Increasing the Accuracy and Coverage of SQL Progress Indicators

Gang Luo[1] Jeffrey F. Naughton[2] Curt J. Ellmann[2] Michael W. Watzke[3]
IBM T.J. Watson Research Center[1] *University of Wisconsin-Madison[2]* *NCR[3]*
luog@us.ibm.com *naughton@cs.wisc.edu* *ellmann@wisc.edu* *michael.watzke@ncr.com*

Abstract

Recently, progress indicators have been proposed for long-running SQL queries in RDBMSs. Although the proposed techniques work well for a subset of SQL queries, they are preliminary in the sense that (1) they cannot provide non-trivial estimates for some SQL queries, and (2) the provided estimates can be rather imprecise in certain cases. In this paper, we consider the problem of supporting non-trivial progress indicators for a wider class of SQL queries with more precise estimates. We present a set of techniques in achieving this goal. We report an initial implementation of these techniques in PostgreSQL.

1. Introduction

Recently, [4, 9] proposed supporting progress indicators for long-running SQL queries in RDBMSs. The goal of these progress indicators is to act as a user-interface tool so that the user can keep track of the percentage of the SQL query that has been completed and the remaining query execution time.

[4, 9] proposed a set of techniques to implement progress indicators for SQL queries. They also demonstrated that their techniques work much better than naive alternatives for a subset of SQL queries. However, the techniques in [4, 9] are preliminary in the sense that (1) they do not provide non-trivial estimates for some SQL queries, and (2) the provided estimates can be rather imprecise in certain cases.

In this paper we propose new techniques that fall into two categories. Techniques in the first category improve the accuracy of the estimates. Our contribution in this category includes the observation that progress indicators can profit from defining segments at a finer granularity than that used in [4, 9], and the observation that even the simple approach of using the optimizer's estimate of whether a segment is CPU or I/O bound can substantially increase the accuracy of a progress indicator.

Techniques in the second category provide new functionality that is not covered in [4, 9]. Specifically, we present new techniques that allow progress indicators to do a reasonable job of estimating progress in the presence of sort operators, set operators, and correlated sub-queries (all the correlated sub-queries tested in [4] were removed by SQL Server's rewriting before execution [13].)

The rest of the paper is organized as follows. In Section 2, we give a brief review of previously proposed techniques for supporting progress indicators for SQL queries. In Section 3, we present techniques that improve the accuracy of the estimates provided by progress indicators. Section 4 covers the techniques that enable non-trivial progress indicators for a wider class of SQL queries. In Section 5, we present results from an initial implementation of our techniques in PostgreSQL. We conclude in Section 6.

2. Review of Previous Work

In this section, we briefly review the techniques proposed in [4, 9] for supporting progress indicators for SQL queries. We first give an overview of the techniques from [9] in Section 2.1. Then in Section 2.2, we describe one step of the procedure in some detail, as this step is referred to in Sections 3 and 4. In Section 2.3, we give a comparison between [9] and [4].

2.1. Overview of Techniques in [9]

The progress indicator described in [9] divides a query plan into one or more segments. Each segment is defined as one or more consecutive operators that can be executed as a pipeline. Each segment can be viewed as a tree of operators. The query plan can be viewed as a tree of segments.

The cost of a segment is the total number of bytes in its input and output. If an operator at the leaves or root of a segment is a multi-stage operator (e.g., a multi-pass sort), then bytes handled by this operator will be counted once each time they are read or written. The query cost is the sum of the costs of all the segments in the query plan. The query cost is measured in *U*'s, where each *U* represents one page of bytes.

Initially, the progress indicator in [9] uses the query optimizer's estimates to estimate the query cost. As a query runs, the progress indicator obtains more precise information about the inputs and outputs of the segments so it can continuously refine the estimated query cost (this is similar to the techniques used in dynamic query optimization [1, 3, 5, 6, 7, 10, 11].) This is accomplished by collecting statistics at the output of each segment and

propagating the improved estimates upwards in the query plan.

At all times, the progress indicator monitors the speed at which bytes are being processed by the query. It then uses this information to continuously refine the estimated remaining query execution time.

2.2. Review of Refining Cardinality Estimates

We turn now to describe how the progress indicator in [9] continuously refines the estimated output cardinality of the current segment that is being processed. For each segment, it defines one or two dominant inputs that are used to approximately indicate the progress of the segment. For example,

(1) If a segment contains only one input, this input is the dominant input.
(2) If a segment contains a single hash join operator, the dominant input is the probe relation of this operator.
(3) If a segment contains a single sort-merge join operator, the dominant inputs are the two input relations of the sort-merge join operator.

The progress indicator uses the percentage of the dominant input that has been processed to refine the estimated output cardinality of the current segment. We first discuss the case that the current segment contains one dominant input. Then we discuss the case that the current segment contains two dominant inputs.

At the time that the current segment starts execution, the progress indicator gives an initial estimate E_1 of its output cardinality. E_1 is computed using the input cardinalities of the current segment and the optimizer's cost estimation module. Suppose that the dominant input cardinality of the current segment is z. Assume that so far, the query processor has processed x of z and generated y output tuples. Then the percentage that the dominant input has been processed is $p=x/z$. If we assume that at any time, the number of output tuples that have been generated is proportional to the percentage that the dominant input has been processed, then we can estimate the final output cardinality of the current segment to be $E_2=y/p$. In practice, this assumption may not be valid and so the progress indicator also considers the initial estimate E_1.

At any time, the progress indicator in [9] uses the following heuristic formula to estimate the final output cardinality E of the current segment: $E=p\times E_2+(1-p)\times E_1$. This heuristic formula intends to smooth fluctuations in the estimator and to let it gradually change from the initial estimate (when the current segment just starts execution, and we know nothing about the actual segment output cardinality) to the actual segment output cardinality (when the current segment finishes execution, we know this quantity exactly.)

Recall that a segment containing a sort-merge join operator has two dominant inputs. In this case, once the query processor reaches the end of either dominant input, the sort-merge join (and thus the segment) immediately finishes execution. Therefore, the progress indicator needs to use the dominant input that is being scanned relatively faster to decide the percentage p of the two dominant inputs that has been processed [16].

2.3. Comparison between [9] and [4]

In general, [4] and [9] use similar techniques. For example:

(1) The "pipeline" in [4] is equivalent to the "segment" in [9].
(2) The "driver node" in [4] is equivalent to the "dominant input" in [9].
(3) The technique in [4] of counting tuples (or the number of getnext() calls) is similar to the technique in [9] of counting tuple bytes.
(4) The technique in [4] of handling spills is similar to the technique in [9] of counting the same byte multiple times, once for each time the byte is logically read or written.
(5) [4] assumes that the actual work done per tuple is the same across all operators in the query plan. [9] assumes that all future segments process tuples at the same speed.

As a result, most of the techniques we propose in this paper as extensions to [9] have analogues that could be used with the approach proposed in [4].

The main differences between [4] and [9] are:

(1) In refining the estimated cardinalities, [4] uses a method based on refining upper bounds and lower bounds, while [9] uses a method based on linear interpolation.
(2) In estimating the completed percentage, [4] considers all operators in the query plan and uses the driver node hypothesis, while [9] only considers the segment inputs and outputs.
(3) [4] does not try to predict the remaining query execution time.

It would be an interesting area of future work to investigate how these three differences impact the utility of the techniques proposed in this paper for the progress indicator proposed in [4].

3. Improving the Accuracy of Predictions

In this section, we describe two new techniques that improve the accuracy of the estimates.

3.1. Refined Definition of Segments

[9] defines a segment as one or more consecutive operators that can be executed as a pipeline. According to this definition, one segment can contain multiple join operators. In this case, this definition is too coarse and makes it difficult for the progress indicator to provide precise estimates. In the following, we use two examples to illustrate the point.

Example 1. Consider the query plan shown in Figure 1. This query plan computes a three-table join of relations *A*, *B*, and *C*, where the join condition is *A.a=B.b=C.c* and each relation has been pre-sorted on the join attribute. This query plan contains only one segment with two sort-merge join operators. (We adopt the convention in [9] of using ovals to represent segments.)

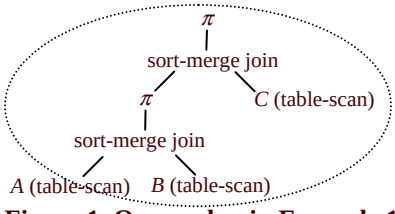

Figure 1. Query plan in Example 1.

For a segment containing multiple join operators, [9] defines the dominant input(s) according to the lowest-level join operator. For example, consider the segment of the query plan in Figure 1. [9] defines the dominant inputs to be *A* and *B*. Suppose at some point, the percentages of the three relations that have been processed are: $p_A=2\%$ for *A*, $p_B=5\%$ for *B*, and $p_C=90\%$ for *C*. Then as reviewed in Section 2, in estimating the output cardinality (and also the cost) of the segment, [9] assumes that $max(p_A, p_B)=5\%$ of the segment has been processed. However, in this case, it is more reasonable to assume that $max(p_A, p_B, p_C)=90\%$ of the segment has been processed. This is because once we reach the end of either *A*, *B*, or *C*, whichever is first, the segment immediately finishes execution.

Example 2. Consider the query plan shown in Figure 2. This query plan contains only one segment with two nested loops join operators. In estimating the cost of the segment (and thus the query cost), we need to estimate the number of times that relation *C* will be index-scanned. That is, we need to estimate the input cardinality of the index nested loops join operator.

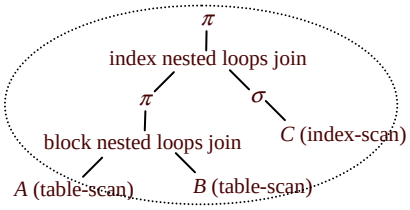

Figure 2. Query plan in Example 2.

During query execution, [9] only collects statistics at the output of each segment. Hence, no statistics are collected about the input cardinality of the index nested loops join operator. This prevents us from continuously refining the estimated segment cost (and thus the query cost.)

From the above two examples, we can see that in order to improve the accuracy of the estimates provided by the progress indicator, we need to define segments at a finer granularity so that at most one join operator exists in each segment.

Therefore, we refine the definition of segments as follows. A segment contains one or more consecutive operators that can be executed as a pipeline, while at most one operator among these operators is a join operator. For a pipeline that connects two join operators, the boundary of the two segments, each containing one of these two join operators, is defined at the input of the upper-level join operator.

According to this refined definition of segments, the query plan in Example 1 now contains two segments, as shown in Figure 3. The case with the query plan in Example 2 is similar.

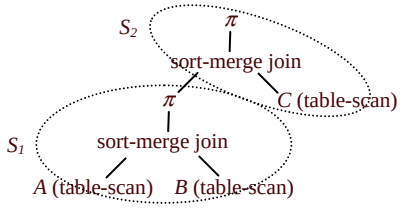

Figure 3. Query plan in Example 1 with redefined segments.

Using this refined definition of segments, when we compute the query cost, if the output of a segment *S* is pipelined as the input to the next segment, then the bytes produced by segment *S* are neither counted as they are output by segment *S* nor counted as they are input by the next segment.

[9] propagates the improved estimates upwards in the query plan to continuously refine the estimated query cost. However, if we use the refined definition of segments, then for multiple consecutive segments that can be executed as a pipeline, we may need to propagate the improved estimates downwards in the query plan to continuously refine the estimated query cost. For example, in Example 1, the relative speed at which relation *C* is scanned will influence the cost of scanning *A* and *B*. This is because once we reach the end of *C*, we can stop scanning *A* and *B* immediately. The general discussion is straightforward and thus omitted here.

3.2. Predicting Work Unit Processing Speed

In [9], both the estimated query cost and the current query execution speed are measured in *U*'s. Each *U* represents one unit of work that is equal to that required to process one page of bytes. The current query execution speed is measured as the rate at which *U*'s are being processed.

[9] assumes that all future segments process *U*'s at the same speed. This assumption can be misleading if segments in the query plan have radically different performance characteristics. For example, consider a two-segment plan, in which segment S_1 feeds segment S_2. If S_1 processes *U*'s more slowly than S_2 (perhaps S_1 is I/O-intensive whereas S_2 has a high buffer pool hit rate), then while S_1 runs the progress indicator will overestimate the time it will take to run S_2.

To address this problem, we explicitly consider the fact that different future segments may process *U*'s at different speeds and try to predict such speeds. In general, for each *U* of a segment *S*, the speed at which *U* will be processed can be represented as a function $f(s_p, s_s)$, where s_p is the property of segment *S* (e.g., how expensive are the operators in segment *S*), and s_s is the system state at the time *U* will be processed (e.g., the load on the system, the buffer pool contents, etc.). The better we can predict this function, the more precise the remaining query execution time estimated by the progress indicator.

Of course, in general, providing an accurate and detailed implementation of $f(s_p, s_s)$ is a daunting task that may not be desirable or even feasible. Our goal in this paper is not to define this function; instead, we propose a simple approximation that is intended to do better than the "uniform processing rate" assumption used in [9].

In a traditional optimizer [15], the cost of an operator is defined as *I/O_cost+CPU_cost*. There, $CPU\_cost=W\times N$, where *W* is a weighting factor and *N* is the number of tuples processed. The weighting factor *W* converts I/O and CPU costs to a common "currency", which in the PostgreSQL optimizer is "number of page reads/writes." Depending on whether or not *I/O_cost>CPU_cost*, we define the operator as either *I/O-intensive operator* or *CPU-intensive operator*.

[9] defines the cost of a segment *S* as the total number of bytes input into/output by *S*. For each input *I* of the segment *S*, we define the number of bytes input into *S* as the *cost of the segment input I*. We define the *cost of the segment output* similarly. Then the cost of the segment *S* is the sum of the costs of the segment inputs and the cost of the segment output.

For each segment input *I*, consider the operator *Op* that is the parent of the input *I* in the segment. Our key heuristic is that according to the above definition, the cost of the segment input *I* is proportional to either the *I/O_cost* or the *CPU_cost* of *Op*, depending on whether *Op* is I/O-intensive or CPU-intensive.

Based on this heuristic, we redefine the cost of the segment input as follows:

(1) If the operator *Op* is a CPU-intensive operator, we use the *CPU_cost* (in *U*) of *Op* as the cost of the segment input *I*.
(2) If the operator *Op* is an I/O-intensive operator, we use the *I/O_cost* (in *U*) of *Op* as the cost of the segment input *I*.

Then we use the same method in [9] to measure the current query execution speed and estimate the remaining query execution time.

There is an interesting question: at run time, should the progress indicator revisit the optimizer's estimate of whether a segment is I/O or CPU intensive? It is certainly possible that this could improve estimates (perhaps the optimizer was anticipating an I/O-intensive scan but the relation was in the buffer pool when the scan actually occurred), although we did not pursue this in our current implementation.

4. Improving the Coverage of the Progress Indicator

In this section, we describe the three techniques that provide new functionality and enable the progress indicator to accurately handle wider classes of SQL queries. One reasonable question at this point is whether or not the job is "finished"; that is, are there other aspects of the SQL language that are not covered and still require future work?

This turns out to be difficult to answer. The issue is that progress indicators cannot really be said to "work" for some classes of queries and "not work" for others – it is more precise to say that they accurately predict the progress for some classes of queries and are less successful for others.

Perhaps this can best be explained by a categorization of the kinds of segments a progress indicator might hope to handle well. One possible categorization is into the following five patterns:

(1) **Pattern 1**: The segment contains only one input and the per-tuple cost of the operators in the segment is small and predictable.
(2) **Pattern 2**: The segment contains two inputs, one of which is the dominant input.
(3) **Pattern 3**: The segment contains two inputs, both of which are dominant inputs.
(4) **Pattern 4**: The segment contains a multi-stage operator. The cost of the segment depends on the number of stages required for this operator.
(5) **Pattern 5**: The segment contains an expensive operator that needs to be evaluated once for each

input tuple. The cost of the operator may vary from tuple to tuple and is hard to predict.

Note that this categorization is not exhaustive and a segment could in fact belong to several patterns.

[9] has already proposed techniques that work well for Pattern 1 (e.g., a segment with a selection operator), Pattern 2 (e.g., a segment with a hash join operator), and Pattern 3 (e.g., a segment with a sort-merge join operator.) The techniques in that paper are less successful at handling segments from Patterns 4 and 5. These previously proposed techniques could be applied – for example, one could treat a multi-stage operator as a monolithic single stage operator, and return some (possibly inaccurate or infrequently revised) estimate to the user. Or one could ignore the subtle issues in a Pattern 5 segment by assuming some average cost per tuple and never revising this estimate, thus treating a Pattern 5 segment as if it were a simple Pattern 1 segment.

Our point in this paper is that Patterns 1 through 3 are not sufficient, and that by explicitly considering Patterns 4 and 5 we can substantially improve the accuracy and responsiveness of a progress indicator. We demonstrate this by considering specific examples of operators in Pattern 4 and 5 segments. Whether or not further refinements of this categorization are useful is an interesting area for future work.

4.1. Refining the Estimated Cost of a Sort Operation

A progress indicator needs to continuously refine the estimated query cost. This is achieved by continuously refining the estimated costs of the segments in the query plan. However, [9] does not show how to continuously refine the estimated cost of a segment during the execution of a multi-pass sort operator (Pattern 4.) In this section, we present a solution to this problem. For ease of description, we assume that:

(1) The multi-pass sort operator is the only operator in the segment.
(2) The multi-pass sort operation does not reduce the number of tuples.

The extension to the general case is straightforward.

To compute the cost of a multi-pass sort operation, we need to know:

(1) The number of sorted runs generated during the first pass.
(2) The sizes of the sorted runs generated during the first pass.

There are two possible cases:

Case 1. The initial sorting algorithm generates sorted runs as large as the allotted buffer space. Then the number and the sizes of the sorted runs can easily be computed once the input size and buffer space are known.

Case 2. The initial sorting algorithm generates sorted runs of varying length depending upon properties of the input (this is the case, for example, with replacement sort [14, page 428; 8, Section 5.4].) In this case, we need to continuously refine both the estimated number and the estimated sizes of the sorted runs that will be generated at the end of the first pass.

We focus our attention on the first pass of the multi-pass sort operation. Our solution is as follows:

(1) We conceptually think of each sorted run as an output tuple of the segment. Then we can use the same method in [9] that is used to estimate the output cardinality of the current segment to estimate the number of sorted runs that will be generated at the end of the first pass of the sort operation. (That is, we use a linear combination of the optimizer's estimate of the number of sorted runs, and the observed number of sorted runs generated by the percentage of input processed at the current point in time.)
(2) For sorted runs that have already been generated, we know their exact sizes. Let T denote the total size of the input that has not been processed. Let E_s denote the estimated number of sorted runs that have not been generated. We estimate each sorted run that has not been generated to be of the same size T/E_s.

4.2. Refining the Estimated Output Cardinality of a Segment that Contains a Set Operator

In this section, we discuss how to continuously refine the estimated output cardinality of a segment that contains a set operator. How to continuously refine the estimated cost of a segment that contains a set operator is similar and thus omitted here. SQL supports three set operations: union, intersection, and set-difference. In a typical query plan, intersection is implemented with a join operator [14]. There are many alternatives for evaluating union and set-difference, and due to space constraints we focus only on an illustrative pair of ways to estimate the progress of the set-difference operator.

Suppose we want to compute *B-A*. Two commonly used methods include the hashing-based method and the sorting-based method [14, page 469]. We first discuss the hashing-based method in Section 4.2.1. Then we discuss the sorting-base method in Section 4.2.2.

Before we start the discussion, we further refine the definition of segments as follows. A segment contains one or more consecutive operators that can be executed as a pipeline, while at most one operator among these operators is a join operator or a set operator. The reason for this refinement is that, as discussed below, a set operator behaves much like a join operator. (Recall that as discussed in Section 3.1, each segment contains at most one join operator.)

4.2.1. Hashing-based Method. The hashing-based method works as follows. We build a hash table H for A. Then we scan B. For each tuple t_B of B, we probe the hash table H. If $t_B \notin A$, we write t_B to the result. Therefore, a set-difference operator that is implemented with the hashing-based method behaves much like a hash join operator (Pattern 2.) We can use the same method in [9] that is used to handle a segment that contains a hash join operator to handle a segment that contains a set-difference operator implemented with the hashing-based method (e.g., the dominant input of the segment is the probe relation.)

4.2.2. Sorting-based Method. The sorting-based method works as follows. We first sort A, then sort B. Then we merge the sorted A and B. During the merging pass, we only write tuples of B to the result, after checking that they are not in A. Therefore, a set-difference operator that is implemented with the sorting-based method behaves much like a sort-merge join operator (Pattern 3.) We can use the same method in [9] that is used to handle a segment that contains a sort-merge join operator to handle a segment that contains a set-difference operator implemented with the sorting-based method (e.g., the segment contains two dominant inputs that are the two inputs of the set-difference operator.) The only exception is that we need to make minor changes to the formula that is used to estimate the output cardinality of the segment.

We use an example to illustrate the point. Consider two relations A and B that have already been sorted. When we compute a sort-merge join between A and B, once we reach the end of either A or B, the sort-merge join immediately finishes execution. However, when we compute B-A, there are two possible cases:

(1) If we reach the end of B first, the set-difference operation immediately finishes execution.
(2) If we reach the end of A first, we still need to continue to output the remaining tuples in B, as these tuples belong to the result of B-A.

Therefore, we need to use a different formula from what is used for a sort-merge join operation to compute the estimate E_2 of $|B-A|$ (E_2 is defined in Section 2.) Here, $|R|$ denotes the cardinality of set R. Suppose that we have processed x tuples from A and y tuples from B. We found that among those y tuples of B, z tuples are not in A. Let $q_A=x/|A|$ and $q_B=y/|B|$. Then we use the same formula as that used in [9] to decide the percentage p: $p=max(q_A, q_B)$, which indicates the progress of the dominant input that is being scanned relatively faster.

Assume that at any time before we reach the end of either A or B, both the number of scanned B tuples that are not in A and the number of scanned B tuples are proportional to the percentage p. Then we can estimate that when we reach the end of either A or B, whichever is first, we will find that $E_3=z/p$ scanned B tuples are not in A. Also:

(1) If $q_A>q_B$, we reach the end of A first and $(1-q_B/p)\times|B|$ B tuples have not been scanned.
(2) If $q_A \leq q_B$, we reach the end of B first and all B tuples have been scanned.

That is, in either case, we can estimate that at that time, $E_4=(1-q_B/p)\times|B|$ B tuples have not been scanned and these B tuples are also not in A. Therefore, we can estimate $|B-A|$ to be $E_2=E_3+E_4$.

4.2.3. Introducing Segment Boundaries. In general, the output of a set operator can be pipelined to the next operator in the query plan. In this case, we need to divide the segment that contains the set operator into two segments, where the boundary of these two segments is defined at the output of the set operator. We use the following example to illustrate the point.

Example 3. Consider the query plan shown in Figure 4. Assume that both the set-difference operator and the aggregate operator with a group by clause are implemented using the sorting-based method [14]. Also, both relations A and B have been pre-sorted and the output of the set-difference operator is pipelined to the aggregate operator. Then if we do not introduce the segment boundary between the set-difference operator and the aggregate operator, the query plan contains only one segment.

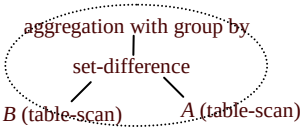

Figure 4. Query plan in Example 3.

When estimating the output cardinality of the segment, we can treat the aggregate operator with a group by clause like a selection operator σ [4]. The two dominant inputs of the segment are A and B.

Suppose that we have processed x tuples from A and y tuples from B. Also, we have generated z aggregate groups. We cannot use the method in [9] that is reviewed in Section 2.2 to estimate the final number of aggregate groups. This is because if we reach the end of A first, the remaining tuples in B can still generate new aggregate groups. However, these B tuples are not considered in the method in [9]. Also, as discussed above, we can estimate when we reach the end of A, how many B tuples have not been scanned. But it is difficult to estimate how many new aggregate groups will be generated from the remaining tuples in B (recall that we do not collect statistics inside segments [9].)

Therefore, we need to divide the segment that contains the set operator into two segments, as shown in Figure 5.

Then we can continuously refine the estimated output cardinalities of both segments easily.

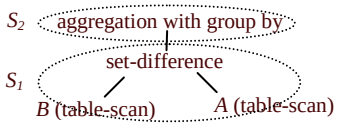

Figure 5. Query plan in Example 3 with redefined segments.

4.3. Handling Nested Queries

In this section, we describe a method for handling nested queries. Many nested queries (including all uncorrelated sub-queries) can be rewritten into equivalent un-nested queries, so they do not need any new techniques. In this section, we focus on techniques to handle correlated sub-queries that are not rewritten into uncorrelated sub-queries. The question of which queries can be rewritten is orthogonal to our discussion; our goal is to provide progress indicator techniques that work whenever an RDBMS decides to run a query plan containing a correlated sub-query, not to determine which query plans must be run as correlated sub-queries. As long as there are some query plans that are not rewritten, techniques along these lines will be needed.

In general, a nested query can have multiple levels. In our discussion, we focus on the top-level query and the second-level sub-query. All the deeper level sub-queries are embedded in the second-level sub-query. Similarly, while a nested query can contain multiple second-level sub-queries, in our discussion, we assume that each nested query contains only one second-level sub-query. It is straightforward to extend our techniques to handle multiple second-level sub-queries.

Query ***Q***:
select *A.a, A.e, A.h, B.b, B.k, C.m*
from *A, B, C*
where *A.a=B.b* and *B.d=C.c* and
A.e=(select *D.f* from *D* where *D.g=A.h* and *D.j=B.k*);

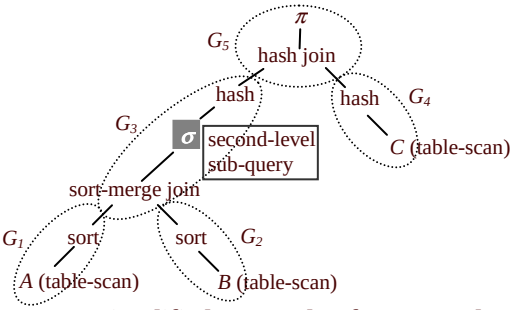

Figure 6. A simplified query plan for a nested query *Q* with a correlated second-level sub-query.

Consider a multi-level query whose second-level sub-query is correlated with the top-level query (Pattern 5.) To facilitate our description, when we draw the query plan, we do not show the second-level sub-query in detail. Rather, we use a *bold selection operator* (σ) to denote the selection condition that is evaluated based on the second-level sub-query. The content of the second-level sub-query is "hidden" in the bold selection operator. We call such a query plan a *simplified query plan*. For example, Figure 6 shows the simplified query plan for a multi-level query *Q* with a correlated second-level sub-query.

If we think of the bold selection operator as a normal selection operator, then the simplified query plan looks the same as a query plan for an un-nested query. Hence, we can use the same techniques in [9] to deal with the simplified query plan. However, there is a problem: the cost of evaluating the second-level sub-query is not counted, which can lead to very inaccurate estimates of progress.

To incorporate the cost of the sub-query, we divide the segment containing the bold selection operator into two segments, where the boundary of these two segments is defined at the input to the bold selection operator. For example, as shown in Figure 7, we divide the segment G_3 in Figure 6 into two segments: S_3 and S_4.

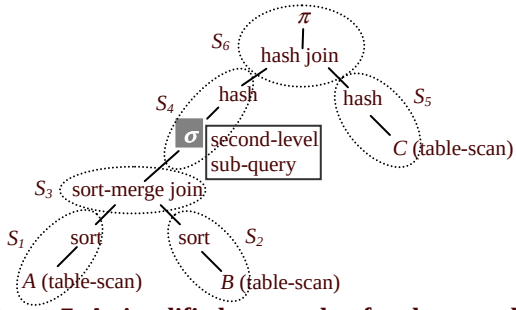

Figure 7. A simplified query plan for the nested query *Q* with redefined segments.

When we compute the cost of the segment *S* that contains the bold selection operator, we take into account the cost C_{total} of evaluating the second-level sub-query for all the input tuples of the bold selection operator.

We consider two ways to continuously refine the estimated C_{total}, which we call the "white box" method and the "black box" method. In the following, we first give an overview of both methods and explain why we prefer the black box method in this paper. Then we describe the black box method in detail.

(1) **White box method**: We treat the second-level sub-query as a normal query. We do the following two operations simultaneously.
   (a) Operation 1: Suppose we are currently processing the input tuple *t* of the bold selection operator. During the evaluation of the second-level sub-query, we use the techniques in [9] to continuously refine the estimated evaluation cost

of the second-level sub-query (for the input tuple t.)

(b) Operation 2: Based on the statistics collected during previous evaluations of the second-level sub-query, for each future input tuple t' of the bold selection operator, we use some method to continuously refine the estimated evaluation cost of the second-level sub-query.

C_{total} is estimated as

$$C_{total} = \sum_{\text{input tuple } t \text{ that has been processed}} C_t + \sum_{\text{input tuple } t' \text{ that has not been processed or is currently being processed}} C_{t'} .$$

For any input tuple t of the bold selection operator, C_t represents either the observed evaluation cost or the estimated evaluation cost of the second-level sub-query for tuple t, depending on whether or not tuple t has been processed.

(2) **Black box method**: We treat the second-level sub-query as a black box. We do the following two operations simultaneously.

(a) Operation 1: When we are evaluating the second-level sub-query for an input tuple of the bold selection operator, we observe the evaluation cost rather than continuously refining the estimated evaluation cost.

(b) Operation 2: We use the previously observed evaluation costs of the second-level sub-query to continuously refine the estimated C_{avg}, which is the average cost of evaluating the second-level sub-query once.

C_{total} is estimated as $C_{total}=N\times C_{avg}$, where N is the input cardinality of the bold selection operator.

Compared to the white box method, the black box method is simpler and less expensive. In our experiments, which are described in Section 5, we show that the black box method works well for nested queries when the correlated sub-query is evaluated many times and each iteration of the sub-query is relatively cheap.

There do exist certain cases, however, where the white box method is more desirable than the black box method. For example, if the input cardinality of the bold selection operator is small (say, one) and the second-level sub-query is complex and takes a long time to execute once, then the estimates provided by the black box method can be rather imprecise.

Now we describe the black box method in detail. We estimate C_{avg} in the following way. Before the query starts execution, the optimizer gives an estimate E_1 of the average cost of evaluating the second-level sub-query once. During query execution, we collect statistics about the average cost E_2 of evaluating the second-level sub-query once. For example, suppose we have evaluated the second-level sub-query x times and the observed total cost of evaluating the second-level sub-query x times is C_x, then $E_2=C_x/x$.

Assume that at some point, we have processed x input tuples of the bold selection operator. That is, we have evaluated the second-level sub-query x times. The percentage that the input to the bold selection operator has been processed is $p=x/N$ (recall N is the input cardinality of the bold selection operator.) Then, as in [9], we use the following heuristic linear interpolation formula to estimate C_{avg}: $C_{avg}=p\times E_2+(1-p)\times E_1$.

4.4 Additional Feature

In certain cases, the user wants to estimate the output cardinality of the query. For example, if the user suspects that the query will return too many answers (i.e., information overload [2]), he/she may want to either refine the query or ask the RDBMS to categorize the query results [2]. Therefore, it would be desirable to continuously refine the estimated output cardinality of the query and display it in the progress indicator interface. This feature can be done easily using the techniques in [9].

5. Performance

In this section, we present results from a prototype implementation of our techniques in PostgreSQL Version 7.3.4 [12]. In all our tests, our prototyped progress indicators could be updated every ten seconds with less than 2% overhead.

5.1. Experiment Description

Our measurements were performed with the PostgreSQL client application and server running on a Dell Inspiron 4000 PC with one 600MHz processor, 512MB main memory, one 40GB IDE disk, and running the Microsoft Windows XP operating system. (We repeated some of the experiments on a computer with a 2.4GHz processor, 512MB main memory, and one 73GB SCSI disk. The results were similar, so we omit them here.)

The seven relations used for the tests followed the schema of the standard TPC-R Benchmark relations [17]:

customer (custkey, name, address, nationkey, phone, acctbal, mktsegment),
orders (orderkey, custkey, orderstatus, totalprice, orderdate, ship-priority),
lineitem (orderkey, partkey, suppkey, linenumber, quantity, extendedprice, discount, tax, returnflag, linestatus),
part (partkey, name, mfgr, brand, type, size, container, retailprice).

Table 1. Test data set.

	number of tuples	total size
customer	50K	7.5MB
orders	1.5M	114MB
lineitem	6M	755MB
part	1K	0.14MB
customer_subset1	100	16KB
customer_subset2	2	306B
lineitem_subset	3M	378MB

The *customer_subset1* and *customer_subset2* relations have the same schema as the *customer* relation. The *lineitem_subset* relation has the same schema as the *lineitem* relation. In our tests, on average, each *customer* tuple matches ten *orders* tuples on the attribute *custkey*. Each *orders* tuple matches four *lineitem* tuples on the attribute *orderkey*. We built an index on the *partkey* attribute of the *lineitem* relation.

We evaluated the performance of our techniques in the following way:

(1) Before we ran queries, we ran the PostgreSQL statistics collection program on all the seven relations.

(2) We tested five queries:

(a) **Query Q_1**:

```
select c1.*, c2.acctbal, o.orderkey, o.totalprice,
        o.ship-priority
from customer c1, customer_subset1 c2, orders o
where mod(c1.custkey+c2.custkey, 100)=0 and
        c1.custkey=o.custkey;
```

(b) **Query Q_2**:

```
select * from orders order by custkey;
```

(c) **Query Q_3**:

```
select *
from lineitem l1, lineitem_subset l2
where l1.partkey=l2.partkey and l2.orderkey>0
order by l1.partkey;
```

(d) **Query Q_4**:

```
select *
from part p
where p.size<(select sum(l.quantity) from lineitem l
                where l.partkey=p.partkey);
```

(e) **Query Q_5**:

```
select *
from customer_subset2 c
where c.acctbal<(select sum(o.totalprice+l.extendedprice)
                from orders o, lineitem l
                where o.orderkey=l.orderkey and
          absolute(l.partkey)>0 and c.custkey<o.custkey);
```

(3) For each query, we performed an unloaded system test by running the whole query on an unloaded system. (We also performed some loaded system tests. The results are similar to that presented in [9], so we do not present them here.)

(4) Before we ran each test, we restarted the computer to ensure a cold buffer pool. (We repeated our experiments with a warm buffer pool. The results were similar, so we do not present them here.) In all tests, we stored the outputs from progress indicators into a file.

5.2. Test Results for Query Q_1

The purpose of the test with query Q_1 is to show that by explicitly considering the fact that different future segments may process *U*'s at different speeds, we can significantly improve the accuracy of the estimates provided by the progress indicator.

The query plan chosen by PostgreSQL for query Q_1 contains two join operators:

(1) The first join operator is a nested loops join operator. It computes the join between the *customer* relation and the *customer_subset1* relation. The optimizer determines that this join operator is a CPU-intensive operator.

(2) The second join operator is a hybrid hash join operator. It computes the join between the output of the nested loops join operator and the *orders* relation. The optimizer determines that this join operator is an I/O-intensive operator.

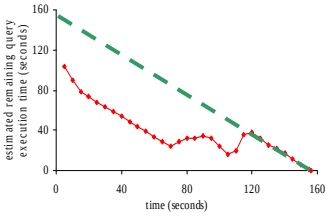

Figure 8. Remaining query execution time estimated over time (test for Q_1 - without considering different work unit processing speeds).

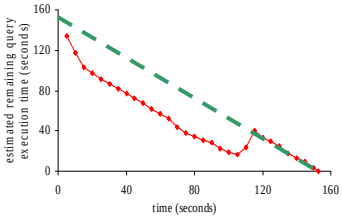

Figure 9. Remaining query execution time estimated over time (test for Q_1 - considering different work unit processing speeds).

We tested two cases. In the first case, we did not consider different work unit processing speeds. In the second case, we considered different work unit processing speeds. For these two cases, we show the remaining query execution time estimated by the progress indicator over time in Figure 8 and Figure 9, respectively. In each figure, the actual remaining query execution time is represented by the dashed line.

In the second case, the estimated remaining query execution time is much closer to the actual remaining query execution time than that in the first case. This is because during the nested loops join, due to caching, bytes are processed much faster than that during the hybrid hash join. As a result, in the first case, during the nested loops join, the progress indicator significantly underestimates the time required for the hybrid hash join and thus the remaining query execution time.

We performed another test with a query that first performs an index-scan, then a hybrid hash join. During

the index-scan, fetching one tuple may require up to one page of I/O. That is, during the index-scan, bytes are processed much more slowly than that during the hybrid hash join. In this case, by considering different work unit processing speeds, the progress indicator can significantly improve the accuracy of the estimated remaining query execution time. The results do not provide additional insights beyond the test results for query Q_1 and therefore have been omitted.

5.3. Test Results for Query Q_2

The purpose of the test with query Q_2 is to show that our progress indicator can continuously refine the estimated cost of a sort operation.

Query Q_2 sorts the *orders* relation according to the *custkey* attribute. We tested two cases, one in which the tuples in the *orders* relation were in random order, the other in which they were almost sorted on the *custkey* attribute.

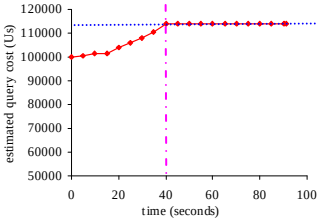

Figure 10. Query cost estimated over time (test for Q_2 - randomly ordered case).

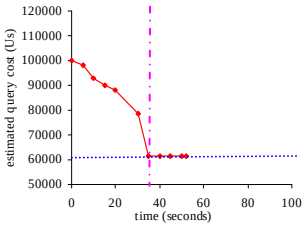

Figure 11. Query cost estimated over time (test for Q_2 - almost sorted case).

For these two cases, we show the query cost estimated by the progress indicator over time in Figure 10 and Figure 11, respectively. Each figure contains (1) a horizontal dotted line that represents the exact query cost, and (2) a vertical dashed-dotted line that represents the time that the first pass of sorting finishes.

PostgreSQL uses replacement sort. Hence, the query cost depends on the order that tuples are arranged in the *orders* relation. In both cases, at the beginning of query execution, the progress indicator starts with the same query cost estimated by the optimizer. Before the first pass of sorting finishes, the progress indicator continuously refines the estimated number and sizes of the sorted runs that will be generated at the end of the first pass. Hence, the query cost estimated by the progress indicator continuously approaches the exact query cost. After the first pass of sorting finishes, we know the exact values of the number and sizes of the sorted runs. Hence, we know the exact query cost.

In the randomly ordered case, at the beginning of query execution, the optimizer gives a fairly good estimate of the number (and also the sizes) of the sorted runs that will be generated at the end of the first pass. Hence, the progress indicator can estimate the query cost fairly precisely from the first second and only needs to make minor adjustment to this estimate during query execution.

In the almost sorted case, at the beginning of query execution, the optimizer overestimates the number of sorted runs that will be generated at the end of the first pass by fifty times (since the tuples in the *orders* relation are almost sorted.) As a result, the optimizer significantly overestimates the query cost. Hence, during query execution, the progress indicator needs to make major adjustment to the estimated query cost.

5.4. Test Results for Query Q_3

The purpose of the test with query Q_3 is to show that our progress indicator can continuously refine the estimates related to a sort-merge join operation.

The query plan chosen by PostgreSQL for query Q_3 computes a sort-merge join between the *lineitem* relation and the *lineitem_subset* relation. The sort key is *partkey*. We first sort the *lineitem* relation (the first sorting phase.) Then we sort the *lineitem_subset* relation (the second sorting phase.) Finally, we merge the sorted result of the *lineitem* relation and the sorted result of the *lineitem_subset* relation together (the merging phase.)

In the *lineitem* relation, the *partkey* attribute values are evenly distributed between 1 and 200K. In the *lineitem_subset* relation, except for one tuple whose *partkey=200K* and *orderkey=0*, the *partkey* attribute values are evenly distributed between 1 and 100K. Hence, after evaluating the select condition *l2.orderkey>0*, the *partkey* attribute values are evenly distributed between 1 and 100K in the sorted result of the *lineitem_subset* relation. However, PostgreSQL's optimizer does not know this and thinks that the maximal *partkey* attribute value in the sorted result of the *lineitem_subset* relation is still 200K.

Figure 12 shows the query cost estimated by the progress indicator over time, with the exact query cost indicated by the horizontal dotted line. There are two vertical dashed-dotted lines: the first one represents the time when the first sorting phase finishes and the second sorting phase starts, and the second one represents the time when the second sorting phase finishes and the merging phase starts.

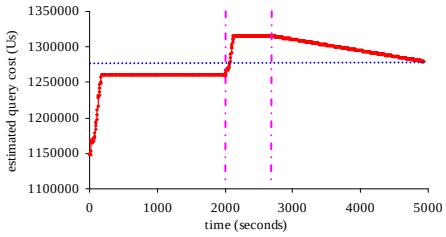

Figure 12. Query cost estimated over time (test for Q_3).

The behavior of the two sorting phases is similar to that discussed in Section 5.3. Hence, we focus our discussion on the merging phase. During the merging phase, the query cost estimated by the progress indicator keeps

decreasing until it reaches the exact query cost. The reason is as follows. Before the merging phase starts, the optimizer thinks that in order to complete the merging phase, we need to reach both the end of the sorted result of the *lineitem* relation and the end of the sorted result of the *lineitem_subset* relation. However, during the merging phase, the progress indicator gradually discovers that in order to complete the merging phase, we only need to scan half of the sorted result of the *lineitem* relation (since *100K/200K=50%*.)

PostgreSQL uses the sorting-based method to implement set operations. We performed several tests for queries that contain set operations. The results are similar to those for queries Q_2 and Q_3 and therefore have been omitted.

5.5. Test Results for Query Q_4

The purpose of the test with query Q_4 is to show that for those nested queries containing correlated sub-queries, the black box method works well when the correlated sub-query is evaluated many times and each iteration of the sub-query is relatively cheap.

Query Q_4 is a nested query that contains a correlated sub-query. The query plan chosen by PostgreSQL for the correlated sub-query is an index-scan on the *lineitem* relation, whose cost depends heavily on the number of *lineitem* tuples that match the *partkey* attribute value of the input *part* tuple.

There is correlation between the *part* relation and the *lineitem* relation. On average, for each *partkey* attribute value existing in the *lineitem* relation, there are 30 *lineitem* tuples whose *partkey* attribute is of this value. However, on average, for each *partkey* attribute value existing in the *part* relation, there are only 5 *lineitem* tuples whose *partkey* attribute is of this value. Because of the correlations in the data, PostgreSQL's optimizer significantly overestimates the evaluation cost of the correlated sub-query (and thus the query cost.)

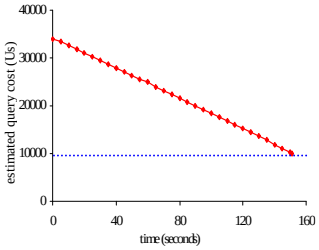

Figure 13. Query cost estimated over time (test for Q_4).

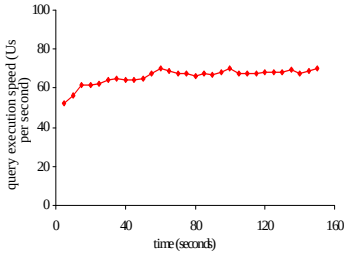

Figure 14. Query execution speed over time (test for Q_4).

Figure 13 shows the query cost estimated by the progress indicator over time, with the exact query cost indicated by the horizontal dotted line. We can see that the query cost estimated by the progress indicator keeps approaching the exact query cost. This is because the *part* relation contains a large number of tuples. For each tuple, evaluating the correlated sub-query once takes a small amount of time. Each time after the correlated sub-query is evaluated once, the progress indicator refines the estimated query cost.

Figure 14 shows the query execution speed monitored by the progress indicator over time. During the entire query execution, the monitored query execution speed remains much the same.

Figure 15 shows the remaining query execution time estimated by the progress indicator over time, with the actual remaining query execution time indicated by the dashed line. The closer to query completion time, the more precise the remaining query execution time estimated by the progress indicator. This is because the closer to query completion time, the more precise the query cost estimated by the progress indicator.

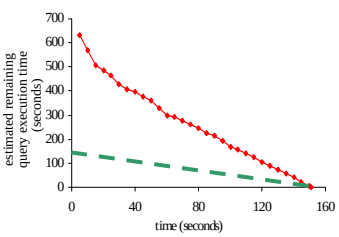

Figure 15. Remaining query execution time estimated over time (test for Q_4).

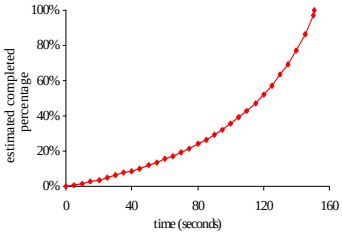

Figure 16. Completed percentage estimated over time (test for Q_4).

Figure 16 shows the progress indicator's estimate of the percentage of the query that has been completed over time. This percentage increases with time super-linearly. This is because: (1) work is continuously being done at a rather steady speed, and (2) the query cost estimated by the progress indicator keeps decreasing with time.

5.6. Test Results for Query Q_5

The purpose of the test with query Q_5 is to show that for nested queries containing correlated sub-queries, the black box method does not work well if the input cardinality of the bold selection operator is small while evaluating the correlated sub-query once (for an input tuple) takes a long time.

Query Q_5 is a nested query that contains a correlated sub-query. PostgreSQL does not give a good estimate of the selectivity of the select condition *absolute(l.partkey)>0* on the *lineitem* relation. Rather, for this select condition, PostgreSQL uses a default value 1/3 as an approximation to the real selectivity. This approximation is far from the real selectivity, which is 1 (since the absolute value of *l.partkey* is always positive.) Hence, PostgreSQL significantly underestimates the evaluation cost of the correlated sub-query (and thus the query cost.)

Figure 17 shows the query cost estimated by the progress indicator over time, with the exact query cost indicated by the horizontal dotted line. There are only two

tuples in the *customer_subset2* relation. Each time after a *customer_subset2* tuple is processed (i.e., after the correlated sub-query is evaluated once), the progress indicator refines the estimated query cost. However, since evaluating the correlated sub-query once takes a long time, the progress indicator refines the estimated query cost rather infrequently. Therefore, there are two sudden jumps in the query cost estimated by the progress indicator, each corresponding to a *customer_subset2* tuple.

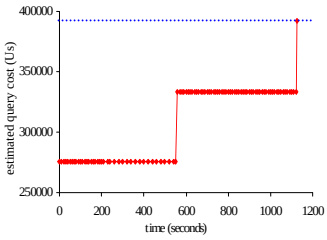

Figure 17. Query cost estimated over time (test for Q_3).

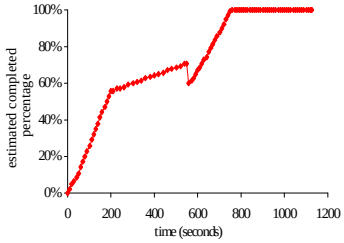

Figure 18. Completed percentage estimated over time (test for Q_3).

Figure 18 shows the progress indicator's estimate of the percentage of the query that has been completed over time. In general, this percentage keeps increasing with time. The only exception is that at 555 seconds, due to the sudden increase of the query cost estimated by the progress indicator, the estimated completed percentage drops suddenly.

There is an undesirable phenomenon in Figure 18. Starting from 755 seconds, according to the progress indicator's estimate, the query has finished execution, although the query keeps running until 1127 seconds. This is because the progress indicator underestimates the query cost and is unable to make up the query cost estimation error in time, as the progress indicator does not refine the estimated query cost a second time until the query completion time.

From the above discussion, we can see that compared to the previous work in [9], our techniques improve both the functionality and the accuracy of progress indicator at a minor increase in overhead (from 1% to 2%.)

6. Conclusion

Progress indicators for SQL queries are a desirable user-interface tool in RDBMSs. However, previously proposed techniques for supporting progress indicators for SQL queries are limited in both functionality and accuracy. In this paper, we propose a set of techniques to improve previously proposed techniques so that we can support non-trivial progress indicators for a wider class of SQL queries more precisely. Our experiments confirm the effectiveness of our techniques.

There is substantial scope for future work. For example:

(1) It is a non-trivial task to make the white box method (or the hybrid method) for handling correlated sub-queries work at a reasonable overhead. Also, it would be interesting to see if this method can bring in significant increase in accuracy from the user's perspective.

(2) How to support progress indicators for SQL queries in ORDBMSs is an interesting open problem. In this case, some of the challenges are: how to continuously refine the estimated costs of UDFs, spatial queries, etc. (UDFs match with Pattern 5 and hence the black box method may apply.)

(3) It would be interesting to investigate how to support progress indicators for SQL queries in parallel DBMSs. One challenge in this case is how to handle skew on different data server nodes.

We intend to pursue these issues in future work.

Acknowledgements

This work was supported by NSF grants CDA-9623632 and ITR 0086002.

References

[1] G. Antoshenkov. Dynamic Query Optimization in Rdb/VMS. ICDE 1993: 538-547.

[2] K. Chakrabarti, S. Chaudhuri, and S. Hwang. Automatic Categorization of Query Results. SIGMOD Conf. 2004: 755-766.

[3] R.L. Cole, G. Graefe. Optimization of Dynamic Query Evaluation Plans. SIGMOD Conf. 1994: 150-160.

[4] S. Chaudhuri, V.R. Narasayya, and R. Ramamurthy. Estimating Progress of Long Running SQL Queries. SIGMOD Conf. 2004: 803-814.

[5] M.A. Derr. Adaptive Query Optimization in a Deductive Database System. CIKM 1993: 206-215.

[6] Y.E. Ioannidis, R.T. Ng, and K. Shim et al. Parametric Query Optimization. VLDB Journal 6(2): 132-151, 1997.

[7] N. Kabra, D.J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. SIGMOD Conf. 1998: 106-117.

[8] D.E. Knuth. The Art of Computer Programming, Volume III: Sorting and Searching, Second Edition. Addison-Wesley, 1998.

[9] G. Luo, J.F. Naughton, and C. Ellmann et al. Toward a Progress Indicator for Database Queries. SIGMOD Conf. 2004: 791-802.

[10] V. Markl, V. Raman, and D.E. Simmen et al. Robust Query Processing through Progressive Optimization. SIGMOD Conf. 2004: 659-670.

[11] K.W. Ng, Z. Wang, and R.R. Muntz et al. Dynamic Query Re-Optimization. SSDBM 1999: 264-273.

[12] PostgreSQL homepage, 2003. http://www.postgresql.org.

[13] R. Ramamurthy. Personal communication, 2004.

[14] R. Ramakrishnan, J.E. Gehrke. Database Management Systems, Third Edition. McGraw-Hill, 2002.

[15] P.G. Selinger, M.M. Astrahan, and D.D. Chamberlin et al. Access Path Selection in a Relational Database Management System. SIGMOD Conf. 1979: 23-34.

[16] M. Stillger, G.M. Lohman, and V. Markl et al. LEO - DB2's LEarning Optimizer. VLDB 2001: 19-28.

[17] TPC Homepage. TPC-R benchmark, www.tpc.org.