Automated Statistics Collection in DB2 UDB

A. Aboulnaga[*] P. Haas[*] M. Kandil[+] S. Lightstone[+] G. Lohman[*] V. Markl[*] I. Popivanov[+] V. Raman[*]

[*]IBM Almaden Research Center
650 Harry Road
San Jose, CA
USA

[+]IBM Toronto Development Lab
8200 Warden Avenue
Markham, ON
Canada

{aashraf, phaas, lohman, marklv, ravijay}@us.ibm.com, {mkandil, light, ivannp}@ca.ibm.com

Abstract

The use of inaccurate or outdated database statistics by the query optimizer in a relational DBMS often results in a poor choice of query execution plans and hence unacceptably long query processing times. Configuration and maintenance of these statistics has traditionally been a time-consuming manual operation, requiring that the database administrator (DBA) continually monitor query performance and data changes in order to determine when to refresh the statistics values and when and how to adjust the set of statistics that the DBMS maintains. In this paper we describe the new Automated Statistics Collection (ASC) component of IBM® DB2® Universal Database™ (DB2 UDB). This autonomic technology frees the DBA from the tedious task of manually supervising the collection and maintenance of database statistics. ASC monitors both the update-delete-insert (UDI) activities on the data as well as query feedback (QF), i.e., the results of the queries that are executed on the data. ASC uses these two sources of information to automatically decide which statistics to collect and when to collect them. This combination of UDI-driven and QF-driven autonomic processes ensures that the system can handle unforeseen queries while also ensuring good performance for frequent and important queries. We present the basic concepts, architecture, and key implementation details of ASC in DB2 UDB, and present a case study showing how the use of ASC can speed up a query workload by orders of magnitude without requiring any DBA intervention.

1. Introduction

Query optimizers employ database statistics to determine the best execution strategy for a query. This metadata usually includes the number of rows in a table, the number of distinct values for a column, the most frequent values in a column, and, for numeric data, the distribution of data values in a column (usually stored as a set of quantiles). The optimizer uses these statistics to compute the *cardinality* (i.e., number of rows processed) at each intermediate step of a query execution plan. Advanced optimizers also use joint statistics on groups of columns within a table in order to deal with possible correlations between column values.

The presence of inaccurate or outdated statistics causes the optimizer to inaccurately estimate the cardinalities and costs of the steps in a query plan, which can result in a poor choice of plan and lead to unacceptably long query processing times. Unfortunately, it is all too easy for the statistics in a DBMS to deteriorate over time. In general, database statistics are not incrementally updated during data manipulations such as insert, update, delete, and load, because such incremental maintenance is too expensive. Statistics for tables with high data change rates are therefore very likely to be out of date. Even if the statistics are refreshed frequently, they may still lead to inaccurate cost estimates if the configuration parameters for the statistics are not set properly. Examples of such parameters include the number of frequent values and the number of quantiles to maintain. These parameters depend heavily on the statistical properties of the data, which can change over time.

Previous commercial database systems have required the DBA to manually configure and schedule the collection and maintenance of statistics, a tedious and time-consuming task. In this paper we describe the new Automated Statistics Collection (ASC) component of DB2 UDB, which has been developed as part of a general effort to incorporate autonomic technology into DB2 UDB products [LLZ02, LSZ03]. The ASC subsystem frees the DBA from the burden of statistics management. ASC monitors update-delete-insert (UDI) activity on the data tables in order to detect outdated statistics. ASC also monitors query feedback (QF), i.e., the results of the queries that are executed on the data, in order to detect

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

and adjust for outdated or improperly configured statistics. Based on this information, ASC decides which statistics to gather, at what level of detail to gather them, and when to gather them, without requiring any DBA intervention.

The novel features of ASC include (1) the simultaneous use of both a "UDI-driven" autonomic process that monitors UDI activity (including load operations) on tables and a "QF-driven" feedback loop that monitors estimated and actual results of query executions, (2) methods for deciding if the data in a table has changed sufficiently to require a refresh of the statistics, (3) methods for deciding which statistics to gather and at what level of detail to gather them based on monitored query results, and (4) methods for scheduling statistics collection that combine and prioritize the recommendations from the UDI-driven and QF-driven analyses.

Neither a UDI-driven nor a QF-driven approach is sufficient by itself. UDI-driven approaches are proactive and therefore can handle unforeseen queries, but may not concentrate enough effort on maintaining statistics that are critical to the users' workload. QF-driven approaches are reactive and require some learning time, but focus on the most critical statistics, and hence use system resources very efficiently. ASC combines the strengths of both approaches, proactively collecting basic statistics on every table periodically so as to be prepared for queries that have not been anticipated, and reactively refining statistics as required by the workload so as to be well prepared for the most important queries.

The remainder of the paper is organized as follows: in Section 2 we describe the overall ASC architecture. Sections 3 and 4 focus respectively on the UDI-driven and QF-driven approaches to detection of outdated and inaccurate statistics. In Section 5 we describe how the ASC scheduler combines and prioritizes the recommendations from both the UDI-driven and the QF-driven autonomic components in order to schedule the actual statistics collection. Section 6 presents a case study using a realistic workload of queries on a database of car-accident records. Section 7 surveys related work. Section 8 presents conclusions and gives an outlook on future work.

2. Automated Statistics Collection

We first review some basic facts about the collection and use of statistics in DB2 UDB. We then describe the modifications to DB2 UDB that comprise the ASC component.

2.1 Statistics in DB2

DB2 UDB stores in the system catalog [IBM04] the statistics pertinent to each table, including overall properties of the table, detailed information about the columns in the table, and information about any indexes on columns of the table. The DB2 UDB optimizer uses the information in the catalog when selecting a query plan. Table 1 summarizes the statistical information used by the optimizer and the names of the tables in the DB2 UDB SYSTAT schema that store the information.

Table Name	Content
tables	number of rows in a table
columns	number of distinct values for that column
indexes	number of distinct index keys, clustering of the table with respect to. the index, physical properties of the index
coldist	quantiles and frequent values of a column
colgroups	distinct number of values for a group of columns

Table 1: DB2 Statistics

The DB2 RUNSTATS utility collects the statistics and populates the system catalog tables. RUNSTATS is executed on a per table basis, and for any given table the user can specify the specific columns and indexes on which statistics are to be created. For each table in a database schema, the system catalog records the most recent time that RUNSTATS has been executed on the table. The exact configuration parameters for RUNSTATS on each table (i.e., the set of columns on which to gather statistics, the number of quantiles and frequent values to collect for a column, the set of column-group statistics to maintain, etc.) are recorded in a *RUNSTATS profile*. RUNSTATS profiles are stored in the system catalog (in the SYSSTAT.PROFILE table) and can be modified through the RUNSTATS command and queried through SQL.

2.2 ASC Architecture

The ASC component introduces both a UDI-driven and a QF-driven autonomic process into the DB2 UDB system. The first process monitors table activity and recommends execution of RUNSTATS on a table whenever UDI or LOAD statements against this table have changed the data distribution so that the present statistics for that table are substantially outdated. The second process monitors query results on a table. The process modifies the RUNSTATS profile for the table and recommends execution of RUNSTATS whenever it detects either that configuration parameters have been set improperly or that the statistics are outdated. The scheduler component combines the output of these two processes and triggers the execution of RUNSTATS on appropriate sets of tables and at appropriate times. In general, the scheduler causes RUNSTATS to be executed on one or more tables during a *maintenance iteration* that is concentrated within a specified time period called a *maintenance window*. The

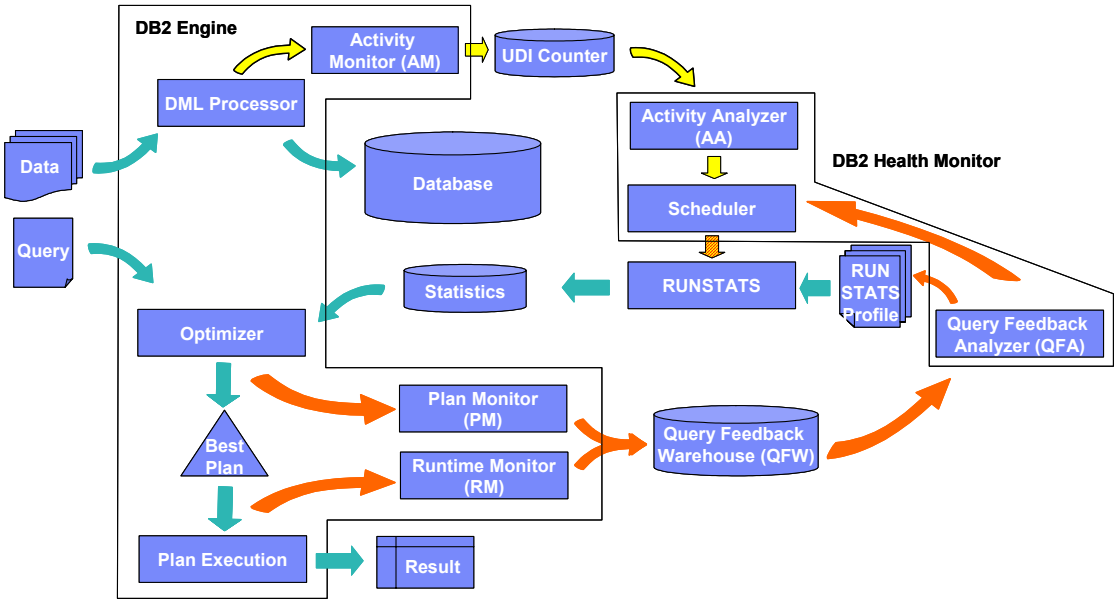

Figure 1: ASC Architecture

frequency and length of maintenance windows can be controlled by the DBA.

Figure 1 depicts the overall architecture of the ASC component. The left side of the figure depicts functionality that is implemented in the DB2 UDB engine, i.e., the query processor with optimizer and plan execution, the data manipulation language (DML) processor, and the monitors that facilitate several of the autonomic capabilities of DB2 UDB. The right side of the figure depicts a pair of analyzers and a scheduler that have been added to the DB2 Health Monitor to realize automated statistics collection. The analyzers periodically investigate the output of the monitors and recommend to the scheduler a set of tables on which to collect statistics.

The upper portion of the figure pertains to the UDI-driven autonomic process. When changing the data in a table according to a UDI or LOAD statement, the DML processor not only modifies the database, but also sends information to an *activity monitor* (AM) that records the number of changes against each table using a UDI-counter. The *activity analyzer* (AA) uses this information to determine whether statistics on an active table have changed enough to justify statistics collection for this table. The AA also estimates the degree to which activity on a table has altered the data distribution; the *scheduler* uses such estimates to prioritize tables for statistics collection. To avoid starvation, "critical" tables that have experienced UDI activity but have been ignored over many past maintenance iterations eventually receive top priority for statistics collection.

The lower portion of the figure pertains to the QF-driven autonomic process. This process observes query activity by using a *plan monitor* (PM), which stores the best plan together with the optimizer's cardinality estimate for each intermediate result. During plan execution, a *run-time monitor* (RM) observes the actual cardinalities. All of this compile-time and run-time information is stored in a *query feedback warehouse* (QFW) in the form of relational tables. A *query feedback analyzer* (QFA) periodically reviews these tables in order to generate modifications to the RUNSTATS profiles. The QFA bases these modifications on the discrepancy between actual and estimated cardinalities. Besides modifying RUNSTATS profiles, QFA communicates to the scheduler its findings about tables with modified RUNSTATS profiles and tables with outdated statistics, so that the scheduler can properly prioritize the automatic execution of RUNSTATS.

The statistics-collection process, like any other background maintenance task, must not significantly impede more important business-critical tasks. Therefore, the scheduler executes each RUNSTATS task as a "throttled" background process in order to guarantee that the user workload is not slowed down by more than a specified amount. During a maintenance window, RUNSTATS tasks are allocated a large portion of the available system resources. If there are tables that still need to be processed when the maintenance window ends, then processing continues, but RUNSTATS is throttled back so that the maximum allowable impact on query performance is limited to a small value (typically around 7%). When it is time to start the next maintenance window, any RUNSTATS tasks that are under way are first allowed to complete. To throttle the maintenance process, the scheduler exploits the general mechanism in DB2 UDB for adaptively tuning resource consumption during process execution [PRH+03a, PRH+03b]. This mechanism, which rests on control-theoretic techniques, is used to manage

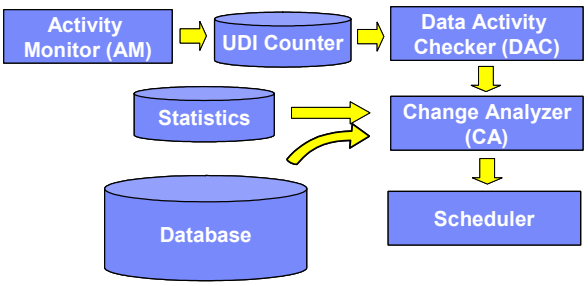

Figure 2: UDI-Driven Autonomic Process

other expensive maintenance processes such as database backup and table reorganization.

3. Detecting Stale Statistics via Data Activity

The UDI-driven autonomic process analyzes both the number of UDI and load operations and the changes in data values to determine whether the statistics on a table *T* have changed sufficiently so that statistics collection is justified. The process takes as input a list *G* of tables to be checked, as provided by the scheduler, and its output is a prioritized list of tables *D*, where *D* is a subset of *G*.

Figure 2 illustrates the overall detection process. As can be seen, the activity analyzer comprises two components. The *data activity checker* (DAC) is first executed to ensure that only tables with a reasonably large amount of data activity are considered for statistics collection. Each table in *G* that is not eliminated by the DAC is inserted into *D*. The list *D* is then passed to a *change analyzer* (CA). For each table *T* in *D*, the CA estimates for each "analyzable" column in *T* the degree of change in the data distribution since RUNSTATS was last executed on *T*; a column is analyzable if quantile statistics for the column are maintained in the system catalog. If no analyzable column in *T* evidences a significant degree of change, then *T* is removed from *D*.

After execution of the CA, the list *D* contains essentially only those tables having both significant data activity and significant changes in data values in at least one column. This list is then passed to the scheduler. We now describe the various components of the detection process in more detail.

3.1 Activity Monitor

The task of the activity monitor (AM) is to quantify the update activity for each table. It monitors both the loading of data into tables and UDI operations on tables. The AM maintains a UDI-counter for each table. The counter is increased by 1 whenever an existing row is updated or deleted, or a new row is inserted. The counter is set to 0 when the table is created, and is reset to 0 whenever RUNSTATS is executed on the table.

The UDI-counter is stored in the table descriptor together with other internal data structures. It is usually cached in memory and flushed to disk using the same discipline as for the rest of the data structures. Therefore, maintenance of the UDI-counter rarely causes extra I/O operations.

3.2 Data-Activity Checker

The DAC is the first process invoked when searching for outdated statistics because the presence of data activity is necessary in order for statistics to change. Lack of data activity means statistics need not be updated unless the QF-driven process gives a different indication, i.e., unless the QFA modifies the configuration parameters for some statistics or detects outdated statistics. This multi-tier approach significantly reduces the number of maintenance tasks performed over time. Tables with either low data activity or marginal changes to the statistics are ignored, so that system resources can be devoted to maintaining the most important tables.

The DAC first verifies that the table-related data structures are cached in memory. Their absence from the cache means that the table has not been used recently; it follows that the table has low data activity and can be ignored. Otherwise, the table is considered to be a candidate for statistics collection, and the DAC inspects the UDI-counter maintained for that table. If the UDI-counter suggests that at least τ% of the rows have been modified, this table is passed on to the change analyzer to further investigate whether statistics on this table need to be collected. The current implementation of DAC uses a value of $\tau = 10$.

It is possible that in some unusual cases a small number of records in a given table are changed, but the data values in these records are altered so drastically that query performance is affected. In this case, the table may not be detected by the DAC, and hence the AA. If this table is referenced in the query workload, however, then it will be detected by the QFA.

3.3 Change Analyzer

For each table *T* in its input list *D*, the CA takes a small sample from *T* and computes a synopsis data structure $S = S(T)$, where *S* comprises histograms of the marginal data

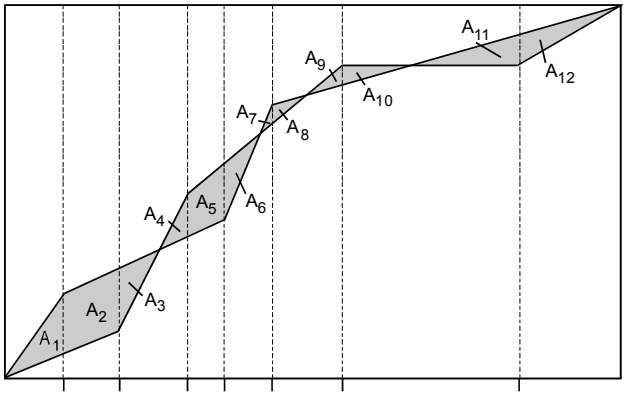

Figure 3: Computing the Change Value

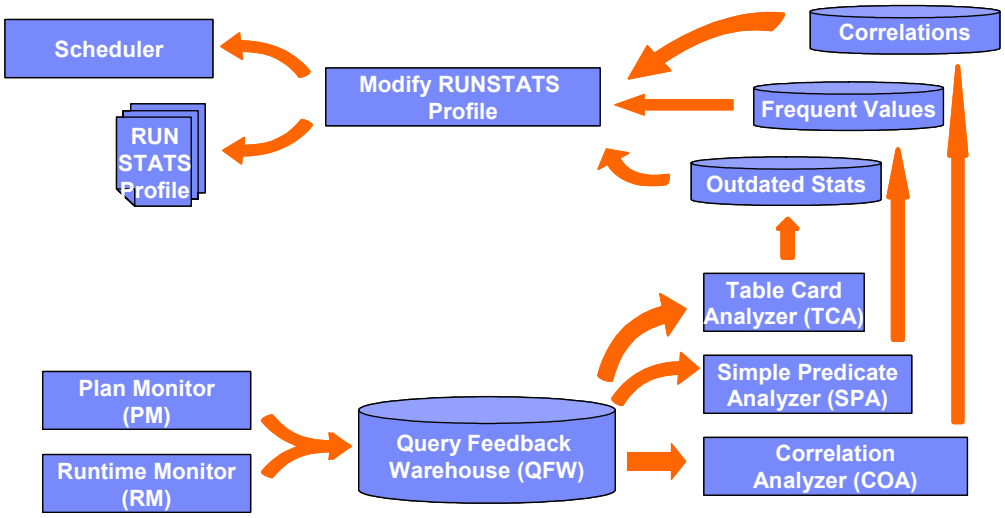

Figure 4: QF-driven Autonomic Process

distribution for each analyzable column. We have found that a sample consisting of about 2000 pages of table data, selected using page-level Bernoulli sampling, provides sufficient statistical precision for our purposes; see [PIHS96, IMHB04]. The CA also obtains an analogous synopsis $R = R(T)$ based on the (possibly outdated) marginal data distributions that are stored in the system catalog. For each analyzable column, the CA then measures the "distance" between the histograms. The CA deletes table T from D, i.e., declares the change in data values to be insignificant, if and only if the distance for each analyzable column lies below a specified threshold. If the change is significant for at least one analyzable column, then the CA leaves table T in D and, as described below, assigns to T a priority that the scheduler can use to determine when to update T relative to other tables.

For a fixed analyzable column $T.C$ (assumed to contain numeric data) the CA uses a normalized L_1 distance to measure the change in the data distribution. Specifically, denote by $e_Y(T.C \le v)$ the cardinality estimate for the predicate $T.C \le v$ (i.e., the estimated number of rows in T that satisfy the predicate) based on synopsis Y, and by l and u the smallest and largest bucket boundary points that appear in R and S. Then

$$\text{change}(T.C, R, S) = \frac{1}{|u-l|}\int_l^u \left| e_R(T.C \le v) - e_S(T.C \le v) \right| dv \cdot$$

Observe that change($T.C,R,S$) can be interpreted as the average absolute discrepancy in cardinality estimates over a family of one-sided inequality predicates.

Suppose that the histogram of $T.C$ values is represented by a set of bucket boundaries (typically quantiles) in both synopses R and S. Then change($T.C,R,S$) can be computed in a simple manner using essentially a "line sweep" algorithm. Specifically, determine the union of the two sets of bucket boundaries, and observe that $e_R(T.C \le v)$ and $e_S(T.C \le v)$ are linear and nondecreasing functions of v over each subinterval defined by a pair of successive bucket boundary points. Thus, the integral $\int_I |e_R(T.C \le v) - e_S(T.C \le v)|\,dv$ can be represented as the area of the region that lies between two piecewise-linear curves; see, for example, the shaded region in Figure 3, where the dashed lines correspond to the combined bucket boundaries. This area can in turn be expressed as a sum of areas of simple trapezoids and triangles, each of which is quick and easy to compute. Summing these areas and dividing by $l-u$ yields the value of change($T.C$, R, S) .

If change($T.C,R,S$) > θ for at least one column, where θ is an empirically determined threshold value, then the CA concludes that data distribution has changed, identifies table T as a candidate for statistics collection, and assigns to T a priority equal to $\max_C$ change($T.C,R,S$).

The CA can also use the foregoing measurement technique to quantify the change in data values as measured by successive sets of catalog statistics. Dividing this change value by the amount of time between the corresponding executions of RUNSTATS yields an estimate of the data change rate. As described in Section 5, the scheduler uses such rate-of-change estimates to project the next time at which a table will be due for statistics maintenance.

4. Detecting Poor Statistics from Queries

The QF-driven autonomic process monitors query execution and records estimation errors in the QFW. The QFA analyzes the data in the QFW to determine which tables have outdated statistics, whether and how the frequent values for columns on a particular table should be reconfigured, and which (intra-table) correlation

statistics should be created in order to reduce estimation errors in the future. As shown in Figure 4, the QFA comprises three components. The *table cardinality analyzer* (TCA) detects whether statistics are outdated by comparing the estimated and actual size of a table. The *simple-predicate analyzer* (SPA) uses estimated and actual cardinalities of simple equality predicates to determine the number of frequent values that should be used when creating the statistics for a particular column. The *correlation analyzer* (COA) uses cardinality information about tables, simple equality predicates, and conjunctive predicates to determine the set of column-group statistics to recommend to the scheduler. The output of the QFA is a prioritized list of tables Q that require statistics collection, along with the configuration parameter changes for the statistics of each table. The list Q is sent to the scheduler, and the configuration changes are stored in the RUNSTATS profiles.

4.1 The QFW and Its Maintenance

The QFW (see Figure 5) is populated periodically using the information generated by the PM and the RM. For each query, the PM records, at compile time, the predicates in the query (i.e., the column names, relational operators, and values) along with the optimizer's cardinality estimate for each predicate. The RM records run-time information about each query that includes the actual cardinalities for each table and predicate, as well as the actual values of parameter markers or host variables used in a query.

The data in the QFW is organized into relational tables. The *feedback query table* stores each query in its entirety, along with a skeleton query plan.

The *feedback predicate table* stores detailed predicate information. In our current implementation, the QFW stores information for simple predicates of the form `COLUMN` $\oplus$ `'literal'` (where $\oplus$ is a relational operator such as "=" or "<"), as well as compound predicates that reference a single table and are conjunctions of simple predicates. During the planning and processing of a query containing a compound predicate that comprises $N \geq 1$ "Boolean factors" (i.e., conjuncts), the PM and RM have the opportunity to observe actual and estimated cardinalities for one or more "sub-predicates," each consisting of the conjunction of a subset of the N Boolean factors. Each such sub-predicate generates an entry in the feedback predicate table that includes the table referenced by the sub-predicate, the number of Boolean factors, and the estimated and observed cardinality.

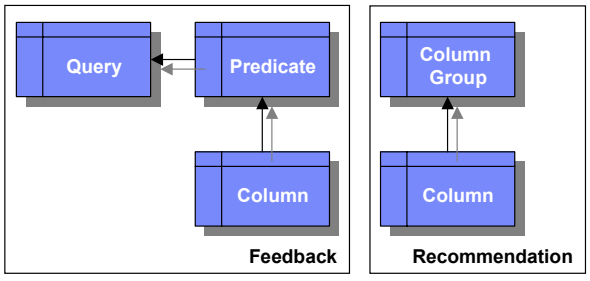

Figure 5: Tables in the QFW

The *feedback column table* contains an entry for each Boolean factor that appears in the feedback predicate table. Each entry includes the column name, relational operator, and literal of the predicate. The literal may come from either PM (in case of hard-coded predicates) or RM (in case of parameter markers or host variables).

The recommendations of the QFA concerning out-dated statistics, frequent values, and correlations are also stored in the QFW. The *recommendation column table* contains column information for these recommendations, i.e., the column name and number of frequent values. The *recommendation column-group* table stores similar information but for column groups rather than individual columns.

The QFW is an autonomic component of DB2 UDB in its own right. It automatically purges old data, when necessary, and it never grows beyond a DBA-specified size.

4.2 Operation of the QFA

The QFA processes the query feedback stored in the QFW and generates recommendations for correcting cardinality estimation errors in the future. The QFA proceeds by measuring, classifying, aggregating, and prioritizing the differences between optimizer-based cardinality estimates and actual cardinalities. Cardinalities considered include those for table size, for simple equality predicates of the form `COLUMN = 'literal'`, and for pairwise conjuncts of simple equality predicates. The QFA determines the cause of each estimation error by sequentially executing the table cardinality analyzer, the simple-predicate analyzer, and then the correlation analyzer. The QFA then aggregates the errors for each column and table, prioritizes the tables, and communicates its results to the scheduler and to the RUNSTATS profiles. We describe each of these operational phases in more detail below.

4.2.1 Table Cardinality Analyzer

The TCA simply compares the actual cardinality of each table in the feedback warehouse with the estimated cardinality based on the system catalog statistics. A discrepancy indicates that the statistics for this table are out of date. (This analysis is similar in spirit to the use of the UDI-counter by the DAC.)

4.2.2 Simple-Predicate Analyzer

For each column represented in the QFW, the SPA examines the errors in the simple equality predicates that reference the column to check whether the number of frequent values maintained for the column in the system

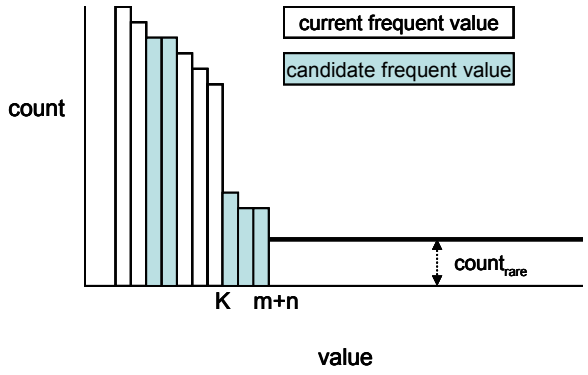

Figure 6: Frequencies Used by SPA

catalog is sufficient. If not, then the SPA automatically recommends an appropriate number of frequent values to maintain. Note that following such a recommendation also results in bringing the frequent-value statistics up to date. Because some of the statistics in the catalog are collected using random-sampling techniques, the QFA considers only those QFW entries where the observed error exceeds the expected error from normal sampling fluctuations.

Use of frequent-value statistics minimizes estimation errors arising from skew in the column-value frequencies. It is difficult, however, for a DBA to manually determine the "right" number of frequent values to track. The automated approach used by the SPA is as follows. First the SPA scans the QFW and the system catalog to compile a list of all "known" value frequencies for the column. These include:

- The frequencies fv_1 .. fv_n of the currently maintained frequent values, as recorded in the system catalog.
- The frequencies cfv_1 … cfv_m of all values for which there is a relevant error record in the QFW. These values can be considered as candidate frequent values to maintain.
- An average frequency assigned to each of the remaining "rare" (i.e., infrequent) values, computed using a uniformity assumption from the estimated number of rows in the table and the number of distinct values in the column.

When multiple frequency estimates are available for a given column value, the SPA uses the most recent one.

Figure 6 illustrates the frequency list as a bar graph, in descending order of frequency. Suppose that the table has d distinct values in total, and a total cardinality of C. Then the successive bar heights are $f_1, f_2, \ldots f_{m+n}$, $count_{rare}$, $count_{rare}$, ... , $count_{rare}$ ($d - m - n$ times), where $f_1, f_2, \ldots, f_{m+n}$ is {fv_1, … fv_n, cfv_1, ... cfv_m} arranged in descending order, and

$$count_{rare} = (C - f_1 - f_2 - \cdots - f_{m+n})/(d - m - n).$$

SPA now determines the number K of frequent values to maintain, where $n \le K \le m + n$. If DB2 UDB maintains K frequent values, then, when estimating cardinalities, the optimizer uses the exact count for these values and an average count of

$$newcount_{rare} = \left(C - \sum_{i=1}^{K} f_i\right) \Big/ (d - K)$$

for each of the remaining values. The total absolute estimation error over all possible simple equality predicates is

$$AbsError(K) = \sum_{i=K+1}^{m+n} |f_i - newcount_{rare}| + (d - m - n)\,|count_{rare} - newcount_{rare}|.$$

The first term represents the contribution due to the $m + n - K$ known frequencies that DB2 UDB chooses not to retain, and the second term is the contribution from the remaining values. Observe that $AbsError(K)$ is decreasing in K. To determine the number of frequent values to maintain, we initially set $K = n$ and then increase the value of K until either $AbsError(K)$ falls below a specified threshold or $K = \min(m + n, \beta)$, where β is a pre-specified upper bound on the number of frequent values to maintain.

```
// G, P, D, Q, C are lists of tables
// T is a table
G := tables to be checked by AA during the initial
        maintenance iteration
P, D, Q, C := {}
while(true)
{
   // Call the AA on the Tables in G
   D := AA(G);
   // Call the Query Feedback Analyzer
   Q := QFA();
  // prioritize D and Q based on the ranking criteria
  //  and merge with list of critical tables C
   P := prioritizeMerge(D, Q, C);
    while (still time in maintenance window)
   {
        T := Pop(P); // T is table in P with highest priority
        execute RUNSTATS on T
                and estimate the data change rate;
   }
   // Construct list for next maintenance interval
   (G, C) := constructDueTables()
   sleep until the next maintenance window;
}
```

Figure 7: Scheduling Algorithm

4.2.3 Correlation Analyzer

The COA focuses on pairwise correlations between columns in a table, because experiments indicate that the marginal benefit of correcting for higher-order correlations is relatively small; see [IMHB04]. For each pair of columns that appear jointly in a QFW record, the COA compares the actual selectivity of each conjunctive

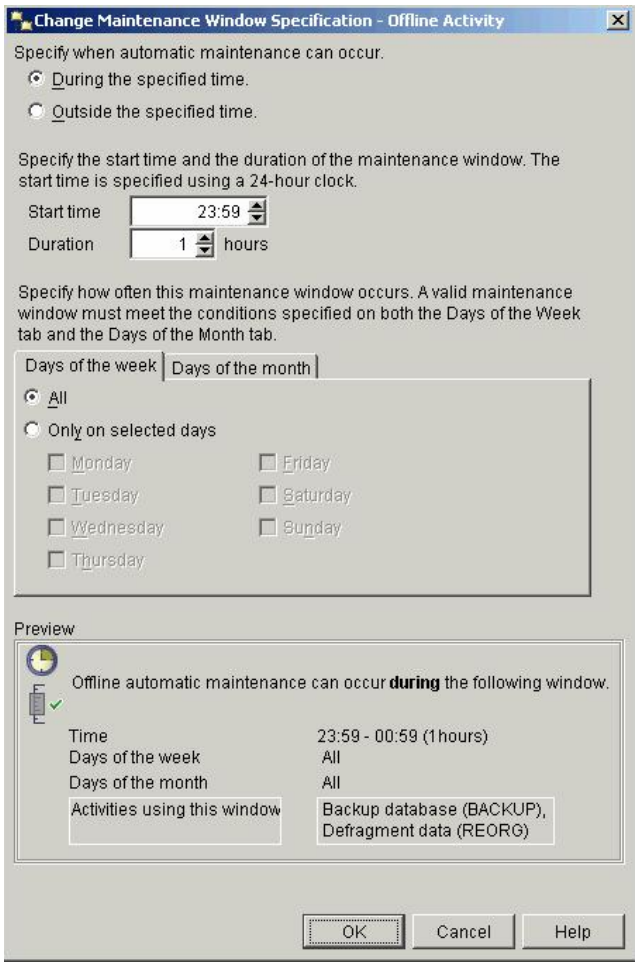

Figure 8: Specifying the Maintenance Window

predicate to the product of the actual selectivity of the Boolean factors of the conjunct, assuming that this information is available. For example, suppose that simple equality predicates p_1 and p_2 are evaluated while processing a query, along with the conjunctive predicate $p_1 \wedge p_2$. Denote by α_1, α_2, and α_{12} cardinalities for these queries that are observed during execution of the query, and denote by m the cardinality of the entire table. Then the COA deems the independence assumption to be valid if and only if

$$1-\Theta \le \frac{\alpha_{12} m}{\alpha_1 \alpha_2} \le 1+\Theta,$$

where $\Theta \in (0,1)$ is a small pre-specified parameter. Otherwise, the COA declares that a correlation error of absolute magnitude $\left|\alpha_{12} - (\alpha_1 \alpha_2 / m)\right|$ has occurred.

The analysis becomes more complicated when one or more of the actual cardinalities are not available, as is often the case in practice. The COA deals with the problem by estimating the missing information and adjusting the error-detection threshold and estimate of the error magnitude accordingly. Details of the complete algorithm will appear in a forthcoming paper.

4.2.4 Synthesizing the Final Outputs

The QFA processes feedback records as described above, grouped either by column name or, for records involving column pairs, by *column-group identifier*, where a column-group identifier comprises the pair of column names enumerated in lexicographic order. The QFA then sums up the absolute errors for each column and column group, and records the column-wise or group-wise error in the appropriate recommendation table. Next, the QFA identifies those columns and column groups that are responsible for the most severe errors. QFA modifies the RUNSTATS profiles so that RUNSTATS will increase the number of frequent-value statistics for each identified column and create joint statistics for each identified column group when it is next executed on the table that contains the column or column group. Finally, the QFA computes the total error for each table by combining the errors for table cardinality, cardinality of simple predicates, and cardinality of pairwise conjunctive predicates, weighing each error by its frequency (number of queries experiencing this error as stored in the QFW). Based on these table-wise errors, the QFA sends to the scheduler a list Q of tables on which to execute RUNSTATS.

5. Scheduling the Collection of Statistics

The scheduler drives the statistics-collection process. During periodic maintenance iterations (with corresponding maintenance windows), the scheduler invokes the AA and QFA, and combines the output D of the AA and the output Q of the QFA to create a combined prioritized list P of tables to be processed. The scheduler also invokes RUNSTATS as a throttled background process to collect statistics on those tables having the highest priority. Figure 7 displays the overall scheduling algorithm. As can be seen from Figure 7, the *prioritizeMerge* and *constructDueTables* procedures form the heart of the scheduling algorithm. We discuss these procedures in the following subsections.

The DBA can control the behavior of autonomic background activities by configuring the scheduler. For example, the DBA can limit the scope of automated statistics collection to certain tables, or can exclude certain tables from automatic maintenance. The DBA can also specify the maintenance window. Finally, the DBA can also control whether the scheduler should invoke QFA, AA, or both, and specify the maximum allowable disk space for the QFW. Figure 8 shows the GUI for specification of the maintenance window.

5.1 Prioritizing Tables for Processing

Prioritizing tables for processing is an important and challenging task. For large databases with potentially

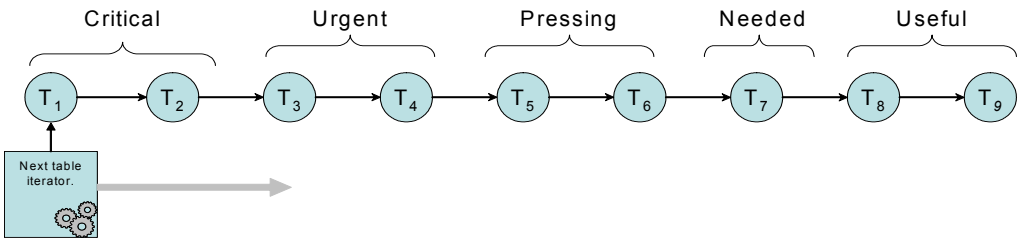

Figure 9: Priority Queue for Scheduler

thousands of tables and terabytes of data, selecting the wrong tables for statistics collection might mean that very needy tables will have to wait an unreasonable length of time, with detrimental effects on query performance.

The scheduler classifies tables into five distinct "urgency" classes. A table is *useful* with respect to statistics refresh if more than 0% but less than 50% of the rows have experienced some data change since the last statistics refresh on the table. A table is *needed* if it has been recommended for processing by the QFA. A *pressing* table has had 50% or more rows experiencing change since the last statistics refresh. An *urgent* table is both needed and either pressing or useful. A *critical* table is a table that has been starved: either the UDI-counter is positive but an excessive number of maintenance iterations have passed since the last statistics refresh, or RUNSTATS has never been executed on the table. Critical tables are always inserted into the list *P* of tables to be processed in the current maintenance window and are given top priority in this list. If a table falls into multiple classes, then the most urgent of the classes defines the table's final categorization.

The scheduler prioritizes critical tables above urgent tables, urgent tables above pressing tables, and so forth. The tables are then prioritized within each class, resulting in a priority queue that specifies the order in which tables are selected for statistics refresh; see Figure 9.

Useful tables are prioritized within their class by the percentage of rows changed, and similarly for pressing tables. Tables within both the needed and urgent classes are prioritized by a combination of their *frequency count* and *aggregated estimation error*. The frequency count of a table is the number of error records in the QFW that reference the table, and measures a table's relative importance within the workload. Aggregated estimation error is the table-wise error that is computed by the QFA, as described in Section 4.2.4. Finally, critical tables are ranked by their data change rate, as defined in Section 5.2 below; tables with no rate-of-change information (because RUNSTATS has been executed on the table less than two times) receive top priority. This ranking scheme ensures that a single table never appears more than once in the queue.

The rationale for the ranking scheme is as follows. It is useful to refresh statistics on tables that experience low to moderate data change, but which have not been detected by QFA as impacting the workload, in case these tables are accessed by the workload in the future. Such refresh activity should be subject to preemption by more important tasks. Tables that are known to be accessed by the workload and have obsolete statistics clearly need a statistics update. Tables that have experienced massive data change will almost surely cause massive query optimization problems if their statistics are not refreshed, and are also likely to show up in the workload, so that there is a pressing need for a statistics update. If such tables have actually shown up in the workload and generated significant estimation errors, then processing these tables becomes even more urgent. Finally, we allow the scheduler to identify tables as critical in order to avoid starvation problems in which tables experience UDI operations or lack statistics altogether, but are deferred indefinitely.

5.2 Constructing the List of Due Tables

After RUNSTATS has been executed on a table *T*, the newly collected statistics *N* for *T* are stored in the system catalog. The scheduler now invokes the CA to estimate the rate of change of the statistics, using *N* and the previous set of statistics *R* for *T*. (See the discussion at the end of Section 3.3.) Based on this rate of change, the scheduler determines the next maintenance iteration at which *T* will be due for consideration by AA.

Prior to the first maintenance iteration, the list *G* of input tables to the AA is initialized to contain all of the tables in the database that are subject to automatic statistics collection. (Recall that the DBA can limit the scope of ASC to a subset of the tables.) At the end of each maintenance window, the *constructDueTables* procedure (see Figure 7) is invoked to create the list *G* of tables that are due to be checked by AA in the next iteration. This function also constructs the list *C* of tables that are now critical (as previously defined).

6. A Case Study using a DMV Database

We illustrate the effect of ASC on query processing using a case study on a database that stores information about car accidents in various countries. The statistics maintenance is completely controlled by ASC: no statistics are configured or collected by the DBA at any

time. The study consists of running and timing a typical workload of 11 reporting queries on the database both before and after the execution of the various components of ASC. Specifically, the case study consists of running the workload after each of the following steps:

A. Initial loading of the database
B. Execution of ASC (AA only)
C. Insertion of additional accidents that occurred in Canada
D. Execution of ASC (AA only)
E. Execution of ASC (QFA only)

We refer to the set of queries executed after step A as "query-group A," the queries executed after step B as "query-group B," and so forth. We modified the operation of ASC in steps D and E above in order to investigate the benefits of QFA over and above those of AA.

We carried out the case study using a one-CPU 333 Mhz PowerPC 604® system with 512 MB memory, two 8.5 GB disks, running AIX® 4.3.3. ML11. The DMV database has a size of 1.5 GB and consists of four major tables: ACCIDENTS, CAR, OWNERS, and owner DEMO-GRAPHICS. These tables use the following schema:

- *Car* (ID, Make, Model, Color, Year, OwnerID*)*
- *Owner* (ID, Name, City, State, Country1, Country2, Country3)
- *Demographics* (ID, Age, Salary, Assets, OwnerID)
- *Accident* (ID, Year, SeatBeltOn, With, Driver, Damage, CarID)

Of particular interest are the column pairs (COUNTRY3, CITY) in the owner table and (MAKE, MODEL) in the car table. The columns in each pair are related by a functional dependency, and each pair is referenced in queries 10 and 11 via predicates of the form `(MAKE = 'Honda') AND (MODEL = 'Accord') AND (CITY = 'Toronto') AND (COUNTRY3 = 'Canada')`. Figure 10 shows the improvement in workload performance after statistics have been generated by ASC. Query-group A was executed using default statistics, whereas query-group B was able to take advantage of detailed distribution statistics on the individual columns. Note that whereas most queries experience a performance benefit, query 10 experiences a major regression. The reason behind this regression is that query 10 contains several predicates that reference correlated columns.

Figure 11 shows query performance after inserting further accidents for Canada into the database. Note that the results in Figure 10 and Figure 11 are not comparable because the queries used to obtain Figure 11 are executed against a larger data set.

Query-group C in Figure 11 uses the outdated statistics collected prior to the insertions. Query-group D uses updated statistics, but with no changes to the previous statistics configuration. Query-group E uses

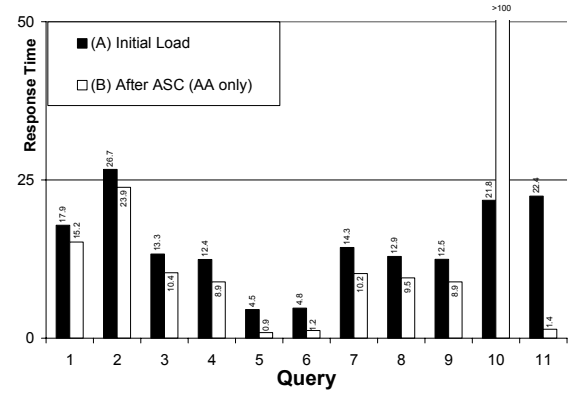

Figure 10: Performance after Loading

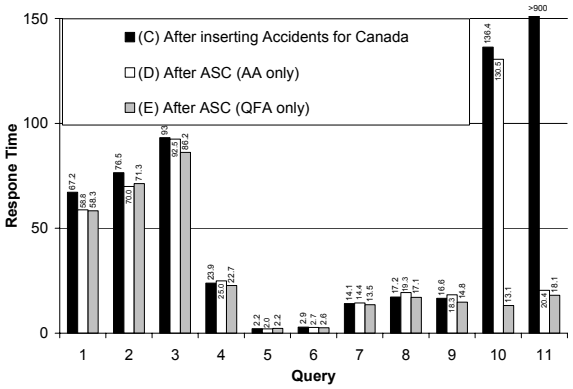

Figure 11: Performance after Inserting Additional Accident Records

statistics that are both updated and reconfigured. The correlations that caused the regression for query 10 in Figure 10 are detected when the QFA is executed at Step E, appropriate column-group statistics are collected, and the performance of query 10 improves dramatically. The performance improvements displayed in Figure 11 for queries 10 and 11 illustrate the strength of combining the UDI-driven and the QF-driven approaches: the orders-of-magnitude speedup of query 10 results primarily from the collection of column-group statistics by the QFA, whereas query 11 benefits mostly from the updating of single-column distribution statistics triggered by the execution of the AA at Step D.

Overall, the case study demonstrates the effectiveness of the autonomic technology in DB2 UDB. The use of ASC resulted in orders-of-magnitude speedups without requiring any intervention by a DBA. Even though autonomic features do not come for free, the overhead of ASC is negligible. In our case study, the use of ASC increased query execution times by 1-2%, primarily because of monitoring overhead. As the analyzers are executed during off-peak times in throttled background processes, their overhead did not really impact the operational DBMS. For example, at a time when the DBMS had previously processed 20,000 queries, the QFA

required less than a minute to analyze the QFW and recommend statistics. Both the performance benefit and the ease of use of this autonomic feature thus easily justify this overhead.

7. Related Work

There have been five major approaches toward automating statistics maintenance in database systems: UDI-driven change detection, static query-workload analysis, QF-driven maintenance, mining-type approaches, and piggybacking.

Several industrial products have delivered statistics-refresh automation features based on detection of UDI operations. These include DB2 UDB for the iSeries™ server [IBM02], Microsoft® SQL Server [MICR04], and Oracle 10g [ATLB03]. These products essentially automate statistics refresh on all tables where the percentage of UDI operations exceeds a threshold. Our approach provides three major extensions to this technology. First, we combine UDI measurement with a histogram analysis to reduce maintenance overhead. Secondly, we combine the UDI-driven process with a QF-driven process to provide a solution that is at once both proactive, preparing the system for unforeseen queries, and reactive to problems with the current system workload. Finally, we provide a prioritization scheme to rank tables so that, within a reasonable time interval, multiple tables can have their statistics refreshed, and the maintenance effort is concentrated on the most important tables.

Static query-workload analysis is based solely on the form of the queries and does not exploit run-time feedback. Primary examples of this approach are given by the SQL Server technique described in Chaudhuri and Narasayya [CN01] and the work of Bruno and Chaudhuri on SITS [BC02]. Both of these techniques analyze the query workload in order to select a set of statistics to maintain, such as multidimensional histograms on base data or query expressions. In contrast, our approach exploits run-time feedback and focuses on very simple statistics that are quick and easy to collect, maintain, and exploit. Although our statistics are relatively simple, we can effectively detect and model correlations between columns.

Use of QF-driven techniques in DB2 UDB was originally described in [SLMK01]. The proposed approach compares estimated and actual cardinalities to create adjustment factors that can be applied in the future to improve selectivity estimates. Our current work builds on these ideas by (1) adding the QFW mechanism for aggregating and prioritizing the feedback information, (2) adding the QF-driven methods for modifying the RUNSTATS profiles and recommending tables for processing, thereby improving selectivity estimates in a manner that does not require major modifications to existing query optimizers, and (3) integrating the QF-driven methods with a UDI-driven approach. Other QF-driven methods include the work in [AC99] and [BCG01], where query feedback is used to incrementally build a multidimensional histogram that can be used to estimate the selectivity of conjunctive predicates. Unlike the current work, these algorithms do not discover correlation between columns; the set of columns over which to build the histogram must be specified *a priori*.

Mining-type approaches attempt to discover correlated columns by systematically enumerating sets of potentially correlated columns and statistically analyzing the data. These techniques do not take into account the amount of change activity on the tables, and so are even more proactive than UDI-driven techniques. The CORDS system described in [IMHB04] exemplifies this approach. To make the detection efficient and scalable, CORDS applies candidate-pruning techniques together with random sampling. CORD can detect correlations between columns in the same or in different tables. Other proposed mining algorithms build sophisticated data synopses such as "probabilistic relational models," Markov-network-based histogram models, and Bayesian network models, which are then used to improve selectivity estimates. These latter techniques, as they currently stand, do not appear to scale well to very large databases, however, which limits their potential use in commercial systems; see [IMHB04] for a more detailed discussion. Lim et al. [LWV03] propose a QF-driven variant of the synopsis approach, called SASH, but this technique also suffers from scalability problems. We note that mining-type techniques such as CORDS (as well as static query-workload analysis) can potentially be used in conjunction with the methods described in the current paper.

Piggybacking was proposed as a technique for automated statistics collection by Zhu et al. [ZDS+98]. The idea is to collect statistics based on observing the data that is scanned during normal DML processing. Piggybacking avoids the asynchronous background refresh of table data statistics used by DB2 UDB for iSeries, DB2 UDB, Oracle 10g, and SQL Server. However, this technique suffers from a serious drawback. Although the overhead for any one SQL statement may be small, the cumulative overhead can be significant, and this adverse impact on query processing is present at all times. Our asynchronous approach to statistics refresh avoids these problems.

8. Conclusions

Our novel methodology for automating the collection of database statistics removes from the DBA the burden of manual statistics maintenance. The ASC learns which statistics are needed for good query performance and collects these statistics in background mode and at appropriate times, without requiring any DBA intervention. The two autonomic processes that comprise the ASC subsystem monitor UDI and query activity to

determine which statistics to collect and when to collect them, automatically determining the number of frequent values to maintain for each column and the appropriate set of column-group statistics to store in the system catalog. Statistics collection takes place as a throttled background process, ensuring minimal impact on mission-critical queries.

ASC is implemented in DB2 UDB v8.2. Our case study using the ACCIDENTS database has shown ASC to be effective in improving query performance over time, in some cases by orders of magnitude.

In future work, we plan to enhance QFA to also recommend the number of quantiles to maintain for a column, and perhaps to recommend more sophisticated column-group statistics such as limited bivariate histogram information. We are also exploring extensions of our techniques to column groups of order 3 and higher. We are also investigating approaches to directly use the query feedback to alter statistics as opposed to triggering RUNSTATS. Moreover, we are looking at ways of enhancing the UDI-driven and QF-driven techniques with mining-type methods such as CORDS [IMHB04]. A further interesting enhancement of autonomic function would be to automatically determine the maintenance intervals, perhaps by monitoring the number of critical tables in the system. Autonomic techniques for allocating and de-allocating CPU and disk resources would further enhance the technology described in this paper.

References

[AC99] A. Aboulnaga and S. Chaudhuri. Self-tuning histograms: Building histograms without looking at data. *Proc. 1999 ACM SIGMOD*, 181-192, June, 1999.

[ATLB03] M. Ault, M. Tumma, D. Liu, D. Burleson. *Oracle Database 10g New Features: Oracle10g Reference for Advanced Tuning and Administration*. Rampant TechPress, 2003.

[BCG01] N. Bruno, S. Chaudhuri, and L. Gravano. STHoles: a multidimensional workload-aware histogram. *Proc. 2001 ACM SIGMOD*, 211-222, June 2001.

[BC02] N.Bruno, S. Chaudhuri. Exploiting statistics on query expressions for optimization. *Proc. 2002 ACM SIGMOD*, 263-274, June, 2002.

[CN01] S. Chaudhuri, V. Narasayya. Automating statistics management for query optimizers. *IEEE Trans. Knowl. Data Engrg*., 13(1), 7-20, 2001.

[IBM02] DB2 Universal Database for iSeries - Database Performance and Query Optimization. IBM Corp., 2002.

[IBM04] DB2 v8.2 Performance Guide. IBM Corp., 2004.

[IMHB04] I. F. Ilyas, V. Markl, P. J. Haas, P. G. Brown, A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. *Proc. 2004 ACM SIGMOD*, June 2004. To appear.

[LLZ02] S. Lightstone, G. Lohman, D. Zilio. Toward autonomic computing with DB2 Universal Database. *SIGMOD Record*, 31(3), 2002.

[LSZ03] S. Lightstone, B. Schiefer, D. Zilio. Autonomic computing for relational databases: the ten year vision. *Proc. IEEE Workshop Autonomic Computing Principles and Architectures (AUCOPA '03)*, 2003.

[LWV03] L. Lim, M. Wang, and J. S. Vitter. SASH: A self-adaptive histogram set for dynamically changing workloads. *Proc. 29th VLDB*, 369-380, 2003.

[MICR04] SQL Server 2000 Books Online v8.00.02. Microsoft Corp., 2004.

[PIHS96] V. Poosala, Y. Ioannidis, P. Haas, E. Shekita. Improved histograms for selectivity estimation of range predicates. *Proc. 1996 ACM SIGMOD*, 294-305, June 1996.

[PRH+03a] S. Parekh, K. Rose, J. Hellerstein , S. Lightstone, M. Huras , V. Chang. *Managing the Performance Impact of Administrative Utilities*. IBM Research Report RC22864, IBM Corp., 2003

[PRH+03b] S. Parekh, K. Rose, J. Hellerstein, V. Chang, S. Lightstone, M. Huras. A general approach to policy-based management of the performance impact of administrative utilities. *Proc. 14th IFIP/IEEE Intl. Workshop Distributed Systems: Operations and Management (DSOM '03),* 20-22, October, 2003.

[SLMK01] M. Stillger, G. M. Lohman, V. Markl, M. Kandil. LEO - DB2's LEarning Optimizer. *Proc. 27th VLDB*, 19-28, 2001.

[ZDS+98] Q. Zhu, B. Dunkel, N. Soparkar, S. Chen, B. Schiefer, T. Lai. A piggyback method to collect statistics for query optimization in database management systems. *Proc. 1998 Conf. Centre for Advanced Studies on Collaborative Research (CASCON '98),* 25, 1998.

Further Information

Further up-to-date information about DB2 and IBM Data Management Solutions can be found at: http://www.software.ibm.com/data/