experimental results. Section 7 concludes the paper and lists some future research directions.

2. TYPES OF PIGGYBACK OPERATIONS

Any operation that is not required for processing a given user query but is performed for other purposes during query processing is a piggyback operation. Several types of piggyback operations can be performed during query processing, such as side data retrieval, data statistics collection and data access monitoring.[1]

2.1. Piggyback side data retrievals

One type of piggyback operation is the side data retrievals, i.e. retrieving additional data that is not required for processing the current query. There are several types of such side data retrievals, which will be discussed in the following subsections. Among them, vertical piggyback side retrievals and horizontal piggyback side retrievals form the basic ones. Using the basic ones, more complex forms of piggyback side data retrievals can be generated.

2.1.1. Vertical piggybacking

A vertical piggyback side retrieval fetches data for extra columns from an operand table(s) during query processing. Performing such a side retrieval during query processing is called vertical piggybacking. Consider the following example.

EXAMPLE 1. Consider the following user SQL query performed on a DBMS:

Q_1: SELECT DISTINCT $R_1 \cdot a_2$ FROM R_1
WHERE $R_1 \cdot a_3$ IN (SELECT $R_2 \cdot b_1$ FROM R_2);

where $R_1(a_1, a_2, a_3, a_4)$ and $R_2(b_1, b_2)$ are two tables in the underlying database. Note that the equivalent relational algebra expression, which will be used in the subsequent discussions, for query Q_1 is:

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

where π and $\bowtie$ denote the project and join operations, respectively.

One feasible execution plan to execute Q_1 is performing the following subquery $Q_1^{(2)} = \pi_{R_2 \cdot b_1}(R_2)$ first, then performing the join and final project operations by using the result of subquery $Q_1^{(2)}$. Clearly, the DBMS can collect and update statistics about column $R_2 \cdot b_1$ by analyzing the result of $Q_1^{(2)}$ during query processing.

To obtain statistics about column $R_2 \cdot b_2$, the DBMS can execute a modified subquery $Q_1^{(2)'} = \pi_{R_2 \cdot b_1, R_2 \cdot b_2}(R_2)$ 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 $R_2 \cdot b_2$), $Q_1^{(2)'}$ increases the processing cost of query Q_1 only slightly. In fact, we have piggybacked the vertical side retrieval: $\pi_{R_2 \cdot b_2}(R_2)$ during the processing of user query Q_1 to obtain necessary statistics with a small additional overhead.

In general, consider a user query Q with operand tables $R_1, R_2, \ldots, R_n$. To process Q, a DBMS usually performs a subquery[2]

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

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, F_i is the qualification condition of the select operation, and CL_i is the target list of the project operation. We call $Q^{(i)}$ an access subquery for table R_i. For instance, for Q_1 in Example 1, two access subqueries are

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

and

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

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

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

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

$$F(Q^{(1)}, Q^{(2)}) = \pi_{Q^{(1)} \cdot a_2}(Q^{(1)} \underset{Q^{(1)} \cdot a_3 = Q^{(2)} \cdot b_1}{\bowtie} Q^{(2)}).$$

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,

$$\omega_{R_2 \cdot b_2}(Q_1^{(2)}) = \pi_{R_2 \cdot b_1, R_2 \cdot b_2}(\sigma_{\text{true}}(R_2))$$

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

[1] This paper only considers the piggyback operations related to statistics collection although other piggyback operations are also possible.

[2] 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 still included in the target list CL_i of the project operation, which simplifies the definition of horizontal piggybacking in the next subsection.

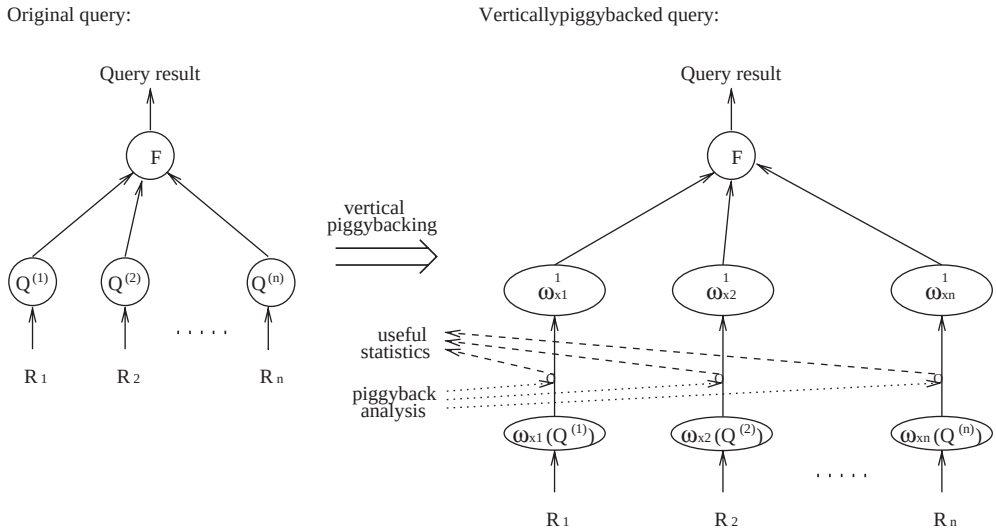

FIGURE 1. Idea of vertical piggybacking.

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 $CL_i' = CL_i \cup X$. As the above equation implies, ω^{-1} is called the inverse vertical piggybacking operator of ω. Note that the net effect of an inverse vertical piggybacking operator is to keep the original project operation after the vertically piggybacked subquery. Furthermore, the original project operation does not have to apply directly to the vertically piggybacked subquery. In other words, some intermediate operations can exist in between, as long as the final result does not change.

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

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

is called a vertically piggybacked query for Q. If at least one vertically piggybacked subquery $\omega_{X_i}(Q^{(i)})$ $(1 \le i \le n)$ 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 (Figure 1) which can be used to produce useful statistics that may improve optimization quality for subsequent queries. How to determine the piggybacked column list X is to be discussed in Section 3.2.

2.1.2. Horizontal piggybacking

A horizontal piggyback side retrieval fetches extra rows from an operation table(s) during query processing. Performing such a side retrieval during query processing is called horizontal piggybacking. Consider the following example.

EXAMPLE 2. For the following user query performed on a DBMS:

Q_2: SELECT DISTINCT $R_1 \cdot a_1, R_1 \cdot a_2$ FROM R_1 WHERE $R_1 \cdot a_1 > 3$;

(i.e. $\pi_{R_1 \cdot a_1, R_1 \cdot a_2}(\sigma_{R_1 \cdot a_1 > 3}(R_1))$) where $R_1 \cdot a_1$ is indexed while $R_1 \cdot a_2$ is not, a feasible execution plan is to retrieve the qualified rows from R_1 via the index on $R_1 \cdot a_1$. Several statistics on $R_1 \cdot a_1$ 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 $R_1 \cdot a_2$ may not be accurately known since not all data values of $R_1 \cdot a_2$ are obtained (e.g. if it is a sparse index). An improvement may be made by utilizing all rows in the retrieved pages from the data file as sample rows to estimate the statistics on $R_1 \cdot a_2$ if the conventional assumptions that the columns in a table are independent (i.e. the independence assumption) and the column values are uniformly (randomly) distributed in the table (i.e. the uniformity assumption) hold. 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 improve further the sample set. Tradeoffs need to be made between the overhead and accuracy. Note that if the independence and uniformity assumptions are not valid for the underlying table, we have several options to mitigate the problem. The first option is to use adaptive page sampling via cross-validation [21]. The key idea is to use all tuples in the retrieved pages but adjust the amount of sampling (with additional pages) depending on the correlation of column

values in retrieved pages. The correlation is tested via cross-validation from two samples. The second option is to adopt a revised backing sample approach [20], which, in turn, is based on reservoir sampling [26]. The basic idea is to maintain a separate backing sample used to estimate statistics and choose some, rather than all, tuples in the retrieved pages to keep the backing sample up-to-date. The last option is to disallow such horizontal piggybacking, which is not assumed to be the choice for the rest of this paper.

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

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

then $\gamma_Y(Q^{(i)})$ is non-trivial. Otherwise, it is trivial. For example, for query $Q_2^{(1)} = Q_2$ in Example 2,

$$\gamma_Y(Q_2^{(1)}) = \pi_{R_1 \cdot a_1, R_1 \cdot a_2}(\sigma_{R_1 \cdot a_1 > 3 \vee Y}(R_1)),$$

where Y represents predicate '$R_1 \cdot pid \in RB$', $R_1 \cdot pid$ denotes the identifier of the page containing the current row being examined in R_1, and RB denotes the set of identifiers of the pages containing at least one qualified row.

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. Note that the net effect of an inverse horizontal piggybacking operator is to keep the original select operation after the horizontally piggybacked subquery. Furthermore, the original select operation does not have to apply directly to the horizontally piggybacked subquery. In other words, some intermediate operations can exist in between as long as the final result does not change.

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

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

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. How to select the piggybacking qualification Y_i is to be discussed in Section 3.3.

2.1.3. 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 (3) to yield a mixed vertical and horizontal piggybacked subquery:

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

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

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

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

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

2.1.4. Multi-query piggybacking

From Section 2.1.2, we know that the data values obtained via the horizontal piggybacking can be used as a sample set for statistical analysis. Extra sample pages besides the retrieved pages from a data file may be used to improve the sample set. However, fetching extra sample pages requires additional overhead.

To improve 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 a sample set. In this way, the sample size is increased. Note that piggyback analysis may be performed on a query-by-query basis (i.e. via pipelining) without waiting for the final sample set to be formed. For example, consider the average length of values in a column, which is a statistic used in query optimization. Note that the length of a value is the number of digits (for an integer) or the number of bytes/characters (for a character string). Let S_i be the set of values for column $R \cdot a$ retrieved by query Q_i $(1 \leq i \leq n)$. Clearly, the total length $\text{len}(S_i)$ 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 $\text{avg\_len}(R \cdot a)$ of column $R \cdot a$ for the final sample set $S = \cup_{i=1}^{n} S_i$ can be estimated as follows:

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

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

TABLE 1. Typical statistics maintained in a catalog.

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 (e.g. average no. of bytes used by values for a *varchar* column)
Table statistics	T_1	Number of rows in a table
	T_2	Number of pages used by a table
	T_3	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 k ($\geq$1) columns 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

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. For simplicity, multi-query piggybacking will not be discussed further in the rest of this paper.

2.2. Piggyback data statistics collection

Retrieving additional data during query processing is not our ultimate goal. Our goal is to gather useful statistical information for query optimization. Hence statistics collection operations need to be piggybacked on query processing.

Different DBMSs may maintain different types of statistics in their catalogs for query optimization. Table 1 shows typical statistics maintained in a DBMS such as DB2. There are some other statistics not shown in the table, which are mainly used by a DBA to monitor system performance and decide how to reconfigure and/or reorganize the database.

The statistics for query optimization can be partitioned into logical and physical categories:

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

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

To obtain statistical information, we have three types of statistics collection operations. The first type of operation calculates (exact) statistics based on a complete data set. We use notation $\varphi_s(x)$ to denote an operation that calculates statistic s for data object x, where s is a statistic in Table 1. For example, $\varphi_{C_1}(R \cdot a)$ and $\varphi_{T_1}(R)$ are to find (calculate) the exact maximum value of column $R \cdot a$ and cardinality of table R respectively. The second type of operation estimates (approximates) statistics based on a sample data set. We use notation $\varepsilon_s(x)$ to denote an operation that estimates statistic s for data object x. Although an estimated statistic may not be as good as an exact statistic from the users' perspective, it usually requires less overhead to obtain. The third type of operation validates whether a statistic is up-to-date or not. We use notation $\upsilon_s(x)$ to denote an operation that validates statistic s for data object x.

There are several methods to validate a statistic. Method I is to calculate the new statistic and compare it with the existing statistic in the system catalog. Since the statistic is calculated, the overhead for $\upsilon_s(x)$ is about the same as that for $\varphi_s(x)$ in this case. Method II for validation is to compare the updating timestamp t_1 of data object x and the updating timestamp t_2 of statistic s. If $t_1 \leq t_2$, s is up-to-date. Although the validation overhead in this approach is smaller than that for $\varphi_s(x)$, we can only get a sufficient condition for a statistic to be up-to-date. That is, if $t_1 > t_2$, we cannot conclude whether s is up-to-date or not. Method III for validation is to make use of a sufficient condition for a statistic to be out-of-date. For example, for statistic C_1 in Table 1, we have the following sufficient condition for it to be out-of-date:

$$(\exists x \in S \text{ such that } x > C_1) \Rightarrow (C_1 \text{ is out-of-date}), \quad (3)$$

where S is the set of retrieved values during query processing and $\Rightarrow$ denotes logical implication. Similarly, we have the following sufficient condition for statistic C_2 to be out-of-date:

$$(\exists x \in S \text{ such that } x < C_2) \Rightarrow (C_2 \text{ is out-of-date}). \quad (4)$$

Clearly Method I is, in fact, making use of a sufficient and necessary condition for a statistic to be up-to-date. Hence, in this case, we can conclude either $\upsilon_s(x) = 1$ (true) or $\upsilon_s(x) = 0$ (false). For Method II, we can only conclude $\upsilon_s(x) = 1$ but not $\upsilon_s(x) = 0$; while for Method III, we can only conclude $\upsilon_s(x) = 0$ but not $\upsilon_s(x) = 1$.

We also employ a locality principle for statistics validation. That is, if a sufficient number of statistics on a given data object x (e.g. a table) are found to be out-of-date, all the other statistics for x are assumed to be out-of-date. The reason to employ such a conservative rule is that updating a statistic that is already up-to-date will not make the statistic out-of-date (idempotent property) except some overhead may be incurred. To complement the limited sufficient conditions

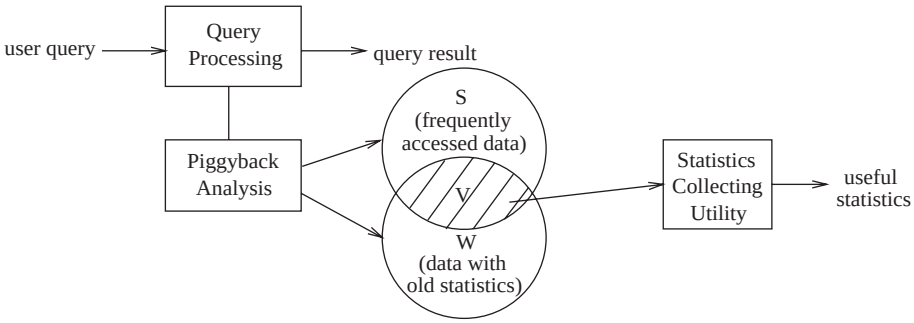

FIGURE 2. Useful lightweight piggybacking.

we can find for $\upsilon_s(x) = 0$, we also use some special validation statistics (not used in query optimization) for the data objects being considered such as the average value $\text{avg}(R \cdot a)$ of a column a in a given table R. If the estimated validation statistic(s) for a given data object x is found to have been changed significantly, all the statistics for x are assumed to be out-of-date.

2.3. Piggyback access statistics monitoring

To reduce the overhead for collecting data statistics during query processing, we can just count the access frequency (access statistic) of each data object (table, column or index). No data statistics are directly collected or estimated during query processing in this case. Based on the access statistics, we can identify the set S of most frequently accessed data objects. 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. This observation is consistent with the one made by Lynch for incorporating users' estimates in query optimization [27]. 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 are already up-to-date. In this case, there is no need to update the statistics on x. We can apply the validation scheme introduced previously to determine the set of data objects with out-of-date statistics. Let W be the set of data objects whose statistics are found to be out-of-date, i.e.

$$W = \{x \mid \exists\ s \text{ such that } \upsilon_s(x) = 0\}.$$

Then the set $V = S \cap W$ contains the data objects whose statistics need to 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 invoked automatically by the system during off-peak hours without DBA interference. Hence the user's burden of manually invoking the utility is relieved. Figure 2 shows such a useful lightweight piggybacking procedure. In the rest of this paper, we use $\rho(x)$ to denote the access statistics monitoring operation that updates the access frequency for object x.

3. PIGGYBACKING LEVELS

Piggybacking can be done at different levels in terms of quality and quantity of the statistics to be gathered. The higher quality or larger quantity of the statistics we want, the greater the piggybacking overhead. On the other hand, certain levels of piggybacking may not be feasible for a particular query processing procedure since no common work exists. In general, if one level of piggybacking is feasible for a particular query processing procedure, all lower levels of piggybacking are also feasible. Hence, it is possible to downgrade piggybacking from one level to a lower level if a lower overhead is desired.

3.1. Accuracy levels of statistical analysis

The quality of a (data) statistic is reflected in its accuracy level. We define three levels of accuracy for statistics collection, i.e. calculation, estimation and validation, corresponding to the three piggyback (data) statistics collection operations $\varphi_s(x)$, $\varepsilon_s(x)$ and $\upsilon_s(x)$ respectively, described in Section 2.2.

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:

Sequential scan (SS): scan the rows in a table sequentially.
Index scan (IS): retrieve the qualified rows via an index.
Index-only access (IOA): get all requested values from an index tree.
Hash access (HA): retrieve the qualified rows via a hash table.

Statistics can be obtained during the execution of an access method although not all statistics at any level are obtainable via every method. Table 2 shows what statistics may be obtained (at what level) during the execution of different access methods. Note that: (1) approximate estimation ($\oplus$) is valid only if the sample size is sufficiently large (e.g. it meets a minimum percentage threshold value); and (2) not

TABLE 2. Statistics obtainable via access methods.

	C_1	C_2	C_3	C_4	C_5	T_1	T_2	T_3	I_1	I_2	I_3	I_4	I_5	I_6	I_7
SS	√	√	√	√	√	√	√	√	×	⊕	√	√	×	×	×
IS															
(I) Full	(√, √)	(√, √)	(√, √)	(√, √)	(√, √)	√	√	√	√	√	√	√	√	√	√
(II) $a < \beta_1,\ a > \beta_2$	(√, ⊙)	(√, ⊙)	(⊕, ⊕)	(⊕, ⊕)	(⊕,⊕)	⊕	⊕	⊕	⊕	√	⊕	⊕	⊕	⊕	⊕
(III) $a < \beta_1$	(⊙, ⊙)	(√, ⊙)	(⊕, ⊕)	(⊕, ⊕)	(⊕, ⊕)	⊕	⊕	⊕	⊕	√	⊕	⊕	⊕	⊕	⊕
(IV) $a > \beta_2$	(√, ⊙)	(⊙, ⊙)	(⊕, ⊕)	(⊕, ⊕)	(⊕, ⊕)	⊕	⊕	⊕	⊕	√	⊕	⊕	⊕	⊕	⊕
(V) Multiple ranges	(⊙, ⊙)	(⊙, ⊙)	(⊕,⊕)	(⊕, ⊕)	(⊕, ⊕)	⊕	⊕	⊕	⊕	√	⊕	⊕	⊕	⊕	⊕
(VI) $a = \beta_1$	(⊙, ⊙)	(⊙, ⊙)	(×, ×)	(×, ×)	(×, ×)	×	×	×	×	√	×	×	×	×	×
IOA															
(I) Full	(√,×)	(√,×)	(√,×)	(√,×)	(√,×)	×	√	×	√	√	√	√	√	√	√
(II) $a < \beta_1,\ a > \beta_2$	(√,×)	(√,×)	(⊕, ×)	(⊕, ×)	(⊕, ×)	×	⊕	×	⊕	√	⊕	⊕	⊕	⊕	⊕
(III) $a < \beta_1$	(⊙, ×)	(√,×)	(⊕, ×)	(⊕, ×)	(⊕, ×)	×	⊕	×	⊕	√	⊕	⊕	⊕	⊕	⊕
(IV) $a > \beta_2$	(√,×)	(⊙, ×)	(⊕, ×)	(⊕, ×)	(⊕, ×)	×	⊕	×	⊕	√	⊕	⊕	⊕	⊕	⊕
(V) Multiple ranges	(⊙, ×)	(⊙, ×)	(⊕, ×)	(⊕, ×)	(⊕, ×)	×	⊕	×	⊕	√	⊕	⊕	⊕	⊕	⊕
(VI) $a = \beta_1$	(⊙, ×)	(⊙, ×)	(×, ×)	(×, ×)	(×, ×)	×	⊕	×	×	√	×	×	×	×	×
HA	(⊙, ⊙)	(⊙, ⊙)	(×, ⊕)	(×, ⊕)	(×, ⊕)	⊕	⊕	⊕	×	×	×	×	×	×	×

'√'—accurate stats; '⊕'—estimated stats via sampling; '⊙'—validity inf. of stats; '×'—no obtainable/applicable stats; 'a'—an indexed column; 'β_1', 'β_2'—constants, $\beta_1 < \beta_2$; '(?,??)'—? is for the indexed/hashed column and ?? is for other columns.

all cases for possible validation (⊙) are listed since:

(i) Although the timestamp approach to validate up-to-date statistics (i.e. $\upsilon_s(x) = 1$) can be applied to any case, we are more interested in identifying out-of-date statistics (i.e. $\upsilon_s(x) = 0$) as we will see later.
(ii) We do not consider situations in which statistics are assumed to be out-of-date based on the locality principle or a validation statistic (see Section 2.2).

For the sequential scan method, since the whole data file of a table is scanned, all column and table statistics can be calculated accurately during its execution. However, since indexes are not accessed, most index statistics cannot be obtained; only I_3 and I_4, which are logical statistics, can be accurately calculated. Based on them I_2 can be estimated.

For the index scan method, there are several cases. Case (I) 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. For cases (II)–(V), if the retrieved values of an indexed column a from the index tree cover ranges $a < \beta_1$ and/or $a > \beta_2$ (where β_1 and β_2 are constants), statistics C_2 and/or C_1 can be obtained accurately. Otherwise, they may be verified based on conditions (3) and/or (4) from Section 2.2. Since the index tree is accessed in all these cases, 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 if the sample set is acceptable. For case (VI), since the number of retrieved values is usually very small, we assume the retrieved values are not sufficient to form a useful sample set. Hence most statistics are not obtainable in this case. Note that a sample set can be improved by applying horizontal piggybacking or multi-query piggybacking if the sample size is too small. It should also be pointed out that, for simplicity, we consider only single-column indexes. If indexes on multiple columns are considered, Table 2 needs to be extended.

Obtaining statistics for the index-only access method is similar to the index scan. However, since the underlying table is not accessed, statistics for non-indexed columns or the underlying table (except statistic T_2) are not obtainable. Statistic T_2 can be obtained or estimated based on the block/page id's in the index tree. In addition, statistic T_1 may also be obtained or estimated when the referenced index(es) is a unique index.

For the hash access method, conditions (3) and (4) in Section 2.2 can still 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 in this case.

3.2. Vertical piggybacking levels

Note that, at the low implementation 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 included extra columns for piggyback analysis with different levels, depending on the user's tolerance for piggybacking overhead. The more overhead allowed, the greater the number of extra columns may be included in piggyback analysis.

With regard to different levels of vertical piggybacking, we divide the columns of an operand table R_i for a given query

Q into the following four classes:

$AC_1 = \{x \mid x$ is a column in $R_i \ \wedge \ x$ is referenced in $Q\}$,
$AC_2 = \{x \mid x$ is an indexed column in $R_i\} - AC_1$,
$AC_3 = \{x \mid x$ is a column in $R_i \ \wedge$ (x is part of the primary key $\vee$ x is part of a foreign key $\vee$ x is referenced by a foreign key $\vee$ x has a unique constraint)$\}$ $- (AC_1 \cup AC_2)$
$AC_4 = \{x \mid x$ is a column in $R_i\} - (AC_1 \cup AC_2 \cup AC_3)$.

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, candidate and/or foreign keys, i.e. those in class AC_3. We term such a column a key-related column. The remaining columns are in class AC_4, which may be included for piggyback analysis if the piggybacking overhead can be tolerated by the user.

Let

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

The following subquery

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

extended from query $Q^{(i)}$ in (1) is called the vertically piggybacked subquery at level V_k. A vertically piggybacked query Q is at level V_k if at least one of its vertically piggybacked subqueries is at level V_k. However, to simplify implementation, we recommend having all vertically piggybacked subqueries in Q at the same level, and this is assumed in 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, i.e. they can be evaluated via the 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 divided further into two subclasses for given query Q:

$AC_{21} = \{x \mid x$ is a column in R_i with an index referenced in $Q\} - AC_1$,
$AC_{22} = \{x \mid x$ is an indexed column in $R_i\}$ $- (AC_1 \cup AC_{21})$.

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 for the columns in subclass AC_{21} may be obtained from the referenced indexes when they are fetched into memory during query processing. Since the columns in subclass AC_{22} only have non-referenced indexes, to obtain their values may require additional I/O cost. Hence, AC_{21} is assigned a higher priority than AC_{22}. The vertical piggybacking level V_2 is then divided into two sublevels. The other vertical piggybacking levels may also be divided into sublevels when necessary. For simplicity, we will not consider the sublevels in the following discussion.

3.3. Horizontal piggybacking levels

To determine a horizontally piggybacked subquery $\gamma_{Y_i}(Q^{(i)})$ from subquery $Q^{(i)}$ in (1), 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 piggybacking 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; i.e. the horizontally piggybacked subquery is trivial. Although there is no piggybacking overhead in this case, statistics that we obtain may be poor. Such a trivial horizontally piggybacked subquery is said to be at the level H_0. At the other extreme, $Y =$ true; i.e. all rows in the operand table will be fetched. Although we can obtain all statistics about the table in this case, the piggybacking overhead may be high. Such a full horizontally piggybacked subquery is said to be at the level H_3.

There are two useful cases between the two extremes. Let

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

where $F_{i|y}$ denotes the truth value of condition F_i when instantiated by row y from table R_i, and $y \cdot pid$ is the identification number[3] 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:

$$Y_{|x} = \text{true} \quad \text{if and only if} \quad x \cdot pid \ \in \ RB,$$

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. Such a horizontally piggybacked subquery is said to be at the level H_1.

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

$$Y_{|x} = \text{true} \quad \text{if and only if} \quad x \cdot pid \ \in \ RB \vee x \cdot pid \ \in \ SB.$$

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 may be improved by using

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

these additional sample rows. The sample pages can be chosen randomly from the data file. The number of additional sample pages depends on the given threshold value for the required sample size. Note that if a strong correlation among column values in retrieved pages exists, we can apply the method of adaptive page sampling via cross-validation mentioned in Example 2. Such a horizontally piggybacked subquery is said to be at the level H_2.

3.4. Integrated piggybacking levels

In general, piggyback analysis can be performed at different levels. At one end of the spectrum, no piggyback analysis is performed during query processing, and statistics for query optimization are collected by applying the utility method. There is no piggybacking 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 for the accessed data objects must be evaluated, even if that data is unnecessary to compute the result of the specific query. 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 piggybacking overhead. The more overhead allowed, the more statistics together with better accuracy may be obtained. To achieve this goal, we define the following levels of piggybacking:

(i) *Level* L_0: No piggyback analysis is performed during query processing.
(ii) *Level* L_1: Frequencies of data objects accessed by queries are recorded, and the validity of relevant statistics is checked during query processing.
(iii) *Level* L_2: Statistics on the accessed index(es), column(s) (i.e. vertical piggybacking level V_1) and table(s) are collected and/or estimated during query processing.
(iv) *Level* L_3: Statistics at level L_2 as well as those on other indexed columns (i.e. vertical piggybacking level V_2) in a referenced table are collected and/or estimated during query processing.
(v) *Level* L_4: Statistics at level L_3 as well as those on other key-related columns (i.e. vertical piggybacking level V_3) in a referenced table are collected and/or estimated during query processing.
(vi) *Level* L_5: Statistics at level L_4 as well as those on the remaining columns (i.e. vertical piggybacking level V_4) in a referenced table are collected and/or estimated during query processing.
(vii) *Sub-level* L_i^j: Each level from L_2 to L_5 can be divided into four sub-levels based on the horizontal piggybacking levels, i.e. sub-level L_i^j of level L_i ($i = 2, 3, 4, 5$) performs piggyback analysis based on data from horizontal piggybacking at level H_j ($j = 0, 1, 2, 3$).

At piggybacking level L_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. To collect statistics, the DBA must invoke the statistics collecting utility manually.

Piggybacking level L_1 corresponds to the useful light-weight piggybacking depicted in Figure 2. No statistics on data in the database are actually collected during query processing at this level of piggybacking.

At piggybacking level L_2, statistics are collected and/or estimated for the data objects (index(es), column(s) and table(s)) that are referenced in a query.

At piggybacking levels L_3–L_5, statistics on extra columns that are included by the vertically piggybacked queries at levels V_2–V_4 (respectively, see Section 2.1.1) are collected and/or estimated.

Note that piggybacking levels L_2–L_5 can be divided further into sublevels by using horizontally piggybacked queries. At sublevel L_i^0, only those rows necessary to compute the query result are considered for piggyback analysis. At sublevel L_i^1, each row, whether qualifying or not, from any retrieved page are considered for piggyback analysis. At sublevel L_i^2, rows in additional sample pages are also used for piggyback analysis. At sublevel L_i^3, rows from all data pages are used for piggyback analysis. Note that, if the sequential scan method is invoked, sublevels L_i^1–L_i^3 are the same, i.e. rows from all data pages is considered since all data pages are retrieved during query processing.

Usually, the higher the level (sublevel) is, the higher the piggybacking overhead. A proper piggybacking level and its sublevel in a system can be determined according to a user-specified tolerance of overhead.

There are several tradeoffs a user must consider when deciding on a piggybacking level. First, the overhead of various piggyback operations varies, sometimes significantly. For a chosen piggybacking level, performance of most queries may be affected only slightly, but some queries may be substantially slower. Second, although piggybacking techniques minimize additional I/O cost for statistics collection, they increase CPU consumption, which sometimes may be significant [28]. Hence the system's CPU capabilities and memory resources should be considered when determining a piggybacking level. Third, piggyback overhead increases with concurrent requests due to the locking implications of retrieving additional data in the face of concurrent updates. Fortunately, locks placed due to piggyback statistics collection operations could be released earlier. Moreover, execution of some piggyback operations could be done in the background, since the correctness criterion for statistics collection is not as high as the one for query processing. However, additional overhead due to the concurrency control is inevitable. A user may desire a lower piggybacking level in a heavily loaded environment, or conversely raise it in a lightly loaded environment. It would be desirable for the system

TABLE 3. Example logical query processing operations.

Operation	Symbol	Description
Select	σ	Given a set of input rows, produce qualified rows according to a given selection criterion
Project	π	Given a set of input rows, produce rows with a subset of columns according to a given set of column names, eliminating duplicate rows
Join	$\bowtie$	For two given sets of input rows, perform a join based on a given condition and produce the output rows
Aggregate	Varies	An aggregation operation is one of `MAX`, `MIN`, `AVG`, `COUNT` or `SUM` that, given a set of input rows, produces an appropriate value for the given (set of) column(s)

to set automatically the piggybacking level, taking into account system load and system performance criteria set by the user, but techniques for doing so are beyond the scope of this paper.

4. EFFICIENT INTERLEAVING OF QUERY PROCESSING AND PIGGYBACK OPERATIONS

When a DBMS supports piggybacking level L_i^j for $i \geq 2$, data statistics are analyzed and collected during query processing. An important issue is how to perform efficiently the piggyback statistics collection operations in conjunction with normal query processing in the DBMS. In this section, we present a multiple-granularity interleaving technique to integrate efficiently a set of piggyback operations with a given user query.

4.1. User query and piggyback operations

For a given DBMS, there are a defined set of query processing primitives/operations at both the logical and physical levels. Tables 3 and 4 list some example logical and physical query processing operations. The piggyback (statistics collection) operations considered here are $\varphi_s(x)$, $\varepsilon_s(x)$ and $\upsilon_s(x)$ that were discussed in Section 2.2.

The operations in Table 3 are most relevant when query processing is viewed from a design perspective, whereas those in Table 4 are most relevant during physical implementation. Also, we recognize that there is a difference between the semantics of SQL and (classical) relational algebra for some operations, because the former follows the semantics that duplicates are not necessarily eliminated (unless the keyword DISTINCT is specified), while the latter follows set semantics, in which duplicate elements are identical. This is why we define both the π and the π' operations and consider π' to be a 'physical' operation, for example. π' followed by unique operation μ is equivalent to π. If duplicate rows are tolerated in a query result in practice, π' can be used without μ in query processing.

TABLE 4. Example physical query processing operations.

Operation	Symbol	Description
Duplicate project	π'	Given a set of input rows, produce rows with a subset of columns according to a given set of column names without duplicate elimination (bag semantics)
Unique	μ	Remove duplicate values from a given set
Scan	Σ or `SCAN`	Given a set of elements or a composite input, produce each element (or sub-component) one at a time
Index scan	`ISCAN`	Produce each element of a given input one at a time, using an index
Gather	Γ	Collect component elements and produce an element at a coarser granularity level
Index operations	$\delta, \iota, \kappa, \lambda, \chi$	These operations take an input table and implicitly associated index to produce one of: a set of ids for qualified blocks (δ), the index itself (ι), the data blocks for a set of ids (κ), the set of nodes along a path from the root to the first qualified leaf node (λ) and the set of qualified leaf nodes via sibling pointers (χ)
Block nested loop join	`BNLJ`	Produce the join result of the two given sets of rows using a block nested loop join strategy
Loop	`LOOP`	Produce a number of copies of an input stream, specified by the given parameter
Sort	`SORT`	Reorder the values of the input stream according to the specified sort criterion
Count	`COUNT`	Count the number of elements in the input stream

EXAMPLE 3. As a concrete example of how we use the notation to represent these operations, consider the simple SQL query SELECT DISTINCT a_1, a_4 FROM R_1 WHERE $a_3 > 200$. This can be written as $Q_3 = \pi_{a_1,a_4}(\sigma_{a_3>200}(R_1))$. If we would like to compute the maximum value of column a_3 in R_1, then our piggyback operation can be represented as $P_1 = \varphi_{C_1}(R_1 \cdot a_3)$. The goal of the piggyback method in this example is to effect an efficient interleaving of the two operations Q_3 and P_1. Since the input operands of both operations are the same, the piggyback method can take advantage of the data transfer from disk required by the user query to perform the tasks necessary for the piggyback operation(s). In the rest of this section, we discuss various elements of our technique for automating this process.

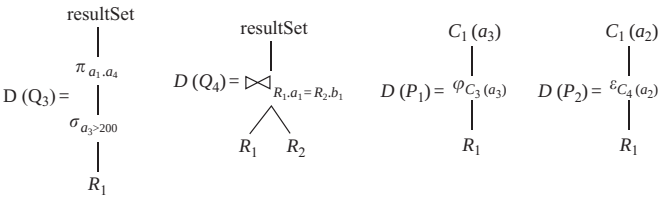

FIGURE 3. Example data flow plans.

4.2. Data flow plans

A relational algebra expression[4] for a query is often represented as a query tree. Such a tree diagram implies a certain flow of data, with the details being left to the implementation and physical representation, which we refer to as a high-level data flow plan for the query. When we put detailed implementation information into the plan, it becomes a low-level data flow plan, or often called as an execution plan. As we will see, piggybacking can be integrated with a data flow plan at various levels. We will use a similar data flow plan to represent each piggyback operation. For a query or piggyback operation $\otimes$, we use $D(\otimes)$ to denote its associated data flow plan being considered. Figure 3 shows some examples of data flow plans. Note that since the piggyback method encompasses both the physical and logical views of data for query processing, we must reflect various data granularities explicitly in our representation of a data flow plan. We will discuss this aspect of data flow plans and their representation in Section 4.4.

In order to avoid using extremely complicated diagrams, we will not indicate the control flow in these plans directly, but rather take the convention that each of the operations in a data flow plan such as those shown in Figure 3 represents an iterator construct that takes an input object and produces output elements one at a time, as consumed by the next operation in the data flow path.

Although the graphical representation of Figure 3 is useful for presenting data flow relationships, space constraints lead us to use an equivalent algebraic representation that is more compact. In this notation, $A \rightarrow \otimes \rightarrow B$ indicates that A is the input to operation $\otimes$ and B is the output. For operations that have multiple inputs (e.g. join), we use a notation similar to the Backus–Naur form of production grammars to represent the connections between the various nodes in a data flow plan. For example, $D(Q_4)$ (shown graphically in Figure 3) would be represented by $\{R_1 \rightarrow \alpha;\ R_2 \rightarrow \alpha;\ \alpha \rightarrow \bowtie_{R_1.a_1=R_2.b_1} \rightarrow \mathit{resultSet}\}$, where the additional symbol α simply represents a common point in the data flow plan.

4.3. Data semantics of operations

In order to manipulate data flow plans to generate correct results, we must always guarantee that the semantics of interleaved operations do not differ from the non-interleaved operations. For this purpose, we define a number of operation classes (specifically for transformations on objects consisting of collections of fixed-size vectors or rows) depending on the data semantics preserved by the operations.

We say that an operation $\otimes$ preserves the full semantics of its data input if there exists an inverse transformation operation (at least conceptually), $\otimes^{-1}$, such that $X \rightarrow \otimes \rightarrow \otimes^{-1} \rightarrow X$ for any valid data input X. In other words, the representation of X may change, but no information is lost from the input to the output of $\otimes$. If an operation $X \rightarrow \otimes \rightarrow Y$ preserves the full semantics of its input X, then we say that its output Y is semantically equivalent to input X (indicated as $Y \equiv_S X$). As an example, the sort operation preserves the full semantics of its input.

If R is a table, we use notation $\mathcal{S}(R)$ to represent its schema, or set of column identifiers associated with each row in R. A table R' is a vertical subset of another table R (written as $R' \subseteq_V R$) if (1) $\mathcal{S}(R') \subseteq \mathcal{S}(R)$ and (2) there is a bijective mapping between the row identifiers of R and R' such that each row (value) in R' is contained in the corresponding row (value) in R. A table R' is a strict vertical subset of another table R (written as $R' \subset_V R$) if $\mathcal{S}(R) - \mathcal{S}(R') \neq \emptyset$. R' is a horizontal subset of another table R (indicated as $R' \subseteq_H R$), if there exists a bijective mapping between the rows of R and $R' \cup (R - R')$[5] such that each row (value) in R equals the corresponding row (value) in $R' \cup (R - R')$. R' is a strict horizontal subset of R (written $R' \subset_H R$) if $R - R' \neq \emptyset$.

An operation $\otimes$ on a set of rows is said to be a vertical reduction if there exists some input X such that $X \rightarrow \otimes \rightarrow Y$ and $Y \subseteq_V X$. If an operation $\otimes$ performs no strict vertical reduction on any input, it is said to preserve vertical semantics. An operation $\otimes$ on a set of rows is said to be a horizontal reduction if there exists some input X such that $X \rightarrow \otimes \rightarrow Y$ and $Y \subseteq_H X$. Operations which preserve horizontal semantics perform no strict horizontal reduction on any input. As an example, the physical project operation π' (without duplicate elimination) preserves horizontal semantics, and the select operation preserves vertical semantics.

Table 5 shows the semantics-preserving properties of some relational query and piggyback operations.[6] The scan (Σ) and gather (Γ) operations will be described in greater detail in Section 4.4. The standard aggregate functions of SQL (i.e. `MAX`, `MIN`, `AVG`, `SUM` and `COUNT`), indicated in the table by *aggr*, all have the same semantics preservation behavior, as do all the piggyback statistics collection operations, represented by *stat* in the table.

4.4. Data granularities

In order to interleave operations at multiple levels of data granularity, we introduce a notation for describing explicitly the granularity at which a given operation is processed. The

[4] To simplify our discussion, we consider only the relational algebra operations listed in Table 3 here, which are also the operations that can be most effectively interleaved with piggyback operations. However, our integration techniques can be extended to handle many other operations.

[5] We assume that each row in a table has a unique (row) identifier. $R - R'$ contains those rows in R whose identifiers are not for any row in R'.

[6] Assume that the operations in Table 5 are non-trivial.

TABLE 5. Semantics-preserving properties.

	$out \equiv_S in$	$out \subset_H in$	$out \subset_V in$
sort	Yes	No	No
Σ	Yes	No	No
Γ	Yes	No	No
σ	No	Yes	No
π'	No	No	Yes
π	No	No	Yes
μ	No	Yes	No
$\bowtie$	No	No	No
aggr	No	No	No
stat	No	No	No

TABLE 6. Data granularities.

Data granularity	Notation
Scalar value	v
Set of distinct scalar values	S
Index node	n
Set of distinct nodes (e.g. tree, path)	T
Vector or row	t
Block of rows	b
Set of distinct blocks (extent)	E
Set of distinct rows (table)	R

graphical notation $\boxed{\otimes}\,{}^{g_2}_{g_1}$ indicates that operation $\otimes$ has an input granularity of g_1 and an output granularity of g_2, where g_1 and g_2 are not necessarily the same.

Table 6 lists the data granularities that we have identified as important in the description of data flow for the various user query and piggyback statistics collection operations. The table also gives a notational identifier for each level of data granularity. The vector-based data collections of Table 6 have a sequential containment relationship to one another. In other words, a block/page (b) is made up of a collection of rows (t), a block extent (E) is composed of multiple blocks and a table (R) consists of multiple extents.

As an example of application for these granularities, consider a query involving a selection σ operation. Figure 4a shows explicitly that both the input and output objects are at the table (granularity) level (as indicated by R). However, in order for the selection and any piggyback operations to be interleaved row by row, we must consider an equivalence between the σ operation at the row level and the table level.

Figure 4b shows the explicit row-level (t) selection operation in a way that is logically equivalent to the table-level σ operation. So, Figure 4b also illustrates the fact that a table can be converted into a stream of rows ($\boxed{\Sigma}\,{}^{t}_{R}$) and that a stream of rows can be gathered into a single table ($\boxed{\Gamma}\,{}^{R}_{t}$). Note that the equivalence shown in Figure 4 is bi-directional, and that we can expand or collapse operations as necessary to represent them at the appropriate level of data granularity for efficient interleaving.

As in Section 4.2, the graphical representation of Figure 4 is illustrative, but space constraints lead us to use the

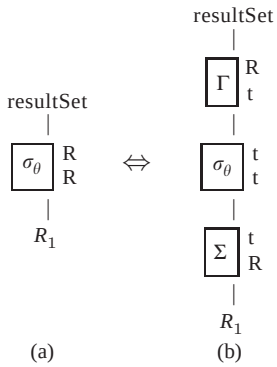

FIGURE 4. Operations with explicit data granularities.

equivalent algebraic representation, augmented with explicit indicators of data granularity. In this notation,[7] $A \rightarrow \otimes|^{g_2}_{g_1} \rightarrow \oplus|^{g_3}_{g_2} \rightarrow B$ indicates that A is the input to operation $\otimes$, which has an input granularity of g_1 and an output granularity of g_2. The output of $\otimes$ is then the input to $\oplus$, and B is the output of the entire process. Note that the output granularity of $\otimes$ must match the input granularity of $\oplus$. So, in this notation, Figure 4a would be represented as $R_1 \rightarrow \sigma_\theta|^{R}_{R} \rightarrow$ *resultSet*.

For operations that have multiple inputs (e.g. join) or that have outputs connected to multiple consumers as a result of combining multiple data flow plans, we again use the same notation described in Section 4.2, augmented with explicit data granularities. For example, a table-level join $A \bowtie B$ that produces a table-level output C would be represented by $\{A \rightarrow \alpha;\ \ B \rightarrow \alpha;\ \ \alpha \rightarrow \bowtie|^{R}_{R,\,R} \rightarrow C\}$.

4.5. Multiple-granularity query processing

The basic idea of the piggyback method is to modify the normal processing of a user query so that statistical information about relevant data can also be efficiently collected during its execution. In this subsection, we discuss how to interleave the operations contained in a query data flow plan with a set of compatible piggyback operations.

4.5.1. Multiple-granularity interleaving algorithm

Our algorithm for multiple-granularity interleaving integrates the data flow plan of a user query with a set of piggyback operations so that the operations can be processed efficiently, subject to the performance tradeoffs of the current piggybacking level. We assume that such a data flow plan for the user query is determined by the query optimizer prior to the execution of this algorithm. However, the data flow plans for the piggyback operations as well as their integration with the data flow plan for the user query are to be determined by this algorithm. For each piggyback operation, there may be multiple feasible data flow plans, each of which represents one data flow path[8] from the underlying table for the argument to the output. We can view all the feasible data flow plans (paths) for a piggyback

[7] Symbols $\otimes$, $\oplus$ and $\ominus$ will be used to denote different operations.

[8] Unlike a data flow plan for a piggyback operation, the data flow plan for a user query may have multiple data flow paths from its leaves to its root.

operation together as a non-deterministic data flow plan for it. The task of our algorithm is to determine a data flow path in the non-deterministic plan that can be most efficiently interleaved with the data flow plan for the given user query. Knowledge is assumed to be embedded in the system itself about which statistics are possible to collect via piggybacking and which data flow paths for each piggyback operation can be instantiated for the current query. In the following discussion, we use $\widetilde{D}(\oplus)$ to denote the non-deterministic data flow plan for piggyback operation $\oplus$.

Note that the data flow plan for a user query from a query optimizer may not explicitly express all available levels of data granularity in the system. Hence our algorithm must first expand the plan to include the implicit levels implemented in the system.

For each user query over which piggybacking is to take place, our algorithm runs as follows:

ALGORITHM I. Multiple-granularity interleaving for query and piggyback operations

- **Input:** *Data flow plan $D(Q)$ for user query Q, as produced by the query optimizer*
- **Output:** *Integrated data flow plan for query Q and the relevant piggyback operations*

(1) *Expand all data flow paths in $D(Q)$ to match available levels of data granularity and physical characteristics implemented by the system, using transformation rules F4, F5 and F7 described in Section 4.5.3.*

(2) *Determine the set, O, of data objects (i.e. tables, indexes and columns) that are referenced in $D(Q)$.*

(3) *Augment Q (and its $D(Q)$) vertically to a piggybacking level L_k ($1 \leq k \leq 5$), according to the desired overhead tolerance. Denote the vertically augmented plan as $D_k(Q)$.*

(4) *If $k > 2$, determine the set O' of additional column objects according to piggybacking level L_k and let $O = O \cup O'$.*

(5) *Based on the objects in O, instantiate the set $P = \{P_1, P_2, \ldots, P_n\}$ of potential piggyback operations for query Q. Note that, unless the piggybacking level is L_1, each piggyback operation in P is assumed to achieve the highest accuracy level (i.e. $\varphi_s(x)$ to calculate an exact statistic s for object $x \in O$). If the piggybacking level is L_1, for each object $x \in O$, set P contains one or more $\upsilon_s(x)$'s (to validate the data statistics on x) and one $\rho(x)$ (to update the access statistic/frequency for x).*

(6) *For each piggyback operation $P_i \in P$, find the table object $o_j \in O$ such that the argument of P_i is related to o_j. Note that there may be multiple references to a single underlying table object in a query data flow plan, and we must consider each of these references as possible candidates for piggybacking. For all such objects o_j's:*

(a) *Merge each path (feasible plan) in $\widetilde{D}(P_i)$ with a path starting from an instance of o_j in $D_k(Q)$ as far as possible toward the root. If there exists at least one path in $\widetilde{D}(P_i)$ that can be merged, choose the best of these paths (denoted by $D(P_i)$) and loop to P_{i+1}. The process of merging and ranking different possible paths will be discussed in Section 4.5.2.*

(b) *If no merging is possible for any instance of o_j in $D_k(Q)$, downgrade P_i to P_i' (if possible), and attempt to merge the paths in $\widetilde{D}(P_i')$ with a path from an instance of o_j in $D_k(Q)$. If there exists at least one path that can be merged, choose the best one (denoted by $D(P_i)$) and loop to P_{i+1}.*

(c) *If no merging for the downgraded operation P_i' is possible, augment Q horizontally to a piggybacking sub-level L_k^j ($1 \leq j \leq 3$) at a proper point on the paths from an instance of o_j to the root in $D_k(Q)$, according to the overhead tolerance specified by the user, and attempt to merge paths in $\widetilde{D}(P_i')$ with the horizontally augmented plan $D_k^j(Q)$. If there exists at least one path that can be merged, choose the best one (denoted by $D(P_i)$) and loop to P_{i+1}.*

(d) *If no merging for $\widetilde{D}(P_i')$ and $D_k^j(Q)$ is possible, further downgrade P_i' to P_i'' (if possible), and attempt to merge paths in $\widetilde{D}(P_i'')$ with $D_k^j(Q)$. If there exists at least one path that can be merged, choose the best one (denoted by $D(P_i)$) and loop to P_{i+1}.*

(e) *If no merging is possible after downgrading P_i and augmenting Q, no statistical information will be collected by P_i. Loop to P_{i+1}.*

(7) *Remove useless non-trivial vertical and horizontal piggybacking operators from $D_k^j(Q)$ if any.*

(8) *Collapse unnecessary expanding in the integrated data flow plan, using transformation rules F4, F5, F6 and F7 described in Section 4.5.3.*

(9) *Where possible, combine piggyback operations of the same type connected at the same point into an equivalent vector operation. For example, if the maximum values of three different columns in a table are being collected during the same scan, all should be collected as a single (vector) operation.*

(10) *Return the integrated*[9] *data flow plan $D_k^j(Q) + D(P_1) + \cdots + D(P_n)$.*

In fact, there are three phases in Algorithm I, i.e. the preparing phase (Steps 1–5), the merging phase (Steps 6(a)–(e)) and the cleaning phase (Steps 7–10).

The preparing phase identifies the potential piggyback operations for a given query and prepares the query data flow plan for piggybacking. Specifically, Step 1 expands the data flow plan for query Q to the implemented levels of data granularity and physical characteristics so that more piggyback operations may have a chance to merge with

[9] We use the '+' symbol in the algorithm only to indicate that the data flow plans have been combined in some appropriate way.

Q. Step 2 identifies the set O of data objects that are related to query Q. To maximize set O so that more piggyback operations can be integrated, Step 3 applies vertical piggybacking to include more column data objects. Note that additional objects are added into the set only if the piggyback level is L_k with $k \geq 2$. Step 4 adds such objects into the set. Step 5 determines the set P of potential piggyback operations that can be performed on objects in O. Unless the piggybacking level is L_1, in which case the piggyback operations update access frequencies and validate statistics for the accessed data objects to realize the lightweight piggybacking depicted in Figure 2, the piggyback operations are initially set to calculate the relevant statistics. Note that a piggyback operation may be downgraded to a lower accuracy level (e.g. estimate rather than calculate a statistic) later on.

The merging phase tries to integrate each potential piggyback operation with the given query plan. Step 6(a) merges the common work between the piggyback operation and the given query as much as possible. When no merging is possible, Step 6(b) downgrades the piggyback operation to a lower accuracy level (if possible). If the downgraded operation still cannot be merged, Step 6(c) augments the query (plan) horizontally to increase the chance for merging. If it still fails, Step 6(d) downgrades the piggyback operation further (if possible). If no merging is possible after all downgrading and augmenting efforts, the algorithm skips the current piggyback operation and moves to the next one.

The cleaning phase cleans up unnecessary augmenting/expanding done in the previous steps and combines some operations into one if possible. Note that vertical and horizontal piggybacking retrieves extra columns and rows from a table, but does not guarantee that the extra data is always used for piggyback operations (Step 6(e)). Step 7 removes useless augmenting from the final plan. Similarly, expanding some paths in the query data flow plan does not guarantee that the expanded details are useful for piggyback operations. Step 8 collapses unnecessary expanding in the query plan. Sometimes several piggybacked operations at the same point in a path of the data flow plan can be combined into one vector operation that can be processed simultaneously to improve efficiency, which is done at Step 9.

In the following subsections, we elaborate on some more details of the algorithm. In particular, we describe the three basic strategies for integrating two data flow plans in Section 4.5.2. In Section 4.5.3, we provide some basic transformation rules and heuristics that can be used by the algorithm to determine when a particular integration is possible or more likely to be efficient. These techniques are demonstrated in Section 4.5.4, where we give some concrete examples of how the data flow plans for a user query and a set of piggyback operations can be integrated.

4.5.2. *Integrating data flow plans*

We have identified three general classes of techniques to integrate data flow plans for a user query and its compatible piggyback statistics collection operations:

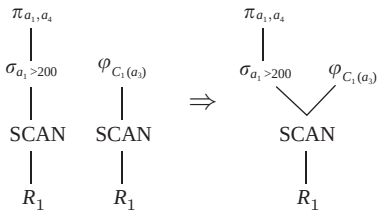

FIGURE 5. Merged data flow plan for Q_3 and P_1.

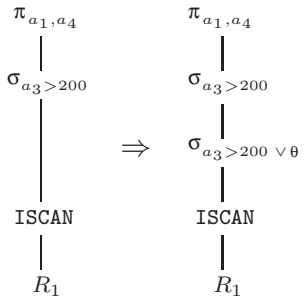

FIGURE 6. Augmented data flow plan for Q_3.

- *Merge* (see Step 6(a))—If an initial sequence of sub-operations from two data flow plans performs the same set of manipulations on the same data input, they can be performed at the same time, rather than repeated for each data flow plan. The reading of data blocks is a good example of this technique, as shown in Figure 5, using the data flow plans for Q_3 and P_1 from Figure 3. Note that transformation rules F1–F3, which will be discussed in the next subsection, allow merging to be also done for two data flow plans with convertible initial sequences (not necessarily identical).
- *Augment* (see Steps 3 and 6(c))—This technique applies only to user queries. A query (and its plan) can be augmented vertically, i.e. vertical piggybacking discussed in Section 2.1.1, to obtain more statistics if the overhead tolerance specified by the user allows. On the other hand, if a user query retrieves a given amount of data and a particular piggyback operation would only be possible with a larger set of data, the original query plan can be augmented horizontally as discussed in Section 2.1.2. As an example, an added node in the plan could permit additional rows to be read into memory, and a subsequent plan node would filter these additional rows to return only those rows required by the query. This is demonstrated in Figure 6, where the presumption is that the larger set of data introduced by the additional condition θ would, for example, provide better estimates of some statistics for columns other than a_3.
- *Downgrade* (see Steps 6(b) and (d))—This technique applies only to piggyback operations. When the available data for a query is not sufficient to perform

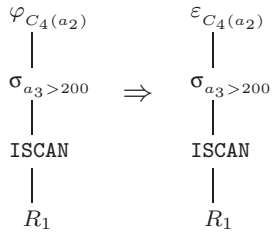

FIGURE 7. Downgraded data flow plan for P_1.

a more accurate statistical analysis, the given operation may be downgraded. For example, if a table is not read by a full scan, but a sufficient number of blocks are retrieved, a piggyback operation to find the distribution of a column could be downgraded to an operation to estimate or verify the distribution. Thus, the plan on the left in Figure 7 could have been the original piggyback operation requested, but the plan on the right is the result of downgrading.

When comparing different ways to merge the paths in the (deterministic) data flow plan for a query with the paths in the non-deterministic data flow plan for a piggyback operation, we must estimate the utility of each choice in order to determine the 'best'. Since the benefit of merging the paths from two data flow plans is to exploit common processing, paths should be merged 'as much as possible'. If we weight the sub-operations in a data flow path according to the amount of work performed for the path, we can use the amount of common work divided by the amount of total work to evaluate the benefit of that choice. The actual values for the weighting functions associated with different paths should reflect the relevant heuristics described in the next section.

For example, one data flow path for finding the maximum value of a column might perform a sort on that column (the first sub-operation, representing 99% of the total effort) followed by a retrieval of the first row from the sorted result (the second sub-operation, representing 1% of the total effort). An alternative data flow path might perform a full scan of the table (the first sub-operation, representing 80% of the total effort) with a test of each row to determine if the maximum should be updated (the second sub-operation, representing 20% of the total effort). If only the first sub-operation from the two data flow paths for the piggyback operation can be shared with the paths from a query data flow plan, merging the first data flow path is preferred since 99% of the work is shared, compared with 80% with the second path.

4.5.3. Transformations and heuristics

When considering possible interleavings of two or more data flow plans, it is useful to take advantage of a number of transformation rules that we have identified, presented in the following list. We use $\mathcal{S}_{\text{in}}(\otimes)$ to indicate the input schema required by an operation $\otimes$ with a set of rows as input and $\mathcal{S}_{\text{out}}(\otimes)$ to indicate the output schema produced.

F1: For $X \to \otimes \to Y$, if $Y \equiv_S X$, then $\{X \to \oplus\} \Rightarrow \{Y \to \oplus\}$
In other words, if an operation $\otimes$ preserves the full semantics of its input X, then its output Y may be connected to another operation $\oplus$ requiring X as an input.

F2: For $X \to \otimes \to Y$, if $Y \subseteq_V X$ and $\mathcal{S}_{\text{in}}(\oplus) \subseteq \mathcal{S}_{\text{out}}(\otimes)$, then $\{X \to \oplus\} \Rightarrow \{Y \to \oplus\}$
In other words, if an operation $\otimes$ performs a vertical reduction, then Y can replace X as an input only to operations requiring an acceptable vertical subset of X. For example, if $\mathcal{S}(X) = \{a_1, a_2, a_3\}$ and $\otimes = \pi'_{a_2,a_3}$, then Y can only be used as an input to operations $\oplus$ for which $\mathcal{S}_{\text{in}}(\oplus) \subseteq \{a_2, a_3\}$.

F3: For $X \to \otimes \to Y$, if $Y \subseteq_H X$ and Y is an acceptable subset[10] of X for operation $\oplus$, then $\{X \to \oplus\} \Rightarrow \{Y \to \oplus\}$
In other words, if an operation $\otimes$ performs a horizontal reduction, then Y can replace X as an input only to operations requiring an acceptable horizontal subset of X.

F4: $\{X \to \otimes|_{g_1}^{g_3} \to Y\} \Leftrightarrow \{X \to \Sigma|_{g_1}^{g_2} \to \otimes|_{g_2}^{g_3} \to Y\}$
Scan operations (Σ) can be added/removed at the input to make a given operation $\otimes$ accept an input at a finer/coarser level of data granularity.

F5: $\{X \to \otimes|_{g_1}^{g_3} \to Y\} \Leftrightarrow \{X \to \otimes|_{g_1}^{g_2} \to \Gamma|_{g_2}^{g_3} \to Y\}$
Gather operations (Γ) can be added/removed to make a given operation $\otimes$ produce an output at a finer/coarser level of data granularity.

F6: $\{\oplus|_{g_1}^{g_2} \to \otimes|_{g_2}^{g_3} \to \otimes^{-1}|_{g_3}^{g_2} \to \ominus|_{g_2}^{g_4}\} \Rightarrow \{\oplus|_{g_1}^{g_2} \to \ominus|_{g_2}^{g_4}\}$
In other words, a pair of mutual inverse operations (such as Γ and Σ) with appropriately matching granularities can be collapsed and replaced by an identity operation.

There are other transformation rules that are not listed above. For example,

F7: $\{X \to \pi|_{\mathsf{R}}^{\mathsf{R}} \to Y\} \Leftrightarrow \{X \to \pi'|_{\mathsf{R}}^{\mathsf{R}} \to \mu|_{\mathsf{R}}^{\mathsf{R}} \to Y\}$

Such rules are mainly used to give more implementation details (i.e. physical characteristics) of an operation, which is needed when interleaving at a higher level is impossible. If no interleaving is eventually done at a lower level, the implementation details can also be hidden by applying the reverse transformation. To simplify the algorithm description, we use label F7 to represent a class of such transformation rules.

We have also identified a number of heuristics intended to improve the efficiency of the data flow plan integration techniques described in Section 4.5.2, which include:

H1: Prefer finer levels of granularity to piggyback.
Sharing of data between two plans should generally be done at the finest available levels of data granularity. So block or block set (extent) levels are preferable to

[10] Acceptance can be determined, for example, by a threshold value for a relative sample size.

table level, and row level is preferred to block level. Finer levels of granularity allow for a tighter integration between the query and piggyback operations, sharing the associated overhead.

H2: Merge as much as possible.
When two plans can be merged, it is most efficient to merge as many of the common sub-operations along the path toward the root of the tree as possible. This maximizes the common work that can be shared.

H3: Merge to the minimum support point.
In other words, operations should share data at the point where the minimum data requirement of a given operation is met. For example, given a choice between merging to before or after a σ operation (implying that the output of σ is acceptable), merge to beyond the σ, so that the smaller set of data will be used to perform the piggyback operation. The overhead is therefore minimized.

H4: Minimize augmenting and downgrading.
In general, query plans should be augmented only as necessary, since augmentation usually implies more work than the user query in isolation. Piggyback operations should be downgraded as little as possible, so as to guarantee the quality of the statistics that can be collected by piggybacking.

H5: Combine individual piggyback operations into a single operation if possible.
For example, some column-statistics collection operations such as finding the maximum and minimum column values on the same table can be combined into a single vector operation and performed during one scan of the table.

4.5.4. *Examples of interleaving*

To show how Algorithm I makes use of the integration techniques and transformation rules described in Sections 4.5.2 and 4.5.3 to interleave the data flow plans for a user query and the relevant piggyback operations, let us consider two examples in this subsection. Once a fully integrated data flow plan has been built, it can be mapped into a set of manipulations to be performed on each data element as it is made available in memory.

EXAMPLE 4. Let us again consider the simple query Q_3 in Example 3 in Section 4.1 (Figure 3), whose plan can be represented as

$$D(Q_3) = \left\{ R_1 \rightarrow \sigma_{a_3>200}\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \pi_{a_1,a_4}\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \mathit{resultSet} \right\}.$$

Step 1 in Algorithm I expands the query plan using transformation rules F4, F5 and F7 as follows:

$$\begin{aligned} D(Q_3) = \Big\{ & R_1 \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \sigma_{a_3>200}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \\ & \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \\ & \rightarrow \pi'_{a_1,a_4}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \mu\Big|_{\mathsf{R}}^{\mathsf{R}} \\ & \rightarrow \mathit{resultSet} \Big\}. \end{aligned}$$

In this example, we focus on illustrating the merging technique and do not consider non-trivial vertical or horizontal piggybacking. In other words, Steps 3 and 6(c) in Algorithm I do not add any additional columns or rows, assuming the piggybacking level is L_2^0. Steps 2–3 identify the set of relevant data objects $O = \{R_1, a_1, a_3, a_4\}$, assuming no index is referenced. Step 5 determines the set of potential piggyback operations for objects in O. Examples of such piggyback operations include: $P_1 = \varphi_{C_1}(a_3)$, $P_3 = \varphi_{T_2}(R_1)$ and $P_4 = \varphi_{T_1}(R_1)$, whose data flow plans are as follows:

$$\begin{aligned} D(P_1) &= \left\{ R_1 \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \varphi_{C_1(a_3)}\Big|_{\mathsf{t}}^{\mathsf{v}} \rightarrow C_1(a_3) \right\}, \\ D(P_3) &= \left\{ R_1 \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \varphi_{T_2(R_1)}\Big|_{\mathsf{b}}^{\mathsf{v}} \rightarrow T_2(R_1) \right\}, \\ D(P_4) &= \left\{ R_1 \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \varphi_{T_1(R_1)}\Big|_{\mathsf{t}}^{\mathsf{v}} \rightarrow T_1(R_1) \right\}. \end{aligned}$$

Note that the data flow plans for the piggyback operations are specified at the same levels of data granularity and physical characteristics as those for the query plan to facilitate interleaving with the query, and we use the same notations for various statistics in Table 1.

If we merge two plans $D(Q_3)$ and $D(P_1)$ following Step 6(a) in Algorithm I, we get a combined data flow plan of

$$\begin{aligned} & D(Q_3) + D(P_1) \\ & \quad = \Big\{ R_1 \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \alpha; \\ & \qquad \alpha \rightarrow \sigma_{a_3>200}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \\ & \qquad\quad \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \pi'_{a_1,a_4}\Big|_{\mathsf{t}}^{\mathsf{t}} \\ & \qquad\quad \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \mu\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \mathit{resultSet}; \\ & \qquad \alpha \rightarrow \varphi_{C_1(a_3)}\Big|_{\mathsf{t}}^{\mathsf{v}} \rightarrow C_1(a_3) \Big\}. \end{aligned}$$

$D(P_3)$ and $D(P_4)$ can also be merged into the combined plan in a similar way as follows:

$$\begin{aligned} & D(Q_3) + D(P_1) + D(P_3) + D(P_4) \\ & \quad = \Big\{ R_1 \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \beta; \\ & \qquad \beta \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \alpha; \\ & \qquad \beta \rightarrow \varphi_{T_2(R_1)}\Big|_{\mathsf{b}}^{\mathsf{v}} \rightarrow T_2(R_1); \\ & \qquad \alpha \rightarrow \sigma_{a_3>200}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \\ & \qquad\quad \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \pi'_{a_1,a_4}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \\ & \qquad\quad \rightarrow \mu\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \mathit{resultSet}; \\ & \qquad \alpha \rightarrow \varphi_{C_1(a_3)}\Big|_{\mathsf{t}}^{\mathsf{v}} \rightarrow C_1(a_3); \\ & \qquad \alpha \rightarrow \varphi_{T_1(R_1)}\Big|_{\mathsf{t}}^{\mathsf{v}} \rightarrow T_1(R_1) \Big\}. \end{aligned}$$

Finally, Step 8 in Algorithm I collapses the plan according to F4, F5 and F6 to get

$$\begin{aligned} &D(Q_3) + D(P_1) + D(P_2) + D(P_3) \\ &\quad = \Big\{ R_1 \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{b}} \rightarrow \beta; \\ &\qquad \beta \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \alpha; \\ &\qquad \beta \rightarrow \varphi_{T_2(R_1)}\Big|_{\mathsf{b}}^{\mathsf{v}} \rightarrow T_2(R_1); \\ &\qquad \alpha \rightarrow \sigma_{a_3>200}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \pi'_{a_1,a_4}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{R}} \\ &\qquad\quad \rightarrow \mu\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \mathit{resultSet}; \\ &\qquad \alpha \rightarrow \varphi_{C_1(a_3)}\Big|_{\mathsf{t}}^{\mathsf{v}} \rightarrow C_1(a_3); \\ &\qquad \alpha \rightarrow \varphi_{T_1(R_1)}\Big|_{\mathsf{t}}^{\mathsf{v}} \rightarrow T_1(R_1) \Big\}. \end{aligned}$$

EXAMPLE 5. Let us consider another example that illustrates the techniques of augmenting a user query and downgrading a piggyback operation. Let $R_3(d_1, d_2, d_3, d_4, d_5)$ be a table, which has only one index $Idx(d_2)$ and two key-related columns d_2 (primary key) and d_3 (referenced by a foreign key). Consider a query Q_5 whose data flow plan generated by the query optimizer is as follows:

$$\begin{aligned} D(Q_5) = \Big\{ & R_3 \rightarrow \texttt{ISCAN}_{d_2>300}\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \pi'_{d_1,d_2,d_4}\Big|_{\mathsf{E}}^{\mathsf{E}} \\ & \rightarrow \sigma_{d_2>300}\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \pi_{d_1,d_4}\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \mathit{resultSet} \Big\}. \end{aligned}$$

After being expanded by Step 1 in Algorithm I, the plan becomes:

$$\begin{aligned} D(Q_5) = \Big\{ & R_3 \rightarrow \texttt{ISCAN}_{d_2>300}\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \pi'_{d_1,d_2,d_4}\Big|_{\mathsf{E}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \\ & \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \sigma_{d_2>300}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \\ & \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \pi'_{d_1,d_4}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \\ & \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \mu\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \mathit{resultSet} \Big\}. \end{aligned}$$

Step 2 in Algorithm I identifies the set of relevant data objects $O = \{R_3, d_1, d_2, d_4, Idx(d_2)\}$. Assume that the user-specified piggyback level is L_4^2. Steps 3–4 add object d_3 into set O, i.e. $O = \{R_3, d_1, d_2, d_3, d_4, Idx(d_2)\}$. The vertically augmented plan is as follows:

$$\begin{aligned} D_4(Q_5) = \Big\{ & R_3 \rightarrow \texttt{ISCAN}_{d_2>300}\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \pi'_{d_1,d_2,d_3,d_4}\Big|_{\mathsf{E}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \\ & \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \sigma_{d_2>300}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \\ & \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \pi'_{d_1,d_4}\Big|_{\mathsf{t}}^{\mathsf{t}} \\ & \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \mu\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \mathit{resultSet} \Big\}, \end{aligned}$$

which is almost the same as the previous $D(Q_5)$ except that column d_3 has been added into the target list of π' following `ISCAN`.

One of the potential piggyback operations determined by Step 5 is $P_5 = \varphi_{C_5}(d_3)$, i.e. calculate the average column length of d_3. Clearly, no feasible data flow plan of P_5 can be merged with $D(Q_5)$ by Step 6(a) in Algorithm I since there is no full scan for d_3. Hence, we downgrade P_5 to $P'_5 = \varepsilon_{C_5}(d_3)$ according to Step 6(b) in Algorithm I, i.e. to estimate the statistic rather than calculate the exact value. A feasible data flow plan for P'_5 is:

$$\begin{aligned} D(P'_5) = \Big\{ & R_3 \rightarrow \texttt{ISCAN}_{d_2>300 \vee \theta}\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \pi'_{d_3}\Big|_{\mathsf{E}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \\ & \rightarrow \varepsilon_{C_5(d_3)}\Big|_{\mathsf{R}}^{\mathsf{v}} \rightarrow C_5(d_3) \Big\}, \end{aligned}$$

where condition θ makes the index scan `ISCAN` to return some extra random sample pages (blocks), assuming the tuples in the qualified pages are not sufficient to form an acceptable sample set.

To merge the query plan with $D(P'_5)$, Step 6(c) in Algorithm I augments Q_5 horizontally to piggybacking level L_4^2. The augmented query plan $D_4^2(Q_5)$ can be then merged with $D(P'_5)$ as follows:

$$\begin{aligned} & D_4^2(Q_5) + D(P'_5) \\ & \quad = \Big\{ R_3 \rightarrow \texttt{ISCAN}_{d_2>300 \vee \theta}\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \pi'_{d_1,d_2,d_3,d_4}\Big|_{\mathsf{E}}^{\mathsf{E}} \rightarrow \alpha; \\ & \qquad \alpha \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \varepsilon_{C_5(d_3)}\Big|_{\mathsf{E}}^{\mathsf{v}} \rightarrow C_5(d_3); \\ & \qquad \alpha \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \sigma_{d_2>300}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \\ & \qquad\quad \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \Sigma\Big|_{\mathsf{R}}^{\mathsf{E}} \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{b}} \\ & \qquad\quad \rightarrow \Sigma\Big|_{\mathsf{b}}^{\mathsf{t}} \rightarrow \pi'_{d_1,d_4}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{b}} \rightarrow \Gamma\Big|_{\mathsf{b}}^{\mathsf{E}} \\ & \qquad\quad \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \mu\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \mathit{resultSet} \Big\}. \end{aligned}$$

Note that this merge utilizes the vertical reduction transformation rule F2. Finally, Step 8 in Algorithm I simplifies the combined plan as follows:

$$\begin{aligned} & D_4^2(Q_5) + D(P_5) \\ & \quad = \Big\{ R_3 \rightarrow \texttt{ISCAN}_{d_2>300 \vee \theta}\Big|_{\mathsf{R}}^{\mathsf{E}} \\ & \qquad\quad \rightarrow \pi'_{d_1,d_2,d_3,d_4}\Big|_{\mathsf{E}}^{\mathsf{E}} \rightarrow \alpha; \\ & \qquad \alpha \rightarrow \Gamma\Big|_{\mathsf{E}}^{\mathsf{R}} \rightarrow \varepsilon_{C_5(d_3)}\Big|_{\mathsf{E}}^{\mathsf{v}} \rightarrow C_5(d_3); \\ & \qquad \alpha \rightarrow \Sigma\Big|_{\mathsf{E}}^{\mathsf{t}} \rightarrow \sigma_{d_2>300}\Big|_{\mathsf{t}}^{\mathsf{t}} \rightarrow \pi'_{d_1,d_4}\Big|_{\mathsf{t}}^{\mathsf{t}} \\ & \qquad\quad \rightarrow \Gamma\Big|_{\mathsf{t}}^{\mathsf{R}} \rightarrow \mu\Big|_{\mathsf{R}}^{\mathsf{R}} \rightarrow \mathit{resultSet} \Big\}. \end{aligned}$$

5. OTHER RELATED ISSUES

In this section, we discuss several related issues that may occur when the piggyback method is incorporated into a real system.

5.1. Initial statistics

The piggyback method collects statistics during query processing, providing a database server with a self-tuning capability. In other words, the system is expected to generate

increasingly better execution plans since more accurate statistics can be collected or estimated by the piggyback method during server operation. A shortcoming of this approach is that the system may initially produce poor execution plans. One possible solution is to use 'default values' for initial statistics, though this will almost certainly result in suboptimal execution plans for some classes of queries. Alternatively, the server may use heuristic query optimization techniques when insufficient statistics are present upon which to base cost-based optimization. Unfortunately, not all DBMSs support both heuristic and cost-based query optimization. Other approaches are to use the utility method to acquire the initial statistics, or implement piggybacking techniques into the system's database load utility to compute a subset of relevant statistics when the data is first loaded, a technique employed by IBM's DB2 and iAnywhere Solutions' SQL Anywhere.

5.2. Piggybacking frequency

There is no need to perform piggyback analysis for every user query, since statistics may already be up-to-date and also inexact statistics may not be detrimental to execution plan selection, though poor execution plans may be chosen if statistics are wildly inaccurate. Our experiments with a commercial database system demonstrated that, in extreme cases, accurate statistics can make the difference between seconds and days for a simple join query over moderately sized tables. Of course, queries that exhibit this severe skew are relatively rare. However, inaccurate statistics can lead to extreme performance degradation in actual practice. A question then arises: when is the proper time to perform piggyback analysis?

A straightforward technique is to perform piggyback statistical analysis periodically, e.g. every 100 queries. This method may not be effective since statistical change may not follow a fixed pattern. Another technique is to let the DBA manually enable/disable piggybacking when appropriate, though with this approach the self-tuning advantages of piggybacking now largely vanish.

A better technique is to let the system decide when piggybacking is advantageous, based on system load and query performance. When the performance of a query worsens, the server could activate piggybacking for relevant statistics to ensure that these are up-to-date. The server could choose an appropriate piggybacking level based on the current system load: the heavier the load, the lower the piggybacking level. If piggybacking level 1 is activated, the server can automatically invoke the utility method for the relevant data objects when the system load is light.

Alternatively, piggybacking level 1 can be set as the default level. When a significant number of statistics are found to be inaccurate, the server can raise the piggybacking level or invoke the utility method to gather the appropriate statistics. Once accurate statistics are computed the server can deactivate piggybacking until statistics are again out-of-date.

5.3. Inconsistent statistics

Invariably, at any given time some database statistics will be up-to-date, while others will not; this problem exists whether or not piggybacking is utilized. For systems that only support a utility method for statistics collection, the DBA must decide, sometimes subjectively, which data objects are candidates for re-analysis. With piggybacking, the opportunity exists for the system to objectively determine which data objects require re-analysis, and to perform this analysis automatically. However, there is still a chance that some statistics required to optimize a given query are not up-to-date. In this case, we could employ the following solution. During optimization, the server can analyze the consistency of those statistics relevant to a specific request. If an inconsistency is identified, either default statistics are used, or the utility method is invoked on the fly to collect up-to-date statistics for the relevant objects. For our piggybacking framework, an improvement can be made by recording the corresponding data objects for the lightweight piggybacking (i.e. piggybacking level 1) illustrated in Figure 2 if the system chooses to use the default statistics for optimizing a specific query when an inconsistency occurs.

To reduce the chance for inconsistent statistics to occur for a specific query, the technique suggested in [29] could be utilized to identify the essential set of statistics needed for optimizing the query. These statistics are then used to guide piggybacking statistics collection during query processing and/or enhance the lightweight piggybacking to update statistics off-line automatically. It is not necessary to maintain statistical consistency for all data objects in the entire database, as long as those statistics needed for optimizing frequently used queries are consistent. Another technique to reduce the chance for an inconsistency to occur is to apply cascading statistics updates during piggybacking statistics collection. With cascading statistics updates, if one statistic is updated, other related statistics may also be updated. For example, if statistic C_3 (Table 1) for a primary key column is updated, the corresponding statistic T_1 is updated as well. In case an inconsistency is identified and cannot be resolved during on-line piggybacking, the relevant data objects are recorded for the lightweight piggybacking to update statistics off-line automatically. On the other hand, as a practical note, statistics are not required to be very accurate and a certain degree of inconsistency is usually tolerated in real systems.

From the above discussion, it should be clear that the piggyback method cannot completely replace the utility method; rather, they complement each other. Piggybacking can reduce the invocation frequency of the statistics collecting utility and, moreover, such collection should occur only for frequently accessed data objects. On the other hand, the utility method can be used to collect statistics that cannot be collected via piggyback analysis or when it may be too expensive to do so. Using each method effectively can generally improve system efficiency and reduce the need for intervention by a DBA. When systems provide only a utility method, our experience tells us that DBAs are reluctant to

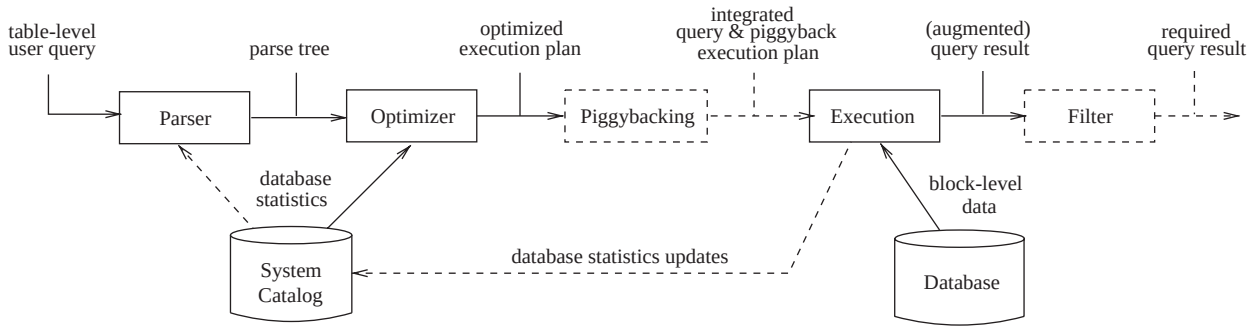

FIGURE 8. Architecture of query processing with piggybacking.

invoke it, due to its effect on system performance, unless absolutely necessary. The piggyback method provides a remedy to this problem.

5.4. Piggybacking for user-defined functions

Some types of statistics can be collected with piggybacking but not the utility method. Examples of such statistics are those for user-defined functions, e.g. the estimated number of read/write requests executed each time a UDF is executed and the estimated number of machine instructions executed when each function is executed. Since the utility method can only be used to gather statistics on the underlying database, it cannot be used to collect statistics about UDFs. However, such statistics are useful for query optimization. The way to collect them in existing commercial systems is to ask users to input them manually [30]. An obvious drawback for this approach is that users may not have sufficient knowledge about compilers and operating systems, and hence their estimates of such statistics may be far from being accurate. With piggybacking, the system can automatically collect such statistics while the relevant UDFs are executed, resulting in more accurate statistics.

6. PIGGYBACKING PROTOTYPE AND EXPERIMENTS

To provide a proof of concept and examine the overhead of piggybacking, we developed a prototype database management system using the piggyback method to collect statistics for query optimization. The architecture of the prototype system is shown in Figure 8. Each of the blocks represents a functional component and the paths between them represent an exchange of information in the indicated direction. The dotted portions of the figure show the main extensions to the architecture of a conventional DBMS. More specifically, when the parser reads schema information from the system catalog to check the semantics of a user query, statistics are also read for verification and possible update. The user query is parsed, and the parse tree is given to the query optimizer. An optimized query execution plan is then generated. If possible, the query execution plan is modified to integrate any relevant piggyback operations. The integrated execution plan is processed by the database execution engine.

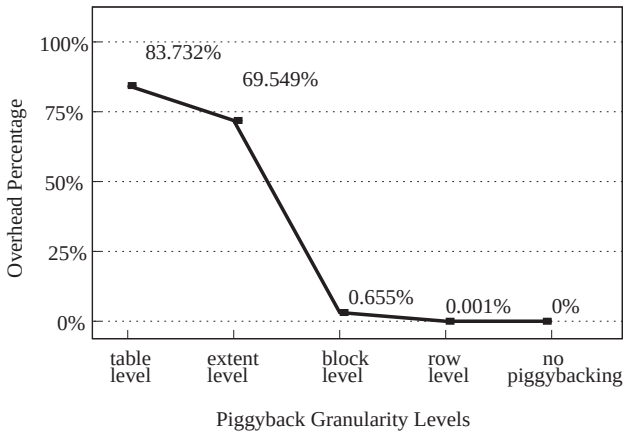

FIGURE 9. Overhead percentage by granularity level.

The database engine performs piggyback operations on the retrieved data during query processing, while the data is in main memory. If needed, the results from the database engine are filtered after statistics have been analyzed, so that the correct result for the original query is returned to the user. Finally, as warranted by the statistics collected and other factors such as system load, the system catalog will be modified to reflect the updated statistics. The prototype system was implemented with the C programming language in a Unix environment.

With the piggybacking prototype, we conducted extensive experiments and obtained promising results. Some typical experimental results are reported below. For our experiments, we randomly generated (uniformly distributed) data for nine tables with the number of columns ranging from 5 to 25 and the number of rows ranging from 10 to 10,000. User queries are expressed in SQL with predicates involving a comparison operation $\in \{=, <, >\}$. The experimental environment is a Pentium II 300 PC running Linux 2.2.12. Overhead for piggyback operations was measured in terms of the increase in query response time, which reflects both CPU and I/O costs.

Our experiments have demonstrated that the amount of overhead for a set of piggyback operations increases substantially as they are interleaved with a user query at coarser levels of data granularity (i.e. less tightly integrated with individual plan operators). Figure 9 shows that the

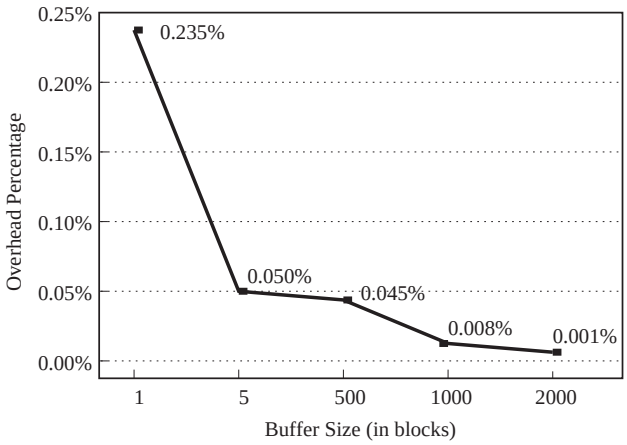

FIGURE 10. Overhead percentage by buffer size.

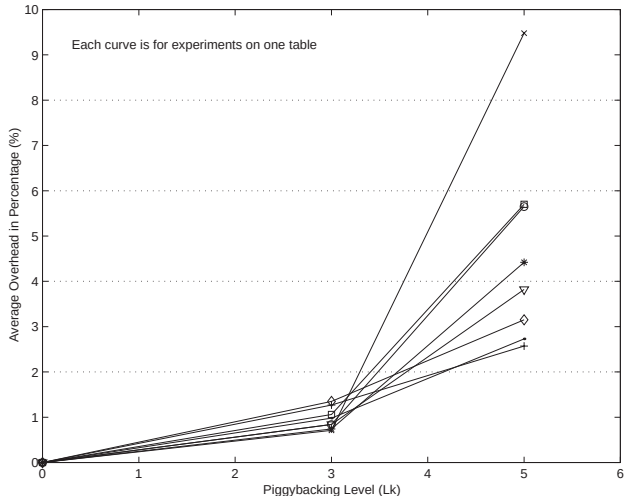

FIGURE 11. Piggybacking overhead in percentage (average) for SS at different piggybacking levels.

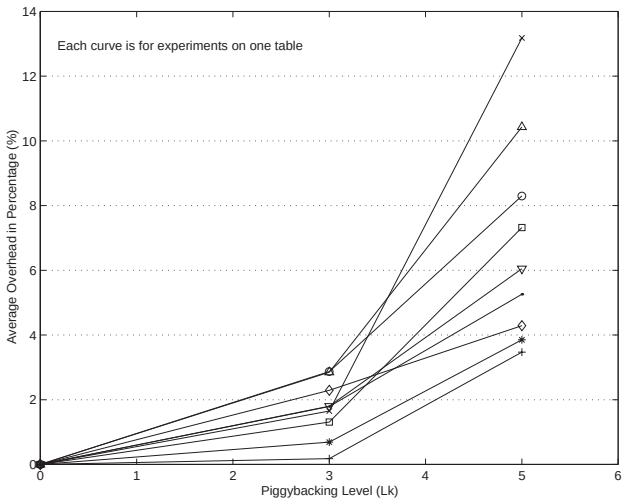

FIGURE 12. Piggybacking overhead in percentage (average) for IS at different piggybacking levels.

overhead of a set of piggyback operations (e.g. $\varphi_{C_1}(x)$ and $\varphi_{C_2}(x)$) is about the same (83%) as that of the user query when piggybacking is performed at the table level (i.e. piggybacking is performed after query processing is completed), while the piggybacking overhead becomes almost negligible (0.001%) when piggybacking is performed at the row level. Clearly, the piggybacking overhead dramatically decreases as the level of data granularity becomes finer and finer. Hence, piggybacking should be performed at the finest possible granularity level for the statistics to be gathered.

From our experiments, we observed that the percentage of overhead incurred for a given piggyback operation decreases as the size of the available data buffer increases. Figure 10 shows a typical effect of buffer size on piggybacking overhead for a piggyback operation.

We also verified that piggybacking overhead increases as the piggybacking level increases. In the experiments, we examined the percentage of piggybacking overhead at representative piggybacking levels L_0, L_3^1 and L_5^1 for both the sequential scan access method and the index scan access method. To avoid a biased observation from an individual execution of a query, we ran a set of queries using different buffer sizes for a given table, access method and piggybacking level. We then took the average of piggybacking overhead percentages observed for each table, access method and piggyback level. Figure 11 shows how the average percentage of piggybacking overhead changes with different piggybacking levels for the sequential scan access method on different tables; Figure 12 illustrates the piggybacking overhead for the index scan access method on different tables. From the figures, we can see that piggybacking overhead increases as the piggybacking level increases for queries using either SS or IS on all tables. In fact, the overall average overhead percentage for SS is 1.23% at piggybacking level L_3^1 (containing level L_2^1) and is 4.90% at piggybacking level L_5^1 (containing level L_4^1). The overall average overhead percentage for IS is 1.85% at piggybacking level L_3^1 and is 6.49% at piggybacking level L_5^1. Given the possible improvements to execution plan quality when the query optimizer can utilize up-to-date statistics, we feel this level of piggybacking overhead is acceptable: note that the most practical piggybacking levels L_2^1 and L_3^1 require only less than 3% overhead in our experiments.

As mentioned previously, our study is based on the widely accepted assumption that more accurate statistics usually lead to more efficient query processing. To verify this assumption, we conducted an experiment on a commercial DBMS running under SunOS 5.1 on SUN UltraSparc 2 workstation. In the experiment, a query to join four tables with randomly generated data and cardinalities ranging from 80,000 to 157,000 was executed. We considered different scenarios in which the statistics in the database catalog reflected varying degrees of accuracy (e.g. accurate statistics, statistics with 30% relative error, statistics with 60% relative error and so on). Figure 13 shows the experimental results, where the performance degradation is measured by the

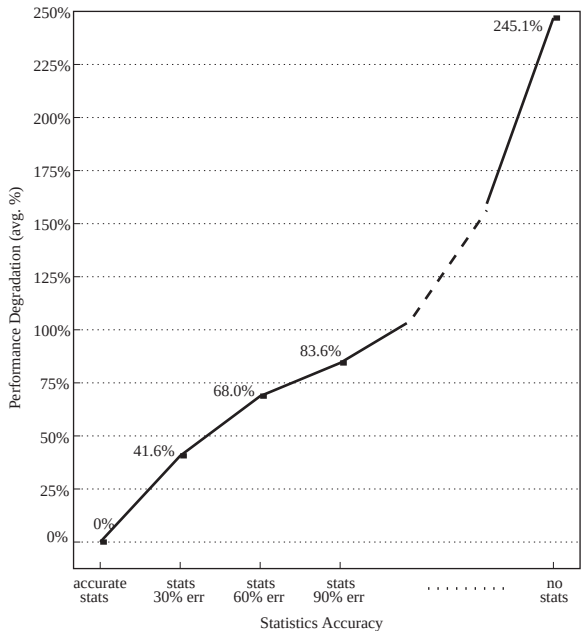

FIGURE 13. Impact of statistics accuracy on query performance.

percent increase of the (average) execution time at each statistics accuracy level with respect to the execution time with accurate statistics. The experiment demonstrates that query performance indeed degrades as accuracy of statistics decreases. Hence a technique such as piggybacking to gather/maintain up-to-date statistics is important in achieving efficient query processing.

In summary, our experiments have demonstrated that the piggyback method is quite promising in automatically maintaining statistics for query optimization in a DBMS.

7. CONCLUSION

Providing a query optimizer with up-to-date statistics on frequently changing data is challenging. While the utility method for statistics collection is widely used by commercial DBMSs, it has three significant disadvantages: it requires intervention on the part of the DBA, it suffers from often lengthy execution times, and it has a negative effect on system load. In this paper, we proposed a framework to incorporate on-the-fly statistics collection, which we term piggybacking, into query execution plans.

The key idea of the piggyback method in our framework is to piggyback some additional work such as side retrievals and statistical analysis on the processing of a user query. Although the additional piggyback work is not necessarily related to the processing of the current query and may incur a small additional overhead, the updated statistics can be utilized to optimize subsequent queries.

We have characterized several types of side piggyback retrievals. Vertical side piggyback retrievals (i.e. vertical piggybacking), which retrieve extra unrequested columns from an operand table, can increase the quantity of obtainable statistics; while horizontal side retrievals (i.e. horizontal piggybacking), which retrieve extra unrequested rows from an operand table, may improve the quality of obtained statistics. Typically, 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 quality statistics with low overhead.

We can perform piggyback analysis at different accuracy levels—precise measurement, estimation or validation—and at one of several piggybacking levels. Accuracy is usually determined by the statistic to be gathered and the execution plan chosen for the user query. The piggybacking level is determined according to a user-specified tolerance of piggybacking overhead. The higher the piggybacking level, the greater the piggybacking overhead, although additional statistics, with higher quality, could be obtained. A useful, lightweight piggyback technique 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 collecting utility is then automatically invoked, when the system load is low, for those frequently accessed data objects whose statistics are out-of-date.

We introduced a multiple-granularity interleaving algorithm to integrate efficiently a set of piggyback operations with a given user query at different levels of granularities. Integration techniques employed include merging shared work, augmenting user queries, downgrading piggyback operations and applying a set of heuristics and transformations.

We developed a prototype database system that incorporates piggybacking to verify the practicality of the piggyback method. Our experiments with this prototype demonstrated that the piggyback method offers a reasonable tradeoff between the advantages of up-to-date statistics versus the cost of collecting them. Useful statistics can usually be obtained via the piggyback method with a small additional overhead.

Note that the piggyback method does not eliminate the overhead of statistics collection altogether; rather, it merely takes advantage of data resident in main memory during query processing, and amortizes this overhead across multiple user queries. On the other hand, without piggybacking, the utility method has to read the relevant data from the disk into memory even though the data may have been in memory before. Therefore, the piggybacking approach can reduce the overall statistics collection overhead, compared with the utility method. Furthermore, applying the lightweight piggyback analysis mentioned previously, it is possible to perform the actual statistics collection during the light system load period when the system resources are not fully utilized.

The other advantages of the piggyback method are:

(i) *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 collecting utility. This advantage offers a great convenience to users.

(ii) *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.

(iii) *More statistical information is collected*, since extra unrequested data are considered and user-defined functions, which cannot be handled via the utility method, can also be handled via piggybacking.

(iv) *Up-to-date statistics are maintained*, since statistics are updated 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 terms of performance and convenience. However, our research is just the beginning of our ongoing work to achieve a truly 'self-maintaining' DBMS. Our future research work includes investigating parallel piggybacking, studying the effect of CPU capability on piggybacking efficiency, developing efficient strategies for managing CPU and memory resources for piggybacking, better managing and handling statistics inconsistency and exploring piggybacking via other non-query processing such as database reorganization.

ACKNOWLEDGEMENTS

The authors would like to thank Nandit Soparkar, Jacob Crossman, Jung-Uk Kim, Wahyudi Gunawan and Nicoles Markevis for their contributions to this project. The authors also wish to thank Peter Haas, John McPherson, Guy M. Lohman, Walid Rjaibi, Tony Lai, Calisto Zuzarte, Gabby Silberman, Per-Åke Larson and Alberto Mendelzon for their valuable suggestions and comments for the work reported in this paper. We sincerely thank the anonymous reviewers for their careful reading of the manuscript and constructive suggestions for improving the paper. The work of this paper was originally inspired by our earlier study on related issues in a multidatabase system [31], and the preliminary work of this paper was presented at two IBM CAS Conferences [32, 33]. Research was supported by the Centre for Advanced Studies at the IBM Toronto Laboratory and The University of Michigan.

REFERENCES

[1] Jarke, M. and Koch, J. (1984) Query optimization in database systems. *ACM Comput. Surv.*, **16**, 111–152.

[2] Yu, C. T. and Chang, C. C. (1984) Distributed query processing. *ACM Comput. Surv.*, **16**, 399–433.

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

[4] Zhu, Q. (1992) Query optimization in multidatabase systems. In *Proc. CASCON Conf.*, Toronto, Canada, November 9–12, Vol. II, pp. 111–127. IBM.

[5] Zhu, Q. and Larson, P.-Å. (1996) Global query processing and optimization in the CORDS multidatabase system. In *Proc. 9th ISCA Int. Conf. on Parallel and Distributed Computing Systems*, Dijon, France, September 25–27, Vol. II, pp. 640–646. ISCA.

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

[7] Kirkwood, J. (1993) *Sybase Architecture and Administration*. Ellis Horwood.

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

[9] Whalen, E. (1996) *Oracle Performance Tuning and Optimization*. SAMS Publishing.

[10] McNally, J. (1997) *Informix Unleashed*. SAMS Publishing.

[11] Haas, P. J., Naughton, J. F., Seshadri, S. and Stokes, L. (1995) Sampling-based estimation of the number of distinct values of an attribute. In *Proc. VLDB Conf.*, Zurich, Switzerland, September 11–15, pp. 311–322. Morgan Kaufmann.

[12] Shapiro, G. P. and Connel, C. (1984) Accurate estimation of the number of tuples satisfying a condition. In *Proc. ACM SIGMOD Conf.*, Boston, Massachusetts, June 18–21, pp. 256–276. ACM Press.

[13] Lipton, R. J., Naughton, J. F. and Schneider, D. A. (1990) Practical selectivity estimation through adaptive sampling. In *Proc. ACM SIGMOD Conf.*, Atlantic City, NJ, May 23–25, pp. 1–11. ACM Press.

[14] Olken, F. (1993) Random Sampling From Databases. Ph.D. Thesis, Department of Computer Science, University of California at Berkeley, Berkeley, CA.

[15] Yu, C. T., Lilien, L., Guh, K. C., Templeton, M., Brill, D., and Chen, A. (1986) Adaptive techniques for distributed query optimization. In *Proc. IEEE ICDE Conf.*, Los Angeles, CA, February 5–7, pp. 86–93. IEEE Computer Society.

[16] Antoshenkov, G. (1993) Dynamic query optimization in RDB/VMS. In *Proc. IEEE ICDE Conf.*, Vienna, Austria, April 19–23, pp. 538–547. IEEE Computer Society.

[17] Antoshenkov, G. (1996) Dynamic optimization of index scans restricted by booleans. In *Proc. IEEE ICDE Conf.*, New Orleans, LA, February 26–March 1, pp. 430–440. IEEE Computer Society.

[18] Yu, M. J. and Sheu, P. C.-Y. (1998) Adaptive query optimization in dynamic databases. *Int. J. Artificial Intell. Tools*, **7**, 1–30.

[19] Greenwald, M. (1996) Practical algorithms for self-scaling histograms or better than average data collection. *Perform. Eval.*, **20**, 19–40.

[20] Gibbons, P. B., Matias, Y. and Poosala, V. (1997) Fast incremental maintenance of approximate histograms. In *Proc. VLDB Conf.*, Athens, Greece, August 25–29, pp. 466–475. Morgan Kaufmann.

[21] Chaudhuri, S., Motwani, R. and Narasayya, V. (1998) Random sampling for histogram construction: how much is enough? In *Proc. ACM SIGMOD Conf.*, Seattle, WA, June 2–4, pp. 436–447. ACM Press.

[22] Aboulnaga, A. and Chaudhuri, S. (1999) Self-tuning histograms: building histograms without looking at data. In *Proc. ACM SIGMOD Conf.*, Philadelphia, PA, June 1–3, pp. 181–192. ACM Press.

[23] IBM (2001) *DB2 Universal Database Command Reference Version 7*. International Business Machines Corporation.

[24] iAnywhere Solutions (2001) *Adaptive Server Anywhere SQL User's Guide*. iAnywhere Solutions—Sybase Comp.

[25] Mannino, M. V., Chu, P. and Sager, T. (1988) Statistical profile estimation in database systems. *ACM Comput. Surv.*, **20**, 191–221.

[26] Vitter, J. S. (1992) Random sampling with a reservoir. *ACM Trans. Math. Software*, **11**, 98–107.

[27] Lynch, C. A. (1988) Selectivity estimation and query optimization in large databases with highly skewed distributions of column values. In *Proc. VLDB Conf.*, Los Angeles, CA, August 29–September 1, pp. 240–251. Morgan Kaufmann.

[28] Ailamaki, A., DeWitt, D. J., Hill, M. D. and Skounakis, M. (2001) Weaving relations for cache performance. In *Proc. VLDB Conf.*, Roma, Italy, September 11–14, pp. 169–180. Morgan Kaufmann.

[29] Chaudhuri, S. and Narasayya, V. (2000) Automating statistics management for query optimizers. In *Proc. IEEE ICDE Conf.*, San Diego, CA, February 28–March 3, pp. 339–348. IEEE Computer Society.

[30] IBM (2001) *DB2 Universal Database Administration Guide: Performance Version 7*. International Business Machines Corporation.

[31] Zhu, Q. (1993) An integrated method of estimating selectivities in a multidatabase system. In *Proc. CASCON Conf.*, Toronto, Canada, October 24–28, pp. 832–847. IBM.

[32] Dunkel, B., Zhu, Q., Lau, W. and Chen, S. (1999) Multiple-granularity interleaving for piggyback query processing. In *Proc. CASCON Conf.*, Toronto, Canada, November 8–11, pp. 24–39. IBM.

[33] Zhu, Q., Dunkel, B., Soparkar, N., Chen, S., Schiefer, B. and Lai, T. (1998) A piggyback method to collect statistics for query optimization in database management systems. In *Proc. CASCON Conf.*, Toronto, Canada, November 30–December 3, pp. 67–82. IBM.