Proceedings of CASCON'1999, Canada, 1999

Multiple-Granularity Interleaving for Piggyback Query Processing *

Brian Dunkel[†] Qiang Zhu[‡] Wing Lau[‡] Suyun Chen[§]

Abstract

Piggyback query processing is a new technique, described in [24], intended to perform additional useful computation (e.g., database statistics collection) during normal query processing, taking full advantage of data resident in main memory. Different types of beneficial piggybacking have been identified and studied, but how to efficiently integrate piggyback operations with a given user query is still an open issue. In this paper, we propose a technique of multiple-granularity interleaving to efficiently integrate multiple piggyback operations with a given query at different levels of data granularity. We introduce an algebraic notation to capture the main characteristics of data flows in a database management system (DBMS), facilitating the study of piggybacking and enabling the automated integration of piggyback operations and user queries in a DBMS supporting the piggyback method. Various integration techniques are introduced to facilitate multiple-granularity interleaving including merging shared work, augmenting user queries, and downgrading piggyback operations. A set of transformations and heuristics are suggested that preserve the semantics of a user query, while efficiently interleaving the operations. Our preliminary experiments indicate that interleaving at proper levels of data granularity is key to the efficient implementation of the piggyback method.

Keywords: Query processing, query optimization, piggybacking, multiple-granularity interleaving, database statistics

*Research was supported by the Centre for Advanced Studies at IBM Toronto Lab and The Univ. of Michigan.

[†]Dept of Electrical Engineering and Computer Sci., The Univ. of Michigan, Ann Arbor, MI 48109, USA.

[‡]Dept of Computer and Information Science, The Univ. of Michigan, Dearborn, MI 48128, USA.

[§]IBM Toronto Lab, North York, Ontario, Canada.

1 Introduction

In our previous work [24], we described a methodology for the collection and maintenance of database statistics during the normal processing of user queries. The purpose of this so-called *piggyback method* is to improve both the quantity and quality of data statistics for use in query optimization, while at the same time relieving the burden of the database administrator to maintain these statistics. Our study has demonstrated that this method is promising in maintaining statistics for query optimization, compared with the current *utility method* employed in most commercial database management systems (DBMS) including DB2, Oracle, Sybase and Informix [5, 11, 15, 20]. Different types of useful piggybacking were studied in our previous work, including vertical, horizontal, mixed, and multi-query piggybacking. However, an important issue left unresolved was how to efficiently perform the piggyback operations in conjunction with the normal query processing in an existing or newly developed DBMS. We address this issue with the current work.

In order to properly express and implement an automated integration of query and piggyback operations, we must recognize that database statistics are collected at a number of levels of data granularity. For example, the size of a table is important both in the number of rows and the number of physical blocks. Since operations in relational algebra are expressed only at level of the table, we need to extend the relational algebra in a way that allows us to handle the different levels of data granularity.

The *multiple-granularity interleaving* approach we propose in this paper allows a DBMS implementing the piggyback method to identify what degree of interleaving or even true parallel processing is possible and appropriate for a given query and a fixed set of piggyback operations. As operations are decomposed into sub-operations at lower and lower levels of data granularity, the possible choices for interleaving and parallelism are increased. The purpose of our work is to develop a technique that will enable the automated integration of query and piggyback processing at appropriate levels of data granularity in an efficient way for their interleaved or parallel execution. We use an algebraic notation, which also has a corresponding graphical representation, for data flows in a query execution plan to express these various data granularities explicitly so that piggyback operations can be integrated with user queries at appropriate levels by a simple algorithm.

In addition to enabling the effective and efficient interleaving of query and piggyback operations in a newly constructed DBMS, the multiple-granularity interleaving approach can also be used to identify which types of piggybacking are appropriate for implementation in an existing DBMS. Clearly the piggyback method can be more easily incorporated into a newly constructed DBMS than into an existing one since the latter may have some restrictions on where the piggyback processing may take place. For example, in an existing DBMS, it may be infeasible for piggyback processing to be implemented directly inside the query processing engine at the row level, perhaps for reasons of security or programming complexity. In this case, only manipulations at the table or statement level will be appropriate. Our technique of multiple-granularity interleaving allows a system to integrate piggyback operations with a query at the most effective levels of data granularity that are accessible.

The integration of multiple piggyback operations with a single user query operation is similar in some ways to the problem of multiple query optimization studied in the literature [2, 4, 18], for example, the techniques related to common subexpressions [3, 7]. However, there are some significant differences between multiple query optimization and our task here. First of all, in multiple query optimization the problem is to combine a set of user query operations that are of equal importance, while in our problem piggyback operations are of secondary importance relative to the user query operation, and in the extreme case, a piggyback operation can even be ignored. This observation provides new opportunities for optimizing piggybacking integration. Secondly, in multiple query optimization the set of query operations can be arbitrary, whereas in piggybacking integration only the user query is arbitrary, and the set of possible piggyback operations is fixed for a given database schema. This observation opens possibilities to adopt special (rather than generic) efficient techniques specifically targeted toward piggyback operations. Finally, multiple query optimization does not usually consider the data explicitly at multiple levels of granularity, as we must do here.

The area of statistics collection and maintenance for query optimization has been studied from a number of perspectives. Mannino *et al.* provides in [14] a comprehensive survey of existing techniques for statistics collection and cost estimation in DBMSs. A number of techniques have been developed as well for the estimation of database statistics, including techniques to estimate physical statistics (e.g., page references) by Copeland, Mackert, and Zander *et al.* in [8, 13, 23]. Christodoulakis, Lipton, Selinger, and Shapiro *et al.* have proposed in [6, 12, 17, 19] various techniques including parametric methods, table-based methods, and sampling methods to estimate the intermediate and target table sizes from some base statistics, and in [9], Haas *et al.* introduced several sampling-based estimators to estimate the number of distinct values of a column in a table. Yu and Lilien *et al.* suggested a number of dynamic (adaptive) query optimization techniques in [21], and classified them into direct ones, which dynamically optimize the current query based on runtime information, and indirect ones, which collect dynamic information from the current query to optimize subsequent queries. An adaptive query optimization algorithm was also proposed by Yu and Sheu to dynamically complete a partial access plan based on latest statistics collected at runtime in [22].

Some of the issues of data granularity that we study here have also been examined, especially in areas related to parallel processing. For example, in [10] Jamieson shows that the granularity of data access impacts the performance of parallel algorithms, while in [1] Ahmed uses transformations of data granularity to implement efficient execution on parallel hardware. More directly related to the management of data, in [16] Pernul *et al.* describe how data granularity affects the design of multilevel secure databases.

The rest of this paper is organized as follows. Section 2 reviews the statistics obtainable via piggybacking and describes a high-level view of query processing in a DBMS with and without piggybacking. Section 3 defines more carefully the necessary elements to describe multiple-granularity interleaving, and Section 4 demonstrates how these can be used to represent multiple-granularity interleaving in a DBMS, along with some experimental results that verify our technique. Finally, Section 5 presents our conclusions and some areas for future work.

2 Background

We review here some of the key points from [24] related to database statistics and examine the context in which piggyback query processing takes place in a DBMS.

2.1 Query Optimization Statistics

Different DBMSs may maintain different types of statistics on databases in their system catalogs for query optimization. Table 1 shows some typical statistics maintained in a DBMS. Moreover, the statistics for query optimization can be classified into logical and physical types, where the logical statistics can be determined by the data values in a database, and physical ones are determined by the properties of the physical organization of the database on a storage medium. Using the labels of Table 1, C_1, C_2, C_3, C_4, T_1, I_3 and I_4 are logical statistics, and T_2, I_1 and I_2 are physical statistics. An implementation of the piggyback method should be able to collect, estimate or validate both logical and physical statistics.

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 to access the results of sub-queries. Hence, we mainly consider unary access methods, and the common ones are:

Sequential scan (SS)	– *scan a table sequentially.*
Index scan (IS)	– *get qualified rows via an index.*
Index-only (IO)	– *get values from an index tree.*
Hash access (HA)	– *get rows via a hash table.*

Certain statistics can be obtained during the execution of each access method, although at differing levels of accuracy. At the top level, an accurate statistic is calculated using a complete data set. At the second level, a given statistic is estimated via sampled data. At the next level, validity information (i.e., whether or not the statistic is up-to-date) may be determined. At the lowest level, no information about a particular statistic can be found at all. Table 2 shows at what level some statistics may be obtained during the execution of different access methods.

For the sequential scan method, since the whole data file of a table is scanned, all column and table statistics can be accurately calculated during its execution. However, since indexes are not accessed, the physical index statistics can at best be estimated.

For the index scan method, there are several cases. The first case occurs when an index tree is used as a means to scan the whole corresponding table in the sorted order of the indexed column. Since both the index and the table are fully scanned, all statistics can be obtained. The second case occurs when the retrieved values of an indexed column a from the index tree cover (at least) both the range $a < \beta_1$ and the range $a > \beta_2$, where β_1 and β_2 are constants. Since $a < \beta_1$ (if not empty) implies $min(a)$ is retrieved and $a > \beta_2$ (if not empty) implies $max(a)$ is retrieved, both the maximum and minimum statistics (C_1 and C_2) can be obtained accurately. Since the index tree is accessed, statistic I_2 can also be obtained accurately. Other statistics can be estimated by using the set of retrieved data as a set of sample data. For example, statistic C_3 for column a in table R can be estimated as $C_3 = |R| * n(a)/|S|$, where $|R|$, $n(a)$, and $|S|$ are the cardinality of R, the number of distinct values in the sample set S, and the cardinality of the sample set, respectively. A sample set can be improved by applying horizontal piggybacking or multi-query piggybacking, which are described in greater de-

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)
Table Statistics	T_1	number of rows in a table
	T_2	number of pages used by a table
Index Statistics	I_1	number of leaf pages
	I_2	number of B-tree index levels
	I_3	number of distinct values for the 1st column of index key
	I_4	number of distinct values for the full index key

Table 1: Typical Statistics Maintained in a System Catalog

			C_1	C_2	C_3	C_4	T_1	T_2	I_1	I_2	I_3	I_4
SS			$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\oplus$	$\surd$	$\surd$
IS	(I)	full	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
	(II)	$a < \beta_1,\ a > \beta_2$	$\surd$	$\surd$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\surd$	$\oplus$	$\oplus$
IO	(I)	full	$\surd$	$\surd$	$\surd$	$\surd$	$\times$	$\surd$	$\surd$	$\surd$	$\surd$	$\surd$
	(II)	$a < \beta_1,\ a > \beta_2$	$\surd$	$\surd$	$\oplus$	$\oplus$	$\times$	$\oplus$	$\oplus$	$\surd$	$\oplus$	$\oplus$
HA			$\odot$	$\odot$	$\oplus$	$\oplus$	$\oplus$	$\oplus$	$\times$	$\times$	$\times$	$\times$

'$\surd$' — accurate statistics; '$\oplus$' — estimated statistics via sampling;
'$\odot$' — validity information of statistics; '$\times$' — no obtainable/applicable statistics;
'a' — an indexed column; 'β_1', 'β_2' — constants, where $\beta_1 < \beta_2$;

Table 2: Statistics Obtainable via Access Methods

tail in [24], along with further discussion of the index-based access methods. Note that when the retrieved values of an indexed column cover the range $a < \beta_1$ but not the range $a > \beta_2$, statistic C_2 can be obtained accurately, but not statistic C_1. For C_1, the following condition can be used to check its validity: ($\exists x \in S$ s.t. $x > C_1$) $\Rightarrow$ (C_1 is out-of-date).

For the index-only access method, the obtainability of statistics is similar to the index scan method. The only difference is that table statistic T_1 cannot be obtained since the data file is never actually accessed. Note that T_1 may be obtained or estimated when the referenced structure is a unique index.

For the hash access method, we have identified conditions (see [24] for more details) that can be used to validate statistics C_1 and C_2. 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.

2.2 Query Processing Architecture

Figure 1 shows a high level block diagram of the query processing performed in a typical DBMS. 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 Figure 1 show extensions to the architecture for a DBMS implementing the piggyback method. In this case, 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 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. As blocks of data are retrieved from the database, piggyback operations are performed on the data while in main memory. If needed, the results from the database engine are filtered after statistics have been analyzed, so that the correct results for the original query are 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.

If we restrict ourselves to the highest level of data granularity, there is very little integration between the piggyback processing and the normal processing of the user query. In a sense, the various components of the query processing system are used as "black boxes". For this reason, we refer to this level of piggybacking as *external* piggybacking. However, if we integrate more fully the query processing with the piggyback processing (*e.g.*, when building a new DBMS "from the ground up"), we can take advantage of the information associated with data at finer

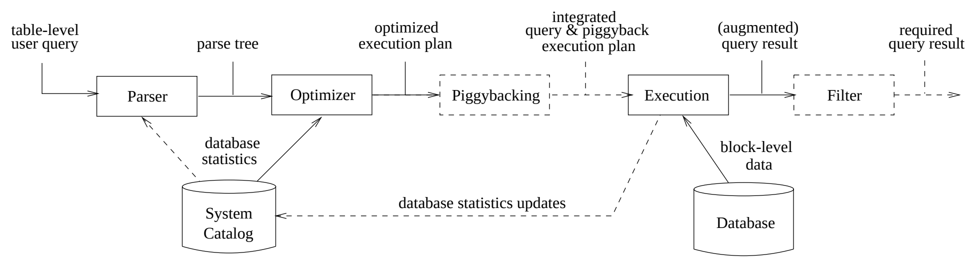

Figure 1: High-level representation of typical query processing

levels of data granularity. For example, external piggybacking cannot determine directly the number of blocks in a base table or the height of an index tree. For this reason, we develop the notion of multiple-granularity interleaving to combine operations at the most effective accessible level.

3 Multiple-Granularity Interleaving

In this section, we describe some elements of multiple-granularity interleaving, including user query and piggyback operations, data flow plans, data semantics of operations, and data granularities.

3.1 User Query and Piggyback Operations

For a given DBMS in which the piggyback method is to be implemented, there will be a defined set of query processing primitives and a defined set of statistics collection operations to be piggybacked on user queries. Tables 3 and 4 list some example logical and physical query processing operations and Table 5 lists the classes of piggyback (statistics collection) operations. In each table, we give a symbol to represent every operation for the remainder of this paper.

A few notes about the operations of Tables 3, 4 and 5. First, all query and piggyback operations have certain logical and physical properties in their processing, so our designation of σ as a logical operation and Σ as a physical operation, for example, may be arguable. However, the operations of Table 3 are most relevant when query processing is viewed from a design perspective, whereas those of Table 4 are most relevant during physical implementation. Also, we recognize that there is a difference between the semantics of SQL and 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.

As a concrete example of how we use the notation to represent these operations, consider the simple query `SELECT DISTINCT A1,A4 FROM R1 WHERE A3>200`, where the operand table `R1` has four columns `A1`, `A2`, `A3`, and `A4`. This can be written as $Q_1 = \pi_{A1,A4}(\sigma_{A3>200}(\texttt{R1}))$. If we would like to compute the maximum value of column `A3` in `R1`, then our piggyback operation can be represented as $P_1 = \varphi_{C_1(A3)}(\texttt{R1})$.[1] The goal of the piggyback method in this example is to effect an efficient interleaving of the two operations Q_1 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.

3.2 Data Flow Plans

A relational algebra expression for a query operation is often represented as a query tree. Such a tree diagram implies a certain flow of data, with the details being left to the implementa-

[1]We use the identifiers of Table 1 to represent the various statistics collected throughout the examples in this paper.

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)

Table 3: Logical 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)
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	δ, ι, κ, λ, χ	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 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

Table 4: Example physical query processing operations

tion and physical representation, which we refer to as a high-level *data flow plan* for the relevant query operation. 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 performed with a data flow plan at various levels. We will use a similar data flow plan to represent piggyback operations. For any operation $\otimes$, we use $D(\otimes)$ to denote its associated data flow plan being considered. 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 3.4.

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 2 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. We do not concern ourselves with distinctions between data-driven and demand-

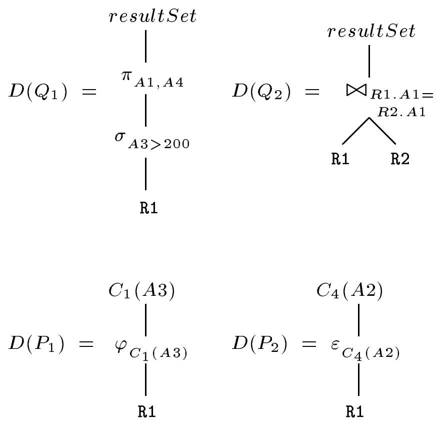

Figure 2: Example data flow plans

driven models for data flows, since the results are equivalent for our purpose.

Although the graphical representation of Figure 2 is useful for presenting data flow relationships, space constraints lead us to use an equivalent algebraic representation that is more compact. In this notation, $A \to \otimes \to B$ 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

Operation	Symbol	Description
find statistic	φ_{stat}	Find the database statistic *stat* for a given set of data or index elements, where *stat* is one of the statistics listed in Table 2.
estimate statistic	ε_{stat}	Estimate the database statistic *stat* for a given set of data or index elements, where *stat* is one of the statistics listed in Table 2.
verify statistic	v_{stat}	Verify the database statistic *stat* for a given set of data or index elements, where *stat* is one of the statistics listed in Table 2.

Table 5: Classes of statistics collection operations for piggybacking

connections between the various nodes in a data flow plan. For example, $D(Q_2)$ (shown graphically in Figure 2) would be represented by $\{$ `R1` $\rightarrow \alpha$; `R2` $\rightarrow \alpha$; $\alpha \rightarrow \bowtie \rightarrow$ *resultSet* $\}$, where the additional symbol α simply represents a common point in the data flow plan.

3.3 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 all valid data inputs 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 $Y \equiv_S X$).

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) there is a one-to-one mapping between the row identifiers of R and R' and (2) $\mathcal{S}(R') \subseteq \mathcal{S}(R)$. 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 one-to-one mapping between the elements of R and $R' \cup (R - R')$. 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 set X such that $X \rightarrow \otimes \rightarrow Y$ and $Y \subset_V X$. 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 $Y \subset_H X$. Operations which *preserve horizontal semantics* perform no strict horizontal reduction on any input. As an example, the physical project operation (without duplicate elimination) preserves horizontal semantics, and the select operation preserves vertical semantics.

Table 6 shows the semantics preserving properties of some relational query and piggyback operations. The scan (Σ) and gather (Γ) operations will be described in greater detail in Section 3.4. 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 of the piggyback statistics collection operations, represented by *stat* in the table.

3.4 Data Granularities

In order to interleave operations at multiple levels of data granularity in a single data flow plan, we introduce a notation for describing explicitly the granularity at which a given operation is processed. The 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 7 lists the logical and physical 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 Figure 7

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

Table 6: Semantics-preserving properties

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

Table 7: Logical and physical data granularities

have a sequential containment relationship to one another. In other words, a collection of rows (t) makes up a block (b), a block extent (E) is composed of multiple blocks, and a table (R) is composed of multiple extents. Note that rows and tables are logical constructs, while blocks and extents are physical. The transformation from one level in this sequence to another is represented in a data flow plan by Σ operations (to transform a composite granularity into its individual components) and Γ operations (to transform a collection of elements into a coarser granularity).

As an example of how these granularities represent operations in practice, consider the relational σ operation. Diagram (a) in Figure 3 shows explicitly that both the input and output objects are tables (as indicated by R). However, in order for the selection and any piggyback operations to be interleaved row by row, we must define an equivalence between the σ operation at the row level and the table level.

Diagram (b) of Figure 3 shows the explicit row-level (t) selection operation in a way that is logically equivalent to the table-level σ operation. So, diagram (b) of Figure 3 also illustrates the fact that a table can be converted into a set of rows ($\boxed{\Sigma}\,^{\mathsf{t}}_{\mathsf{R}}$) and that the output rows can then be gathered into a single table ($\boxed{\Gamma}\,^{\mathsf{R}}_{\mathsf{t}}$) . Note that the equivalence shown in Figure 3 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 3.2, the graphical representation of Figure 3 is illustrative, but space constraints lead us to use the equivalent algebraic

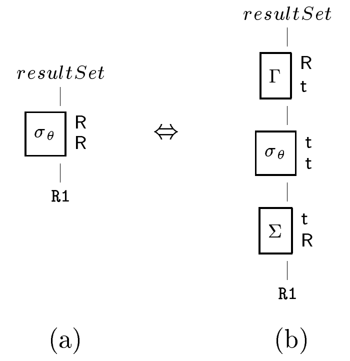

Figure 3: Operations of explicit data granularities

representation, augmented with explicit indicators of data granularity. In this notation, $A \rightarrow \otimes\Big|^{g_2}_{g_1} \rightarrow \oplus\Big|^{g_3}_{g_2} \rightarrow B$ 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 necessarily match the input granularity of $\oplus$. So, in this notation, diagram (a) of Figure 3 would be represented as $\mathtt{R1} \rightarrow \sigma_\theta\Big|^{\mathsf{R}}_{\mathsf{R}} \rightarrow resultSet$.

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 3.2, 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 $\Big\{ A \rightarrow \alpha;\ B \rightarrow \alpha;\ \alpha \rightarrow \bowtie\Big|^{\mathsf{R}}_{\mathsf{R,\,R}} \rightarrow C \Big\}$.

4 Multiple-Granularity Query Processing

The basic idea of the piggyback method is to modify the processing of a normal user query in an efficient way so that statistical information can be collected about the relevant data. How to interleave the sub-operations of a user query with a fixed set of piggyback operations is the issue to be discussed in this section.

4.1 Multiple-Granularity Interleaving Algorithm

Our algorithm for multiple-granularity interleaving provides an integration of a given user query with a fixed set of piggyback operations so that all the operations can be performed efficiently. In fact, the integration is done for the execution plans (i.e., the data flow plans at low levels) of the operations. 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 [2] from the underlying table for the argument to the output. We can view all the feasible data flow plans (paths) for a piggyback 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 (i.e., determining piggyback operations) and which data flow paths for each piggyback operation can be instantiated for the current query (i.e., determining the non-deterministic data flow plan). 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 typically does 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:

- 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 implemented by the system, using transformation rules **F4** and **F5** described in Section 4.3.
2. Determine the set, O, of data objects (*i.e.*, tables, indexes and their related columns) that are referenced in $D(Q)$.
3. Based on the objects in O, instantiate the set of relevant piggyback operations $P = \{P_1, \ldots, P_n\}$.
4. 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 $D(Q)$ 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 4.2.
   (b) If no merging is possible for any instance of o_j in $D(Q)$, augment Q properly along the paths from an instance of o_j to the root in $D(Q)$, according to the overhead tolerance specified by the user, and attempt to merge paths in $\widetilde{D}(P_i)$ with the augmented plan for 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 augmented plan is possible, downgrade P_i to P_i', and attempt to merge the paths in $\widetilde{D}(P_i')$ with a path from an instance of o_j in

[2] 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.

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

5. Collapse the data granularity of the integrated data flow plan, where necessary, using transformation rules **F4**, **F5** and **F6** in Section 4.3.

6. 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.

7. Return integrated[3] data flow plan $D(Q) + D(P_1) + \ldots + D(P_m)$.

It should be noted that the manner in which these plans can be combined is generally not unique. However, this is not to say that the complexity of this algorithm grows unchecked. To demonstrate this, we outline briefly the complexity of step 4 in our algorithm, where the dominating portion of the work is done. Let N be the total number of occurrences of all tables in a query (which we will take to be the variable, in this case). Also, let I be the maximum number of indexes and C the maximum number of columns associated with any table in the database. Theoretically, I could be as large as 2^C (which is a constant, in any case), but in practice the overhead for values of $I \geq \frac{C}{2}$ tend to be prohibitive and thus avoided. If we use s to represent the maximum number of of statistics to be piggybacked on any object (usually s is on the order of 10), then the maximum number of piggyback operations considered by our algorithm will be $s * N + s * C * N + s * I * N$, i.e., $O(N)$. For each piggyback operation P_i, we may need to attempt to merge every possible path in $\widetilde{D}(P_i)$ with each of the N paths in the query data flow plan from a table recurrence to the root. Since no more than N paths will be possible to instantiate in $\widetilde{D}(P_i)$ for a given query Q, the merging complexity for one piggyback operation will be at worst $O(N^2)$ and the complexity of the entire algorithm (for $O(N)$ piggyback operations) is at worst $O(N^3)$. Note that, since we use heuristics to integrate the data flow plan for a given query with the data flow plans for a limited set of compatible piggyback operations, the complexity of our algorithm is not exponential.

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

In the following subsections, we elaborate some details of the algorithm. In particular, we describe the three basic strategies for integrating two data flow plans in Section 4.2. In Section 4.3, 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 4.4, 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. Finally, in Section 4.5, we discuss some of our experimental results using a research prototype for piggybacking.

4.2 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:

- Merge — If an initial sequence of sub-operations from two data flow plans perform 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 is shown graphically in Figure 4, using the data flow plans for Q_1 and P_1 from Figure 2. Note that transformation rules **F1** $\sim$ **F3** allow merging to be also done for two data flow plans with convertible initial sequences (not necessarily to be identical).

- Augment — This technique applies only to user queries. If a user query retrieves a given amount of data and a particular

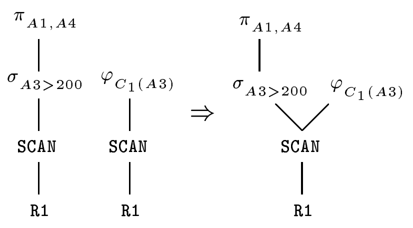

Figure 4: Merged data flow plan for Q_1 and P_1

piggyback operation would only be possible with a larger set of data, the original query plan can be augmented (see the concept of "horizontal piggybacking" in [24]). As an example, an added node in the plan could allow more rows to be read into memory (*e.g.*, by relaxing the selection criterion for an encapsulated operation implementing both scan and select from which only qualified rows are accessible), and then another node would filter out the original rows required by the query. This is demonstrated in Figure 5, where the presumption is that the larger set of data introduced by the additional condition θ would, for example, provide a better estimate of some statistics for columns other than A3. Similarly, the idea of "vertical piggybacking" in [24] can also be applied here.

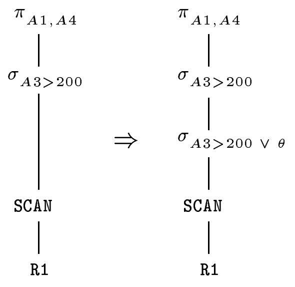

Figure 5: Augmented data flow plan for Q_1

- Downgrade — This technique applies only to piggyback operations. When the available data for a query is not sufficient to perform a more exact 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 is 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 6 could have been the original piggyback operation requested, but the plan on the right is the result of downgrading.

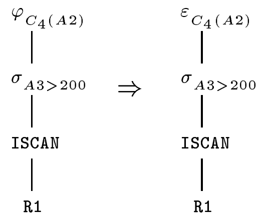

Figure 6: Downgraded data flow plan for P_1

Note that, when comparing multiple possible 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 define 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 share the common work, 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 from 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 scan of the first row in the sorted result (the second sub-operation, representing 1% of the total effort). An alternate 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-operations from the two data flow paths for the piggyback operation can be shared with the paths from a query data flow plan, the merging for the first data flow path is preferred since 99% of the work is shared, compared with 80% shared work for the second merging.

4.3 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, given in the following list. We use $\mathcal{S}_{in}(\otimes)$ to indicate the input schema required by an operation $\otimes$ with a set of rows as input and $\mathcal{S}_{out}(\otimes)$ to indicate the output schema produced.

F1: For $X \rightarrow \otimes \rightarrow Y$, if $Y \equiv_S X$, then $\{X \rightarrow \oplus\} \Rightarrow \{Y \rightarrow \oplus\}$

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 $X \rightarrow \otimes \rightarrow Y$, if $Y \subseteq_V X$ and $\mathcal{S}_{in}(\oplus) \subseteq \mathcal{S}_{out}(\otimes)$, then $\{X \rightarrow \oplus\} \Rightarrow \{Y \rightarrow \oplus\}$

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 $\mathcal{S}(X) = \{ \texttt{A1}, \texttt{A2}, \texttt{A3} \}$, and $\otimes = \pi_{A2,A3}$, then Y can only be used as an input to operations $\oplus$ for which $\mathcal{S}_{in}(\oplus) \subseteq \{ \texttt{A2}, \texttt{A3} \}$.

F3: For $X \rightarrow \otimes \rightarrow Y$, if $Y \subseteq_H X$ and Y is an acceptable subset[4] of X for operation $\oplus$, then $\{X \rightarrow \oplus\} \Rightarrow \{Y \rightarrow \oplus\}$

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: $\{X \rightarrow \otimes\Big|_{g_1}^{g_3} \rightarrow Y\} \Leftrightarrow \{X \rightarrow \Sigma\Big|_{g_1}^{g_2} \rightarrow \otimes\Big|_{g_2}^{g_3} \rightarrow Y\}$

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.

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

F5: $\{X \rightarrow \otimes\Big|_{g_1}^{g_3} \rightarrow Y\} \Leftrightarrow \{X \rightarrow \otimes\Big|_{g_1}^{g_2} \rightarrow \Gamma\Big|_{g_2}^{g_3} \rightarrow Y\}$

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: $\{ \oplus\Big|_{g_1}^{g_2} \rightarrow \otimes\Big|_{g_2}^{g_3} \rightarrow \otimes^{-1}\Big|_{g_3}^{g_2} \rightarrow \ominus\Big|_{g_2}^{g_4} \} \Rightarrow \{ \oplus\Big|_{g_1}^{g_2} \rightarrow \ominus\Big|_{g_2}^{g_4} \}$

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.

We have also identified a number of heuristics intended to improve the efficiency of the data flow plan integration techniques described in Section 4.2. These are given in the following list.

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 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, making the associated overhead be shared.

H2: Merge as much as possible.

When two plans can be merged, it is most efficient to merge as many of 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.

4.4 Examples of Interleaving

Using our notation for explicit representation of data granularities, we have developed data flow plans for a number of representative query operations and piggyback statistics collection operations to show how they can be combined. The plans for these operations can be interleaved according to the transformation rules and heuristics described in Section 4.3. Once the 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.

<u>**Example 1:**</u> Let us again consider the simple query Q_1 and piggyback operation P_1 of Figure 2. Using the notation introduced in Section 3.2 plans for these operations can be represented as

$$\begin{array}{ll} D(Q_1) &= \left\{ \texttt{R1} \to \sigma_{A3>200}\Big|^{\mathsf{R}}_{\mathsf{R}} \to \pi_{A1,A4}\Big|^{\mathsf{R}}_{\mathsf{R}} \to \mathit{resultSet} \right\}, \\ D(P_1) &= \left\{ \texttt{R1} \to \varphi_{C_1(A3)}\Big|^{\mathsf{v}}_{\mathsf{R}} \to C_1(A3) \right\}. \end{array}$$

If we expand both $D(Q_1)$ and $D(P_1)$ to a finer level of data granularity using **F4** and **F5**, we get

$$\begin{array}{l} D(Q_1) = \Big\{ \texttt{R1} \to \Sigma\Big|^{\mathsf{E}}_{\mathsf{R}} \to \Sigma\Big|^{\mathsf{b}}_{\mathsf{E}} \to \Sigma\Big|^{\mathsf{t}}_{\mathsf{b}} \to \sigma_{A3>200}\Big|^{\mathsf{t}}_{\mathsf{t}} \\ \quad \to \Gamma\Big|^{\mathsf{b}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{E}}_{\mathsf{b}} \to \Gamma\Big|^{\mathsf{R}}_{\mathsf{E}} \to \Sigma\Big|^{\mathsf{E}}_{\mathsf{R}} \to \Sigma\Big|^{\mathsf{b}}_{\mathsf{E}} \to \Sigma\Big|^{\mathsf{t}}_{\mathsf{b}} \\ \quad \to \pi_{A1,A4}\Big|^{\mathsf{t}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{b}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{E}}_{\mathsf{b}} \to \Gamma\Big|^{\mathsf{R}}_{\mathsf{E}} \to \mathit{resultSet} \Big\}, \end{array}$$

$$\begin{array}{l} D(P_1) = \Big\{ \texttt{R1} \to \Sigma\Big|^{\mathsf{E}}_{\mathsf{R}} \to \Sigma\Big|^{\mathsf{b}}_{\mathsf{E}} \to \Sigma\Big|^{\mathsf{t}}_{\mathsf{b}} \to \varphi_{C_1(A3)}\Big|^{\mathsf{v}}_{\mathsf{t}} \\ \quad \to C_1(A3) \Big\}. \end{array}$$

If we then merge the two plans, beginning with `R1`, we get a combined data flow plan of

$$\begin{array}{lll} D(Q_1) + D(P_1) &=& \Big\{ \texttt{R1} \to \Sigma\Big|^{\mathsf{E}}_{\mathsf{R}} \to \Sigma\Big|^{\mathsf{b}}_{\mathsf{E}} \to \Sigma\Big|^{\mathsf{t}}_{\mathsf{b}} \to \alpha; \\ \alpha &\to& \sigma_{A3>200}\Big|^{\mathsf{t}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{b}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{E}}_{\mathsf{b}} \to \Gamma\Big|^{\mathsf{R}}_{\mathsf{E}} \to \Sigma\Big|^{\mathsf{E}}_{\mathsf{R}} \\ & \to & \Sigma\Big|^{\mathsf{b}}_{\mathsf{E}} \to \Sigma\Big|^{\mathsf{t}}_{\mathsf{b}} \to \pi_{A1,A4}\Big|^{\mathsf{t}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{b}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{E}}_{\mathsf{b}} \\ & \to & \Gamma\Big|^{\mathsf{R}}_{\mathsf{E}} \to \mathit{resultSet}; \\ \alpha &\to& \varphi_{C_1(A3)}\Big|^{\mathsf{v}}_{\mathsf{t}} \to C_1(A3) \Big\}. \end{array}$$

If we then merge according to transformation rule **F3** the plan for piggyback operation P_2 and an additional plan $D(P_3) = \left\{ \texttt{R1} \to \varphi_{T_2}\Big|^{\mathsf{v}}_{\mathsf{b}} \to T_2(R1) \right\}$ to find the block count (*i.e.*, statistic T_2) of `R1`, we would have

$$\begin{array}{lll} \multicolumn{3}{l}{D(Q_1) + D(P_1) + D(P_2) + D(P_3) \;=\; \Big\{} \\ \texttt{R1} &\to& \Sigma\Big|^{\mathsf{E}}_{\mathsf{R}} \to \Sigma\Big|^{\mathsf{b}}_{\mathsf{E}} \to \beta; \\ \beta &\to& \Sigma\Big|^{\mathsf{t}}_{\mathsf{b}} \to \alpha; \\ \beta &\to& \varphi_{T_2}\Big|^{\mathsf{v}}_{\mathsf{b}} \to T_2(R1); \\ \alpha &\to& \sigma_{A3>200}\Big|^{\mathsf{t}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{b}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{E}}_{\mathsf{b}} \to \Gamma\Big|^{\mathsf{R}}_{\mathsf{E}} \to \Sigma\Big|^{\mathsf{E}}_{\mathsf{R}} \\ & \to & \Sigma\Big|^{\mathsf{b}}_{\mathsf{E}} \to \Sigma\Big|^{\mathsf{t}}_{\mathsf{b}} \to \pi_{A1,A4}\Big|^{\mathsf{t}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{b}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{E}}_{\mathsf{b}} \\ & \to & \Gamma\Big|^{\mathsf{R}}_{\mathsf{E}} \to \mathit{resultSet}; \\ \alpha &\to& \varphi_{C_1(A3)}\Big|^{\mathsf{v}}_{\mathsf{t}} \to C_1(A3); \\ \alpha &\to& \varphi_{T_1}\Big|^{\mathsf{v}}_{\mathsf{t}} \to T_1(R1) \Big\}. \end{array}$$

Finally, we collapse the plan according to **F4**, **F5** and **F6** to get

$$\begin{array}{lll} \multicolumn{3}{l}{D(Q_1) + D(P_1) + D(P_2) + D(P_3) \;=\; \Big\{} \\ \texttt{R1} &\to& \Sigma\Big|^{\mathsf{b}}_{\mathsf{R}} \to \beta; \\ \beta &\to& \Sigma\Big|^{\mathsf{t}}_{\mathsf{b}} \to \alpha; \\ \beta &\to& \varphi_{T_2}\Big|^{\mathsf{v}}_{\mathsf{b}} \to T_2(R1); \\ \alpha &\to& \sigma_{A3>200}\Big|^{\mathsf{t}}_{\mathsf{t}} \to \pi_{A1,A4}\Big|^{\mathsf{t}}_{\mathsf{t}} \to \Gamma\Big|^{\mathsf{R}}_{\mathsf{t}} \to \mathit{resultSet}; \\ \alpha &\to& \varphi_{C_1(A3)}\Big|^{\mathsf{v}}_{\mathsf{t}} \to C_1(A3); \\ \alpha &\to& \varphi_{T_1}\Big|^{\mathsf{v}}_{\mathsf{t}} \to T_1(R1) \Big\}. \end{array}$$

<u>**Example 2:**</u> Now, let us consider another example using a scan of the table `R2` via an index on column `A3` and a piggyback operation φ_{I_2} to find the height of this index tree.[5] We use the convention that a table such as `R2` in a data flow plan implicitly includes by reference all available index structures. An index scan can be represented as a data flow plan that generates a

[5]This example is used only to illustrate the notation and technique. In practice, the height of the index tree may simply be maintained as the root node is split.

set of values for qualified block ids, with concurrent access to the corresponding data blocks of the table. Using these two data flows as the input, an iterator fetches only those blocks of data that have at least one qualified row. Finally a σ operation selects rows that actually qualify for the given condition from the retrieved blocks. Using our notation and the symbols for operations defined in Tables 3, 4 and 5 of Section 3, a data flow plan for an index scan to select all rows satisfying `A3>200` from `R2` can be described as

$$
\begin{array}{lll}
\multicolumn{3}{l}{D(\texttt{ISCAN}_{A3>200}(R2)) \; = \; \Big\{} \\
\texttt{R2} & \rightarrow & \delta_{A3>200}\Big|^{\mathsf{S}}_{\mathsf{R}} \rightarrow \; \alpha; \\
\texttt{R2} & \rightarrow & \alpha; \\
\alpha & \rightarrow & \kappa\Big|^{\mathsf{E}}_{\mathsf{S,\ R}} \rightarrow \; \sigma_{A3>200}\Big|^{\mathsf{R}}_{\mathsf{E}} \rightarrow \; \mathit{resultSet} \;\; \Big\}.
\end{array}
$$

In order to determine the height of the index tree in this way, we must have access to the iterator that traverses the index from root to leaf. So, we might implement the φ_{I_2} operation by getting the index, traversing a single leaf path from the root, scanning the nodes in that path and counting them, as follows:

$$
\begin{array}{l}
D(\varphi_{I_2(A3)}(R2)) \; = \; \Big\{ \; \texttt{R2} \rightarrow \; \iota_{A3}\Big|^{\mathsf{T}}_{\mathsf{R}} \rightarrow \; \lambda_{A3>200}\Big|^{\mathsf{T}}_{\mathsf{T}} \\
\qquad\qquad \rightarrow \; \Sigma\Big|^{\mathsf{n}}_{\mathsf{T}} \rightarrow \; \texttt{COUNT}\Big|^{\mathsf{v}}_{\mathsf{n}} \rightarrow \; I_2(A3) \;\; \Big\}.
\end{array}
$$

If we expand the sub-operation δ, we can replace `R2` $\rightarrow \; \delta_{A3>200}\Big|^{\mathsf{S}}_{\mathsf{R}} \rightarrow \; \alpha$ by

$$
\begin{array}{lll}
\texttt{R2} & \rightarrow & \iota_{A3}\Big|^{\mathsf{T}}_{\mathsf{R}} \rightarrow \; \lambda_{A3>200}\Big|^{\mathsf{T}}_{\mathsf{T}} \rightarrow \; \chi_{A3>200}\Big|^{\mathsf{T}}_{\mathsf{T}} \\
 & \rightarrow & \Sigma\Big|^{\mathsf{n}}_{\mathsf{T}} \rightarrow \; \delta_{A3>200}\Big|^{\mathsf{S}}_{\mathsf{n}} \rightarrow \; \alpha.
\end{array}
$$

Then merging of the two plans would give us

$$
\begin{array}{lll}
\multicolumn{3}{l}{D(\texttt{ISCAN}_{A3>200}(R2)) + D(\varphi_{I_2(A3)}(R2)) \; = \; \Big\{} \\
\texttt{R2} & \rightarrow & \iota_{A3}\Big|^{\mathsf{T}}_{\mathsf{R}} \rightarrow \; \lambda_{A3>200}\Big|^{\mathsf{T}}_{\mathsf{T}} \rightarrow \; \beta; \\
\beta & \rightarrow & \Sigma\Big|^{\mathsf{n}}_{\mathsf{T}} \rightarrow \; \texttt{COUNT}\Big|^{\mathsf{v}}_{\mathsf{n}} \rightarrow \; I_2(A3); \\
\beta & \rightarrow & \chi_{A3>200}\Big|^{\mathsf{T}}_{\mathsf{T}} \rightarrow \; \Sigma\Big|^{\mathsf{n}}_{\mathsf{T}} \rightarrow \; \delta_{A3>200}\Big|^{\mathsf{S}}_{\mathsf{n}} \rightarrow \alpha; \\
\texttt{R2} & \rightarrow & \alpha; \\
\alpha & \rightarrow & \kappa\Big|^{\mathsf{E}}_{\mathsf{S,\ R}} \rightarrow \; \sigma_{A3>200}\Big|^{\mathsf{R}}_{\mathsf{E}} \rightarrow \; \mathit{resultSet} \; \Big\}.
\end{array}
$$

Example 3: Finally, let us consider a block nested loop join that uses the data flow plan of Example 1 for the outer loop, and the plan of Example 2 for the inner loop. As described in [24], more complex queries such as this one can be piggybacked via the component unary queries.

A block nested loop join requires an additional sub-operation, `LOOP`, that produces a given number of copies of its input. Conceptually, we will represent the block nested loop as a count of the number of block extents in the outer loop, which will be used as an input to the `LOOP` operation. This can be represented in our notation as follows, where `R1` is taken to be the outer table and `R2` is the inner table

$$
\begin{array}{lll}
\multicolumn{3}{l}{D(\texttt{BNLJ}_{R1,R2}) \; = \; \Big\{} \\
\texttt{R1} & \rightarrow & \Sigma\Big|^{\mathsf{E}}_{\mathsf{R}} \rightarrow \alpha; \\
\alpha & \rightarrow & \texttt{COUNT}\Big|^{\mathsf{v}}_{\mathsf{E}} \rightarrow \beta; \\
\texttt{R2} & \rightarrow & \beta; \\
\beta & \rightarrow & \texttt{LOOP}\Big|^{\mathsf{R}}_{\mathsf{R,\ v}} \rightarrow \; \Sigma\Big|^{\mathsf{E}}_{\mathsf{R}} \rightarrow \gamma; \\
\alpha & \rightarrow & \gamma; \\
\gamma & \rightarrow & \bowtie\Big|^{\mathsf{R}}_{\mathsf{E,\ E}} \rightarrow \; \mathit{resultSet} \;\; \Big\}.
\end{array}
$$

If the query optimizer generates a plan that replaces `R2` by `ISCAN`$(R2)$, we can perform exactly the same manipulations on the outer table as in Example 1 and on the first loop of the inner table as in Example 2. Note that for some piggyback operations, such as φ_{C_4} to find the distribution of a column, multiple passes over the data are useful or required. When integrating these piggyback operations, the DBMS can take advantage of the multiple passes over the inner table to compute some of these statistics.

4.5 Experimental Validation

We have demonstrated experimentally that the amount of overhead for a relatively simple set of piggyback operations increases substantially as the interleaving with a user query is implemented at coarser levels of data granularity (*i.e.*, less tightly integrated with the query processing engine). In our preliminary experiments — which were conducted using our piggybacking prototype for single-predicate queries

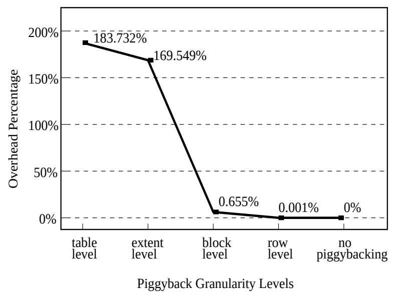

Figure 7: Overhead percentage by granularity level

on a Pentium II 300 PC running Linux 2.2.12 — the execution time to collect the minimum- and maximum-value statistics for all columns in a table is about as twice (183.732%) as that of the user query when the piggybacking was performed at the table level, while the execution time becomes almost negligible (0.001%) when the piggybacking was performed at the row level. The differences in overhead according to the levels of data granularity at which the piggybacking is performed is shown graphically in Figure 7. Clearly, the piggybacking overhead dramatically decreases when the level of data granularity becomes finer and finer. In addition, we have found that the percentage of overhead incurred for a given piggyback operation (See Figure 8 for a row-level piggybacking) decreases as the size of the available data buffer increases.

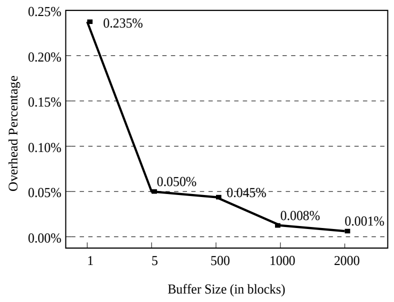

Figure 8: Overhead percentage by buffer size

5 Conclusions

The concept of multiple-granularity interleaving is a useful tool for the efficient implementation of the piggyback method within a DBMS. We have demonstrated the need for such a mechanism to quickly and automatically combine the processing of user queries and piggyback operations at an appropriate level of data granularity, especially in the context of an existing DBMS. The number and type of statistics to be collected and maintained and the dynamic nature of user query workloads require a generic approach that can determine efficient interleavings with a limited description of the rules for doing so. Our representation of query and piggyback operations as iterators in data flow plans at multiple levels of data granularity gives a flexible and consistent model for combining the two types of operations efficiently. Using the various techniques for interleaving and plan manipulation, one can build a general toolkit for implementing this model in practice. Descriptions of the semantics preservation properties of the various operations provides a guarantee that the behavior of the interleaved operations is consistent with the results of the individual operations in isolation. Finally, our work with commercial and research prototype DBMSs indicates that the technique of multiple-granularity interleaving is an effective mechanism for the implementation of the piggyback method.

We note that there are still some open issues not addressed by this work. For example, we recognize that it should be possible in a DBMS using the piggyback method to selectively disable piggybacking or even specific piggyback operations. Currently, our algorithm for multiple-granularity interleaving does not explicitly take this into account. Also, on parallel hardware processing of common data can be performed by different physical processors, which brings up issues of synchronization, scheduling, and buffer management. However, exclusive (*i.e.*, write) access to the data is not required by most piggyback operations, so some of these issues will not be as complex as in arbitrary parallel processing.

Acknowledgments

The authors would like to thank Nandit Soparkar, Berni Schiefer, Calisto Zuzarte, Cathy McArthur and Gabby Silberman for their support and valuable comments for the work reported in this paper.

About the Authors

Brian Dunkel is a Ph.D. candidate in the Dept of Electr. Eng. and Comp. Sci. at The Univ. of Michigan in Ann Arbor. His primary areas of research interest are in the use of pre-computed and materialized information to support data mining, query optimization, and Internet information retrieval technology. He has published several research papers in these areas, and has worked with IBM Research at the CAS in Toronto and the T.J. Watson Center in New York. He has also worked on software engineering issues at the Union Bank, Switzerland, and on neural network systems at Mitre Corp., USA. His undergraduate research at the Massachusetts Institute of Technology (MIT) involved hypertext authoring and the visual representation of algorithms, which contributed to an interactive CD-ROM version of textbook "*Introduction to Algorithms*" by Cormen *et al.* (McGraw-Hill/MIT Press).

Qiang Zhu is an Assistant Professor in the Dept of Comp. and Inf. Sci. at The Univ. of Michigan - Dearborn. He received his Ph.D. in Comp. Sci. from the Univ. of Waterloo in Canada in 1995. He also holds an M.Sc. from the McMaster Univ. in Canada, an M.Eng. and a B.Sc. both from the Southeast Univ. in China. He was a lecturer in Comp. Sci. at the Southeast Univ. from 1984 to 1988. Dr. Zhu has over fourteen years research experience on centralized/distributed database systems. He is a principal investigator for a number of database research projects funded by sources including the National Science Foundation and IBM at The Univ. of Michigan. He has published many research papers in various journals and conference proceedings. Some of his research results have been included in several database research/text books. Dr. Zhu has served as a session chair and program committee member for some research conferences. His current research interests include query processing and optimization, multidatabase systems, data mining, and Web database technology.

Wing Lau is an M.Sc. student in the Dept of Comp. and Inf. Sci. at the The Univ. of Michigan - Dearborn. She is also a research assistant and the computer lab manager in the department. She received her B.Sc. in Comp. Sci. from the same university in 1997. She was a software engineer at Marquip Inc. (Madision, WI) from 1997 - 1998. Her research interests include query processing and optimization in database systems.

Suyun Chen is a Staff Development Analyst in the Database Technology group at IBM Toronto Lab. She joined IBM in 1995 after she received her Ph.D. degree from the McMaster Univ. in Canada. Dr. Chen also obtained a M.Sc. degree from the Jiangxi Normal Univ. and a B.Sc. degree from the Zhongshan Univ. in China. Dr. Chen has been actively involved in a number of research projects on query optimization and system reliability at IBM. She served as a referee for several technical journals. Her current research interests include statistical techniques in databases, query optimization, performance evaluation, system testing, and software reliability.

References

[1] H. Ahmed et al. A vector dataflow architecture. In *Proc. of the Int'l Conf. on Databases, Paral. Arch. and Their Appl.*, 1990.

[2] J.R. Alsabbagh and V.V. Raghavan. A framework for multiple-query optimization. In *2nd Int'l Workshop on Research Issues in Data Eng.: Trans. and Query Proc.*, pp 157–62, Tempe, Arizona, 1992.

[3] J.R. Alsabbagh et al. Analysis of common subexpression exploitation models in multiple-query processing. In *Proc. of the Int'l Conf. on Data Eng.*, pp 488–97, Houston, Texas, 1994.

[4] U.S. Chakravarthy et al. Multiple query processing in deductive databases using query graphs. In *Proc. of VLDB'86*, pp 384–91, Kyoto, Japan, 1986.

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

[6] S. Christodoulakis. Estimating record selectivites. *Inf. Syst.*, 8(2):105–15, 1983.

[7] S. Christodoulakis. Common subexpression processing in multiple-query processing. *IEEE Trans. on Knowl. and Data Eng.*, 10(3):493–99, 1998.

[8] G. Copeland et al. Buffering schemes for permanent data. In *Proc. of the Int'l Conf. on Data Eng.*, pp 214–21, Los Angeles, California, 1986.

[9] P.J. Haas et al. Sampling-based estimation of the number of distinct values of an attribute. In *Proc. of VLDB'95*, pp 311–22, Zurich, Switzerland, 1995.

[10] L.H. Jamieson. Features of parallel algorithms. In *Proc. of the 2nd Int'l Conf. on Supercomp.*, 1987.

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

[12] R.J. Lipton et al. Practical selectivity estimation through adaptive sampling. In *Proc. of SIGMOD'90*, pp 1–11, 1990.

[13] L. Mackert and G. Lohman. Index scans using a finite LRU buffer: A validated I/O model. Technical Report RJ4836, IBM Almaden, 1985.

[14] M.V. Mannino et al. Statistical profile estimation in database systems. *ACM Comp. Surveys*, 20(3), Sept. 1988.

[15] J. McNally et al. *Informix Unleashed.* SAMS Publishing, 197.

[16] G. Pernul, A.M. Tjoa, and T.J. Teorey. A view integration approach for the design of multilevel secure databases. In *Proc. of the 10th Int'l Conf. on the E-R Approach*, 1991.

[17] P.G. Selinger et al. Access path selection in distributed data base management systems. In *Proc. of the Int'l Conf. on Data Bases*, pp 204–15, Aberdeen, Scotland, 1980.

[18] T.K. Sellis. Multiple-query optimization. *ACM Trans. on DB Syst.*, 13(1):23–52, March 1988.

[19] G.P. Shapiro et al. Accurate estimation of the number of tuples satisfying a condition. In *Proc. of SIGMOD'84*, pp 256–76, 1984.

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

[21] C.T. Yu et al. Adaptive techniques for distributed query optimization. In *Proc. of the Int'l Conf. on Data Eng.*, pp 86–93, Los Angeles, California, 1986.

[22] M.J. Yu et al. Adaptive query optimization in dynamic databases. *Int'l J. on Artif. Intelli. Tools*, 7(1):1–30, 1998.

[23] V. Zander et al. Estimating block accesses when attributes are correlated. In *Proc. of VLDB'86*, pp 119–27, Kyoto, Japan, 1986.

[24] Q. Zhu, B. Dunkel, N. Soparkar, S. Chen, B. Schiefer, and T. Lai. A Piggyback Method to Collect Statistics for Query Optimization in Database Management Systems. In *Proc. of the 1998 CASCON*, pp 67–82, Toronto, Canada, 1998.