Collecting and Maintaining Just-in-Time Statistics

Amr El-Helw[1] Ihab F. Ilyas[1] Wing Lau[2] Volker Markl[3] Calisto Zuzarte[2]

[1]University of Waterloo
{aelhelw, ilyas}@cs.uwaterloo.ca

[2]IBM Toronto Lab
{lauwing, calisto}@ca.ibm.com

[3]IBM Almaden Research Center
marklv@us.ibm.com

Abstract

Traditional DBMSs decouple statistics collection and query optimization both in space and time. Decoupling in time may lead to outdated statistics. Decoupling in space may cause statistics not to be available at the desired granularity needed to optimize a particular query, or some important statistics may not be available at all. Overall, this decoupling often leads to large cardinality estimation errors and, in consequence, to the selection of suboptimal plans for query execution. In this paper, we present JITS, a system for proactively collecting query-specific statistics during query compilation. The system employs a lightweight sensitivity analysis to choose which statistics to collect by making use of previously collected statistics and database activity patterns. The collected statistics are materialized and incrementally updated for later reuse. We present the basic concepts, architecture, and key features of JITS. We demonstrate its benefits through an extensive experimental study on a prototype inside the IBM DB2 engine.

1. Introduction

Cost-based optimizers use database statistics to determine the best execution plan for a query. The optimizer uses these statistics to estimate the cost of each alternative plan, and to choose the plan with the least estimated cost. Ideally, the optimizer should have access to statistics about tables, groups of predicates and other constructs involved in the query in order to accurately estimate the cost of each operator (sub-plan). However, traditional systems suffer from two main problems:

First, statistics collection and query optimization are usually decoupled. The statistics collection module has no knowledge of the queries posed to the system. It only collects general statistics that can be used with any query. Example general statistics include the number of rows in a table, the number of distinct values for a column, the most frequent values in a column, and the distribution of data values (usually stored as a histogram). The optimizer relies on several assumptions to estimate the output cardinalities of query operators from this subset of statistics, e.g., uniformity of data distribution and independence of predicates. These assumptions often do not reflect the real world data, leading to very large estimation errors.

Second, because of data updates, these statistics become stale very quickly. Statistics are not incrementally updated during data manipulation because such incremental maintenance is prohibitively expensive. Traditional systems try to address this problem by periodically updating the stored statistics. This, however, is not particularly useful for tables with high data change rates, or temporary tables that get created and dropped during a workload. The presence of outdated statistics causes the optimizer to inaccurately estimate the costs of the operators in a query plan, which results in choosing a suboptimal plan.

A brute-force approach to get accurate cost estimation would be to collect statistics on all data sources, and the combinations of predicates in a given query before optimization. However, the problem with this approach is that (1) it is non-trivial to enumerate all statistics needed by the optimizer; (2) collecting all needed statistics for each query can be prohibitively expensive; and (3) it is hard to determine the most crucial statistics for the optimization process.

Numerous efforts have been proposed to address these issues. For example, reactive techniques monitor operators' cardinalities at run time and detect large estimation errors. Reacting to estimation errors may involve re-optimizing the running query [9, 11] or adjusting stored statistics to compensate for these errors in future queries [14, 1]. In contrast, proactive techniques try to minimize the optimizer's mistakes by collecting statistics at compilation time [6] or by keeping multiple query plans and using the cardinalities monitored at run time to choose among them [3]. Section 5 gives more details on related work.

In this paper, we propose an efficient approach to proactively determine, collect, and materialize Just-in-Time Statistics (JITS) for the currently optimized query. In contrast to earlier attempts, our approach: (1) employs a lightweight sensitivity analysis based on the query structure, the existing statistics and the data activity to identify the crucial statistics; and (2) materializes and incrementally updates the collected partial statistics for future reuse. Materializing and reusing query-specific statistics is a challenging problem since the statistics cover partial (possibly overlapping) regions of the data space. JITS integrates these partial statistics in a reusable form by maintaining maximum-entropy-based structures. Our proposed framework can easily be extended to employ more sophisticated sensitivity analysis techniques. We implemented a prototype in the IBM® DB2® Universal Database™ product (DB2 UDB) that demonstrates our approach and evaluates its benefits through an extensive experimental study.

The rest of this paper is organized as follows: Section 2 discusses the concept of Query-Specific Statistics. We explain the JITS framework in Section 3 and discuss our experimental results in Section 4. Section 5 outlines similar efforts done in query optimization, and Section 6 concludes our work.

2. Query-Specific Statistics

As mentioned in Section 1, the assumptions made by traditional optimizers (independence and data uniformity) are often not true, and usually yield high estimation errors. This raises the need for *Query-Specific Statistics (QSS)*, which take into consideration the predicates and the values used in a particular query. QSS allow the optimizer to accurately estimate the cost of different execution (sub)plans, while minimizing the assumptions made on the underlying data. For example, consider a simple query that selects all employee records with $salary > \$100k$ and $age > 30$ from Table $Employees$. A QSS for this query is an accurate estimate of joint selectivity of the two predicates, $salary > \$100k$ and $age > 30$, to allow the optimizer to choose the best access path to Table $Employees$. However, even with the existence of histograms on Columns age and $salary$, the optimizer assumes uniformity within each bucket. Furthermore, since the optimizer cannot maintain multi-dimensional histograms for all possible combinations of columns, the optimizer may assume independence to compute the joint selectivity of the two predicates. These assumptions often lead to huge errors in cost estimation. The problem is further magnified with more complex queries that involve large numbers of predicates and joins.

Note that predicate selectivities, in addition to being query-specific, can also be *plan-specific*; the selectivity of a particular predicate can be useful for costing a certain plan but useless for costing another plan representing the same query. For example, consider a query that contains a join of 3 tables A, B, and C. In one possible plan, the join order is $(A \bowtie B) \bowtie C$. This plan needs the selectivity of the join predicate of $A \bowtie B$. Another plan with a different join order $A \bowtie (B \bowtie C)$ would need the selectivity of the join predicate of $B \bowtie C$. Hence, in general, collecting all needed statistics would involve collecting the selectivity of all possible join predicates and other plan-specific statistics, which is prohibitively expensive.

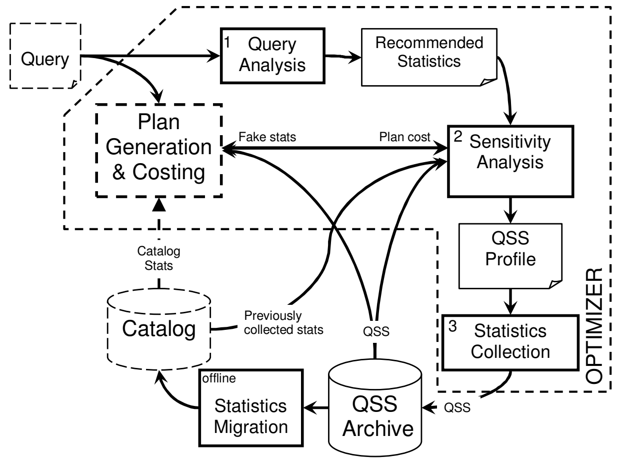

Figure 1. JITS architecture

QSS can be viewed as a compromise between: (1) general statistics currently collected by query optimizers, where multiple unrealistic assumptions are made to generate the required statistics for cost estimation; and (2) plan-specific statistics that can be directly used in cost estimation, eliminating the need for assumptions but suffering from prohibitively expensive collection cost.

A system that collects and exploits QSS in query optimization needs to address the following issues: (1) which QSS to collect among a large number of possible candidates; and (2) how to efficiently materialize the collected QSS for later reuse. Section 3 describes the JITS framework, and how it tackles these issues.

3. JITS Framework

In this section, we describe JITS, a system for proactively collecting query-specific statistics during query compilation. We give the overall architecture of JITS, describe the structure of the QSS archive, and give details of our implementation of the various modules.

3.1. System Architecture

Figure 1 gives a strawman architecture of JITS. Entities in dotted lines already exist in current query engines, while entities in solid lines are new JITS modules.

Algorithm 1 Query analysis

```
1: procedure QueryAnalysis($Q$)
2:     $PG \leftarrow \{\}$
3:     $B \leftarrow$ set of query blocks in $Q$
4:     for all $b \in B$ do
5:         $P \leftarrow$ set of predicates in $b$
6:         $T \leftarrow$ set of tables involved in $b$
7:         for all $t \in T$ do
8:             $P_t \leftarrow \{p | p \in P, p$ is local on $t\}$
9:             for $i \leftarrow 1, |P_t|$ do
10:                $G \leftarrow$ all $i$-pred. groups; predicates $\in P_t$
11:                $PG \leftarrow PG \cup G$
12:            end for
13:        end for
14:    end for
15:    return $PG$
16: end procedure
```

Algorithm 2 Sensitivity analysis

```
1: procedure SensitivityAnalysis($Q, PG$)
2:     $T \leftarrow$ set of tables involved in $Q$
3:     for all $t \in T$ do
4:         $PG_t \leftarrow \{g | g \in PG, g$ is local on $t\}$
5:         if ShouldCollectStats($t, PG_t$) then
6:             Mark $t$ for statistics collection
7:             for all $g \in PG_t$ do
8:                 if ShouldMaterialize($g$) then
9:                     Mark $g$ for materialization
10:                end if
11:            end for
12:        end if
13:    end for
14: end procedure
```

The *Query Analysis* module analyzes the query structure, after parsing and rewriting, to determine all relevant statistics, and generates a list of candidate statistics. The *Sensitivity Analysis* module processes the candidate statistics to decide the most crucial statistics to collect. Our implementation examines the query, existing statistics, as well as the history of data activity (e.g., frequency of updates and deletes on a particular table). In general, the sensitivity analysis module can use any more sophisticated technique, e.g., by incorporating the planning module to determine the sensitivity of the query to a particular statistic [6]. The *Statistics Collection* module collects the required statistics, and uses them to update the QSS archive. The Plan Generation and Costing module uses the information in the QSS archive and the system catalog to select an execution plan. The information in the QSS archive can be used to periodically update the system catalog using the *Statistics Migration* module.

The QSS archive is a repository of adaptive single- and multi-dimensional histograms. Categorical and character data types can be represented as numerical values using a mapping function to allow for interpolation. We maintain a time stamp with each bucket, which is used by the sensitivity analysis module to determine the recentness of the statistics. We elaborate on the details of the histograms and their update strategy in Section 3.4.

3.2. Query Analysis

The *Query Analysis* module determines which statistics are relevant to the query, regardless of whether or not they should be collected. These statistics can be classified as: (1) table statistics (e.g., number of rows), which are needed for every table involved in the query, and (2) column statistics, which basically include the selectivities of predicates or groups of predicates.

The query analysis (Algorithm 1) takes as input a query Q and returns the set PG of candidate predicate groups on which statistics are needed in order to optimize Q. Each element in PG is a group of predicates that appear in the query. The algorithm analyzes the query by examining its internal structure after parsing and rewrite. It enumerates all possible combinations of predicates belonging to each table involved in the query. For each table t, it considers groups of i-predicates, for $i = 1, 2, ..., m$ (m is the number of local predicates on this table). For example, consider the following query:

```
SELECT price FROM car
WHERE make = 'Toyota'
AND model = 'Corolla'
AND year > 2000
```

Applying the query analysis algorithm, the set P_t will contain the predicates *(make = 'Toyota')*, *(model = 'Corolla')* and ($year > 2000$). The first iteration of the loop in line 9 produces all single predicates. The second iteration produces 3 groups of 2 predicates each, and the last iteration produces a single group with 3 predicates.

Since the aim of QSS is to be directly used by the optimizer, Algorithm 1 collects predicate groups per query block (SPJ block), since most optimizers, including our prototype DBMS, perform intra-block optimization. Note that the input to the algorithm is the query after rewrite, so the query blocks are finalized and ready to be processed by the optimizer.

3.3. Sensitivity Analysis

Statistics collection during query compilation is an expensive process. Collecting all statistics recommended by the query analysis module is not always necessary, so it is crucial to decide which statistics are necessary to collect.

The sensitivity analysis (Algorithm 2) takes as input the query Q and the list of predicate groups recommended by the query analysis module. The algorithm makes use of two other algorithms: Algorithm $ShouldCollectStats(t, PG_t)$ (Algorithm 3) determines if statistics should be collected on a table t based on the candidate statistics PG_t. Ideally, the sensitivity analysis evaluates the "importance" of each candidate statistic. We adopt a simplification heuristic that decides on all the statistics PG_t of a table as one unit. The rationale is that most of the cost of computing the statistics is in the sampling process. Once a table is sampled, it is relatively cheap to collect the selectivities of all predicate groups that belong to this table. However, to meet the space constraints, it is not always necessary to materialize all the collected statistics, especially if they involve creating new QSS histograms. Algorithm $ShouldMaterialize(g)$ (Algorithm 4) determines if a certain predicate group g should be materialized. The details of the two subroutines are given in Sections 3.3.2 and 3.3.3, respectively.

3.3.1 Data Structures

The sensitivity analysis module maintains two main data structures:

UDI counter: For each table, we maintain a counter that encapsulates the number of *updates*, *deletions* and *insertions* that took place since the last statistics collection on this table. We use the UDI counter as an indication of the change in the data.

StatHistory: The optimizer can estimate the selectivity of conjuncts of predicates such as $sel(p_1 \wedge p_2 \wedge p_3 \wedge p_4)$ by using partial selectivities such as $sel(p_1)$, $sel(p_2 \wedge p_3)$, and $sel(p_2 \wedge p_3 \wedge p_4)$ [10]. We maintain a history of statistics collection that we use to evaluate the effectiveness of the optimizer's assumptions in computing column group statistics from partial statistics. Each entry is of the form $(T, colgrp, statlist, count, errorfactor)$, where T is the table name to which the column group belongs; $colgrp$ are the columns in the group; $statlist$ is a set of column groups that were used to estimate the selectivity of this group; $count$ is the number of times that the column groups in $statlist$ have been used to compute the selectivity of this group; and $errorfactor$ is the estimated selectivity divided by the actual selectivity.

The $errorfactor$ is usually provided by a feedback system that monitors the actual selectivities and compares them to the optimizer's estimates. In our case, we use the output of *LEO* [14] to provide the $errorfactor$ values in the $StatHistory$. For example, suppose that the selectivity of the predicate group *(a=5 AND b>10 AND c<100)* was estimated to be 0.2 using a histogram on (a, b) and a histogram on *c*. However, during the query execution, the actual selectivity was 0.5. Thus $errorfactor = 0.2/0.5 = 0.4$. Table 1 gives a sample of the stored $StatHistory$.

Table 1. Statistics collection history

T	colgrp	statlist	count	errorfactor
$T1$	(a, b, c)	$\{(a, b), (c)\}$	5	0.4
$T1$	(a, b, c)	$\{(a), (b, c)\}$	2	0.7
$T1$	(a, b, c)	$\{(a, b, c)\}$	10	0.98
$T1$	(a, b, d)	$\{(a, b), (d)\}$	4	0.8

3.3.2 Determining Crucial Statistics

Deciding whether or not to collect statistics on a particular table is mainly based on evaluating two metrics: s_1 reflects the accuracy of currently existing statistics on this table; and s_2 reflects the data activity on the table. Each of the two metrics can be viewed as a value ranging from 0 to 1; where 0 means that no statistics collection is needed and 1 meaning that statistics must be collected.

Computing these scores is described in Algorithm 3. To compute s_1, the algorithm first gets the predicate group that has the maximum number of predicates (i.e., the one with all predicates), and fetches all the history entries that refer to this group. For example, if this group is *(a=5 AND b>10 AND c<100)*, and the history is as shown in Table 1, then H will have the first 3 entries. For each of these entries, Algorithm 3 calculates the accuracy ($accu$) of using $statlist$ to estimate the selectivity of $colgrp$. The accuracy depends on the $errorfactor$ value in that entry, as well as the accuracy of each of the $statlist$ elements. We show how to calculate the accuracy in case of histograms later in this section. s_1 can be calculated as *(1 - the maximum accuracy)*. The second metric, s_2, can be calculated as the ratio between the UDI and the table cardinality.

The total score of the table is computed as an aggregate function of the two metric values. One way to use the aggregated score is to use a threshold of statistic importance; if the value of the total score exceeds a threshold s_{max}, statistics must be collected on this table. As s_{max} approaches 1, no QSS are collected during compilation. As s_{max} decreases, the system becomes more aggressive and tends to collect all possible QSS. In our implemented prototype, the aggregate function is the average of the two scores. Section 4.3 elaborates on the effect of changing the value of s_{max} on the system performance.

Algorithm 3 Is a particular table important?

```
1:  procedure ShouldCollectStats($t, PG$)
2:      $g \leftarrow$ group from $PG$ with the max. # of predicates
3:      $H \leftarrow \{h | h \in history; h.T = t, h.colgrp = g\}$
4:      $MaxAcc \leftarrow 0$
5:      for all $h \in H$ do
6:          $n \leftarrow$ no. of elements in $h.statlist$
7:          $accu \leftarrow h.errorfactor$
8:          for $i \leftarrow 1, n$ do
9:              $accu \leftarrow accu * accuracy(h.statlist[i], g)$
10:         end for
11:         if $accu > MaxAcc$ then
12:             $MaxAcc \leftarrow accu$
13:         end if
14:     end for
15:     $s_1 \leftarrow (1 - MaxAcc)$
16:     $s_2 \leftarrow min(UDI(t)/cardinality(t), 1)$
17:     $score \leftarrow f(s_1, s_2)$
18:     if $score \geq s_{max}$ then
19:         Return TRUE
20:     else
21:         Return FALSE
22:     end if
23: end procedure
```

Since computing s_1 depends on calculating the accuracy of the underlying statistics, we show how to compute the accuracy of a histogram. The accuracy of a histogram with respect to a predicate (group) is a value in the range [0,1] that represents how accurately the selectivity of this predicate (group) can be estimated from this histogram.

Consider a one-dimensional histogram on column a. A histogram has n buckets $B_1, B_2, ..., B_n$. Bucket B_i lies between the boundaries b_{i-1} and b_i. Now consider a predicate $a > value$. If $value \approx b_i$ for some i, then estimating the selectivity is very accurate. As $value$ gets further from any boundary, the accuracy of the histogram decreases. The accuracy further decreases if $value$ lies within a wide bucket.

To calculate the accuracy on a single dimension, we follow these steps:

1. Locate the bucket that contains $value$. Let this bucket be B_j with boundaries b_{j-1} and b_j.
2. Let $d_1 = value - b_{j-1}$ and $d_2 = b_j - value$
3. Let $u = \frac{min(d_1,d_2)}{max(d_1,d_2)} * \frac{b_j - b_{j-1}}{b_n - b_0}$
4. $accuracy = 1 - u$

For multi-dimensional histograms, we use a simple method where the overall accuracy can be computed as the product of the accuracy in each dimension.

Algorithm 4 Is a statistic useful for other queries?

```
1:  procedure ShouldMaterialize($g$)
2:      if histogram exists on $g$ then
3:          Return TRUE
4:      end if
5:      $F = \sum_{history} count$
6:      $H = \{h | h \in history; g \in h.statlist\}$
7:      $score \leftarrow 0$
8:      for all $h \in H$ do
9:          $score \leftarrow score + (h.errorfactor * h.count/F)$
10:     end for
11:     if $score \geq s_{max}$ then
12:         Return TRUE
13:     else
14:         Return FALSE
15:     end if
16: end procedure
```

3.3.3 Which Statistics to Materialize?

Once a table is sampled, the selectivities of all the candidate predicate groups given by the query analysis are computed and are used to optimize the query. However, we must decide which of these statistics to store in the QSS archive to be reused by future queries, i.e., which statistics can be useful for later queries. JITS estimates the usefulness of materializing given statistics by monitoring how useful they were for previous queries. The usefulness score of a statistic depends on the number of times this particular statistic has been used, and the accuracy of the estimates computed using it. Algorithm 4 lists all history entries that have the statistic in question as one of their $statlist$ elements. For example, if the statistic in question is the predicate group *(a=5 AND b>10)*, and the history is as shown in Table 1, then H will have the first and fourth entries. The statistic is given a score that represents how beneficial it was for computing needed statistics. The algorithm uses a weighted average of $errorfactor$ to compute this score. If this score exceeds a certain threshold, it is considered useful, and is marked for materialization.

3.4. Updating the QSS Archive

To keep them up-to-date, the histograms in the QSS archive are updated whenever new statistics are collected. The newly collected statistics are a set of buckets, each of which has the form $(dimension_1, ..., dimension_n, count)$. This information is obtained from the query predicates. The general form of a predicate is *(exp between A and B)* where one or both of A and B can be present. The update process is based on the maximum entropy principle. We extended the technique in [13] to update the histograms by finding

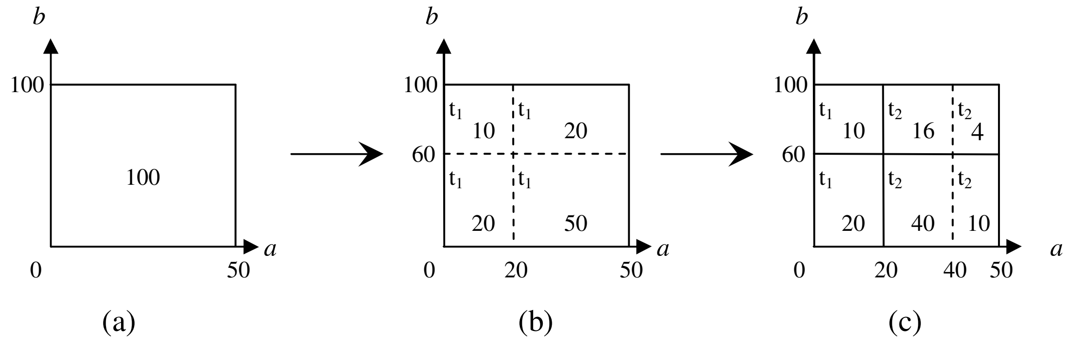

Figure 2. Histogram update

a distribution that satisfies the knowledge gained by the new statistics without assuming any further knowledge of the data, i.e., assuming uniformity unless more information is known. Note that exp can be any expression involving table columns, although in the current prototype we limit exp to be a column name. However, A and B have to be constant values to be used for updating the histograms. For instance, for columns a, b, c, the predicate *(a between* $b + 10$ *and* $c - 20$*)* cannot be used to update the histogram using maximum entropy[1].

For example, consider a 2-dimensional histogram on attributes a and b. The values of a range from 0 to 50 and the values of b range from 0 to 100. The total number of tuples is 100. Initially the histogram has just one bucket, as shown in Figure 2(a). Now consider a query with the predicates ($a > 20$ AND $b > 60$). After sampling, the system finds that 20 tuples satisfy this predicate group. This information is used directly to optimize the query. However, from the same sample we can determine the number of tuples that satisfy each of the 2 predicates individually (assume it is 70 and 30, respectively). This new information is used to update the histogram as in Figure 2(b). The 4 resulting buckets get a new time stamp (shown in the upper left corner of each bucket). Assume another query that has the predicate ($a > 40$), and assume there are 14 tuples satisfying this predicate. Since no further information is known, we assume uniformity within the buckets in the last histogram. As a result, the newly inserted boundary splits the buckets as shown in Figure 2(c). The time stamp of the 4 new buckets (on both sides of the dotted line) is updated.

As the system collects QSS, storage space becomes an issue, especially since a single column can be involved in multiple histograms, each of which can be arbitrarily large. To avoid this problem, we keep a limit on the size of QSS to maintain. In case the dedicated space is full, and more statistics have to be materialized, we remove the histograms that are almost uniformly distributed (as they are close to the optimizer's assumptions). In case more than one histogram satisfies this property, we use *LRU (Least Recently Used)* policy to choose, assuming that the each histogram is associated with the last time it has been used by the optimizer.

[1]We can store such predicates and the number of tuples that satisfy them separately, and possibly reuse them for later queries. LRU can be used to prune unused predicates.

3.5. JITS Applicability

It is worth mentioning that JITS is more useful for complex, long-running queries such as those used in OLAP and Decision Support Systems. Such queries usually include a relatively large number of joined tables, aggregate functions, and predicates, which means more alternative plans to choose from. The long running time of these queries justifies spending time on statistics collection to guarantee the optimizer's access to recent accurate statistics, thus bringing down the total response time of the query. On the other hand, simple OLTP queries usually do not involve a large number of tables, and their running time is usually very short. For this reason, they might not benefit much from using the approach presented in this paper. In fact, using such architecture can increase the time of query processing if all the queries are very simple. This is further illustrated in our experimental study.

4. Experimental Evaluation

We implemented the prototype within DB2 UDB. The database that we generated contains four relations: `CAR, OWNER, DEMOGRAPHICS, and ACCIDENTS`. Several primary-key-to-foreign-key relationships exist between the tables, as well as a number of correlations between attributes, such as `Make` and `Model`. Table 2 shows the number of tuples in each of the four tables.

The prototype uses the Query Graph Model (QGM) [7] to analyze the query structure. For statistics collection, the prototype invokes the RUNSTATS tool with the appropriate

Table 2. Table sizes

Table	No. of Tuples
CAR	1,430,798
OWNER	1,000,000
DEMOGRAPHICS	1,000,000
ACCIDENTS	4,289,980

parameters. Based on earlier work [1, 8, 12], the best sample size sufficient to give accurate statistics of a database table is independent of the table size, and thus can be scaled to large tables. To collect specific predicate selectivities, we had to construct and invoke sampling queries on-the-fly.

As a future extension, techniques such as the work described in [15] can be employed to reduce the time used for sampling by making use of the existing catalog statistics.

4.1. JITS for a Single Query

To evaluate the benefit of our model, we issued a query given different scenarios. In this experiment, the automatic sensitivity analysis module was turned off. The query used for this experiment is:

```
SELECT o.name, driver, damage
FROM car as c, accidents as a, demographics
as d, owner as o
WHERE d.ownerid = o.id
AND a.carid = c.id
AND c.ownerid = o.id
AND make = 'Toyota' AND model = 'Camry'
AND city = 'Ottawa' AND country = 'CA'
AND salary > 5000
```

This query was issued in 4 different scenarios:

1. No initial statistics (a) with JITS disabled, and (b) with JITS enabled
2. With Initial basic and distribution statistics on all tables (a) with JITS disabled, and (b) with JITS enabled

Table 3 shows the compilation, execution, and total times of the query under the different cases. The times shown are in seconds. Note that the total time is slightly larger than the sum of the compilation and execution times because it also includes the fetch time, which is the same in all cases.

In the first 2 cases, no statistics are known initially. When JITS is enabled (case 1-b), some overhead is encountered for collecting statistics. However, the execution time decreases significantly. The decrease in execution time is almost 27% and the overall gain is around 18% reduction in total query time. In the existence of all recent general statistics, JITS might not outperform the traditional model for a single query. The reason is that the saving in the execution time can be overweighed by the JITS overhead. However, once we consider a sequence of queries in a workload, the overhead is amortized by reusing the statistics in the QSS archive. We show the workload effect in Section 4.2.

Table 3. Compilation and execution times

Case #	Compilation	Execution	Total
1-a	0.098	138.756	138.855
1-b	11.864	101.681	113.547
2-a	0.073	104.912	104.986
2-b	3.698	101.323	105.023

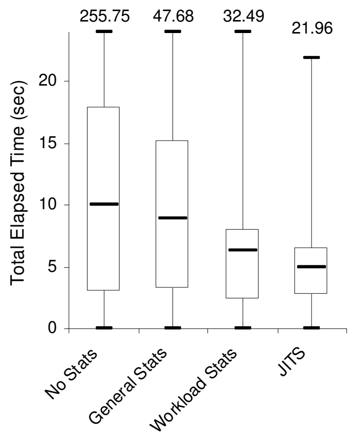

Figure 3. JITS benefit

4.2. JITS for a Workload

This experiment demonstrates the performance of JITS as opposed to traditional query processing. We observed the performance of the system using a workload of 840 queries, including data updates to simulate a real-world operational database. The workload was executed in four settings:

1. JITS disabled, having no initial statistics
2. JITS disabled, having general (basic and distribution) statistics about all tables and columns
3. JITS disabled, having general (basic and distribution) statistics about all tables and columns in addition to workload statistics (i.e., all column groups that occur in all the queries)
4. JITS enabled, having no initial statistics

Figure 3 shows a box plot (a graph depicting the smallest observation, lower quartile, median, upper quartile and largest observation) of the elapsed time of the workload

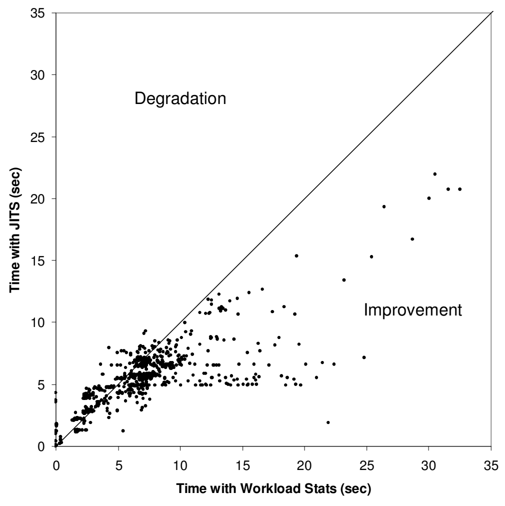

Figure 4. Workload stats vs. JITS

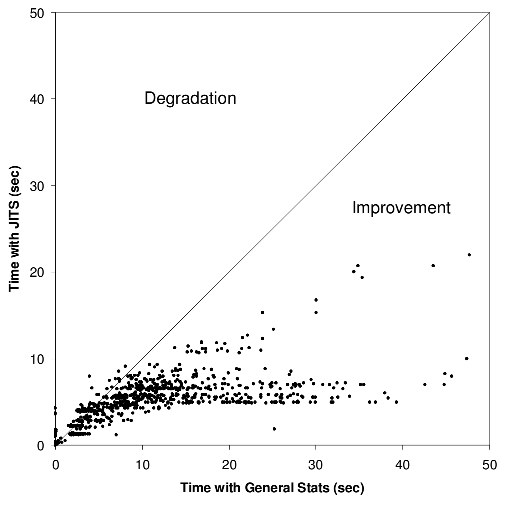

Figure 5. General stats vs. JITS

queries in the four settings. The results show that having general statistics only results in a slight benefit. However, if the workload information is available, it can be analyzed and all the needed statistics can be collected beforehand, which improves the overall performance. However, even in this case, the statistics are general statistics, not query-specific, i.e., the optimizer still has to make some uniformity and/or independence assumptions. In addition, due to data updates, these statistics soon become stale, and the estimation error increases. With JITS enabled, the system samples the data to get the actual selectivities of the predicates in the query. The benefit of having these very accurate values outweighs the sampling overhead. In addition, the data updates have no effect on the accuracy of the collected statistics, since the system detects the staleness of these statistics, and recollects them when needed. Figure 3 shows that JITS outperforms the other cases.

Figure 4 shows a scatter chart of the elapsed times of individual queries when JITS is enabled (with no prior statistics) versus when JITS is disabled (having workload statistics to start with). Some of the queries, especially in the beginning of the workload, suffer from the overhead of collecting statistics when JITS is enabled. As the data gets updated, the workload statistics become stale, and the benefits of JITS become evident. However, in the majority of the systems, where prior workload knowledge is not available, only general statistics can be collected initially. Figure 5 depicts JITS versus having general statistics. Almost all of the queries have a significant improvement, while only a few ones lie in the degradation region.

4.3. Tuning the Sensitivity Analysis

As mentioned in Section 3.3.2, the sensitivity analysis module determines whether or not to collect certain statistics. Each statistic is given an overall score based on individual scoring factors. A statistic is to be collected if its overall score exceeds a certain threshold s_{max}.

We used the same workload used in Section 4.2. Figure 6 shows the average elapsed time per query for $s_{max} = 0, 0.1, 0.5, 0.7, 0.9$, and 1. At $s_{max} = 0$, all possible statistics are always collected, i.e., there is no actual sensitivity analysis. This explains the very large compilation time. The compilation time decreases as s_{max} increases since fewer statistics are collected. At $s_{max} = 1$, no statistics are ever collected. Note that if there is no sensitivity analysis ($s_{max} = 0$), our system performs worse than traditional query optimization ($s_{max} = 1$) because of the added overhead. Increasing s_{max} from 0 to 0.5 decreases the average compilation time significantly while the average execution time is not affected, which means that there has been useless statistics collection at the lower values of s_{max}. At $s_{max} = 0.7$, there is an increase in the average execution time, outweighed by the decrease in the average compilation time. This means that setting $s_{max} = 0.7$ might be a better choice if we have a workload. However, $s_{max} = 0.5$ would be better for a single query (where the system collects the minimum amount of statistics to achieve the least possible execution time).

5. Related Work

Most of the published work regarding the optimizer's dependence on statistics only addresses the problem of stale

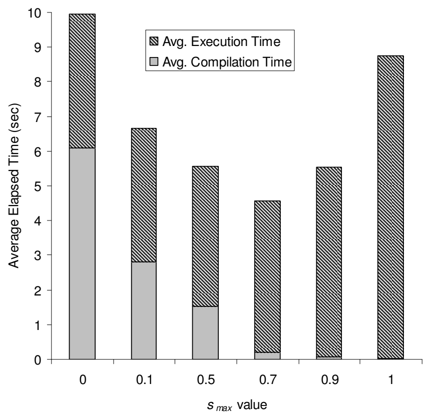

Figure 6. Sensitivity analysis threshold

or outdated database statistics and their effect on query optimization. The problem of deciding which statistics to collect has not been thoroughly explored. The approaches that tackle the statistics aspect of cost-based optimization can be categorized mainly as being either *reactive* or *proactive*.

Reactive approaches are based on monitoring a query *during* execution, and reacting to observed errors between the initial estimates and the actual values from the query feedback. In contrast, proactive approaches try to predict, identify and possibly solve potential problems by doing additional work *before* query execution.

5.1. Reactive Approaches

The first class of reactive approaches reacts to observed estimation errors by correcting the stored statistics to benefit future queries. The *Learning Optimizer (LEO)* in DB2 UDB [14] keeps track of actual cardinality values along the edges of the query plan. These values are compared to the estimates used by the optimizer. The error is used as an adjustment factor to correct statistics for future queries. A slightly different approach [1] uses the error to trigger statistics collection if it exceeds a certain threshold.

However, the current query still suffers from wrong estimates. The problem is further magnified in systems with long-running ad hoc unrelated queries, in which case the adjustment in statistics is unlikely to be used by future queries.

The second class of reactive approaches reacts to errors by re-optimizing the current query. In [9], if the optimizer decides that the execution plan is suboptimal, it tries to optimize execution, either by changing the resource allocation or by changing the execution plan and re-executing the query using the new plan. In *Progressive OPtimization (POP)* [11], the optimizer starts with the existing statistics, and chooses a plan. The optimizer then calculates a validity range for the cardinality at each step of execution, in which this plan is still optimal. During execution, if the actual values are outside the validity ranges, the current plan is not optimal anymore, and the query is re-optimized.

However, it is hard to collect actual statistics during query execution without blocking the execution pipeline. Furthermore, the decision to re-optimize raises a question whether to reuse the part that has been executed already or to start execution from the beginning with the new plan.

5.2. Proactive Approaches

Babu et al. [3] proposed an approach that is partly proactive but mostly reactive. This approach is based on the possible error in the cardinalities at every edge of the query plan. At each operator, the system computes the possible range of values for the two input relations to this operator, getting *bounding box*. The system maintains three alternative sub-plans for each operator; ones that are optimal at the lowest, middle, and highest points in the bounding box. During execution, the approach is reactive; the system detects the actual cardinality and chooses one of these sub-plans accordingly if the value is inside the bounding box; otherwise, it re-optimizes the query. The main problem is that the three maintained sub-plans do not necessarily cover the whole spectrum of possible sub-plans, i.e., there could be another plan that is better than the selected three plans for a particular input cardinality value.

To the best of our knowledge, the only work that addresses the issue of choosing which statistics to collect before query execution – and hence the closest to our work – is the approach proposed in [6]. It includes a technique to perform sensitivity analysis in order to decide which statistics to collect so that the optimizer will have enough information to optimize that query. The idea is to check whether the currently available set of statistics is sufficient or not. If not, then collect the most important statistic, and then repeat the check again until the available statistics are sufficient. The decision of whether the current set of statistics is sufficient or not is taken by invoking the optimizer twice. In the first invocation, all unknown selectivities are set to a very small value $\varepsilon > 0$. In the second invocation, all unknown selectivities are set to a large value $1 - \varepsilon$. If the estimated costs of the two generated plans are within $t\%$ of each other (for a predefined value of t), the current set of statistics is sufficient. If not, the system identifies the *most important* statistic by calling the optimizer again to get an execution tree based on the current set of statistics, then comparing the estimated costs of the operators in the tree, assuming that expensive operators are associated with important statistics.

However, this approach differs from ours in multiple aspects. It requires multiple calls to the optimizer for every statistic, which can be very time-consuming especially for complex queries, whereas our approach employs a lightweight sensitivity analysis to limit the overhead. Moreover, although the framework in [6] can be particularly useful if most of the tables involved in the query have up-to-date statistics, it would not be as useful if most of the statistics are outdated. This is because it decides the importance of statistics based on the estimated operator cost in the execution tree that is already built using inaccurate information. In addition, our approach relies on collecting and reusing *Just-in-Time Statistics* that can be query-specific.

Other approaches tackling different aspects of the query optimization problem include generating robust plans that are resilient to estimation errors [2], or detecting the correlation in the data to avoid the independence assumption [8]. A different approach makes use of statistics on query expressions to optimize the whole query [4, 5]. Several efforts have been proposed to eliminate the independence assumption by estimating the selectivity of a group of predicates from smaller groups or individual predicates [10, 16].

6. Conclusion and Future Work

Collecting Just-in-Time Statistics guarantees that the optimizer has access to statistics that are recent and truly representative of the underlying data, including any possible correlations. The presence of these statistics minimizes the effect of the optimizer's uniformity and independence assumptions, and significantly reduces cost estimation error. Using this approach is more useful for complex long-running queries such as the ones used in OLAP and Decision Support Systems since investing some time to collect statistics significantly reduces the total query response time.

For future work, we would like to consider a more sophisticated sensitivity analysis technique to determine which statistics to collect. It would also be interesting to investigate methods to further reduce the time spent on statistics collection by integrating catalog statistics with sampled data, and/or inferring some of the absent statistics.

7. Acknowledgments

We thank Torsten Ziegler from SAP, who inspired our thinking about Just-in-Time statistics. We also thank John Hornibrook, Qi Cheng, Ivan Popivanov, Xiaoyan Qian, Matthew Huras and Berni Schiefer for their feedback and comments, which have greatly improved the prototype, as well as Peter Haas, who steered us towards using a decision tree for the sensitivity analysis.

References

[1] A. Aboulnaga, P. J. Haas, M. Kandil, S. Lightstone, G. M. Lohman, V. Markl, I. Popivanov, and V. Raman. Automated Statistics Collection in DB2 UDB. In *Proc. 30th VLDB conference*, pages 1146–1157, 2004.

[2] B. Babcock and S. Chaudhuri. Towards a Robust Query Optimizer: A Principled and Practical Approach. In *Proc. 2005 ACM SIGMOD*, pages 119–130, 2005.

[3] S. Babu, P. Bizarro, and D. DeWitt. Proactive Re-optimization. In *Proc. 2005 ACM SIGMOD*, pages 107–118, 2005.

[4] N. Bruno and S. Chaudhuri. Exploiting Statistics on Query Expressions for Optimization. In *Proc. 2002 ACM SIGMOD*, pages 263–274, 2002.

[5] N. Bruno and S. Chaudhuri. Conditional Selectivity for Statistics on Query Expressions. In *Proc. 2004 ACM SIGMOD*, pages 311–322, 2004.

[6] S. Chaudhuri and V. Narasayya. Automating Statistics Management for Query Optimizers. *IEEE Transactions on Knowledge and Data Engineering*, 13(1):7–20, 2001.

[7] L. M. Haas, J. C. Freytag, G. M. Lohman, and H. Pirahesh. Extensible Query Processing in Starburst. In *Proc. 1989 ACM SIGMOD*, pages 377–388, 1989.

[8] I. F. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic Discovery of Correlations and Soft Functional Dependencies. In *Proc. 2004 ACM SIGMOD*, pages 647–658, 2004.

[9] N. Kabra and D. J. DeWitt. Efficient Mid-query Re-optimization of Sub-optimal Query Execution Plans. In *Proc. 1998 ACM SIGMOD*, pages 106–117, 1998.

[10] V. Markl, N. Megiddo, M. Kutsch, T. M. Tran, P. J. Haas, and U. Srivastava. Consistently Estimating the Selectivity of Conjuncts of Predicates. In *Proc. 31st VLDB Conference*, pages 373–384, 2005.

[11] V. Markl, V. Raman, D. Simmen, G. Lohman, H. Pirahesh, and M. Cilimdzic. Robust Query Processing through Progressive Optimization. In *Proc. 2004 ACM SIGMOD*, pages 659–670, 2004.

[12] V. Poosala, Y. E. Ioannidis, P. J. Haas, and E. J. Shekita. Improved Histograms for Selectivity Estimation of Range Predicates. In *Proc. 1996 ACM SIGMOD*, pages 294–305, 1996.

[13] U. Srivastava, P. J. Haas, V. Markl, N. Megiddo, M. Kutsch, and T. M. Tran. ISOMER: Consistent Histogram Construction Using Query Feedback. In *Proc. 22nd ICDE*, 2006.

[14] M. Stillger, G. M. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. 27th VLDB Conference*, pages 19–28, 2001.

[15] X. Yu, N. Koudas, and C. Zuzarte. HASE: A Hybrid Approach to Selectivity Estimation for Conjunctive Predicates. In *Proc. 10th EDBT Conference*, pages 460–477, 2006.

[16] X. Yu, C. Zuzarte, and K. C. Sevcik. Towards Estimating the Number of Distinct Value Combinations for a Set of Attributes. In *Proc. 14th CIKM*, pages 656–663, 2005.

Trademarks