Real-Time, Load-Adaptive Processing of Continuous Queries Over Data Streams

Dhananjay Kulkarni
Boston University
808 Commonwealth Ave
Boston, MA, USA
kulkarni@bu.edu

Chinya V. Ravishankar
Univ of California-Riverside
900 University Ave
Riverside, CA, USA
ravi@cs.ucr.edu

Mitch Cherniack
Brandeis University
415 South Street
Waltham, MA, USA
mfc@cs.brandeis.edu

ABSTRACT

We introduce a new type of query, called a real-time continuous query (RCQ), that captures the real-time requirements of processing data streams. We develop techniques to efficiently process the RCQs in the presence of fluctuating query load and data load. We show that Rate-Monotonic scheduling is applicable to this problem domain, and show how to make this method adaptive to varying load conditions. When a set of queries becomes unschedulable due to load variations, we perform controlled input load shedding by dropping tuples using a novel feedback-based approach to decide which tuples to drop. Our work shows how to provide response time guarantees for processing RCQs, and enables making the appropriate trade-off between penalty due to missed deadlines and result accuracy. Our experiments show that our approach works very well and is usable in practice.

Categories and Subject Descriptors

H.4 [**Information Systems Applications**]: Miscellaneous

General Terms

Algorithms, Experimentation

Keywords

Real-time queries, Scheduling, Data streams, Load shedding

1. INTRODUCTION

A *data stream* is a continuous, unbounded, and time-varying sequence of data elements [7]. Streams arise naturally in numerous applications. Important examples include data transmitted by sensors, stock market data, and network monitoring data. It is common for users to issue *continuous queries* (CQ) over data streams [19]. A Data Stream Management System (DSMS) holds the query and executes it whenever new data arrives, returning the result to the user. Continuous queries are also called *standing* queries.

There has been considerable recent interest in building Data Stream Management Systems [11, 10, 20, 12, 23, 18, 21, 1]. Most such

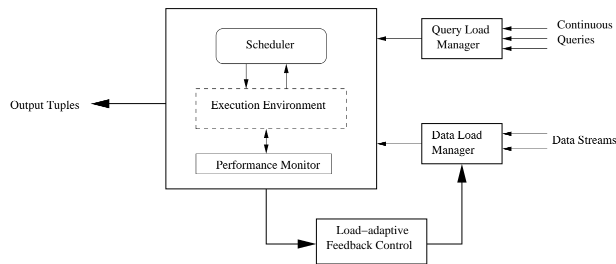


Figure 1: Conceptual Model of a Real-time Data Stream Management System

work addresses the issues of efficiency or memory utilization. However, very little attention has been paid in the literature to the processing of continuous queries in real-time environments.

1.1 Real-time Data Stream Management

Real-time processing of continuous queries is essential in numerous applications. For example, production management systems may require the diagnosis of a problem within a few seconds, and vehicular traffic management systems may require statistics to be computed or corrective actions to be taken before the information becomes stale. We refer to such systems as a Real-time Data Stream Management Systems (RDSMS). Work exists on real-time systems for relational data [16], but these techniques are mostly unsuited for data stream environments. A survey of the literature shows no DSMS designed to address real-time queries over data streams.

Figure 1 shows the conceptual model of a RDSMS, where users submit continuous queries with real-time requirements, to be executed whenever a new set of stream data arrive. The volume of data processed per query execution is called the *data load*, and is monitored and controlled by the *data load manager*. The number of queries present in the RDSMS system is called the *query load*. The *query load manager* admits new queries and discards obsolete queries. The *performance monitor* is responsible for monitoring the system performance, such as changes in load conditions and frequency of missed deadlines. In our work we introduce a new component, called the *load-adaptive feedback control*, whose job is to provide timely feedback in tuning the performance of the queries.

We now briefly review some applications requiring real-time queries on stream data. **Real-time monitoring of sensor data** is widely used for surveillance and data acquisitions. In the TAO Project [4],

for example, data is collected by sensors deployed in the ocean to measure parameters such as temperature, salinity, pressure, and streamed to a nearby station for real-time analysis. Some classes of real-time applications can have life-or-death consequences. Examples include tsunami alert systems [3], or robotic monitoring systems to identify people trapped in burning buildings, health monitors, and real-time control applications for aircraft or Intelligent Transportation Systems. **Real-time monitoring of Internet-related data** is another application. With the growth of the Internet, a large number of web application now generate online data streams. Examples include online stock monitoring portals [24], online bidding systems [2] and pay-per-click advertisements. Users (or companies that host these systems) query the data streams, and expect results within real-time constraints. For example, a slight delay in placing a buy or sell order on a stock can mean a huge gain or loss for institutional or other high-volume traders. In some cases, the users may be able to tolerate some degree of imprecision.

1.2 Motivation and Problem Definition

The characteristics of data streams can vary dynamically. Data rates may change, and the data values may vary erratically. The query mix may also vary dynamically. RDSMS systems must meet the real-time requirements of queries in the presence of changing query load and data load, which is a challenging problem.

In the following examples, we assume that the queries are periodic, with a period of 1 second. Unless stated, each query has a execution time of 0.5 seconds. For ease of exposition, we assume that the execution time for any query is directly proportional to the arrival rate of the input stream and directly proportional to the selectivity of the query operator.

1.2.1 High-Volume Data Streams

The RDSMS typically has no control over the volume of incoming data stream (data load), since streams are continuous and unpredictable. Since the execution time for a query depends on data load, it is practical to assume that the execution time of a query is different each time it runs. In Figure 2, Q2 misses its deadline in the 2nd iteration, because an increase in data volume causes it to executes for an extra 0.5 seconds.

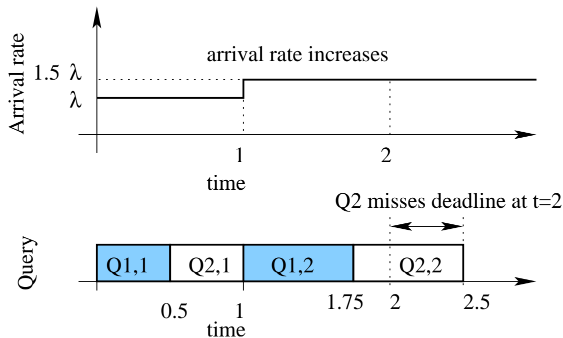


Figure 2: Query Q2 misses deadline when arrival rate increases from λ to 1.5 λ

1.2.2 Unpredictable Data Distributions

The selectivity of a query operator depends on some predicate (say, a selection predicate or join condition) and the distribution of input values. However in a data stream environment, the data distribution of tuples might vary with time due to external events which are beyond the RDSMS control. Thus, variations in the selectivity of the query operators may cause the query execution time to vary. In Figure 3, Q1 misses its deadline in the 4th iteration, because Q1 executes for an extra 0.25 seconds after the selectivity increases by 25%.

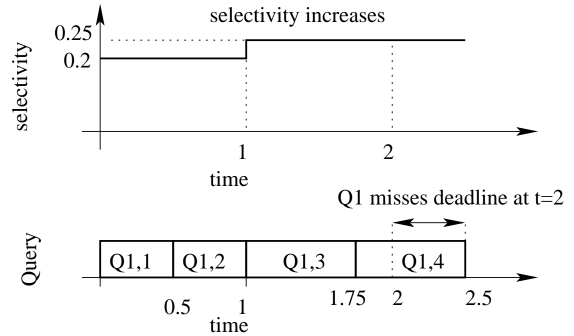


Figure 3: Query Q1 misses deadline when selectivity increases from 0.2 to 0.25

1.2.3 Dynamic Admission and Removal of Queries

In a general purpose RDSMS, the query load changes when users submit new queries or remove old queries. When a higher priority queries are added, some query (that met its deadline in the previous iteration) may miss its deadline due to unavailability of the processor time. For example, query load on a RDSMS becomes very high when multiple users try to monitor a certain interesting event such collapse of a stock value. As shown in Figure 4, Q2 misses its deadline in the 2nd iteration, because a higher priority query Q3 is introduced into the system.

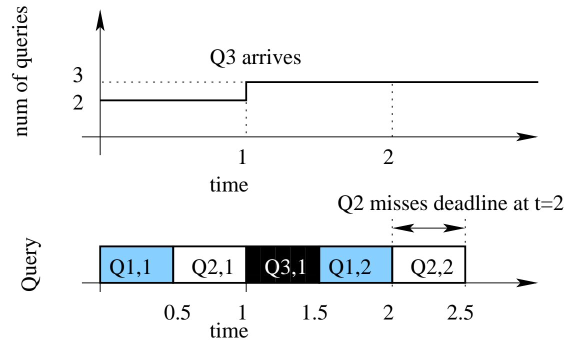


Figure 4: Query Q2 misses deadline when query load increases from 2 to 3

In this work, we will first define the semantics of real-time queries and then address the problem of scheduling real-time queries under the load constraints.

1.3 Approach

We treat the scheduling of queries as a real-time scheduling problem amenable to rate-monotonic scheduling [17]. Our approach is based on sacrificing some amount of computation accuracy in order to meet the query deadline. Initially, we perform a schedulability test to determine if all the queries can meet their deadlines. If the query set is not schedulable, we determine what reduction in the processing load (hence the execution times), which will make the queries schedulable. The idea will be to reduce the data load to achieve this reduction in execution times.

This reduction in the data loads leads to imprecision in query results. We must reduce data loads in such a fashion that the overall error is minimized. Once we determine an appropriate schedule that meets all deadlines and minimizes the error, we allow the schedule to remain in place for a time determined by the rescheduling policy. For example, one option might be to monitor the error, and reschedule when the error exceeds a given threshold. Another option may be to reschedule periodically, say, when each query has executed at least once.

2. PROBLEM FORMULATION

In this section we formalize various concepts that are used later in the presentation. We define real-time queries and describe the query processing model. Next, we discuss the query load and data load issues that make real-time processing very challenging. Finally, we state the problem that we wish to address.

2.1 Data Streams

We define a data stream as a sequence of data tuples that conform to a particular schema. Let S be a schema for a relation. The data stream s is represented as the sequence of tuples $\langle a_{t_1}, \ldots, a_{t_n} \rangle$, where each a_{t_j} conforms to schema S and the subscript t_j is the time-stamp of the tuple. We assume that the tuples are partially ordered by the time-stamps when they arrive into the system.

Let T be the current time instant. Given a time range $[t_1, t_2]$, where $t_1 \leq t_2 \leq T$, the sequence $s[t_1, t_2] = \langle a_{j_1}, a_{j_2} \ldots, a_{j_n} \rangle$, where $t_1 = j_1 < j_2 < \cdots < j_n = t_2$ represents the subset of data stream tuples that arrived in the range $[t_1, t_2]$.

The arrival rate is the number of data tuples arriving per unit time. We assume that the arrival rate λ is time-varying and the RDSMS does not have a priori knowledge of the arrival rate characteristics of the data stream. Although the tuples arrive at various times, we assume, in line with standard practice, that arrival rate is aggregated over discrete time intervals. Thus, arrival rate at time t is specified a $\lambda(t)$ tuples per unit time.

Let $Z_l(t)$ denote the data distribution of the values of an attribute l, for all the tuples arriving before time t. Although $Z_l(t)$ can be approximated by storing summaries of all tuples that have arrived before t, it need not be the case that $Z_l(t) = Z_l(t')$ for $t' > t$. That is, we cannot predict future values in the data stream.

2.2 Real-time Continuous Queries

We define a real-time continuous query (RCQ) as a continuous query with real-time constraints. The set of tuples over which a query is executed is called the *working window* of that iteration. For each successive iteration, we slide the working window by δ time units, where δ is the *stride* of the sliding window.

Let t_0 be the time when the query is admitted into the system. In general, the ith iteration of the query processes tuples in the set $W_i = \{a_t | t_s < t \leq t_e\}$, where $t_s = (t_0 \times (i-1) \times \delta)$ and $t_e = (t_0 \times (i-1) \times \delta + T_w)$ represents the time-stamp range $[t_s, t_e]$ of tuples in the working window.

In our work, a RCQ is specified as a tuple $\langle q, T_w, \delta, d, g \rangle$, where

- q is a query composed of operators that are members of the set { `select`, `project`, `join`, `aggregate` },
- T_w is the time *window* used to define the subset of stream tuples that are processed per execution,
- δ is the *stride* of the time window,
- d is the relative *deadline* for query execution, and
- g is the *weight* of the query, characterizing its importance, or equivalently, the penalty for missing its deadline.

The *life-time* of a query refers to the time between *admission* of the query and the *deletion* of the query from the system. A real-time continuous query is executed multiple times during its life-time. An *iteration* refers to the instance of execution of the query. The query is said to be executed for the ith time in iteration i.

2.3 Real-time Query Processing Model

We consider executions of a RCQ to be preemptive, so that a query q in execution may be preempted by a query that has a higher priority than q. However, our processing model requires iteration i of any query q to complete before iteration $i+1$ of q is ready for execution. We next define some terms that we will use in the rest of the paper.

1. The *admission time* A of the query is the time at which the user submits the query to the system.
2. The *release time* R of the query is the time at which all the data tuples (in window) are available to start the next iteration of the query.
3. The *begin time* B of the query is the time when the query is scheduled for execution.
4. The *finish time* F of the query is time when the query completes its execution.
5. The *execution time* E of the query is the time taken to process the tuples in the window.
6. The *absolute deadline* D of the query is the time by which the query must complete its execution.
7. The *relative deadline* d of the query is the absolute deadline minus the release time.
8. A *schedule* is an assignment of processor to the queries in the system. A query is said to *executing* during the time interval that the processor is assigned to the query.

Consider the ith iteration of a query. We say that the query *misses* the deadline if $F_i > D_i$. A query is said to be *successful* in meeting the deadline if $F_i \leq D_i$.

A *schedulable query* is one which successfully meets its deadline. The ith iteration of the query is said to be schedulable if $F_i \leq D_i$. In contrast, a query is *tardy* if it misses its deadline. The ith iteration of the query is said to be tardy if $F_i > D_i$.

The *overrun time* Φ_i is the difference between the finish time of the ith iteration and the absolute deadline D_i of a tardy query. Overrun time of a schedulable query is 0.

2.3.1 The Generic Model

In our work, a query tree in equivalent to a *task* in a real-time environment. We used a simple heuristic to push down selections and projection, but our approach is independent of optimization heuristics, hence we do not discuss further. Each query tree is a directed acyclic graph composed of query operators (vertices), with the output of the child operator feeding the input queue (edge) of the parent operator. We assume that the dependencies among the operators are maintained during query execution, which means that the root operator is executed last. Although query execution can be preempted, the query cannot be decomposed into logical sub-tasks for scheduling purposes.

When an operator is scheduled for execution, it performs the following three steps.

1. **READ** the w tuples from input queue (in main memory), where w is the cardinality of the working window.
2. **PROCESS** the w tuples by executing the operator-specific routines.
3. **WRITE** the resulting tuples $sel \times w$ to the output queue (in main memory), where sel is the selectivity of the operator.

Given a working window, the cost (in terms of time) of READ is fixed. The cost to PROCESS depends on the operator-type. The cost to WRITE depends on the how many resulting tuples are produced. A highly selective operator produces less tuples as output. Let T_R, T_P, and T_W be the time needed to perform the READ, PROCESS and WRITE steps, respectively. The execution time for any operator is estimated using the generic Equation 1. The execution time for individual operators is discussed in Section 3.1.1.

$$E_{\text{opr}} = T_R + T_P + T_W \tag{1}$$

2.3.2 Query Load

The query load is characterized by the number of queries in the system. We allow the query load to be dynamic, which means that users may submit new queries or delete existing queries at any time.

Let L_t be the query load at time t. The query load at some time $t' = t + \Delta t$ is given by $L_{t'} = L_t + \beta_A - \beta_D$, where β_A is the number of *new* queries admitted and β_D is the number of queries *deleted* during the time interval Δt.

2.3.3 Data Load

The data load is characterized by the number of tuples that must be processed per iteration. Let W_i be a stream's working window for the ith iteration of the query. Given the working window range $[t_s, t_e]$, the cardinality of the working window is $w_i = \int_{t_s}^{t_e} \lambda(t)dt$, where λ is the arrival rate of the stream at time t. Thus, the data load for the ith execution of a single-stream query is $D_i = w_i$.

For a query involving m data streams, the data load for the ith iteration is $D_i = \sum_{j=1}^{m} w_i^j$, where w_i^j is cardinality of the working window over stream j.

2.3.4 Imprecise Query Execution

Let W be the working window in some iteration of a query Q. The execution of Q is *precise* if it runs over all the tuples in W. The execution is *imprecise* if Q is executed over some of these tuples only, that is, over a window $W' \subset W$.

Such imprecision can be quantified using various error functions. Consider the set $\{Q_1, Q_2, \ldots, Q_n\}$ of queries. Let query Q_i have weight g_i, and execute for I_i iterations. Let ϵ_i^k be the error in executing the kth iteration of Q_i. We discuss how to quantify ϵ_i in Section 3.3. The total error of a schedule for these n queries is

$$\epsilon = \sum_{i=1}^{n} g_i \left(\sum_{k=1}^{I_i} \epsilon_i^k \right) \tag{2}$$

2.4 Problem Statement

We address the following problem: *Given a set of RCQs $\{Q_1, \ldots, Q_n\}$ over data streams $\{s_1, \ldots, s_m\}$, find a schedule such that each Q_i, $1 \le i \le n$ meets its deadline, and the total error due the imprecise computations is minimized.*

We solve this problem under the following constraints.

- Data distribution of the stream attribute-values is not known a priori.
- Arrival rate of a data stream is not known a priori.
- Users may submit new queries or delete existing queries.

3. APPROACH

We treat the scheduling of queries as a real-time scheduling problem amenable to rate-monotonic scheduling. Our approach is outlined in Figure 5. Initially, we perform a schedulability test to determine if all the queries can meet their deadlines. If the query set is not schedulable, we determine what reduction in the processing load (hence the execution times) will make the queries schedulable. The idea will be to reduce the data load to achieve this reduction in execution times.

This reduction in the data loads leads to imprecision in query results. We must reduce data loads in such a fashion that the overall error (see Equation 2) is minimized. Once we determine an appropriate schedule that meets all deadlines and minimizes the error, we allow the schedule to remain in place for a time determined by the rescheduling policy. For example, one option might be to monitor the error, and reschedule when the error exceeds a given threshold. Another option may be to reschedule periodically, say, when each query has executed at least once.

3.1 Estimating the Execution Time of Query

We abstract the query execution time in terms of the cost variables described in Table 1. Further, we define selectivity (*sel*) as the ratio of number of tuples that are generated as output, to the number of tuples that are processed by the operator. For selection, projection and aggregate operations the selectivity is in the range $[0, 1]$. Selectivity of a join operation in the range $[0, (w_l w_r)/(w_l + w_r)]$, where w_l and w_r are the cardinalities of left and right working windows, respectively.

3.1.1 Operator Execution Time

Let w be the cardinality of the working window for the selection, projection and aggregate operations. These operations require one scan of all tuples over the working window. The execution times for a select, project and aggregate are show in Equation 3, 4, and 5, respectively.

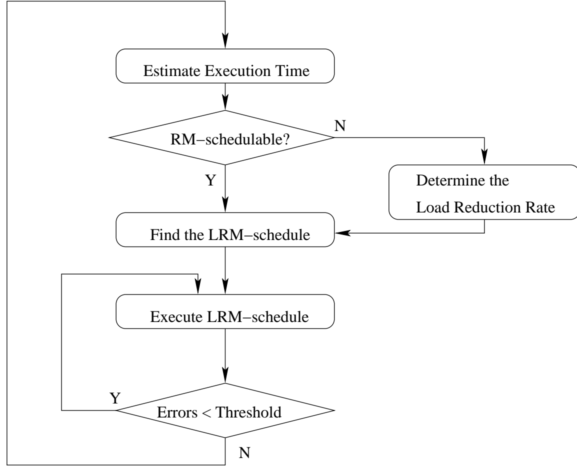


Figure 5: Our Framework for Scheduling Real-time Continuous Queries

Table 1: Cost variables

Name	Description	Value used in Experiments (millisecond)
C_R	Time taken to READ a tuple from input queue	1×10^{-3}
C_W	Time taken to WRITE a tuple to output queue	1×10^{-3}
C_σ	Time taken to perform one comparison operation, such as $val = 10$	2.5×10^{-3}
C_π	Time taken to identify the attribute to project	2.0×10^{-3}
C_Ω	Time taken to perform a arithmetic operation, such as $val1 + val1$	3.0×10^{-3}
$C_\bowtie$	Time taken to perform 1 probe of the hash table	3.0×10^{-3}
C_h	Time taken to hash a tuple into hash table	2.5×10^{-3}

$$E_{\texttt{select}} = \{wC_R\} + \{wC_\sigma\} + \{swC_W\} \quad (3)$$
$$E_{\texttt{project}} = \{wC_R\} + \{wC_\pi\} + \{wC_W\} \quad (4)$$
$$E_{\texttt{aggregate}} = \{wC_R\} + \{wC_\Omega\} + \{C_W\} \quad (5)$$

For equi-joins we use a symmetric probe technique [14]. Let w_l and w_r be the cardinalities of the right and left working windows for a 2-way join. The join is performed in 2 stages. First, we hash the tuples in the left window as they arrive at the input. Second, we probe the right hash table, using each tuple in the left window as the key. The same steps are applied for the tuples in the right window. Hence, the execution time for a 2-way join is as shown in Equation 6.

$$E_{\texttt{join}} = (w_l + w_r)C_R + (w_l + w_r)(C_\bowtie + C_h) + s(w_l + w_r)C_W \quad (6)$$

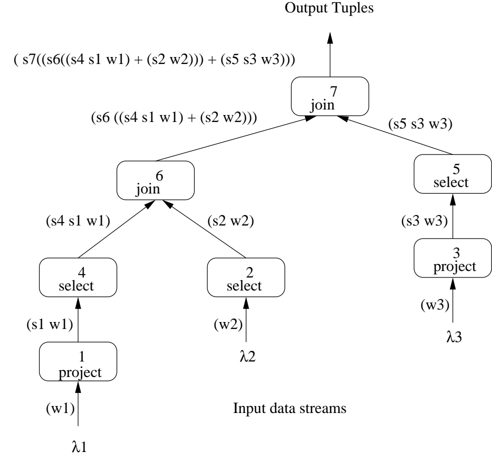


Figure 6: Example showing how to estimate the number of tuples processed by a join query over 3 data streams

3.1.2 Query Execution Time

Given a query specification, the query optimizer is responsible for generating a query plan (also called the query tree) to execute the query. We estimate the execution time of a query tree as follows.

A query tree is a directed-acyclic graph of v vertexes (operators) and e edges (input streams). When the output of a certain operator (parent) serves as an input for a next operator (child) in the query tree, the processing load of the child operator is calculated as $w_{child} = s \times w_{parent}$, where w_{parent} is the cardinality of working window of parent operator, and s is the selectivity of the parent operator.

Example: Consider a join query over 3 data streams, with arrival rate $\lambda_1, \lambda_2, \lambda_3$. The input edges in Figure 6 show the number of tuples processed by each operator in query tree, where s_i is the selectivity of operator i. The *total data load* is given by $\sum_{j=1}^{v} w_j$, where w_j is the number of tuples processed by operator j. Let E_j be the execution time of operator j. The *total execution time* the query is given by

$$E = \sum_{j=1}^{v} E_j \quad (7)$$

3.2 Load-adaptive RM Scheduling

We first show that real-time processing of continuous queries can be achieved using rate monotonic (RM) scheduling. RM algorithm [17] assigns static priorities to tasks, so that tasks with shorter periods are given higher priorities. As noted in earlier literature [17], a set of tasks can be scheduled under RM scheduling if the tasks are preemptive, each task has a relative deadline equal to its period, and there is no exclusive sharing of resources. A schedule for executing a set queries is said to be *RM-feasible* if all queries complete before their deadlines. A query is said to be *RM-schedulable* if it can be executed using some RM-feasible schedule.

In our model, a query will process a new working window after every stride interval. In the standard terminology of scheduling, we would say that a new iteration of the query is released (for execution) at each stride interval. RM scheduling is applicable in our case because of the following properties of the continuous queries.

- All queries are periodic, with the period equal to the stride of the working window.
- All queries are preemptive, and the cost of preemption is negligible.
- All queries are independent, with no precedence constraint.
- All queries have relative deadlines equal to the stride.
- All queries compete for CPU time and the memory available is large enough for query execution.

We propose a preemptive static priority scheme (like RM scheduling), in which the priority of the query is inversely proportional to its period. The query with the highest priority is scheduled and it is executed until it either completes, or it is preempted by the arrival (release) of a higher priority query. The focus of our work, however, is to adapt the schedule to changing load conditions.

3.2.1 Conditions for RM-schedulability

We assume that there is a query set $\zeta = \{Q_1, \ldots, Q_n\}$ of n real-time continuous queries with execution times $E_1, E_2, \ldots, E_n$, and periods $P_1, P_2, \ldots, P_n$, such that $P_1 \leq P_2 \leq \ldots \leq P_n$. We will now determine conditions for RM-schedulability (a detailed discussion can be found in [17]).

Since the query periods in the relative order $P_1 \leq P_2 \leq \ldots \leq P_n$, the only condition to be met for ensuring that Q_1 can be feasibly scheduled is $E_1 \leq P_1$. The first iteration of query Q_2 will meet its deadline if it can find enough time to finish over $[0, P_2]$. Let Q_2 finish at time t. Since Q_1 is a higher priority task, $\lceil \frac{t}{P_1} \rceil$ iterations of Q_1 are completely executed in time range $[0, t]$. Thus, for Q_2 to complete, in addition to multiple iterations of Q_1, there must be at least E_2 time available. That is, the following condition must hold:

$$t = \left\lceil \frac{t}{P_1} \right\rceil E_1 + E_2 \tag{8}$$

This means that Q_2 can meet its deadline if there exists some $t \in [0, P_2]$ that satisfies this condition. Next, we generalize this condition for the entire query set ζ and show that each query $Q_i \in \zeta$ is RM-schedulable iff conditions C1 and C2 hold. We use the following notations:

$$V_i(t) = \sum_{j=1}^{i} E_j \left\lceil \frac{t}{P_j} \right\rceil \tag{9}$$

$$L_i(t) = \frac{V_i(t)}{t} \tag{10}$$

$$L_i = \min \{L_i(t)\}, \; 0 < t < P_i \tag{11}$$

$$L = \max \{L_i\} \tag{12}$$

Query Q_i is schedulable if there exists some $t \in [0, P_i]$, such that $t = V_i(t)$. We observe that $V_i(t)$ is a constant in Equation 9, except at a finite number of points when the queries are released for the next iteration. Thus, we need to compute $V_i(t)$ at the times

$$\tau = \left\{ lP_j \mid 1 \leq j \leq i, \; 1 \leq l \leq \left\lfloor \frac{P_i}{P_j} \right\rfloor \right\} \tag{13}$$

From this result, we can say that the queries are RM-schedulable under the conditions $C1$ and $C2$ below. Under these conditions, each RM-schedulable query completes at a time t', such that $V_i(t') = t'$.

C1: If $\min_{t \in \tau_i} \{V_i(t)\} \leq t$, then Q_i is RM-schedulable

C2: If $\max_{1 \leq i \leq n} \{\min_{t \in \tau_i} V_i(t)/t\} \leq 1$ for $1 \leq i \leq n$, and $t \in \tau_i$, then query set ζ is RM-schedulable.

3.2.2 Reducing the Execution Time to Satisfy RM-schedulability

If a query that is not RM-schedulable misses its deadline and is allowed to execute after the deadline, it will finish at time $t > P_i$. We now outline our method to reduce the execution time of queries through load shedding, such that all queries meet the deadlines at the cost of some imprecision. Our technique is outlined in Algorithm 1

We focus our discussion on a query Q_k which fails the RM-schedulability conditions stated in Section 3.2.1, such that all $Q_j, 1 < j < k$ are RM-schedulable. We also observe that Q_k is also the highest priority query in the query set that does not meet its deadline in the first iteration.

Next, we define *overrun time* as amount of execution time remaining when it deadline is passed. Clearly, the overrun time of RM-schedulable queries is 0. The time available for query to execute before its deadline $t = P_k$ is

$$\widehat{E}_k = P_k - \sum_{j=1}^{k-1} E_j \left\lceil \frac{P_k}{P_j} \right\rceil \tag{14}$$

Thus, the overrun time of a query that may misses its deadline will be

$$\Phi_k = \max \left[0, E_k - \widehat{E}_k\right] \tag{15}$$

We propose a technique of transforming this non-feasible schedule into a feasible one. We first note that, making Q_k RM-schedulable means reducing the execution time of some queries that complete at a time $t \leq P_k$. Let the reduction in the execution times be $e_1, \ldots, e_k$. Next, we show how to derive these values.

We want to distribute Φ_k over $\left\lceil \frac{P_k}{E_1} \right\rceil$ instances of Q_1 and $\left\lceil \frac{P_k}{P_2} \right\rceil$ instances of Q_2, and so on, and finally over a single instance of Q_k. We will distribute Φ_k in inverse proportion both to the weight and the selectivity of the query.

Let the *reduction ratio* r_i of query i be $(s_i \times g_i)^{-1}$, where s_i is the selectivity, and g_i is the weight of the query i. Note that the weights are user (or application) specified, hence our approach is very general. if an application decided to make some queries very importance (high weight), we make sure that it is penalized less, that others that are less important. We explain this flexibility further as follows. We divide Φ_k into k parts in the ratio $r_1 : \ldots : r_k$. The fraction of Φ_k distributed to each query Q_i is called the cumulative reduction $\widehat{e}_i$ and is given by

$$\widehat{e}_i = \Phi_k \frac{r_i}{\sum_{j=1}^{k} r_i} \tag{16}$$

Since $\left\lceil \frac{P_k}{P_i} \right\rceil$ iterations of Q_i complete before P_k, the per-iteration reduction in the execution time is given by

$$e_i = \frac{\widehat{e}_i}{\left\lceil \frac{P_k}{P_i} \right\rceil} \tag{17}$$

Algorithm 1 Load Reduction Technique

Require: Query set ζ of size n, such that $Q_i \in \zeta$ has period P_i, deadline δ_i, weight g_i.
  Let the periods be in the order $P_1 \leq \ldots \leq P_n$
  for $i = 1$ to n **do**
    Estimate the execution time E_i as described in Section 3.1
    Check if Q_i is RM schedulable by applying conditions stated in Section 3.2.1
    if Q_i is not RM schedulable **then**
      Find the overrun time Φ_i using Equation 15
      Find the execution time reduction e_i using Equation 17
      $E'_i \leftarrow E_i - e_i$ {Q_i with the new execution time E'_i is now RM schedulable}
      Find the data load reduction using Equation 20
    end if
  end for

The reduced execution time, after the applying above technique is given by

$$E'_i = E_i - e_i \tag{18}$$

With the new execution times, each query finishes at time $t = V'_i(t)$ given by

$$V'_i(t) = \sum_{j=1}^{i} E'_j \left\lceil \frac{t}{P_j} \right\rceil \tag{19}$$

3.2.3 Determining the Reduction in Data Load

Given the reduced execution time E' of the RM-schedulable query, we must determine the number of tuples to drop so that execution time drops to E'. From Equations 3–7, we know that the execution time is a function of the cardinality of the working window w and operator selectivity. Assuming that the selectivity is constant during the iteration, we can represent the execution time of a query as $F(w)$, where w is the cardinality of the working window.

We want to determine the w', such that $F(w') = E'$. Assuming the load reduction is shared equally among the participating data streams, we first substitute $w'_i = kw_i,\ 1 \leq i \leq m$, where m is the number of streams participating in the query and $0 < k < 1$. Next, we solve the equation $F(w') = E'$ to determine k.

Let w be the size of the original working window. Once we determine the size of the reduced working window for stream i as $w'_i = k \times w_i$, the number of tuples to be dropped is given by

$$\widehat{w}_i = w_i - w'_i \tag{20}$$

Once we know the number of tuple to be dropped $\widehat{w}_i$, we sample the working window and uniformly drop tuples over the range of the working window. Uniformly dropping the tuples is a simple policy which does not lead to extra processing. Semantically choosing tuples to drop is also another alternative, however, it is likely to consume precision CPU every-time a new LRM schedule has to be determined

3.2.4 The LRM-schedule

We call our method *load-adaptive rate-monotonic* (LRM) scheduling because any LRM schedule ensures RM-schedulability of queries through load shedding. Algorithm 2 shows how the queries are scheduled under LRM scheduling.

Algorithm 2 LRM Schedule

Require: Query set ζ of size n, such that $Q_i \in \zeta$ has execution time E'_i and period P_i.
  Let the periods be in the order $P_1 \leq \ldots \leq P_n$
  for $t = 0$ to ∞ **do**
    $P_i = \min(P_j)\ \forall 1 \leq j \leq n$
    repeat
      Q_i is executed for 1 time unit
      $t \leftarrow t + 1$
    until $(t' - t \neq E'_i)$ OR (t'=release time of a query Q_k, such that $P_k < P_i$)
    if $(t' - t) = E'_i$ **then**
      Finish time $F_i \leftarrow t'$
    else
      Begin time $B_k \leftarrow t'$
    end if
  end for

3.3 Error Function

Dropping input tuples causes imprecise query execution. We can quantify the error in two ways.

The **processing error** ϵ'_i of a query Q_i is the number of tuples that are not processed before the deadline. Let Q_i be a query that is schedulable under LRM. Let $\widehat{w}_i$ be the reduction in the data load, which is obtained as stated in Section 3.2.3. The processing error is given by

$$\epsilon'_i = \widehat{w}_i \tag{21}$$

The **output error** ϵ_i of a query Q_i is the number of tuples that do not appear in the output of the query computation. Let Q_i have a selectivity of sel_i. The output error of executing query Q_i under LRM scheduling is given by

$$\epsilon_i = sel_i \times \widehat{w}_i \tag{22}$$

3.4 Run-time Adaptations of Schedules

An LRM schedule for a set of queries may become sub-optimal if kept unchanged for a long time. Since most continuous queries run for a long time, rescheduling periodically is useful in further adapting the schedule. We outline three approaches to do perform such schedule management during run-time.

3.4.1 Pessimistic Rescheduling

In this model, we re-examine all the load conditions and test for RM schedulability as soon as each query finishes. If the data and load conditions at this time are such that the same LRM schedule cannot be re-used, we apply the load reduction techniques stated in Section 3.2.3 and determine a new LRM schedule adapted to the new load.

This technique has high overhead, because a schedulability test must be performed every P_1 time units, which is the smallest period among the queries in the given query set. Thus, at a given time t the overhead is proportional to $\lfloor \frac{t}{P_1} \rfloor$. The advantage of this method is that we have finer control over the schedule. It is particularly useful if the load conditions are highly dynamic, and the performance of

a static schedule is likely to degrade very quickly.

3.4.2 Optimistic Rescheduling

In this approach, we allow to keep the LRM schedule in place until each query has run at least once. Hence, we test for schedulability once every P_n time units. The technique suffers a much lower overhead than the pessimistic approach. At any given time t the number of re-scheduling decisions made is upper-bounded by $\lfloor \frac{t}{P_n} \rfloor$.

Since $P_i < P_n, \forall i < n$, this technique works very well if the load conditions do not change frequently. Consider a query k, with period $P_k < P_n$. Using this technique, the query Q_k completes $\left\lceil \frac{t}{P_k} \right\rceil$ iterations, without having to switch to a new schedule.

3.4.3 Permissive Rescheduling

The optimistic and the pessimistic approaches perform *periodic* rescheduling. The period is upper-bounded by the period P_1 for pessimistic rescheduling, and by P_n for the optimistic technique.

We now describe an aperiodic technique in the section, called *permissive schedule management*. In this approach, we determine a new LRM schedule if at time t the total error exceeds a threshold τ. Thus, the same schedule is re-used as long as the total error remains below the threshold τ.

This technique is more adaptive than the periodic techniques. It suffers lower overhead when the load fluctuations are infrequent and a high overhead only when the load conditions are highly dynamic. Moreover, this technique allows the user to upper bound the imprecision of the LRM schedule.

4. EXPERIMENTAL ANALYSIS

We used a synthetic, as well as a real-world dataset to test our techniques. Figure 7 shows the arrival rates of the stream used in the datasets.

We used the DEC-PKT dataset [5] as real-world dataset. For single stream queries we worked with the TCP trace and for join queries we used the TCP, UDP and SF traces, with suitable equi-join conditions. The arrival rate of packets is very random, hence, we refer to it as RANDOM in our experimental setup.

We generated two types of workloads for our synthetic dataset. The STEADY dataset consists of 3 streams that have arrival rates, such that the *demand* for the CPU utilization is always more that 100 %. This ensures that queries are not RM-schedulable during the entire run of the experiment. The STEP dataset consists of a stream, whose arrival rate increases as time progresses. We use a step function, that increases the arrival rate by a fraction of 0.1, at each step.

A query load of 10 queries was used in most of the experiments. We used a decent mix of selection, projection and join queries to reflect a real-world scenario. The queries and their weights, deadlines and periods are listed in Table 2. For sake of brevity we omit the SQL specifications for the queries. The processing time for the various query operators shown in Table 1 were collected offline, by multiple executions of the operators (in isolation) and then averaging their execution times. All the simulations were run for 20 seconds, using a prototype we built for our work. Unless specified, optimistic schedule management was used as rescheduling policy. The per execution overhead of rescheduling was between 0.1-0.2 milliseconds, which can easily be accounted for while determining a LRM schedule.

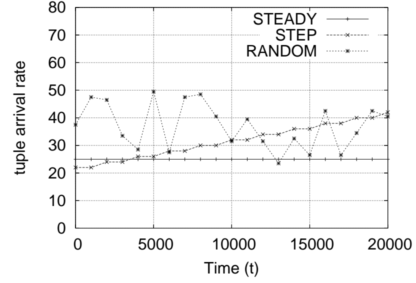


Figure 7: Types of data workloads used in the experiments

Table 2: Queries used

id	Type	Period	Weight
Q1	select	1	5
Q2	select	2	6
Q3	select-project	2	3
Q4	select-project	3	2
Q5	2-way join	4	1
Q6	2-way join	5	7
Q7	3-way join	5	8
Q8	2-way join-project	6	9
Q9	project-2 way join	6	10
Q10	avg	6	4

We compared our approach **LRM** with two alternative approaches. First, the traditional rate-monotonic approach, called **RM**, in which a tardy query was allowed to run to completion even after it misses its deadline. Second, the non-preemptive scheduling [17] approach called **HPF** in which the highest priority (lower period) query is scheduled first and allowed to run until completion. Our goal was to measure the performance of these 3 scheduling techniques in terms of the following.

1. **Miss rate:** The fraction of queries that miss their deadline at any given time t.
2. **Completion rate:** The fraction of queries that finished by time t. This includes the queries that may have missed deadlines.
3. **Total error:** The total error due to the imprecise computation, for all the iterations of a query.

4.1 Performance of a Static Schedule

A *cycle* refers to 1 complete iteration of the least frequent query in the given query task. For example, in our setup, a cycle is said to be complete when query Q10 (with period 6 seconds) finishes its first iteration. We applied our LRM scheduling at the start of the cycle (at t=0) and allowed to schedule to run until Q10 finished. The STEADY dataset was used for this experiment. We did not apply any of the rescheduling heuristics in this experiment, hence we consider the schedule to be *static*.

Figure 10 shows the results of this experiment. As expected, LRM misses no deadline in this cycle. The RM and HPF suffer because

the execution time of some queries may be high, and there is no provision to cut-down on the execution time. The number of queries completed by LRM as are also much higher than both the alternatives. Since, the total error is weighted, we see that our policy to distribute the load leads to smaller total error in the output.

4.2 Performance under Steady Load

Figure 9 shows that when the load is steady, our policy suffers lower missed rates than either alternatives. In fact, before the completion of 1 cycle (t=6000) we do not miss any deadline. A small fraction of deadlines are missed later in the run because of the rescheduling policy is initiated only later. Since we only periodically reschedule the LRM schedule, we are able to only tune the miss rates only at the rescheduling instances.

4.3 Performance under Step Load

Figure **??** shows our results under step workload. In this experiment, the CPU demand is increasing with time. Hence, as expected the RM scheduling misses more deadlines as time progresses. LRM also suffers more miss rate than in the case of a STEADY workload, but the miss rates are considerably lower than RM or HPF. Moreover, LRM consistently completes more queries than both the alternatives.

4.4 Performance under Random Load

Figure 11 shows the performance of our scheme under a more dynamic workload. Again, the LRM schedule completes far more queries in a given time and is able to limit the miss rate to around 2 %. After the 1st cycle completes, the total % error is also stable in case of the LRM schedule. For both, RM and HPF techniques, the total error is much higher than LRM.

4.5 Performance under Varying Query Load

In this experiment we vary the number of queries from 4 to 20. The performance of all schedules is expected to degrade as new queries are added. However, Figure 12 shows that the LRM policy is able to adapt to the query load much better than the simple RM policy. In such cases, a permissive policy to rescheduling is likely to be more beneficial.

5. RELATED WORK

Work exists in the area of real-time database systems [16]. However, all work in this area assumes that the data is static and hence, most of the data-related properties, such as the cardinalities of the relations, data distribution etc are known a priori. Thus, we believe that the techniques studied for relation-data are not directly applicable to streaming data.

Recently there has been numerous works on optimizing continuous queries over data streams. However, the bulk of the work is related to minimizing the memory utilization [20, 8, 9, 6] during query execution. A novel technique to maximize the output rate is proposed in [25]. However, very little has been done in the context of real-time application of these techniques.

The load shedding [25] approach has some overlap with our work, but the authors do not study this in the context of real-time queries. Moreover, the quality of service metric used in the semantic approach does not consider performance measures for real-time systems, such as miss ratios or latency of query execution.

Imprecise computation of real-time tasks was first proposed in [13]. This and other extensions to this work [22], however, assume that the task can be logically divided into 2 parts, namely the mandatory and the optional part. Moreover, they study the performance of schedule under static load conditions. We make no such assumptions and we are interested in quantifying the imprecision in the context of data streams.

The authors in [15] address how to handle over-running tasks and propose a techniques to run the optional part of the task on an aperiodic server. We make no such assumptions about the query, and treat each query as an indivisible task that cannot be run separately.

[26] is the only work we have come across in the context of real-time queries over data streams. However, this work is still preliminary and the paper does not provide any theoretical results of how to tune the query performance. The paper only outlines an approach to drop tuples when the arrival rates are high. The parameters that define the tuple drop rate are chosen experimentally, which is largely impractical for dynamic data stream environment. Moreover, the scheduling scheme is not discussed in detail. We address the problem of meeting deadline in a much broader context, by considering the fluctuations in the query load, data load, as well as selectivity of the queries.

A straight-forward approach to our problem is to run a overrunning query until it passes its deadline, and then stop its execution. This approach is similar to the one studied in [15], and is not suitable in our context because of various reasons. First, a query with very high frequency (small period), but a high execution time ($\geq$ the period) will hog the processor at all times and hence starve other queries in the system. Moreover, in a hard real-time systems the penalty due to unprocessed data is likely to become unacceptable.

Our goal is the address the problem in the context of a weakly hard DSMS, the goal being to meet all the deadlines of the queries – at the cost of some imprecision in the computation. When the queries are have a long life-time, this advantage of meeting deadlines easily outweighs the approximation error.

6. CONCLUSION

We have proposed a novel approach to schedule real-time queries over data streams. We have focussed on the problem of overrunning queries in a weakly hard real-time systems, where a small degree of missed deadlines and imprecision can be tolerated. Our work is novel because, we make no assumptions about data arrival rates, query load or the data distribution. Meeting query deadline in such setting is challenging. Our approach of meeting the query deadline by reduction the execution time of queries is very practical, and is based on well-known properties of rate-monotonic scheduling. Our load adaptive rate-monotonic (LRM) scheduling policy is dynamic and quickly adapts to the changing load conditions. LRM is able to minimize the total error caused to reduction in the processing load. Moreover, experimental results show that LRM works for various workloads and consistently out-performs the simple rate-monotonic policy and a non preemptive priority schedule. Since most continuous queries are long-running, rescheduling may be required. For dynamic settings, permissive rescheduling seems more suitable. For a workload where the arrival rate or query load can be estimated a priori, periodic rescheduling is sufficient.

As part of future work, we wish to explore how our scheduling can be applied in Aurora/Borealis [10].

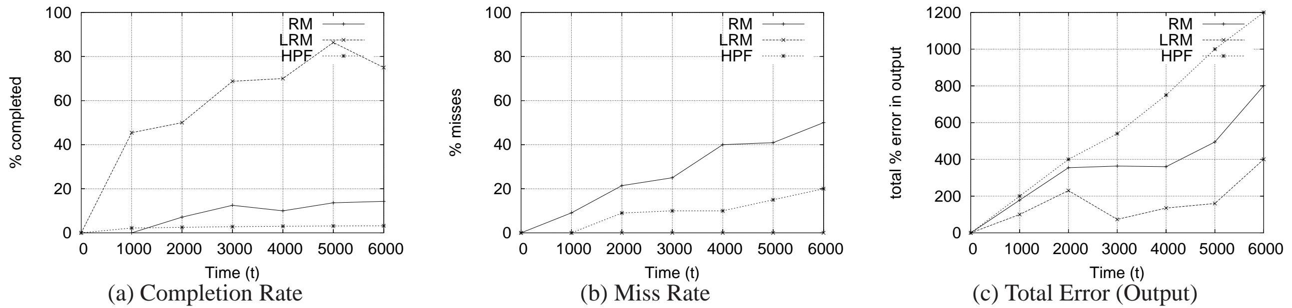


Figure 8: Experimental results for a single cycle

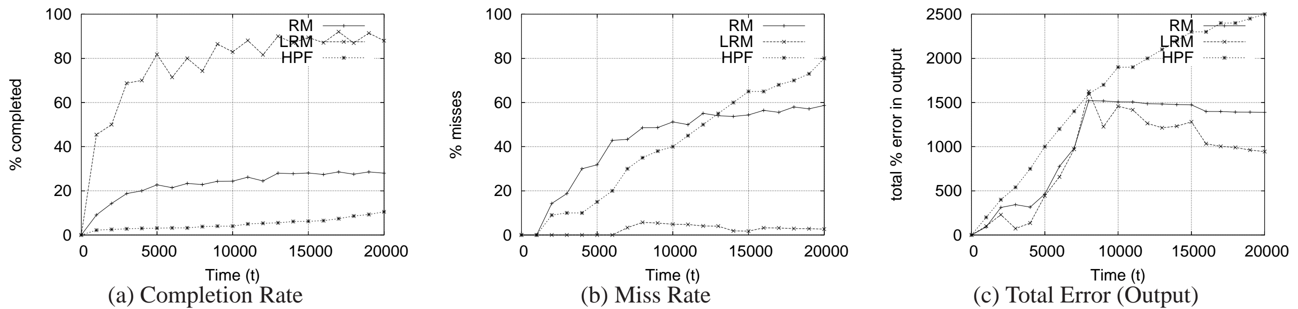


Figure 9: Experimental results for STEADY workload

7. REFERENCES

[1] Streambase systems inc. homepage, http://www.streambase.com/.

[2] ebay homepage, http://pages.ebay.com, 2003.

[3] Pacific tsunami warning center, http://www.prh.noaa.gov/pr/ptwc/, 2006.

[4] Tao project, http://www.pmel.noaa.gov/tao/, 2006.

[5] I. T. Archive. Ita homepage, http://www.acm.org/sigcomm/ita, 2005.

[6] R. Avnur and J. M. Hellerstein. Eddies: Continuously adaptive query processing. In *Proceedings of ACM SIGMOD*, volume 29. ACM, 2000.

[7] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *Proceedings of the twenty-first ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems*, pages 1–16, 2002.

[8] B. Babcock, S. Babu, R. Motwani, and M. Datar. Chain: operator scheduling for memory minimization in data stream systems. In *Proceedings of ACM SIGMOD*, pages 253–264. ACM Press, 2003.

[9] M. Cammert, J. Kramer, B. Seeger, and S. Vaupel. An approach to adaptive memory management in data stream systems. In *ICDE '06: Proceedings of the 22nd International Conference on Data Engineering (ICDE'06)*, page 137, Washington, DC, USA, 2006. IEEE Computer Society.

[10] D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams - A new class of data management applications. Technical Report CS-02-04, Department of Computer Science, Brown University, Feb. 2002.

[11] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proceedings of CIDR*, 2003.

[12] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. In *Proceedings of ACM SIGMOD*, pages 379–390, 2000.

[13] J.-Y. Chung, J. W. S. Liu, and K.-J. Lin. Scheduling periodic jobs that allow imprecise results. *IEEE Trans. Comput.*, 39(9):1156–1174, 1990.

[14] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Rec.*, 33(1):44–49, 2004.

[15] M. Gardner and J. Liu. Performance of algorithms for scheduling real-time systems with overrun and overload, 1999.

[16] B. Kao and H. Garcia-Molina. An overview of real-time database systems. pages 463–486, 1995.

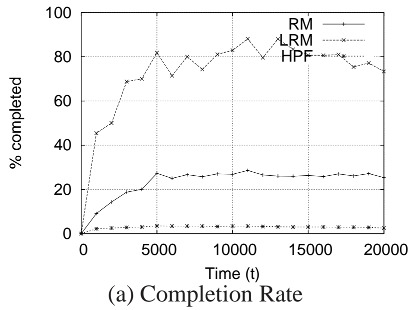
[17] C. M. Krishna and K. G. Shin. *Real-time Systems*. McGraw Hill, Reading, Massachusetts, 1997.

[18] S. Madden and M. J. Franklin. Fjording the stream: An architecture for queries over streaming sensor data. In *Proceedings of ICDE*, 2002.

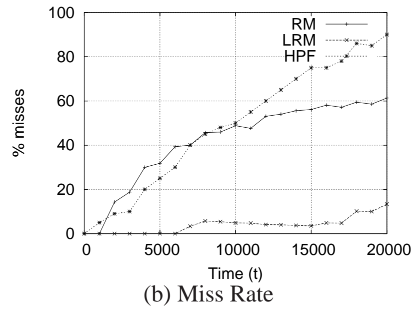
[19] S. Madden, M. A. Shah, J. M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proceedings of SIGMOD*, 2002.

[20] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. In *Proceedings of CIDR*, 2003.

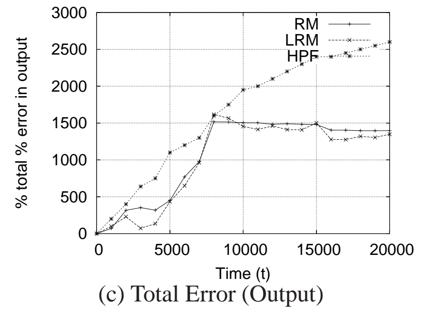
[21] P. Seshadri, M. Livny, and R. Ramakrishnan. Sequence query



(a) Completion Rate

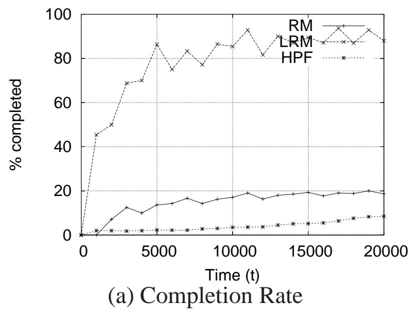


(b) Miss Rate

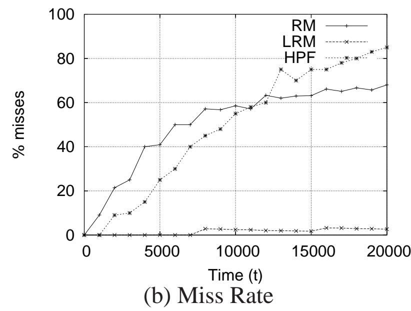


(c) Total Error (Output)

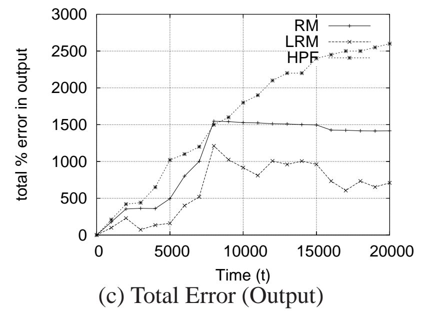
Figure 10: Experimental results for STEP workload



(a) Completion Rate

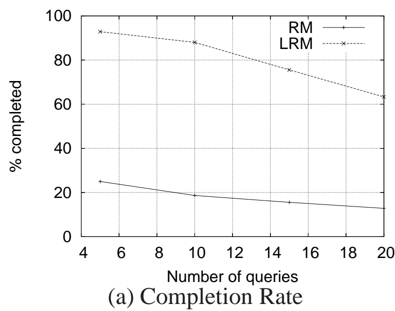


(b) Miss Rate

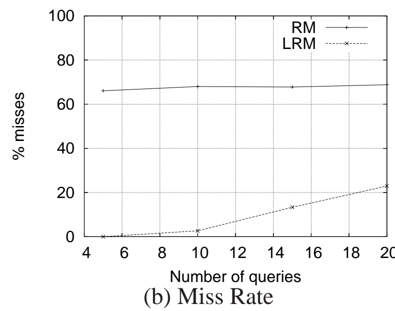


(c) Total Error (Output)

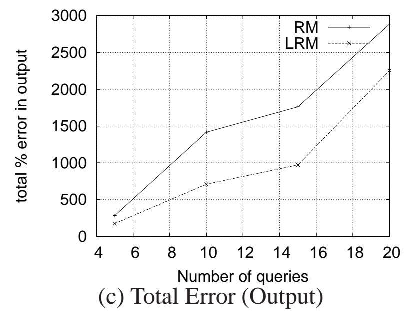
Figure 11: Experimental results for RANDOM workload



(a) Completion Rate



(b) Miss Rate



(c) Total Error (Output)

Figure 12: Experimental results for varying query load

processing. In *Proceedings of ACM SIGMOD*, pages 430–441, 1994.

[22] W. K. Shih and J. W.-S. Liu. Algorithms for scheduling imprecise computations with timing constraints to minimize maximum error. *IEEE Transactions on Computers*, 44(3):466–471, 1995.

[23] M. Sullivan and A. Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of USENIX Annual Technical Conference*, pages 13–24, 1998.

[24] Traderbot. Traderbot homepage, http://traderbot.com, 2003.

[25] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proceedings of SIGMOD*, 2002.

[26] Y. Wei, S. H. Son, and J. A. Stankovic. Rtstream: Real-time query processing for data streams. In *ISORC '06: Proceedings of the Ninth IEEE International Symposium on Object and Component-Oriented Real-Time Distributed Computing (ISORC'06)*, pages 141–150, Washington, DC, USA, 2006. IEEE Computer Society.