An approach to QoS Aware Resource Scheduling in Data Stream Systems

Sumit Ganguly
Indian Institute of Technology, Kanpur
Kanpur, INDIA 208016
sganguly@cse.iitk.ac.in

Pushpraj Shukla*
University of Texas at Austin
Austin, TX 78712
pushpraj@cs.utexas.edu

ABSTRACT

In a data stream management system (*DSMS*), there are a large number of simultaneously executing queries, each of which performs a specific monitoring function on some subset of the streaming data. In order to keep up with the rapid arrival rates, a *DSMS* may sample or drop tuples to match execution rate with the arrival rate. However, as the sampling rate reduces, so does the *quality of answers* of the queries. Since, there may be substantial overlap between the queries that are simultaneously active in the system, an optimal choice of the points and extent of load-shedding cannot be obtained without taking a "global view" of this problem. In this paper, we formulate the problem of optimizing the rates of sampling with the the overall objective of minimizing the sum of the loss of quality of answers of the queries, assumed to be associated with each query (e.g., as in the Aurora system).

Specifically, we consider the problem of obtaining the optimal sampling probabilities for a given collection of sliding-window queries with known loss of quality plots, so that the objective function of the sum of loss of quality across the queries is minimized. The problem can be generally posed and solved as convex optimization problems. However, we present a significantly more efficient $O(n^2)$ time algorithm for finding the optimal sampling rates, where, n is the number of queries registered over the stream.

1. INTRODUCTION

The *streaming data model* [3, 2, 5, 1, 8, 7, 6] was proposed as a model for a class of monitoring applications, such as network management, RF-id based applications, sensor networks, etc. These applications are characterized by high volumes of rapidly and continuously arriving records in the form of data streams. A typical scenario in a data stream management system (*DSMS*) consists of a multitude of simultaneously executing queries, each of which performs a specific monitoring function on the subset of the streaming data. A large class of such queries involves computing query-specific functions over a sliding window range (which may be different for different queries). A well-recognized problem in this scenario is that if data arrives at a rapid rate, then it may not be always possible to complete processing a stream tuple before the next tuple arrives. The typical solution approach is to *sample* tuples (also called *load shedding*) in order to match the overall processing rate with the arrival rate. However, it is clear that as the rate of sampling decreases, the *quality of answer* of the queries reduces. In this paper, we study the problem of optimizing the rates of sampling with the the overall objective of maximizing the sum of the quality of answers of the queries.

*Work done while at IIT Kanpur

The AURORA stream management system [5] associated a quality of service function with each query. The quality of service function [5] can be thought of as a quantification of the relevance or the accuracy of the answer given by a query to the application, as a function of the sampling rate. Further, different queries may be assigned different priorities; for example, queries that monitor critical metrics in the system may be given a higher priority over other more mundane queries. As tuples are dropped to match the system processing rate to the data arrival rates, it is more reasonable to tolerate a greater loss of quality for the low priority queries than for the higher priority queries. We therefore extend the AURORA viewpoint in this paper and consider the problem of optimizing the sampling probabilities (or, in other words, how should the load be shed) so as to *minimize a global objective function defined as the sum of loss of quality functions*. Example 1 presents an example of loss of quality function for a typical scenario.

Example 1. Sampling based loss of quality functions. Suppose that a query uses a standard sampling algorithm to track the frequent items in the stream. If each item is sampled with a probability of x, then, by standard Chernoff's bounds, the frequency f_i of an item i is estimated by an additive error whose absolute value is $O(\sqrt{\frac{f_i}{x}})$ (with high probability). The loss of quality of the answer may be measured as the error in the answer, which is of the form $\frac{c}{\sqrt{x_i}}$, where, c is some constant multiplied by $\sqrt{f_i}$. Such functions, which quantify the drop in the the quality are referred to as the *loss of quality* function. □

1.1 System Model and Scope of Work

In this paper, we limit our attention to a basic, yet interesting class of queries, namely, the *class of single stream queries that are specified over sliding windows over time*. We consider a set of sliding window queries $Q_1, Q_2, \ldots, Q_n$ over a single stream, where the query Q_i is defined over a sliding window that is specified as the set of tuples arriving over the stream in the time interval

$[\text{NOW} - s_i - w_i + 1, \text{NOW} - s_i]$. In this notation, NOW represents the current time, s_i is the shift in time where the most recent tuple in the query window may be allowed to lie, and w_i is the length of the window. For example, the sliding window $[\text{NOW}-19, \text{NOW}-10]$ is a sliding window of length 10 and contains all tuples in the stream that have arrived between 10 and 19 time units from NOW.

Constraints

Several constraints arise naturally in the context of optimizing for the loss of quality functions of queries defined over sliding windows over streams. We identify three main constraints, namely, the *monotonicity constraint*, the *space constraint* and the *rate constraint*. These constraints are derived and explained in Section 2. Here, we give a brief overview of these constraints. In the remainder of the paper, we assume without any loss of generality, that the queries are indexed in non-descending order of $s_i + w_i$, that is, $s_1 + w_1 \leq s_2 + w_2 \leq \ldots \leq s_n + w_n$.

The sampling monotonicity constraint arises as follows. Suppose that there are two queries Q_i and Q_j defined over sliding windows $[\text{NOW} - a_i, \text{NOW} - b_i]$ and $[\text{NOW} - a_j, \text{NOW} - b_j]$ respectively. Suppose that $a_i > a_j$, that is, the window for Q_i expires later than the window for Q_j. Suppose that the sampling probability assigned to Q_i is x_i and the sampling probability assigned to Q_j is x_j, $0 \leq x_i, x_j \leq 1$. It follows that in the entire window $[\text{NOW} - a_i, \text{NOW}]$, the sampling probability cannot be strictly smaller than x_i. That is, every tuple that is relevant to Q_j is sampled with a probability that is at least x_j. This gives the constraint $x_i \leq x_j$. The conjunction of such constraints for all pairs of distinct queries Q_i and Q_j is called the *sampling monotonicity constraint*.

The *space constraint* is arrived at as follows. If the sampling probability of query Q_i is x_i, and if m_i tuples arrive in its window, then, on expectation, the number of tuples that must be stored is $m_i \cdot x_i$. The total expected space consumed in the multi-query processing is the sum of the expected space consumption for each of the queries, while accounting for overlaps among the windows of queries. The space constraint sets the total expected space consumption to be upper bounded by the system memory budgeted for multi-query processing (a normalization factor may be added for "padding" and only changes the constants). The constraint so obtained is called the *space constraint*.

The *rate constraint* can be motivated as follows. Assume that it takes an estimated time t_i to process each stream item using the data structures of Q_i. Using the window definitions and the sampling probabilities, and accounting for overlaps in window definitions and any sharing of computation between queries, we formulate an expression for the expected time required to process the tuples arriving in the window of each query Q_i. The expected time for processing the tuples for query Q_i must be smaller than the window size of Q_i, for each of the queries $Q_1, Q_2, \ldots, Q_n$. We thus get n constraints, which are collectively referred to as the rate constraint. The general optimization problem can be expressed as follows.

Problem OPT. *Minimize* the *sum of the loss of quality LoQ functions of* $Q_1, Q_2, \ldots, Q_n$ *while satisfying* the *Monotonicity* Constraint, the *Space* Constraint and the *Rate* Constraint.

Loss of Quality Functions.

It seems reasonable to assume that the loss of quality does not decrease as more resources are allocated to the query, that is, the functions $LoQ_i(x)$ are monotonically non-decreasing functions of x. However, despite this, the optimization problem remains too complex, and requires further assumptions before it can be solved satisfactorily. We make the assumption that the loss of quality functions are convex and decreasing functions in the interval $[0, 1]$. A large class of "natural" loss of quality functions fall in this category, for example, loss of quality functions arising out of sampling, as illustrated in Example 1.

Contributions

We present a general framework for solving the problem OPTas a convex optimization problem. The assumption that *LoQ* functions are convex make the problem OPT an instance of a convex optimization problem. Such problems can be solved using standard techniques such as the interior point methods [**?**]. This observation allows us to leverage powerful optimization techniques from the literature, and also to use extensions developed for online versions of such problems [4].

In the quest for more efficient algorithms, we then restrict the class of *LoQ* functions to *homogeneous* functions, namely, that $LoQ_i(x) = a_i f(x)$, for queries indexed from 1 to n. That is, the loss of quality functions are similar in nature across the queries Q_i, and differ from each other in terms of the constant a_i associated with query Q_i. The constant a_i corresponding to query Q_i is a parameter and is chosen depending on the priority of the query, etc.. We present an efficient $O(n^2)$ time algorithm for solving problem OPT under the assumption that the *LoQ* functions are homogeneous, where, n is the number of queries.

1.2 Organization.

The remainder of the paper is organized as follows. We present the problem formulation and derive the constraints in Section 2. In Section 3, we present our algorithm and its analysis. Finally, we conclude in Section 4.

2. PROBLEM FORMULATION AND GENERAL SOLUTION

We are given a collection of queries $Q_1, \ldots, Q_n$ monitoring the same stream, where, Q_i operates on a sliding window of size w_i time units starting from s_i time units from the current time NOW . That is, the sliding window over which Q_i operates is the interval $[\text{NOW} - s_i - w_i + 1, \text{NOW} - s_i]$. Without loss of generality, we assume that the queries are ordered in ascending order of the expiry time of the window, $s_i + w_i$. Thus, $s_1 + w_1 \leq s_2 + w_2 \leq \ldots \leq s_n + w_n$. There are three natural constraints that emerge in this setting that are referred to as the *sampling monotonicity constraint*, the *space constraint* and the *rate constraint*. We discuss these constraints below.

2.1 Problem Constraints

Monotonicity Constraint.

Consider the following example. Suppose there are two queries Q_1 and Q_2, where, Q_1 operates over the sliding window $[\text{NOW} - 5, \text{NOW}]$ of length 6 $(= 5 + 1)$ and Q_2 operates over the sliding window $[\text{NOW}-10, \text{NOW}-3]$ of length 8 $(= 10-3+1)$. Suppose that tuples of Q_1 are sampled at a rate of $x_1 = 0.8$. This means that the effective sampling rate at the time instant $\text{NOW} - 5$ is $x_1 = 0.8$. Thus, after time instant $\text{NOW} - 5$, there is no possibility of increasing the sampling rate, since, tuples that have been sampled and rejected are lost. Thus, the sampling rate for Q_2 is at most as large as the sampling rate of Q_1, or that, $x_2 \leq x_1$. The argument must hold irrespective of the relative priorities of Q_1 and Q_2. In this manner, we obtain the following monotonicity constraint.

Monotonicity Constraint: $$0 \leq x_n \leq x_{n-1} \leq \cdots \leq x_2 \leq x_1 \leq 1 \quad (1)$$

Space Constraint.

Suppose that the queries are ordered in ascending order of their window expiry times, and let x_i be the sampling probability for query Q_i. For $1 \leq i \leq n$, let m_i denote the number of tuples that have arrived in the window corresponding to Q_i, that is, the tuples that have arrived between time $\text{NOW} - s_i - w_i + 1$ and $\text{NOW} - s_i$). The expected number of tuples sampled for Q_1 is $m_1 x_1$. For tracking the query Q_2, each of the tuples belonging to the sample for Q_1 is further sub-sampled at a rate of $\frac{x_2}{x_1}$, resulting in an effective sampling rate of $x_1 \frac{x_2}{x_1} = x_2$. The additional storage required for processing Q_2, is on expectation, $(m_2 - m_1)x_2$ tuples. Continuing in this manner, we obtain that the expected number of tuples stored is $m_1 x_1 + \sum_{i=2}^{n}(m_i - m_{i-1})x_i$. Let b be the total space available for processing these queries. Define $u_1 = m_1$ and $u_i = m_i - m_{i-1}$, for $2 \leq i \leq n$. The space constraint equation states that the total expected number of tuples does not exceed b.

Space Constraint: $$\sum_{i=1}^{n} u_i x_i \leq b \quad (2)$$

Rate Constraint for Steady State Computation.

Let t_i be the estimated time required to process a tuple for the query Q_i, for $i = 1, 2, \ldots, n$. The expected time required for processing the tuples that arrive in the first window of size w_1 across all the queries is $m_1 x_1 t_1 + m_1 x_2 t_2 + \ldots + m_1 x_n t_n$. For steady state, this processing must be completed within time w_1, that is, $m_1 x_1 t_1 + m_1 x_2 t_2 + \ldots + m_1 x_n t_n \leq w_1$. In the same manner, we can argue that the expected time to process the tuples in the second window w_2 is $m_2 x_1 t_1 + m_2 x_2 t_2 + m_2 x_3 t_3 + \ldots + m_2 x_n t_n$ and this processing must be completed within time w_2. We therefore obtain the following family of constraints: $m_i \sum_{k=1}^{n} x_k t_k \leq w_i, \ 1 \leq i \leq n$. This can be equivalently written as follows.

Rate Constraint: $$\sum_{k=1}^{n} x_k t_k \leq \min_{i=1}^{n} \frac{w_i}{m_i} \quad (3)$$

The optimization problem is to minimize the sum of the loss of quality functions for each of the queries, subject to satisfying the monotonicity constraint, the space constraint and the rate constraint, given by equations (1)— (3).

$$\begin{aligned}
&\text{Problem OPT}: \\
&\text{Minimize} \quad \sum_{j=1}^{n} LoQ_j(x_j) \\
&\text{subject to} \quad \sum_{j=1}^{n} u_j x_j \leq b, \text{ and } \quad \sum_{k=1}^{n} x_k t_k \leq \min_{i=1}^{n} \frac{w_i}{m_i} \quad (4) \\
&\text{and } 0 \leq x_n \leq x_{n-1} \leq \cdots \leq x_2 \leq x_1 \leq 1
\end{aligned}$$

General Solution

The problem defined by (4) has a feasible space defined by linear constraints and whose objective function is a convex function. Therefore, the problem falls in the very well-studied class of convex optimization problems. The interior point method [4] is particularly suited for this problem since the feasible space is a polytope, and yields a polynomial time solutions to the problem at hand. Moreover, this view of the problem also allows the use of work in the area of convex optimization for online problems, that is, when constraints are either added to or removed from the set, and, when changes to the objective function are local. The online version of this problem is of particular interest to the stream scheduling problem, since, new queries may join the system at any time and old queries may de-register themselves.

In the rest of the paper, we explore special purpose algorithms to solve special cases of the problems in (4).

3. ALGORITHM FOR HOMOGENEOUS LOQ FUNCTIONS

In this section, we consider the problem of optimizing for the sum of homogeneous loss of quality functions while considering the monotonicity constraint and the space constraint. The same approach can be used to solve the problem for optimizing for the net loss of quality while satisfying the monotonicity and the rate constraint (i.e., ignoring the space constraint). The optimization problem can be expressed as follows.

$$\begin{aligned}
\text{Min} \quad & \sum_{j=1}^{n} a_j f(x_j) \\
\text{subject to} \quad & \sum_{j=1}^{n} u_j \cdot x_j \leq b \\
& \text{and } 0 \leq x_n \leq \cdots \leq x_2 \leq x_1 \leq 1 \quad (5)
\end{aligned}$$

Here, $f(x)$ is assumed to be a convex, decreasing and twice differentiable function, such that $a_j > 0$ and $u_j > 0$ for $j = 1, 2, \ldots, m$. If $\sum_{j=1}^{n} u_j \leq b$, then, the problem has a trivial optimal solution, namely, $x_i = 1$, for $i = 1, 2, \ldots, m$. Therefore, in the remainder of this section, we assume that $\sum_{j=1}^{n} u_j > b$. The algorithm works in two sequential phases, namely, the *Aggregation* phase, presented in Section 3.1 and the *Optimization* phase, presented in Section 3.2.

3.1 Phase I: Aggregation Phase

In this section, we describe the first phase of the two-phase optimization algorithm, called the aggregation phase. It is assumed that the queries are numbered in non-increasing order of the expiry times $s_i + w_i$ of their windows.

Data Structure and Algorithm

The pseudo-code for the aggregation-phase algorithm is given in Figure 1. The data structure is an array $A[1, \ldots, n]$ of records, such that, record i corresponds to query Q_i. Each record $A[i]$ is a structure that consists of the following fields. These fields are designed to support the structure of *aggregated* nodes, where, each aggregated node is a contiguous sequence of indices of A.

1. The fields *next* and *prev* refer to the next node and the previous node that are members of the same aggregate node. This easily allows the sequence of nodes that form an aggregate node to be maintained as a *doubly linked circular list*. For each i such that $1 \leq i \leq n$, $A[i]$.*next* and $A[i]$.*prev* are both initialized to i. Each aggregate node is referred to by the index of the node with the lowest index among the nodes in the aggregate.

2. The fields *nextaggnode* and *prevaggnode*, refer to the previous aggregate node and the next aggregate node, respectively, of the current node. The sequence of aggregated nodes are maintained as a *doubly linked list*. $A[i]$.*nextaggnode* is initialized to $i+1$, and $A[i]$.*prevaggnode* is initialized to $i-1$, for $1 \leq i \leq n$.

3. The fields a and u are the cost coefficients, that is, corresponding to the index i, $A[i].a$ and $A[i].u$ refer to a_i and u_i in (5).

The procedure *Aggregate* , shown in pseudo-code in The algorithm is designed using the divide and conquer paradigm. For simplicity, the procedural details of correctly setting the pointers after a pair of nodes are merged, is encapsulated in the procedure *Merge*.

The procedure *Aggregate* works as follows. The procedure returns A without change if *lindex* is the same as *rindex*. Otherwise, it divides the array into two (about) equal parts and makes recursive calls on both halves. The result of the recursive call is that each of the halves is returned as a sequence of aggregated nodes in descending order of their a to u ratio. The remainder of the procedure checks if the a to u ratio of the last aggregate node of the first half is at most the ratio of the first aggregate node of the second half. If it is, then the nodes are merged, and this merging procedure is continued to the right, until, either all nodes are merged or the condition of aggregation fails. In this case, the nodes are checked for aggregating to the left of the last aggregated node. The iterative process terminates when nodes cannot be aggregated any further.

The algorithm repeatedly aggregates adjacent nodes indexed i and $i+1$ such that $\frac{a_i}{c_i} \leq \frac{a_{i+1}}{u_{i+1}}$. Once a pair of nodes are aggregated, the new a and c fields for the aggregated node is set as follows: $a' = a_i + a_{i+1}$ and $u' = u_i + u_{i+1}$. The process is continued until the sequence of aggregated nodes have their a to u ratio in strictly descending order.

Proof of Correctness

Lemma 1 basically shows the correctness of the aggregation procedure.

LEMMA 1. *Suppose* $\frac{a_{j+1}}{u_{j+1}} \leq \frac{a_j}{u_j}$. *Then, there exists an optimal solution* $x_1, x_2, \ldots, x_n$ *such that* $x_j = x_{j+1}$.

PROOF. Let x be a feasible solution vector for (5) and let x' be a feasible solution vector that is obtained by perturbing j^{th} and $(j+1)^{st}$ coordinates of x and leaving the other coordinates of x unchanged. That is, $x'_j = x_j + dx_j$ and $x'_{j+1} = x_{j+1} + dx_{j+1}$, where, dx_j and dx_{j+1} are infinitesimal changes, and for all indices except for the indices j and $j+1$, $x'_i = x_i$. Since, x' is a feasible solution, therefore, $\sum_{i=1}^{n} u_i \cdot x_i = \sum_{i=1}^{n} u_i \cdot x'_i$, or equivalently, that,

$$u_j \cdot dx_j + u_{j+1} \cdot dx_{j+1} = 0 \tag{6}$$

procedure Merge(A, l, r)
Description. Takes as input two contiguous aggregated nodes rooted at indices l and r of the array A, such that $l < r$, and merges them in place into a single aggregated node.

```
begin
    tempright := A[r].prev
    templeft := A[l]. prev
    A[templeft].next := r
    A[tempright]. next := l
    A[r]. prev := templeft
    A[l]. prev := tempright
    A[l]. a := A[l].a + A[r].a
    A[l].u := A[l].u + A[r].u
    A[l]. nextaggnode := A[r]. nextaggnode
end.
```

procedure Aggregate (*A*,*lindex*, *rindex*)
Description. A divide and conquer based procedure that merges nodes of A whose indices lie in the range [*lindex*,*rindex*], until the aggregated nodes are arranged in descending order of a_i/u_i.

```
begin
    if (lindex = rindex) then return
    mid := (lindex + rindex)/2
    Aggregate ( lindex,mid)
    Aggregate (mid+1,rindex)
    l := A[mid].prev
    r := mid+1
    state := RIGHT
    MERGE:      if (A[l].a/A[l].u ≤ A[r].a/A[r].u) then
                  begin
                      state:=RIGHT
                      Merge(A, l, r)
                      r := A[l]. nextaggnode
                      if (r > rindex) then goto MOVELEFT
                  end
                else if (state = LEFT) then goto TERMINATE
                    else goto MOVELEFT
    MOVELEFT:   r := l
                l := A[l]. prevaggnode;
                if (l < lindex) then goto TERMINATE
                else  begin
                          state := LEFT
                          goto MERGE
                      end
    TERMINATE:  return
end.
```

Figure 1: Phase I algorithm for recursively aggregating nodes

The change in the objective function can be calculated as follows.

$$
\begin{aligned}
\Delta f = f(x') - f(x) &= \sum_{i=1}^{n} a_i \cdot f(x_i') - \sum_{i=1}^{n} a_i \cdot f(x_i) \\
&= a_j \cdot f'(x_j) \cdot dx_j + a_{j+1} \cdot f'(x_{j+1}) \cdot dx_{j+1} \\
&= u_j dx_j \left(\frac{a_j}{u_j} \cdot f'(x_j) - \frac{a_{j+1}}{u_{j+1}} \cdot f'(x_{j+1}) \right) \qquad \text{by (6)} \\
&= a_j f'(x_j) d(x_j) \left(1 - \frac{\frac{a_{j+1}}{u_{j+1}}}{\frac{a_j}{u_j}} \frac{f'(x_{j+1})}{f'(x_j)} \right) \\
&\leq 0
\end{aligned}
$$

The last step can be inferred as follows. It is given that $\frac{a_{j+1}}{u_{j+1}} \leq \frac{a_j}{u_j}$. Therefore, $\frac{\frac{a_{j+1}}{u_{j+1}}}{\frac{a_j}{u_j}} \leq 1$. Further, since, f is convex, $f''(x) > 0$, and since, $x_{j+1} \leq x_j$, it follows that $f'(x_{j+1}) \leq f'(x_j)$. Thus, $\frac{\frac{a_{j+1}}{u_{j+1}}}{\frac{a_j}{u_j}} \frac{f'(x_{j+1})}{f'(x_j)} \leq 1$, or that, $\left(1 - \frac{\frac{a_{j+1}}{u_{j+1}}}{\frac{a_j}{u_j}} \frac{f'(x_{j+1})}{f'(x_j)}\right) \geq 0$. Therefore, if $dx_j < 0$, then, $\Delta f < 0$. $a_j > 0$, $f'(x_j) \leq 0$ (since, f is a decreasing function). Thus,

a_j and $f'(x_j)$ are positive (since, f is a and dx_j is assumed to be positive. Thus, the sign of Δf is the same as the sign of $f'(x_j)$ which is non-positive, since, f is a loss of quality function which is a non-increasing function of x. □

Lemma 1 directly implies the correctness of the procedure *Aggregate* as follows. Suppose nodes with indices j and $j+1$ satisfy the condition that $\frac{a_j}{u_j} \geq \frac{a_{j+1}}{u_{j+1}}$. Then, by Lemma 1 there exists an optimal solution such that $x_{j+1} = x_j$. Therefore, in this solution, the aggregated node corresponding to the set $\{j, j+1\}$ can be thought of as having an a value of $a_j + a_{j+1}$ and a u value corresponding to $u_j + u_{j+1}$. It may be noted that the result of the aggregation phase is unique and independent of the order of aggregation.

Time Complexity.

We now show that the time complexity of the procedure *Aggregate* $(A, 1, n)$ is $O(n)$. The time complexity of procedure *Aggregate*, for the initial call *Aggregate*$(A, 1, n)$ is determined by the total number of executions y of the *if* statement in line number 9 of the procedure in Figure 1 for the following reasons. This particular statement is executed at least once for every recursive call that does not exit from line numbered 2. Therefore, the number of times the program exits at line numbered 2 is at most $\max(2y, 1) < 2y+1 = O(y)$, since in a balanced binary tree, the number of leaves is either 1 or is at most twice the number of the nodes at a level just above the leaves. Further, for each such exit processing, only a constant number of instructions are executed.

Every run of procedure *Aggregate* starting at the *if* statement at line numbered 9 either terminates at statement numbered 25 or returns to the *if* statement at line 9. Each such run executes a number of statements that is bounded above by a constant (since, procedure *Merge* is an $O(1)$ procedure). It therefore follows that in every recursive call to procedure *Aggregate* , the time complexity of the "conquer" phase requires time that is bounded by some constant factor of the number of times the *if* statement at line 9 is executed in that call. The total time complexity is the sum of the times taken by all the "conquer" phases and all the "divide" phases. By the above arguments, the sum of the time required by the conquer phases is bounded by a constant times y. Since, each division phase takes constant time per call, the total time taken by the divide phase for the call *Aggregate* $(A, 1, n)$ is $O(n)$.

Consider the forest of binary trees where, each index i is a leaf node, for $1 \leq i \leq n$, and each internal node is an aggregated node, whose children are the two nodes that are aggregated as per the algorithm in Figure 1. Then, each call to the procedure *Merge* can be identified with a unique internal node. Let z denote the number of internal nodes of the forest, or equivalently, the number of times the *Merge* procedure is called. Then, $z = n - 1$, since the number of internal nodes in a forest of binary trees is one less than the number of leaf nodes. In each recursive call to *Aggregate* (A, *lindex*, *rindex*) (i.e., when *lindex* < *rindex*), the number of times the statement at line 9 is executed is at most two more than the number of calls to the *Merge* procedure. Therefore, $y \leq z + 2r$, where, r is the total number of recursive calls made. By a similar argument, it can be shown directly that, $r = n - 1$. Therefore,

$$y \leq z + 2 \cdot r = n - 1 + 2 \cdot (n - 1) = O(n) \ .$$

Therefore, the time complexity of procedure *Aggregate* is the sum of the times taken by the division phase and the conquering phase, which is $O(n)$.

3.2 Phase II: Optimization Phase

We now present the second phase of the two-phase optimization technique. Suppose that at the end of the aggregation phase, there are $m \leq n$ aggregate nodes, numbered from 1 to m, where, the a and u values of the aggregate node numbered i is denoted by a_i' and u_i' respectively. Therefore, $\frac{a_i'}{u_i'} < \frac{a_{i+1}'}{u_{i+1}'}$, for $1 \leq i \leq m-1$. The optimization problem can be written as follows.

$$
\begin{aligned}
&\text{Minimize } \sum_{i=1}^{m} a_i' \cdot f(x_i) \\
&\text{subject to } \sum_{i=1}^{m} u_i' \cdot x_i = b \\
&\qquad \text{and } 0 \leq x_m \leq x_{m-1} \leq \cdots \leq x_1 \leq 1 \\
&\text{given that } \frac{a_1'}{u_1'} < \frac{a_2'}{u_2'} < \cdots < \frac{a_m'}{u_m'} \ . \qquad (7)
\end{aligned}
$$

Corresponding to each feasible solution to the problem (7) (and therefore, corresponding to an optimal solution), we can identify two indices s and t between 1 and m, where, $s \leq t$, such that s is the smallest index such that $x_s < 1$ and t is the largest index such that $x_t > 0$. The optimization problem (7) for this class of feasible solutions can be equivalently written as follows, and is denoted as $R_{s,t}$.

$$
\begin{aligned}
R_{s,t}: \qquad &\text{Minimize } \sum_{i=s}^{t} a_i' \cdot f(x_i) \\
&\text{subject to } \sum_{i=s}^{t} u_i' \cdot x_i = b - \sum_{j=1}^{s-1} u_j' \\
&\qquad \text{and } 0 < x_t \leq x_{t-1} \leq \cdots \leq x_1 < 1 \\
&\text{given that } \frac{a_1'}{u_1'} < \frac{a_2'}{u_2'} < \cdots < \frac{a_m'}{u_m'} \ . \qquad (8)
\end{aligned}
$$

Clearly, the problem $R_{s,t}$ is feasible iff $\sum_{j=1}^{s-1} u_j' < b < \sum_{j=1}^{t} u_j'$. The optimal solution for (7) is the solution that has the smallest objective value among the feasible instances of $R_{s,t}$, for $1 \leq s, t \leq m$.

LEMMA 2. *There exists an optimal solution to a feasible instance of $R_{s,t}$ such that $x_s = x_t$.*

PROOF. Let x be a feasible solution to $R_{s,t}$ such that $x_i > x_{i+1}$, for some index i, $s \le i \le t-1$. Let x' be a feasible solution to $R_{s,t}$ obtained by perturbing the values of x_i and x_{i+1} and leaving the values of x_j, $j \notin \{i, i+1\}$ unchanged. That is, $x'_i = x_i + dx_i$ and $x'_{i+1} = x_{i+1} + dx_{i+1}$, where, $dx_i < 0$, and $u'_i dx_i + u'_{i+1} dx_{i+1} = 0$. Then,

$$\begin{aligned}\Delta f &= f(x') - f(x) \\ &= a'_i f'(x_i) dx_i + a'_{i+1} f'(x_{i+1}) dx_{i+1} \\ &= a'_i f'(x_i) dx_i - a'_{i+1} f'(x_{i+1}) \frac{u'_i}{u'_{i+1}} dx_i \\ &= u'_i dx_i \left(\frac{a'_i}{u'_i} f'(x_i) - \frac{a'_{i+1}}{u'_{i+1}} f'(x_{i+1}) \right) \\ &= u'_i f'(x_i) dx_i \left(\frac{a'_i}{u'_i} - \frac{a'_{i+1}}{u'_{i+1}} \frac{f'(x_{i+1})}{f'(x_i)} \right) \qquad (9)\end{aligned}$$

$f'(x_i) = f'(x_{i+1}) + \int_{t=x_{i+1}}^{x_i} f''(t)dt > f'(x_{i+1})$, since f is convex and therefore, $f''(t) > 0$ in $[0, 1]$. Further, since f is a decreasing function of its argument (loss of quality function), $0 > f'(x_i) > f'(x_{i+1})$. This implies that $\frac{f'(x_{i+1})}{f(x_i)} = \frac{|f'(x_{i+1})|}{|f(x_i)|} > 1$. Thus,

$$\frac{a'_i}{u'_i} - \frac{a'_{i+1}}{u'_{i+1}} \frac{f'(x_{i+1})}{f(x_i)} < \frac{a'_i}{u'_i} - \frac{a'_{i+1}}{u'_{i+1}} < 0 \ .$$

Substituting in (9), we have

$$\begin{aligned}\text{sign}(\Delta f) &= \text{sign}\left(u'_i f(x_i) dx_i \left(\frac{a'_i}{u'_i} - \frac{a'_{i+1}}{u'_{i+1}} \frac{f'(x_{i+1})}{f'(x_i)} \right) \right) \\ &= \text{sign}(u'_i)\text{sign}(f'(x_i))\text{sign}(dx_i)\text{sign}\left(\frac{a'_i}{u'_i} - \frac{a'_{i+1}}{u'_{i+1}} \frac{f'(x_{i+1})}{f'(x_i)} \right) \\ &= (1)(-1)\text{sign}(dx_i)(-1) \\ &= \text{sign}(dx_i)\end{aligned}$$

Thus if $dx_i < 0$, then, $\Delta f < 0$. It follows that there exists an optimal solution such that $x_i = x_{i+1}$. Repeating the argument, it follows that there exists an optimal solution such that $x_s = x_{s+1} = \cdots = x_t$. □

Let $A_{s,t} = \sum_{j=s}^{t} a'_j$ and $U_{s,t} = \sum_{j=s}^{t} u'_j$. It follows from Lemma 2 that an optimal solution for $R_{s,t}$ is a value $x_s = x_{s+1} = \ldots = x_t = \alpha_{s,t}$, where,

$$f(\alpha_{s,t}) = \frac{1}{U_{s,t}}(b - U_{1,s-1}) \ . \qquad (10)$$

If $\alpha_{s,t} \le 0$ or $\alpha_{s,t} \ge 1$, then, there is no feasible solution for $R_{s,t}$. If a feasible solution exists, then, the value of the corresponding optimal solution, denoted by $C_{s,t}$, for (7) is given as follows.

$$C_{s,t} = f(1)A_{1,s-1} + f(\alpha_{s,t})A_{s,t} + f(0)A_{t+1,n} \ . \qquad (11)$$

3.3 Algorithm and Time Complexity

An algorithm can now be designed to solve the optimization problem (7) as follows. The optimal value C_{opt} is given by the following equation.

$$C_{\text{opt}} = \min_{t=1}^{n} \min_{s=1}^{t} C_{s,t}$$

Since, $A_{s,t} = A_{s+1,t} + a'_s$ and $U_{s,t} = U_{s+1,t} + u'_s$, $U_{1,s-1} = U_{1,s} - u'_s$, it follows that

$$C_{s,t} = C_{s+1,t} - f(1)a'_s + \frac{1}{U(s,t)}(b - U_{1,s-1})$$

However, if $0 < \frac{1}{U(s,t)}(b - U_{1,s-1}) < 1$, then, the value of $C_{s,t}$ is not considered for minimum computation, although, it is still considered for the recurrence computation.

Thus, the expression $\min_{s=1}^{t} C_{s,t}$ can be computed in time $O(n)$ for each value of t, by first computing $C_{t,t}$ and then computing $C_{t-1,t}, C_{t-2,t}, \ldots, C_{1,t}$ in succession, using the above recurrence relation in time $n \cdot O(1) = O(n)$. The above process can be repeated for each value of t, $1 \le t \le n$, to yield an $O(n^2)$ time complexity algorithm. The space required by the algorithm is $O(n)$, namely, to store the arrays u' and a', respectively.

4. CONCLUSION

The paper presents a preliminary investigation into the problem of resource scheduling in a *DSMS* while taking a "global view" of minimizing the sum of the loss of quality functions. We show that the problem in its generality is a convex optimization problem and therefore can be solved using the interior point method in polynomial time. We also present a very efficient algorithm for a subset of this problem for homogeneous loss of quality functions.

5. REFERENCES

[1] A. Arasu, B. Babcock, S. Babu, J. Cieslewicz, M. Datar, K. Ito, R Motwani, U. Srivastava, and J. Widom. "STREAM: The Stanford Data Stream Management System". In *Data Stream Management Processing , Garofalakis, M.N. and Gehrke, J. and Rastogi, R (Eds.), Springer Verlag, 2006.*

[2] R. Avnur and J. M. Hellerstein. "Eddies: Continuously Adaptive Query Processing". In *Proc. SIGMOD*, 2000.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. "Models and Issues in Data Stream Systems". In *Proc. PODS*, 2002.

[4] Stephen Boyd and Lieven Vandenberghe. *"Convex Optimization"*. Cambridge University Press, 2004.

[5] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. B. Zdonik. "Monitoring Streams - A New Class of Data Management Applications". In *Proc. VLDB*, 2002.

[6] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. R. Madden, V. Raman, F. Reiss, and M. A. Shah. "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World. CIDR 2003.". In *Proc. CIDR*, Available from db.cs.wisc.edu/cidr03/p24.pdf, 2003.

[7] Krishnamurthy, S. Chandrasekaran, C. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Madden, F. Reiss, and M. A. Shah. "telegraphcq: An architectural status report.". *IEEE Data Eng. Bull.*, 2003.

[8] S. R. Madden, J. M. Hellerstein, and W. Hong. *"TinyDB: In-Network Query Processing in TinyOS, Version 0.4"*. Available from http://telegraph.cs.berkeley.edu/tinydb/doc/index.html, September 2003.