LEO – DB2's LEarning Optimizer

Michael Stillger[3*], Guy Lohman[1], Volker Markl[1], Mokhtar Kandil[2]

[1]IBM Almaden Research Center
650 Harry Road, K55/B1
San Jose, CA, 95139
USA

[2]IBM Canada Ltd.
1150 Eglinton Ave. E.
Toronto, ON M3C 1H7
Canada

[3*]Siebel Systems, Inc.
2207 Bridgepointe Parkway
San Mateo, CA 94404
USA

mstilger@siebel.com, {lohman, marklv}@almaden.ibm.com, mkandil@ca.ibm.com

Abstract

Most modern DBMS optimizers rely upon a cost model to choose the best query execution plan (*QEP*) for any given query. Cost estimates are heavily dependent upon the optimizer's estimates for the number of rows that will result at each step of the QEP 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 introduce LEO, DB2's LEarning Optimizer, as a comprehensive way to repair incorrect statistics and cardinality estimates of a query execution plan. By monitoring previously executed queries, LEO compares the optimizer's estimates with actuals at each step in a QEP, and computes adjustments to cost estimates and statistics that may be used during future query optimizations. This analysis can be done either on-line or off-line on a separate system, and either incrementally or in batches. In this way, LEO introduces a feedback loop to query optimization that enhances the available information on the database where the most queries have occurred, allowing the optimizer to actually learn from its past mistakes. Our technique is general and can be applied to any operation in a QEP, including joins, derived results after several predicates have been applied, and even to DISTINCT and GROUP-BY operators. As shown by performance measurements on a 10 GB TPC-H data set, the runtime overhead of LEO's monitoring is insignificant, whereas the potential benefit to response time from more accurate cardinality and cost estimates can be orders of magnitude.

**Proceedings of the 27th VLDB Conference,
Roma, Italy, 2001**

1. Introduction

Most modern query optimizers for relational database management systems (*DBMSs*) determine the best query execution plan (*QEP*) for executing an SQL query by mathematically modeling the execution cost for each plan and choosing the cheapest QEP. This execution cost is largely dependent upon the number of rows that will be processed by each operator in the QEP. Estimating the number of rows – or *cardinality* – after one or more predicates have been applied has been the subject of much research for over 20 years [SAC+79, Gel93, SS94, ARM89, Lyn88]. Typically this estimate relies on statistics of database characteristics, beginning with the number of rows for each table, multiplied by a *filter factor* – or *selectivity* – for each predicate, derived from the number of distinct values and other statistics on columns. The selectivity of a predicate P effectively represents the probability that any row in the database will satisfy P.

While 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: The statistics are assumed to reflect the current state of the database, i.e. that the database characteristics are relatively stable.

Uniformity: Although histograms deal with skew in values for "local" selection predicates (to a single table), we are unaware of any available product that exploits them for joins.

Independence of predicates: Selectivities for each predicate are calculated individually and multiplied together, even though the underlying columns may be related, e.g. by a functional dependency. While multi-dimensional histograms address this problem for local predicates, again they have never been applied to join predicates, aggregation, etc. 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 multi-dimensional histograms.

* Work performed while the author was a post-doc at IBM ARC.

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", i.e. that each value of the smaller domain has a match in the larger domain (which is frequently true for joins between foreign keys and primary keys).

When these assumptions are invalid, significant errors in the cardinality – and hence cost -- estimates result, causing sub-optimal 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. Although there has been considerable success in using histograms to detect and correct for data skew [IC91, PIHS96, PI97], and in using sampling to gather up-to-date statistics [HS93, UFA98], there has to date been no comprehensive approach to correcting all modeling errors, regardless of origin.

This paper introduces LEO, the **LE**arning **O**ptimizer, which incorporates an effective and comprehensive technique for a query optimizer actually to learn from any modeling mistake at any point in a QEP, by automatically validating its estimates against actuals for a query after it finishes executing, determining at what point in the plan the significant errors occurred, and adjusting its model dynamically to better optimize future queries. Over time, LEO 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 also can suggest where statistics gathering should be concentrated or even can supplant the need for statistics collection. LEO has been prototyped on IBM's DB2 Universal Data Base (UDB) on the Windows, Unix, and OS/2 platforms (hereafter referred to simply as "DB2"), and has proven to be very effective at correcting cardinality estimation errors.

This paper is organized as follows. Section 2 explores the previous literature in relation to LEO. We give an overview of LEO and an example of its execution in Section 3. Section 4 details how LEO works, including the four major components of capturing the optimizer's plan, monitoring the execution, analyzing the actuals vs. estimates, and exploiting what is learned in the optimizer for subsequent queries. In Section 5, we evaluate LEO's performance – both its overhead and benefit. Section 6 discusses advanced topics and Section 7 contains our conclusions and future work.

2. Related Work

Much of the prior literature on cardinality estimates has utilized histograms to summarize the data distribution of columns in stored tables, for estimating the selectivity of predicates against those tables. Recent work has extended one-dimensional equi-depth histograms to more sophisticated and accurate versions [PIHS96] and to multiple dimensions [PI97]. This classical work on histograms concentrated on the accuracy of histograms in the presence of skewed data and correlations by scanning the base tables completely, at the price of high run-time cost. The work in [GMP97] deals with the necessity of keeping histograms up-to-date at very low cost. Instead of computing a histogram on the base table, it is incrementally derived and updated from a backing sample of the table, which is always kept up-to-date. Updates of the base table are propagated to the sample and can trigger a partial re-computation of the histogram, but there is no attempt to validate the estimates from these histograms against run-time actuals.

The work of [CR94] and [AC99] are the first to monitor cardinalities in query executions and exploit this information in future compilations. In [CR94] the result cardinalities of simple predicates after the execution of a query are used to adapt the coefficients of a curve-fitting formula. The formula approximates the value distribution of a column instead of employing histograms for selectivity estimates. In [AC99] the authors present a query feedback loop, in which actual cardinalities gleaned from executing a query are used to correct histograms. Multiple predicates can be used to detect correlation and update multi-dimensional histograms. This approach effectively deals with single-table predicates applied while accessing a base table, but the paper does not deal with join predicates, aggregation, and other operators, nor does it specify how the user is supposed to know on which columns multi-dimensional histograms should be created. LEO's approach extends and generalizes this pioneering work. It can learn from any modeling error at any point in a QEP, including errors due to local predicates, expressions of base columns involving user-defined functions, predicates involving parameter markers or host variables, join predicates, keys created by the DISTINCT or GROUP BY clauses, derived tables, and any correlation between any of the above. Most of these operations that change cardinality in some way cannot be addressed by histograms. LEO can even adjust estimates of other parameters such as buffer utilization, sort heap consumption, I/Os, or the actual running time -- the only real limitation to LEO's approach is the overhead of collecting the actuals for those estimates.

Another research direction focuses on dynamically adjusting a QEP after the execution has begun, by monitoring data statistics during the execution (dynamic optimization). In [KDeW98] the authors introduce a new *statistic collector* operator that is compiled into the plan. The operator collects the row stream cardinality and size and decides whether to continue or to stop the execution and re-optimize the remainder of the plan. Query scrambling in [UFA98] is geared towards the problem of distributed query execution in wide area networks with

uncertain data delivery. Here the time-out of a data-shipping site is detected and the remaining data-independent parts of the plan are re-scheduled until the problem is solved. Both solutions deal with dynamic re-optimization of (parts of) a single query, but they do not save and exploit this knowledge for the next query optimization run. LEO is aimed primarily at using information gleaned from one or more query executions to discern trends that will benefit the optimization of future queries. This benefit is not limited to just the same query, because the exact same query is seldom re-executed in modern data warehouses, data marts, and business intelligence applications. Any query with predicates or aggregation on the same column(s) can exploit LEO's learning. LEO does not (yet) address the issue of changing in mid-stream the QEP of a running query, as did [KDeW98] and [UFA98], although it could. Doing this correctly in a real product needs to resolve many hard issues not addressed by that work, such as determining points where such changes produce correct results (i.e., where data is fully materialized, before any results are returned to the user), and reliably predicting the times to re-optimize and execute a new plan so that they can be traded off against the time to complete the original plan.

3. A Learning Optimizer

This section gives an overview of LEO's design, a simplified example of how it learns, and some of the practical issues that it must deal with.

3.1 An Overview of LEO

LEO 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.

LEO is comprised of four components: a component to save the optimizer's plan, a monitoring component, an analysis component, and a feedback exploitation component. The analysis component is a standalone process that may be run separately from the DB2 server, and even on another system. The remaining three components are modifications to the DB2 server: plans are captured at compile time by an addition to the code generator, monitoring is part of the run-time system, and feedback exploitation is integrated into the optimizer.

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.

Figure 1 shows how LEO is integrated into the architecture of DB2. The left part of the figure shows the usual query processing flow with query compilation, QEP generation and optimization, code generation, and code execution. The gray shaded boxes show the changes made to regular query processing to enable LEO's feedback loop: for any query, the code generator dumps essential information about the chosen QEP (a plan "skeleton") into a special file that is later used by the LEO analysis daemon. In the same way, the runtime system provides monitored information about cardinalities for each operator in the QEP. Analyzing the plan skeletons and the runtime monitoring information, the LEO analysis daemon computes adjustments that are stored in the system catalog. The exploitation component closes the feedback loop by using the adjustments in the system catalog to provide adjustments to the query optimizer's cardinality estimates.

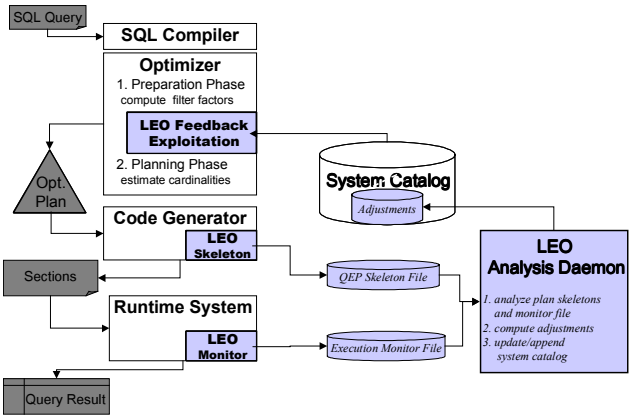

Figure 1: LEO Architecture

3.2 Monitoring and Learning: An Example

In the following we use as an example the SQL query:

```
SELECT * FROM X, Y, Z
WHERE X.Price >= 100 AND Z.City = 'Denver'
AND Y.Month = 'Dec' AND X.ID = Y.ID
AND Y.NR = Z.NR
GROUP BY A
```

Figure 2 shows the skeleton of a QEP for this statement, including the statistical information and the optimizer's cardinality estimates. In addition, the figure also shows the actual cardinalities that the monitoring component of LEO determined during query execution.

In the Figure, cylinders indicate base table access operators such as index scan (IXSCAN) or table scan (TBSCAN), ellipses indicate other operators, such as nested loop joins (NLJOIN) and grouping (GROUP BY). "Stat" denotes the base table cardinality, as stored in the system catalog, and "Est:" denotes the optimizer's estimate for the result cardinality of each table access operator. after application of any predicates (e.g., X.Price >= 100), as well as for each of the nested-loop join operators. During query execution, the LEO monitoring component measures the comparable actual cardinality ("*Act*") for each operator.

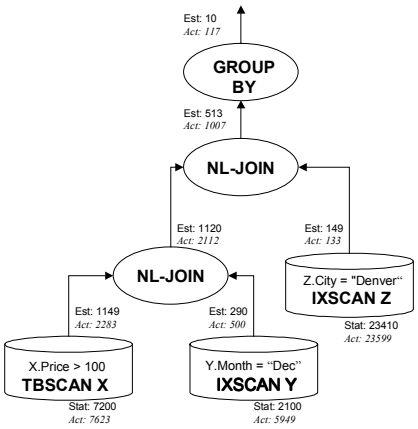

Figure 2: Optimal QEP (Skeleton)

Comparing actual and estimated cardinalities enables LEO to give feedback to the statistics that were used for obtaining the base table cardinalities, as well as to the cardinality model that was used for computing the estimates. This feedback may be a positive reinforcement, e.g., for the table statistics of Z, where the table access operator returned an actual cardinality for Z that is very close to that stored in the system catalog statistics. The same holds for the output cardinalities of each operator, such as a positive feedback for the estimate of the restriction on Z that also very closely matches the actual number. However, it may also be a negative feedback – as for the table access operator of Y, where the statistics suggest a number almost three times lower than the actual cardinality – or for the join estimates of the nested-loop join between X and Y. In addition, correlations can be detected, if the estimates for the individual predicates are known to be accurate but some combination of them is not. In all of the above, "predicates" can actually be generalized to any operation that changes the cardinality of the result. For example, the creation of keys by a DISTINCT or GROUP BY clause reduces the number of rows. LEO uses this feedback to help the optimizer to learn to better estimate cardinalities the next time a query involving these tables, predicates, joins, or other operators is issued against the database.

3.3 Practical Considerations

In the process of implementing LEO, several practical considerations became evident that prior work had not addressed. We now discuss some of these general considerations, and how they affected LEO's design.

Modifying Statistics vs. Adjusting Selectivities

A key design decision is that LEO never updates the original catalog statistics. Instead, it constructs a second set of statistics that will be used to adjust (i.e. repair) the first, original layer. The adjustments are stored as special tables in the system catalog. The compilation of new queries reads these adjustments, as well as the base statistics, and adjusts the optimizer's estimates appropriately. This two-layered approach has several advantages. First, we have the option of disabling learning, by simply ignoring the adjustments. This may be needed for debugging purposes or as a fallback strategy in case the system generated wrong adjustments or the new optimal plan shows undesired side effects. Second, we can store the specific adjustment value with any plan that uses it, so that we know by how much selectivities have already been adjusted and avoid incorrect re-adjustments (no "deltas of deltas"). Lastly, since we keep the adjustments as catalog tables, we introduce an easily accessible mechanism for tuning the selectivities of query predicates that could be updated manually by experienced users, if necessary.

Consistency between Statistics

DB2 collects statistics for base tables, columns, indexes, functions, and tablespaces, many of which are mutually interdependent. DB2 allows for incremental generation of statistics and checks inconsistencies for user-updateable statistics. LEO also must ensure the consistency of these interdependent statistics. For example, the number of rows of a table determines the number of disk pages used for storing these rows. When adjusting the number of rows of a table, LEO consequently also has to ensure consistency with the number of pages of that table -- e.g., by adjusting this figure as well -- or else plan choices will be biased. Similarly, the consistency between index and table statistics has to be preserved: If the cardinality of a column that is (a prefix of) an index key is adjusted in the table statistics, this may also affect the corresponding index statistics.

Currency vs. Accuracy

Creating statistics is a costly process, since it requires scanning an entire table or even the entire database. For this reason, database statistics are often not existent or not accurate enough to help the optimizer to pick the best access plan. If statistics are expected to be outdated due to later changes of the database or if no statistics are present, DB2 fabricates statistics from the base parameters of the table (file size from the operating system and individual column sizes). The presence of adjustments and fabricated statistics creates a decision problem for the optimizer -- it must decide whether to believe possibly outdated adjustments and statistics, or fuzzy but current fabricated statistics.

When statistics are updated, many of the adjustments calculated by LEO no longer remain valid. Since the set of adjustments that LEO maintains is not just a subset of the statistics provided by RUNSTATS, removing all adjustments during an update of the statistics might result in a loss of information. Therefore any update of the statistics should re-adjust the adjustments appropriately, in order to not loose information like actual join selectivities and retain consistency with the new statistics.

LEO vs. Database Statistics

LEO is not a replacement for statistics, but a rather a complement: LEO gives the most improvement to the modeling of queries that are either repetitive or are similar to earlier queries, i.e., queries for which the optimizer's model exploits the same statistical information. LEO 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 (as compared to statistics, which are collected uniformly across the database, to be ready for any possible query). However, for correctly costing previously unanticipated queries, the statistics collected by RUNSTATS are necessary even in the presence of LEO.

4. The LEO Feedback Loop

The following sections describe the details of how LEO performs the four steps of capturing the plan for a query and its cardinality estimates, monitoring queries during execution, analyzing the estimates versus the actuals, and the exploitation of the adjustments in the optimization of subsequent queries.

4.1 Retaining the Plan and its Estimates

During query compilation in DB2, a code generator component derives an executable program from the optimal QEP. This program, called a *section*, can be executed immediately (dynamic SQL) or stored in the database for later, repetitive execution of the same query (static SQL). The optimal QEP is not retained with the section; only the section is available at run-time. The section contains one or more *threads*, which are sequences of operators that are interpreted at run-time. Some of the section's operators, such as a table access, closely resemble similar operators in the QEP. Others, such as those performing predicate evaluation, are much more detailed. Though in principle it is possible to "reverse engineer" a section to obtain the QEP from which it was derived, in practice that is quite complicated. To facilitate the interpretation of the monitor output for LEO, we chose to save at compile-time a "skeleton" subset of the optimal QEP for each query, as an analysis "road map". This *plan skeleton* is a subset of the much more complete QEP information that may optionally be obtained by a user through an EXPLAIN of the query, and contains only the basic information needed by LEO's analysis, including the cumulative cardinality estimates for each QEP operator, as shown in Figure 2.

4.2 Monitoring Query Execution

LEO captures the actual number of rows processed by each operator in the section by carefully instrumenting the section with run-time counters. These counters are incremented each time an operator processes a row, and saved after the query completes. LEO can be most effective if this monitoring is on all the time, analyzing the execution of every query in the workload. For this to be practical, LEO's monitoring component must impose minimal overhead on regular query execution performance. The overhead for incrementing these counters has been measured and shown to be minimal, as discussed in Section 5.1.

4.3 Analyzing Actuals and Estimates

The analysis component of LEO may be run off-line as a batch process, perhaps even on a completely separate system, or on-line and incrementally as queries complete execution. The latter provides more responsive feedback to the optimizer, but is harder to engineer correctly. To have minimal impact on query execution performance, the analysis component is designed to be run as a low-priority background process that opportunistically seizes "spare cycles" to perform its work "post mortem". Any mechanism can be used to trigger or continue its execution, preferably an automated scheduler that supervises the workload of the system. Since this means LEO can be interrupted by the scheduler at any point in time, it is designed to analyze and to produce feedback data on a per-query basis. It is not necessary to accumulate the monitored data of a large set of queries to produce feedback results.

To compare the actuals collected by monitoring with the optimizer's estimates for that query, the analysis component of LEO must first find the corresponding plan skeleton for that query. Each plan skeleton is hashed into memory. Then for each entry in the monitor dump file (representing a query execution), it finds the matching skeleton by probing into the skeletons hash table. Once a match is located, LEO needs to map the monitor counters for each section operator back to the appropriate QEP operator in the skeleton. This is not as straightforward as it sounds, because there is not a one-to-one relationship between the section's operators and the QEP's operators. In addition, certain performance-oriented optimizations will bypass operators in the section if possible, thus also bypassing incrementing their counters. LEO must detect and compensate for this.

```
analyze_main(skeleton root) {
  preprocess (root); error = OK;
  // construct global state and
  // pushdown node properties
  for (i = 0; i < children(root); i++)
  // for each child
    {error |= analyze_main(root->child[i]); }
    // analyze
  if (error) return error;
  // if error in any child: return error
  switch (root->opcode) // analyze operator
    case IXSCAN: return analyze_ixscan(root)
    case TBSCAN: return analyze_tbscan(root)
    case …
```

Figure 3: LEO algorithm

The analysis of the skeleton tree is a recursive post-order traversal (see Figure 3). Before actually descending down the tree, a preprocessing of the node and its immediate children is necessary to construct global state information and to push down node properties. The skeleton is analyzed bottom up, where the analysis of a branch stops after an error occurred in the child. Upon returning from all children, the analysis function of the particular operator is called.

Calculating the Adjustments

Each operator type (*TBSCAN, IXSCAN, FETCH, FILTER, GROUP BY, NLJOIN, HSJOIN, etc.*) can carry multiple predicates of different kinds (*start/stop keys, pushed down, join*). According to the processing order of the predicates within the operator, LEO will find the actual monitor data (input and output cardinalities of the data stream for the predicate) and analyze the predicate. By comparing the actual selectivity of the predicate with the estimated selectivity that was stored with the skeleton, LEO deduces an adjustment factor such that the DB2 optimizer can later compute the correct selectivity factor from the old estimate and the new adjustment factor. This adjustment factor is immediately stored in the database in new LEO tables. Note that LEO does not need to re-scan the DB2 catalog tables to get the original statistics, because the estimates that are based on these statistics are stored with the skeleton.

LEO computes an adjustment such that the product of the adjustment factor and the estimated selectivity derived from the DB2 statistics yields the correct selectivity. To achieve that, LEO uses the following variables that were saved in the skeleton or monitor result:

old_est: the estimated selectivity from the optimizer

old_adj: an old adjustment factor that was possibly used to compute old_est

act: The actual selectivity that is computed from the monitor data

After detecting an error ($| old\_est - act | / act > 0.05$) for the predicate $col < X$, LEO computes the adjustment factor so that the new estimate equals the actual value (*act*) computed from the monitor: $est = actual = stats * adj$; where *stats* is the original selectivity as derived from the catalog. The old estimate (*old_est*) is either equivalent to the original statistic estimate (*stats*) or was computed with an old adjustment factor (*old_adj*). Hence this old adjustment factor needs to be factored out. ($adj = act / stats = act/(old\_est/old\_adj) = act*(old\_adj/old\_est)$.

Since the selectivity for the predicate ($col >= X$) is 1 – selectivity($col < X$), we invert the computation of the estimate and the adjustment factor for this type of predicate. Note that we derive an adjustment factor for the < -operator from the results of the > -operator, and we apply the adjustment factor of a < -operator for the computation of the > -operator.

Using the example from Figure 2 and a TBSCAN on table X with the predicate Price >= 100, we can compute the adjustment factors for the table cardinality and the predicate. The cardinality adjustment factor is 7632/7200 = 1.06. The estimated selectivity of the predicate was 1149/7200 = 0.1595 while the actual selectivity is 2283/7632 = 0.2994. The adjustment factor for the corresponding *Price < 100* -predicate is (1 - 0.2994) * 1.0 / (1 -0.1595) = 0.8335. The optimizer will compute the selectivity for this predicate in the future to be 1 – 0.8335 * (1 – 0.1595) = 0.2994. The adjusted table cardinality of the TBSCAN (1.06*7200) times the adjusted predicate selectivity 0.2994 computes the correct, new estimate of the output cardinality of the TBSCAN operator (2283).

However, different types of section operators can be used to execute a particular predicate such as 'Price >= 100'. If the Price column is in the index key, the table access method could be an IXSCAN-FETCH combination. If Price is the leading column of the index key, the predicate can be executed as a start/stop key in the IXSCAN operator. Then IXSCAN delivers only those rows (with its row identifier or *RID*) that fulfill the key predicate. FETCH uses each RID to retrieve the row from the base table. If the predicate on Price cannot be applied as a start/stop key, it is executed as *a push-down predicate* on every row returned from the start/stop key search. When using a start/stop key predicate, we scan neither the index nor the base table completely, and hence cannot determine the actual base table cardinality. In order to determine the real selectivity of an index start/stop key predicate, we can only approximate the needed input cardinality by using the old cardinality estimates, if a previously computed table adjustment factor was used[1]

The merge-join algorithm demonstrates a similar problem that we have named *implicit early out*. Recall that both inputs of the merge join are sorted data streams. Each row will be matched with the other side until a higher-valued row or no row at all is found. Reaching the end of the data stream on one side immediately stops the algorithm. Thus any remaining rows from the other side will never be asked for, and hence are not seen or counted by the monitor. As a result, any monitor number for merge-join input streams is unreliable unless we have encountered a "dam" operator such as SORT or TEMP, which by materializing all rows ensures the complete scan and count of the data stream prior to the merge join.

Storing the Adjustments

For storing the adjustments, the new tables LEO_TABLES, LEO_COLUMNS and LEO_JOINS have been introduced into the DB2 system catalog.

Take as an example the column adjustment catalog as stored in LEO_COLUMNS. The columns (tablespaceID, tableID, columnID) uniquely identify a column (i.e. X.Price), while the *Adj_factor* = 0.8335 and *Col_Value* =

[1] The existence of an adjustment factor indicates that we have seen a complete table scan earlier and successfully repaired an older statistic.

'100'. *Timestamp* is the compile time of the query and is used to prohibit learning from old knowledge. *Type* indicates the type of entry: 'F' for a frequent value or 'Q' for a quantile adjustment for the corresponding *Col_Value* value. In LEO_JOINS, a join is sufficiently described by two triplets for the two join columns. Introducing a simple rule of (lexicographic) order on the columns' triplets is sufficient to store the adjustment factors only once: the 'smaller' column is stored with its join partner and the adjustment factor. A simple index scan with a search key on the "smaller" join column allows us to efficiently update or retrieve the adjustment factor from the database.

4.4 Using Learned Knowledge

Before the DB2 Optimizer begins constructing candidate plans, it first retrieves the schema and statistics for each base table referenced in that query from the catalog cache. From these statistics, the optimizer gets the base-table cardinality and computes selectivity factors for each predicate. At this point, if LEARNING is enabled by a control flag, the optimizer will also search the catalog for any adjustment factors that may be relevant to this query, and adjust the base table statistics, predicate selectivities, and other statistics accordingly. How this is done for each type of adjustment is the subject of this section.

Base Table Cardinalities

We start first with adjusting the base table cardinalities, since these are thebasis for all cardinality estimates of plans. The statistics for the base-table's cardinality need only be multiplied by the adjustment factor, if any, for that table.

As discussed earlier, the difficulty comes in maintaining the consistency of this adjusted cardinality with other statistics for that table. The number of pages in the table, NPAGES, is collected during RUNSTATS and is directly used in the cost model as a more accurate measurement for the number of I/O operations during TBSCAN operations than computing it from the table cardinality, the row width, and the page size. As a result, LEO must adjust NPAGES for base tables, as well as the index statistics (the number of leaf and non-leaf pages) accordingly. In addition, the column cardinalities for each column obviously cannot exceed the table cardinality, but increasing the number of rows may or may not increase the cardinality of any column. For example, adding employee rows doesn't change the cardinality of the Sex column, but probably changes the cardinality of the EmployeeID column. Similarly, the consistency between index and table statistics has to be preserved. If a column that is in one or more index keys has its cardinality adjusted in the table statistics, the corresponding index cardinality statistics (FIRSTKEYCARD, FIRST2KEYCARD, …, FULLKEYCARD) must also be adjusted accordingly.

Single-Table Predicates

Next, we consider adjustments to the selectivity of a simple, single-table predicate, illustrated by adjusting the column X.Price for the predicate X.Price < 100. Figure 4 shows the actual cumulative distribution for X.Price.

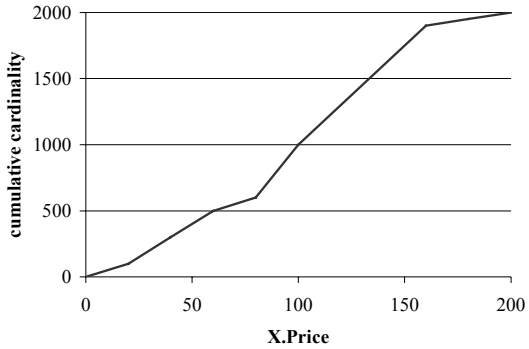

Figure 4: Actual Data Distribution

Figure 5 shows the column statistics collected for X.Price and Figure 6 the corresponding adjustments.

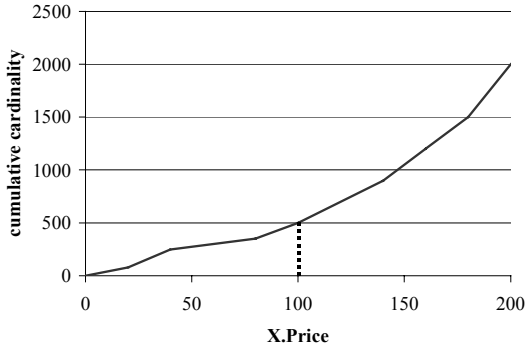

Figure 5: Column Statistics

The optimizer computes the selectivity for X.Price < 100 from the statistics by cardinality(X < 100) / Maximal_Cardinality = 500/2000 = 0.25. Applying the adjustments results in adjusted_selecticity(X.Price < 100) = cardinality(X.Price < 100) * adjustment(X.Price < 100) = 0.25 * 2 = 0.5. If there is no exact match in the column statistics for a column value (i.e. X.Price < 100), the adjustment factor is computed by linearly interpolating within the interval in which the value '100' is found.

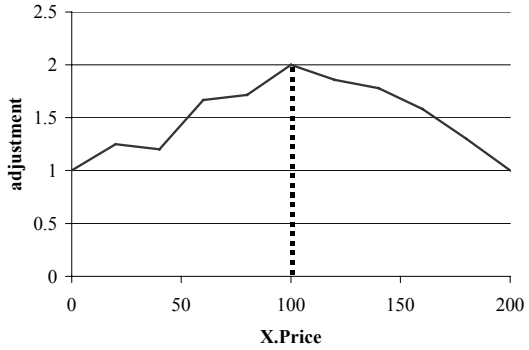

Figure 6: Adjustments

In Figure 7, statistics do not exist (which is equivalent to a default selectivity of 1/3, i.e., a uniformly distributed cardinality of 667). The adjustment curve here shows higher or lower amplitudes than the one for the statistics. For our example: adjustment(X.Price < 100) = 1.5.

Suppose that the optimizer had used an earlier adjustment factor of 2 to compute the estimate for the predicate 'X.Price < 100'. Suppose further that, due to more updates, the real selectivity of the predicate is 0.6 instead of the newly estimated 0.5. LEO needs to be aware of this older adjustment factor to undo its effects.

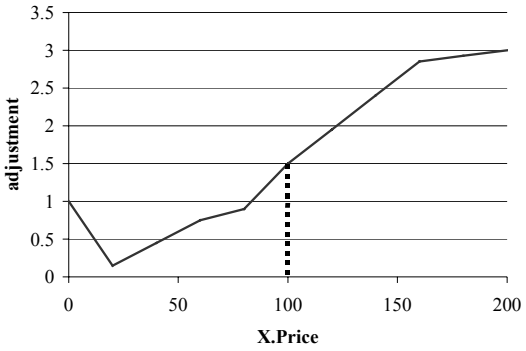

Figure 7: Adjustments without Statistics

In our model, an adjustment factor is always based on the systems statistics and never an adjustment of an older adjustment. The new factor is computed by act_selectivity * old_adj / est = 0.6 * 2 / 0.5 = 2.4. Thus any previously used adjustment factor must be saved with the QEP skeleton. Note that it is not sufficient to look up the adjustment factor in the system table, since LEO cannot know if it was actually used for that query or if it has changed since the compile time of that query.

The LEO approach is not limited to simple relational predicates on base columns, as is the histogram approach of [AC99]. The "column" could be any expression of columns (perhaps involving arithmetic or string operations), the "type" could be LIKE or user-defined functions, and the literal could even be "unknown", as with parameter markers and host variables.

Join Predicates

As indicated above, LEO can also compute adjustment factors for equality join operators. The adjustment factor is simply multiplied by the optimizer's estimate. Note that having the actuals and estimates for each operator permits LEO to eliminate the effect of any earlier estimation errors in the join's input streams.

Other Operators

The GROUP BY and DISTINCT clauses effectively define a key. An upper bound on the resulting cardinality of such operations can be derived from the number of distinct values for the underlying column(s): the COLCARD statistic for individual columns, or the FULLKEYCARD statistic for indexes, if any, on multiple columns. However, predicates applied either before or after these operations may reduce the real cardinalities resulting. Similarly, set operations such as UNION (DISTINCT), UNION ALL, and EXCEPT may combine two or more sets of rows in ways that are difficult for the optimizer to predict accurately. Although not implemented in the current prototype, the analysis routine can readily compute the adjustment factor as adj = act * old_adj / old_est, and adjust the cardinality output by each of these operators by multiplying its estimate by adj. It is doubtful that the histogram approach of [AC99] could provide adjustments for these types of operations in SQL.

Correlation between predicates

Optimizers usually assume *independence* of columns. This allows for estimating the selectivity of a conjunctive predicate as a product of the selectivity of the atomic predicates. However, *correlations* sometimes exist between columns, when the columns are not independent. In this case, the independence assumption underestimates the selectivity of a conjunctive predicate.

In practical applications, data is often highly correlated. Types of correlations include functional dependencies between columns and referential integrity, but also more complex cases such as a constraint that a part is supplied by at most 20 suppliers. Furthermore, correlations may involve more than two columns, and hence more than two predicates. Therefore, any set of predicates may have varying degrees of correlation. How are errors due to correlation discerned from errors in the selectivities of the individual predicates? LEO's approach is to first correct individual predicate filter factors, using queries that apply those predicates in isolation. Once these are adjusted, any errors when they are combined must be attributable to correlation. A single query can provide evidence that two or more columns are correlated for specific values; LEO must cautiously mine the execution of multiple queries having predicates on the same columns before it can safely conclude that the two columns are, in general, correlated to some degree. The multi-dimensional histogram approach of [AC99] could be used here, but presumes that the user knows which columns are correlated and predefines a multi-dimensional histogram for each. LEO can automatically detect good candidates for these multi-dimensional histograms through its analysis.

In our current implementation of LEO, we only take advantage of correlations between join columns. An extension of LEO might take further advantage of correlation in order to provide even better adjustments.

5. Performance

5.1 Overhead of LEO's Monitoring

LEO requires monitoring query execution in order to obtain the actual cardinalities for each operator of a QEP.

Our performance measurements on a 10 GB TPC-H database [TPC00] show that for our prototype of LEO the monitoring overhead is below 5% of the total query execution time, and therefore may be neglected for most applications.

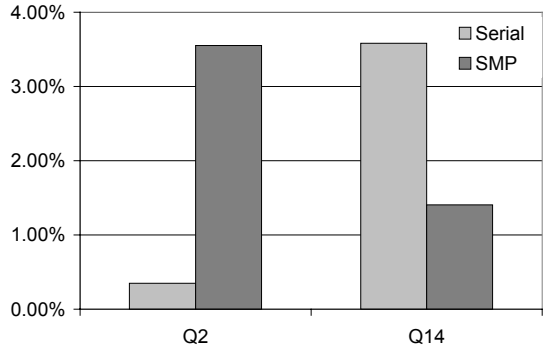

Figure 8: Monitoring Overhead for a 10 GB TPC-H Database

Figure 8 shows the actual measurement results for the overhead for TPC-H queries Q2 and Q14, measured both on a single-CPU (serial) and on an SMP machine. These overheads were measured on a LEO prototype. For the product version, further optimizations of the monitoring code will reduce the monitoring overhead even further.

Our architecture permits dynamically enabling and disabling monitoring, on a per-query basis. If time-critical applications cannot accept even this small overhead for monitoring, and thus turn monitoring off, they can still benefit from LEO, as long as other – uncritical – applications monitor their query execution and thus provide LEO with sufficient information.

5.2 Benefit of Learning

Adjusting outdated or incorrect information may allow the optimizer to choose a better QEP for a given query. Depending on the difference between the new and the old QEP, LEO may drastically speed-up query execution

Suppose now that the database in our example has changed significantly since the collection of statistics: the Sales stored in table Y increased drastically in December and the inventory stored in table X received many updates and inserts, where most new items had a price greater than 100. This results in an overall cardinality of more than 21623 records for X and 17949 records for Y. Suppose further that these changes also introduce a skew in the data distribution, changing the selectivities of the predicates X.Price > 100 and Y.Month = 'Dec'. Finally, suppose that a query referencing table X with the predicate X.Price > 150[2], and another query referencing Y with the predicate Y.Month ='Dec', have been executed, providing LEO with some adjustments. Figure 9 shows how LEO changes the query execution plan for the query of Section 3.2 after these changes. The optimizer now chooses to use a bulk method for joining X and Y for this query, thus replacing the nested-loop join with a hash join. Note that the index scan on Y was also replaced by a table scan, due to the adjustments. This new plan resulted in an actual execution speed-up of more than one order of magnitude over the earlier plan executing on the same data.

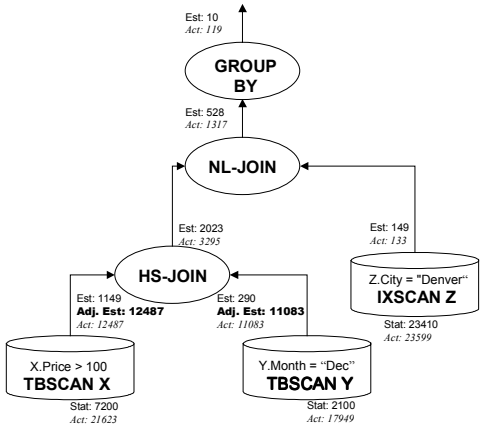

Figure 9: Optimal QEP after Learning

Our experiments on two highly dynamic test databases (artificial and TPC-H) showed that the adjustments enabled the optimizer to choose a QEP that performed up to 14 times better than the QEP without adjustments, while LEO consumed an insignificant runtime overhead, as shown in Section 5.1. Of course, speed-ups can be even more drastic, since LEO's adjustments can cause virtually any physical operator of a QEP to change, and may even alter the structure of the QEP. The most prominent changes are table access operators (IXSCAN, TBSCAN), join method (NLJOIN, HSJOIN, MGJOIN), and changing the join order for multi-way joins.

6. Advanced Topics

6.1 When to Re-Optimize

A static query is bound to a plan that the optimizer has determined during query compilation. With LEO, the plan for a static query may change over time, since the adjustments might suggest an alternative plan to be better than the plan that is currently used for that query. The same holds for dynamic queries, since DB2 stores the optimized plan for a dynamic query in a statement cache.

Currently we do not support rebinding of static queries or flushing the statement cache because of learned knowledge. It remains future work to investigate whether and when re-optimization of a query should take place. The trade-off between re-optimization and improved runtime must be weighed in order to be sure that re-optimization will result in improved query performance.

[2] Note that it is not necessary to run a query with exactly the predicate X.Price > 100, since LEO performs interpolation for histograms. Thus an adjustment for X.Price >150 is also be useful for a query X.Price > 100.

6.2 Learning Other Information

Learning and adapting to a dynamic environment is not restricted to cardinalities and selectivities. Using a feedback loop, many configuration parameters of a DBMS can be made self-tuning. If, for instance, the DBMS detects by query feedback that a sort operation could not be performed in main memory, the sort heap size could be adjusted in order to avoid external sorting for future sort operations. In the same way, buffer pools for indexes or tables could be increased or decreased according to a previously seen workload. This is especially interesting for resources that are assigned on a per-user basis: Instead of assuming uniformity, buffer pools or sort heaps could be maintained individually per user. If dynamic adaptation is possible even during connections, open but inactive connections could transfer resources to highly active connections.

Another application of adjustments is to "debug" the cost model of the query optimizer: If – despite correct base statistics – the cost prediction for a query is way off, analyzing the adjustment factors permits locating which of the assumptions of the cost model are violated.

Physical parameters such as the network rate, disk access time, or disk transfer rate are usually considered to be constant after an initial set-up. However, monitoring and adjusting the speed of disks and networks enables the optimizer to adjust dynamically to the actual workload and use the effective rate.

7 Conclusions

LEO provides a general mechanism for an optimizer to actually learn from its mistakes by adjusting its cardinality and other estimates using the actuals from the execution of previous queries having similar predicates. Regardless of the source of error – old statistics, invalid assumptions, inadequate modeling, unknown literals, etc. – LEO can detect and correct the mistake for any kind of operation that changes the cardinality, at any point in a plan. This is a far more general mechanism than multi-dimensional histograms, which are limited to local predicates on columns of a base table. Our performance measurements have demonstrated that LEO can improve cardinality estimates by orders of magnitude, changing plans to improve performance by orders of magnitude, while adding less than 5% overhead to execution time when monitoring actuals. We feel that 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 future work includes completing the implementation of LEO's adjustments for all types of predicates, measuring the benefit on a realistic set of user queries, finding conclusive ways to discern correlation among predicates, applying LEO's approach to parameters other than cardinality, and possibly using LEO's adjustments to change a query's plan dynamically during its execution in a robust, industrial-strength way.

Acknowledgements

The authors thank Kwai Wong for her help with the measurements of the LEO runtime overhead, and Ashutosh Singh and Eric Louie for systems support.

Bibliography

AC99 A. Aboulnaga and S. Chaudhuri, Self-tuning Histograms: Building Histograms Without Looking at Data, SIGMOD, 1999

ARM89 R. Ahad, K.V.B. Rao, and D. McLeod, On Estimating the Cardinality of the Projection of a Database Relation, TODS 14(1), pp. 28-40.

CR94 C. M. Chen and N. Roussopoulos, Adaptive Selectivity Estimation Using Query Feedback, SIGMOD, 1994

Gel93 A. Van Gelder, Multiple Join Size Estimation by Virtual Domains, PODS, pp. 180-189.

GMP97 P. B. Gibbon, Y. Matias and V. Poosala, Fast Incremental Maintenance of Approximate Histograms, VLDB, 1999

HS93 P. Haas and A. Swami, Sampling-Based Selectivity Estimation for Joins - Using Augmented Frequent Value Statistics, IBM Research Report, 1993

IBM00 DB2 Universal Data Base V7 Administration Guide, IBM Corp., 2000

IC91 Y.E. Ioannidis and S. Christodoulakis. On the Propagation of Errors in the Size of Join Results, SIGMOD, 1991

KdeW98 N. Kabra and D. DeWitt, Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans, SIGMOD, 1998

Lyn88 C. Lynch, Selectivity Estimation and Query Optimization in Large Databases with Highly Skewed Distributions of Column Values, VLDB, 1988

PI97 V. Poosala and Y. Ioannidis, Selectivity Estimation without the attribute value independence assumption, VLDB, 1997

PIHS96 V. Poosala, Y. Ioannidis, P. Haas, and E. Shekita, Improved histograms for selectivity estimation of range predicates, SIGMOD. 1996, pp. 294-305

SAC+79 P.G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, T. G. Price, Access Path Selection in a Relational Database Management System, SIGMOD 1979, pp. 23-34

SS94 A. N. Swami, K. B. Schiefer, On the Estimation of Join Result Sizes, EDBT 1994, pp. 287-300

TPC00 Transaction Processing Council, TPC-H Rev. 1.2.1 specification, 2000

UFA98 T. Urhan, M.J. Franklin and L. Amsaleg, Cost-based Query Scrambling for Initial Delays, SIGMOD, 1998