LEO: An autonomic query optimizer for DB2

by V. Markl
G. M. Lohman
V. Raman

Structured Query Language (SQL) has emerged as an industry standard for querying relational database management systems, largely because a user need only specify what data are wanted, not the details of how to access those data. A query optimizer uses a mathematical model of query execution to determine automatically the best way to access and process any given SQL query. This model is heavily dependent upon the optimizer's estimates for the number of rows that will result at each step of the query execution plan (QEP), especially for complex queries involving many predicates and/or operations. These estimates rely upon statistics on the database and modeling assumptions that may or may not be true for a given database. In this paper, we discuss an autonomic query optimizer that automatically self-validates its model without requiring any user interaction to repair incorrect statistics or cardinality estimates. By monitoring queries as they execute, the autonomic optimizer compares the optimizer's estimates with actual cardinalities at each step in a QEP, and computes adjustments to its estimates that may be used during future optimizations of similar queries. Moreover, the detection of estimation errors can also trigger reoptimization of a query in mid-execution. The autonomic refinement of the optimizer's model can result in a reduction of query execution time by orders of magnitude at negligible additional run-time cost. We discuss various research issues and practical considerations that were addressed during our implementation of a first prototype of LEO, a LEarning Optimizer for DB2® (Database 2™) that learns table access cardinalities and for future queries corrects the estimation error for simple predicates by adjusting the database statistics of DB2.

The remarkable growth of the relational database management systems (DBMS) industry over the last two decades can be largely attributed to the standardization of its Structured Query Language, SQL. SQL is a declarative language, that is, it requires the user to specify only *what* data are wanted, leaving to the *query optimizer* of the DBMS the difficult problem of deciding *how* best to access and process the data. For a given SQL query, there may be many different ways to access each table that is referenced, to join those tables, and, because the join operation is commutative, to order those joins and perform other operations necessary to complete the query. Hence, there may be hundreds or even thousands of possible ways to process a given query correctly. For example, suppose the SQL query is:

```
SELECT name, age, salary
FROM employees
```

```
WHERE age > 60 AND city = 'SAN JOSE' AND
    salary < 60,000
```

This query asks for the name, age, and salary of each employee who is over age 60, lives in San Jose, and makes less than $60,000 annually in salary. Each filtering condition in the WHERE clause that is joined by AND is called a *predicate*. Since this query references only one table, there are no choices of join order or join method, yet the optimizer still may consider many possible ways for the DBMS to process this simple query. It can always sequentially scan all the rows in the table and apply each predicate to each row. Or, if the appropriate indexes exist, it could exploit one or more indexes to access only the rows satisfying one or more of the predicates. For example, an index on age would permit accessing only those rows where the value of age is greater than 60 and then applying the other predicates (on city and salary). Alternatively, an index on city would limit the access to those rows having city equal to "San Jose" and subsequently applying the other predicates (on age and salary) to those rows retrieved. Alternatively, indexes on multiple columns, for example a combined index on city and age, or a combined index on city and salary, might be exploited, if they existed, or strategies combining any of the indexes discussed above (so-called "index ANDing"). Which strategy might be preferable depends upon the characteristics of the database, the availability of various indexes, and how selective each predicate is, that is, how many rows are satisfied by each condition.

Most modern query optimizers determine the best query execution plan (QEP, or simply *plan*) for executing an SQL query by mathematically modeling the execution cost for each of many alternative QEPs and choosing the one with the lowest estimated cost. The execution cost is largely dependent upon the number of rows that will be processed by each operator in the QEP, so the optimizer first estimates this incrementally as each predicate is applied. Estimating the *cardinality* (i.e., number of rows) of a result, after one or more of the predicates have been applied, has been the subject of much research for over 20 years.[1–5] To avoid accessing the database when optimizing queries, this estimate typically begins with statistics of database characteristics, specifically, the number of rows for each table. The cardinality of each intermediate result is derived incrementally by multiplying the base table's cardinality by a *filter factor*—or *selectivity*—for each predicate in the query, which is derived from statistics on the columns affected by that predicate, such as its number of distinct values or a histogram of its distribution. The selectivity of a predicate P effectively represents the probability that any row in the table will satisfy P, and that selectivity depends upon the characteristics of the database. For example, in the query above, the predicate on city might be quite selective if the database were a worldwide database of a large, multinational company, but it might be a lot less selective if the database contained all the employees of some small start-up firm centered in San Jose. For the latter case, the predicates on age and/or salary would be more selective. The optimizer would tend to favor QEPs that could exploit indexes to apply the most selective predicates and QEPs that utilize simple table scans if there were no indexes, or if the optimizer estimated that most employees would satisfy all of the predicates. In DB2* (Database 2*), the choice of QEP is based solely upon the detailed cost estimate for each of the competing plans, and not upon such simplistic heuristics.

When there are multiple tables in the FROM clause of the query, the number of alternative strategies considered by the optimizer increases exponentially, because the myriad choices mentioned above are compounded with additional decisions about the order in which tables are joined and the method by which they are joined. DB2, for example, supports three major types of join method, and there are several variants within each of these. For a two-table join with a handful of predicates, the DB2 optimizer might consider over a hundred different plans; for six tables, the number of plans could be well over a thousand! The DB2 optimizer considers all of these alternatives automatically for the user, who is not even aware that it is going on!

Although query optimizers do a remarkably good job of estimating both the cost and the cardinality of most queries, many assumptions underlie this mathematical model. Examples of these assumptions include: currency of information, uniformity, independence of predicates, and a principle of inclusion.

Currency of information: Updating the statistics each time a row is updated or deleted would create a locking bottleneck in the system catalogs, where statistics are stored. It is difficult or impossible to calculate some statistics incrementally, such as the number of distinct values for each column, and so it is common for statistics to be recomputed periodically as a user-invoked batch operation (called *RUNSTATS* in DB2). Despite this, the optimizer assumes that the statistics reflect the current state of the database, that is, that the database characteristics are relatively stable, and it relies upon the user to know when any table has changed enough to warrant the expensive recollection of statistics.

Uniformity: Although many products use histograms to deal with skew in the distribution of values for "*local*" selection predicates (on columns within a single table), we are unaware of any available product that exploits them for *join* predicates, that is, those relating columns in multiple tables. Thus for join predicates, the query optimizer still relies on the assumption of uniformity.

Independence of predicates: Selectivities for each predicate are calculated individually and multiplied together, essentially assuming the predicates are statistically independent of each other, even though the underlying columns may be related, for example by a functional dependency. While multidimensional histograms address this problem for local predicates, they have never been applied to join predicates, aggregation, and so on. Applications common today have hundreds of columns in each table and thousands of tables, making it impossible to know on which subset(s) of columns to maintain multivariate statistics.

Principle of inclusion: The selectivity for a join predicate $X.a = Y.b$ is typically defined to be $1/\max\{|a|, |b|\}$, where $|b|$ denotes the number of distinct values of column b. This implicitly assumes the "principle of inclusion," that is, that each value of the smaller domain has a match in the larger domain. Fortunately, this assumption is frequently true for the most common joins between a *primary key* to a table (e.g., a product number in the Products table) and a reference to that key (a *foreign key*) in another table (e.g., the Orders table).

When these assumptions are invalid, significant errors in the cardinality—and hence cost—estimates result, causing suboptimal plans to be chosen. From the authors' experience, the primary cause of major modeling errors is the cardinality estimate on which costs depend. Cost estimates might be off by 10 or 15 percent, at most, for a given cardinality, but cardinality estimates can be off by orders of magnitude when their underlying assumptions are invalid or uncertain. Although there has been considerable success in using histograms to detect and correct for data skew,[6–8] and in using sampling to gather up-to-date statistics,[9,10] there has been to date no comprehensive approach to correcting all modeling errors, regardless of origin.

In this paper we describe our approach toward autonomic query optimization for overcoming modeling errors and incorrect statistics, which has led to the prototype of a LEarning Optimizer (LEO) for DB2.[11] LEO learns from any modeling mistake, at any point in a QEP, by automatically validating its estimates against actual cardinalities for a query. Determining at what point in the plan the significant errors occurred then allows for reoptimizing the query at this point[12] and adjusting its model dynamically to better optimize future queries. Over time, this feedback method amasses experiential information that augments and adjusts the database statistics for the part of the database that enjoys the most user activity. Not only does this information enhance the quality of the optimizer's estimates, but it can also suggest where statistics gathering should be concentrated and can even supplant the need for statistics collection.

A learning optimizer

This section describes the architecture of LEO, an *autonomic optimizer* that observes actual query execution and uses actual cardinalities to autonomically validate and refine the estimates from its model and to reoptimize the current query, without requiring user intervention. In the following sections we discuss the two essential functions of LEO: deferred learning for future queries and progressive optimization of the query currently under execution.

Deferred feedback-based learning. Deferred learning exploits empirical results from actual executions of queries to validate the optimizer's model incrementally, deduce what part of the optimizer's model is in error, and compute adjustments to the optimizer's model for future query optimizations. Deferred learning with LEO works under the assumption that future queries will be similar to previous queries, that is, they will share one or several predicates. Our LEO prototype currently corrects the statistics for tables (which may be out of date) and estimates the selectivity of individual predicates in this way.

The LEO feedback loop is comprised of four steps, as seen in Figure 1: monitoring, analysis, feedback, and feedback exploitation. At query compilation time, the monitoring component saves the cardinality estimates derived by the optimizer for the best (i.e., least-cost) plan, and during query execution

Figure 1 Deferred learning

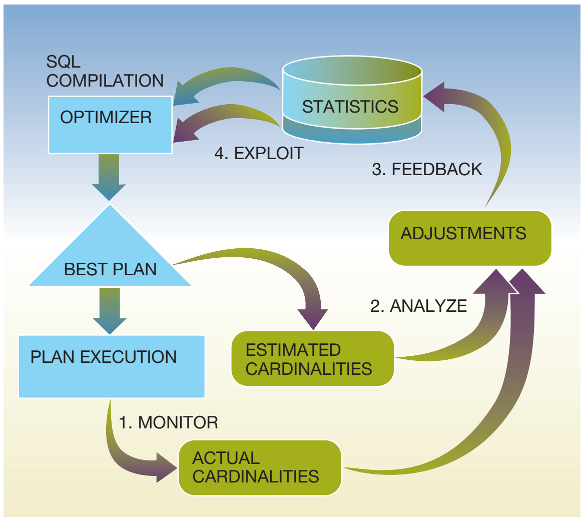
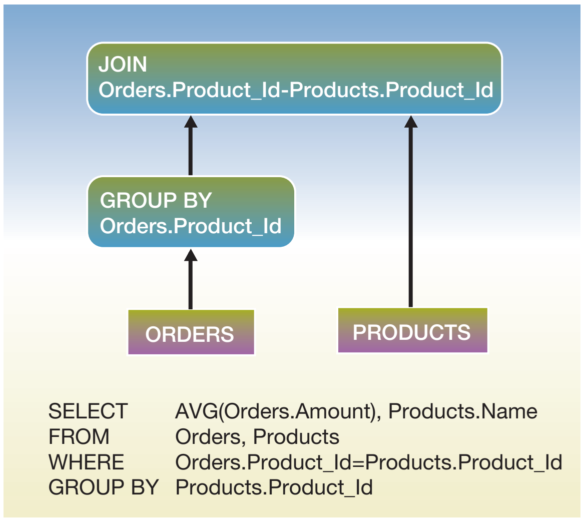


Figure 2 A query plan that can be reoptimized dynamically



```
SELECT     AVG(Orders.Amount), Products.Name
FROM       Orders, Products
WHERE      Orders.Product_Id=Products.Product_Id
GROUP BY   Products.Product_Id
```

saves the actual cardinalities observed for that plan. The analysis component uses the information thus learned to identify modeling errors and compute corrective adjustments. This analysis is a stand-alone process that may be run separately from the database server and even on another system. The feedback component modifies the catalog statistics of the database according to the learned information. The exploitation component closes the feedback loop by using the learned information in the system catalog to provide adjustments to the query optimizer's cardinality estimates.

The four components can operate independently, but form a consecutive sequence that constitutes a continuous learning mechanism by incrementally capturing plans, monitoring their execution, analyzing the monitor output, and computing adjustments to be used for future query compilations. This mechanism enables deferred learning, since only future queries will benefit from the feedback.

The deferred learning mechanism has been implemented in a prototype using DB2 Universal Database* (UDB) for Linux**, UNIX**, and Windows**. Experiments with the protype[18] showed the monitoring overhead to be below 4 percent of the total query execution time, whereas performance may improve by orders of magnitude, particularly when the optimizer learns that a bulk join method should be used instead of a nested-loop join, due to a large input cardinality.

Immediate feedback-based learning. The monitored cardinalities need not be used for subsequent queries alone. If the actual cardinalities are significantly different from the estimated cardinalities, the chosen query plan could be highly suboptimal. As part of the LEO project, we are currently investigating how to use this knowledge immediately by dynamically reoptimizing the current query and changing its execution plan, if all of the rows for an intermediate result are materialized before proceeding at any point in the plan.

Generally, response time and memory are optimized if each row is processed completely and returned to the user in a *pipelined plan*. But occasionally, the rows of an intermediate result must be fully materialized, either as a sorted or unsorted temporary table (TEMP), which we call a *materialization point*. TEMPs afford a natural opportunity to count the number of rows and possibly to reoptimize the plan before any rows are returned to the user, thereby avoiding returning duplicate rows that are caused by restarting the query. However, two important issues arise:

- Since reoptimization involves a cost, when is it worthwhile?
- How can we reoptimize efficiently?

We address the first question in the subsection, "When to reoptimize." For the second question, the easiest solution is to simply rerun the query "from scratch" under a new plan. However, this would waste all the (possibly substantial) work done up to the materialization point, which was saved in the TEMP. In most cases, it is preferable for the reoptimized plan to avoid having to redo that work by instead accessing that TEMP in the reoptimized plan.

For example, Figure 2 shows a query plan for a simple two-table join that groups/aggregates the Orders table by Product_Id before the join. The sort that may be needed to accomplish this aggregation must materialize its entire input before proceeding and thus constitutes a TEMP. Since most aggregations can be performed incrementally as the rows are sorted, the TEMP will, at its conclusion, contain the GROUP BY result. The optimal join algorithm (nested loop join, hash join, or merged join) for subsequently joining Orders and Products depends crucially on the size of this GROUP BY result. The query optimizer could choose a suboptimal join algorithm if it under- or over-estimates the size of this result.

However, during query execution, the optimizer can monitor the size of the GROUP BY result, and reoptimize in case of severe estimation errors, for example, by changing the join algorithm if needed. Such reoptimization becomes more complex for more elaborate query plans with multiple materialization points. Reference 12 suggests encapsulating TEMPs as tables and converting the remaining portion of the query plan after the TEMPs into an SQL query, which can then be resubmitted to the query optimizer. Unfortunately, this approach has two problems. First, it may not be optimal to reuse a TEMP as is. In cases where the size of the TEMP is much larger than expected, the optimal plan might be to reuse only a part of the TEMP, or even ignore the TEMP completely, in favor of a totally new plan that directly uses the base tables. Second, the remaining portion of the plan beyond the TEMP may not always be expressible as an SQL statement, especially if it contains update operations, which are fed from subqueries.

A better alternative is not to encapsulate the TEMPs, but instead to define them as *materialized views*[13] (known in DB2 as *Automatic Summary Tables* or ASTs[14]) and expose their definition to the query optimizer. The optimizer can then rely on standard view-matching techniques[14,15] to identify TEMPs that are worthwhile to reuse. The cost of reoptimization using additional materialized views is almost identical to the cost of optimizing the original query, since the optimizer only has to investigate one alternative intermediate table access method per materialized view.

Moreover, once TEMPs are defined as materialized views, there is no reason to limit their use to the current query only. All subsequent queries can potentially exploit materialized TEMPs, just as they exploit user-defined materialized views. Of course, this approach could lead to an avalanche of such views, so that the query engine would have to periodically delete rarely used ones; this is akin to the materialized view selection problem.[16]

Research issues in autonomic query optimization

Our initial prototype of LEO has uncovered a number of challenging research problems that require solutions for any practical application of the optimizer in a product. We now discuss these problems and possible approaches to their solution.

Stability and convergence. A cardinality model refined by feedback has to take incomplete information into account. While some cardinalities may be deduced from query feedback—constituting *hard facts*—others are derived from statistics and modeling assumptions—forming *uncertain knowledge*. The learning rate of the system is largely dependent on the workload and the accuracy of statistics and assumptions.

Assuming independence of predicates, when in fact the data are correlated, usually results in underestimation of the cardinalities of the intermediate results, which are used by the optimizer when determining the cost of a QEP. This underestimation will cause the optimizer to prefer a plan based on uncertain knowledge over one based on hard facts. The underestimation of cardinalities can result in a complete exploration of the search space; the system will converge only after trying out and learning about all QEPs that contain underestimation. Overestimation, however, may result in a local minimum (i.e., a suboptimal QEP); the optimizer will prefer other QEPs over a QEP with overestimates. Hence overestimates are unlikely to ever be discovered or corrected.

To reach a reasonable form of stability, the autonomic optimizer should initially use an exploratory mode, for example, before going into production. This mode will initially involve more risks by choosing promising QEPs based on uncertain knowledge, thus validating the model and gathering hard facts about data distribution and workload. A second operational mode will be biased toward QEPs that are based on experience. This mode favors QEPs based on hard facts over slightly cheaper QEPs based on uncertain knowledge. The transition between the modes would be gradual, resembling simulated annealing[17] methods in machine learning.

To overcome the local optimum caused by overestimation, it is necessary to explore uncertain knowledge used for presumably suboptimal, but promising QEPs, for example, by synchronous or asynchronous sampling.[10]

Detecting and exploiting correlation. In practical applications, data are often highly correlated. In a car database, for instance, the selectivity of the conjunction (*make* = "Honda" and *model* = "Accord") is *not* correctly derived by multiplying the individual selectivities of *make* = "Honda" and *model* = "Accord," because the columns *make* and *model* are correlated—only Honda makes an Accord model. Since correlation constitutes a violation of the independence assumption, selectivity estimates for predicates involving correlation can be off by orders of magnitude in state-of-the-art query optimizers. With our approach, we have the opportunity to detect and correct such errors.

Correlations pose many challenges. First, there are many types of correlation, ranging from functional dependencies between columns, especially referential integrity, to more subtle and complex cases, such as an application-specific constraint that an item is supplied by at most 20 suppliers. Second, correlations may involve more than two columns, and hence more than two predicates in a query, with subsets of those columns having varying degrees of correlation. Third, a single query can only provide evidence that two or more columns are correlated for specific values. For complex queries involving several predicates, isolating which subsets of predicates are correlated and the degree of correlation can be extremely difficult. Another difficult research problem is to generalize correlations from specific values to relationships between columns: How many different values from executing multiple queries having predicates on the same columns are required to safely conclude that those columns are, in general, correlated, and to what degree? Instead of waiting for that many queries to execute, correlation detection could instead identify promising combinations of columns—even from different tables—on which the statistics utility would then collect multidimensional histograms. In addition, the observed information can be used to pinpoint errors in the cardinality model, populate the database statistics, or to adjust the erroneous estimates by creating an additional layer of statistics.

When to reoptimize. As discussed in the subsection, "Immediate feedback-based learning," immediate learning can change the plan for a query at run time, when the actual cardinalities are significantly different from the estimated cardinalities. But the new plan could itself be quite expensive, if it cannot make use of prior TEMPs efficiently. The optimizer will find this out during reoptimization, but the cost of reoptimization could itself be significant. Therefore it is crucial to determine, *without reoptimizing*, when it will be worthwhile to reoptimize.

Reference 12 uses the difference between the estimated and actual cardinalities as a heuristic to determine whether to reoptimize. However the question is not how inaccurate the optimizer's estimate is; it is whether the plan is suboptimal under the new cardinalities and whether the cost difference is enough to pay for the reoptimization. One heuristic looks at the nature of the plan operators and decides whether a change in the input cardinality for an operator is likely to make the operator suboptimal. Alternatively, the optimizer can be enhanced to pick not only the optimal plan, but also the range of selectivities for each predicate within which the plan is optimal. This prediction of the sensitivity of any plan to any one parameter is extremely hard, because of nonlinearities in the cost model.

We also need to limit the number of reoptimization attempts for a single query, because the convergence problem of the subsection, "Stability and convergence" is even more serious here. We do not want the query execution to get into a long loop where it repeatedly tries out all alternative plans before making progress.

Learning other information. Learning and adapting to a dynamic environment is not restricted to cardinalities and selectivities. Using a feedback loop, many costs and parameters currently estimated by the optimizer can be made self-validating. For example, the dominant aspect of cost, the number of physical I/Os, is currently estimated probabilistically from estimated hit ratios, assuming each application gets an equal share of the buffer pool. The optimizer could validate these estimates by observing actual

I/Os, actual hit ratios, and/or actual times to access tables for a given plan. Another example is the maximum amount of memory allocated to perform a particular sort in a plan. If the DBMS detected by query feedback that a sort operation could not be performed in main memory, it could adjust the sort heap size to avoid external sorting for future sort operations.

Feedback is not limited to services and resources consumed by the DBMS, but also extends to the applications that the DBMS serves. For example, the DBMS could measure how many of the rows in a query's result are actually consumed by each application and optimize each query's performance for just that portion of the result, for example, by effectively appending the OPTIMIZE FOR <n> ROWS clause of SQL to that query. Similarly, feedback from executions could be used to automatically set many configuration parameters for shared resources that are currently set manually. Physical parameters such as the network rate, disk access time, and disk transfer rate are used to weight the contribution of these resources to plan costs, and are usually considered to be constant after an initial set-up. However, setting these parameters using measured values is more autonomic and more accurately captures the effective rate. In the same way, the allocation of memory among different buffer pools, the total sort heap, and so on, can be tuned automatically according to hit ratios that were recently observed.

Practical considerations

In the process of implementing LEO, several practical considerations also needed to be addressed.

The Hippocratic oath: "Do no harm!" The overall goal of an autonomic optimizer is to improve query performance by adjusting an existing model based upon previously executed queries. Ideally, this adjusted model provides a better decision basis for selecting the best execution plan for a query. However, this learned knowledge must be arrived at extremely conservatively: we should not make hasty conclusions based upon inconclusive or spotty data. In critical applications, stability and reliability of query processing are often favored over optimality with occasional unpredictable behavior. If adjustments are immediately taken into account for query optimization, even on a highly dynamic database, the same query may generate a different execution plan each time it is issued, and thus may result in thrashing of execution plans. This instability can be avoided if reoptimization of queries takes place after the learned knowledge has converged to a fixed point or has reached a defined threshold of reliability.

Consistency between statistics. DB2 collects statistics for base tables, columns, indexes, functions, and tablespaces, many of which are mutually interdependent. DB2 permits users to update the statistics in the catalogs and performs checks for inconsistencies in such updates. An autonomic optimizer must similarly ensure the consistency of these interdependent statistics when adjusting any of them. For example, the number of rows of a table determines the number of disk pages used for storing those rows. When adjusting the number of rows of a table, we must consequently ensure consistency with the number of pages of that table—for example, by adjusting this figure as well—or else plan choices may be biased, depending on which plan uses which statistic. Similarly, the consistency between index and table statistics has to be preserved, since there may be interdependencies between the number of distinct values of a column and the number of rows in a table. However, an increase in the number of rows will not always result in an increase in the number of distinct values: although subsequent inserts are likely to alter the number of distinct values for a *date* column, this is very unlikely for a column like *sex* that can only assume the values *male* or *female*, regardless of the number of rows.

Adjustments vs database statistics. An autonomic optimizer is not a replacement for database statistics, but rather a complement to them. Statistics are collected uniformly across the database, to prepare for any possible query. Feedback gives the greatest improvement to the modeling of queries that are either repetitive or are similar to earlier queries, that is, queries for which the optimizer's model exploits the same statistical information. Feedback extends the capabilities of the RUNSTATS utility by gathering information on derived tables (e.g., the result of several joins) and gathering more detailed information than RUNSTATS might. Over time, the optimizer's estimates will improve most in regions of the database that are queried most. However, for correctly estimating the cost of previously unanticipated queries, the statistics collected by RUNSTATS are necessary, even in the presence of query feedback.

Conclusions

Although today's query optimizers autonomically determine the best way to process a declarative SQL query (one which specifies only what data are wanted), they do so using a complex mathematical model having many inherent assumptions and parameters. The ideas on autonomic query optimization outlined in this paper have led to the implementation of LEO, DB2's LEarning Optimizer. By self-validating these assumptions and parameters using feedback garnered from earlier executions, LEO provides a major step forward in improving the quality of query optimization and reducing the need for "tuning" of problem queries, a major contributor to cost of ownership. Our current LEO prototype enables deferred learning of table access cardinalities and simple predicates,[18] demonstrating significant performance improvements and a low monitoring overhead of below 4 percent of the total query execution time.

Our future work includes completing the LEO prototype for deferred learning, aggregating and summarizing the observed information, finding conclusive ways to discern and generalize occurrences of correlation among predicates, measuring the benefit of using LEO on a realistic set of user queries, and extending LEO's approach to parameters other than cardinality. In addition, we are carrying out a prototype implementation of immediate learning in order to analyze and validate the performance of this progressive query optimization approach.

Cited references

1. P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price, "Access Path Selection in a Relational Database Management System," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Boston, MA, May 1979, ACM, New York (1979), pp. 23–24.
2. A. Van Gelder, "Multiple Join Size Estimation by Virtual Domains," *Proceedings of the Twelfth ACM Symposium on Principles of Database Systems* (May 1993), pp. 180–189.
3. A. N. Swami and K. B. Schiefer, "On the Estimation of Join Result Sizes," 4th International Conference on Extending Database Technology (March 1994), pp. 287–300.
4. R. Ahad, K. V. B. Rao, and D. McLeod, "On Estimating the Cardinality of the Projection of a Database Relation," *ACM Transactions on Database Systems* **14**, No. 1, pp. 28–40 (1989).
5. C. Lynch, "Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values," *Proceedings of the 14th International Conference on Very Large Databases* (August 1988), pp. 240–251.
6. Y. E. Ioannidis and S. Christodoulakis, "On the Propagation of Errors in the Size of Join Results," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Denver, CO, May 1991, ACM, New York (1991), pp. 268–277.
7. V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita, "Improved Histograms for Selectivity Estimation of Range Predicates," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Montreal, Canada, June 1996, ACM, New York (1996), pp. 294–305.
8. V. Poosala and Y. Ioannidis, "Selectivity Estimation Without the Attribute Value Independence Assumption," *Proceedings of the 23rd International Conference on Very Large Databases* (VLDB 1997).
9. P. Haas, J. Naughton, S. Seshadri, and A. Swami, *Selectivity and Cost Estimation for Joins Based on Random Sampling*, Research Report RJ-9577, IBM Thomas J. Watson Research Center, Yorktown Heights, NY 10598 (1993).
10. T. Urhan, M. J. Franklin, and L. Amsaleg, "Cost-Based Query Scrambling for Initial Delays," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Seattle, WA, June 1998, ACM, New York (1998), pp. 130–141.
11. M. Stillger, G. Lohman, V. Markl, and M. Kandil, "LEO—DB2's Learning Optimizer," *Proceedings of the 27th International Conference on Very Large Databases* (September 2001), pp. 19–28.
12. N. Kabra and D. DeWitt, "Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans," *Proceedings of the ACM SIGMOD International Conference on Management of Data* (June 1998), pp. 106–117.
13. N. Roussopoulos, "Materialized Views and Data Warehouses," *SIGMOD Record* **27**, No. 1, 21–26, ACM, New York (1998).
14. M. Zaharioudakis, R. Cochrane, G. Lapis, H. Pirahesh, and M. Urata, "Answering Complex SQL Queries Using Automatic Summary Tables," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Dallas, TX, May 2000, ACM, New York (2000), pp. 105–116.
15. S. Chaudhuri, R. Krishnamurthy, S. Potamianos, and K. Shim, "Optimizing Queries with Materialized Views," *Proceedings of the Eleventh International Conference on Data Engineering* (March 1995), pp. 190–220.
16. R. Chirkova, A. Y. Halevy, and D. Suciu, "A Formal Perspective on the View Selection Problem," *Proceedings of the 27th International Conference on Very Large Databases* (September 2001), pp. 59–68.
17. S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi, "Optimization by Simulated Annealing," *Science* **220**, No. 4598, 671–680 (May 1983).
18. V. Markl and G. M. Lohman, "Learning Table Access Cardinalities with LEO," *Proceedings of the ACM SIGMOD International Conference on Management of Data*, Madison, WI, June 2002, ACM, New York (2002), p. 613.

Accepted for publication October 11, 2002.

Volker Markl *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: marklv@us.ibm.com).* Dr. Markl is a research staff member in the Advanced Database Solutions Department at the Almaden Research Center in San Jose, California, conducting research in query optimization, indexing, and self-managing databases. Dr. Markl is spearheading the LEO project at IBM, an effort in autonomic computing with the goal of creating a self-tuning optimizer for DB2. Dr. Markl holds a Ph.D. degree and M.S. degree in computer science from Technische Universität München, as well as a degree in business administration from Universität Hagen, Germany. In his earlier professional career, Dr. Markl coinvented and developed the enabling indexing technology for the relational database management system TransBase HyperCube, which was awarded the European Information Society Technology Prize in 2001 by the European Commission.

Guy M. Lohman *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: lohman@almaden.ibm.com).* Dr. Lohman is manager of advanced optimization in the Advanced Database Solutions Department at the Almaden Research Center in San Jose, California, and has 20 years of experience in relational query optimization. He is the architect of the Optimizer of the DB2 Universal Database (UDB) for Linux, UNIX, and Windows, and was responsible for its development in Versions 2 and 5. During that period, Dr. Lohman also managed the overall effort to incorporate into the DB2 UDB product the Starburst compiler technology that was prototyped at the Almaden Research Center. More recently, he was a coinventor and designer of the DB2 Index Advisor, and cofounder of the DB2 SMART (Self-Managing And Resource Tuning) project, part of IBM's autonomic computing initiative. In 2002, Dr. Lohman was elected to the IBM Academy of Technology. His current research interests involve query optimization and self-managing database systems.

Vijayshankar Raman *IBM Research Division, Almaden Research Center, 650 Harry Road, San Jose, California 95120 (electronic mail: ravijay@us.ibm.com).* Dr. Raman is a research staff member at the Almaden Research Center, with a focus on data management issues in grid computing, and on adaptive query optimization. He is also interested in algorithmic mechanism design, and in data cleaning and integration. Dr. Raman graduated from the University of California, Berkeley, in 2001 with a Ph.D. degree in computer science, specializing in database management systems. His research resulted in a dozen refereed papers in international conferences and journals, one of which was selected as one of the best papers at the 25th International Conference on Very Large Databases. One component of his research has evolved into the Potter's Wheel open source data cleaning software. Dr. Raman was awarded a Microsoft Fellowship during his graduate study. He also won an AT&T Asia-Pacific Leadership Award for achievements during his undergraduate study at the Indian Institute of Technology, Madras.