Primitives for Workload Summarization and Implications for SQL

Surajit Chaudhuri
Microsoft Research
Redmond, USA
surajitc@microsoft.com

Prasanna Ganesan
Stanford University
Palo Alto, USA
prasannag@cs.stanford.edu

Vivek Narasayya
Microsoft Research
Redmond, USA
viveknar@microsoft.com

Abstract

Workload information has proved to be a crucial component for database-administration tasks as well as for analysis of query logs to understand user behavior and system usage. These tasks require the ability to summarize large SQL workloads. In this paper, we identify primitives that are important to enable many important workload-summarization tasks. These primitives also appear to be useful in a variety of practical scenarios besides workload summarization. Today's SQL is inadequate to express these primitives conveniently. We discuss possible extensions to SQL and the relational engine to efficiently support such summarization primitives.

1. Introduction

The past few years have seen the emergence of a large class of tasks that benefit from analysis of SQL workload information. Some examples of such tasks are database administration [3,27] and understanding user/application behavior [14,15]. The problem of collecting a SQL workload is itself rather easy as most commercial database vendors provide profiling tools that enable logging of SQL activity on the server over a representative period of time, ranging from minutes to days. However, little attention has been paid to understanding the requirements for analysis of the collected workload.

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the VLDB copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the Very Large Data Base Endowment. To copy otherwise, or to republish, requires a fee and/or special permission from the Endowment

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

In analyzing the workloads, one of the key requirements is identifying a *small representative subset* that captures the essence of the large workload that has been logged using automated tools. There are multiple reasons why picking such a small representative subset is necessary. First, the resources needed to accomplish tasks such as index tuning grows (often super-linearly) with increasing sizes of workloads. There is a significant benefit to be gained by "filtering'' the workload before it is fed to these tasks, while not compromising its characteristics. A second motivation arises from the necessity of having to meaningfully summarize data for viewing by DBAs or analysts. It is important to be able to offer relatively small representative synopses of the workload, before "drilling down" to identify the queries of interest.

The simplest way to obtain a representative subset is to pick a uniform random sample of the workload. While sampling is conceptually simple and, in fact, useful in many situations, a DBA may like to obtain a representative subset that has additional constraints, e.g., pick a representative workload with the 100 most expensive queries while ensuring that every table in the database occurs in at least 5 queries. Thus, the specification for picking a representative subset for a workload (henceforth called *summary*) depends on the task at hand and, more often than not, sampling in itself is not adequate.

A natural approach is to express the specification for a summary workload as a SQL statement over the entire workload represented in a relational database. We discover that today's SQL provides inadequate support for conveniently specifying such summary workloads. In trying to understand the precise implications for SQL to better support workload analysis, we identify two key primitives. As with traditional filtering, both these primitives take as input a set of statements and output a subset of those statements. The first primitive, that we call *dominance,* allows filtering based on a partial order among the statements in the workload. Thus, statements that are "dominated" by other statements in the workload (as determined by the partial order) are filtered out. For

example, when considering workload summarization for the index-selection task, whenever two queries Q_1 and Q_2 are identical in all respects except, say, in their GROUP BY clause, and if Q_1 contains a superset of the GROUP BY columns of Q_2 (e.g., Q_1 has GROUP BY A,B,C and Q_2 has GROUP BY A,B), we could declare that Q_1 dominates Q_2 , since the indexes chosen for Q_1 are likely to be adequate for Q_2 as well.

The second primitive, which we call *representation,* is a form of combinatorial optimization. It allows specification of a subset of the workload such that a certain objective function (an aggregate expression over an attribute) is maximized or minimized, subject to a set of constraints. For example, in the index-selection task, we may wish to constrain the output to no more than 1000 statements in the workload, while trying to maximize "coverage" (say total execution time of all statements in the output) of the workload. It should also be noted that both the dominance and representation primitives need to be able to provide aggregate information as well. For example, we should able to know the count of the total number of statements a "dominating" statement in the workload (Q1 in the example above) has "dominated".

Of course, studying the implications of and mechanisms for supporting these primitives in SQL may not be worthwhile if these primitives are useful only for a specialized application such as workload analysis. However, the above primitives are useful in a variety of scenarios besides workload summarization. This broader applicability motivates us to investigate their inclusion in SQL. The dominance primitive is a generalization of the Skyline operator [5], and, we describe how, with suitable extensions, it can leverage algorithms that have been proposed [5,23,26] for implementing the Skyline operator. Implementing the representation primitive in SQL requires us to address trade-offs between functionality and complexity. While the representation primitive can potentially have wide usage, in its full generality it requires us to solve the Integer Programming (IP) problem that is known to be NP-hard [19]. While several algorithms are known for solving the IP problem e.g., [31], their integration into SQL directly can result in a very heavyweight operator whose usefulness on large data sets may be rather limited. Thus, we propose an alternative, simpler extension to SQL that allows efficient and scalable implementation strategies. While falling short of the optimization guarantees of the representation primitive, this simpler extension appears to be adequate for many common summarization tasks.

As a proof-of-concept of the expressiveness and utility of these primitives, we evaluate them in the context of summarizing workloads for index selection. We compare our solution against a previously presented method for workload compression [11] that requires the definition of a distance function for index selection, which then uses a clustering-based method for compression.

The rest of this paper is organized as follows. Section 2 describes primitives that are useful for several summarization tasks. Section 3 presents a language that exposes these primitives declaratively and shows how various workload-summarization tasks can be expressed in this language. In Section 4, we describe a few different scenarios (besides workload summarization) where these primitives also appear to be useful. In Section 5, we discuss possible extensions to SQL and the query engine to support these primitives. Section 6 presents a brief evaluation of these primitives relative to workload compression [11]. We discuss related work in Section 7 and conclude in Section 8.

2. Workload Summarization

In this section, we describe several examples of workload summarization tasks, and then identify the logical operations that are common across these tasks. To make the examples that we present specific, we first present a *workload schema* that describes the attributes of each statement in the workload.

2.1 Workload Schema

Every workload-summarization task expects its workload input to conform to a specific schema for workloads. A workload schema defines the set of attributes of each statement in the workload that the task may reference. In this section, we present an example of such a workload schema. Our purpose here is to introduce attributes that can be referenced in the examples presented in the paper. In general, we expect different applications to use potentially different workload schemas. We note that as workload analysis matures, a core standardized schema can facilitate easy sharing of information across different applications.

Attribute	***Type***	***Meaning***
StmtType	Atomic	Statement type: (SELECT, UPDATE, INSERT, DELETE)
SQLString	Atomic	SQL String of statement
Timestamp	Datetime	Time when statement was executed
User	Atomic	Name/Id of user issuing statement
Application	Atomic	Name of application issuing statement
Weight	Atomic	A number representing the importance of this statement in the workload
FromTables	Set	Set of tables referenced in FROM clause
WhereCols	Set	Set of columns referenced in WHERE clause
JoinConds	Set	Set of join conditions in

		statement
GroupByCols	Set	Set of columns in GROUP BY clause
OrderByCols	Sequence	Set of columns in ORDER BY clause
ProjCols	Set	Set of columns projected in query
EstimatedCost	Atomic	Optimizer estimate of query cost from plan
ExecutionCost	Atomic	Elapsed time executing statement
CPUTime	Atomic	CPU time executing statement
IOTime	Atomic	I/O time executing statement
Memory	Atomic	Max memory consumed during statement execution
IndexesUsed	Set	Set of indexes used in answering statement

Table 1. Example of schema for workload.

Our schema (shown in Table 1) contains attributes of the following types: (1) Atomic-valued (e.g., Execution cost of statement, Number of tables referenced) (2) Set-valued (e.g., Set of tables referenced) (3) Sequence (e.g., Sequence of ORDER BY columns). We note that today's relational database systems do not support set and sequence data types. In Section 5, we discuss the implications of this non-support for workload summarization as a database application.

We broadly categorize the attributes of a statement in the workload into three categories: (1) *Syntactic and Structural*. These include all attributes that describe the syntax or structure of the statement. (2) *Plan Information* (3) *Execution Information.*

2.2 Examples of Workload Summarization

In this section, we present examples of workload summarization that appear to be useful either for preparing input to automated tools or for consumption by DBAs or analysts.

Example 1: **Summarizing workloads for input to index selection tools.** Workloads often consist of different templates (e.g., stored procedures, batches) that get invoked repeatedly with different parameters. Within each template, we may want to take advantage of certain relationships between statements to filter out some queries. For example, whenever two queries (say Q_i and Q_j) are identical in all respects except their GROUP BY and ORDER BY clauses, and if Q_i.*GroupByCols* $\subset$ Q_j.*GroupByCols* and Q_i.*OrderByCols* is a prefix of Q_j.*OrderByCols*, we could require that only Q_j be included in the workload summary, since indexes that are beneficial for Q_j are likely to be adequate for Q_i as well. After this filtering step, we would like to obtain a "summary'' while ensuring that each template receives adequate representation (e.g., proportional to the number of statements in each template). Finally, we would like the workload summary to contain no more than 1000 queries such that the sum of the *Weight* attribute is maximized, i.e., we capture as much of the total weight of the original workload as possible.

Example 2: **Finding queries that are potential resource bottlenecks.** DBAs often need to find queries that are responsible for consuming the most resources (CPU, I/O, Memory). Suppose the total CPU time (resp. I/O time, Memory) consumed by all queries is CPUTotal (resp. IOTotal, MemoryTotal). One natural summarization task is to find the smallest subset of queries that covers at least 50% of CPUTotal, IOTotal and MemoryTotal.

Example 3: **Identifying columns for potential building/updating of statistics.** For this task, we want to detect queries with (a) a large discrepancy between the optimizer's estimated time and the actual execution time, and (b) having large errors in cardinality estimation. We filter out statements that do not have at least 50% error in cardinality estimation or take less than one second to execute. We then partition statements based on the tables referenced (*FromTables* attribute) and join conditions (*JoinConds* attribute). Within each partition, we narrow down the columns that could benefit from the creation/update of statistics by eliminating statements which have a superset of the columns involved in some other statement. Finally, we want no more than 5 statements per partition in the summary, and no more than a total of 100 statements, while maximizing the total value of *CostRatio* (defined as *ExecutionCost* / *EstimatedCost*) over the statements in the summary.

2.3 Key Primitives in Workload Summarization

We now introduce the common primitives that are necessary to accomplish the kinds of workload summarization tasks described in Section 2.2.

2.3.1 Filtering

This primitive is simply the "traditional" filter, i.e., it eliminates any statement in the workload that does not satisfy a given Boolean expression. An atomic condition in the filter is any predicate on an attribute of the workload schema. For example, in Example 3 above, we filter out statements with low errors in cardinality estimation or with low execution times.

2.3.2 Dominance

The dominance primitive can be used to specify a *partial order* among statements in the workload. Moreover, this partial order is used also as a filtering and aggregation (described later) operator. In particular, for any pair of statements S_1 and S_2 in the workload, if per the

partial order S_1 is "dominated by" S_2, then S_1 must be eliminated (filtered) from the output of the dominance operator, with the exception of the case when S_1 dominates S_2 *and* S_2 dominates S_1. In the latter case, S_1 and S_2 are considered *equivalent*, and it is acceptable to include either one (but not both) in the output. Thus, the semantics of dominance is that it outputs a smallest subset such that every statement not included in the output is dominated by some statement in the output. The specification of the partial order, or equivalently the test for whether a statement is dominated by another statement, is expressed by a conjunction of conditions on the attributes of the statements. In general, we expect some of these conditions to be strict *equality* conditions, and the rest to be *partial order* conditions. We refer to the attributes mentioned in the strict equality conditions as the ***partitioning attributes*** associated with the dominance primitive. In Example 1, the strict equality conditions are that *FromTables, JoinConds*, and *WhereCols* of both statements are identical, and the partial order conditions are: (1) *GroupByCols* of the first statement is a subset of *GroupByCols* of the second statement; (2) *OrderByCols* of the first statement is a prefix of *OrderByCols* of the second statement. We observe that the strict equality conditions imply a partitioning of the statements. Note also, that if in Example 1, two statements in the workload have the same group by and order by columns, then either of them (but not both) may be included in the output.

We illustrate the dominance relationship graphically in Figure 1. Each node in the graph denotes a statement and an edge from node X to node Y denotes that X dominates Y. In the figure below, the output set of statements is {A,B,C}.

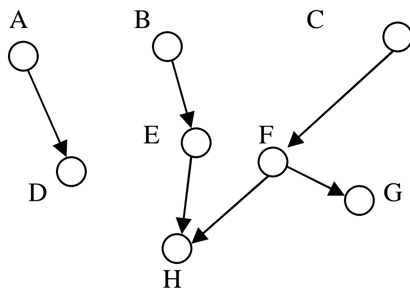

Figure 1. Graphical representation of dominance relationship.

Dominance is important for certain workload summarization tasks. For example, an index that is appropriate for statement S_2 may also be appropriate for S_1, but not necessarily the other way around. In this paper we restrict the comparison operators of the inequality conditions to *transitive* operators, such as $<$, $>$, $\leq$, and $\geq$ for atomic-valued attributes, $\subset$, $\subseteq$, $\supset$, and $\supseteq$ for set-valued attributes, and prefix-of, subsequence-of, and supersequence-of for sequence-valued attributes. We also restrict an attribute of a statement to be compared with the same attribute of the other statement. Since the dominance relationship imposes a partial order, it is transitive. As we discuss later (see Section 5), transitivity is useful in enabling an efficient implementation of dominance.

As noted earlier, the dominance primitive also represents aggregation. With every statement S in the output of the dominance operator, aggregation information over the statements that were dominated by S could be included. Each such specified aggregate becomes a new attribute (which we refer to as a *dominance-based aggregate attribute*) of each statement output by the dominance primitive. A dominance-based aggregate attribute corresponding to an output statement S is the SUM or COUNT function (or, in principle, any other aggregate function) applied to any attribute, over all the statements that were eliminated by S, and including S itself. Note that a statement can be dominated by two different statements, neither of which dominates one other. In Figure 1 above, H is dominated by E and F, but neither E nor F dominates the other. In this case, we establish a convention that H's value will contribute to the aggregate of either E or F but not both[1].

2.3.3 Representation

The representation primitive allows specification of a subset of the workload such that a certain *objective function* (an aggregate expression over an attribute) is maximized (or minimized), subject to a set of *constraints*. Thus the representation primitive specifies an optimization problem. Representation consists of five parts, each of which we describe below:

Partitioning Attributes

Representation can specify a partitioning of the input to be used, so that constraints can be specified at a per-partition level. Partitioning is similar to a GROUP BY, in that each partition corresponds to all statements that have the same values for all the partitioning attributes.

Optimization Criterion

The optimization criterion can be specified in one of the following forms: (1) Minimize an aggregate over an attribute, e.g., the number of statements in the output, subject to the constraints; (2) Maximize an aggregate over an attribute, e.g., maximize sum of *ExecutionCost* subject to the constraints.

Global Constraints

Global constraints are constraints on an aggregate computed over the entire output set of statements. We permit any conditional expression involving aggregates

[1] In general, we could allow H's value to be distributed in some manner across all statements that dominate it (E and F), but we do not consider this possibility in this paper.

over any of the attributes of the input. Referring to Example 1, we see that the requirement that the chosen statements should cover at least 75% of the total *ExecutionCost* of statements in the input workload is a global constraint.

Local Constraints
Local constraints are identical to global constraints, except that the constraint applies to each individual partition. For example, we could specify that every partition should contain enough output statements to cover at least 75% of the total statements in that partition.

Filter Constraints
Filter constraints are constraints that apply to each individual statement chosen by the representation process. A statement not satisfying the filter constraint may not be a part of the output. Note that a filter constraint may involve aggregate expressions computed over partitions or the entire input workload. For example, we could have a filter constraint requiring that every statement chosen has an execution cost at least 30% higher than the average execution cost in its partition.

3. A Language for Workload Summarization

We now briefly describe a language for *declaratively* specifying workload-summarization tasks such as the ones described earlier. This language, which we refer to as **WAL** (Workload Analysis Language), supports the primitives of filtering, partitioning, dominance and representation presented in Section 2. The purpose of introducing WAL is to highlight (a) the importance of exposing these primitives in a declarative manner, thereby making it possible to specify sophisticated summarization tasks easily, and (b) the expressiveness necessary for accomplishing these tasks. We present this language as though it were an extension of SQL and, in Section 5, we discuss the implications actually extending SQL in this fashion.

We describe the syntax for specifying a query in WAL in Section 3.1 and provide several example queries in WAL in Section 3.2. We note that the focus of this section is not on the specific syntax we propose, but rather on the tasks that are enabled by it.

3.1 WAL Syntax

The overall structure of a query in WAL is shown below.

SELECT <*select clause*>
FROM <*from clause*>
WHERE <*filter condition*>
DOMINATE <*dominate-clause* >
REPRESENT <*representation-clause*>

We now discuss each of the clauses in more detail. We do not go into details on clauses that are already part of the SQL language.

SELECT clause
The SELECT clause is similar to that of any SQL query. It permits the specification of a subset of the attributes in the workload schema, together with other aggregates. In addition, we allow defining dominance-based aggregate attributes (see Section 2.3.2). Note that dominance-based aggregates are similar to aggregates computed for each group specified by the GROUP BY clause of a traditional SQL query. The difference is that, for each output statement S, the aggregate is computed over the set of all statements dominated by S. These aggregates are defined by prefixing DOM to the traditional keywords for computing aggregates. For example, DOMSUM(*ExecutionCost)* returns, for each statement S output by the dominance primitive, the sum of the *ExecutionCost* of all statements dominated by S, and including S itself. Thus, for each aggregate function (e.g., SUM, COUNT) in SQL, there is a corresponding dominance-based aggregate function. We note that the scope of such an attribute is the block defined by the SELECT clause. Thus e.g., this attribute can be referenced in the REPRESENT clause in a constraint.

FROM clause
The FROM clause simply specifies a *single* table or view. In the context of workload summarization, it is implicit that this table or view conforms to the workload schema. Note that the view may itself be an arbitrary SQL query whose result conforms to the workload schema. For example, traditional SQL operators such as UNION, DIFFERENCE etc., could be used to combine two or more workloads in meaningful ways.

WHERE clause
As in SQL, the WHERE clause is permitted to be an arbitrary boolean condition applicable to each tuple of the table or view specified in the FROM clause.

DOMINATE clause
The syntax of the DOMINATE clause is:
DOMINATE WITH (PARTITIONING BY <*attr-list*>)
(SLAVE.*Attr Op* MASTER.*Attr)* *

Thus, the strict equality dominance conditions are specified by the shorthand PARTITIONING BY (<*attr-list*>). This is followed by the conditions that define the partial order. All conditions are implicitly ANDed. Logically, each condition for the partial order is specified by a *comparison operator* (e.g., ≤) and *an attribute* over which the comparison condition is applied. Note that SLAVE.Attr Op MASTER.Attr is a syntactic redundancy to make the query examples in this paper more readable. In this paper MASTER and SLAVE can be viewed as keywords. For reasons mentioned earlier (see Section

2.3.2) *Op* is restricted to any comparison operator that is transitive.

We observe that the SKYLINE OF clause proposed in [5] is a special case of the DOMINATE clause. The SKYLINE OF clause specifies a set of attributes A_1, … A_k and a direction (MIN/MAX) with each attribute. Thus, for example, SKYLINE of A min, B max maps to *SLAVE.A* > *MASTER.A* and *SLAVE.B* < *MASTER.B* in our definition of dominance. Our definition of dominance generalizes SKYLINE by allowing creation of dominance-based aggregate attributes (see Section 2.3.2 and SELECT clause above), which may be referenced in other parts of the WAL query (e.g., in the representation clause).

REPRESENT Clause

This clause allows specification of the representation primitive. In particular, it allows specifying: (1) the partitioning attributes, (2) the objective function to maximize or minimize, and (3) the constraints the output must satisfy. We now present the syntax of the representation clause and then describe it in more detail.

REPRESENT WITH (PARTITIONING BY <*attr-list*>)
[MAXIMIZING | MINIMIZING] <*aggr-expr*>
(GLOBAL CONSTRAINT <*global-constraint*>)*
(FILTER CONSTRAINT <*filter-constraint*>)*
(LOCAL CONSTRAINT <*local-constraint*>)*

The optional PARTITIONING BY specifies a set of attributes on which to partition the statements in the workload. Note that LOCAL CONSTRAINTs are meant to be used only if PARTITIONING BY is specified.

<*aggr-expr*> is an aggregate expression of the form *Aggregate(Attribute)* that is to be maximized (or minimized) subject to specified constraints, and we refer to it as the optimization *criterion*. *Aggregate* can be the SUM or COUNT aggregate function.

<*filter-constraint*> is a condition of the form *(Attr Op Expression)*. <*global-constraint*> and <*local-constraint*> are both conditions of the form (*Aggregate(Attr) Op Expression)*. In all these constraints, *Expression* can involve constants, aggregates on an attribute computed over the entire set of statements input to the representation primitive (obtained by prefixing the aggregate by the keyword GLOBAL), or aggregates computed over the set of tuples within a partition (obtained by prefixing the aggregate with the keyword LOCAL). Note that LOCAL can only be used in a filter or local constraint, and not in a global constraint.

3.2 Examples of Summarization Tasks in WAL

We now present several examples of workload-summarization tasks expressed as queries in WAL. These examples highlight the expressiveness and usefulness of our primitives exposed in a declarative interface. We begin by giving corresponding WAL queries for Examples 1-3 presented in Section 2.2, and then give a couple more examples. In all the examples below, we assume that *WorkloadTable* is the name of the table containing the workload statements according to the schema described in Table 1 (see Section 2.1).

Example 1. Preparing workload for input to index selection tool (see Section 2.2 for detailed description).

SELECT *, DOMSUM(*Weight*) AS *Dom_Weight*
FROM *WorkloadTable*
DOMINATE WITH PARTITIONING BY
FromTables, *JoinConds*, *WhereCols*
SLAVE.*GroupByCols* SUBSET MASTER.*GroupByCols*
SLAVE.*OrderByCols* PREFIX MASTER.*OrderByCols*
REPRESENT WITH PARTITIONING BY
FromTables, *JoinConds*, *WhereCols*
MAXIMIZING Sum(DOM_Weight)
GLOBAL CONSTRAINT *Count(*)* ≤ 200
LOCAL CONSTRAINT *Count(*)* ≥
*int(200*LOCAL.Count(*)/GLOBAL.Count(*))*

Example 2. **Finding queries that are potential performance bottlenecks** (see Section 2.2 for detailed description)**.**

SELECT * FROM *WorkloadTable*
REPRESENT WITH
MINIMIZING COUNT(*)
GLOBAL CONSTRAINT SUM(*CPUTime*) >
0.50 * GLOBAL.SUM(CPUTime)
GLOBAL CONSTRAINT SUM(IO*Time*) >
0.50 * GLOBAL.SUM(IOTime)
GLOBAL CONSTRAINT SUM(*Memory*) >
0.50 * GLOBAL.SUM(*Memory*)

Example 3. **Identifying columns for potential building/updating of statistics** (see Section 2.2 for detailed description)**.**

SELECT * FROM *WorkloadTable*
WHERE ABS(*CardEst – CardActual*)/*CardActual* > 0.5
AND *(ExecutionCost > 1.0)*
DOMINATE WITH PARTITIONING BY
FromTables, JoinConds
SLAVE.*SelectCols* SUBSET MASTER.*SelectCols*
SLAVE.*WhereCols* SUBSET MASTER.*WhereCols*
REPRESENT WITH PARTITIONING BY
FromTables, JoinConds
MAXIMIZING SUM(*CostRatio*)
GLOBAL CONSTRAINT *Count*(*) ≤ 100
LOCAL CONSTRAINT *Count*(*) ≤ 5

Example 4**. Obtaining summary of workload for use in building samples of database for approximate processing of aggregation queries.** We refer the reader to [9,10,18] for more details on the role of workload information in approximate query processing**.** In this example, among all queries in each partition specified in

the representation clause, we are requesting at most 10 queries. We also require that the total number of queries does not exceed 500, while maximizing the total weight of all queries that are selected.

SELECT * FROM *WorkloadTable*
REPRESENT WITH PARTITIONING BY
FromTables, JoinConds, GroupByCols, WhereCols
MAXIMIZING SUM(*Weight*)
GLOBAL CONSTRAINT Count(*) ≤ 500
LOCAL CONSTRAINT Count(*) ≤ *10*

***Example 5*. Finding queries in each application with low relative index usage.** Find a subset of at most 100 queries in the workload maximizing total execution cost such that, for each application, we pick a subset of queries that has lower-than-average index usage despite having a higher-than-average number of tables referenced, compared to other queries from that application.

SELECT * FROM WorkloadTable
REPRESENT WITH PARTITIONING BY *Application*
MAXIMIZING Sum(*ExecutionCost*)
GLOBAL CONSTRAINT COUNT(*) <= 1000
LOCAL CONSTRAINT AVG(*NumIndexesUsed*) <
0.75 * LOCAL.AVG(*NumIndexesUsed*)
LOCAL CONSTRAINT AVG(*NumTables*) >
1.25 * LOCAL.AVG (*NumTables*)

3.3 Discussion on Complexity

We briefly discuss the algorithmic complexity of each of the primitives discussed in Section 2.2 and supported in WAL. We assume that the table (or view) specified in the FROM clause has n statements. The *filtering* primitive can be performed in linear time, and the partitioning operation (required both for dominance and representation primitives) can also be performed in linear time using a hash-based scheme (or $O(n \log n)$ time using a sort-based scheme). The *dominance* operation can be performed in worst-case $O(n^2)$ time by testing the dominance conditions for each pair of statements.

The complexity of the *representation* primitive depends on the specific formulation. In its full generality, the problem of finding a subset of statements that maximizes (or minimizes) an aggregate subject to a set of linear constraints is equivalent to the 0-1 Integer Programming problem, which is known to be NP-Hard [19]. Even simplifications of the general problem are known to be hard. For example, the problem of minimizing Count(*) subject to a *bounded* number of global constraints of the form $\text{SUM}(Attr_i) \geq k_i$, where each of the attributes is confined to being arbitrary, non-negative real numbers, can be shown to be NP-hard by a simple reduction to the Partition problem (see [19] for a description of the Partition problem). Likewise, the problem of maximizing $\text{SUM}(Attr_0)$ subject to c global constraints: $\text{SUM}(Attr_1) \leq k_1$, …. $\text{SUM}(Attr_c) \leq k_c$ is the well-known multi-dimensional knapsack problem which is also NP-Hard [7].

Given the complexity of the representation operation, a natural question is how this operation can be supported in practice. We note that the general problem (0-1 Integer Programming problem) has been well-studied and several standard software packages exist for solving it (e.g., [31]). Such solutions could be invoked outside the database server directly by the workload-summarization application. Unfortunately, even the best-known solutions do not scale well for large inputs (e.g., millions of statements in the workload). Our observation is that a large class of workload-summarization tasks does not need to solve the most general optimization problem. In Section 5, we discuss incorporating support for less general forms of representation inside a SQL query engine that can provide optimal answers efficiently for some simple, common cases and approximate or heuristic answers for more complex cases.

4. Applicability of Primitives to Other Scenarios

Although the dominance and representation primitives presented in this paper are motivated by the need for effective workload summarization, the applicability of these operations is not limited to workload summarization. In this section we present a few other domains where these operations could be useful in complementing existing analysis techniques in the respective domains.

Scenario 1: Customer Relationship Management (CRM)

Consider a company that wants to mail product catalogs to its customers. The company has a fixed budget for mailing costs. The concept of dominance can help in this scenario as follows: To avoid sending multiple catalogs to a single address, the company considers all customers with the same address as equivalent, and will pick exactly one customer at that address, for example, the person in the household with the highest income. To maximize the expected benefit from the mailing, the company may like to select a subset of customers with largest total "importance" (e.g., measured by money spent on their products in the past). Representation is useful for specifying such a subset while not exceeding the mailing-cost budget (a global constraint), and ensuring that exactly one customer is picked from each address (a local constraint).

Scenario 2: Personalization

Consider personalization of web pages based on user profiles. When a user requests a web page, only a fixed number of targeted ads (say **K**) can typically be displayed on that page. The concept of partitioning and local

constraints can be useful to specify that that at most two ads from each category (such as food, jewelry, books etc.) should be chosen. Dominance can be useful in specifying whether, within a category (based on user's profile), the expensive items or the inexpensive items should dominate. Representation is necessary since the company running the web site wishes to pick a subset of ads such that a certain objective function (e.g., likelihood of click-throughs) is maximized, while not exceeding the global constraint of **K** ads.

Scenario 3: Web-Community Management

Consider a web community scenario where an incoming question needs to be answered by locating a certain set of "experts" on the subject. The goal is to provide a timely response from as highly rated an expert as possible. For cost effectiveness, we do not want to request more than **N** experts for any given question. Dominance can be useful to partition experts according to different time zones in order to improve chances of a quick response. Within each zone, we can define a person with higher expertise rating *and* average response time as "dominating" any other expert with lower rating and higher average response time. Representation can be useful for specifying that we find the subset of at most N experts while maximizing the sum of the expertise rating, subject to having at least one expert from each partition.

5. Dominance and Representation in SQL Engine

The examples of workload-summarization tasks and the tasks from other domains that we have presented indicate that the primitives of *dominance* and *representation* have broad applicability. There are two issues to consider for implementing workload-summarization tasks as SQL applications. First, workload-summarization queries reference attributes of type Set and Sequence (see schema in Table 1), which are not supported in today's database systems. We do not address this issue in the paper other than to note that these data types are already finding their way into SQL standards (e.g., array types are part of the SQL 1999 standard [30] and multi-set types are part of the SQL 2003 standard [30], currently in the final stages of standardization).

The second issue is that today's commercial database systems do not support the primitives of dominance and representation. We have already indicated in Section 3.1 how dominance and representation may possibly be exposed in the query syntax of SQL. While our focus is not on the narrow specifics of the syntax, for ease of exposition, in the rest of this section we will assume the syntax for exposing dominance and representation as described in Section 3.1.

Before discussing how these two primitives could be supported by a SQL query engine, we briefly consider the physical operator for partitioning. Partitioning is important as partitioning attributes are specified as part of both dominance and representation clauses (see Sections 2.3 and 3.1). We note that partitioning of the input can be achieved either by hashing or by sort-based methods (with the latter possibly exploiting indexes or existing orders on the input). Once the input is partitioned, dominance and representation operators may need to be invoked within each partition. Note that techniques from group-wise processing [8,12] can be leveraged for implementation of the dominance and special forms of representation. Such group-wise processing allows an arbitrary sub-query to be executed inside each partition. The result of the overall query is the union over the results of the sub-query over each partition. However, the general problem of representation involves challenges that go beyond group-wise processing. Moreover, explicitly supporting dominance and representation as part of the syntax facilitates specification as well as optimization[2].

5.1 Implementing Dominance

We now discuss the physical operator necessary for the dominance primitive, i.e., executing a SQL query with the DOMINATE clause (see Section 3). The specification of the attributes in PARTITIONING BY induces a partitioning of the input. Thus the checking of dominance conditions is limited to tuples within a partition. We note that this partitioning can be performed as described above. Therefore, we focus below on the processing necessary within each partition only.

As noted in Section 3.1, the dominance operator generalizes the Skyline operator. Despite these generalizations, the techniques for implementation of Skyline operator [5,23,26] can be leveraged for implementing the dominance operator. This is because the optimizations in the physical operators that implement Skyline only require a transitive dominance condition, which is preserved by our generalizations.

However, one important new requirement for the physical operator for dominance (not required for Skyline) is computation of dominance-based aggregate attributes (see Section 3.1), if used by the query.

We now mention an optimization that can be applied to a SQL query containing the DOMINATE clause. When the FROM clause references a view containing a foreign key join of two or more tables, it may be possible to push the dominance operator below a join operator, thereby potentially improving execution efficiency. A full exploration of the available transformations involving the dominance operator is an interesting area of future work.

[2] We observe that as proposed in [8,12], it may be interesting to expose partitioning as a separate operator by itself.

5.2 Implementing Representation

As described in Section 3.3, implementing a *representation operator* inside a SQL query engine to support the REPRESENT clause (Section 3.1) in its full generality requires the ability to implement solvers for the Integer Programming problem. Several such industry strength solvers e.g., [31] can provide exact or approximate answers to mathematical optimization problems. While incorporating such a solver into the SQL query engine may be possible and indeed useful for a class of applications, the resulting operator will be very expensive to execute, particularly on large data sets that are typical in today's databases.

Thus, in the rest of this section, we first discuss a less expensive physical operator that may sometimes be more suitable in the database context (Section 5.2.1). We then present two important special cases of the general problem for which there are efficient implementations with *guaranteed* quality (Sections 5.2.2 and 5.2.3 respectively).

5.2.1 User-Guided Search

We design this solution so that: (a) it is efficient and (b) application developers can exercise control, if necessary, over the heuristic for performing the search. We achieve (a) by using a simple greedy heuristic that examines one tuple at a time in a single pass over the input. To achieve (b), we extend the syntax of the REPRESENT clause with an optional RANKING BY <*Expression-List* >. The full syntax of the REPRESENT clause is shown below for completeness:

REPRESENT WITH (PARTITIONING BY <*attr-list*>)
 [MAXIMIZING | MINIMIZING] <*aggr-expr*>
  (GLOBAL CONSTRAINT <*global-constraint*>)*
  (FILTER CONSTRAINT <*filter-constraint*>)*
  (LOCAL CONSTRAINT <*local-constraint*>)*
 (RANKING BY <*Expression-List*>)

When the RANKING BY clause is specified, <*aggr-expr*> is limited to being COUNT(*). The RANKING BY specifies the order in which the input tuples should be accessed. We scan the input tuples in the order specified by the <*Expression-List*> in RANKING BY. For example, RANKING BY (A+B) DESC means that tuples must be considered for inclusion in the output in *descending* order of the expression (A+B) evaluated on each tuple. When the RANKING BY clause is not specified, the implementation for the general case of IP is invoked.

Despite the restriction of only allowing <*aggr-expr*> to be COUNT(*), it may be possible (for the application developer) to map a query that requires maximizing/minimizing a SUM(*Attr*) aggregate to a query that only maximizes/minimizes COUNT(*) but using RANKING BY. Of course, while the two queries are not equivalent, the quality/performance trade-off may be acceptable for the application. For example, consider maximizing SUM(*Attr*) subject to Count(*) $\leq k$. This could be mapped e.g., to a maximizing COUNT(*) query with constraint Count(*) $\leq k$ and RANKING BY *Attr* DESC.

The semantics of a query with RANKING BY can be procedurally described as follows. We first describe it for the MINIMIZING case. Observe that before the input is scanned, all $\leq$ constraints are trivially satisfied. If addition of the next tuple would violate any $\leq$ constraint, then the tuple is discarded. Otherwise, the tuple is added to the output. We terminate as soon as all $\geq$ constraints are satisfied or we reach the end of the input. A final check is performed to see if all $\geq$ constraints are satisfied, and if not, we report that we were unable to find a feasible solution. The procedure for the MAXIMIZING case is identical except that the termination condition is only when the end of the input is reached. Finally, we note that there are known algorithms for Top-K query processing e.g., [16,17], and it may be possible to leverage these algorithms to obtain an efficient implementation.

Observe that RANKING BY provides a particular way to specify how the general mathematical optimization problem should be solved. Another such approach is uniform random sampling or stratified sampling. Sampling is already of interest to the database community at large as evidenced by their incorporation into SQL standards [30]. Although we do not get into the details of syntactic extensions, we observe that it is possible to expose stratified sampling using our approach, e.g., by combining PARTITIONING BY with a local equality constraint on Count(*). Likewise uniform random sampling can be exposed using a single global equality constraint. Note that in these cases we are not maximizing/minimizing an aggregate, but rather finding a random subset with the specified count.

5.2.2 Maximizing SUM(Attr)

We now describe an operator for efficiently finding an optimal solution for a special class of queries. Consider a query MAXIMIZING an aggregate over some attribute, say *A*, subject to arbitrary filter constraints, together with a local constraint on Count(*), and a global constraint on Count(*). Intuitively, such queries require selecting some set of tuples to maximize an objective, while being constrained by the total number of tuples to be selected, as well as having constraints on how these chosen tuples are distributed across the different partitions. There are many interesting queries that fall into this class, including four of the five examples queries in Section 3.2 as well as all of the queries we use in our evaluation (see Section 6).

Providing an efficient operator to solve this special case exactly (i.e., obtain an optimal solution) is relatively straightforward. When both the global and local constraints are of the form Count(*) $\geq c_1$, the solution is trivially the entire input. When the global constraint is

Count(*) $\geq c_1$ and the local constraint is Count(*) $\leq c_2$, we ignore the global constraint and within each partition add tuples to the output in descending order of attribute *A* until we cannot add any more tuples without violating the constraint. Note that we can take advantage of the group-wise processing operator (as discussed earlier) to repeatedly execute this operation within each partition. Finally, we check if the global constraint is satisfied or not and terminate. When the global constraint is Count(*) $\leq c_1$ and the local constraint is Count(*) $\geq c_2$, we minimally satisfy each local constraint separately, i.e., pick exactly c_2 tuples from each partitioning in descending order of *A*. If we have already violated the global constraint, then no solution exists. Otherwise, we now pick the remaining tuples from the input in descending order of *A* and add to the output as long as the global constraint is not violated. The final case is when the global constraint is Count(*) $\leq c_1$ and the local constraint is Count(*) $\leq c_2$ (c_2 must be $\leq c_1$). In this case, we access the input in descending order of *A*, and keep adding a tuple as long as it does not violate the local constraint of a partition. We stop when we have added c_1 tuples to the output (or reach the end of input).

5.2.3 Minimizing Count(*)

The second class of queries for which optimal solutions or solutions with guarantees can be implemented efficiently is when we want to MINIMIZE Count(*).

Exact Solution: In the case when there are arbitrary filter conditions, and at most one other constraint, either global or local, we can obtain an exact solution. The input tuples are scanned in decreasing order of the attribute involved in the constraint, and are added to the output until the constraint is satisfied. As in 5.2.2, if the constraint is a local constraint, we use the groupwise operator to execute this operation within each partition. Finally, when there is a global *and* local constraint on the same attribute and in the same direction, we can similarly get an optimal solution.

Approximate Solution: When there are multiple (say *c*) global/local constraints, all of which are "≥" constraints, we can use the same idea but in a multi-pass fashion to an approximate solution. The constraints are are satisfied one by one. In the ith pass, tuples are scanned in descending order of the attribute in the ith constraint, and added to the output until that constraint is satisfied. Proceeding in this fashion until all the constraints are satisfied leads to a solution with an approximation ratio of *c,* where *c* is the total number of constraints. An optimization that can lead to a better approximation ratio in practice is to perform the i*th* pass only over tuples that are not already in the output (and adjust the constraints to take into account the contribution from tuples that are already part of the output). Finally, it may be possible to get a better approximation ratio by ordering the constraints in an intelligent manner.

6. Evaluation

We have implemented a prototype application that supports the dominance and representation primitives described in this paper. Below, we present a preliminary evaluation that shows the expressiveness and utility of these primitives. In particular, we evaluate the quality of workload summarization for the task of index selection. Index selection is an extremely computation-intensive task and the scalability of index-selection tools [3,27] depends on the number of queries in the workload. The simplest way to produce a smaller-sized workload to use as input for index tuning is to use naïve random sampling. Reference [11] introduced the idea of workload compression and showed that the use of workload compression to produce a smaller-sized workload was a considerable improvement over naïve random sampling. We use workload-summarization queries to generate a small workload as input for index tuning. We show that the performance obtained using our workload summarization is comparable to that of workload compression, while the process of obtaining the summarized workload is itself considerably faster than performing workload compression. We omit comparison with random sampling here since [11] has already established the poor performance of random sampling.

Methodology: We show results of our experiments for four different workloads: **SPJ** (select-project-join queries), **SPJ-GB** (select-project-join queries with GROUP BY), **SPJ-GB-OB** (select-project-join queries with GROUP BY and ORDER BY), and **Single table** (single-table queries only). These workloads execute against the TPC-H 1GB database, and are generated using a query-generation program that is capable of varying a number of parameters such as the number of joins, group-by and order-by columns, selection conditions and their selectivity etc. The number of queries in these workloads varies from about 1000 to 2000 (see Figure 6 for exact counts). For the sake of ensuring a fair comparison, we restrict our comparison to summarization queries that are constrained to generate exactly the same number of output queries as produced by the workload-compression algorithm. For each workload, we tune the physical database design separately using the summarized workload obtained (a) by our summarization queries, and (b) by Workload Compression. We use the Index Tuning Wizard [4] that is part of Microsoft SQL Server 2000 to perform physical-design tuning. We measure the *quality* of summarization by the optimizer-estimated cost of the entire (i.e., original) workload on the tuned database.

Results: Figure 3 compares the quality of Workload Compression with three different WAL queries (lower

Estimated Cost is better). All our WAL queries use the dominance and partitioning conditions shown in Example 1 (Section 2). The WAL query which imposes a global constraint on the total number of statements, along with a local constraint requiring proportionate representation per partition by Count (denoted by Proportionate (Count) in the figure), appears to provide quality comparable to the workload-compression algorithm. The WAL queries that apply a local constraint requiring proportionate representation per partition by *EstimatedCost* or only return top queries by *Dom_Weight* (i.e., weights after applying dominance) appear to be somewhat inferior in quality. We also note that the execution of WAL queries is about three orders of magnitude faster than workload compression, which internally employs a clustering-based solution.

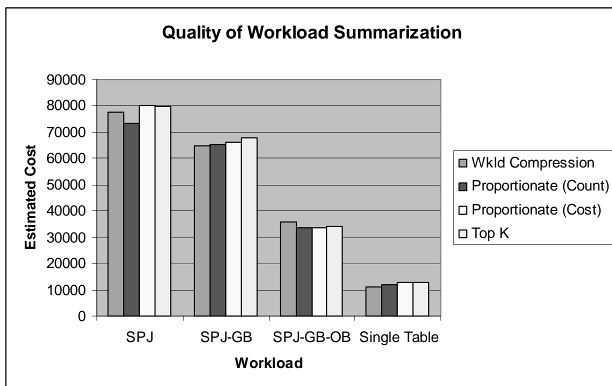

Figure 3. Quality of workload summarization for index selection.

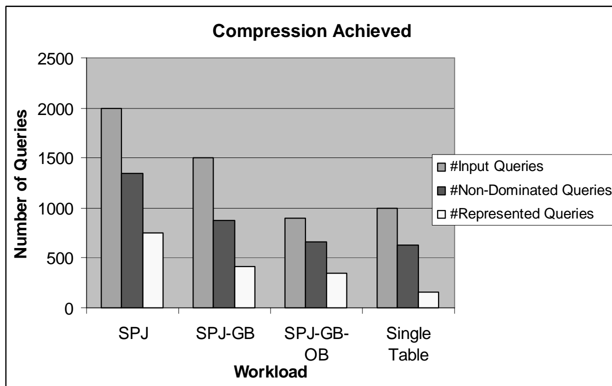

Figure 4. Importance of dominance and representation in workload summarization

Figure 4 shows the reduction in the number of statements in the workload achieved by dominance and representation respectively. We see from the figure that that both these concepts are critical for pruning out statements. We would expect the savings from dominance and representation to be even higher when the input contains even larger workloads.

7. Related Work

Recently, several tools have emerged that exploit knowledge of the database workload for a variety of tasks such as physical-design tuning [3,27], feedback-based optimization [1,25], and approximate query processing [2,9,10,18]. There has also been work [14,15] on classifying database workloads (e.g., OLTP vs. DSS), so as to enable automatic tuning and configuration of database-system parameters based on workload type. Typically, workloads collected by today's database profiling tools can be very large (millions of statements), whereas most of the above tools work efficiently for relatively small workload sizes. Thus, we view our work as back-end infrastructure to help improve the performance and scalability of such tools by intelligently pre-processing and summarizing the workload.

In [11], the authors present the idea of *workload compression* to find a smaller workload whose use results in the same application quality as when the original workload is used. Their work requires applications to specify custom distance functions that quantify how close two statements in the workload are. We note that providing such distance functions may not be easy for all applications. Our infrastructure, being declarative, is simpler to use for applications. The work in this paper can be viewed as a mechanism for quick pre-filtering, after which more sophisticated compression using much richer information, as in the above work, can be performed.

The group-wise processing techniques [8,12,21] are useful in implementing the primitives of dominance and representation. As discussed in Section 5, the SKYLINE operator [5] is a special case of the dominance operator proposed in this paper. Since both dominance and SKYLINE operators satisfy the transitivity property, we are able to leverage efficient execution strategies previously proposed for SKYLINE to also implement dominance. Also, as discussed in Section 5, the work on processing Top-K queries e.g., [16,17] can be potentially leveraged for implementing the representation operator.

In [29], the authors present a system called REDWAR (Relational Database Workload Analyzer). This system allows simple aggregations over the structure and complexity of SQL statements and transaction run-time behavior. While our WAL infrastructure supports such analysis, it also allows more sophisticated summarization through the dominance and representation primitives.

A survey of techniques for construction of statistical *workload models* for different kinds of systems (database, network-based, parallel etc.) is presented in [6]. The "representativeness" of such models is quantified. In our infrastructure, rather than automatically building models based on the workload, we allow applications to customize their workload summarization by using the primitives proposed in this paper.

The idea of workload analysis for studying the impact of physical design on workload cost and index usage was

presented in [13]. Our work complements this idea with new primitives that allow more sophisticated analysis and summarization of such workload information.

8. Conclusion

In this paper we have identified the primitives of *dominance* and *representation* that are crucial in various tasks that require summarizing workloads. These primitives also appear to be useful in many other practical scenarios. Tighter integration of these primitives into a traditional SQL query processing engine and their evaluation for a broader set of tasks is an interesting area of future work.

Acknowledgments

We thank Gautam Das for his valuable observations on the complexity of representation primitive, as well as the solutions for representation. We also thank Nicolas Bruno, Venky Ganti and Christian Konig for their insightful comments on the paper. Finally, we thank the anonymous reviewers of this paper for their important feedback.

References

[1] Aboulnaga, A. and Chaudhuri, S. *Self-Tuning Histograms: Building Histograms Without Looking at Data.* Proceedings of ACM SIGMOD, Philadelphia, 1999.

[2] Acharya S., Gibbons P.B., and Poosala V. *Congressional Samples for Approximate Answering of Aggregate Queries.* Proceedings of ACM SIGMOD, 2000.

[3] Agrawal, S., Chaudhuri, S., and Narasayya, V. *Automated Selection of Materialized Views and Indexes for SQL Databases.* Proceedings of VLDB 2000.

[4] Agrawal S., Chaudhuri S., Kollar L., and Narasayya V. *Index Tuning Wizard for Microsoft SQL Server 2000. White paper*. http://msdn.microsoft.com/library/techart/itwforsql.htm

[5] Borzsonyi S, Stocker K., Kossmann D. *The Skyline operator*. Proceedings of ICDE 2001.

[6] Calzarossa M., and Serazzi G. *Workload Characterization: A Survey*. Proceedings of IEEE, 81(8), Aug 1993.

[7] Chandra, A. K., Hirschberg, D. S., and Wong, C. K. *Approximate algorithms for some generalized knapsack problems*, Theoretical Computer. Science. **3**, 293-304, 1976.

[8] Chatziantoniou D. and Ross, K.A. *Groupwise Processing of Relational Queries*. Proceedings of VLDB 1997.

[9] Chaudhuri S., Das G., Datar M., Motwani R., and Narasayya V. *Overcoming Limitations of Sampling for Aggregation Queries*. Proceedings of ICDE 2001.

[10] Chaudhuri S., Das G., and Narasayya V. *A Robust Optimization Based Approach for Approximate Answering of Aggregate Queries.* Proceedings of ACM SIGMOD 2001.

[11] Chaudhuri S., Gupta A, and Narasayya V. *Workload Compression*. Proceedings of ACM SIGMOD 2002.

[12] Chaudhuri S., Kaushik R., and Naughton J.F. *On Relational Support for XML Publishing: Beyond Sorting and Tagging*. Proceedings of ACM SIGMOD 2003.

[13] Chaudhuri, S., and Narasayya, V. *AutoAdmin What-If Index Analysis Utility*. Proceedings of SIGMOD 1998.

[14] Elnaffar S. *A Methodology for Auto-Recognizing DBMS Workloads*. Proceedings of CASCON'02.

[15] Elnaffar S., Martin P., and Horman R. *Automatically Classifying Database Workloads.* In Proceedings of CIKM'02.

[16] Fagin R. *Combining fuzzy information from multiple systems*. Proceedings of ACM PODS 1996.

[17] Fagin R. *Fuzzy queries in multimedia database systems.* Proceedings of ACM PODS 1998.

[18] Ganti V., Lee M.L., and Ramakrishnan R. *ICICLES: Self-tuning Samples for Approximate Query Processing*. Proceedings of VLDB 2000.

[19] Garey M.R., and Johnson D.S. *Computers and Intractability. A Guide to the Theory of NP-Completeness*. W.H. Freeman and Company, New York, 1979.

[20] Johnson, D.S. *Approximation Algorithms for Combinatorial Problems*, Journal of Computer and System Sciences, 9 (1974),pp. 256—278.

[21] Legaria, C.G., and Joshi, M.M. *Orthogonal optimization of subqueries and aggregation*. Proceedings of SIGMOD 2001.

[22] Lovasz L. *On the Ratio of Optimal Integral and Fractional Covers*. Discrete Mathematics, 13(1975), pp. 383-390.

[23] Papadias D., Tao Y., Fu G., Seeger B.: *An Optimal and Progressive Algorithm for Skyline Queries*. Proceedings of ACM SIGMOD 2003.

[24] Srinivasan, A. *Improved approximations of packing and covering problems*, Proc. 27th Ann. ACM Symposium. on Theory of Comp., pp 268-276, 1995.

[25] Stillger M., Lohman G., and Markl V. *LEO: DB2's Learning Optimizer*. In Proceedings of VLDB 2001.

[26] Tan K., Eng P., Ooi B.C.: *Efficient Progressive Skyline Computation*. Proceedings of VLDB 2001.

[27] Valentin, G., Zuliani, M., Zilio, D., and Lohman, G. *DB2 Advisor: An Optimizer That is Smart Enough to Recommend Its Own Indexes.* Proceedings of ICDE 2000.

[28] V. V. Vazirani. *Approximation Algorithms*. Springer-Verlag, Berlin, 2001.

[29] Yu P.,Chen M., Heiss H., and Lee S. *On Workload Characterization of Relational Database Environments*. IEEE Transactions on Software Engineering, Vol 18, No. 4, April '92.

[30] http://www.wiscorp.com/SQLStandards.html

[31] http://www.ilog.com/products/cplex