A Framework for Re-optimizing Repetitive Queries

Feng Yu, Wen-Chi Hou
Department of Computer Science
Southern Illinois University
Carbondale, IL 62901
{fyu, hou}@cs.siu.edu

Cheng Luo
Department of Mathematics and Computer Science
Coppin State University
Baltimore, MD 21216-3698
cluo@coppin.edu

ABSTRACT
In this paper, we develop a comprehensive framework for re-optimization of a large and useful set of queries, called repetitive queries. Repetitive queries refer to those queries that are likely to be used repeatedly or frequently in the future. They deserve more optimization efforts than ordinary ad hoc queries. In this research, we identify statistics, called sufficient statistics, that are sufficient to compute the exact frequency distributions of the intermediate results of all plans of a query. We present two innovative techniques to conduct re-optimization, an eager and a lazy re-optimization. The eager approach gathers all the sufficient statistics for a query at once and generates the best plan. The lazy re-optimization gathers only the statistics that are needed to correct large estimation errors found in the plan and generates a revised plan. We further adapt the two basic techniques to constantly changing environments by continuously monitoring and revising the plans, called adaptive re-optimization. The adaptive re-optimization is devised to detect and remedy potential sub-optimality in the plans in a timely manner for the entire lifetime of the query. Our work realizes the promise made by the query optimizers, namely, executing queries in the optimal fashions, at least for the repetitive queries.

1. INTRODUCTION
A query optimizer generally uses statistics on databases [7, 9, etc.] and assumptions about attribute values [3, 11] to estimate the cost of alternative plans and selects the best for what is important for an optimizer ofto find the most efficient plans because studies [3, 6] have shown that executions with sub-optimal plans can be orders of magnitude slower than with the optimal ones. Unfortunately, due to the sufficiency of statistics stored in the database and the validity of assumptions made, query optimizers often cannot find the plan that is truly the best given the search spaces for the queries. Thus, some database systems, like Sybase and Oracle, allow users to force the join orders; some, e.g., Sybase, even allow users to explicitly edit the plans [10]. Unfortunately, such measures cannot guarantee success and can become cumbersome and slow for complex queries.

Query re-optimization aims to refine the execution plans of queries. It can have a tremendous impact on the performance of systems. Unfortunately, there hasn’t been much work on this subject yet. In the literature, some [1, 5, 6, 8] have focused on refining execution plans of ongoing queries on-the-fly, while others [12, 2] focused on refining cost estimation of future queries using statistics collected in previous executions. In this paper, we are interested in refining cost estimation and execution plans for future queries, similar to the latter.

Stillger et al. [12] collected cardinality information from queries and used it to adjust the cost estimation of future queries. Unfortunately, statistics, like cardinalities and selectivities, obtained from one plan may not be sufficient for estimation of another plan because as the orders of joins change, the inputs of the joins and the outputs thereof are all changed; certainly, they can hardly be sufficient for estimation of different queries either. The situation is further exacerbated by adding other operators, such as selects, projects, unions, and differences, to the queries. It can require prohibitively large amounts of statistics, if ever possible, to compute the intermediate results’ sizes for all possible plans of all queries. Therefore, in this research, we set a more realistic goal by restricting ourselves to optimizing a subset, but a large and useful subset, of queries, called repetitive queries.

There are also many useful queries, such as those used for generating periodic reports, performing routine maintenance, summarizing and grouping data for analysis, etc., that are run frequently, periodically, or repeatedly, hereby called repetitive queries. They are often stored in the database for convenient reuse for the long term. They can constitute a large portion of the daily activities. Any sub-optimality in the execution plans of repetitive queries could mean repetitive and continued wastes of system resources and time in the future. Repetitive queries have profound impacts on the performance of systems and deserve more optimization efforts than ad hoc queries.

In this research, we identify statistics, called sufficient statistics, that are sufficient to compute the sizes of intermediate results of all plans of a query. We consider queries with select, bag/set project, Cartesian product, bag/set union, and bag/set difference operations.
in the forms of linear as well as bushy trees [4, 10]. The relations can be bags as well as sets of tuples. The sufficient statistics can be gathered either on-line at runtime or off-line in spare time. The gathered statistics can be used by the (existing) query optimizer to find the best plan in its search space, conveniently called the optimal plan here. We present two innovative methods to conduct query re-optimization, an eager and a lazy approach. The eager approach gathers all sufficient statistics at once and derives the optimal plans. The lazy re-optimization gathers only the statistics that are needed to correct large cardinality estimation errors found in the plan and generates a revised plan. We adapt the two basic re-optimization techniques to constantly changing database environments by continuously monitoring the executions and revising the plans, called adaptive re-optimization.

Our work distinguishes itself from others in the area of query re-optimization. We present a comprehensive solution to the re-optimization of repetitive queries. We cover a large class of queries that includes the select, project, Cartesian product, union and difference operations on bags and sets of tuples in the forms of linear and bushy trees. The identification of the sufficient statistics makes it entirely possible to compute the exact intermediate results of all plans of a query and hence the re-optimization. We propose several innovative ways to conduct re-optimizations, realizing the ultimate goal of query optimization for repetitive queries.

The rest of the paper is organized as follows. In Section 2, we briefly review papers related to the query re-optimization. In Section 3, we propose a re-optimization framework and give a functional overview of each component of it. In Section 4, we identify statistics that are sufficient to compute the intermediate results of all plans of a query. In Section 5, we present different ways to collect the sufficient statistics. In Section 6, we propose several innovative ways to conduct re-optimization. Section 7 is the conclusions and future work.

2. LITERATURE SURVEY

The work on query re-optimization can be classified into two categories: (1) re-optimizations of ongoing queries and (2) re-optimization of future queries. Our work falls into the second category.

Most re-optimization work [1, 5, 6] belongs to the first category. Kabra et al. [5] collected statistics during the execution of a query and used them to optimize the rest of the execution by either changing the execution plan or improving the resource allocations. Heuristics were used to determine if the benefits of re-optimization outweighed the overheads. Markl et al. [6] computed the validity range of each plan. When the actual result falls outside the validity range, a re-optimization is triggered. Intermediate results are saved for potential re-uses in the new join order. Instead of computing a point estimate of the cardinality of an operator, Babu et al. [2] used interval estimates to account for the uncertainty of the estimation. Within the bounding box, they selected robust and switchable plans that avoid re-optimization and loss of pipelined work done earlier.

[2, 12] fall in the second category. Stillger et al. [12] collected cardinalities of operators and used them to adjust selectivity estimation of future queries. It may work well if the queries bear a strong resemblance to the previous ones. Chaudhuri, et al. [2] used distinct page count, as opposed to the cardinality, as the cost measure for an operator. The page count accounts for the clustering effect of data on the disks, reflecting the computation cost or time in a more direct way.

3. RE-OPTIMIZATION FRAMEWORK

In Figure 3.1, we outline the re-optimization framework with solid lines and arrows highlighting the new components and paths added to the database system while the dotted lines the existing components and paths. Note that we do not intend to modify the existing query optimizers, but just to provide them with sufficient statistics to find the execution plans that are truly the best in their search spaces.

When a repetitive query is being executed, statistics can be collected, as indicated by the on-line statistics gathering box in the figure. One can also gather statistics off-line whenever it is convenient, especially in spare time, as indicated by the off-line statistical gathering box. The on-line approach takes advantage of the query evaluation to gather readily available statistics while the off-line approach gathers the sufficient statistics without interfering with the executions of queries. Section 5 has more details on these approaches.

The re-optimizer supplies the statistics gathered to the query optimizer to generate the optimal or a revised plan. Depending upon the amounts of statistics supplied to the optimizer, the re-optimization can be conducted either in an eager or a lazy fashion. The eager re-optimization refers to the situations where all sufficient statistics of a query are provided to the optimizer at once to generate the optimal plan. As for the lazy re-optimization, only selected statistics that are likely to correct large estimation errors found in the plan are gathered and provided to the optimizer to generate a revised plan. It may take a few cycles to arrive at the optimal plan. The optimal or revised plan is stored in the system for subsequent executions of the query.

The adaptive re-optimization is devised to maintain the optimality of plans throughout the lifetime of the queries. By continuously monitoring the executions and, if necessary, modifying the plans, optimality can be
maintained even though the underlying database has undergone substantial changes. The adaptive re-optimization can be accomplished by using either the eager or the lazy re-optimization method. Section 6 has detailed discussions on these re-optimization methods.

Sincethe re-optimization is mainly an off-line process, it can afford to use more time and resources to search for a better plan. Therefore, it may be worth employing a more sophisticated query optimizer that searches a larger solution space. Note that the proposed sufficient statistics are sufficient for computing the exact intermediate results' sizes of all plans of a query derived by using the commonly used algebraic laws and optimization heuristics.

4. SUFFICIENT STATISTICS

We assume relational algebraic laws, such as the commutative and associative laws for joins, Cartesian product, and unions, and useful heuristics, such as pushing of the selections and projections down the tree, are used to generate alternative plans. Due to space limitation, readers are referred to [13] for proofs of all lemmas and theorem.

4.1 Common Properties of Execution Trees

First, we identify some useful properties of the plans generated. Consider a university database with three relations: Assignment(course_id, tname, dept), Books (book_id, title, publisher), and CoursesText(course_id, book_id, title), with their keys underlined. We assume a course can use more than one textbook. Consider the query: Print the titles of the books published by "PH" and used in the courses taught by teachers in the CS department. Figure 3.1 show two alternative plans of the query in which selections and projections have been pushed down.

Example 1. In Figure 3.1 (a) and (b), operand relations A, B, and C each are preceded by the same select operation in both plans. That is, in both plans, A is preceded by \( \sigma_{dept=CS} \) (i.e. \( \sigma_{dept=CS}(A) \)), B by \( \sigma_{publisher=PH} \) (i.e. \( \sigma_{publisher=PH}(B) \)), and C by no selection.

LEMMA 1. Each operand relation is preceded by the same selection conditions, if any, in all plans of the query.

Definition. Select-modified relation. Let \( R \) be an operand relation of a query. After selections have been pushed as far down the tree as possible, the select-modified relation of \( R \), denoted by \( R' \), refers to \( R \) and its immediate preceding selection, if any, in the tree. If \( R \) has no selection preceding it, \( R \) itself is the select-modified relation of \( R \).

4.1 Sufficient Statistics

Let \( \text{attr}(mr) \) be the set of attributes of the modified relation "mr", and \( \text{basis}(mr) \) be the set of basis attributes for "mr". We assume the highest node representing the modified relation has been marked as a modified relation. For example, for the select-modified relation \( \sigma_{dept=CS}(A) \), we assume the node \( \sigma_{dept=CS} \) has been marked as a modified relation.

Algorithm Dist_Domain_Bases (node, bases)
{
    if (node is not a modified relation)
    {
        for (each childnode of node)
            Dist_Domain_Bases (childnode, bases);
    }
    switch (type of node)
    {
        Case root:
        for (each modified relation mr)
        {
            basis(mr)=bases \& attr(mr);
        }
    }

Figure 3.1 Execution Plans

In the figure, we have used, for simplicity, A for Assignment, B for Books, and C for CourseText. The two plans have different join orders.

Example 1. In Figure 3.1 (a) and (b), operand relations A, B, and C each are preceded by the same select operation in both plans. That is, in both plans, A is preceded by \( \sigma_{dept=CS} \), (i.e. \( \sigma_{dept=CS}(A) \)), B by \( \sigma_{publisher=PH} \) (i.e. \( \sigma_{publisher=PH}(B) \)), and C by no selection.

LEMMA 1. Each operand relation is preceded by the same selection conditions, if any, in all plans of the query.

Definition. Select-modified relation. Let \( R \) be an operand relation of a query. After selections have been pushed as far down the tree as possible, the select-modified relation of \( R \), denoted by \( R' \), refers to \( R \) and its immediate preceding selection, if any, in the tree. If \( R \) has no selection preceding it, \( R \) itself is the select-modified relation of \( R \).

4.1 Sufficient Statistics

Let \( \text{attr}(mr) \) be the set of attributes of the modified relation "mr", and \( \text{basis}(mr) \) be the set of basis attributes for "mr". We assume the highest node representing the modified relation has been marked as a modified relation. For example, for the select-modified relation \( \sigma_{dept=CS}(A) \), we assume the node \( \sigma_{dept=CS} \) has been marked as a modified relation.

Algorithm Dist_Domain_Bases (node, bases)
{
    if (node is not a modified relation)
    {
        for (each childnode of node)
            Dist_Domain_Bases (childnode, bases);
    }
    switch (type of node)
    {
        Case root:
        for (each modified relation mr)
        {
            basis(mr)=bases \& attr(mr);
        }
    }

Figure 3.1 Execution Plans

In the figure, we have used, for simplicity, A for Assignment, B for Books, and C for CourseText. The two plans have different join orders.

Example 1. In Figure 3.1 (a) and (b), operand relations A, B, and C each are preceded by the same select operation in both plans. That is, in both plans, A is preceded by \( \sigma_{dept=CS} \), (i.e. \( \sigma_{dept=CS}(A) \)), B by \( \sigma_{publisher=PH} \) (i.e. \( \sigma_{publisher=PH}(B) \)), and C by no selection.

LEMMA 1. Each operand relation is preceded by the same selection conditions, if any, in all plans of the query.

Definition. Select-modified relation. Let \( R \) be an operand relation of a query. After selections have been pushed as far down the tree as possible, the select-modified relation of \( R \), denoted by \( R' \), refers to \( R \) and its immediate preceding selection, if any, in the tree. If \( R \) has no selection preceding it, \( R \) itself is the select-modified relation of \( R \).

4.1 Sufficient Statistics

Let \( \text{attr}(mr) \) be the set of attributes of the modified relation "mr", and \( \text{basis}(mr) \) be the set of basis attributes for "mr". We assume the highest node representing the modified relation has been marked as a modified relation. For example, for the select-modified relation \( \sigma_{dept=CS}(A) \), we assume the node \( \sigma_{dept=CS} \) has been marked as a modified relation.

Algorithm Dist_Domain_Bases (node, bases)
{
    if (node is not a modified relation)
    {
        for (each childnode of node)
            Dist_Domain_Bases (childnode, bases);
    }
    switch (type of node)
    {
        Case root:
        for (each modified relation mr)
        {
            basis(mr)=bases \& attr(mr);
        }
    }
if (basis(mr) = ∅) basis(mr) = {count};
    
    }  
    Case select:
    bases = basesU {attributes in the
    selection condition};
    
    Case set projection:
    bases = basesU {attributes in the
    projection list};
    
    Case difference:
    bases = basesU {keyattributes in the
    operandrelations};
    
    Case set union:
    bases = basesU {keyattributes in the
    operandrelations};
    
}  

Figure 4.2 Attributes of Frequency Distributions

Example 2. Consider the plans in Figure 3.1(a) and (b) again and let the projections in the trees now be the set projections. For modified relation A', basis(A') = {course_id} because course_id is a join attribute and book_id is both a
join and a set project attribute while title appears in a set projection. Notice that title would not have been included in basis(B') if the projections were the bag projections. For C', basis(C') = {course_id, book_id} because course_id is a join attribute and book_id is both a
join and a set project attribute. □

Again, in Example 2, one can use either the plan in Figure 3.1(a) or (b) to find the bases of the frequency
distributions’ domains and the results are the same. The following Lemma gives a formal proof of such
phenomenon under the assumption that selections and projections are pushed as far down the trees are possible
in all plans.

Lemma 2. The algorithm Dist_Domain_Bases() derives the same bases for all plans of a query.

Example 3. Consider the plan in Figure 3.1(a) with all
projections being the set projections. As discussed in
Example 2, basis(A') = {course_id}, basis(B') = {course_id, title}, and basis(C') = {course_id, book_id}. Let f_{A'}(course_id), f_{B'}(book_id, title), and
f_{C'}(course_id, book_id) be the frequency distributions constructed for modified relations A', B',
and C', respectively.

Let A'' = \pi_{\text{course}_id}(A'). The frequency distribution
of A'' (= \pi_{\text{course}_id}(A')) , denoted by f_{A'}(course_id), can be computed by first coalescing the frequency
values of f_{A'}(course_id) on attribute course_id. Since both f_{A'} and f_{A''} are functions on course_id, the
coalescing really has no effect. As for the duplication
elimination function of the set projection, we just need to set any frequency values that are greater than 1 to 1 to reflect the effect.

Let B'' = \pi_{\text{book}_id, title}(B'). The frequency distribution
of B'' (= \pi_{\text{book}_id, title}(B')) , denoted as f_{B''}(\text{book}_id, title), can be computed in a similar way. The
coalescing of frequencies on attributes book_id and title has no effect on the frequencies because f_{B'} and
f_{B''} are all defined on the same set of attributes {\text{book}_id, title}. To reflect the duplicate elimination of
the set projection, any frequency values greater than 1 need to be set to 1. Then, we can use
f_{C'}(\text{course}_id, \text{book}_id) and f_{\text{A'}}(\text{course}_id) to derive the resulting
frequency distribution of C' \bowtie_{\text{course}_id} A'', denoted by
f_{C''}(\text{course}_id, \text{book}_id), as

f_{C''}(c, b) = f_{C'}(c, b) \times f_{A'}(c).  \tag{1}

for given c (\text{course}_id) and b (\text{book}_id) values. To compute the distribution of
\pi_{\text{book}_id}(C' \bowtie_{\text{course}_id} A''), with duplicates deleted, denoted by
f_{\pi(C' \bowtie_{\text{course}_id} A'')}(\text{book}_id), we first compute

f_{\pi(C' \bowtie_{\text{course}_id} A'')(b)} = \sum_{c \in \text{Dom}(\text{course}_id)} f_{C''}(c, b).  \tag{2}

for given course_id c and book_id b values to coalesce the frequencies on book_id and then set any frequency
values that are greater than 1 to 1 to reflect the effect of the
duplicate elimination of the set projection. Finally,
f_{\pi(C' \bowtie_{\text{course}_id} A'')(\text{book}_id)} can be used with
f_{B''}(\text{book}_id, title) to derive the frequencies of the final join result as

f_{\pi(C' \bowtie_{\text{course}_id} A'')(\text{book}_id, title)} = f_{\pi(C' \bowtie_{\text{course}_id} A'')(b)} \times f_{B''}(b, t).  \tag{3}

for given title t and book_id b. The attribute title will be
used to coalesce the frequencies and any frequency
values greater than 1 are set to 1 to reflect the last set
projection on the attribute title. □

To derive the resultant frequency distributions of a
bag union, one simply adds the corresponding input
frequencies; for a Cartesian-product, one multiplies the
frequencies; for a Cartesian-product, one multiplies the
frequencies; for a Cartesian-product, one multiplies the
frequencies; for a Cartesian-product, one multiplies the
frequencies; for a Cartesian-product, one multiplies the
frequencies; for a Cartesian-product, one multiplies the
frequencies; for a Cartesian-product, one multiplies the
frequencies; for a Cartesian-product, one multiplies the
frequencies.

Theorem. The proposed sufficient statistics for a
query are sufficient to compute the frequency
distributions of the intermediate and final results of all
plans of the query.

5. Statistical Gathering

Generally speaking, one can either gather the
statistics while the query is being executed (i.e., an
online approach), or whenever it is not interfering with the
execution of the query (i.e., an off-line approach).
5.1 On-line Statistical Gathering
As tuples flow through the select operator of a modified relation, we construct the frequency distribution of the modified. Figure 5.1 shows the points where the sufficient statistics are gathered.

5.2 Off-line Statistics Gathering
Instead of gathering statistics at runtime, one can collect the statistics off-line insystem’s spare time or whenever the DBA feels appropriate. This approach does not interfere with the executions of the queries and thus has no runtime overheads. However, re-evaluations of the modified relations would be necessary.

6. RE-OPTIMIZATION
In this section, we discuss different ways to attain the optimal plans. In general, an optimal plan can be obtained in one step or multiple steps, which are called an eager and a lazy re-optimization, respectively. We will also discuss other more sophisticated methods that are built on the tops of the two fundamental methods. It is noted thata re-optimization is mainly an off-line process. It can be conducted whenever appropriate, for example, at system’s spare time.

6.1 Eager Re-optimization
The eager re-optimization is probably the most straightforward way to conduct re-optimization. For a given query, we first collect all the sufficient statistics and then provide them to the query optimizer to search for the optimal plan. The sufficient statistics can be obtained by any of the statistics gathering methods mentioned in Section 5 and discarded after use. The optimal plan is stored in the system for subsequent executions of the query.

6.2 Lazy Re-optimization
The plans generated using the conventional statistics stored in the database catalog may sometimes generate plans of good quality. In the lazy re-optimization, we attempt to use the conventional statistics in the catalog as much as possible unless large estimation errors (on the intermediate results’ cardinalities (sizes)) have been found in the plan (to be discussed shortly); and only then is the statistics gathering process invoked. Note that we only gather the statistics that are likely to correct the large estimation errors found in the plan, not the entire set of the sufficient statistics. This lazy approach tries only to replace the statistics in the catalog that are not accurate enough for uses; it can also spread the cost of statistics gathering over a longer period of time.

We intend to monitor the cardinalities (not the frequency distributions) of intermediate results during the execution to see if any large estimation errors have occurred. In Figure 6.1, a redraw of Figure 4.1(a), the arrows indicate the checkpoints where cardinalities are to be monitored.

The gathering of cardinalities, done by counting the numbers of tuples flowing through the checkpoints, is simple and should incur no noticeable overhead. By comparing the actual cardinalities against the estimated ones, large estimation errors can be identified. Here, the estimated cardinalities refer to the cardinalities calculated by the optimizer when selecting the plans.

A threshold (e.g., 5%, to be discussed in Section 6.2.3) is set up to determine if any of the estimation errors is large enough to warrant the gathering of accurate statistics. We shall, in the next subsection, discuss how to identify modified relations for which accurate statistics should be gathered to correct the large estimation errors. The desired statistics can be gathered by the off-line approach discussed in Section 5.

The optimizer uses the newly gathered accurate statistics, supplemented with the statistics in the catalog, to derive a revised plan. If there are still large estimation errors found in the revised plan in subsequent executions, more statistics are to be gathered. In order to reuse previously gathered statistics, the statistics need to be stored in the system with the query. It can be observed that in the worst case, all the sufficient statistics will eventually be collected to attain the optimal plan.

6.2.1 Placement of Checkpoints
Checkpoints can be placed essentially at every place in the plan. While there may be many good schemes to place checkpoints, here, we discuss a simple one that places checkpoints only at the inputs of binary operators of the plan, as shown in Figure 6.1. One reason is that the inputs of an operator determine the output of it. Moreover, the inputs of a binary operation often have been modified by a series of unary operations like selections and projections, such as the input of the join on course_id π course_id (σdept = CS(A)) in Figure 6.1. It is
very difficult for an optimizer to get good estimates of the inputs using the statistics stored in the catalog. Consequently, the inputs are good places to catch potential estimation errors.

6.2.2 Gathering Selected Statistics

When large estimation errors are found, we need to identify the modified relations for which accurate statistics can likely correct the estimation errors.

The inaccuracies in the estimations of cardinalities and data distributions of the input relations are the two main factors contributing to the errors. The checkpoints placed between the binary operators and the modified relations, e.g., checkpoints at 1, 2, and 3 in Figure 6.1, serve just the purpose of detecting cardinality estimation errors on the inputs of the operators. Thus, if large estimation errors have occurred at such places, the accurate statistics for the respective modified relations should be gathered.

Inaccuracies in the input data distributions estimation are another factor contributing to the estimation errors. For example, dependent upon how the join attribute values of the input tuples match, the results of a join can be quite different even though the inputs are of fixed sizes. To detect estimation errors caused by the lack of information or inappropriate assumptions on the data distributions, one has to rely on the checkpoints placed above the binary operators, such as the checkpoint 4 in Figure 6.1. Unfortunately, such checkpoints cannot pinpoint exactly the modified relation(s) for which accurate statistics can correct the estimation errors found in the plan because inaccuracies in many of modified relations below the operators can contribute to the errors. Therefore, we can only select modified relations for which accurate statistics can most likely correct the errors.

We summarize the above discussions to provide the following heuristics to determine the modified relations for which accurate statistics should be collected.

**Heuristics:**

1) if a large estimation error has occurred between a binary operator and a modified relation (e.g., 1, 2 and 3 in Figure 6.1), gather statistics for that modified relation.

2) if a large estimation error has occurred above a binary operator (e.g., 4 in Figure 6.1) and no relation under it in the tree for which statistics has been gathered (in the current run), gather statistics for one or more modified relations at which larger errors have occurred.

In rule (2), one can be conservative to select only one modified relation, or be aggressive to select more than one modified relation for which statistics are collected. Other heuristics are certainly possible and are left for future research.

**Example 8.** In Figure 6.1, if there is a large error (greater than the threshold value) found at 1 (2 or 3), then we gather statistics for the modified relation of C (A or B). If a large error is found at checkpoint 4 and no modified relation under it has been selected, then we can select either the modified relation of C or A, dependent upon where a larger error has occurred, at 1 or 2. Certainly, we can be more aggressive to select both. □

6.2.3 Threshold

If large estimation errors are found in the plan, it could mean that the previous plan selection was flawed and a sub-optimal plan might have been selected. We wish to use the estimation errors found in the plan as an indicator for potential sub-optimality in the plan.

Estimation errors, dependent upon where they occur, could have different impacts on the cost of the plans. There is probably no single or even a set of best threshold values for all possible operations, queries, and data distributions. In order not to complicate the discussion, here for simplicity, we assume a single threshold value for each query. We shall investigate the potential of using multiple thresholds in the future research.

It is important to find a good threshold value. If the threshold is set too high, sub-optimality in the plan can easily elude the checkpoints. On the other hand, if the threshold is set too low, minor errors that would not cause any change to the plan can trigger the gathering of statistics (i.e., a false alarm).

We propose to use a dynamically adjustable threshold value, described as follows. First, the threshold is assigned a low initial value e.g., 5%. When the estimation errors exceed the threshold, we gather desired statistics (according to the heuristics in Section 6.2.2) and generate a "new" plan. If the "new" plan is a sub-optimal plan, we decrease the threshold value (because a lower threshold could also lead to the same change); otherwise, increase the threshold (as it might have been set too low and caused the false alarm).

There are many possible ways to adjust the threshold value. One simple way is to increase or decrease it by a fixed amount (e.g., 5%). One can also change the threshold value by an amount proportional to the amount of errors, etc. We shall leave these options for future research.

In the following, we summarize the essence of the lazy re-optimization into the following algorithm. The variable "plan" stores the plan of the query and the "statistics" stores whatever the sufficient statistics that have been gathered. The "cards" stores the actual cardinalities recorded at the checkpoints, while the "est_cards" stores the estimated cardinalities derived by
the optimizer using available statistics. “T” is the threshold and the “errors” stores the estimation errors at the checkpoints. “changed” is a Boolean flag indicating whether a different new plan has been generated by the optimizer or not. “re_optimize()” represents the process of generating a plan using the statistics available in the catalog and the accurate statistics gathered and stored in “statistics”.

Algorithm Lazy-Re-Opt (plan, statistics, cards, est_cards, T)
{
   errors = compare(cards, est_cards); /* estim.errors*/
   if(max(errors) > T)
      get_desired_stat(statistics, errors); /* Sec. 6.2.2
      changed = re_optimize(plan, statistics);
   if(changed==true) /* Sec. 6.2.3
      increase(T);
   else
      decrease(T);
}

Figure 6.2 Lazy Re-optimization

6.3 Adaptive Re-optimization

An optimal plan can degenerate to a sub-optimal one once the database has undergone substantial changes. To guarantee the optimality of the plan for the entire lifetime of the query, an adaptive scheme is devised. The adaptive re-optimization can be achieved by constantly monitoring the executions and performing necessary re-optimizations. There are two simple ways to implement the adaptive re-optimization. The first possibility is to couple the cardinality monitoring process of the laze re-optimization with the eager re-optimization, that is, to gather all the sufficient statistics whenever estimation errors are found to be greater than the threshold. Another possibility is to simply extend the cardinality monitoring period of the lazy re-optimization to the entire lifetime of the query, and the lazy re-optimization immediately becomes an adaptive re-optimization scheme. By detecting and remediating the sub-optimality timely, the adaptive re-optimization may enable queries to run in the most efficient ways for the entire lifetime of the queries.

7. CONCLUSIONS

In this paper, we propose a comprehensive re-optimization framework for an important and large class of queries—repetitive queries. We first discussed statistics that are sufficient to find the best plan for a query. The proposed sufficient statistics make re-optimization of queries a realistic and achievable goal. Then, we discussed different ways to gather the sufficient statistics and presented two innovative methods to conduct re-optimization—the eager and the lazy re-optimization. The eager re-optimization attains the optimal plans in one step, while the lazy re-optimization in multiple steps. We have also designed an adaptive re-optimization method to adjust the plans dynamically so that the queries can always be executed in the optimal fashions for their entire lifetime. The approximate re-optimization presents an efficient and effective alternative to refining query plans.

In the future, we shall extend the coverage of queries to those with other useful operators and aggregate functions. Although we have proved that with the proposed sufficient statistics on hand, one can always derive the optimal plans (i.e., an eager re-optimization), we still need to verify by experiment how effective the proposed heuristics in the lazy approach are and examine the quality of the plans generated by the proposed approximate re-optimization method.

8. REFERENCES