Automating Statistics Management for Query Optimizers

Surajit Chaudhuri, Vivek Narasayya
Microsoft Research, Redmond
{surajitc, viveknar}@microsoft.com

Abstract

Statistics play a key role in influencing the quality of plans chosen by a database query optimizer. In this paper, we identify the statistics that are essential for an optimizer. We introduce novel techniques that help significantly reduce the set of statistics that need to be created without sacrificing the quality of query plans generated. We discuss how these techniques can be leveraged to automate statistics management in databases. We have implemented and experimentally evaluated our approach on Microsoft SQL Server 7.0.

1. Introduction

The increasing importance of decision-support systems has amplified the need to ensure that optimizers produce query plans that are as optimal as possible. The query optimizer component of a database system relies on *statistics* on data for generating query execution plans. The availability of relevant statistics can greatly improve the quality of plans generated by the optimizer. In fact, in the absence of statistics, cost estimates can be dramatically different, often resulting in a poor choice of execution plans. On the other hand, unnecessary statistics may result in substantial overhead due to creation and update costs for statistics. As an example, consider a tuned TPC-D 1GB database on Microsoft SQL Server 7.0 with 13 indexes, and a workload consisting of the 17 queries defined in the benchmark. We recorded the plans for each query when no additional statistics on columns (besides statistics on indexed columns) were available. We then created a set of relevant statistics for the workload and re-optimized each query and recorded its plan. We observed that in all but 2 queries, the execution plans chosen with additional statistics were different, and resulted in improved execution cost.

Despite its importance, the problem of *automatically* determining the right set of statistics to build and maintain for a database has received little or no attention. The task of deciding which statistics to create and maintain is a complex function of the workload the database system experiences, the optimizer's usage of statistics, and the data distribution itself. Indeed, the choice of statistics changes as the workload and data distribution evolves. Today, this challenging task falls on the database administrator (DBA). Only a few DBA-s can do justice to the task of picking appropriate statistics for an enterprise database.

In this paper, we present techniques towards automating the management of statistics in databases. The choice of statistics must be guided by the nature of the workload that the database system experiences. By syntactic analysis of the workload, it is possible to identify a set of relevant statistics. However, the usefulness of statistics is determined not just by syntactic relevance, but also by the nature of the data distribution associated with the statistics. This is a chicken-and-egg problem, i.e., there is no way to determine if statistics are useful to build until we have actually built them! We address this difficulty by using a novel technique called *Magic Number Sensitivity Analysis* that significantly reduces the need to create all syntactically relevant statistics. In other words, this technique helps avoid creation of many "non-essential" statistics, i.e., statistics that are syntactically relevant but do not affect the quality of the plans.

Despite the success of magic number sensitivity analysis in building a smaller number of statistics, a mechanism to identify non-essential statistics is still needed since there is no way to escape building at least some non-essential statistics even using magic number sensitivity analysis. We present algorithms to detect such statistics. We also touch upon some of *the policy issues* that determine how the above techniques may be used to suit the performance needs of a system. We implemented our techniques on Microsoft SQL Server 7.0, and we present experimental results showing their effectiveness. This research work was done in the context of the AutoAdmin research project [1]. The goal of AutoAdmin is to make databases self-tuning and self-administering.

Finally, we note that the techniques described in this paper are general in the sense that the proposed algorithms do not depend on the specific structure of statistics used in a DBMS (e.g., type of histogram). However, the experimental evaluation of the techniques has been in the context of statistics used in Microsoft SQL Server (see Section 8 for details).

The rest of this paper is organized as follows. We discuss related work in Section 2. In Section 3, we present a formal framework that defines *essential statistics*. In Section 4, we present an effective algorithm for creating a set of statistics for a query using the magic number sensitivity analysis technique. In Section 5, we describe techniques for detecting non-essential statistics. Section 6 discusses policy issues for automating statistics management. We describe server extensions and implementation issues on Microsoft SQL Server 7.0 in Section 7. We present experimental results that validate our techniques in Section 8 and conclude in Section 9.

2. Related work

Traditionally, database administrators decided which statistics to create for a database. Recently, Microsoft SQL Server 7.0 has introduced the "auto-statistics" mode in which it creates all single-column statistics that are syntactically relevant for the incoming query. Statistics are updated when the number of rows modified in the base table exceeds a threshold, and statistics are dropped if they have been updated more than a predetermined number of times. Our work builds upon this state-of-the-art solution by: (a) avoiding creation of many statistics that are syntactically relevant (b) considering both single and multi-column statistics (c) taking into consideration the usefulness of a statistic in determining whether it should be dropped. Thus, we try to avoid the situation where we drop a useful statistic only to re-create it immediately for a subsequent query.

There is a large body of work that studies representation of statistics on a given column or columns (e.g., different kinds of histograms [10,13]). In this paper we have studied the orthogonal problem of deciding *which* column (or sets of columns) to build statistics on. While much work has also focussed on how to efficiently construct and maintain statistics (primarily histograms) using sampling [3,8,9,12,14], the large space of possible statistics makes our techniques important, particularly for large databases. Furthermore, if all statistics on a table are created from a *single* random sample of the data undesirable correlation in the statistics may result, particularly when block-level sampling is used.

The problem of identifying essential statistics has great relevance to the problem of selecting the right indexes for a database [2,4,6,7,11]. Specifically, the new generation of index tuning tools (e.g., Microsoft SQL Server Index Tuning Wizard [4]) builds statistics to determine the appropriate choice of indexes. Such tools will directly benefit from the techniques proposed in this paper since our techniques would help minimize the overhead of building statistics.

3. Framework for statistics selection

A *statistic* is a summary structure associated with a set of one or more columns in a relation. A *set of statistics* S can consist of single as well as multi-column statistics. Thus, {R_1.a, R_1.c, (R_2.c, R_2.d)} represents a set of 3 statistics consisting of single column statistics on R_1.a (on column a of relation R_1), R_1.c, and a *two-column* (also called *2-dimensional*) statistic (R_2.c, R_2.d). We represent the number of statistics in the set S by |S|.

An example of a commonly used statistic is a histogram. Many variants of histogram structures have been proposed, e.g., Equi-depth, MaxDiff. Multi-dimensional histogram structures can be constructed using Phased or MHIST-p [14] strategy over the joint distribution of multiple columns of a relation. Although all database systems exploit histograms, the nature of the specific histograms varies across systems.

For a *given* query Q, we can identify a set of statistics that potentially affects the plan chosen by the query optimizer for Q. We refer to this set as the *candidate statistics* for the query. In Section 3.1 we discuss how to determine candidate statistics for a given query. However, as we demonstrate in this paper, creating all candidate statistics for a query is infeasible in practice, even using sampling based techniques. Therefore, our goal is to create a smaller subset of the candidate statistics such that the plan chosen by the optimizer using the smaller subset is as good (or nearly as good) as the plan chosen if all candidate statistics had been created. We call such a minimal subset an *essential set*, and we formally define this notion in Sections 3.2 and 3.3.

3.1. Candidate statistics for a query

The syntax of a query helps identify a set of *relevant* columns for the query, i.e., columns referenced in the query such that statistics on these columns potentially impacts the optimization of the query. For example, a column that occurs in the WHERE clause or in the GROUP BY clause is relevant for the query[1]. Statistics on a condition are useful since they help estimate the selectivity of the predicate. Statistics on a column in a GROUP BY clause can help estimate the number of groups (i.e. distinct values) in the column.

[1] Note that a column that is only referenced in an ORDERY BY clause is not relevant since statistics on that column cannot affect the cost estimation or plan of that query.

The simplest approach for defining candidate statistics for a query is to consider one single-column statistic on each relevant column in the query. However, a set of multi-column statistics on combinations of relevant columns from the same table may also be very useful for the optimizer. For example, if a query has **k** join predicates between two tables R_1 and R_2: $R_1.a_1=R_2.b_1$, … $R_1.a_k=R_2.b_k$, then multi-column statistics on $(a_1,…,a_k)$ and $(b_1,…,b_k)$ may be useful since they provide information to the optimizer on the joint data distribution of columns $\{a_1,…,a_k\}$ and $\{b_1,…,b_k\}$ respectively. However, rather than building a single multi-column statistic containing all relevant columns, it may sometimes be advantageous to build one or more multi-column statistics on a smaller number of the relevant columns. Thus, the space of candidate multi-column statistics (and hence, the set of candidate statistics) for a query can be very large and deciding which multi-column statistics to build is complex. In Section 7.1, we describe the algorithm for candidate statistics we used in our implementation on Microsoft SQL Server. We emphasize that the techniques presented in this paper are oblivious to the exact set of candidate statistics, i.e., they work with any technique to choose candidate statistics.

3.2. Equivalence of sets of statistics

The quality of two plans that use *different* sets of statistics can be compared by their execution cost. Unfortunately, the above definition of comparing quality of plans is not suitable since testing *equivalence of sets of statistics* would require execution of queries, which is prohibitively expensive. In this paper, we will consider alternative definitions of equivalence of sets of statistics that can be used more effectively:

- *Execution-Tree equivalence*: Two sets of statistics S and S' are Execution-Tree equivalent for a query Q if the optimizer generates the same query execution tree for Q for both choices of S and S'. We note that Execution-Tree equivalence is the strongest notion of equivalence since it implies execution cost equivalence.
- *Optimizer-Cost equivalence*: Two sets of statistics S and S' are optimizer-cost equivalent for a query Q if the optimizer-estimated costs of the plans are the same irrespective of whether S or S' is used. In such cases, the plans generated by the optimizer for Q can still be different. Thus, this is a weaker notion of equivalence than Execution-Tree equivalence.
- ***t**-Optimizer-Cost equivalence*: This definition generalizes optimizer-cost equivalence. Let us use Estimated-Cost (Q, S) to denote the optimizer-estimated cost for query Q when the existing set of statistics is S. We say that two sets of statistics S and S' are **t**-Optimizer-Cost equivalent if Estimated-Cost (Q, S) and Estimated-Cost (Q, S') are within **t**% of each other[2]. This definition reflects the intuition that while analyzing the choice of statistics we can afford to ignore the differences among plans that differ in cost in a limited fashion. The value of **t** reflects the degree of rigor used to enforce equivalence. In particular, Optimizer-Cost equivalence is a special case of **t**-Optimizer-Cost equivalence.

The above definitions have been presented in the order of increasing flexibility and may be chosen to trade cost of statistics selection with accuracy of plans. Although **t**-Optimizer-Cost equivalence is the weaker than the execution-tree equivalence, it is a pragmatic choice for statistics selection and we used this definition for our implementation. In our experiments on Microsoft SQL Server, we have found that a value of **t** = 20% is a conservative choice (see Section 8.2).

3.3. Essential set of statistics

Most query optimizers are known to improve their estimation with additional statistical information. Thus, we *assume* that the execution cost of a plan for a query does not increase as additional statistics are created. Indeed, in the experiments that we have carried out on Microsoft SQL Server 7.0, we have not observed any violation of the above assumption. However, note that pathological violation of the above assumption is clearly possible. For example, estimation of selectivity of a predicate via a system-wide constant may coincidentally be more accurate than the selectivity computed via a histogram.

If we accept the above assumption, then it follows that the optimizer chooses a plan with the least execution cost when the entire set of candidate statistics **C** for Q is created. Since the candidate set **C** can be quite large, our goal is to find $S \subseteq \mathbf{C}$ that is a *much smaller* compared to **C,** but the quality of the plan is "as good". The following definition formalizes this intuition. The following definition is with respect to a snapshot of the database and the implicit parameters have been omitted from the definitions.

Definition 1. *Essential set of statistics for a query.* Given a query Q and a set of candidate statistics **C** for the query, an *essential set* of statistics for Q, is a subset S of **C**, such that S, but no proper subset S_1 of S, is *equivalent* to **C** with respect to Q.

[2] More precisely, |Estimated-Cost(Q,S) – Estimated-Cost(Q,S')| /Estimated-Cost(Q,S) < t/100, where Estimated-Cost(Q,S) < Estimated-Cost(Q,S') .

Note that *any* of the equivalence definitions presented in the previous section may be used. An *essential set* identifies a subset of statistics such that we can remove no more statistics without violating the property of equivalence with respect to the set of candidate statistics. This definition does not preclude the possibility of having multiple essential sets of statistics for the same given query. In fact, it is possible to further refine the definition to choose among the essential sets by some other criterion. For example, we may prefer the essential set that has the minimal cost of updating statistics. The following example illustrates the definition of essential set for execution-tree equivalence.

Example 1. *Essential set of statistics*. Let Q be a query SELECT * FROM T_1, T_2 WHERE T_1.a = T_2.b AND T_1.c < 100. Let **C** = {T_1.a, T_2.b, T_1.c} be the set of candidate statistics. Let us use the notation Plan(Q,X) to denote the execution tree for the query Q where X is the set of available statistics. Let S = {T_1.a, T_2.b}. For S to be an essential set of statistics for Q, using *execution tree* equivalence, each of the following conditions must be true: (1) Plan(Q,S) = Plan(Q,**C**) (2) Plan(Q,{T_1.a}) $\neq$ Plan(Q,**C**) (3) Plan(Q,{T_2.b}) $\neq$ Plan(Q, **C**) and (4) Plan(Q,{}) $\neq$ Plan(Q,**C**).

So far, we have defined candidate statistics and essential statistics with respect to a query. We now define these concepts for a *workload*, i.e., a set of queries. We define the candidate statistics for a workload as the union of candidate statistics for each query in the workload. Likewise, an essential set for a workload must contain an essential set of statistics for each query, but no more.

Definition 2. *Essential set of statistics for a workload*. An essential set of statistics for a workload W with respect to the set of candidate statistics **C** for the workload, is a minimal subset S of **C** that is equivalent to **C** with respect to every query Q in the workload W.

4. Identifying and creating an essential set of statistics

In this section, we present techniques to identify *an* essential set for a query (and a workload) efficiently. We accomplish this using a novel technique called *magic number sensitivity analysis (MNSA).* The MNSA algorithm has the following structure:

```
Repeat
    If Enough statistics
        Return
    Identify and build the next statistic
End Repeat
```

Thus, the technique has two key components: (a) a test to determine whether or not any additional statistics are needed and (b) a strategy to pick the next statistic to create from the set of remaining candidate statistics for the query. We describe these two steps in Section 4.1 and 4.2 respectively. Section 4.3 presents the overall algorithm. Note that we can create a set of statistics for a workload by repeatedly invoking MNSA for each query in the workload.

4.1. Testing for essential set property

Suppose that S is the existing set of statistics in the database. For a given query Q, with a set of candidate statistics **C**, we do not need to create any more statistics for Q if S *includes* an essential set of statistics. However, verifying the above condition is difficult since we cannot test that S is equivalent to **C** with respect to the query Q until all remaining candidate statistics in **C**-S have been created! Therefore, what we need is a method for determining whether the existing set of statistics is adequate *without requiring creation* of the remaining candidate statistics. The following example illustrates a scenario where the remaining candidate statistics for a query are not needed:

Example 2. Let Q be the query: SELECT E.EmployeeName, D.DeptName FROM Employees E, Department D WHERE E.DeptId = D.DeptId AND E.Age < 30 AND E.Salary > 200K. Assume that statistics on the join columns as well as E.Salary already exist but no statistics on E.Age are available. Further, assume that the fraction of Employees satisfying the predicate (E.Salary > 200K) is very small. In such a case, the join method chosen by the optimizer, and hence the plan for Q, is not likely to change even if we build statistics on E.Age. The question is how can we determine that statistics on E.Age are not useful *without* first building statistics on E.Age?

We now discuss how to effectively test if S includes an essential set of Q, but without creating statistics in **C**-S. For the rest of this section, we will focus only on Select-Project-Join (SPJ) queries[3] and use **t**-Optimizer-Cost equivalence. The key to our technique is to carefully consider how the presence of statistics impacts optimization of queries. The optimizer uses statistics primarily for estimation of selectivity of predicates. For example, if a histogram is available on R.a and the query Q has a condition R.a < 10, then the histogram is used to estimate a selectivity for the predicate. If statistics appropriate for a predicate are not available, then the

[3] Without loss of generalization, we assume that the SPJ query is normalized and does not contain the NOT Boolean operator.

optimizer uses a *default* "magic number" for the corresponding selectivity. Magic numbers are system wide constants between 0 and 1 that are predetermined for various kinds of predicates. For example, consider query Q in Example 2 above. Since statistics on the column E.Age are not present, most relational optimizers use a default magic number, say 0.30, for the selectivity of the range predicate on E.Age. Thus, for an SPJ query Q, the dependence of the optimizer on statistics can be conceptually characterized by a set of selectivity *variables*, with one selectivity variable corresponding to each predicate in Q. The specific value used for the selectivity variable for a predicate must be between [0,1]. The value is determined either by using an existing statistic or by a default magic number. Thus, one way to confirm that S includes an essential set for the query Q is:

(a) Identify which selectivity variables of Q are forced to use default magic numbers due to lack of available statistics in the existing set S. Let this set be: $\{s_1,s_2,\ldots,s_k\}$.

(b) Verify that the optimizer-cost estimate is insensitive (i.e., satisfies the **t**-Optimizer-Cost equivalence) to changes in the values of the selectivity variables identified in (a).

It would seem that in order to test (b), we would need to optimize Q once for all possible values of each selectivity variable in (a). However, by and large, it is a safe assumption that the *optimizer-estimated cost of an SPJ query is monotonic in the values of the selectivity variables*. We will refer to the above as the *cost-monotonicity* assumption. Therefore, the following steps suffice to check if the existing set of statistics S contains an essential set for an SPJ query Q:

(i) Let P_{low} be the plan where we associate for each s_i ($1\le i \le k$) a selectivity ε, where ε is a *small* value > 0. Similarly, let P_{high} be the plan where we associate for each s_i ($1\le i \le k$) a selectivity 1-ε. The implementation needed to construct P_{low} and P_{high} are described in Section 7.

(ii) If all predicates have selectivity between ε and 1-ε, then due to our assumption on cost-monotonicity, P_{low} is the plan with the lowest cost, and P_{high} is the plan with the highest cost.

(iii) If P_{low} and P_{high} are **t**-Optimizer-Cost equivalent, then due to cost monotonicity assumption, all plans where s_j is in the range [ε, 1-ε] (j = 1,…,k) must also be **t**-Optimizer-Cost equivalent [5]. Thus, **t**-Optimizer-cost equivalence exists between the plan using the default magic numbers and the plan using statistics for one or more of the selectivity variables $s_1,\ldots,s_k$.

Note that the test in (iii) is *sufficient* to verify that the existing set of statistics S includes an essential set for Q. If the test fails, i.e., P_{low} and P_{high} are not **t**-Optimizer-Cost equivalent, then we assume that one or more additional statistics are necessary to improve the plan for Q.

The basic strategy outlined above needs to be qualified with several comments. First, note that even for a single selectivity variable, multiple statistics (e.g., multi-column as well as single-column histograms) may be applicable with different degrees of accuracy. Therefore, the choice of selectivity variables in step (a) needs to be extended. Second, for an SPJ query, MNSA guarantees inclusion of an essential set of the query only as long as the selectivity of predicates in the query (for the given snapshot of data) is between ε and 1-ε. Therefore, it is important to choose a small value for ε. In our implementation, we used ε=0.0005. Third, although for SPJ queries MNSA ensures that an essential set is included among the statistics, it is necessary to extend the above methodology beyond simple queries. We can handle aggregation (GROUP BY/ SELECT DISTINCT) clauses by associating a selectivity variable that indicates the fraction of rows in the table with distinct values of the column(s) in the clause. For example, a value of 0.01 for such a selectivity variable for the clause GROUP BY ProductName implies that the number of distinct values of ProductName is 1% of the number of rows in the table. For *complex* (e.g., *multi-block*) SQL queries, the basic MNSA strategy is not guaranteed to ensure inclusion of an essential set since the cost-monotonicity assumption may not be true for such queries. However, analyzing relationships among predicates in the query may enable us to extend the applicability of the algorithm. For example, consider the case where the cost increases monotonically with increasing selectivity s_i of predicate p_i, and decreases monotonically with increasing selectivity s_j of predicate p_j. In this case, the selectivity variables that should be used (in steps (i)-(iii)) correspond to p_i and NOT(p_j) respectively since they ensure cost-monotonicity. Finally, the above presentation has centered on **t**-Optimizer-Cost equivalence, and does not consider cases where execution-tree equivalence is desired. We defer a complete discussion of these issues to [5].

4.2. Finding the next statistic to build

Once we determine that additional statistics need to be created to capture an essential set for the query, we must decide *which* statistic to build next. Our goal is to decide which of the remaining candidate statistics, if created, is most likely to benefit query optimization. A good heuristic to identify the next statistic to build can sharply lower the number of statistics that need to be created.

Our approach exploits information obtained from the *plan* of the query. The plan for a query contains valuable information about individual operators in the plan tree, including the cost of the operator and the statistics that are potentially relevant to that operator. We use the intuition that statistics that are relevant for the *most expensive operators in the current query plan* are likely to have the most impact on the optimizer. Thus, we identify the most expensive operator in the plan tree for which one or more candidate statistics have not yet been built, and consider those statistics. The most expensive operator/node is the node **n** in the plan for which the following expression is maximal:

*cost (plan subtree rooted at **n**) – Σ cost (Children (**n**))*. The space of candidate statistics considered for our implementation is described in Section 7.1. We note that there may exist dependency among statistics. An example of such dependence is statistics on columns of a join predicate. In such situations, we need to create a pair of statistics rather than a single statistic.

```
1.  Let S be the current set of statistics.
2.  Let P = Plan of Q with default magic numbers
3.  Repeat
4.      Let $s_1 \ldots s_k$ be the current selectivity variables for Q
        for which the optimizer must use magic numbers
        due to lack of statistics.
5.      $P_{low}$ = Plan of Q when $\{s_1=\varepsilon, \ldots, s_k=\varepsilon\}$.
6.      $P_{high}$ = Plan of Q when $\{s_1=1-\varepsilon, \ldots, s_k=1-\varepsilon\}$
7.      If (($|Cost(P_{high}) - Cost(P_{low})| / Cost(P_{low})) \le t\%$)
            Return
8.      s = FindNextStatToBuild (P)
9.      If (!s) Return
10.     Build statistic s; $S = S \cup \{s\}$.
11.     P' = Plan of Q with default magic numbers
12.     P = P'
13. End Repeat
```

Figure 1. Statistics Creation using Magic Number Sensitivity Analysis for a query Q.

4.3. Algorithm for creating a set of statistics for a query

Figure 1 describes our algorithm for creating a set of statistics for a query based on the techniques presented in Sections 4.1 and 4.2. We denote the optimizer-estimated cost of a plan P by Cost(P). The algorithm computes P_{low} and P_{high} for Q and terminates statistics creation if the cost differential between these two plans is below the threshold of **t**%. We decide which statistic to create next by invoking the function *FindNextStatToBuild* (described in Section 4.2). This function proposes the next statistic to build based on the "current" plan of the query, i.e., the plan P obtained with the *default* magic number settings. For *FindNextStatToBuild*, it is appropriate to use P instead of P_{low} or $P_{high,}$ since using the latter correspond to extreme selectivity values. Therefore, the overhead of MNSA is three optimizer calls for each statistic that is created. The running time of the algorithm is linear in the number of candidate statistics for the query. Since the time to create a statistic typically far exceeds the time to optimize a query, the algorithm in Figure 1 is a significant improvement over algorithms that create all candidate statistics. We note that since creating candidate statistics on "small" tables is inexpensive, MNSA can be augmented with a threshold such that the analysis in MNSA is triggered only if the table size *exceeds* the threshold. Finally, note that a sufficient set of statistics for a workload can be obtained by invoking MNSA for each query in the workload.

5. Identifying non-essential statistics

MNSA ensures that given a query (or workload), the existing set of statistics is augmented to include an essential set of statistics for the query (or workload). However, MNSA uses only a sufficient condition (see Section 4.1) to conclude that statistics need not be created for an essential set. Therefore, despite using MNSA, we cannot always avoid building some non-essential statistics. A database system that is sensitive to the update cost of statistics requires the ability to drop non-essential statistics. In this section, we describe techniques to identify such non-essential statistics.

In Section 5.1, we present an adaptation of MNSA, called Magic Number Sensitivity Analysis *with Drop* (**MNSA/D**) that interleaves detection of non-essential statistics with creation of statistics. This extension ensures that relevance of the statistics is evaluated as soon as it is created. This adaptation significantly reduces the number of non-essential statistics left behind by MNSA but can no longer guarantee that an essential set for the query is included even where the query is (or, queries in the workload are) restricted to SPJ. Therefore, we also present the *Shrinking Set* algorithm (Section 5.2) that for a given query (or workload), together with MNSA, can *guarantee* an essential set of statistics, and does not leave behind any non-essential statistics. However, we should note that MNSA/D performs satisfactorily in practice. We present detailed experimental comparison of MNSA/D and the Shrinking Set algorithm in [5].

Finally, we note that identifying non-essential statistics only marks the statistics as candidates for eventual deletion. Such statistics are added to a *drop-list*. A statistic in the drop-list is a candidate for being physically deleted. However, physical deletion of statistics may be delayed as described in Section 6. Furthermore, if the statistic *s* is subsequently found to be useful for another query, then instead of re-creating the statistic s, it can simply be removed from the drop-list and made accessible to the optimizer.

5.1. Statistics creation with non-essential statistics identification (MNSA/D)

MNSA/D is a simple adaptation of the MNSA algorithm. If we observe that creation of the statistic *s* (step 10, Figure 1) leaves the plan of the query Q (step 11, Figure 1) equivalent, then *s* is *heuristically* identified as non-essential. The increase in running time of MNSA/D compared to MNSA is negligible since the only overhead is: (a) comparing the plan tree of the query obtained in the current iteration with the plan tree obtained in the previous iteration of MNSA (b) maintaining the drop-list.

Unlike MNSA, even for SPJ queries, MNSA/D cannot guarantee an essential set because it might be erroneously aggressive in detecting non-essential statistics. For example, the plan for a query Q may be the same for the set of statistics S and S $\cup$ {*g*}, but the plan may be different for S $\cup$ {*g*,*h*}. In such a case, MNSA/D may greedily drop the statistic *g* if the statistic *g* happened to be created first, since plans with S and S $\cup${g} are the same. MNSA/D cannot also guarantee that all non-essential statistics will be eliminated since its decision to include a statistic is done in a greedy manner, i.e., once a statistic is included, it is not considered for dropping as other statistics get created. Nonetheless, in practice we observe that MNSA/D eliminates a significant fraction of non-essential statistics and only rarely fails to preserve an essential set.

5.2. The Shrinking Set algorithm for finding an essential set

We now describe an algorithm, called *Shrinking Set,* which for a given query (or a workload) can produce an essential set of statistics. The Shrinking Set algorithm *assumes* that the current set of statistics S is a *superset* of the required essential set. Such a set S of statistics may be created by using the vanilla MNSA algorithm. We can detect a non-essential statistic *s* if we find that removing *s* from S has no impact on the plan of *any* query in the workload. In such a case, we are assured that there is an essential set $\subseteq$ S that does not contain s. The algorithm, outlined in Figure 2, considers each statistic one at a time. We use the notation Plan(Q,X) to denote the execution tree for the query Q where X is the set of available statistics. If the absence of the statistic (Step 4) does not affect the plan, we discard that statistic and never consider it again. We continue shrinking the set until we have iterated over all statistics in the initial set S. The result of the algorithm is guaranteed to be an essential set. Observe that the specific set of statistics that is produced may depend on the order in which the statistics are tested for inclusion in the essential set.

Figure 2 presents the Shrinking set algorithm for execution-tree equivalence criteria. The algorithm can be used in a similar fashion for **t**-Optimizer-Cost equivalence [5]. In the worst case, the shrinking set algorithm must make |S|*|W| calls to the optimizer, where S is the initial set of statistics and W is the workload. A key technique for improving the efficiency of the Shrinking Set algorithm is based on the intuition that it is often possible to quickly find a small set of statistics that is essential for many queries in the workload. Once such a set (call it S') is found, we subsequently need to consider only those queries for which S' is not adequate. We defer the details of this technique as well as experimental evaluation of Shrinking Set to [5].

```
1. S = Initial Set of Statistics
2. R = S
3. For each s ∈ S Do
4.    If Plan (Q, R – {s}) = Plan (Q, S) for each query Q
         in W for which s is potentially relevant
5.       R = R - {s}
6. Return R
```

Figure 2. The Shrinking-Set (W,S) algorithm for finding an essential set for a workload W from an initial set S.

6. Issues in automating statistics management

Sections 4 and 5 provided the basic *mechanisms* for automating statistics. In this section, we comment on some of the *policy* issues in leveraging these techniques. The database administrator (DBA) will need to set these policies to suit the needs of their databases. However, the DBA's job is still considerably simplified since he/she is no longer responsible for selecting the appropriate statistics.

Creating Statistics*:* The most aggressive policy for statistics creation is to build statistics *on the fly* for each incoming query. Although this policy incurs increased query compilation time, it ensures the best possible execution plan for every incoming query. Microsoft SQL Server 7.0 is an example of such a system. For each incoming query, it creates syntactically relevant statistics. For such a system, using MNSA or MNSA/D can significantly reduce the time spent on creating statistics on the fly.

On the other hand, the most conservative policy for statistics creation is for the system (or the DBA) to *periodically* initiate an off-line process to determine a set of statistics for the workload, and this set is retained until the next invocation of the off-line process. In such an off-line process, MNSA can be used to create a set of statistics for each query in the workload, and this can be followed by the Shrinking Set algorithm to eliminate non-essential statistics. We also note that in such an off-line process, we may exploit workload characteristics to optimize statistics creation. For example, in MNSA we may only consider building statistics that would potentially serve a significant fraction of the workload cost.

The mechanism of *aging* can help strike a balance between the quality of the plan for the incoming query and the cost of creating and updating statistics. The basic idea behind aging is that statistics with high creation/update cost that have been dropped after being found non-essential for a workload should not be re-created immediately if the same (or similar) workload repeats on the server. In other words, aging allows us to "dampen" the process of re-creation of recently dropped statistics. However, we need to ensure that optimization of significantly *expensive* queries are not adversely affected due to aging. We defer the details of the aging algorithm and its experimental evaluation to [5].

Dropping Statistics: The decision of *when* to physically drop non-essential statistics that have been placed in the drop-list (see Section 5) is also a policy issue. Such deletion may be triggered when the cost of maintaining statistics in the drop-list is high. Statistics may be either dropped individually or together (e.g. all non-essential statistics on a table). An example of a policy for dropping statistics is the one used in Microsoft SQL Server 7.0. The server maintains a counter for each table that records the number of rows modified in the table since the last time statistics on the table were updated. When this counter exceeds a specified fraction of the table size, statistics on the table are updated. When a statistic has been updated more than a predetermined number of times, it is physically dropped. Such a policy can be easily modified to take advantage of our techniques (such as MNSA/D or Shrinking Set) by only dropping statistics that are in the drop-list. Finally, we note that in an update sensitive system, we may consider dropping even an essential statistic with a high update cost.

7. Implementation

In this section we describe the necessary server extensions to support the techniques presented in this paper; and aspects of our implementation that are specific to Microsoft SQL Server 7.0.

7.1. Candidate statistics for a query

As described in Section 3.1, for a given query we can derive a set of candidate single and multi-column statistics. In general, given a multi-column candidate statistic for a query, any subset of those columns is also a candidate statistic. However, many of today's relational systems associate less than complete information with multi-column statistics. For example, in Microsoft SQL Server 7.0, a multi-column statistic on (*a*,*b*,*c*) is asymmetric since it contains a histogram on *a*, and density information on the leading prefixes of (*a*,*b*,*c*). Our Candidate Statistics algorithm considers for a query: (a) a single-column statistic on each relevant column (b) one multi-column statistic per table on the columns in selection predicates (c) one multi-column statistic per table on the join columns (d) one multi-column statistic per table on the group by columns. Our experiments (see Section 8.2) show that the above choice of candidate statistics performs satisfactorily.

Example 3: Let Q_2 = SELECT * FROM R_1, R_2 WHERE R_1.a = R_2.b AND R_1.c = R_2.d AND R_1.e < 100 and R_1.f > 10 and R_1.g = 25. Then, the candidate statistics proposed are (a), (b), (c), (d), (e), (f), (a,c), (b,d), (e,f,g). We do *not* propose statistics (e,f), (f,g), (e,g).

7.2. Server extensions

The techniques described in this paper assume an interface that is not standard in today's relational database systems but can be readily implemented on most systems. This interface obtains Plan(Q,S') where S' is a *subset* of the existing set of statistics S. By default, today's optimizers consider all available statistics during optimization. Thus, the optimizer must be told *not to consider* the set of statistics (S-S') when optimizing Q. We extended Microsoft SQL Server to support this interface:

Ignore_Statistics_Subset (*db_id*, *stat_id_list*), where *db_id* is an identifier for the database, and *stat_id_list* is a list of statistics that should be ignored during query optimization. To implement this call, we store the arguments in a *connection specific* buffer in the server. Subsequently, when a query is optimized from the same connection, the optimizer ignores the subset of statistics currently in the buffer.

As discussed earlier, optimizers typically use a default magic number for the selectivity of a predicate for which no statistics are available. Therefore, to support the MNSA technique (see Section 4), we had to modify the selectivity estimation module to accept the selectivity of such predicates as a parameter rather than using the default magic number (a compile-time constant).

8. Experiments

In this section we describe experiments that show the effectiveness of algorithms presented in this paper. We show through experiments that:

- Our algorithm for *candidate statistics* proposes significantly fewer statistics than an exhaustive approach, without adversely affecting the quality of plans chosen by the optimizer.
- *MNSA* significantly reduces the time spent on statistics creation without significantly affecting the execution cost of queries in the workload.
- The *MNSA with drop (MNSA/D)* algorithm reduces the update cost of statistics compared to MNSA by identifying many non-essential statistics.

8.1. Experimental setup

Databases: We use the popular TPC-D benchmark [16] schema for our experiments. One of the requirements of the benchmark however, is that the data is generated from a *uniform* distribution. Since we wanted to verify the effectiveness of our techniques over different data distributions, we modified the TPC-D generation program to support data generation with varying degree of *skew*. In particular, the program generates data for each column in the schema from a Zipfian distribution. The degree of skew in the data is controlled by the Zipfian parameter z, which can be varied between 0 (uniform) and 4 (highly skewed). The program can also generate an instance of TPC-D containing mixed data distributions by assigning each column a randomly chosen value of z between 0 and 4. We have made this program (which runs on x86/Windows NT platform) available for download from [17]. Our experiments were run on an Intel Pentium 400 MHz processor with 256MB RAM, and two internal 9 GB disk drives.

Workloads: In addition to experimenting with the queries defined in the TPC-D benchmark, we used the automatic query generation tool Rags [15] to generate a variety of workloads. We varied the following workload characteristics in Rags: (a) Percentage of SQL insert/delete/update statements (0%, 25%, 50%) (b) Complexity of queries, determined by max number of tables in a query (*Simple* – 2 tables, *Complex* – 8 tables). (c) Number of queries (100, 500, 1000). In our graphs, we use the notation U25-S-1000, for example, to denote a *Simple* 1000 statement workload consisting of 25% update/insert/delete statements generated using Rags. In this section, we report results on the original TPC-D workload (TPCD-ORIG) as well as workloads generated using Rags.

8.2. Results

Candidate Statistics Algorithm: In this experiment, we measure the reduction in statistics creation time as well as the drop in quality due to our heuristic approach for proposing candidate statistics (Section 7.1) compared to an algorithm that creates *all* syntactically relevant statistics (referred to as Exhaustive). We measure the drop in quality by the increase in execution cost of workload due to changed plans. Figure 3 shows that across various data distributions, our approach dramatically reduces the time for creating candidate statistics (by 50-80%). Moreover, we found that across the combination of databases and workloads shown in Figure 3, the increase in execution cost of the workload due to the pruning of statistics never exceeded 3%. This experiment shows that our candidate statistics algorithm significantly reduces the number of statistics that need to be considered for a query.

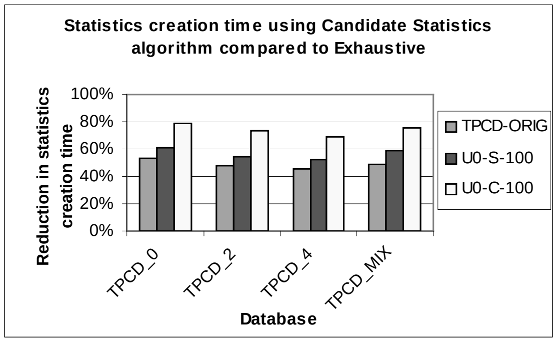

Figure 3. Evaluation of Candidate Statistics Algorithm

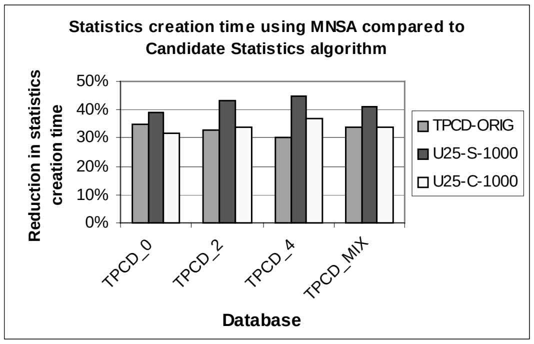

Figure 4. Evaluation of MNSA

Effectiveness of MNSA: To study the effectiveness of MNSA, we compare the following two alternatives: (a) create all statistics proposed by our Candidate Statistics algorithm (b) create statistics by using MNSA on the statistics proposed by the Candidate Statistics algorithm. We measured (1) reduction in statistics creation time by using MNSA and (2) impact of reduced statistics creation on the execution cost of the workload. We included the overhead for performing MNSA as part of statistics creation time. We set the threshold **t** (see Section 3.2) to 20% in our experiments. Figure 4 shows that for various data distributions and workloads, using MNSA reduces the statistics creation time significantly (30-45%). Moreover, we found that the impact of not creating these statistics on the quality of the plan was minimal, since the workload execution cost never increased by more than 2% for the workloads shown in Figure 4. We also conducted an experiment where the candidate statistics considered were only single-column statistics on relevant columns of a query. Here too we saw reduction in statistics creation time of above 30% in all cases, with small increase in execution cost across workloads. This experiment establishes the high quality of pruning by MNSA.

TPCD_0	TPCD_2	TPCD_4	TPCD_MIX
31%	34%	32%	30%

Table 1. Reduction in update cost of statistics using MNSA/D compared to MNSA (U25-C-100 workload)

Quality of MNSA/D: We measured quality of MNSA/D using two metrics: (1) the cost of updating the set of statistics left behind by the algorithm (2) the increase in execution cost if the original workload is re-run after dropping statistics. Table 1 shows that MNSA/D significantly reduces the update cost compared to MNSA across databases. For these databases, we also found that using MNSA/D to drop statistics for the workload, followed by re-running the same workload, increased the execution cost by no more than 6% (in TPCD_4) for the above databases. We observed similar results for other workloads. We conclude that MNSA/D detects many non-essential statistics at low overhead, which makes it an attractive alternative for automatic statistics management.

9. Conclusion

In this paper, we have presented techniques that help automate the important task of statistics management in database systems. We have provided a set of formal definitions that guide our selection criteria for statistics. The problem is intrinsically hard due to the chicken-and-egg nature of the problem – the usefulness of statistics can be determined only after they have been constructed. We presented the Magic Number Sensitivity Analysis technique that helps avoid creation of statistics that are not useful, and thus sidesteps the chicken-and-egg problem. We also presented techniques to identify non-essential statistics, which may then be dropped to reduce overhead of statistics update. Our implementation and experimental evaluation on Microsoft SQL Server 7.0 showed the promise of these techniques. Extending and evaluating our methodology for complex queries is necessary. It also remains an interesting question as to how the specific structure of statistics and the knowledge of physical database design can be exploited to increase efficiency of our techniques.

10. Acknowledgements

We thank Atul Adya, Sanjay Agrawal, Luis Gravano and the anonymous referees for their valuable feedback.

11. References

[1] The AutoAdmin Project, Microsoft Research. http://research.microsoft.com/db/AutoAdmin.

[2] Choenni S., Blanken H. M., Chang T., “Index Selection in Relational Databases”, Proceedings of IEEE ICCI 1993.

[3] Chaudhuri S., Motwani R., Narasayya V. “Random Sampling For Histogram Construction: How much is enough?” Proc. of ACM SIGMOD, 1998.

[4] Chaudhuri, S., Narasayya, V., "An Efficient, Cost-Driven Index Selection Tool for Microsoft SQL Server." Proc. of the 23rd VLDB Conference Athens, Greece, 1997.

[5] Chaudhuri, S., Narasayya, V., "Automating Statistics Management for Query Optimizers". In preparation for submission to IEEE Transactions on Knowledge and Data Engineering.

[6] Finkelstein S, Schkolnick M, Tiberio P.“Physical Database Design for Relational Databases”, ACM TODS, March 1988.

[7] Frank M., Omiecinski E., Navathe S., “Adaptive and Automative Index Selection in RDBMS”, Proc. of EDBT 1992.

[8] Gibbons P.B., Matias Y., Poosala V. “Fast Incremental Maintenance of Approximate Histograms”. Proc. of the VLDB Conference, 1997.

[9] Haas P.J., Naughton J.F., Seshadri S., Stokes L. “Sampling-based estimation of the number of distinct values of an attribute”. Proc. of VLDB Conference, 1995.

[10] Ioannidis Y., Poosala V. “Balancing Histogram Optimality and Practicality for Query Result Size Estimation.” Proc. of ACM SIGMOD, 1995.

[11] Labio W.J., Quass D., Adelberg B., “Physical Database Design for Data Warehouses”, Proc. of ICDE 1997.

[12] Manku G.S., Rajagopalan S., Lindsay B., “Approximate Medians and other Quantiles in One Pass and with Limited Memory”. Proc. of ACM SIGMOD 1998.

[13] Poosala V., Ioannidis Y. “Selectivity Estimation Without the Attribute Value Independence Assumption”. Proc. of VLDB Conference, 1997.

[14] Poosala V., Ioannidis Y., Haas P., Shekita E. “Improved Histograms for Selectivity Estimation of Range Predicates”. Proceedings of the ACM SIGMOD, 1996.

[15] Slutz, D., Massive Stochastic Testing of SQL. Proc. of VLDB Conference, 1998.

[16] TPC. TPC Benchmark D. (Decision Support). Working Draft 6.0. August 1993.

[17] TPC-D Data Generation with Skew. Chaudhuri S., Narasayya V. January 1999. Available via anonymous ftp from ftp.research.microsoft.com/users/viveknar/tpcdskew