Dynamic Prioritization of Database Queries

Sivaramakrishnan Narayanan, Florian Waas

Greenplum Inc.
1900 S Norfolk St
San Mateo, CA, USA 94403
siva@greenplum.com
flw@greenplum.com

***Abstract*— Enterprise database systems handle a variety of diverse query workloads that are of different importance to the business. For example, periodic reporting queries are usually mission critical whereas ad-hoc queries by analysts tend to be less crucial. It is desirable to enable database administrators to express (and modify) the importance of queries at a simple and intuitive level. The mechanism used to enforce these priorities must be robust, adaptive and efficient.**

In this paper, we present a mechanism that continuously determines and re-computes the ideal target velocity of concurrent database processes based on their run-time statistics to achieve this prioritization. In this scheme, every process autonomously adjusts its resource consumption using basic control theory principles.

The self-regulating and decentralized design of the system enables effective prioritization even in the presence of exceptional situations, including software defects or unexpected/unplanned query termination with no measurable overhead.

We have implemented this approach in Greenplum Parallel Database and demonstrate its effectiveness and general applicability in a series of experiments.

I. Introduction

Databases and BI systems are a staple in modern enterprises and pivotal when it comes to supporting businesses in their decision making process. With an ever increasing breadth of data sources integrated in data warehousing scenarios and advances in analytical processing, the classic categorizations of query workloads such as OLTP, OLAP, loading, reporting, or massively concurrent queries have long been blurred. Mixed workloads have become a reality that today's database management systems have to be able to facilitate and support concurrently.

Processing of mixed workloads poses a series of interesting problems because different components of workloads compete for resources and, depending on the resource profiles, often impact each other negatively. This calls for mechanisms that allow users to assign *priorities* to different workloads which are enforced by allotting resources accordingly. The following list illustrates some of the most prominent scenarios of competing workloads with different priorities experienced by users:

- *Loading vs. reporting*. The quality of analytical processing relies, among other things, on the freshness of data as provided by periodic loads. Loads are typically performed in on-line fashion, i.e., the database system is used for reporting while loads are going on. The timely completion of loads is essential for all further analyses and processing. A variant of this scenario are nightly loads. Periodic loads are usually assigned higher priority than reporting workloads.
- *Tactical vs. strategic analysis*. Concurrent reports may differ in their general importance to the business in terms of timeliness with which the results are needed for business decisions. Tactical analysis reports typically have near-term impact on business and are often assigned higher priority than strategic analysis reports.
- *Operational workloads*. Operational emergencies require administrators to act quickly for damage control, e.g., rectify data contamination that is the result of faulty load procedures etc. These workloads should have precedence over other ongoing activity.
- *Operational safety*. By assigning ad-hoc users' workloads appropriately low priorities, administrators can limit the impact of experimental and accidentally complex queries without having to monitor all activity on the system continuously or even deny users access as a preventive measure.

In this paper, we describe a mechanism that automatically balances resources between workloads according to their priorities by (1) determining the ideal resource consumption rate for each individual query, and (2) employing a *back-off technique* based on control-theory principles wherein every participating process periodically checks if it has exceeded its *current* target rate and adjusts its consumption as necessary. The continuous application of this principle results in rapid convergence between actual and ideal resource consumption rate.

Prioritization in mixed workloads also has *dynamic* aspects which may complicate matters quite substantially:

- How to determine the ideal resource consumption rate under on-line query arrival?
- How to adjust priorities on-the-fly to resolve locking conflicts when the query of lowest priority keeps others from making progress?
- How to ensure maximal resource utilization?
- How to achieve robustness to be able to recover from unexpected situations such as the cancellation of a query

or a crash due to software defects?

Our solution addresses these concerns in a *wait-free* fashion without measurable overhead and achieves near optimal results under a broad variety of conditions. This mechanism is general and applicable to any time-shared resource such as I/O or network bandwidth. In this work, we focus on CPU time as the resource to control with this technique as it turned out to be highly effective for typical large-scale data warehouse configurations and workloads. The mechanism has been implemented in the Greenplum Parallel Database version 4.0 and initial feedback from customers confirmed both the effectiveness and accuracy of the solution.

The remainder of this paper is organized as follows: in Section II we discuss the motivation for our choice of CPU time as the primary resource. The analytic model of the problem in Section III is complemented with practical requirements from an engineering point of view in Section IV. Sections V and VI describe the architecture in detail and present extensions for parallelism. Finally, Section VII demonstrates the effectiveness of the solution with a series of experiments.

II. CPU-time Matters

In our design and implementation we chose CPU-time as the resource by which to prioritize queries, i.e., queries of higher priority get proportionately more CPU-time assigned. At first, this might appear counter-intuitive. However, in preliminary experiments we found CPU-time—not I/O nor network bandwidth—to be the single most effective resource for prioritization.

This observation is not product-specific but appears to be rather typical for the standard hardware configurations deployed in large-scale data warehousing [2]. Workloads tend to be CPU-heavy for several reasons:

1) Large-scale data warehouse systems or appliances are usually configured to provide significant I/O capacity. As a by-product, these configurations provide substantial I/O bandwidth.
2) OLAP workloads typically consist of scan-based queries of high query complexity and are thus more compute intensive. In addition, these workloads often leverage user-defined functions written in hosted languages such as Java, R, Python, etc., and usually CPU-bound.
3) Storage systems provide improved I/O bandwidth; ranging from striping of data over RAID sets [13] to new storage media such as Solid State Devices [11].

Also, a number of recently proposed techniques offer significant opportunity to improve the utilization of I/O bandwidth by eliminating unnecessary I/O operations. Examples include vertical partitioning of tables [5], compression (often along with vertical partitioning) [21], [20], or co-operative scans [22] to name just a few.

In the future, we expect these or similar techniques to be adopted in commercial products contributing to a continuing trend of emphasizing the role of CPU-time as the critical resource.

III. Model for the Problem

In this section, we present an abstraction of the problem introduced in Section I and establish a basic understanding of the objectives a solution for dynamic query prioritization has to accomplish. For clarify of exposition, we will refer to independent units of execution as *processes* regardless of whether the database system implementation uses processes or threads. Also, we assume that a query always corresponds to a single process i.e. query execution is serial. We will address extensions for a parallel query execution model in Section V. To express the *importance* of a query, we introduce the notion of *weight of a query*, or short *weight*. Consider the following example:

Example. Let Q_1 and Q_2 be two queries of weight w_1 and w_2 respectively. When executed individually, we expect the executor processes corresponding to these queries to achieve a CPU utilization of 100%. If w_1 is equal to w_2, they would each occupy 50% of the CPU over an overlapping interval of time T. If Q_1 is twice as important as Q_2, i.e., $w_1 = 2 \times w_2$, we expect Q_1 to see a CPU utilization of 66.66% and Q_2 33.33% of CPU time in that interval. •

Without loss of generality, we will primarily reason using the interval of time T where queries overlap. No assumptions are made regarding their exact start time. Consider a set of queries $\{Q_1, Q_2, ..., Q_N\}$ with weights $\{w_1, w_2, ..., w_N\}$ executing simultaneously in some interval of time T. The CPU time consumed by query Q_i should be

$$E_i^{(1)} = T \times \frac{w_i}{\sum_1^N w_j}$$

if only one CPU is available. Now, consider the case when K CPU's are available to execute the set of queries. Because of our assumption of a serial executor, a query can utilize at most one CPU during the interval of time T. Therefore, the CPU time spent by query Q_i should be

$$E_i^{(K)} = \min(T, T \times K \times \frac{w_i}{\sum_1^n w_j}) \tag{1}$$

We will abbreviate $E_i^{(K)}$ with E_i in the following when there is no risk of ambiguity. Equation 1 does not take into account the utilization of the system. For example, in a system with two CPU's and two queries with weights 500 and 50, we will end up with $E_1 = T$ and $E_2 = \frac{1}{11}T$ and, hence, severe underutilization of the second CPU.

To help model utilization explicitly, we introduce a coefficient of assignment such that $\{a_{i,j}\}$ represent the amount of time CPU P_j is assigned to query Q_i. All $a_{i,j}$ are subject to the following constraints:

$$\forall i, j \quad a_{i,j} \geq 0 \tag{2}$$
$$\forall j \quad \textstyle\sum_i a_{i,j} \leq T \tag{3}$$
$$\forall i \quad \textstyle\sum_j a_{i,j} \leq T \tag{4}$$

Equation 3 states that the total amount of time spent by all queries on a specific processor is less than the interval of interest T. Equation 4, along with other equations, implicity captures the constraint that query execution is single-threaded i.e. any valid solution to these equations can be translated to a serial execution schedule of queries and vice versa. It is rather straight-forward to handle the situation where a query can utilize multiple CPU's at a time as we will see later.

The utilization of the system is the total CPU time that may be utilized during the interval T. Avoiding under-utilization of CPU's is imperative. To capture this requirement, we add the constraint that the allocations have to maximize CPU utilization:

$$\sum_i \sum_j a_{i,j} = \min(K, N) \times T \tag{5}$$

Finally, we need to define an objective function that captures the notion of *proportionate sharing of CPUs*. We chose to model unfairness of a solution as:

$$UF = \frac{1}{K} \times \sum_i \frac{|\sum_j a_{i,j} - E_i|}{E_i} \tag{6}$$

The objective is to minimize UF while satisfying the constraints on $a_{i,j}$. The problem corresponds to a linearly constrained optimization problem and is amenable to solving with well-established techniques [17].

IV. Requirements

Section III described the abstract problem of sharing CPU between multiple queries. For a solution to the problem of dynamically prioritizing queries to be both practical and effective it has to meet the following additional requirements.

1) *Effectiveness*. The solution must provide a reasonably good approximation of the optimization problem as outlined in Section III. This includes maximizing resource utilization in situations that demand inversion of priorities, e.g., in order to avoid classic convoy effects caused by low priority queries that hold locks that prevent high priority queries from making progress. OS-based scheduling methods are *not* aware of these effects and therefore not suitable.
2) *Adaptivity*. The solution must be able to handle on-line arrival of queries and adjust to changing workloads. No assumptions can be made about the arrival rate of queries nor their duration or complexity. Even in cases where admission-control mechanisms throttle the number of incoming queries, it is highly desirable that the solution does not have to be aware of these parameters and components. Furthermore, it is desirable to allow for weights being adjusted mid-flight by administrators.
3) *Performance*. The solution must be lightweight and not incur any significant overhead. This includes both the overhead to determine target CPU times as well as possible wait times due to implementation issues, e.g., locking of additional shared data structures etc. It is therefore desirable to solve the optimization problem as detailed in the previous section only in approximation.
4) *Robustness*. The solution should be of low implementation complexity and must be able to cope with exceptional situations encountered during the execution of queries such as cancellation of queries or software defects, e.g., query reset due to crashed query processes.
5) *Portability*. The solution must be portable across different platforms and must not rely on OS-specific components or libraries. Even portable system calls may be unsuitable if they differ in implementation and hence performance on different platforms. This rules out most OS scheduler extensions.

In the case of Greenplum Parallel Database, an additional requirement arises from the parallel execution of queries: a query may consist of several independent query processes. In this case, all processes pertaining to the same query should be treated equivalent to single process. In other words, prioritization should be independent of the shape or complexity of the query including its potential for parallel execution. We discuss this topic in more detail in Section V-D.

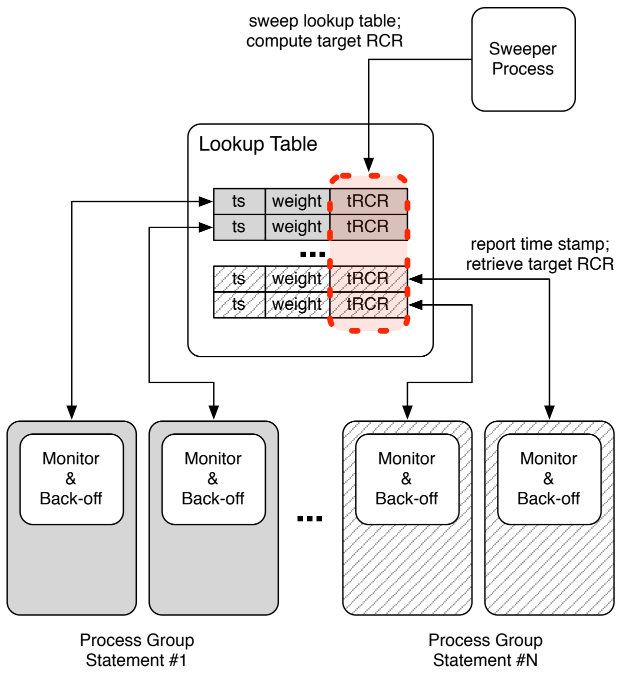
V. Dynamic Prioritization of Database Queries

In this section, we present the architecture and relevant implementation details for a solution for the abstract problem described in Section III that also addresses the requirements outlined in Section IV. The solution consists of two conceptually independent components: one dynamically determines ideal *Resource Consumption Rate (RCR)* for all processes involved; the other enforces it on all processes.

A. Basic Principles and Architecture

Database technology has a long-standing tradition of dealing with scheduling problems very effectively by deploying *collaborative scheduling*, i.e., instead of processing asynchronous interrupts all processes actively relinquish CPU and check frequently for situations that may need attentions such as cancellation requests, etc. This design requires software engineers to be rather disciplined during development but comes with significant payback: besides making for a simpler programming model in general, collaborative scheduling avoids convoy effects on locks as the calling code is fully aware of critical sections and can yield the CPU at the most suitable points in time [8]. As a result, this design achieves higher performance and more robust code. Our approach was inspired by the collaborative design paradigm: by letting all processes assess their actual RCR frequently, compare it against their individual target RCR, and adjust their own resource intake accordingly, the responsibility of enforcing target RCR's is distributed across all participating processes without creating a central bottleneck. This simple feedback loop, when executed at sufficiently high frequency, addresses the enforcing of the RCR.

In order to determine the target RCR for *any* process accurately, a full overview of *all* processes that are actively



sweep lookup table; compute target RCR
Sweeper Process
Lookup Table
ts
weight
tRCR
report time stamp; retrieve target RCR
Monitor & Back-off
Process Group Statement #1
Process Group Statement #N

Fig. 1. Architecture and process structure of the query prioritization mechanism.

competing for CPU at this point in time is required, i.e., we need to determine how many query processes are present and aggregate their weights. The fact that the number of process may be large–potentially very large–makes gathering this information a delicate task that has the potential to impact the surveyed processes or to incur significant CPU load by itself. Fortunately, it is sufficient to determine approximate values only as CPU usage is fluctuating rapidly. Finding an appropriate trade-off between the accuracy of the approximation and the statistical significance of the data collected is an emperical task that relies on implementation specifics of the underlying database system. See Section VII for more details on the calibration of our implementation.

For the approximate assessment of competing processes, we observe that some query processes may not be actively competing for CPU time. For example, processes that are blocked because they are waiting to acquire locks currently held by other processes or are stalled by a producer/consumer relationship and are waiting for data, do not need to be accounted for. In short, only processes that are making progress and are actually able to adjust their resource consumption need to be taken into account. Therefore, we can simply combine a process' periodic assessment of its RCR with reporting its data to a central instance, i.e., a data structure in shared memory. Given the high frequency with which processes check and, hence, report their RCR, this mechanism establishes a fairly accurate approximation of all RCRs.

The gathered data is then used to determine target RCRs asynchronously by "sweeping" the table periodically and re-computing all active processes' RCRs (we discuss the logic the sweeper process implements below in more detail). The targets are then retrieved by the processes by piggy-backing a request on the reporting of the actual RCR. This ensures that any process that is blocked is not be taken into account for CPU allotment. Figure 1 illustrates the overall architecture of the solution.

B. Sweeper Logic

The sweeper is implemented as a database service process. For simplicity of exposition, we assume it is aware of all the queries currently executing in the system i.e. all queries register with it at start time and un-register once they terminated. Our actual implementation goes beyond this and determines based on the reporting of RCR's by query processes what queries are in the system to avoid unnecessary synchronization at start time. The sweeper is run periodically and scans the shared memory data structure that contains all reported RCR's of currently running queries to determine

- "active" queries and their weights
- the fair share of CPU for each active query.

Determining if a query is active is done by looking at the timestamp of the last reporting of its RCR. If no new data has been reported within a small grace period, the query is considered inactive and unable to use CPU, e.g., blocked, waiting, etc. Once the set of active queries has been determined, we can compute their fair share of CPU time as target RCR by solving the optimization problem described in Section III. Formal methods for solving Linear Programming problems like the Simplex method or Ellipsoid method (cf. [18]) are prohibitively expensive given the high frequency at which we need to solve the problem as it evolves with newly arriving queries and queries frequently transitioning between being active and inactive. We therefore propose a lightweight heuristic to solve the following simplified variant of the original problem.

In the original problem we did break out the exact assignment of a query to a CPU. However, in practice the assignment of processes to CPU is best left to the scheduler of the OS kernel. This simplifies our problem and we introduce new variables r_i where:

$$r_i = \frac{\sum_j a_{i,j}}{T}$$

Thus, r_i $(0 < r_i \leq 1)$ is the fraction of CPU time available to Q_i regardless of which CPU executes the process and becomes the target RCR. Note that this eliminates T from the equations. This is important because T may not stay the same between subsequent invocations of the sweeper. The utilization constraint from Equation 5 can then be stated as:

$$\sum_i r_i = \min(K, N) \qquad (7)$$

Algorithm 1: Compute target RCR

```
input : Weights of active queries $w_i$'s
output: Target RCR$r_i$'s
/* Initialize */
1  $W \leftarrow \sum w_i$;
/* All queries are unallocated */
2  $\forall i \;\; r_i \leftarrow -1$;
3  for $j = 1$ to $K$ do
4      $pegger \leftarrow false$;
5      for $i = 1$ to $N$ do
6          if $(w_i * K \geq W) \wedge (r_i = -1)$ then
               /* This query pegs a CPU */
7              $pegger \leftarrow true$;
8              $W \leftarrow W - w_i$;
9              $r_i \leftarrow 1.0$;
               /* Decrease CPU availability */
10             $K \leftarrow K - 1.0$;
               /* Invariant: $\frac{K_+}{W_+} \geq \frac{K}{W}$ */
11             break;
12         end
13     end
14     if $\neg pegger$ then
           /* No peggers left */
15         goto rest;
16     end
17 end
/* No peggers left */
18 rest:
19 for $i = 1$ to $N$ do
20     if $r_i = -1$ then
           /* Unallocated query */
21         $r_i \leftarrow w_i \times \frac{K}{W}$
22     end
23 end
```

i.e., the fractions sum up to either the number of CPU's, in which case we maximized system utilization, or to the number of query processes; whichever is smaller. This condition reflects the discrete nature of CPU as a resource: no matter how high the priority of a query process, it can utilize only a single CPU at a time. We refer to the query process as *pegging a CPU* when it has an extremely high priority vis-a-vis the other queries and deserves exclusive access to a CPU. Note that this case arises only when $K > 1$. In the same spirit, we introduce F_i as a variant of E_i, the expected CPU share for a query (see Equation 1).

$$F_i = min(1, K \times \frac{w_i}{\sum_1^N w_j}) \tag{8}$$

A *pegger* per the model is a query where $F_i = 1.0$ i.e. the query deserves a full CPU because of its extremely high weight compared to the rest of the queries.

Algorithm 1 shows the pseudo code employed by the sweeper to solve for r_i's, given the active queries and their weights. The algorithm checks if there are any *peggers*. These queries are assigned a full CPU i.e. $r_i = 1.0$. If a *pegger* is found, then the algorithm re-starts from the first query to find any other peggers. This is because the number of available CPUs and the expected fair share of remaining queries has changed. If there are more queries than CPUs, we are guaranteed that there will be some non-peggers. In the second part of the algorithm we assign the r_i's of the non-peggers proportionally.

Example. Let there be 4 queries with weights $\{1, 100, 10, 1000\}$ with 3 CPUs available to share. The algorithm determines that Q_4 is a pegger ($1000 * 3 > 1111$) and assigns $r_4 = 1.0$. It restarts from the beginning and determines that Q_2 is now a pegger as well ($100 * 2 > 111$) and assigns $r_3 = 1.0$. There is only one CPU and no peggers left. The algorithm now goes the *rest* block and assigns CPU ratios of $r_1 = \frac{1}{11}$ and $r_2 = \frac{10}{11}$. Final assignments are $\{0.09, 1.0, 0.91, 1.0\}$. ●

The algorithm has two nice properties which make it suitable for our purposes:

- **Property 1:** The algorithm ensures that the modified utilization constraint (Equation 7) is satisfied i.e. there is no underutilization of CPU. We state this without proof.
- **Property 2:** After the algorithm completes, every query gets allocated a share not less than the expected value per the model i.e. $\forall i \; r_i \geq F_i$ (see Equation 8). We present an informal proof of this property.

The key to proving property 2 is the invariant in the *i* loop of the algorithm. In the invariant, W_+ and K_+ refer to the updated values of W and K after a pegger (Q_l) is identified by the algorithm. Specifically, $K_+ = K - 1$ and $W_+ = W - w_l$. Since Q_l is a pegger, we know that:

$$\begin{aligned} w_l \times K &\geq W \\ \implies w_l &\geq \frac{W}{K} \\ \implies W - w_l &\leq W - \frac{W}{K} \\ \implies \frac{K-1}{W-w_l} &\geq \frac{K}{W} \\ \implies \frac{K_+}{W_+} &\geq \frac{K}{W} \end{aligned}$$

Thus, the ratio $\frac{K}{W}$ can never decrease in the algorithm. Therefore, if a query was not pegged by the algorithm, i.e. it is processed in the *rest* block. The effective allocation r_i is guaranteed to be more than the expected value F_i. This completes the proof of property 2.

The algorithm guarantees maximal utilization of CPU while ensuring that no query gets less than its expected share of CPU. The algorithm runs in $O(N)$ (K is constant for a

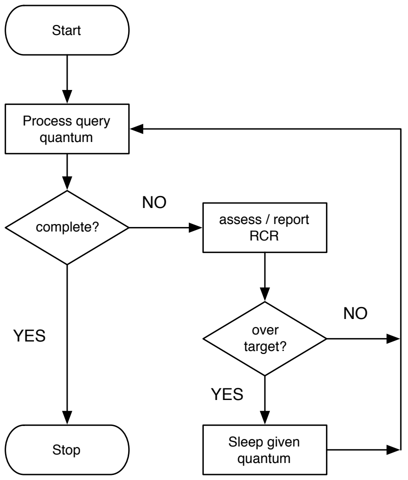


Fig. 2. Feedback Control Loop of the Backoff module for reporting and adjusting process' RCR; simplified model

database installation), and is overall fairly lightweight. Typically, the number of concurrent queries (N) we have seen are in the dozens or low hundreds in production systems. We also implemented a simple optimization to cache the results r_i's and reuse them if there has been no change in the active queries or their weights.

C. Back-off Heuristic

The back-off heuristic is a subroutine that the query execution engine calls frequently. The back-off module uses a `sleep()` subroutine to achieve its fair share of CPU usage as calculated by the sweeper process. The `sleep()` subroutine is implemented using the `select` system call with a timeout. By calling the `sleep()` routine, the database process tells the OS that it is relinquishing its right to the CPU for the determined amount of time, thus reducing its RCR. However, calculating sleep time required from a database process is not straightforward since it depends on the state of the entire system and no mathematical system can precisely model a real physical system. Therefore, we employed control theory principles to achieve our goals. Feedback control is a mechanism of applying input to cause system variables to conform to desired values called the reference. In our situation, sleep time is the input, system variable is actual CPU usage and reference is the desired CPU usage r_i.

The backoff module maintains a variable *sleepTime* and routinely sleeps that amount. It also routinely checks its CPU usage using system calls. If the actual CPU usage varies from the required CPU usage, the control function changes *sleepTime* accordingly. The control function we employ is:

$$sleepTime_{m+1} = sleepTime_m \times \frac{Actual\ CPU\ usage}{r_i}$$

In our calculations, only the CPU usage since the last check is considered. If the actual CPU share is higher than the desired value, *sleepTime* grows. We found that this simple control function works very well in practice and converges rapidly. Also, in our experiments this function has proven to dampen the system sufficiently without causing underutilization. Figure 2 illustrates the feedback loop in a simplified fashion.

As explained earlier, the back-off module also records a reporting timestamp in the shared state that enables discerning active and inactive queries. This mechanism covers both regular and exceptional situation, e.g., when a query crashes due to a software defect or is waiting on database locks. As soon as a waiting query awakens after acquiring its locks, it will enter the set of active queries by reporting to the sweeper.

D. Extensions for Parallelism

The Greenplum Parallel Database (GPDB) has a shared-nothing, massively parallel processing (MPP) architecture. It enables parallel execution of a query using both intra-operator and inter-operator parallelism.

A unit of intra-operator parallelism is a *segment* - an operator is parallelized by partitioning its inputs. Each segment has a dedicated set of CPUs assigned to it and does not share state information with other segments. The execution of a single query may involve database processes spread across segments. We implemented our solution on a per-segment basis i.e. database processes belonging to the same segment compete with one another for the segment's CPUs. Thus, we avoid any expensive communication over the network and minimize overhead of the mechanism.

Inter-operator parallelism is an orthogonal form of parallelism supported by GPDB and it poses an interesting challenge to our solution. Figure 3 presents an example of inter-operator parallelism. The query here involves the join of four relations and the execution plan depicted is a bushy join plan. The plan is executed using three database processes as shown in the figure (note that each segment will have a set of three processes as well). Thus, there are three processes corresponding to a single query competing for CPU with other queries. In the example, $Process_3$ is blocked on $Process_1$ and cannot make progress. Therefore, there are only 2 "active" processes for this query. Note that the number of "active" processes involved in a query may vary during the lifetime

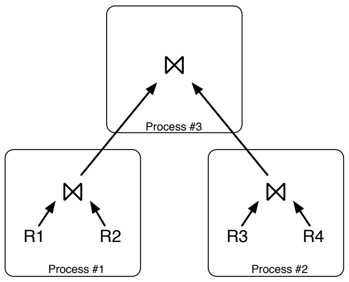


Fig. 3. Parallel Query Execution of a Bushy Join Plan in GPDB

of a query.

The solution we described extends in a fairly straightforward way for such scenarios. The RCR for each query r_i is divided equally among all the "active" database processes belonging to a query. Thus, the CPU share consumed by a query remains independent of the number of processes used to implement it.

E. User Interface

For completeness, we briefly mention possible integration of prioritization mechanisms with a user interface. The user interface is not of much concern to the internal implementation but essential to users and administrators.

In the following we assume syntax constructs to express priorities either through extensions to the respective SQL dialect or equivalent mechanisms such as attributes to connectivity components. From a usability point of view it is important that priorities be intuitive and easy to understand, e.g., based on the user's identity. In the Greenplum Parallel Database, we chose to extend existing Workload Management components to accommodate priorities as an additional attribute.

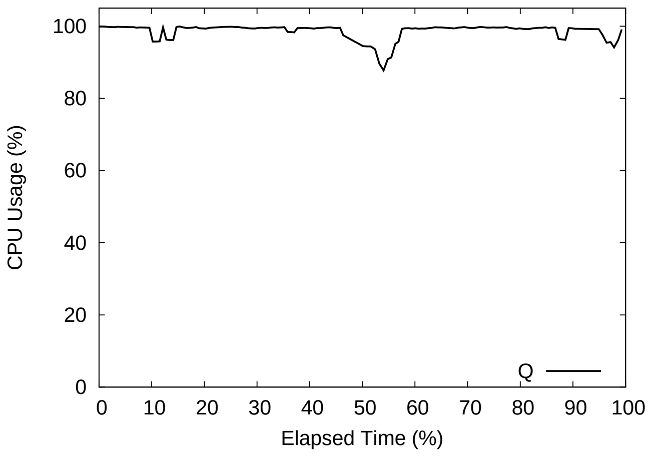
In our experiments we also found it useful to provide special functionality that allows administrators to alter the priority of an already running statement mid-flight. We believe such a mechanism can be beneficial in operational emergencies as well as for tuning purposes.

VI. Discussion

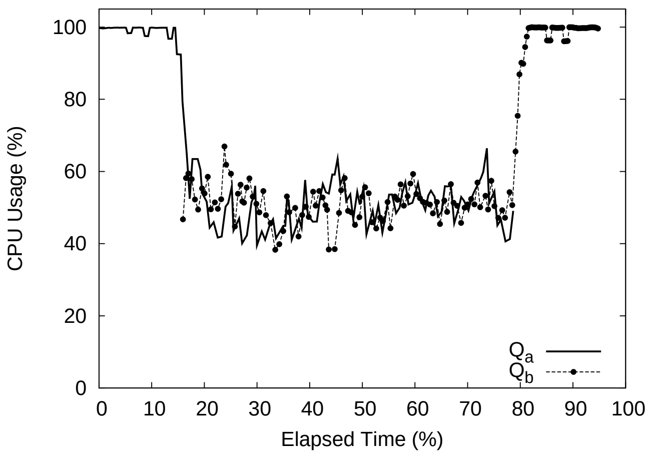
We encountered several notable edge cases while implementing the back-off module, and, while of more esoteric nature, it is important to mention them.

First, consider the case of a severe discrepancy between the priorities of two concurrent queries. The low priority query sleeps for extended periods of time. If the high-priority query suddenly becomes inactive (e.g. due to lock conflicts), the low priority query gets to use extra CPU-time. However, the paused query may have backed off for too long a quantum and CPU-time is wasted as both queries are not reclaiming CPU. We prevent this situation from turning into noticeable under utilization by implementing the back-off in a way that it pauses a low-priority query in small quanta of time rather than the full pre-determined back-off period. This causes the back-off module to check the assigned RCR frequently and pick up any changes in target RCR without delay.

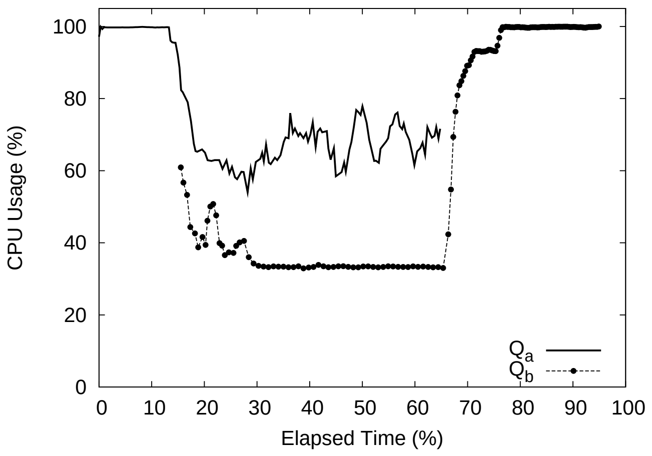
Another challenge we faced is the fact that different operating systems supported different resolutions in their implementation of the system calls used to pause a process as result of backing off. Processes with high target RCR often need to pause for extremely short intervals which maybe below the resolution as supported by the operating system. This may lead to an inadvertent increase in sleep time which in turn may cause a dip in overall CPU usage. By crafting experiments that trigger this phenomenon we were able to investigate this effect in more detail. However, due to the rare occurrence of this constellation we found this problem to be of little impact on higher-level metrics such as response times of entire workloads (see Section VII).



(a) Single query uses 100% of CPU



(b) Two concurrent queries with equal weights, each use about 50% of CPU



(c) Query with weight $w_a = 2$ uses about 66% CPU and query with $w_b = 1$ uses 33% CPU

Fig. 4. CPU Usage Measurements

Finally, our prioritization mechanism can also be used to control the overall CPU utilization of the database system and enable reserving of CPU time for external processes by throttling the overall CPU time available to query processes. This can be beneficial if additional applications such as

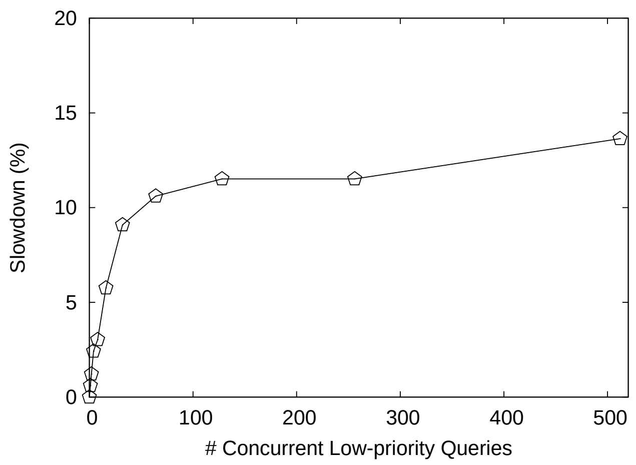


Fig. 5. CEO query executed concurrently with upto 512 low priority queries experiences negligible slowdown

monitoring agents and other maintenance processes need to be executed on the database hosts.

VII. Experimental Evaluation

The experiments in this section were designed to demonstrate that:

- The heuristic backoff mechanism described in this paper achieves proportionate CPU sharing.
- Proportionate CPU sharing may be used to provide soft response-time and throughput guarantees.

In our experiments, we used a SunFire x4540 with 8 virtual processors clocked at 2.3GHz running SunOS 5.10 with a read bandwidth of 999.33 MB/s. On this machine, we configured a 8-segment configuration of Greenplum Parallel Database. We utilized datasets and queries from the TPC-H [3] benchmark of different scales.

The first experiment is designed to test the efficacy of the backoff mechanism using a TPC-H 10 GB dataset and Query 1 (Q) from the benchmark. First, a single instance of Q was executed and the CPU usage of the database executor processes was measured using the `getrusage()` system call. Figure 4(a) shows that this query saturates the CPU. Next, we issued the one instance of the same query (Q_a) against the database and issued another instance (Q_b) after the first instance began execution and measured CPU usages in a similar fashion. In this run, the weights of the two queries were identical ($w_a = w_b = 1$). Figure 4(b) shows that both queries get approximately 50% of CPU in the overlapping interval. Next, we repeated the experiment with $w_a = 2$ and $w_b = 1$ to simulate the example from Section III and plotted the CPU usage in Figure 4(c). The results show that the achieved CPU usage for Q_a hovers at 66% within a small error margin and Q_b achieves 33% as the example suggests. The pertubation in the CPU usage of Q_a is an artifact of the resolution of the sleep system call and it does not adversely affect higher-level metrics such as response time or throughput adversely.

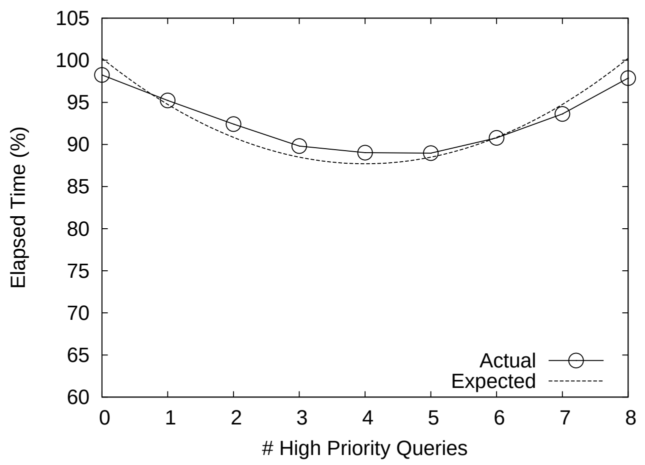


Fig. 6. Average response time of 8 queries varying the number of high priority queries matches expected behavior of Equation 10. The y-axis is normalized with respect to the expected response time when all 8 queries are high priority.

The next experiment is designed to examine the CEO query situation using TPC-H 100 GB dataset. In this situation, a query of extremely high importance (i.e. CEO query) arrives during the execution of a heavy workload. Ideally, the CEO query must see no impact in its response time due to other queries. In this experiment, one CEO query with an extremely large weight of 1,000,000 was executed concurrently with multiple low priority queries with weights of 100. In our experiment, query 1 was used for the CEO query and the low priority queries. Figure 5 shows that the CEO query has a nearly constant response time even in the presence of 512 concurrent low priority queries. This shows how the backoff mechanism may be employed to implement response-time guarantees for extremely important queries/reports.

In the next experiment, we show how the CPU sharing mechanism allows us to predict response times of queries in more complex workloads using analytical methods. Consider a mix of N identical queries with response time t when executed individually against the database. Say M of those queries have a weight of 2 and $N - M$ have a weight of 1. The response time of a high-priority task should be:

$$t_{high} = \frac{N+M}{2} \times t$$

The response time of a low-priority task may be computed as:

$$\begin{aligned} t_{low} &= t_{high} + (t - \frac{t_{high}}{N+M}) \times (N-M) \\ &= N \times t \end{aligned}$$

The average response time for the entire set of queries is:

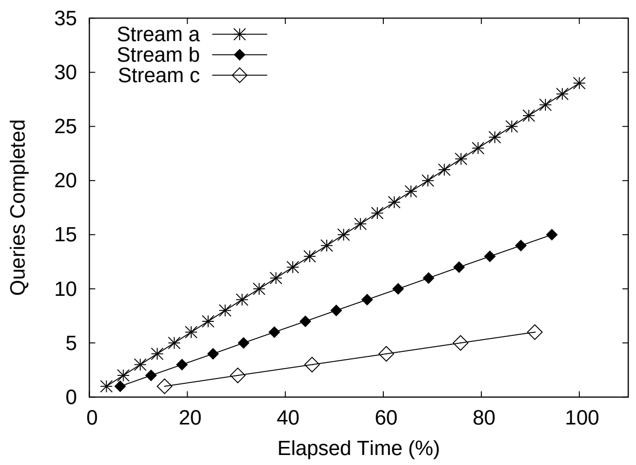


Fig. 7. Three streams with $w_a = 10$, $w_b = 5$ and $w_c = 2$. The throughput of queries on the streams is proportional to the weights.

$$t_{high} \times \frac{M}{N} + t_{low} \times \frac{N-M}{N} \tag{9}$$

$$= \quad t \times (\frac{M^2}{2N} - \frac{M}{2} + N) \tag{10}$$

To emperically verify that the mechanism produces the expected behavior, we fix $N = 8$ and vary M, the number of high priority queries. The dataset used was TPC-H 30 GB and the Query 1 was used as the high-priority and low-priority query. Figure 6 plots the measured mean response times and the expected value per the analytical formula from Equation 10. The system is able to match the expected behavior in this mixed workload situation.

Some database applications like BI and ETL tools open a persistent connection to the database and issue queries serially and it may be necessary to enforce stream-level throughput performance for these applications. The final experiment is designed to show how this may be accomplished. The test involves three streams of TPC-H Query 4 against a TPC-H 10 GB dataset with queries weighed in the ratio 10:5:2. Figure 7 plots the number of queries completed on each stream over time and it indicates that that the three streams experience throughputs in proportion to the weights. This shows how higher-level throughput requirements may be satisfied using the approach described in this paper.

VIII. Related Work

Some databases have implemented more centralized CPU management schemes. For example, Oracle's Resource Manager [15] determines what set of tasks (equivalent to database processes) should be active at a point in time and presents these to the operating system, thus removing all scheduling options from the operating system. A similar centralized approach is followed by Real-time Database Systems [10]. This makes the CPU scheduler a complex piece of software that must understand notions of transactions and locks. Our backoff approach is more de-centralized, but relies on a multi-tasking OS to switch between database processes when they yield.

The problem of fair resource sharing occurs in bandwidth allocation in wireless networks [6], [9]. Typically, these works assume a centralized protocol that coordinates access to the wireless medium. Vaidya et al [19] take an approach which is similar to ours but deals with a much simpler scenario. In their case there is exactly one resource (wireless medium) that is to be shared. We exploit the presence of shared state to communicate the fair share of the resource (CPU) and address the problem of multiple resources.

Powley et al [14] employ constant throttling to achieve DBMS workload control. Our solution adapts to more complex scenarios using feedback control. Parekh et al [12] also employ feedback mechanism to control the impact of database utilities during production workloads. Our solution includes a CPU share calculation algorithm and is applicable to database processes involved in query execution.

Many databases, including Greenplum [1], provide admission control as a way to manage workloads. Teradata's Active System Management [7] and IBM DB2 Query Patroller [4] provide mechanisms to classify and control workloads based on multiple parameters. Schroeder et al [16] deal with a related problem of QoS at the level of transactions. Admission control determines what set of queries may execute concurrently while the work in this paper tackles the problem of how they must behave when executing concurrently.

IX. Conclusion

In this paper, we studied the problem of dynamic prioritization of queries. The solution presented is based on the periodic assessment of a query's resource consumption rate and automatic adjustment towards the ideal rate using control theory principles. Our approach achieves near-optimal results and is distinguished by its robustness, responsiveness to highly dynamic query workloads, and negligible overhead. We implemented this mechanism in the Greenplum Parallel Database, a full-featured parallel and distributed database system.

We focused on CPU as the most critical resource. However, the principles of our approach are not limited to any specific type of resource. The computation of ideal resource consumption rates extends easily to any resource for which there is a programmatic interface to assess a query process' current rate efficiently. Resource-specific mechanisms for adjusting the rate are needed though and, depending on the type of resource, may not be trivial to implement.

References

[1] Greenplum database administrator's guide. `http://gpn.greenplum.com`.

[2] Teradata database: Performance management release v2r6.2. `http://www.teradataforum.com/teradata_pdf/b035-1097-096a.pdf`.

[3] TPC Benchmark^{TM} A: Standard Specification. In J. Gray, editor, *The Benchmark Handbook*. Morgan Kaufmann, 1993.

[4] DB2 Query Patroller Guide: Installation, Administration and Usage. IBM Corporation, 2003.

[5] D. J. Abadi. Column Stores for Wide and Sparse Data. In *CIDR*, pages 292–297, 2007.

[6] J. C. R. Bennett and H. Zhang. WF^{2}Q: Worst-Case Fair Weighted Fair Queueing. In *INFOCOM*, pages 120–128, 1996.

[7] D. P. Broan, A. Richards, R. Zeehandelaar, and D. Galeazzi. Teradata active system management. `http://www.teradata.dk/t/assets/0/206/276/a33a0366-56ab-4e46-b2a6-b16225a3be82.pdf`.

[8] D. Chamberlin et al. A History and Evaluation of System R. *Communications of the ACM*, 24(10), Oct 1981.

[9] A. Demers, S. Keshav, and S. Shenker. Analysis and Simulation of a Fair Queueing Algorithm. *ACM SIGCOMM 89*, 19(4):2–12, August 19-22, 1989.

[10] B. Kao and H. Garcia-Molina. An Overview of Real-Time Database Systems. In *Proceedings of NATO Advanced Study Institute of Real-Time Computing*. Springer-Verlag, May 20 1994.

[11] S.-W. Lee, B. Moon, C. Park, J.-M. Kim, and S.-W. Kim. A Case for Flash Memory SSD in Enterprise Database applications. In J. T.-L. Wang, editor, *SIGMOD Conference*, pages 1075–1086. ACM, 2008.

[12] S. S. Parekh, K. Rose, J. L. Hellerstein, S. Lightstone, M. Huras, and V. Chang. Managing the Performance Impact of Administrative Utilities. In M. Brunner and A. Keller, editors, *DSOM*, volume 2867 of *Lecture Notes in Computer Science*, pages 130–142. Springer, 2003.

[13] D. A. Patterson, G. Gibson, and R. H. Katz. A Case for Redundant Arrays of Inexpensive Disks (RAID). In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 1988.

[14] W. Powley, P. Martin, and P. Bird. DBMS workload control using throttling: Experimental insights. In *Proceedings of the Center for Advanced Studies on Collaborative Research*, pages 1–13, New York, NY, USA, 2008. ACM.

[15] S. C. Resources, A. Rhee, S. Chatterjee, and T. Lahiri. The Oracle Database Resource Manager. In *High Performance Transaction Systems*, 2001.

[16] B. Schroeder, M. Harchol-Balter, A. Iyengar, and E. M. Nahum. Achieving Class-Based QoS for Transactional Workloads. In L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, editors, *ICDE*, page 153. IEEE Computer Society, 2006.

[17] D. F. Shanno and R. L. Weil. 'Linear' Programming with Absolute-Value Functionals. 19:120–124, Jan 1971.

[18] Spielman and Teng. Smoothed Analysis of Algorithms: Why the Simplex Algorithm Usually Takes Polynomial Time. *JACM: Journal of the ACM*, 51, 2004.

[19] N. H. Vaidya, P. Bahl, and S. Gupta. Distributed Fair Scheduling in a Wireless LAN. In *MOBICOM*, pages 167–178, 2000.

[20] T. Westmann, D. Kossmann, S. Helmer, and G. Moerkotte. The Implementation and Performance of Compressed Databases. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 29(3):55–67, Sept. 2000.

[21] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Super-Scalar RAM-CPU Cache Compression. In L. Liu, A. Reuter, K.-Y. Whang, and J. Zhang, editors, *ICDE*, page 59. IEEE Computer Society, 2006.

[22] M. Zukowski, S. Héman, N. Nes, and P. A. Boncz. Cooperative Scans: Dynamic Bandwidth Sharing in a DBMS. In *VLDB*, pages 723–734, 2007.