Compressing SQL Workloads

Surajit Chaudhuri
Microsoft Research
One Microsoft Way
Redmond WA 98052
+1(425) 703-1938
surajitc@microsoft.com

Ashish Kumar Gupta*
University of Washington, Seattle
114, Sieg Hall, Box 352350
Seattle, WA 98125
+1(206) 616-1853
akgupta@cs.washington.edu

Vivek Narasayya
Microsoft Research
One Microsoft Way
Redmond WA 98052
+1(425) 703-2616
viveknar@microsoft.com

ABSTRACT

Recently several important relational database tasks such as index selection, histogram tuning, approximate query processing, and statistics selection have recognized the importance of leveraging workloads. Often these tasks are presented with large workloads, i.e., a set of SQL DML statements, as input. A key factor affecting the scalability of such tasks is the size of the workload. In this paper, we present the novel problem of *workload compression* which helps improve the scalability of such tasks. We present a principled solution to this challenging problem. Our solution is broadly applicable to a variety of workload-driven tasks, while allowing for incorporation of task specific knowledge. We have implemented this solution and our experiments illustrate its effectiveness in the context of two workload-driven tasks: index selection and approximate query processing.

1. INTRODUCTION

Information on how a database system is used can be important in performance tuning and management of the system. In the context of relational databases, one specific form of usage information is the *workload*, which is typically a set of SQL statements. Over the past few years, database practitioners and vendors have recognized the opportunity to tune and manage various aspects of database systems by analyzing workload information. Several workload-driven tasks have emerged recently for solving problems such as histogram tuning [1,7,15], improving query optimization [23], index selection [13,21], approximate answering of aggregation queries [2,9,10,17], and statistics selection [14]. We use the term *application* in this paper to generically refer to such workload-driven tasks.

A key factor affecting the scalability of these applications is the *size* of the workload, i.e., the number of SQL statements in the workload. In many cases, the workload consumed by the application is gathered using mechanisms in modern DBMSs that allow recording of SQL statements that execute on the server. In order to capture a representative collection of statements that execute against the system, the user of the application, such as a database administrator – could collect as the workload a log of SQL statements over a sufficiently large window of time (e.g., a day or week). Consequently, workloads tend to be large in size. Moreover, these applications often perform detailed analysis of queries in the workload and their inter-relationships, and hence their running time can be affected significantly as the workload size increases.

It is therefore natural to ask whether these applications can be sped up significantly by finding a substitute workload of smaller size (which we refer to as the *compressed* workload) as input, while qualitatively not degrading the result of the application. In other words, the result of the application when run on the compressed workload should be identical (or close) to the result when it is run on the original workload. It is crucial that this compressed workload be found efficiently, since otherwise the very purpose of the exercise is defeated. In this paper, we formalize the above question as the novel problem of *workload compression*. To the best of our knowledge, this is the first paper to study the problem of workload compression in the context of SQL workloads. We believe that this problem, and the solution we present in this paper can be applied to a variety of applications on relational databases.

An obvious solution to the workload compression problem is to use uniform random sampling to pick a smaller subset of the original workload. While this strategy is efficient, we have observed that it is not an effective method for workload compression. The key reason for the poor compression achieved by uniform random sampling is that it is oblivious to the application for which the workload is being used, and hence ignores potentially valuable information about the statements in the workload. Thus, we seek a solution that can exploit application knowledge to obtain significantly better workload compression as compared to uniform random sampling.

Our approach for incorporating application knowledge into the solution is via the use of a *Distance* function that quantitatively measures the "difference" between pairs of SQL statements with respect to the application. In our framework, we allow the Distance function to take on arbitrary values – e.g., unlike Euclidean distances, we do not impose restrictions that the function be symmetric or satisfy the triangle inequality. We have found this generality to be crucial in the two applications that we discuss in this paper.

Workload compression is a computationally difficult combinatorial optimization problem. In fact, we show via a reduction from the well known Minimum k-Median problem [16]

* Work done while author was visiting Microsoft Research

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, or republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.
ACM SIGMOD 2002, June 4-6, 2002, Madison, Wisconsin, USA.
Copyright 2002 ACM 1-58113-497-5/02/06...$5.00.

that the workload compression problem is NP-Hard. In this paper, we present two algorithms for solving workload compression. Our first algorithm adapts a well-known and efficient solution to the Minimum k-Median problem. We also present an alternative greedy algorithm which we have found gives better compression in the two applications to which we have applied and compared these algorithms. Understanding the relative strengths of these two algorithms, and characterizing their effectiveness for different kinds of SQL workloads is an interesting issue that requires further study, and is part of our ongoing work. An important characteristic of both these algorithms is that they are application independent, and can therefore be used for a variety of applications by specifying an appropriate Distance function. We show the generality of our framework by applying and demonstrating its effectiveness for two different applications: *index selection*, and *approximate answering of aggregation queries*. We have designed and implemented Distance functions for these two applications and experimentally evaluated our solution in the context of (a) two commercial index selection tools: Index Tuning Wizard for Microsoft SQL Server [13] and the Index Advisor for IBM DB2 [21], (b) as well as a tool for approximate answering of aggregation queries [10]. These results show the superior quality of our approach compared to sampling. Moreover, we observe significant improvement in the scalability of both these applications due to workload compression.

This work was done in the context of the **AutoAdmin** [6] project at Microsoft Research. The goal of this project is to automate the challenging task of tuning a database system by exploiting information about the workload faced by the system. The rest of the paper is organized as follows. In Section 2, we formally define the workload compression problem and analyze its complexity. We present the framework of our solution in Section 3, and the search algorithms for solving the optimization problem in Section 4. In Section 5, we describe the application specific part of our solution, namely the *Distance* function, for both the above applications. Section 6 describes how we determine the relative importance of statements in the compressed workload. Section 7 contains a thorough experimental evaluation of our solution. We review related work in Section 8.

2. PROBLEM STATEMENT

In this section, we provide a formal statement of the workload compression problem and analyze its complexity. We begin with a general version of the workload compression problem.

2.1 General Workload Compression Problem

For the purposes of this paper, we define a workload as a set of SQL DML statements. Thus **W** = $\{q_1, \ldots, q_i, \ldots\}$ where q_i is a SQL DML statement (i.e., SELECT, UPDATE, INSERT, DELETE). We associate a weight w_i (a real number) with statement q_i. We present workload compression as a technique for improving the scalability of an application A that consumes a workload **W** as input and produces a result **R** (Figure 1). Instead of passing **W** as input to the application, our objective is to first perform workload compression on **W** to obtain a compressed workload **W'**, and then pass **W'** as input to the application, thereby obtaining result **R'**. Such use of workload compression is meaningful only if the following two criteria hold:

Efficiency criterion: The total running time, i.e., time taken for workload compression *plus* the running time of the application, is less than the running time of the application on the original workload. This condition imposes the requirements that the workload compression algorithm itself is efficient *and* that it finds a **W'** such that the running time of the application on **W'** is less than the running time of the application on **W**.

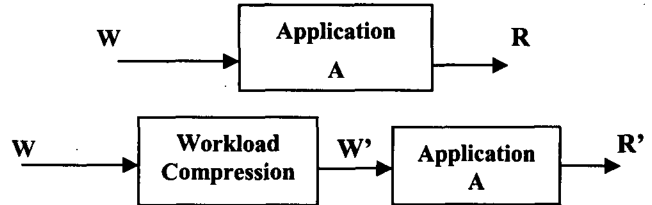

Figure 1. Workload compression

Quality criterion: Informally, this condition requires that the quality of the result **R'** is "close enough" to the quality of the result **R**. More formally, let A be an application and F_A be a function that quantitatively evaluates the result of the application with respect to the given workload **W**, i.e., $F_A(W,R)$ returns a real number that measures the quality of result **R**. Then this condition ensures that values $F_A(W,R)$ and $F_A(W,R')$ are close enough. Note that the exact definition of the function F_A is application dependent.

The generalized workload compression problem can be stated as:

> **Problem GEN-WCOMP:** Let A be an application that takes as input a workload **W** and produces a result **R**. Let $F_A(W,R)$ be a function that quantitatively evaluates **R** with respect to **W**. Given δ, $0 < \delta < 1$, find the workload **W'** that minimizes the total running time of application A (including time for finding **W'**), subject to the quality constraint:
> $|F_A(W,R) - F_A(W,R')|/|F_A(W,R)| < \delta$, where **R'** is the result produced by running A on input **W'**.

We note that in the above formulation, **W'** need not be a subset of **W**, i.e., **W'** may contain statements not present in **W**. We now illustrate how workload compression can be applied in the context of two applications that consume a workload as input. For each example, we specify the result **R** of the application, and the evaluation function **F**.

Example 1*: Workload Compression for Index Selection*
Selecting the right set of indexes is crucial for the performance of a database system. Automatically selecting appropriate indexes for a database is an important task since it reduces the burden on database administrators, and hence the total cost of managing the database. Recently, several major commercial database systems [13,21] have developed tools to automate this task. An index selection tool takes as input a workload **W** and a database, and produces as output **R** a set of indexes appropriate for the given workload. To evaluate the quality of the result **R**, these tools typically use as F_A (where A = index selection), the *query optimizer estimated execution time* of statements in **W** if the result **R** is implemented (i.e., if the set of indexes **R** is materialized in the database). Thus, e.g., specifying $\delta = 0.05$ for index selection implies that we are willing to accept a compressed workload **W'** such that the optimizer estimated execution time of statements in

W when **R'** is implemented cannot deviate by more than 5% compared to the optimizer estimated execution time if **R** had been implemented.

Example 2: *Workload Compression for Approximate Answering of Aggregation Queries*
The goal of *approximate query processing* (AQP) is to allow efficient but approximate answers to ad-hoc queries against large relational databases. Random sampling is an approach for approximately answering *aggregation* queries (e.g., queries computing SUM or COUNT aggregate expressions). In this approach, the query is executed on a sample of the data rather of the entire data, thereby returning approximate answers but speeding up the query significantly. Recently, several papers [2,9,10,17] have recognized the importance of using *workload* information to pick samples of the data and thereby improve upon the straightforward approach of *uniform* random sampling. Thus, the workload **W** is analyzed in a preprocessing step and this information is used to produce as result **R**, an appropriate set of samples of one or more tables in the database. These samples are chosen with the objective of minimizing the average relative error[1] in answering queries in **W**[2] over the sample. The preprocessing step described above can be expensive if the workload **W** is large, and is therefore a suitable application for workload compression. For example, in the stratified sampling approach presented in [10] and the weighted sampling technique presented in [9,17], this preprocessing step requires executing the queries in **W**. A commonly used definition of $\mathbf{F_A}$ is the average relative error over all queries in **W** when answered using the samples. Thus, specifying $\delta = 0.1$, for example, implies that we are willing to accept a compressed workload **W'** that results in a sample over which the average relative error of queries in **W** cannot deviate by more than 10% compared to the case when we choose the sample based on the original workload **W**. For the rest of this paper we refer to the above application as AQP for short.

2.2 Distance-Based Workload Compression Problem

While the problem GEN-WCOMP is general, it does not appear to be amenable to efficient solutions for two reasons. First, statements in the compressed workload **W'** need not be a subset of the statements in **W**. Thus, the space of possible statements that need to be considered during workload compression is potentially much larger. The second reason that makes it hard to solve GEN-WCOMP efficiently is that exact verification of the constraint on the loss of quality, $|\mathbf{F_A(W,R)} - \mathbf{F_A(W,R')}|/|\mathbf{F_A(W,R)}| < \boldsymbol{\delta}$, is expensive, since computing $\mathbf{F_A}$ requires running the application A itself.

In this paper, we therefore consider a simpler version of GEN-WCOMP called WCOMP that sacrifices some of the generality of GEN-WCOMP, but is more amenable to efficient solutions. In particular, WCOMP is obtained by applying the following two restrictions on GEN-WCOMP: (1) We require **W'** to be a subset of **W**. (2) The quality constraint is defined in terms of "distances" between pairs of statements in the workload. We therefore assume the availability of an application specific *Distance* function between any pair of statements in the workload. $\text{Distance}_A(q_i, q_j)$ estimates the loss in quality of the result of application A for statement q_i if q_i is discarded, but statement q_j is present in the compressed workload – independent of the other statements in the workload. More precisely, if R_i is the result of the application when the workload used is $\{q_i\}$ and R_j is the result of the application when the workload used is $\{q_j\}$, then $\forall\ q_i, q_j \in \mathbf{W}$ $\text{Distance}_A(q_i, q_j)$ estimates the quantity $F_A(\{q_i\}, R_j) - F_A(\{q_i\}, R_i)$. Problem WCOMP can be visualized as shown in Figure 2.

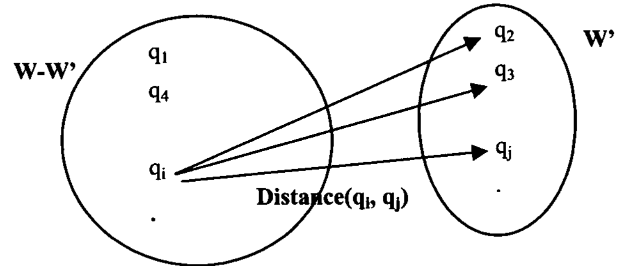

Figure 2. Visualizing WCOMP

W' is the compressed workload and **W-W'** is the set of statements in **W** that have been discarded by workload compression. For each statement $q_i \in$ **W-W'**, we can find the "closest" statement in **W'** as determined by the Distance function. WCOMP requires that the smallest **W'** must be chosen such that if we take a *weighted sum* of the distances between each discarded statement and the retained statement closest to it, that sum should not exceed a pre-specified value.

We now formulate the problem WCOMP, which is the focus of this paper:

> **Problem WCOMP:** Let A be an application that takes as input a workload **W**. Let $\text{Distance}_A(q_i, q_j)$ be a function for application A that $\forall\ q_i, q_j \in \mathbf{W}$, returns an estimate of the loss in quality for statement q_i if it is discarded but q_j is retained in the compressed workload. Given $\boldsymbol{\Delta}$, which is the limit on the maximum allowable loss in quality, find the smallest workload $\mathbf{W'} \subseteq \mathbf{W}$, such that $\Sigma_{q_i \in W - W'} \min_{q_j \in W'} \{ w_i \cdot \text{Distance}_A(q_i, q_j) \} < \boldsymbol{\Delta}$.

We make a few observations about WCOMP and the Distance function. First, observe that unlike in GEN-WCOMP where the constraint on quality $\boldsymbol{\delta}$ is relative, $\boldsymbol{\Delta}$ is specified in absolute terms and has the same units as the Distance function. Second, we note that variations of WCOMP are possible by replacing the *min* in the quality constraint with other functions such as *max* or *average*. For example, using *max* makes the worst-case assumption that loss in quality for q_i could be as large as the distance to the "furthest" statement from it in the compressed workload. Thus, if we use *max*, we potentially expect less degradation in quality of the application, but also less compression of the workload. Third, WCOMP makes no assumptions about properties of the Distance function. For example, it does not require that the Distance function be symmetric or obey the triangle inequality. In fact, the very definition of $\text{Distance}_A(q_i, q_j)$ is *asymmetric*, since it measures the distance with respect to $\{q_i\}$. We believe this generality is important since for the applications of workload compression we

[1] Relative error of an aggregation query Q is defined as: |Exact Answer(Q) – Approximate Answer(Q)|/ |Exact Answer(Q)|

[2] To be exact, the errors are optimized for a *distribution* of queries of which **W** is an instance.

present in this paper, we found that these properties were not satisfied (see Section 5). Fourth, as we will show in Section 2.3, the WCOMP problem is provably hard when the Distance function can return arbitrary values. Finally, we emphasize that the exact definition of Distance[3] (q_i, q_j) is application dependent since it must capture the impact on quality of the result produced by the application when q_i is discarded and statement q_j is retained in the compressed workload. Table 1 summarizes the distance-based workload compression problem WCOMP for the two applications of workload compression described in Section 2.1.

Table 1. Applying WCOMP to different applications

Application	Meaning of Distance(q_i, q_j)	Meaning of Δ
Index Selection	Estimated increase in the cost of executing statement q_i if it is discarded but q_j is retained	Maximum allowable increase in (estimated) running time of the workload **W**
Approximate Answering of Aggregation Queries	Increase in the relative error of answering query q_i, if q_i is discarded but q_j is retained	Maximum allowable increase in average relative error of queries in **W**

2.3 Hardness of WCOMP

The problem WCOMP, defined in Section 2.2, aims to minimize the size of the set **W'**, while satisfying the constraint $\Sigma_{q_i \in W-W'} \min_{q_j \in W'} \{ w_i \cdot \text{Distance}(q_i, q_j) \} < \Delta$. We now show that when the Distance function can generate arbitrary values, WCOMP is NP-Hard. We will use a reduction from the decision version of the Minimum k-Median problem which is known to be NP-Complete [16]. First, we define the decision problem of the Minimum k-Median problem:

> **Problem Minimum k-Median:** Given a complete graph G (V, E), costs C(u,v) $\in$ **N** (the set of natural numbers), $\forall$ u, v $\in$ V, an integer **k**, and a number **s**. Does there exist a set of medians V'$\subseteq$ V of size **k** such that the sum of the distances from each vertex to its nearest median is less than **s**, i.e., $\Sigma_{u \in V-V'} \min_{v \in V'} \{ C(u,v) \} < s$? ♦

Lemma 1. Problem WCOMP (defined in Section 2.2) is NP-Hard if the Distance function can return arbitrary values.

Proof: The decision problem for WCOMP is as follows: Let A be an application that takes as input a workload **W**. Let Distance (q_i, q_j) be a function that quantifies the distance between any pair of statements q_i, $q_j \in$ **W**. Given an integer **k**, and a number Δ, does there exist a workload **W'** $\subseteq$ **W** of size **k** such that $\Sigma_{q_i \in W-W'} \min_{q_j \in W'} (\text{Distance}(q_i, q_j)^4) < \Delta$.? There is a direct correspondence of the two problems as follows: **V** $\Leftrightarrow$ **W**, **V'** $\Leftrightarrow$ **W'**, **k** $\Leftrightarrow$ **k**, Cost $\Leftrightarrow$ Distance, and **s** $\Leftrightarrow$ Δ. Hence, the decision problem of WCOMP is NP-Complete. Therefore WCOMP is NP-Hard. ♦

Although for metric spaces there exist constant factor approximation algorithms [4,12] for the Minimum k-Median problem, Lin & Vitter [22] showed that the Minimum k-Median problem with arbitrary costs *does not* have a constant factor approximation algorithm.

3. ARCHITECTURE OF SOLUTION

In this section, we outline the architecture of the solution we have implemented for the WCOMP problem presented in Section 2.

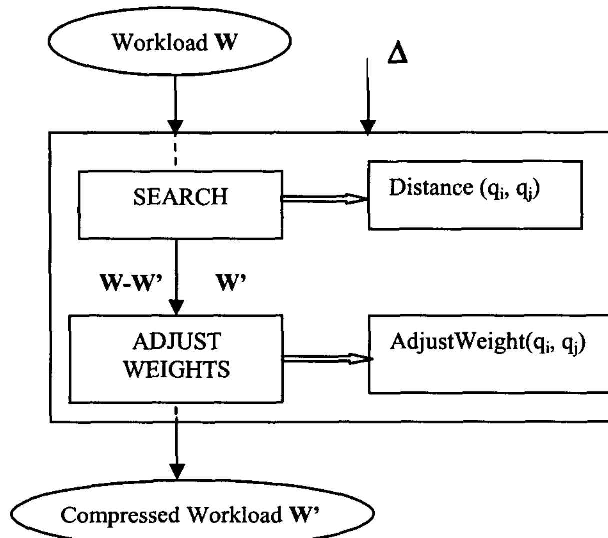

Figure 3. Architecture of solution for WCOMP

An overview of the architecture is shown in Figure 3. We take as input a workload **W** and a constraint Δ, and produce as output a compressed workload **W'**. A key part of our architecture is the *Search* module (described in Section 4) that finds the compressed workload **W'**. WCOMP requires us to find the smallest (cardinality) subset of the given workload **W** that satisfies the given constraint. For this optimization problem, we consider two algorithms (besides random sampling). As described earlier, our search module is designed such that the algorithms consult a Distance function (described in Section 5), but make no assumptions about *properties* of the Distance function – in other words the Distance function can return arbitrary values. The Distance function serves as the basis for estimating the loss in quality due to workload compression and is *application specific.* It must be designed carefully since the quality of the compressed workload depends critically on the accuracy and efficiency of the Distance function. Efficiency in computing the Distance function is crucial since the function can be invoked many times for a large workload by the Search component. The accuracy of the Distance function is also important since overestimation of the loss in quality achieves less compression of the workload than ideally possible, whereas underestimation can cause an unacceptable result when the compressed workload is used. In Section 5, we illustrate how we make the above trade-off of efficiency vs. accuracy in designing the Distance functions for each of the two applications: index selection and AQP.

[3] From this point onwards, for notational convenience, we assume that the subscript A in Distance (q_i, q_j) is implicit.

[4] For this reduction, we assume a workload where the weight of each statement is 1.

Recall from Section 2.1 that we associate a weight w_i with each statement q_i in the workload, which reflects the relative importance of that statement. The weight of a statement can significantly influence the result of the application. For example in index selection, higher the weight of a statement, the more likely it is that the indexes that are suitable for that statement are part of the final result. When a statement is discarded by workload compression, the obvious solution is to add the weight of the discarded statement to the "closest" statement (as defined by the Distance function) in the compressed workload. However, as illustrated by the following example for index selection, simply adding the weight can be inappropriate.

Example 3: *Problem with simple addition of weights*
Consider a workload **W** with the following queries:
Q_1: SELECT * FROM persons WHERE *age* < 10.
Q_2: SELECT * FROM persons WHERE *age* < 20.
Q_3: SELECT * FROM persons WHERE income < 20000
Assume the weights of these queries in **W** are all 1. Suppose the compressed workload is $\{Q_2, Q_3\}$. Using the obvious solution, since Q_2 is the closest retained query to Q_1, the adjusted weights of these queries is 2 and 1 respectively. However, from the queries, it is clear that the presence of an index on column *age* would result in more benefit for Q_1 as compared to Q_2. Thus, the compressed workload has been biased *against* selecting an index on column *age*. ♦

Therefore, in our architecture, we include a post-processing step called *Adjust Weights* (described in Section 6) that uses the application specific AdjustWeight (q_i, q_j) function to adjust the weight of each statement in the compressed workload. Note that for certain applications, the obvious solution of simply adding weights to the nearest retained statement may be appropriate. Finally, an interesting issue worth investigating is whether adjusting of weights during the search step itself can lead to better workload compression.

We believe that the architecture described in this section is general enough to handle workload compression for a broad class of applications beyond those discussed in this paper.

4. SEARCH STRATEGY

As described in Section 3, the search component is responsible for finding a subset of **W** of smallest cardinality satisfying the constraint that the loss is quality is less than the given **Δ**. In this section, we then present and compare two search algorithms (besides random sampling) for solving WCOMP. The first algorithm is based on the K-Mediod clustering algorithm and the second is a new greedy algorithm. In Section 7, we present a detailed experimental comparison of the algorithms presented in this section.

Based on the hardness result in Section 2.3, we do not expect a polynomial time algorithm that computes an optimal solution to WCOMP when the Distance function can return arbitrary values. We have therefore designed our solutions to WCOMP to leverage well-known and efficient heuristic search algorithms. We note however, that for specific Distance functions, the problem WCOMP may be solvable in polynomial time, and alternative search algorithms customized for that application may be appropriate.

4.1 K-Mediod Algorithm

The Minimum k-Median problem referred to in Section 2.3, is in fact a *clustering* problem. Our first algorithm therefore adapts the well known K-Mediod clustering algorithm [19]. We use the K-Mediod algorithm as a building block for constructing an algorithm for WCOMP by performing binary search on the size of the workload **W**. The pseudo code for our overall search algorithm WC-KMED, and the modified K-Mediod algorithm KMED are presented in Figure 4 and Figure 5 respectively.

Input: Workload **W**, Constraint **Δ**
Output: Compressed workload **W'**
1. Let Min_k = 0, Max_k = |**W**|, **W'** = **W**
2. **While** (Min_k   Max_k)
3. k = (Min_k + Max_k)/2
4. Let WTemp = **KMED(W**, k)
5. Let D be the weighted sum of distances from each statement in **W** to the closest statement in WTemp as determined by the Distance function.
6. **If** D   **Δ**, **W'** = WTemp, Max_k = k - 1
7. **Else** Min_k = k + 1
8. **End If**
9. **End While**
10. Return **W'**

Figure 4. Algorithm WC-KMED

Input: Workload **W**, **k**
Output: Workload **W'** of size **k**
1. Pick **k** statements $s_1..s_k$ from W at random. Each statement chosen forms the "seed" of a cluster.
2. For each statement e ∈ W, assign it to the cluster that contains the seed closest to e, as determined by the Distance function.
3. For each cluster C, re-compute the seed for that cluster as the "median" statement within the cluster, i.e., the statement e ∈ C such that $\Sigma_{v \in C}$ Distance (v, e) is the smallest.
4. Repeat steps 2-3 until convergence, i.e., until the same clusters are obtained in some two iterations.

Figure 5. Algorithm KMED

We mention a few important properties of algorithm KMED (Figure 5). First, it can be shown that the algorithm will terminate in a finite number of iterations of Steps 2-3. Second, the solution obtained by this algorithm is (at least) a local optimum. Third, the running time of KMED depends primarily on the number of invocations of the Distance function. As we can see from the pseudo code, KMED performs $O((|\mathbf{W}|-k)*k + k*(|\mathbf{W}|/k)^2)$ invocations of Distance function assuming each cluster on average contains the same number of statements. Note that since the algorithm WC-KMED (Figure 4) performs a binary search over the range 0..|**W**|, it invokes KMED at most $\log_2|\mathbf{W}|$ times. WC-KMED produces a solution that is a local optimum.

4.2 All-Pairs Greedy Algorithm

Unlike the WC-KMED algorithm that does not invoke the Distance function on every pair of statements in **W**, the WC-ALL-PAIRS algorithm *does* look at the Distance of each pair statements in the workload. Our goal in proposing this algorithm

was to investigate how the computing of all Distances would impact the quality of workload compression. Once the pair-wise distances are computed, the WC-ALL-PAIRS algorithm adopts a greedy approach that discards the next "best" statement from **W** until it is unable to discard any more statements without violating the given constraint **Δ**. The pseudocode is given in Figure 6.

Input: Workload **W**, Constraint **Δ**
Output: Compressed workload **W'**

1. Let the sets *Candidates* = **W**, *Keep* = { }, *Prune* = { }
2. Let Total-Dist = 0
3. **While** (Total-Dist < Δ)
4. For each statement $q_i \in$ *Candidates*, compute $d_i = \min_{(i \neq j,\ qj \in Candidates \cup Keep)} (w_i \cdot \text{Distance}(q_i, q_j))$
5. Let q_{min} be the statement with the minimum distance d_{min} computed in Step 4.
6. **If** (d_{min} + Total-Dist < Δ)
   Move q_{min} from *Candidates* to *Prune*
   Old-Dist = Total-Dist
   Total-Dist = $\Sigma_i \min_k (w_i \cdot \text{Distance}(q_i, q_k))$ where $q_i \in$ *Prune* and $q_k \in$ *Candidates* ∪ *Keep*
   If (Total-Dist > Δ)
   Move q_{min} from *Prune* to *Keep*,
   Total-Dist = Old-Dist
   EndIf
7. **Else Break**
8. **EndIf**
9. **End While**
10. **Return** W' = *Candidates* ∪ *Keep*

Figure 6. Algorithm WC-ALL-PAIRS.

At each step the algorithm maintains three sets, *Keep*, *Prune* and *Candidates*. *Keep* consists of statements which are definitely going to be retained in the compressed workload **W'**. *Prune* consists of the statements which are currently not in **W'** and *Candidates* consists of the statements whose outcome hasn't yet been decided. In each iteration of the **While** loop in Step 3, for each statement in *Candidates*, we compute the distance to the closest statement (as defined by the Distance function) that hasn't been pruned (Step 4). The statement for which this value is the smallest (Step 5) is considered next for pruning. Prior to actually pruning this statement however, we verify that removal of this statement does not violate the constraint **Δ**, since this statement may have been the closest statement to one or more statements that had been pruned previously. This check is performed in Step 6. At the end of the algorithm, the statements in the sets *Candidates* and *Keep* constitute the compressed workload **W'**.

Algorithm WC-ALL-PAIRS performs O(|**W**|^2) computations of the Distance function since in the first execution of Step 4 all pair-wise invocations of Distance are performed. Thus, we expect WC-KMED to scale better with workload size compared to WC-ALL-PAIRS. Also, unlike WC-KMED, WC-ALL-PAIRS cannot guarantee that the solution obtained is a local optimum for the problem WCOMP.

While WC-KMED is based on a well-known algorithm for solving the k-Median problem, in our experiments (see Section 7) we found that WC-ALL-PAIRS often achieves more compression than WC-KMED for WCOMP. Intuitively, WC-ALL-PAIRS does better when the workload has many small clusters and the *intra*-cluster distances are small relative to *inter*-cluster distances. Analyzing and understanding the relative strengths of these algorithms is part of our ongoing work. We are exploring opportunities to combine these two algorithms to obtain even better compression, e.g., by running WC-KMED using the output of WC-ALL-PAIRS as the seed.

4.3 Random Sampling

An obvious technique for improving the scalability of an application that consumes a workload **W** is to use sampling to select a subset **W'** of **W**. The simplest of these schemes is *uniform* random sampling, where each statement in **W** has an equal probability of being selected. However, this approach can result in poor quality workload compression due to the following problems: (a) Uniform sampling ignores valuable information about statements in the workload and therefore misses opportunity for more compression. (b) When the sampling fraction is small, certain small "clusters" of important statements may be altogether discarded and never make it into the compressed workload. This follows from a well known statistical result [8]. In our experiments, we therefore considered a *stratified sampling* based algorithm (which we refer to as WC-PARTSAMP), which partitions the workload into strata and then samples uniformly within each stratum. The partitioning scheme used is described in Section 5. One issue with applying sampling is how much (i.e., what fraction of **W**) to sample? We start with a sampling fraction f_0 and verify if the constraint **Δ** is satisfied for that sample – note that this step requires invoking the Distance function. If not, we repeat the process by increasing the sampling fraction by a factor $m > 1$ and sampling an additional set of statements. We terminate when we find a sample that satisfies the constraint.

5. DISTANCE FUNCTION

As described in the previous section, a key component of our solution to WCOMP is the computation of Distance (q_i, q_j) for any pair of statements q_i, q_j in the given workload **W**. Recall that the function Distance (q_i, q_j) measures the expected loss in quality of the result of the application on workload {q_i} if the workload {q_j} is provided as input to the application. A judicious trade-off between accurate and efficient computation of the Distance function is crucial for ensuring the success of workload compression. *Accuracy* is important since overestimation of Distance (q_i, q_j) results in less workload compression than possible, while underestimation of Distance (q_i, q_j) can result in poor quality of the result of the application when the compressed workload is used. *Efficiency* of computing Distance is important since the search algorithms for workload compression (see Section 4) may invoke the Distance function many times for different pairs of statements.

An *exact* method for computing Distance (q_i, q_j) is: (i) run the application on workload {q_i} and compute the quality of the result for {q_i}, (ii) run the application on {q_j} and compute the quality of the result for {q_i} and (iii) take the difference in quality between Steps (i) and (ii). However, for most applications such a method is inefficient since it requires running the application, and hence negates the very purpose of workload compression. Thus, the challenge of developing an appropriate Distance function for an application is to *estimate* this loss in quality efficiently. In this section, we first present two guidelines that we followed in developing Distance functions for the two applications described

in Section 2.1: index selection and approximate answering of aggregation queries (AQP). We believe that these guidelines are more broadly applicable in the context of other applications as well. We then present the specific Distance functions we have developed for the above two applications.

Our first guideline is driven by the requirement that the computation of Distance function be efficient. We therefore identify a core set of information about each statement in the workload that *can be derived with low overhead* and rely only this information for computing Distance. For example, in our implementation of Distance function for both applications, we limit ourselves to information that can be derived from the SQL parser and a selectivity estimation module. This information includes: (a) Type of the statement, (SELECT, INSERT, UPDATE, DELETE) (b) Structure of the query, e.g., tables referenced, projection columns, selection and join predicates etc. (c) For selection predicates, the selectivity of the predicate (computed by using selectivity estimation module based on available statistics in the database) (d) If available, the (estimated) cost of executing the statement is available. This cost information can be obtained either via one invocation of the query optimizer (e.g., in Microsoft SQL Server using the Showplan interface, or in IDB DB2 using the EXPLAIN mode) or from previously recorded information about the actual execution time of the statement.

Our second guideline is to leverage the technique of logically *partitioning* the workload. The idea is that for any two queries q_i, q_j belonging to different partitions, Distance(q_i, q_j) between the two queries is ∞. Partitioning can be incorporated within the Distance (q_i, q_j) function by generating a "signature" for each statement and returning ∞ if the two signatures are not identical. Thus, each statement with a distinct signature belongs to a different logical partition. As a simple example, in the index selection application, when two statements reference disjoint sets of tables, it is reasonable to separate them into different partitions since indexes that are useful for one statement cannot be useful for the other. There are two benefits of partitioning the workload. First, it provides a way to ensure that at least a minimum number of statements (i.e., at least one statement per partition) will be retained in the compressed workload. Second, since the signature of a query can typically be computed very efficiently compared to the more careful analysis that goes into the rest of the Distance function, partitioning serves as a "shortcut" that reduces the computational overhead of invoking the Distance function. Finally, we note that the signature to be used to partition the workload is application dependent, and is therefore incorporated into the Distance function.

5.1 Distance Function for Index Selection

For simplicity of exposition, in this section we assume that the workload **W** consists of SELECT, INSERT, UPDATE, DELETE statements, where the SELECT statements are limited to single-block Select, Project, Join (SPJ) queries with Group-By, Aggregation and Order-By. We first present the Distance function for queries (i.e., SELECT statements) and briefly mention the extensions for handling updates. For index selection, the Distance (q_i, q_j) function measures the expected loss of benefit for $\{q_i\}$ if the set of indexes recommended for $\{q_j\}$ were used to answer it instead of the set of indexes recommended for $\{q_i\}$ itself. Our goal is to *estimate* this expected loss of benefit efficiently without actually invoking the index selection application. While more sophisticated Distance functions could be designed for index selection, we believe that our design captures essential aspects of index selection, without making assumptions about the specific algorithms used inside the index selection tool. This is backed by our experimental results (see Section 7) which show the effectiveness of our Distance function for index selection tools on two different commercial database systems.

5.1.1 Partitioning the Workload

Our first step in the Distance function is to detect if the two queries belong to same partition or not. If not, we return immediately from the Distance function with a value of ∞. As mentioned above, the intuition behind partitioning is to logically place queries that are "far apart" into disjoint partitions. In the context of index selection, two queries can be considered far apart, if there is little or no overlap in the set of indexes that would be chosen for each query. Based on this intuition we partition the workload on the basis of the tables accessed in each query and the join predicates (if any). This is done by generating a signature for each query that consists of the table IDs referenced in the query and (table, column) IDs accessed in the join predicate.

Example 4: *Motivating example for selectivity based partitioning*
Consider the following two queries:
Q_1: SELECT * from persons where age > 80
Q_2: SELECT * from persons where age > 1 ♦

However, as the above example shows, the simple scheme above may still include queries into the same partition that are still "far apart". According to the above scheme, both Q_1 and Q_2 will be assigned to the same partition because they both have the same signature. However, note that the queries are still far apart from the point of view of indexes that are appropriate for each query. Due to the respective selectivities of predicates on age, for Q_1, an index on column age is likely to be very useful, whereas for Q_2 an index on column age is likely to be of no use. Motivated by this observation, we further split each partition on the basis of selectivity information. For a single-table query, we compute the *joint* selectivity of all the predicates. All queries with joint selectivity less than or equal to a predetermined selectivity s_0 (we used a value of $s_0 = 0.1$) are assigned to one partition, and those with selectivity exceeding s_0 are assigned to a different partition. Thus, for single-table queries, we can generate at most two partitions. We adopt the straightforward extension of this partitioning scheme to the case of multi-table queries. Under this scheme, all queries belong to a *t*-table partition (i.e., a partition with queries accessing those *t* tables) get split into at most 2^t partitions (some of which may be empty). Although the number of such partitions can, in principle, become large, we found in practice that over a variety of large workloads (real and synthetic), the number of partitions grew very slowly with the number of tables.

5.1.2 Quantifying Distance

Our approach for computing the Distance function is based on the observation that the effectiveness of an index for a query can be broadly categorized into one or more of the following performance categories: (a) Reduces the number of rows that need to scanned from the table, (b) Eliminates the need to access the

table altogether since the index contains all columns required to answer the query (i.e., the index is "covering" for the query) or (c) Reduces/eliminates the cost of sorting for some operator in the query. Thus, when computing Distance (q_i, q_j) we analyze each query and classify the columns referenced in the query as: (1) Selection columns – contains all columns that occur in selection conditions. Indexes on these columns fall into performance category (a) above. (2) Required Columns – contains all columns that were referenced in any part of the query (including projection columns). Indexes on these columns fall into performance category (b) above. (3) Group-By Columns – contains all columns that occur in the GROUP BY clause of the query. (4) Order-By Columns – contains all columns that occur in the ORDER BY clause of the query. Indexes on columns in Category (3) and (4) fall into performance category (c) above.

We then compute four functions Dist-Sel (q_i, q_j), Dist-Reqd (q_i, q_j), Dist-GB (q_i, q_j) and Dist-OB (q_i, q_j). Each of these functions captures the loss of benefit for a particular performance category. For example, Dist-Sel computes a distance by examining only the columns in the Selection category of the queries q_i and q_j, and thereby tries to capture the difference in performance category (a). Once each of the functions is computed we define Distance (q_i, q_j) as the *maximum* of the four values. Intuitively, by considering the maximum value, we take the conservative approach of considering two queries as "close" only if they are "close" in *each* of these categories. We now briefly describe how each of the above four functions is computed, and omit details due to lack of space.

Computing Dist-Sel: Our approach is based on the intuition that the column (or sequence of columns) in the predicate with the lowest selectivity[5] (resp. joint selectivity) is the one that will be picked to be indexed by the index selection tool for that query. In other words, while considering 1-column indexes, we assume that the column with the smallest selectivity will be picked. On the other hand, when considering 2-column indexes, we assume that it is the sequence of two columns with the two lowest selectivities that will be picked. The following example illustrates how Dist-Sel(q_i, q_j), is computed.

	c_1	c_2	c_3
Q_1	0.1	0.3	0.2
Q_2	0.5	0.2	0.4

Example 5. Computing Dist-Sel: Suppose we have two queries Q_1 and Q_2 referencing a single table T, with predicates on columns c_1, c_2, c_3. The selectivities of these predicates are given by the adjoining table. The best 1-column index for Q_1 is $I_1 = (c_1)$. On the other hand, the best 1-column index for Q_2 is $I_2 = (c_2)$. The loss of benefit for Q_1 if it is pruned and Q_2 is retained, is given by difference of cost of evaluating Q_1 in presence of I_2 and cost of evaluating Q_1 in presence of I_1, which is given by (0.3 – 0.1) * Cost ({Q_1}, {}) = 0.2 * Cost ({Q_1}, {}). (Note that Cost ({Q_1}, {}) corresponds to a scan of the entire table i.e., no indexes are present). The intuition is that the presence of index I_2 would require scanning 30% of the base relation for answering Q_1, whereas the presence of index I_1 would require scanning only 10% of it. Examining 2-column indexes, we see that the best 2-column index for Q_2 is (c_2, c_3) and the best 2-column index for Q_1 is (c_1, c_3). Therefore, the loss of benefit is given by (0.3*0.2 – 0.1*0.2) * Cost ({Q_1}, {}) = 0.04* Cost ({Q_1}, {}). Similarly, for 3-column indexes, we see that the loss of benefit is 0. In general, this analysis can similarly be extended for up to **p**-column indexes. We take Dist-Sel (Q_1, Q_2) as the maximum of the numbers computed – in our example, 0.2*Cost ({Q_1}, {}).♦

Note that in case of multi-table queries, we perform the same analysis as in the above example on a per table basis and then take a weighted average of the table-wise Dist-Sel (q_i, q_j) values, the weight being the size of the table in pages. We use size of the table as weight because for same selectivity value, the amount of I/O required to answer the query is proportional to the size of the table.

Computing Dist-Reqd: Dist-Reqd(q_i, q_j) tries to capture the loss of benefit for performance category (b) , i.e., use of covering indexes. We present the intuition behind our scheme for single-table queries. The extension for the multi-table case is similar to the extension for Dist-Sel. Intuitively, if q_i is pruned away, and the required columns of q_i are a *subset* of the required columns of q_j, then the covering index for q_j can be used to answer q_i and hence Dist-Reqd (q_i, q_j) is relatively small. However, if the required columns of q_i are *not* a subset of the required columns of q_j, then the covering index chosen for q_j will not be useful for answering q_i Pruning away q_i in this case requires scanning the entire table for answering q_i, and therefore the loss of benefit (i.e., Dist-Reqd) is large.

Computing Dist-GB & Dist-OB: We first discuss Dist-OB (q_i, q_j). We assume the model of no partial sort benefits, i.e., an index (c_1, c_2, c_3) is of no use in answering a query with ORDER BY c_1, c_3, c_2, even though there is some overlap in the prefix of the index and the prefix of the order-by clause. Under this simplifying assumption, Dist-OB (q_i, q_j) is 0 if the ORDER BY clause of q_i is a leading prefix of the ORDER BY clause of q_j. Otherwise Dist-OB (q_i, q_j) is equal to the Cost ({q_i}, {}). Computation of Dist-GB (q_i, q_j) is done similarly, except that we require that the group-by columns of q_i to be a subset (rather than a leading prefix) of the group-by columns of q_j in order for the index chosen for q_j to be applicable to q_i.

Extensions for updates: If both statements are "pure" updates, e.g., of the form INSERT INTO T_1 VALUES (...), then we set the Distance between such statements to 0, since one statement can be safely pruned against the other without any impact on choice of indexes so long as the weight of the pruned statement is added to the weight of the retained statement. However, in general, an UPDATE/INSERT/DELETE statement can have an "update" part *and* a "query" part: e.g., INSERT INTO T_1 SELECT * from T_2 WHERE <condition>. Such a statement *s* can be viewed as *(q,u)* where *q* is the query part and *u* is the update part. In this case, we define Distance (s_i, s_j) between two such statements as Distance (q_i, q_j) only if AdjustWeight(q_i, q_j) is (approximately) equal to AdjustWeight(u_i, u_j) (see Section 4 for the definition of the AdjustWeight function), and ∞ otherwise. The reason for this is that otherwise we would be biasing the compressed workload either towards the query part or towards the update part. We omit further details of this procedure due to lack of space.

[5] By low selectivity we mean a predicate that selects few records from the table.

5.1.3 Properties of the Distance function

As noted earlier, properties of the Distance function such as symmetry or triangle inequality can save us a (potentially significant) number of explicit computations of the function. From the definition of the above Distance function, it is clear that it does not obey symmetry, i.e., Distance (q_i, q_j) ≠ Distance (q_j, q_i). It is also easy to construct examples that show that the triangle inequality doesn't hold true for this distance metric, i.e., Distance (q_i, q_j) + Distance (q_j, q_k) is not necessarily greater than Distance (q_i, q_k) However, as shown by the following Lemma, our Distance function for index selection (defined in Section 5.1.2) satisfies the following property.

Lemma 2: For the Distance function defined in Section 5.1, if Distance (q_i, q_j) = Distance (q_j, q_i) = 0, then $\forall$ q_k, Distance (q_k, q_i) = Distance (q_k, q_j).

Proof: Omitted due to lack of space.

We can exploit the above Lemma to refine the query signature used in partitioning and hence reduce the work done when the Distance function is invoked. We omit the details of this optimization due to lack of space. In our experiments over real as well as synthetic workloads, we found that this optimization saved us anywhere between 10%-40% of the Distance computation overhead.

5.2 Distance Function for Approximate Answering of Aggregation Queries

For an overview of the AQP application, we refer the reader to Example 2. As explained in the example, the preprocessing step consumes a workload **W** and produces as output samples of one or more tables in the database. We assume that the workload consists of aggregation queries containing the COUNT/SUM aggregates possibly with selections, GROUP BY and foreign-key joins. The preprocessing step chooses the sample so as to minimize the average relative error of answering queries in **W** over the sample as compared to answering the queries on the full table. Note that for GROUP BY queries the relative error is averaged over all groups of the query – missing groups are assigned a relative error of 1.

Distance (q_i, q_j) for AQP therefore attempts to estimate the relative error in answering q_i if it is pruned but q_j is part of the compressed workload. As with the Distance function for index selection, we leverage the idea of *partitioning* and return ∞ if q_i and q_j have different signatures. Our signature of a query is defined by the subset of tables referenced in the query. If both queries belong to the same partition, we analyze them based on whether they have selections or GROUP-BY. If both queries are pure selection queries, i.e., do not contain GROUP BY, then we define Distance (q_i, q_j) as the fraction of records selected by q_i that are *not* selected by q_j. The intuition behind this definition is that the error in answering a pure selection query q_i depends on the number of records in the sample that are selected by q_i. If q_j is used to determine the sample, then the error for q_i *increases* as the overlap of q_i with q_j *decreases*. We note that if the database engine supports the DIFFERENCE operator, then this function can be estimated by invoking the query optimizer. Otherwise, this metric must be estimated based on analysis of selection predicates and using selectivity estimates.

When one query is a pure selection query and the other is a GROUP BY query, we set Distance (q_i,q_j) = ∞. When both queries have GROUP BY columns, Distance (q_i, q_j) is defined as follows. Let G be the set of grouping columns that occur in query q_i and let G' be the set of grouping columns that occur in both q_i and q_j. Let D(X) be the number of groups in a query (without selections) that contains exactly the grouping columns X. Then Distance (q_i, q_j) = 1 – D(G')/D(G). The intuition is that (i) the error for GROUP BY queries is dominated by missing groups, and (ii) the number of missing groups is likely to increase as the overlap between the grouping columns of q_i and q_j decreases.

6. ADJUSTING WEIGHTS

Recall that we defined a workload as a set of statements where each statement has an associated weight w_i (Section 2). The weight of a statement signifies the importance of the statement in the workload, and plays a role in determining the optimization function of the application. For example, index selection tools typically optimize a weighted function of the (estimated) execution cost of statements in the workload. Thus, an index that is useful for a query with large weight is more likely to be chosen by the tool. In order to prevent statements in the compressed workload from having unduly high or low weight relative to other statements, it is important that the weights of statements in the compressed workload be set appropriately.

In our architecture (see Figure 3), we address this issue in the *Adjust Weights* module as follows. At the end of the search algorithm (see Section 4), we find for every pruned statement q_i, the statement q_j nearest to it in the compressed workload (in terms of the Distance function) and adjust the weight of q_j. However, as illustrated by Example 3 (see Section 3) the naïve approach of simply adding the weight of the pruned statement to the nearest retained statement can result in poor quality of the compressed workload. In our solution, the application specific AdjustWeight (q_i, q_j) function serves the purpose of specifying the amount by which the weight of a retained statement q_j should be incremented if q_i is pruned and q_j is the closest statement to q_i.

For the index selection problem, we now present an appropriate AdjustWeight function. If q_i is pruned and its nearest statement is q_j, then we set the weight of q_j in the compressed workload to $w_j + w_i * \alpha_{ij}/\alpha_{jj}$ where α_{ij} is the benefit that query q_i gets from the indexes recommended for q_j. Due to lack of space, we omit details of how α_{ij} and α_{jj} can be computed efficiently. Instead, we revisit Example 3 and illustrate how our approach solves the problem of biasing.

Example 3 (Continued from Section 3): Suppose the benefits of an index on column *age* for Q_1 and Q_2 are 50 units and 40 units respectively. The actual total benefit from index on column age for **W** is 50*1 + 40*1 = 90 units, whereas for **W'**, this benefit is 40*2 = 80 units. Therefore, as pointed out earlier, we have biased the workload away from picking an index on column age. Using the approach described above, the weight of Q_2 in the compressed workload would be $w_2' = w_2 + w_1 * \alpha_{12}/\alpha_{22} = 1 + 1 * 50/40 = 2.25$. We can now easily verify that the benefit of the index on column age for the compressed workload is 2.25 * 40 = 90, which is same as the benefit for the original workload. ♦

Finally, we note that for the AQP application, we use the default AdjustWeight(q_i,q_j) function, which simply adds the weight of q_i to q_j.

7. EXPERIMENTS

In this section, we present an experimental evaluation of our solution for workload compression. We demonstrate through these experiments that: (1) In the context of the Index Tuning Wizard for Microsoft SQL Server 2000, our Distance function for index selection produces significant compression of the workload while obeying the given quality constraint. (2) The same Distance function for index selection works well on another index selection tool, viz., IBM DB2's Index Advisor. (3) The WC-KMED algorithm scales better than WC-ALL-PAIRS but the latter can achieve significantly more compression. (4) Our framework for workload compression can be applied to another application as well, viz. AQP, by simply providing an appropriate Distance function.

Setup: All experiments were run on an x86 900 Mhz dual processor machine with 512MB RAM and an internal 30GB hard drive running Microsoft Windows 2000 Server. We tested our solution on several databases and workloads, including real and synthetic schemas and workloads. We present results on two benchmark workloads (TPC-H [25] and APB [3]), two real workloads (Real-1 and Real-2) used within our corporation, and several synthetic workloads. The database for Real-1 is about 600MB and contains about 90% update statements, whereas Real-2 workload contains decision support queries against a 500 MB database. All the synthetic databases conform to the TPC-H schema and were generated using a synthetic data generation program [24]. The size of the synthetic databases were 1GB. The synthetic workloads were generated using a query generation program, which has the ability to vary a number of parameters including number of joins, number of group-by columns, number of order-by columns, number of selection conditions in a query, and percentage of update statements in the workload.

Evaluation Metrics: For the index selection application, we use the following metrics to evaluate the workload compression solution: (a) Percentage of queries pruned by workload compression (b) Percentage reduction in total tuning time, i.e., sum of running time of index selection tool on the compressed workload and time spent in compressing the workload as compared to running the tool on the original workload. (c) Percentage loss in quality of the solution produced by the index selection application. We use the percentage change in the optimizer-estimated cost of the original workload as the metric of quality. We obtain this by running the index selection tool on both the original workload as well as the compressed workload, implementing the recommendations and calculating the optimizer estimated running time of the original workload for both the cases. In the following experiments, we specify the constraint (Δ) on loss in quality to be 10% of the cost of the original workload **W** on the current database. For the AQP application, we measured loss in quality due to workload compression as follows. We report the difference in the average relative error of queries in **W** when the entire workload is used in the preprocessing phase (see Section 2.1) and the average relative error of queries in **W**, when the compressed workload is used in the preprocessing phase.

7.1 Effectiveness of Distance Function for Index Selection Tool on Microsoft SQL Server

We first evaluate our Distance function (see Section 5.1) for index selection against the Index Tuning Wizard for Microsoft SQL Server. Figure 7 shows the results of workload compression for the two real workloads and the two benchmark workloads. We fixed the search strategy to WC-KMED. We see that a large percentage of the queries (between 50%-90%) were pruned in three of the workloads and about 20% of the queries were pruned in the Real-2 workload. In each of these cases, the total time to run the Index Tuning Wizard was also significantly reduced due to workload compression, which shows that the workload compression step itself added little overhead. From the figure, we also see that the maximum loss in quality due to workload compression was less than 11%. This shows that our Distance function does a reasonable job of estimating the loss of quality.

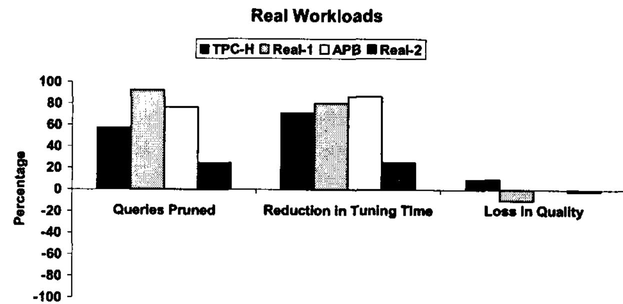

Figure 7. Results on Real and Benchmark Workloads

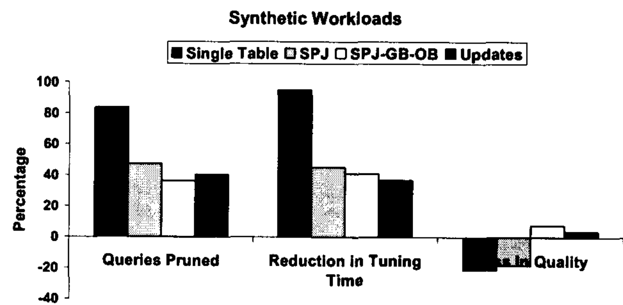

Figure 8. Results on Synthetic Workloads

Figure 8 shows the results of workload compression for index selection on the four kinds of synthetic workloads described earlier (each of size 300 queries), including a workload with 30% update statements. For each of these workloads, we once again see significant reduction in workload size as well as total tuning time while the loss in quality is very small (< 8%). Overall, this experiment illustrates the effectiveness of our Distance function for index selection.

7.2 Comparison of Search Strategies

In our next experiment, we compare the quality and scalability of the three search strategies presented in Section 4: WC-KMED, WC-ALL-PAIRS, and WC-PARTSAMP when applied to the index selection problem. Figure 9 shows the total compression achieved (as a percentage of the workload size) for each of the search strategies as the workload size is increased. The workload was a synthetic workload consisting of SPJ queries with Group-

By and Order-By. We see that WC-ALL-PAIRS achieves the most compression among the three strategies (see Section 4.2 for a qualitative comparison with WC-KMED). We see that not surprisingly WC-PARTSAMP (which we found was superior to uniform sampling) has the least compression – significantly less than WC-ALL-PAIRS and WC-KMED.

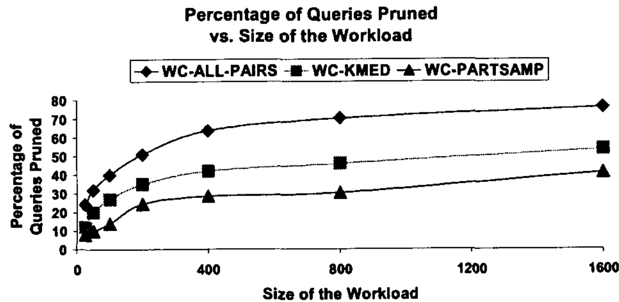

Figure 9. Search Algorithms: Compression Achieved

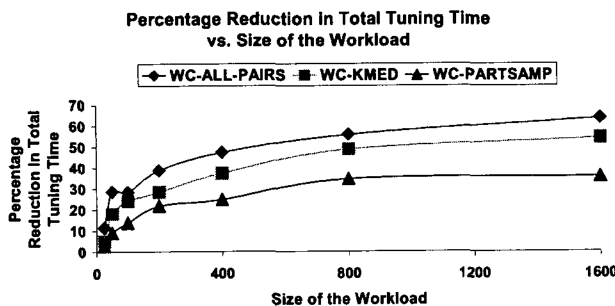

Figure 10. Search Algorithms: Total Reduction in Tuning Time

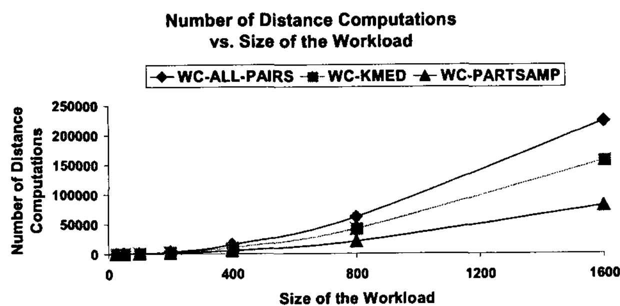

Figure 11. Search Algorithms: Number of Distance Computations

Figure 10 shows the total reduction in tuning time as the workload size is varied (same data points as in Figure 9). Here we see that WC-ALL-PAIRS and WC-KMED significantly outperform WC-PARTSAMP due to the superior compression achieved by these algorithms. The reason that WC-KMED shows similar reduction in running time to WC-ALL-PAIRS despite achieving less compression (Figure 9) is that the time for workload compression using WC-KMED is significantly less than when using WC-ALL-PAIRS. Figure 11 shows the reason for this – the total number of Distance function computations scales significantly better for WC-KMED as compared to WC-ALL-PAIRS.

Overall, this experiment emphasizes the fact that by ignoring workload information during compression, a sampling based approach loses out on the amount and quality of compression compared to both WC-ALL-PAIRS and WC-KMED.

7.3 Evaluation on Index Selection Tool for IBM DB2

The goal of our next experiment is to evaluate whether the Distance function we developed for index selection (see Section 5.1) works effectively when used for another index selection tool. We therefore used the Index Advisor for IBM DB2 to tune the original and compressed workloads respectively for the Real-2 workload. We then compared the execution time of the original workload when indexes recommended for the original workload were present to the execution time of the original workload when indexes recommended for the compressed workload were present. The workload compression achieved is 25%, and the loss in quality (i.e. increase in execution time) for this workload was only 1%. This experiment demonstrates that our Distance function is robust in the sense that it is not dependent on the specific implementation of the index selection tool.

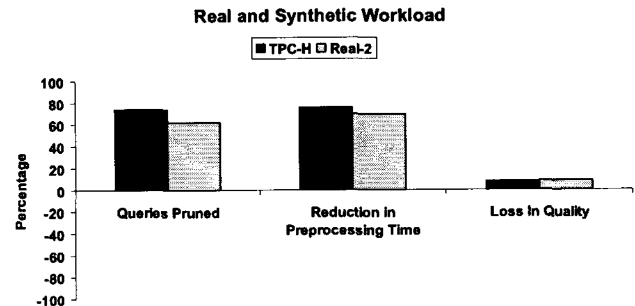

Figure 12. Results for Approximate Answering of Aggregation Queries application.

7.4 Evaluation for Approximate Aggregation Query Answering Application

Our final experiment illustrates the generality of our framework for workload compression and shows that it can be applied to other applications with only a modest effort that is required in providing an appropriate Distance function for the application. We evaluate the effectiveness of our solution for the AQP application (see Section 2.1 for more details on the application and Section 5.2 for the Distance function). We experimented with two workloads: the Real-2 workload and a synthetic workload of 100 aggregation queries on the TPC-H 1GB database. For this application the quality of the result is measured by the average relative error (see Section 2.1) of queries in the workload. We see from Figure 12 that we achieve significant workload compression and reduction in preprocessing time. At the same time the loss in quality (i.e., increase in average relative error of queries) is small.

8. RELATED WORK

There have been several papers that have applied sampling in the area of databases e.g., [18,20]. However, unlike our paper, these studies focus on the problem of sampling *data* and not the *workload*. Random sampling, has also been studied extensively in the statistics literature [8]. A key difference between our techniques and random sampling is that random sampling ignores

interaction among the objects being sampled (which is the primary source of its efficiency). In contrast, our technique focuses on achieving compression while taking into account interactions among statements via use of the Distance function. As shown by our experimental evaluation, this aspect is crucial for achieving significantly better quality of workload compression. In principle, our techniques are complementary to sampling, and the two techniques can be combined.

Our work also has strong similarity to the problem of clustering [19] which has been studied in the context of machine learning and data mining. Indeed as we have shown, our problem is equivalent to the well-known Minimum k-Median clustering problem. Also, a large class of work e.g., [4,12] on clustering has focused on cases when the points are in a metric space, i.e., distance function is symmetric and satisfies the triangle inequality. A key novelty of our work is applying clustering techniques in a principled manner in the context of workloads. Moreover, since we cannot assume that our distance functions satisfy metric spaces, we adapt well-known heuristic approaches to clustering to solve workload compression. Finally, there is a large body of work on query equivalence e.g., [5,11], which is also complementary to our work. These techniques may be useful in deriving distance functions for specific applications. One form of equivalence that can be exploited (and is application independent) is when two queries are semantically *identical*, i.e., they return the same result. Of course, applying these techniques is not free of cost could require significant computational effort.

9. ACKNOWLEDGMENTS

We would like to thank Gautam Das for important feedback on the algorithms and proofs in the paper, and Christian König for his valuable comments and help in experiments.

10. REFERENCES

[1] Aboulnaga, A. and Chaudhuri, S. Self-tuning Histograms: Building histograms without looking at data. Proceedings of the ACM SIGMOD, 1999.

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

[3] *APB-1 OLAP Benchmark, Release II.* OLAP Council Nov. 1998. http://www.olapcouncil.org/

[4] Arora, S., Raghavan P., and Rao, S. *Approximation schemes for Euclidean k-medians and related problems.* Proceedings of the 30th Annual Symposium on Theory of Computing, 1998.

[5] Aho, A.V., Sagiv, Y., and Ullman, J.D. *Equivalence of relational expressions.* SIAM Journal of Computing, Vol 8, 1979.

[6] AutoAdmin project at Microsoft Research (http://research.microsoft.com/dmx/AutoAdmin).

[7] Bruno, N., Chaudhuri S., and Gravano, L. *STHoles: A Multidimensional Workload-Aware Histogram.* Proceedings of the ACM SIGMOD, 2001.

[8] Cochran W.G. *Sampling Techniques.* John Wiley & Sons, New York, Third Edition 1977.

[9] Chaudhuri, S., Das G., Datar, M., Motwani R., and Narasayya V. *Overcoming Limitation of Sampling for Aggregation Queries.* Proceedings of the 17th Intl. Conference on Data Engineering, 2001.

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

[11] Ceri, S., and Gottlob G. Translating SQL into relational algebra: Optimization, semantics and equivalence of SQL queries. IEEE Transactions on Software Engineering, SE 11, 4 1985.

[12] Charikar, M., Guha S., Tardos E., and Shmoys D.B. *A constant-factor approximation algorithm for the k-median problem.* Proceedings of the 31st Annual Symposium on Theory of Computing, 199`9.

[13] Chaudhuri, S., and Narasayya V. *An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server.* Proceedings of the 23rd Intl. Conference on Very Large Databases, 1997.

[14] Chaudhuri, S., and Narasayya V. *Automating Statistics Management for Query Optimizers.* Proceedings of the 16th Intl. Conference on Data Engineering, 2000.

[15] Donjerkovic, D., Ioannidis, Y., and Ramakrishnan, R. *Dynamic Histograms: Capturing Evolving Data Sets.* Proceedings of the 16th Intl. Conference on Data Engineering, 2000.

[16] 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.

[17] Ganti, V., Lee M.L., and Ramakrishnan R. *ICICLES: Self-tuning samples for Approximate Query Answering.* Proceedings of the 26rd Intl. Conference on Very Large Databases, 2000.

[18] Gibbons, P.B., Matias Y., and Poosala V. Fast Incremental Maintenance of Approximate Histograms. Proceedings of the 17th Intl. Conference on Very Large Databases, 1997.

[19] Han, J., and Kamber M. *Data Mining: Concepts and Techniques.* Morgan Kaufmann Publishers, 2001.

[20] Haas P.J., Naughton, J.F., Seshadri S., and Stokes L. *Sampling based estimation of the number of distinct values of an attribute.* Proceedings of the 21st Intl. Conference on Very Large Databases, 1995.

[21] Lohman G., Skelley A., Valentin G., Zilio D., and Zuliani, M. *DB2 Advisor: An Optimizer Smart Enough to Recommend Its Own Indexes.* Proceedings of the 16th Intl. Conference on Data Engineering, 2000.

[22] Lin, J.-H. and Vitter, J.S. *ε-Approximations with minimum packing constraint violation.* Proceedings of the 24th Annual Symposium on Theory of Computing, 1992.

[23] Stillger, M., Lohman, G.M., Markl, V., and Kandil, M. *LEO – DB2's learning optimizer.* Proceedings of the 27th Intl. Conference on Very Large Databases, 2001.

[24] Chaudhuri S., and Narasayya V. TPC-D data generation with skew. Available via anonymous ftp from ftp.research.microsoft.com/users/viveknar/tpcdskew

[25] TPC Benchmark H. Decision Support. http://www.tpc.org