Multi-query SQL Progress Indicators

Gang Luo[1], Jeffrey F. Naughton[2], and Philip S. Yu[1]

[1] IBM T.J. Watson Research Center
[2] University of Wisconsin-Madison
luog@us.ibm.com, naughton@cs.wisc.edu, psyu@us.ibm.com

Abstract. Recently, progress indicators have been proposed for SQL queries in RDBMSs. All previously proposed progress indicators consider each query in isolation, ignoring the impact simultaneously running queries have on each other's performance. In this paper, we explore a multi-query progress indicator, which explicitly considers concurrently running queries and even queries predicted to arrive in the future when producing its estimates. We demonstrate that multi-query progress indicators can provide more accurate estimates than single-query progress indicators. Moreover, we extend the use of progress indicators beyond being a GUI tool and show how to apply multi-query progress indicators to workload management. We report on an initial implementation of a multi-query progress indicator in PostgreSQL and experiments with its use both for estimating remaining query execution time and for workload management.

1 Introduction

Recently, [4, 6, 11, 12] proposed progress indicators (PIs) for SQL queries in RDBMSs. For a SQL query, a PI keeps track of the work completed and continuously estimates the remaining query execution time. [4, 6, 11, 12] proposed a set of techniques to implement single-query PIs. By single-query, we mean that in estimating the progress of a SQL query Q, these estimators only consider the current load and the progress of query Q itself, ignoring the effect of concurrently running queries and future queries. The main contributions of this paper are the first proposal of a multi-query PI, the exploration of its performance as compared to single-query PIs, and an application of the multi-query PI to problems arising in workload management.

Clearly there are cases where a single-query PI gives bad estimates. For example, if one query is substantially impeding the progress of another, but the first query is about to finish, a single-query PI will grossly overestimate the remaining execution time of the second query. Avoiding such behavior was our original motivation for developing multi-query PIs.

When estimating the remaining execution time for a query Q, a multi-query PI considers Q, other concurrently running queries, and, if available, predictions about new queries that can be expected to arrive while Q is running. As multi-query PIs consider more information than single-query PIs, they can provide more accurate estimates. A reasonable concern is whether we are depending on accurate predictions of the future. The answer is no – our multi-query PIs continuously monitor the system

Y. Ioannidis et al. (Eds.): EDBT 2006, LNCS 3896, pp. 921 – 941, 2006.

and adjust their predictions as time progresses. The closer the predictions about future queries are to reality, the better the initial estimates – but eventually the PIs will detect and correct their estimates even in situations in which their initial estimates were based on highly inaccurate information about the future.

In the published literature, SQL PIs have been proposed as a graphical user interface (GUI) tool [6, 11]. In this paper, we also present a new motivation for considering multi-query PIs: workload management. We formulate several workload management problems and show how to solve them by using information provided by multi-query PIs. Traditionally, workload management is static in that once workload management decisions are made, they are not changed. In this paper, we exploit multi-query PIs to facilitate more dynamic workload management. PIs are used to continuously monitor the system status. If the system status differs significantly from what was predicted, the original workload management decisions are revised accordingly. That is, our workload management methods are adaptive, hence they are consistent with the industry trend of autonomic computing [8] and automatic administration [13].

The rest of the paper is organized as follows. Section 2 describes our multi-query PI. Section 3 discusses three workload management problems, and describes our solution to each workload management problem by using the information provided by our multi-query PIs. Section 4 discusses some practical considerations for building multi-query PIs. Section 5 presents results from an initial implementation of our techniques in PostgreSQL. We discuss related work in Section 6 and conclude in Section 7.

2 Multi-query Progress Indicator

In this section, we describe our multi-query PI. The single-query PIs in [11, 12] (the PIs described in [4, 6] predict only percentage of completion, not remaining query execution time) work roughly as follows. For a query Q, the PI initially takes the optimizer's estimated cost for Q measured in some unit we call U's. The choice of U can be somewhat arbitrary, so for concreteness we let U represent the amount of work required to process one page of bytes. At any time during Q's execution, based on the statistics collected so far, the PI refines the estimated remaining query cost c. The PI also continuously monitors the current query execution speed s, and the remaining query execution time is estimated as $t=c/s$.

Although monitoring the current query's execution speed means that the single-query PI implicitly considers the impact of other queries running in the system (since the measured speed will be slower if other queries are running), the single-query PI does not explicitly consider other queries in that it has no idea how long they will run. In the following, we show how to build a multi-query PI that explicitly considers other queries. The main idea in multi-query PIs is that they should predict future execution speeds by considering the expected remaining execution time for concurrently running queries, and, if statistics are available, they should even attempt to predict the impact of queries that might arrive while the current query is running.

2.1 Initial Simplifying Assumptions

We first describe some simplifying assumptions that enable a framework for describing and analyzing multi-query PIs. This framework is useful even when they only roughly approximate true system behavior. Section 4 gives our rationale for these assumptions and discusses how our PI is affected when they are relaxed.

Assumption 1: The RDBMS processes work units at a constant rate C (work units per second) that is independent of the number of running queries.

Assumption 2: The PI has perfect knowledge about the remaining cost c_i of each running query Q_i.

Assumption 3: Queries execute at speed proportional to the weights associated with their priorities. In more detail, suppose n queries Q_1, Q_2, …, and Q_n are running in the RDBMS concurrently. Q_i ($1 \le i \le n$) has priority p_i. The corresponding weight for priority p_i is w_i. Then each Q_i ($1 \le i \le n$) is executed at speed $s_i = C \times w_i / W$, where $W = \sum_{j=1}^{n} w_j$.

2.2 Multi-query Progress Estimation

We first consider the simple case where no new queries arrive while the current queries are executing. Although this is an artificial case, it is useful in providing insight for the more general case we discuss in Section 2.4. Also, as we will see, this case turns out to be important in its own right in the context of workload management.

Suppose n queries Q_1, Q_2, …, and Q_n are running in the RDBMS, where Q_i ($1 \le i \le n$) has priority p_i and weight w_i. The current time is time 0. To estimate the remaining query execution time, the n queries Q_1, Q_2, …, and Q_n are first sorted in the ascending order of c_i/s_i. That is, after sorting, we have $c_1 / s_1 \le c_2 / s_2 \le \cdots \le c_n / s_n$ (or equivalently,

$$c_1 / w_1 \le c_2 / w_2 \le \cdots \le c_n / w_n \tag{1}$$

This order will be useful in the discussion below.

The execution of the n queries is divided into n stages. At the end of each stage, a query finishes execution. Stage i ($1 \le i \le n$) lasts for time t_i. We call this *the standard case* in the remainder of this paper.

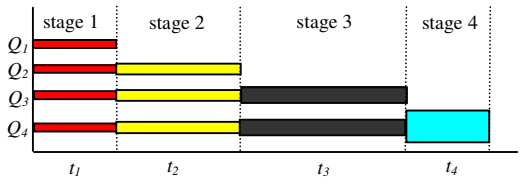

Fig. 1. Sample execution of $n=4$ queries

To give the reader a feeling of how the n queries will behave, Figure 1 shows a sample execution of $n=4$ queries. All these queries have the same priority. At the end of stage i ($1 \le i \le n$), query Q_i finishes execution. During each stage i, the amount of work

completed for Q_j ($i \le j \le n$) is re-presented as a rectangle, where the height of the rectangle represents the execution speed of Q_j.

Now we describe our algorithm in detail. We first discuss stage 1. Recall that $c_1 / s_1 \le c_2 / s_2 \le \cdots \le c_n / s_n$. Hence, among all the n queries Q_1, Q_2, …, and Q_n, Q_1 will be the first one to finish, and it will finish at time $t_1 = c_1 / s_1$.

During stage 1, for each i ($2 \le i \le n$), the amount of work completed for query Q_i is $a_i^{(1)} = s_i \times t_1 = s_i \times c_1 / s_1 = c_1 \times w_i / w_1$. Hence, at the end of stage 1, the remaining cost of Q_i ($2 \le i \le n$) is $c_i^{(1)} = c_i - a_i^{(1)} = c_i - c_1 \times w_i / w_1$.

Now we discuss stage 2. During this stage, there are $n-1$ queries running: Q_2, Q_3, …, and Q_n. Each Q_i ($2 \le i \le n$) executes at speed $s_i^{(1)} = C \times w_i / W^{(1)}$, where $W^{(1)} = \sum_{j=2}^{n} w_j = W - w_1$.

For each i ($2 \le i \le n$), $c_i^{(1)} / s_i^{(1)} = (c_i / w_i) \times W^{(1)} / C - (c_1 / C) \times W^{(1)} / w_1$. According to (1), $c_2 / w_2 \le c_3 / w_3 \le \cdots \le c_n / w_n$. Hence, $c_2^{(1)} / s_2^{(1)} \le c_3^{(1)} / s_3^{(1)} \le \cdots \le c_n^{(1)} / s_n^{(1)}$. That is, among the queries Q_2, Q_3, …, Q_n, Q_2 will finish first, and it will take time t_2, where $t_2 = c_2^{(1)} / s_2^{(1)}$.

During stage 2, for each i ($3 \le i \le n$), the amount of work completed for query Q_i is $a_i^{(2)} = s_i^{(1)} \times t_2 = s_i^{(1)} \times c_2^{(1)} / s_2^{(1)} = c_2^{(1)} \times w_i / w_2$. Hence, at the end of stage 2, the remaining cost of Q_i ($3 \le i \le n$) is $c_i^{(2)} = c_i^{(1)} - a_i^{(2)} = c_i^{(1)} - c_2^{(1)} \times w_i / w_2 = c_i - c_1 \times w_i / w_1 - (c_2 - c_1 \times w_2 / w_1) \times w_i / w_2$ $= c_i - c_2 \times w_i / w_2$.

This procedure is repeated for all the n stages to compute all the t_i's ($1 \le i \le n$). By induction, we find that Q_1, Q_2, …, and Q_n will finish in the order Q_1, Q_2, …, and Q_n. That is, at the end of each stage i, Q_i finishes execution. At time 0, the remaining execution time of Q_i is $r_i = \sum_{j=1}^{i} t_j$.

The time complexity of the above algorithm is $O(n \times \ln n)$, and the space complexity is $O(n)$. (The derivation details are omitted due to space constraints.)

2.3 Non-empty Query Admission Queues

An RDBMS typically contains a query admission queue. If the RDBMS is overloaded, newly arrived queries will be put into the query admission queue rather than starting execution immediately. Since queries already in the query admission queue are also "known" queries, a multi-query PI can extend its visibility into the future by examining this queue. An example of this is given in our experimental evaluation in Section 5.

2.4 Considering Future Queries

The above discussion assumed that no new queries arrive while the queries currently in the RDBMS are running. In general, new queries will keep arriving, hence they will influence the load on the RDBMS, and a PI must somehow account for these queries. These queries are different from those in the admission queue – they have not yet arrived and predictions about them necessarily involve speculation.

If nothing at all is known about the future, then one guess about future loads is as good as another, and there is no point in trying to do any forecasting. However, in practice we think it is rare that absolutely nothing can be predicted about the future, and that rough approximate information is likely to be available. The goal of the PI then is to use such approximate information to improve its guesses about the future.

In our approach, we assume that we know the average query priority $\bar{p}$, the average cost $\bar{c}$, and the average arrival rate λ. (The average inter-arrival time is then $\bar{t} = 1/\lambda$.) Of course such predictions are only approximate, and as will be shown in our experimental section, they need not be very accurate for the multi-query PI to outperform a single-query PI. In many applications, the overall load on the system over time is at least partially predictable, and these numbers can be obtained from past statistics. Then we proceed in a way similar to that in Section 2.2. The only difference is that every $\bar{t}$ seconds, we predict that a new query with priority $\bar{p}$ and cost $\bar{c}$ will arrive at the RDBMS, and it is considered in the PI's estimates.

3 Workload Management

Workload management for RDBMS has been extensively studied (e.g., [3, 7, 14, 19, 23]), and major commercial RDBMSs come with workload management tools [8, 10, 13, 15]. However, due to a lack of information about the progress of queries running in the RDBMS, these tools cannot always make intelligent decisions.

For example, consider the following scheduled maintenance problem. Suppose at time 0, we need to schedule maintenance (e.g., we need to install some new software, or add several new data server nodes to a parallel RDBMS), and that the maintenance is scheduled to begin at time *t*. A common practice is to perform two operations [22]:

O1: Starting from time 0, no new queries are allowed to enter the RDBMS.

O2: The existing queries are allowed to run until time *t*, when any queries that have not completed are aborted.

The challenge is how to choose the maintenance time *t* so as to minimize the amount of lost work without over-delaying the maintenance. In general, workload management tools do not know which queries can finish by time *t*, so the DBA needs to guess an arbitrary time that he/she thinks is appropriate. However, if we can estimate query running times, then more intelligent decisions can be made. For example, operation *O2* can then be replaced with the following two operations:

O2': Predict which queries cannot finish by time *t* and abort them at time 0. (Note: aborting queries will reduce the load on the RDBMS and hence change the estimate about which queries cannot finish by time *t*.)

O3: Let other queries in the RDBMS keep running. Suppose at time *t*, some of these queries have not finished execution (this is possible if our estimation has errors). Then they are either aborted or allowed to run to completion – the appropriate action depends on both the application requirement and the estimate of how soon those queries are going to finish.

Compared to operation *O2*, operations *O2'* and *O3* have the following advantages. First, even for the same maintenance time *t*, by aborting some "hopeless" queries, more queries can finish. Second, the amount of lost work can be controlled by adjusting the maintenance time *t*.

As a second example, suppose that for some reason, the DBA needs to speed up the execution of a target query *Q*. The DBA decides to do this by choosing one running query (the victim query) and blocking its execution. In this case, a common approach is to choose the victim query to be the heaviest resource consumer. However, if it happens that this victim query will finish quickly, then blocking the execution of this query will not speed up the execution of *Q* as much as blocking some other query that has a longer remaining execution time. If the remaining execution time of the running queries can be estimated, we can avoid choosing a victim query that is about to finish.

From the above discussion, we can see that it is desirable to give the workload management tool more information about the remaining execution time of running queries, and to use this information to make more intelligent decisions.

In this section, we discuss how to do this for three workload management problems. Variants of these workload management problems are frequently encountered in practice. Our goal is not to give an exhaustive account of all ways that PIs could be useful for workload management; rather, it is to demonstrate by example that the information provided by multi-query PIs can improve the quality of decisions made by workload management tools.

In our discussion, for ease of description, we assume that the *n* queries Q_1, Q_2, …, and Q_n are numbered so that $c_1 / s_1 \le c_2 / s_2 \le \cdots \le c_n / s_n$. Furthermore, we present our techniques for making workload management decisions based on the current system status (the *n* queries Q_1, Q_2, …, and Q_n).

3.1 Single-Query Speed Up Problem

Suppose we want to speed up the execution of a target query Q_i ($1 \le i \le n$). A natural choice is to increase the priority of Q_i. However, if Q_i is already of highest priority, then we must either block one or more other queries, or lower the priority of one or more other queries. In this paper, the first alternative is considered.

Assume that at time 0, we want to speed up the execution of query Q_i by blocking $h \ge 1$ victim queries. Which *h* queries should be blocked? This is our single-query speed up problem. We first consider the simple case where $h=1$, and then discuss $h \ge 1$. Intuitively, the optimal victim query Q_v should satisfy the following two conditions:

C1: Q_v should be the heaviest resource consumer.
C2: If not blocked, Q_v should run for the longest time (at least longer than Q_i).

In other words,

C1: The weight of Q_v, w_v, should be the largest.
C2: c_v/s_v, or *v* (since all queries are sorted in the ascending order of c_j/s_j), should be the largest.

It is not always possible to find a victim query that satisfies both conditions. Rather, the optimal victim query should be chosen based on a tradeoff between these two conditions. This tradeoff leads to a mathematical optimization problem.

The sketch of our method is as follows. The *n-1* queries Q_1, Q_2, …, Q_{i-1}, Q_{i+1}, Q_{i+2}, …, and Q_n are divided into two sets: $S_1=\{Q_1, Q_2, \ldots, Q_{i-1}\}$ and $S_2=\{Q_{i+1}, Q_{i+2}, \ldots, Q_n\}$. In either set S_j ($j=1, 2$), the best candidate victim query Q_{v_j} is picked. This is achieved by quantifying the "benefit" of speeding up the execution of the target query Q_i that is gained by blocking the execution of the victim query. Then the optimal victim query Q_v is the better one of Q_{v_1} and Q_{v_2}.

Our algorithm contains three steps.

Step 1: The queries in set S_2 are examined first. In this case, condition *C2* does not matter, as each Q_j ($i+1 \le j \le n$) runs longer than Q_i. To satisfy condition *C1* as much as possible, a natural choice is to choose query Q_{v_2} to be the query with the highest weight. That is, $w_{v_2} = \max\{w_j \mid i+1 \le j \le n\}$.

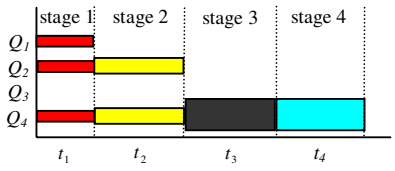

Fig. 2. Sample execution of *n=4* queries (the execution of Q_3 is blocked at time 0)

We justify this choice formally. Suppose Q_m ($i+1 \le m \le n$) is chosen as the victim query. To compute the "benefit" of blocking Q_m, the following key technique is used. The entire period of executing the *n-1* queries Q_1, Q_2, …, Q_{m-1}, Q_{m+1}, …, and Q_n is divided into *n* stages. During each stage *j* ($1 \le j \le n$), except for Q_m, the amount of work completed for Q_k ($1 \le k \le n$, $k \ne m$) remains the same as that in the standard case (recall that the standard case is defined in Section 2.2).

It is easy to see that except for stage *m*, at the end of each stage *j* ($1 \le j \le n$, $j \ne m$), a query (Q_j) finishes execution. Also, at stage *j* ($1 \le j \le i$), compared to the standard case, the execution of each Q_k ($j \le k \le n$, $k \ne m$) is sped up by a factor of $\sum_{p=j}^{n} w_p / (\sum_{p=j}^{n} w_p - w_m)$. As a result, the duration of stage *j* is shortened from t_j to $t'_j = t_j \times (\sum_{p=j}^{n} w_p - w_m) / \sum_{p=j}^{n} w_p$. In other words, the duration of stage *j* is shortened by $\Delta t_j = t_j - t'_j = t_j \times w_m / \sum_{p=j}^{n} w_p$.

Hence, the remaining execution time of query Q_i is shortened by $T_m = \sum_{j=1}^{i} \Delta t_j = \sum_{j=1}^{i} (t_j / \sum_{p=j}^{n} w_p) \times w_m$. In order to maximize T_m, w_m needs to be maximized.

Step 2: Now the queries in set S_1 are examined. Suppose Q_m ($1 \le m \le i-1$) is chosen as the victim query. To compute the "benefit" of blocking Q_m, the technique of Step 1 is used again. The entire period of executing the *n-1* queries Q_1, Q_2, …, Q_{m-1}, Q_{m+1}, …, and Q_n is divided into *n* stages. During each stage *j* ($1 \le j \le n$), except for Q_m, the amount of work completed for Q_k ($1 \le k \le n$, $k \ne m$) remains the same as that in the standard case.

The remaining execution time of query Q_i is shortened by $T_m = c_m / C$. This is because in the first *i* stages, by blocking the execution of Q_m at time 0, c_m's work is saved. To maximize T_{v_1}, we should choose Q_{v_1} such that $c_{v_1} = \max\{c_m \mid 1 \le m \le i-1\}$.

Step 3: The optimal victim query Q_v is the better one of Q_{v_1} and Q_{v_2}. That is, $T_v = \max\{T_{v_1}, T_{v_2}\}$.

From the above analysis, it can be seen that at time 0, by blocking a query Q_m ($1 \le m \le n$) whose remaining execution time is r_m, no more than r_m can be saved from the execution of other queries. This agrees with our assertion at the beginning of Section 3 that if the victim query will finish soon, blocking its execution will not help much. The time complexity of the above algorithm is $O(n \times \ln n)$, while the space complexity is $O(n)$.

We now consider the special case where all n queries Q_1, Q_2, …, and Q_n have the same priority. In this case, the solution to the problem is greatly simplified:

(1) If $i<n$, the optimal victim query is any Q_j ($i+1 \le j \le n$).
(2) If $i=n$, the optimal victim query is Q_{n-1}.

The time complexity of this solution algorithm is $O(n)$. This is because in this case, there is no need to either sort the n queries Q_1, Q_2, …, and Q_n in ascending order of c_j/s_j or compute all the t_j's. Rather, given the target query Q_i whose remaining cost is c, to find the optimal victim query, all the other queries need to be scanned (at most) once. If we find a query whose remaining cost is no less than c, we are done. Otherwise the query with the largest remaining cost is picked.

Now we return to the general case of our single-query speed up problem, where $h \ge 1$. Suppose the h victim queries are chosen to be Q_{g_1}, Q_{g_2}, …, and Q_{g_h}, where $\{g_1, g_2, ..., g_h\} \subseteq \{1, 2, ..., n\} - \{i\}$. Assume by blocking Q_{g_j} ($1 \le j \le h$) at time 0, the remaining execution time of Q_i is shortened by T_{g_j}. Then from an analysis similar to that above, it can be shown that by blocking the h victim queries Q_{g_1}, Q_{g_2}, …, and Q_{g_h} at time 0, the remaining execution time of Q_i is shortened by $\sum_{j=1}^{h} T_{g_j}$.

Based on this observation, the following greedy method can be used to deal with the general case of our single-query speed up problem. First, the optimal victim query is chosen according to the algorithm presented previously. Then, among the remaining queries, the next optimal victim query is chosen. This procedure is repeated h times to get h victim queries. These h victim queries are the optimal h victim queries.

3.2 Multiple-Query Speed Up Problem

Suppose now that we want to block a single query to speed up the execution of the other $n-1$ queries. Which query should be blocked? This is the multiple-query speed up problem.

Suppose Q_m ($1 \le m \le n$) is chosen as the victim query. From an analysis similar to that in Section 3.1, we know that for each j ($1 \le j \le m$), compared to the standard case, the duration of stage j is shortened by $\Delta t_j = t_j \times w_m / \sum_{p=j}^{n} w_p$. Also, each stage j ($m+1 \le j \le n$) is the same as that in the standard case.

At each stage j ($1 \le j \le m$), n-j queries Q_j, Q_{j+1}, …, Q_{m-1}, Q_{m+1}, …, and Q_n are running, and their total response time is improved by $(n-j)\times\Delta t_j$. Hence, by blocking Q_m at time 0, the total response time of all the other n-1 queries Q_1, Q_2, …, Q_{m-1}, Q_{m+1}, …, and Q_n is improved by $R_m = \sum_{j=1}^{m}(n-j)\times\Delta t_j = \sum_{j=1}^{m}(n-j)\times t_j \times w_m / \sum_{p=j}^{n} w_p$. To maximize R_m, we should choose the optimal victim query Q_v such that $R_v = \max\{R_m \mid 1 \le m \le n\}$. The time complexity of the above algorithm is $O(n \times \ln n)$. Also, the space complexity of the above algorithm is $O(n)$.

3.3 Scheduled Maintenance Problem

In this section, we discuss the problem mentioned at the beginning of Section 3: how can we choose the maintenance time t and the queries to abort so that the amount of lost work can be minimized without over-delaying the maintenance? In practice, the amount of lost work L_w can be defined in multiple ways. Due to space constraints, in this paper, only the following two cases are discussed:

Case 1: L_w is the total amount of work that has been completed for the queries that will be aborted.

Case 2: L_w is the total cost of the queries that will be aborted. In this case, it is more appropriate to call L_w the amount of unfinished work, since the aborted queries need to be rerun after the RDBMS is restarted.

For each i ($1 \le i \le n$), let e_i denote the amount of work that has been completed for query Q_i at time 0. We only describe the solution to Case 1. For Case 2, the solution is the same except that for each i ($1 \le i \le n$), e_i needs to be replaced with e_i+c_i. Recall that c_i is the remaining cost of query Q_i at time 0.

In our discussion, we assume that the overhead of aborting queries is negligible compared to the query execution cost. This will be true in a primarily read-only environment. In general, aborting jobs may introduce non-negligible overhead. How to handle this case is left as an interesting area for future work.

We define the *system quiescent time* to be the time when all the n queries Q_1, Q_2, …, and Q_n (except for those queries that are aborted, if any) finish execution. The estimated system quiescent time is our estimation of the earliest time when the system maintenance can start. Suppose for each i ($1 \le i \le n$), by aborting Q_i at time 0, the system quiescent time is shortened by V_i. It is easy to see that $V_i = c_i/C$. Also, by aborting h queries Q_{g_1}, Q_{g_2}, …, and Q_{g_h} at time 0, where $1 \le h \le n$ and $\{g_1, g_2, ..., g_h\} \subseteq \{1, 2, ..., n\}$, the system quiescent time is shortened by $\sum_{j=1}^{h} V_{g_j}$.

Our goal is to maximize $\sum_{j=1}^{h} V_{g_j}$ while minimizing $\sum_{j=1}^{h} e_{g_j}$. This is the standard knapsack problem [5]. Consequently, we use a greedy method to solve it. First the n queries Q_1, Q_2, …, and Q_n are re-sorted in ascending order of e_i/V_i (recall that we assume that originally, the n queries Q_1, Q_2, …, and Q_n are sorted in ascending order of c_i/s_i). After re-sorting, we have $e_{f_1}/V_{f_1} \le e_{f_2}/V_{f_2} \le \cdots \le e_{f_n}/V_{f_n}$ (or equivalently,

$e_{f_1} / c_{f_1} \le e_{f_2} / c_{f_2} \le \cdots \le e_{f_n} / c_{f_n}$), where $\{f_1, f_2, ..., f_n\}$ is a permutation of $\{1, 2, ..., n\}$. Then we keep aborting Q_{f_1}, Q_{f_2}, …, until the system quiescent time becomes satisfactory.

4 Revisiting the Assumptions

Sections 2 and 3 are based on the three assumptions in Section 2.1. Although we believe that these assumptions approximate reasonable system behavior, in practice, the system behavior will deviate from that predicted by these assumptions. Overall, the impact of relaxing these assumptions is that the multi-query PI now gives only approximate estimates, and for this reason the "advice" it gives for workload management becomes heuristic rather than provably optimal. As mentioned in the introduction, our method is adaptive and can make dynamic adjustments to ameliorate previous errors. This can mitigate the effect of imprecise estimates. We discuss this in more detail in the following subsections.

4.1 Assumptions 1 and 2

Assumption 1 says that for all the running queries, the RDBMS processes *C* units of work per second in total. When Assumption 1 is not valid, the PI may either underestimate the speedup that will occur when a query terminates (if, for example, the system was thrashing until that query finished), or overestimate the speedup that will occur when a query terminates (if, for example, a CPU-intensive query terminates and the other queries are all I/O-intensive). While this will hurt the accuracy of the multi-query PI, it is still likely to be superior to that of a single-query PI, which pays no attention whatsoever to other queries.

Assumption 2 says that for each running query, the exact remaining cost is known. If these estimates turn out to be far off, the accuracy of the multi-query PI will again be harmed, although again it is likely to be better than that of a PI that completely ignores these other queries. These scenarios could be dealt with in a number of ways, including augmenting the PI to have a more accurate performance model (including better modeling of a lightly loaded system), being willing to tolerate inaccuracies in the PI's estimates, or even revisiting the workload management decisions periodically if the inaccuracies of the model have resulted in suboptimal decisions. Which approach is best under which circumstances is an interesting question for future research. We suspect that because the PI adjusts its estimates "on the fly" as it discovers that they are inaccurate, it may not be worth the effort to improve the precision of these estimates – but this is still an open question and also scope for interesting future research.

4.2 Assumption 3

Assumption 3 says that each query's execution speed is proportional to the weight associated with its priority. This assumption is mainly for concreteness, for to discuss

workload management problems in the context of queries with priorities, some policy needs to be specified for how priority affects execution speed. If a system implements a different approach to priorities, a priority model for the multi-query PI would need to be developed for that approach. Even if the system attempts to implement a policy where execution speed is proportional to priority, the true behavior may be different for a variety of reasons – one example is the details of query interactions (e.g., a high-priority I/O-intensive query might not substantially block a low-priority CPU-intensive query, or two queries compete for/share buffer pool pages and thus slow down/speed up each other's execution). As was the case in Section 4.1, these factors will harm the accuracy of the multi-query PI, and ways to deal with this include building a more accurate model, tolerating errors, or periodically revising decisions.

4.3 Other Practical Considerations

The time complexity of most algorithms described in this paper is $O(n \times \ln n)$, where n is the number of queries in the RDBMS. This is a cause for some concern if n is large. However, in general, we would expect that the majority of queries are short (i.e., queries that can finish in a few seconds) and not really candidates for progress estimation or relevant individually for workload management. For this reason we think it is reasonable for the purposes of workload management and progress estimation to ignore these short queries and focus on long-running queries. Thus the effective n in the preceding formula is likely to be small and the computational cost will be small.

5 Performance Evaluation

In this section, we present results from a prototype implementation of our techniques in PostgreSQL Version 7.3.4 [17].

5.1 Experiment Environment

Our measurements were performed with the PostgreSQL client application and server running on a Dell Inspiron 8500 PC with one 2.2GHz processor, 512MB main memory, one 40GB disk, and running the Microsoft Windows XP operating system. The relations used for the experiments followed the schema of the standard TPC-R Benchmark relations [21]:

lineitem (partkey, quantity, extendedprice, …),
part_i (partkey, retailprice, …) ($i \geq 1$).

Table 1. Test data set

	number of tuples	total size
lineitem	24M	3.02GB
part_i ($i \geq 1$)	$10 \times N_i$	$1.4 \times N_i$ KB

In our experiments, each *part_i* relation ($i \geq 1$) contains $10 \times N_i$ tuples. (How the N_i's are chosen is discussed later.) The *partkey* attribute values in the *part_i* relations are randomly distributed between the minimal *partkey* attribute value and the maximal *partkey* attribute value in the *lineitem* relation. In a given *part_i* relation, all the tuples have different *partkey* attribute values. On average, each *part_i* tuple matches with 30 *lineitem* tuples on the attribute *partkey*. We built an index on the *partkey* attribute of the *lineitem* relation.

The following queries were tested, which find parts that are on average selling for 25% below suggested retail price:

Query Q_i ($i \geq 1$): select * from part_i p where p.retailprice×0.75>
(select sum(l.extendedprice)/sum(l.quantity) from lineitem l where l.partkey=p.partkey);

Each query 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. We repeated our experiments with other kinds of queries. The results were similar and thus not presented here.

Before we ran queries, we ran the PostgreSQL statistics collection program on all the relations. PostgreSQL does not support priorities for queries. Hence, all the queries Q_i ($i \geq 1$) have the same priority. In all experiments, the outputs from each PI were stored into a separate file.

5.2 Multi-query Progress Indicator

Three experiments were performed to compare single-query PIs with multi-query PIs. In the first two experiments, we ensure that no new queries arrive at the RDBMS while the queries under consideration are running. In the third experiment, we explore the situation in which new queries keep arriving at the RDBMS.

5.2.1. Multiple Concurrent Query (MCQ) Experiment

In this experiment, ten queries were used: Q_i ($1 \leq i \leq 10$). Their N_i's followed a Zipfian distribution with parameter $a=1.2$. At time 0, each of these ten queries was at a random point of its execution.

This experiment was performed multiple times. A typical run is examined here. In this run, among the $n = 10$ queries Q_i ($1 \leq i \leq 10$), we focus on a typical large query Q. For this Q, Figure 3 shows the remaining query execution time estimated by the PI over time. Figure 4 shows the query execution speed monitored by the PI over time. In Figure 3, the actual remaining query execution time is represented by the dashed line, the

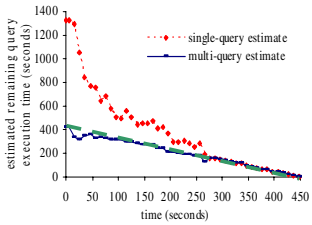

Fig. 3. Remaining query execution time estimated over time for Q (MCQ experiment)

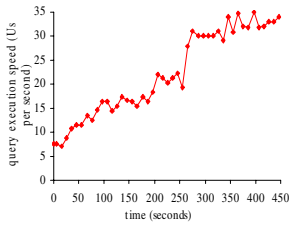

Fig. 4. Query execution speed monitored over time for Q (MCQ experiment)

single-query estimate is provided by the single-query PI, and the multi-query estimate is provided by the multi-query PI.

From time 0 to the completion time of query Q, due to the completion of other concurrent queries, the execution speed of Q gradually increases by almost a factor of five. The multi-query PI is able to predict the change in the load on the RDBMS while the single-query PI cannot. As a result, the multi-query estimate is fairly close to the actual remaining query execution time, while the single-query estimate differs from the actual remaining query execution time by almost a factor of three at the beginning.

5.2.2. Non-empty Admission Queue (NAQ) Experiment

In this experiment, three queries were used: Q_1, Q_2, and Q_3, with $N_1=50$. $N_2=10$. $N_3=20$. The query admission policy was that at any time, at most two queries could run concurrently in the RDBMS. At time 0, Q_1, Q_2, and Q_3 entered the RDBMS admission queue. Q_1 and Q_2 started execution first, with Q_3 blocked until Q_2 finishes.

The purpose of this experiment is to show that when the admission queue is not empty, multi-query PIs that consider the admission queue can provide more accurate estimates than either single-query PIs or multi-query PIs that do not consider the admission queue. In effect, examining the admission queue lets the PI see farther into the future.

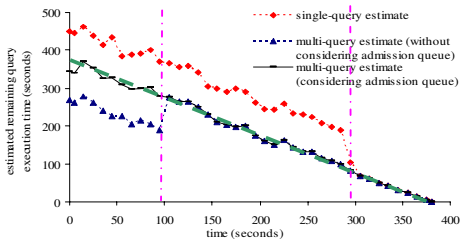

Fig. 5. Remaining query execution time estimated over time for Q_1 (NAQ experiment)

For query Q_1, Figure 5 shows the remaining query execution time estimated by the PIs over time. There, the actual remaining query execution time is represented by the dashed line. Two vertical dashed-dotted lines are used, one representing the start time of Q_3, and another representing the finish time of Q_3.

The execution time of query Q_1 is longer than the sum of the execution time of Q_2 and the execution time of Q_3. Before Q_2 finishes, without considering Q_3 that is waiting in the admission queue, neither the single-query PI nor the multi-query PI can accurately predict the load on the RDBMS after the completion of Q_2. Hence, the multi-query estimate considering the admission queue is more precise than the other approaches.

At the 97th second, query Q_2 finishes and Q_3 starts. The query admission queue becomes empty. The multi-query PI is able to predict that Q_3 will finish before Q_1 and then the execution speed of Q_1 will increase, while the single-query PI incorrectly assumes that the execution speed of Q_1 will remain the same during the execution of Q_1. As a result, the multi-query estimate becomes more precise than the single-query estimate until Q_3 finishes at the 291st second.

5.2.3. Stream Concurrent Query (SCQ) Experiment

In this experiment, at time 0, ten queries Q_i ($1 \le i \le 10$) were running in the RDBMS and each of them was at a random point of its execution. New queries kept arriving at the

RDBMS according to a Poisson process with parameter λ. (The unit of λ is second^{-1}.) For both Q_i's ($1 \le i \le 10$) and new queries, their N_i's followed a Zipfian distribution with parameter $a=2.2$. (We also tested other values of a. The results are similar and thus omitted.)

Consider any Q_i ($1 \le i \le 10$). Suppose the actual remaining query execution time is t_{actual}. At time 0, the multi-query PI estimates the remaining query execution time to be t_{multi}. The relative error of the multi-query estimate is defined as $|t_{multi} - t_{actual}| / t_{actual} \times 100\%$. The relative error of the single-query estimate is defined in a similar way.

Among all Q_i's ($1 \le i \le 10$), the one with the largest remaining cost at time 0 will finish last and is thus called the *last finishing query*. The test was repeated one hundred times (one hundred runs). Unless otherwise specified, all the reported numbers are averaged over these one hundred runs.

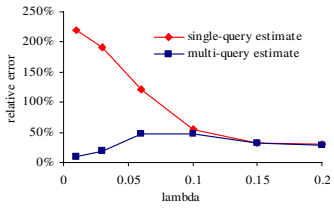

Fig. 6. Relative error of estimated remaining execution time for the last finishing query ($a=2.2$)

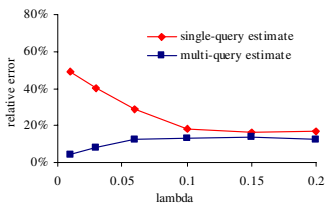

Fig. 7. Average relative error of estimated remaining execution time for all ten queries ($a=2.2$)

We first discuss the case where the multi-query PI knows the exact average arrival rate λ and the exact average cost $\bar{c}$ of future queries. For the last finishing query, Figure 6 shows the relative error of the estimated remaining execution time. For all Q_i's ($1 \le i \le 10$), Figure 7 shows the average relative error of the estimated remaining execution time.

When producing estimates, the multi-query PI considers both concurrently running queries and future queries. In contrast, the single-query PI incorrectly assumes that the load will remain stable in the future. As a result, the relative error of the multi-query estimate is always smaller than that of the single-query estimate.

When the system is stable, the relative error of the single-query estimate decreases as λ increases. This is because the larger the λ, the closer to reality the assumption made by the single-query PI. In contrast, the relative error of the multi-query estimate increases with λ, as the faster new queries arrive, the larger and the more random their influence on existing queries. Note that the stable system case is the most common case encountered in practice. In this case, the relative error of the multi-query estimate is much smaller than that of the single-query estimate.

When $\lambda > 0.07$, new queries come faster than the RDBMS can process them and thus the system becomes unstable. In this case, the influence of new queries on existing queries becomes fairly large and random. Hence, single-query and multi-query estimates have roughly the same (large) relative error.

Among all Q_i's ($1 \le i \le 10$), the last finishing query gets the largest and most random influence from new queries. Consequently, PIs provide the least precise estimate for the last finishing query. This leads to the effect that for both single-query and

multi-query estimates, the average relative error for the ten queries is smaller than the relative error for the last finishing query.

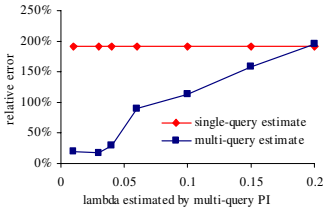

Fig. 8. Relative error of estimated remaining execution time for the last finishing query (*a=2.2* , *lambda=0.03*)

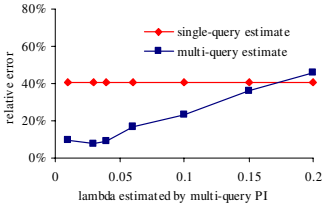

Fig. 9. Average relative error of estimated remaining execution time for all ten queries (*a=2.2* , *lambda=0.03*)

Now we discuss the case where the multi-query PI cannot estimate λ, the average arrival rate of future queries, precisely. We include this experiment to illustrate one example of the multi-query PI detecting when its estimates were wrong and then adapting and correcting its estimates. This is not the only way it does so; like single-query PIs, multi-query PIs also react to incorrect cost estimates (due perhaps to bad cardinality estimates or an inaccurate hardware cost model) and incorrect assumptions about how concurrently executing queries affect the performance of a given query (even single-query PIs notice, e.g., that they have slowed down when another query starts, even though they do not know why, or how long the slowdown might last, or if a similar slowdown might occur again in the future from a yet-to-arrive query.) Because we have explored this sort of adaptivity in our prior work [11, 12], we do not explore it here. Instead, we focus on a kind of adaptivity unique to multi-query PIs, i.e., adapting to errors in expected query arrival rate.

Let $\lambda=0.03$. The multi-query PI makes its estimate based on λ' while $\lambda' \neq \lambda$. For the last finishing query, Figure 8 shows the relative error of the estimated remaining execution time. For all Q_i's ($1 \leq i \leq 10$), Figure 9 shows the average relative error of the estimated remaining execution time.

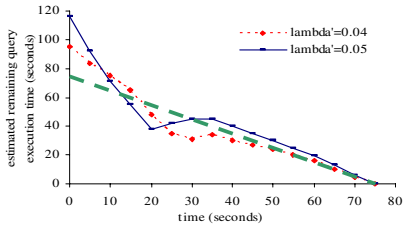

Fig. 10. Remaining query execution time estimated by multi -query PI over time (lambda=0.03)

The bigger the difference between λ' and λ, the more inaccurate the multi-query estimate. However, unless λ' is more than five times larger than λ, the relative error of the multi-query estimate is always smaller than that of the single-query estimate. This shows that, at least in these tests, even somewhat inaccurate information about the future is better than no information about the future.

We pick a typical run among the one hundred runs. In this run, for the last finishing query, Figure 10 shows the remaining query execution time estimated by the multi-query PI over time. There, the actual remaining query execution time is represented by the dashed line. At the beginning, due to the incorrectly estimated arrival rate λ', the

multi-query estimate is quite different from the actual remaining query execution time. The bigger the difference between λ' and λ, the more inaccurate the multi-query estimate. However, the multi-query PI is adaptive and can correct its own errors. The closer to query completion time, the more precise the multi-query estimate is.

In summary, as long as there is some reasonable (approximate) information about the future load, the multi-query PI can provide (often much) more accurate estimate of remaining query execution time than the single-query PI. This information need not be extremely accurate - the multi-query PI is adaptive and can correct its own errors over time.

5.3 Workload Management

Section 3 discussed three workload management problems. The experiment results for these three workload management problems were similar, since similar techniques were used for each problem. Accordingly, in this section, only the experiment results for Case 2 of the scheduled maintenance problem are presented, where the amount of unfinished work is defined as the total cost of all queries that will be aborted.

5.3.1. Experiment Description

We wanted to simulate a typical situation in practice, where the number of small queries submitted to the RDBMS is much larger than the number of large queries submitted to the RDBMS. To achieve this, a large number of queries Q_i ($i \geq 1$) are used. We let all the N_i's follow a Zipfian distribution with parameter $a=2.2$. (We also tested other values of a. The results were similar and thus are omitted.) Note that N_i "represents" the cost of Q_i. Each Q_i ($i \geq 1$) has the same probability to be submitted to the RDBMS.

We evaluated the performance of our workload management techniques in the following way. At any time, $n=10$ queries Q_{f_j} ($f_j \geq 1$, $1 \leq j \leq 10$) are running in the RDBMS. At the time that a query Q_{f_j} finishes execution, a random k ($k \geq 1$) is picked and query Q_k is submitted to the RDBMS for execution. Hence, for all the queries Q_k submitted to the RDBMS, the N_k's follow a Zipfian distribution with parameter a.

A random time r_t is chosen. At time r_t, the RDBMS is inspected and decisions are made to prepare for system maintenance scheduled for t seconds later. By a simple mathematical derivation, it can be shown that for the $n=10$ queries Q_{g_j} ($g_j \geq 1$, $1 \leq j \leq 10$) running at time r_t, their N_{g_j}'s follow a Zipfian distribution with parameter $a-1$. Due to space constraints, we only describe the main ideas in the derivation while omitting the details. For a particular Q_k ($k \geq 1$), the probability that Q_k is running at time r_t is proportional to both the probability that Q_k is submitted and the cost of Q_k (larger queries will run longer and hence are easier to be "seen"). Thus, $probability(N_{g_j} = m) \propto (1/m^a) \times m = 1/m^{a-1}$.

We compare the following three methods:

No PI method: No PI was used. Rather, we performed operations *O1* and *O2* described in Section 3.

Single-query PI method: We used the single-query PI and performed operations *O1*, *O2'*, and *O3*. When operation *O2'* was performed, the query with the largest estimated remaining cost was first aborted. Then if necessary, we further aborted the query with the second largest estimated remaining cost, and so on.
Multi-query PI method: We used the multi-query PI and performed operations *O1*, *O2'*, and *O3*. When operation *O2'* was performed, the algorithm described in Section 3.3 was used.

In all three methods, at the scheduled maintenance time r_t+t, the queries that had not finished execution were aborted. The test was repeated ten times (ten runs). Unless otherwise specified, all the reported numbers are averaged over these ten runs.

For the $n=10$ queries Q_{g_j} ($g_j \geq 1$, $1 \leq j \leq 10$) running at time r_t, the total work TW is defined to be their total cost. The unfinished work UW is defined to be the total cost of those queries that are aborted between time r_t and the scheduled maintenance time r_t+t. (Recall that unfinished queries are aborted at time r_t+t.) Finally, t_{finish} is defined to be their remaining execution time under the *no interruption condition*. That is, under the condition that no new queries enter the RDBMS for execution and there is no scheduled maintenance so that none of the existing $n=10$ queries is aborted, all the existing $n=10$ queries can finish by time r_t+t_{finish}.

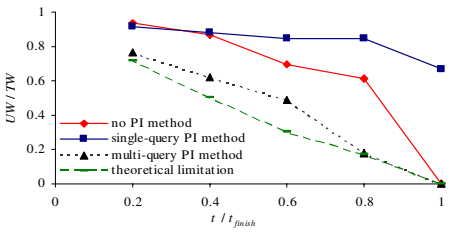

Fig. 11. Unfinished work of the three methods ($a=2.2$)

Figure 11 shows the unfinished work of the three methods. Note that the x-axis is t/t_{finish}. The y-axis is UW/TW. That is, both the x-axis and the y-axis have been "normalized," as the specific values of t_{finish} and TW vary from one run to another. In the rest of Section 5.3, when we refer to the amount of unfinished work, we always mean UW/TW.

Figure 11 also shows the theoretical limit that any method can achieve. This limit is computed using the exact information that comes from the actual run-to-completion execution of the $n=10$ queries. That is, based on this exact information, we compute the optimal set of queries that should be aborted at time r_t so that all the other queries can finish by the scheduled maintenance time r_t+t.

If $t=t_{finish}$, then in both the no PI method and the multi-query PI method, all queries can run to completion and there is no unfinished work. However, in the single-query PI method, 67% of the total work TW is not finished. The reason is as follows. In general, as can be seen from the experiment results in Section 5.2.1, the single-query PI tends to significantly overestimate the remaining execution time of those queries whose remaining costs are large at time r_t. Consequently, the single-query PI method thinks that a large portion of those queries cannot meet the scheduled maintenance time and aborts them unnecessarily at time r_t.

If $t<t_{finish}$, each of the three methods needs to abort queries. Among the three methods, the multi-query PI method has the least amount of unfinished work.

Compared to the no PI method, the multi-query PI method reduces the amount of unfinished work by 18%~44%. Compared to the single-query PI method, the multi-query PI method reduces the amount of unfinished work by 15%~67%. The reason for this reduction of work is as follows. First, in this case the multi-query PI can estimate the remaining query execution time fairly precisely. As a result, the multi-query PI method can estimate which queries cannot finish in time and abort them early so that more queries can meet the scheduled maintenance time. Second, as explained above, the single-query PI method aborts a large number of queries unnecessarily. Finally, the no PI method does not abort any query until the scheduled maintenance time. As a result, before the scheduled maintenance time, queries compete with each other for resources and are executed relatively slowly. Hence, compared to the multi-query PI method, fewer queries can meet the scheduled maintenance time.

In general, the no PI method has less unfinished work than the single-query PI method. However, when t is small (say, $t=0.2\times t_{finish}$), the no PI method has a little bit more unfinished work than the single-query PI method. This is because in this case, at time r_t, the single-query PI method aborts those queries whose remaining costs are large. Then other queries can run faster and finish by the scheduled maintenance time. In contrast, the no PI method does not abort any query at time r_t. This leads to the effect that all queries are executed very slowly. As a result, very few queries can meet the scheduled maintenance time. Note that if t is large, this effect is not so significant. This is because those queries whose remaining costs are small at time r_t are going to finish in a small amount of time. Then other queries can run faster.

In all ten runs, in most cases, the multi-query PI method performs better than both the no PI method and the single-query PI method. In the extreme case, compared to the no PI method and the single-query PI method, the multi-query PI method reduces the amount of unfinished work by 73% and 94%, respectively. (Note: the maximum percentage by which the multi-query PI method can reduce the amount of unfinished work is at most 100%.)

Occasionally, the multi-query PI method performs worse than either the no PI method or the single-query PI method. In the worst case, compared to the no PI method and the single-query PI method, the multi-query PI method increases the amount of unfinished work by 12% and 3%, respectively. This is because in the multi-query PI method, the greedy method only provides an approximate solution to the knapsack problem (finding the optimal solution to the knapsack problem is NP-hard). Also, the estimates provided by multi-query PIs have errors, mainly due to the imprecise statistics collected by PostgreSQL.

Among all the three methods, the multi-query PI method performs the closest to the theoretical limitation. When $t<t_{finish}$, compared to the theoretical limitation, on average, the multi-query PI method increases the amount of unfinished work by 3%~12%. In the worst case, the multi-query PI method increases the amount of unfinished work by 60%.

In summary, the average performance of the multi-query PI method is better than both that of the no PI method and that of the single-query PI method. The multi-query PI method can avoid extremely bad decisions. In the best case, the multi-query PI method can perform much better than both the no PI method and the single-query PI method. In

the worst case, compared to both the no PI method and the single-query PI method, the multi-query PI method performs only a little worse. Moreover, in a large number of cases, the multi-query PI method performs fairly close to the theoretical limitation.

6 Related Work

As mentioned in the introduction, all previous work on PIs has considered only single-query PIs, and none of the previous work has considered the application of PIs to workload management. Of course, there is a great deal of related work dealing with workload management. In general, the workload management problems discussed in Section 3 are scheduling problems. In this section, we give a brief survey of existing work related to scheduling.

Process scheduling has been exhaustively studied in the context of operating systems. In general, the process scheduler in the operating system does not know the job sizes [20]. By contrast, in our workload management environment, the query costs are known (or at least the query costs can be roughly estimated).

Process scheduling and transaction scheduling have been extensively studied in real-time operating systems [9, 24] and real-time database systems [1, 18]. In general, the main concern there is to meet deadlines rather than to maximize resource utilization. Most real-time systems are memory resident and the jobs there can be finished in a short amount of time (say, less than a few seconds). Hence, they need special time-cognizant protocols (e.g., to handle critical sections). Many real-time systems use hard deadlines. As a result, the jobs there are usually pre-defined (i.e., "canned" jobs). Also, almost all jobs there have deadlines.

In our workload management environment, we do not want to sacrifice resource utilization ratio in our general-purpose RDBMS. Queries may incur substantial I/Os and run for a long time. Therefore, short-term effects can be ignored and no special time-cognizant protocol is needed. Before queries are submitted to the RDBMS, we have only approximate knowledge of their resource requirements. Also, most queries do not have hard deadlines.

Job scheduling has been extensively studied in operations research and in computer science theory [2, 16]. In these studies, jobs usually have precedence constraints. On a single machine, jobs are typically executed one after another. Also, the main concern is to maximize the throughput/utilization ratio of the machines. In our database workload management environment, queries do not have precedence constraints and are executed concurrently.

7 Conclusion

In this paper we considered going beyond the state of the art in RDBMS PIs by considering the impact queries have on each other's progress and eventual termination. Our multi-query PIs consider not only currently executing queries, but

also predictions about queries that might arrive in the future. Even approximate information about future queries is helpful, and the PIs are adaptive in that they detect when they were given "bad" information about the future and correct their estimates as they learn more about the true query workload. We also demonstrated how to apply the resulting multi-query PIs to several workload management problems. As shown in experiments with a prototype implementation, for both estimating remaining query execution time and workload management purposes, the proposed multi-query PIs have significant advantages over single-query PIs or no PIs, suggesting that multi-query PIs may be a useful addition to RDBMSs.

Acknowledgements

We would like to thank Curt J. Ellmann and Michael W. Watzke for helpful discussions. This work was supported in part by NSF grants CDA-9623632 and ITR 0086002.

References

1. R.K. Abbott, H. Garcia-Molina. Scheduling Real-time Transactions: a Performance Evaluation. VLDB 1988: 1-12.
2. J. Blazewicz, K.H. Ecker, and E. Pesch et al. Scheduling Computer and Manufacturing Processes, Second Edition. Springer-Verlag, 2001.
3. M.J. Carey, S. Krishnamurthi, and M. Livny. Load Control for Locking: The 'Half-and-Half' Approach. PODS 1990: 72-84.
4. S. Chaudhuri, R. Kaushik, and R. Ramamurthy. When Can We Trust Progress Estimators for SQL Queries? SIGMOD Conf. 2005.
5. T.H. Cormen, C.E. Leiserson, and R.L. Rivest et al. Introduction to Algorithms, Second Edition. MIT Press, 2001.
6. S. Chaudhuri, V.R. Narasayya, and R. Ramamurthy. Estimating Progress of Long Running SQL Queries. SIGMOD Conf. 2004: 803-814.
7. C. Faloutsos, R.T. Ng, and T.K. Sellis. Predictive Load Control for Flexible Buffer Allocation. VLDB 1991: 265-274.
8. IBM Autonomic Computing homepage. http://www.research.ibm.com/autonomic.
9. S. Khanna, M. Sebree, and J. Zolnowsky. Realtime Scheduling in SunOS 5.0. USENIX Winter 1992: 375-390.
10. G.M. Lohman, S. Lightstone. SMART: Making DB2 (More) Autonomic. VLDB 2002: 877-879.
11. G. Luo, J.F. Naughton, and C.J. Ellmann et al. Toward a Progress Indicator for Database Queries. SIGMOD Conf. 2004: 791-802.
12. G. Luo, J.F. Naughton, and C.J. Ellmann et al. Increasing the Accuracy and Coverage of SQL Progress Indicators. ICDE 2005: 853-864.
13. Microsoft AutoAdmin Project homepage. http://research.microsoft.com/dmx/autoadmin.
14. D.T. McWherter, B. Schroeder, and A. Ailamaki et al. Priority Mechanisms for OLTP and Transactional Web Applications. ICDE 2004: 535-546.
15. Oracle Database: Manageability. http://www.oracle.com/database/index.html?db_manageability.html.

16. M. Pinedo. Scheduling: Theory, Algorithms, and Systems, Second Edition. Prentice Hall, 2001.
17. PostgreSQL homepage, 2005. http://www.postgresql.org.
18. K. Ramamritham. Real-Time Databases. Distributed and Parallel Databases 1(2): 199-226, 1993.
19. D. Shasha, P. Bonnet. Database Tuning: Principles, Experiments, and Troubleshooting Techniques. Morgan Kaufmann Publishers, 2002.
20. A. Silberschatz, P. Galvin, and G. Gagne. Operating System Concepts, Sixth Edition. John Wiley, 2002.
21. TPC Homepage. TPC-R benchmark, www.tpc.org.
22. Michael W. Watzke. Personal communication, 2005.
23. G. Weikum, C. Hasse, and A. Moenkeberg et al. The COMFORT Automatic Tuning Project. Inf. Syst. 19(5): 381-432, 1994.
24. W. Zhao. Editor. Special Issue on Real-Time Computing Systems. Operating Systems Review 23(3), 1989.