

Evaluating Trigger Conditions on Streaming Time Series with User-given Quality Requirements¹

Like Gao² Min Wang³ X. Sean Wang²

² CS Dept., Univ. of Vermont, VT, USA. {lgao, xywang}@cs.uvm.edu

³ IBM T.J. Watson Research Center, NY, USA. min@us.ibm.com

Abstract: For many applications, it is important to evaluate trigger conditions on streaming time series. In a resource constrained environment, users' needs should ultimately decide how the evaluation system balances the competing factors such as evaluation speed, result precision, and load shedding level. This paper presents a basic framework for evaluation algorithms that takes user-specified quality requirements into consideration. Three optimization algorithms, each under a different set of user-defined probabilistic quality requirements, are provided in the framework: (1) minimize the response time given accuracy requirements and without load shedding; (2) minimize the load shedding given a response time limit and accuracy requirements; and (3) minimize one type of accuracy errors given a response time limit and without load shedding. Experiments show that these optimization algorithms effectively achieve their optimization goals while satisfying the corresponding quality requirements.

Key Words: QoS (Quality of Service), trigger, streaming time series, prediction model

Category: H.2

1 Introduction

In many situations an application system needs to react to preset trigger conditions in a timely manner. For example, in network management systems, we may want to quickly divert traffic from certain area once a congestion is discovered; in road traffic control, we may want to immediately enact a camera when a car crosses a certain line while the traffic signal is red; and in environmental monitoring, we may want to start taking rain samples soon after certain event happens. In this paper, we tackle this monitoring problem in which trigger conditions need to be evaluated continuously on streaming time series.

Consider a stock market monitoring system where each stock s^i is a streaming time series. A stock broker may register many requests for his/her clients such as “notify Alice whenever stocks s^1 and s^2 are correlated with a correlation value above 0.85 during a one-hour period.” The system is to monitor the streaming stock data and trigger the actions in the requests whenever the corresponding conditions are evaluated true.

A perfect system would trigger the corresponding action without any delay whenever a condition becomes true. However, delay is unavoidable in a resource constrained environment. This delay can be alleviated by allowing approximate

¹ Partly supported by the NSF grants IIS-0415023, IIS-0430165, and IIS-0242237.

evaluation and load shedding. We here define approximate evaluation as to allow some conditions that are actually true (false) to be reported false (true, resp.), and load shedding as to selectively skip a fraction of conditions from evaluation.

Approximation and load shedding may lead to errors. However, their results may still be useful. These two methods have appeared in the literature in other contexts as well, e.g., ANN [2] is developed to searching for approximate nearest neighbors and a load shedding strategy is deployed in Aurora system [13, 1, 4].

Approximation can be useful in at least two ways. (1) End users may be satisfied with approximate results, but it is important that users are given the ability to specify an error bound. (2) Or an application system may use approximate results for optimization purposes. For example, a “speculative optimization strategy” can use fast approximate results to prepare (e.g., prefetch) for subsequent complex activities, while precise results can later be used to correct any error made in the aggressive, speculative phase [14]. In this scenario, precision and response time need to be balanced in order to achieve the best overall performance. In this case, the application system is the “user”, and it is important that the approximate algorithms can satisfy the “user” requirements.

Response time, approximation error, and load shedding level are competing factors. In this paper, we advocate that it’s the users’ needs that should ultimately decide how the evaluation system balances them. That is, the evaluation system should satisfy user-specified quality requirements in terms of these three factors. In other words, a concept of quality of service is needed. The resulting evaluation system is called *quality-driven*.

For a quality-driven system, it is important to have the ability to measure the (intermediate) result quality during the evaluation process. However, it is impossible to measure the accuracy (i.e., false positive and negative ratios) precisely when an approximation method is used. Indeed, the precise accuracy can only be measured *a posteriori*, i.e., only after we know the actual evaluation results of the trigger conditions. Instead, we measure accuracy in an *a priori* manner, i.e., the false ratios are estimated before the conditions are evaluated. To do this, we build a prediction model based on historical data analysis. At the evaluation time, we use this prediction model to derive accuracy estimates that are used to guide the evaluation.

Since the evaluation procedure is based on probabilistic prediction models, our system cannot satisfy accuracy in a strict sense. Instead, we use a concept similar to the *soft* quality of service (QoS) in computer networks [5]. That is, the system guarantees with enough confidence that the *expected* accuracy will be more than the given thresholds. Specifically, the system allows users to impose the following quality constraints:

- *Response time constraint*: All evaluation results must be reported within a given time limit.

- *Drop ratio constraint*: The percentage of the conditions that are skipped (no evaluation results are reported for them) cannot exceed a given threshold.
- *Accuracy constraints*: The expected false positive and negative ratios must not exceed the given thresholds, with enough confidence (the confidence is deduced from the prediction model).

Ideally, users should be able to impose any combination of the above constraints. But the evaluation system may not be able to satisfy all the constraints simultaneously. To solve this problem, one way is to ask the users to give up on some constraints (i.e., leave as unconstrained) and let the evaluation system to do its best in terms of these constraints, while satisfying the imposed constraints on the remaining parameters. In our scenario, if one constraint is left unspecified, the system can always satisfy all other constraints. This leads to three possible choices. For each choice, we provide an evaluation algorithm in this paper.

This paper makes three contributions. First, we initiate the study of a quality-driven system for evaluating trigger conditions on streaming time series. Second, we show how to use a prediction model to deduce the accuracy estimates. Third, we provide quality-driven evaluation algorithms, and show their effectiveness.

The rest of the paper is organized as follows. In Section 2, we review related work. In Section 3, we formally define the trigger conditions and the quality parameters. In Section 4, we introduce our prediction model and define the probabilistic quality constraints. We present evaluation algorithms in Section 5 and present our experimental results in Section 6. We conclude the paper with discussion on future research directions in Section 7.

2 Related Work

The quality-driven aspect of our work is similar to the QoS concept in computer networks [8, 12]. This paper adopts the QoS concept into trigger condition evaluation on streaming time series and presents a basic design strategy for developing such a quality-driven system.

Aurora [13, 1, 4] seems to be the only data stream processing system that contains a QoS component. In Aurora, a user may register an application with a QoS specification. Aurora tries to maximize an overall QoS function when it makes scheduling and load-shedding decisions. In our system, we allow multiple QoS parameters, and our system optimizes an unconstrained quality parameter while satisfying users-imposed constraints on all the remaining parameters.

With limited resources, using approximation techniques in processing continuous queries on data streams has been studied in [7, 9, 6, 11]. However, most approximate evaluation strategies only consider one quality aspect and neglect the others. For example, Chain [3] minimizes the memory usage without considering the response time at all. Our work differs from all such work in that we take different user-specified quality requirements into consideration.

3 Preliminary

A *time series* is a finite sequence of real numbers and the number of values in a time series is its *length*. A *streaming time series*, denoted s , is an infinite sequence of real numbers. At each time position t , however, the streaming time series takes the form of a finite sequence, assuming the last real number is the one that arrived at time t . In this paper, we assume that all streams are synchronized, that is, each stream has a new value available at the same time position.

In general, a trigger condition can be any user-defined predicate on streaming time series, and needs to be evaluated after every data arrival. We denote a set of conditions as $C = \{c_1, c_2, \dots, c_n\}$ and the reported evaluation result of condition c_i at time position t as $r(c_i, t)$. We denote the precise evaluation result (the actual value of the given condition if a precise evaluation process is used) of c_i at time position t as $R(c_i, t)$. Obviously, we have $r(c_i, t) \in \{\text{True}, \text{False}\}$, and $R(c_i, t) \in \{\text{True}, \text{False}\}$.⁴ Note that $r(c_i, t)$ may not be equal to $R(c_i, t)$ due to the approximate nature of the system. Let C_T denote all the conditions in C whose reported results are **True** at time position t , i.e., $C_T = \{c_i \in C | r(c_i, t) = \text{True}\}$. Similarly, let $C_F = \{c_i \in C | r(c_i, t) = \text{False}\}$ and $C_D = \{c_i \in C | c_i \text{ is dropped at time position } t\}$. We call C_T , C_F and C_D the reported-True set, reported-False set, and dropped set, respectively.

Using the above notation, we define four parameters:

1. *Response Time*, RT , is the duration from the data arrival time t to the time when last condition c_i , $r(c_i, t) = \text{True}$, is reported.
2. *Drop Ratio*, DR , is the fraction of the conditions (among all the conditions in C) that are dropped (not reported).
3. *False Positive Ratio*, FPR , of a reported-True set C_T is the fraction of the conditions (among all the conditions in C_T) whose actual values are **False**. We define $FPR = 0$ if C_T is an empty set.
4. *False Negative Ratio*, FNR , of a reported-False set C_F is similarly defined.

Note that response time is defined only on conditions that are reported true.

4 Prediction Model

While it is easy and straightforward to measure the quality parameters RT and DR at any time, it is difficult to measure the other two parameters, FPR and FNR , without actually evaluating all the conditions. A practical solution is to build a prediction model using historical evaluation results and calculate the *expected FPR* and *FNR* based on the model in a probabilistic manner.

For each condition c_i , we define a random variable X_i to state the outcome of its evaluation. Clearly, X_i follows Bernoulli distribution $X_i \sim B(\rho_i)$, that is,

⁴ In the rest of the paper, we may omit t and use $r(c_i)$ ($R(c_i)$) to denote the reported evaluation result (actual value) of condition c_i at time position t when the context is clear.

$$X_i = \begin{cases} 1 & \text{if } R(c_i) = \text{True} \\ 0 & \text{if } R(c_i) = \text{False} \end{cases} \text{ with } \begin{cases} P(X_i = 1) = \rho_i \\ P(X_i = 0) = 1 - \rho_i \end{cases}.$$

In this paper, we make the simplifying assumption that all X_i 's (for $1 \leq i \leq n$) are mutually independent. Clearly, the mean ρ_i is also the expected value of X_i . Depending on how the system treats c_i , we see three different cases for ρ_i : (1) c_i is precisely evaluated. In this case, we have $\rho_i = 1$ if c_i is evaluated to be True and $\rho_i = 0$ if c_i is evaluated to be False. (2) c_i 's result is reported based on an approximation procedure, e.g., prediction. In this case, ρ_i cannot be known exactly. Instead, its estimate $\hat{\rho}_i$ will be used. Specifically, $\hat{\rho}_i$ can be approximated by a normal distribution function $Norm(\mu_i, \sigma_i^2)$, where the mean value $\mu_i = \bar{X}_i$ and the variance $\sigma_i^2 = \frac{(1-\mu_i)\mu_i}{N}$, i.e., $\hat{\rho}_i \sim Norm(\mu_i, \frac{(1-\mu_i)\mu_i}{N})$. (Here, \bar{X}_i denotes the sample mean, and N the sample size.) We may obtain the above by analyzing the historical evaluation results for each condition, i.e., by adopting a data mining approach. (Examples can be found in [10, 14].) (3) Too little historical data to estimate ρ_i . For all the three cases, the estimate of ρ_i can be viewed as a random variable that follows normal distribution as shown below.

Case	μ_i	σ_i^2	Note
c_i is precisely evaluated	1 (or 0)	0	known True or False
c_i is predicted	μ_i	$\frac{(1-\mu_i)\mu_i}{N}$	from N samples
otherwise	0.5	0.25	unknown

With the prediction model, given a reported-True set of size m , we have: $E(FPR) \sim Norm(\mu, \sigma^2)$, where $\mu = \sum_{i=1}^m (1 - \mu_i)/m$ and $\sigma^2 = \sum_{i=1}^m \sigma_i^2/m^2$. Here μ_i and σ_i take values from the above table accordingly. We can do the same for $E(FNR)$.

We are now ready to define false positive ratio (FPR) constraint and false negative ratio (FNR) constraint by using the expected FPR and FNR .

Definition 1. An FPR -constraint is in the form of a pair $\tau_{FPR} = (\theta_E, \alpha)$ ($0 \leq \theta_E, \alpha \leq 1$). A set of reported-True conditions C_T satisfies τ_{FPR} if $P\{E(FPR) \leq \theta_E\} \geq \alpha$. We call θ_E and α the *expected-mean threshold* and the *confidence threshold*, respectively.

Symmetrically, we can define FNR -constraint τ_{FNR} and FNR -quality of a reported-false set C_F .

In addition to FPR - and FNR -constraints, which provide an effective way to guarantee the *overall* evaluation quality, we allow the users to specify two constraints for each individual condition c_i such that c_i cannot be reported as true (or false) unless μ_i is greater (or less) than a given true (or false) quality threshold. For simplicity, we assume that the users use a global threshold of 0.5 for all individual constraints in this paper.

5 Optimization Algorithms

In this section, we provide optimization algorithms for the three optimization problems mentioned in the introduction. Each problem requires a different strategy. For simplicity, in all these algorithms, we assume that the precise evaluation of each condition has the same cost. Thus, the response time can be measured by the number of conditions that have been precisely evaluated (to either True or False) before the last condition in the reported-True set is reported.

As mentioned earlier, the user imposes constraints on all quality parameters except for one that is left unconstrained. The evaluation system will optimize for the unconstrained parameter while satisfying the constraints on others. For the cases that the unconstrained parameter is either *FPR* or *FNR*, we just give the algorithm for the *FNR* unconstrained case since the other one is symmetric. Thus, we study three optimization problems, namely, 1) minimize response time given accuracy requirements and no drop, 2) minimize drop ratio given response time and accuracy requirements, and 3) minimize false negative ratio given response time and false positive error requirements and no drop.

5.1 Algorithm for minimizing response time

The algorithm is shown in Fig. 1. The basic idea is to increase the size of C_T and C_F aggressively, and at the same time, try to report as early as possible those trigger conditions in C_T . We use a greedy algorithm for this purpose.

In the algorithm, we need to exam if the *FPR*-quality of a set $C_T = \{c_1, \dots, c_m\}$ satisfies a given *FPR*-constraint τ_{FPR} (using our predicted μ_i values). Since we already know that the expected value of *FPR* follows normal distribution, we can get the confidence β_T with the standard normal distribution function η :

$$\beta_T = \eta(z_T), \text{ where } \eta(x) = \frac{1}{\sqrt{2\pi}} \int_{-\infty}^x e^{-t^2/2} dt, \text{ and } z_T = \frac{\sum_{i=1}^m (\theta_E - (1 - \mu_i))}{\sqrt{\sum_{i=1}^m \sigma_i^2}}.$$

Note z_T value can be calculated incrementally.

The algorithm uses a list (called $\mu\mathbf{List}$) that contains all the conditions in C arranged in the order, say c_1, \dots, c_n , such that $\mu_1 \geq \dots \geq \mu_n$. The algorithm starts with using `InitExpand C_T` to get initial report-True set C_T . (This `InitExpand C_T` procedure obtains, without evaluating any conditions, an initial set of conditions that can be reported true without violating the user requirements.) The trigger conditions in C_T are reported to be True. Both the *FPR*-constraint τ_{FPR} and $\mu > 0.5$ are satisfied by C_T by the property of `InitExpand C_T` . These are the trigger conditions that can be reported True without doing any precise evaluation. This is Step 1.

After Step 1, we need to precisely evaluate trigger conditions in order to report them true (without violating either τ_{FPR} or $\mu > 0.5$). In Step 2, as a greedy algorithm, we pick up the condition having the highest μ value. This is the one immediately after the conditions in the initial C_T . Hence, we pick it (i.e.,

Consts.:	no drop and satisfy τ_{FPR} and τ_{FNR}
Goal:	minimize response time (RT)
Step 1.	Form and report the reported-True set: Init. $z_T = 0$, $[z_T, i_T] = \text{InitExpandC}_T(z_T)$ and report c_1, \dots, c_{i_T-1} as True.
Step 2.	Process all conditions c_i with ($\mu_i > 0.5$): Do loop until ($\mu_{i_T} \leq 0.5$) or ($i_T > n$) { - Precisely evaluate c_{i_T} . Two outcomes: o If c_{i_T} is evaluated False, update z_F with an extra reported-False condition. Continue the loop with $i_T = i_T + 1$; o If c_{i_T} is evaluated True, update z_T with an extra reported-True condition, $[z_T, \Delta_T] = \text{ExpandC}_T(z_T, i_T + 1)$, and report $c_{i_T}, \dots, c_{i_T+\Delta_T}$ as True. Continue the loop with $i_T = i_T + \Delta_T + 1$. }
Step 3.	Form the reported-False set: Init. $z_F = 0$, $[z_F, i_F] = \text{InitExpandC}_F(z_F)$
Step 4.	Process all conditions c_i with ($\mu_i \leq 0.5$): Continue to use the same i_T from Step 2, do loop until ($i_T > i_F$) { - Precisely evaluate c_{i_T} . Two outcomes: o If c_{i_T} is evaluated True, report c_{i_T} as True and continue the loop with $i_T = i_T + 1$; o If c_{i_T} is evaluated False, update z_F with an extra reported-False condition, $[z_F, \Delta_F] = \text{ExpandC}_F(z_F, i_F)$. Update $i_F = i_F - \Delta_F$ and continue the loop with $i_T = i_T + 1$. }
Step 5.	Report as False all those conditions that were not reported True.

Figure 1: Algorithm MinResponse.

c_{i_T}) up for precise evaluation. If the condition is evaluated True, we add it to C_T and try to expand C_T without precise evaluation again (by calling Procedure ExpandC_T), report the conditions in the expanded C_T and keep going. If the condition is evaluated False, then we just keep going to precisely evaluate the next condition.

During Step 2, if we run out of conditions in μList , we can stop (just report all the conditions that were evaluated False as false and thus achieve $FNR = 0$). If the μList is not exhausted, then we need to reach the first trigger condition in the μList such that its μ value is no greater than 0.5.

Once we only have conditions with μ no greater than 0.5, we need to precisely evaluate them and report them as soon as they are evaluated True. However, there is a chance we may be able to report them False. Therefore, Step 3 tries to get the maximum set of trigger conditions to report False without precise evaluation (note that all the conditions that were evaluated False need to be taken into account, hence the z_F value may not start with 0 in Step 3).

After Step 3, if we still have trigger conditions that need to be processed (i.e., if $i_T \leq i_F$), we will pick them up for evaluation. Since we want to minimize the response time for the conditions in C_T , we precisely evaluate the conditions starting from those with greater μ values. Again, if any condition is evaluated False, we will try to expand C_F .

5.2 Algorithm for minimizing drop ratio

In this optimization problem, we have a deadline to report all conditions in the reported-True set, which is exactly the limit on the total number of conditions that can be precisely evaluated. In addition, we still want both C_T and C_F to satisfy the given quality constraints (i.e., τ_{FPR} and τ_{FNR}). In this case, we want to reduce the number of conditions that are dropped, i.e., minimizing DR . Fig. 2 shows the pseudo-code for our algorithm.

Consts.:	response time θ_{RT} , τ_{FPR} and τ_{FNR}
Goal:	minimize drop ratio (DR)
Step 1.	For the reported-True set C_T : init. $z_T = 0$, $[z_T, i_T] = \text{InitExpand}C_T(z_T)$.
Step 2.	For the reported-False set C_F : init. $z_F = 0$, $[z_F, i_F] = \text{InitExpand}C_F(z_F)$.
Step 3.	Expand C_T and C_F :
3.1.	Let $z'_T = z_T$ and update z_T with an extra reported-True condition, then $[z_T, \Delta_T] = \text{Expand}C_T(z_T, i_T)$
3.2.	Let $z'_F = z_F$ and update z_F with an extra reported-False condition, then $[z_F, \Delta_F] = \text{Expand}C_F(z_F, i_F)$
3.3.	Do loop until $(\mu_{i_T + \Delta_T} \leq 0.5) \& (\mu_{i_F - \Delta_F} \geq 0.5)$ or $(i_T > i_F)$ or $(\theta_{RT}$ is reached). { - If $\Delta_T > \Delta_F$, $k = i_T + \Delta_T$, else $k = i_F - \Delta_F$; - Precisely evaluate c_k . There are two outcomes: o If c_k is evaluated True, do Step 3.1 again, then continue the loop; o If c_k is evaluated False, do Step 3.2 again, then continue the loop; }
Step 4.	Clean up the uncertain conditions: Precisely evaluate all conditions whose μ values are 0.5 until θ_{RT} is reached.
Step 5.	Report all conditions that were evaluated True or are in C_T as True. Report all conditions that were evaluated False or are in C_F as False. The remaining conditions are dropped.

Figure 2: Algorithm MinDrop.

The difference between this algorithm and **MinResponse** lies in what we emphasize on. In **MinResponse**, we want to report the trigger conditions in C_T as soon as possible (to minimize the response time). For that, we always precisely evaluate, as early as possible, trigger conditions that are most likely to be True. However, if we use the same strategy, we are not aggressively increasing the size of C_F , which may be more beneficial to decrease the number of dropped conditions. Therefore, to minimize DR , we also need to precisely evaluate, as soon as possible, the trigger conditions that are most likely to be False.

In **MinDrop**, we try to maximize C_T and C_F at the same time. Steps 1 and 2 respectively get the initial C_T and C_F without precisely evaluating any condition. In Steps 3.1 and 3.2, we obtain the potential increases of C_T and C_F if we add an additional reported-True (reported-False) condition into C_T (C_F , respectively). If C_T can increase faster, we will precisely evaluate the condition (not in C_T) that is most likely to be True. Otherwise, we precisely evaluate the

one that is most likely to be **False**. After each precise evaluation, we will again see the potential increases to C_T and C_F and repeat the process. This is Step 3.3.

We will continue Step 3 until either the response deadline is reached or all the remaining trigger conditions have μ values equal to 0.5. In the former case, we just drop all the trigger conditions that are in neither C_T nor C_F . In the latter case, we will just need to precisely evaluate all these trigger conditions until θ_{RT} is reached. This is Step 4.

5.3 Algorithm for minimizing *FNR*

Our last optimization situation is when there are a response time deadline and a quality constrain on C_T only (i.e., τ_{FPR}). Different from the second problem, this one does not allow any drop of conditions, i.e., all trigger conditions must be reported either **True** (in C_T) or **False** (in C_F). The optimization target is to minimize the false negative ratio *FNR*, and hence, there is no quality constraint on set C_F and no individual quality constraint for each condition c_i in C_F (i.e., we do not require $\mu_i < 0.5$ for each condition c_i in C_F). Fig. 3 shows this algorithm in pseudo-code.

Consts.:	response time θ_{RT} , $DR = 0$, and τ_{FPR}
Goal:	minimize false negative ratio (<i>FNR</i>)
Step 1.	Form the reported-True set C_T : init. $z_T = 0$, $[z_T, i_T] = \text{InitExpand}C_T(z_T)$.
Step 2.	Decrease the uncertainty in C_F : do the following until θ_{RT} is reached or $(i_T > n)$ or $(\mu_{i_T} \leq 0.5)$: { - Precisely evaluate c_{i_T} . There are two outcomes: o If c_{i_T} is evaluated False , continue the loop with $i_T = i_T + 1$; o If c_{i_T} is evaluated True , update z_T with an extra reported-True condition, $[z_T, \Delta_T] = \text{Expand}C_T(z_T, i_T + 1)$, and continue the loop with $i_T = i_T + \Delta_T + 1$. }
Step 3.	Report all the conditions that were evaluated True or are in C_T as True , and report the remaining conditions as False .

Figure 3: Algorithm MinFNR.

This algorithm turns out to be the simplest. Since no drop is allowed, to reduce the false negative ratio, we want to spend time on trigger conditions that is most unlikely to be **False**. So the algorithm first looks for an initial C_T . After that, the trigger condition immediately after the conditions in C_T is most unlikely to be **False**, and hence we precisely evaluate that condition. If this condition turns out to be **False**, we just pick up the next one for precise evaluation. If it turns out to be **True**, we expand C_T and repeat. When the response time deadline is reached, we just report C_F to be all conditions that are neither reported **True** (in C_T) nor precisely evaluated.

6 Experimental Results

In this section, we present our experimental results.

Data set: We generate synthetic data for the experiments. The data set consists of 100 streaming time series. Each time series is independently generated with a random walk function. For stream $s = \langle v_1, v_2, \dots \rangle$, $v_i = v_{i-1} + rand$, where $rand$ is a random variable uniformly distributed in the range of $[-0.5, 0.5]$.

Condition set: The trigger condition set includes 400 conditions defined over these 100 streams. Each condition may contain one or more correlation (or distance) functions. Each function is defined on two streams that are randomly selected from the 100 streams.

A prediction model for each condition is built based on the method in [14] on the data sets generated above. We assume that when a condition is precisely evaluated, the corresponding feature values (used for prediction) are extracted. When the prediction of a condition is required, we will look back in time to find the nearest time position when the condition was precisely evaluated. We use the extracted feature values and the prediction model to predict the probability for the condition to be true.

Performance parameters: We use the four quality parameters described in Section 3 (i.e., RT , DR , FPR and FNR) to measure the performance of our algorithms. Note that DR , FPR and FNR are all real numbers in $[0, 1]$ and can be computed precisely by comparing the reported results with the precise evaluation results (done for the purpose of performance evaluation). The response time is measured by the number of conditions that are precisely evaluated (either to **True** or **False**) before all the conditions in the reported-**True** set are reported. By using this measure (instead of using real time), we can clearly separate the overhead of the optimization procedure and the condition evaluation time.

6.1 Results for minimizing response time

This set of experiments is to assess **MinResponse** that minimizes the response time under quality constraints on C_T and C_F and no drop allowed. Here, we set the confidence threshold $\alpha = 95\%$ for both FPR - and FNR -constraints and $DR = 0$. We vary the expected-mean threshold θ_E from 0.05 to 0.3 and execute the algorithm for 1,000 time positions in each run.

Fig. 4(a) and (b) show the evaluation quality achieved in terms of actual FPR and FNR . The two plots of Fig. 4(a) present the actual FPR and FNR values at each time position for 200 time positions with $\theta_E = 0.01$ (for both τ_{FPR} and τ_{FNR}). We can see that these actual FPR (FNR) values are in the range $[0.01, 0.04]$ with a mean of 0.008 (which is very close to the given $\theta_E = 0.01$). Fig. 4(b) presents how well FPR (FNR) constraints with various mean thresholds (varying from 0.01 to 0.3) are satisfied by our algorithm. We calculate the average of the actual FPR (FNR) values over 1000 time positions for each run, and we can see that the average is either below or very close to the corresponding required expected-mean threshold θ_E for all the runs.

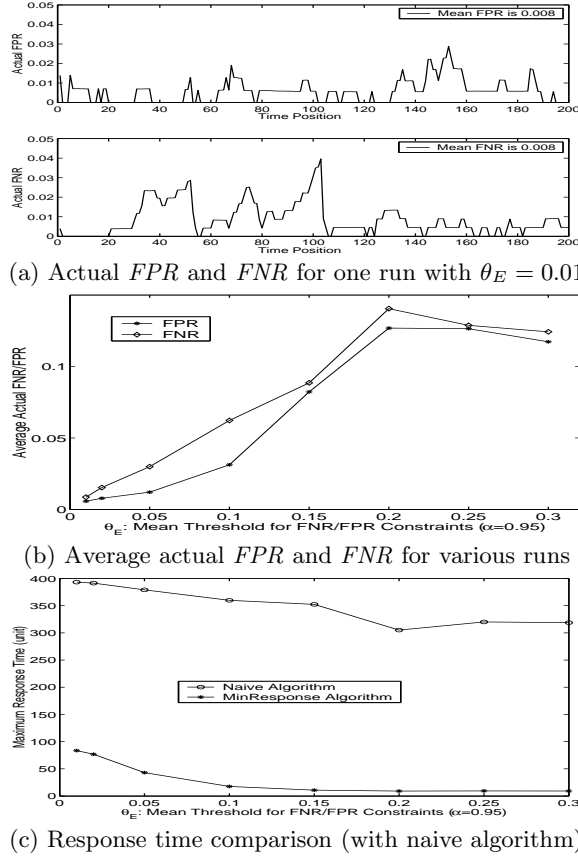
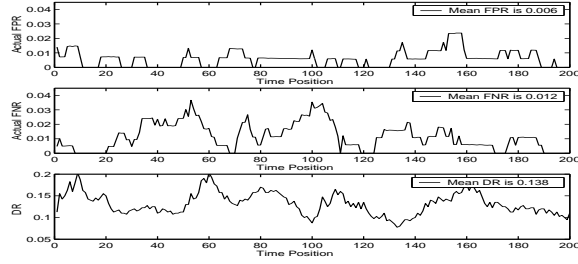


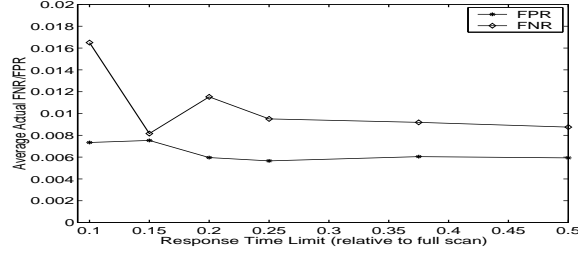
Figure 4: Quality (and performance) of `MinResponse`.

Fig. 4(c) shows the performance of `MinResponse` in terms of response time. For comparison, a naive algorithm is implemented: It randomly picks up a condition for precise evaluation, until it has reported k Trues, where k is the number of real Trues in the reported-True set from `MinResponse` (i.e., k is the number of c_i s such that $R(c_i) = \text{True}$ and $c_i \in C_T$). This is to make the naive algorithm report the same number of true conditions. We compare the response time of `MinResponse` with this naive algorithm for different runs with θ_E values in $[0.01, 0.3]$. We can see that `MinResponse` consistently outperforms the naive algorithm. Note that the response time of `MinResponse` decreases as θ_E increases, because the greater the θ_E value, the coarser approximation is allowed, and thus fewer precise evaluations are needed.

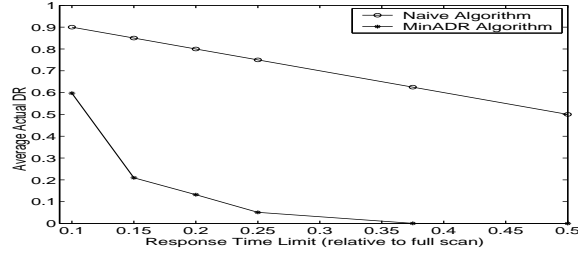
The performance gain of `MinResponse` is significant. For example, given $\theta_E = 0.01$, `MinResponse` only takes about 1/15 time of the naive algorithm, but maintains the quality of *FPR* and *FNR* at around 1%. When θ_E is set to higher values, the performance gain becomes more significant.



(a) Actual FPR , FNR , and DR for one run with response time limit set to 20% of full scan



(b) Average actual FPR and FNR for various runs



(c) Average drop ratio comparison (with naive algorithm)

Figure 5: Quality and Performance of MinDrop.

6.2 Results for minimizing drop ratio

This set of experiments is to assess the performance of Algorithm `MinDrop`, which aims at minimizing the drop ratio under quality constraints for both C_T and C_F and a response time limit (deadline). Here, we set $\theta_E = 0.01$ and $\alpha = 0.95$ (for both τ_{FPR} and τ_{FNR}), and vary the response time limit θ_{RT} for different runs over 1000 time positions.

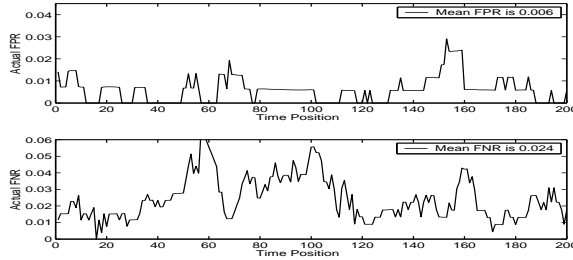
Fig. 5(a) shows the results from one run in detail. In this run, θ_{RT} is set to be equivalent to 20% of the time for a full scan. More precisely, since we have a total of 400 conditions, θ_{RT} is set to be the time to precisely evaluate 80 conditions. We can see from Fig. 5(a) that `MinDrop` achieves the quality constraints very well (top two plots of Fig. 5(a)) with an average drop ratio of 14% (bottom plot of Fig. 5(a)). That means among the 400 conditions, only about 55 conditions are dropped on average.

Fig. 5(b) presents how the FPR (FNR) constraints are satisfied in various runs with different deadlines ranging from 10% to 50% of the full scan time.

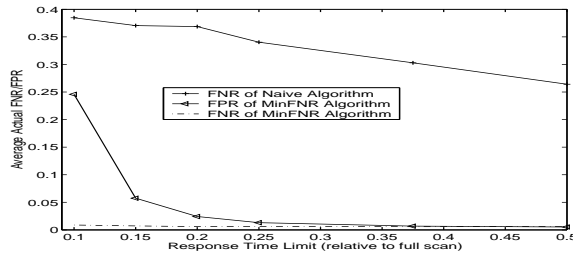
The average actual FPR (FNR) is close to the required mean threshold (0.01) in most cases. Of course, the tougher the deadline (i.e., smaller θ_{RT} value), the greater the actual FPR (FNR) is.

Fig. 5(c) compares the drop ratio of **MinDrop** with a naive algorithm, which precisely evaluates all conditions in a random order until the deadline is reached. We can see that **MinDrop** out-performs the naive algorithm significantly.

6.3 Results for minimizing FNR



(a) Actual FPR and FNR for one run with response time limit set to 20% of full scan



(b) Actual FNR comparison (with naive algorithm)

Figure 6: Quality and Performance of **MinFNR**.

This set of experiments is to assess the performance of Algorithm **MinFNR**, which aims at minimizing FNR under a quality constraint for C_T and a response time limit (deadline) when no drop is allowed. Here, we set $\theta_E = 0.01$ and $\alpha = 0.95$ for τ_{FPR} , and vary the response time limit θ_{RT} for different runs over 1000 time positions.

Fig. 6(a) presents the results from one run in detail. In this run, θ_{RT} is set to be 20% of full scan (same as the experiment setting in the previous subsection). We can see that **MinFNR** achieves very high quality for C_F with a mean FNR of 0.024 (bottom plot of Fig. 6(a)) while satisfying the quality constraint on C_T very well with a mean FPR of 0.006 (top plot of Fig. 6(a)).

Fig. 6(b) compares **MinFNR** with the naive algorithm described in the previous subsection. We can see that **MinFNR** provides much better FNR quality (i.e., smaller FNR value) than the naive algorithm. Also, when more time is allowed (i.e., greater θ_{RT} values), the FNR achieved by **MinFNR** decreases very quickly.

7 Conclusion

In this paper, we proposed a framework for designing trigger condition evaluation system that considers user-specified quality requirements. We used statistical analysis to derive the likelihood of a condition to be true at a time position. By using this likelihood and the associated confidence (due to finite sampling), we estimated the quality of our approximate answers. Based on this prediction method, we designed algorithms for three different optimization problems. Our experiments showed that these algorithms are effective in reaching corresponding optimization goals.

It will be interesting to see how our quality-driven strategy works in different settings. For example, we may define the optimization goal based on a global function computed from several quality measures, or we may form complex trigger conditions based on a few consecutive basic windows.

References

1. D. J. Abadi et al. Aurora: A data stream management system. In *SIGMOD*, page 666, 2003.
2. S. Arya, D. M. Mount, N. S. Netanyahu, R. Silverman, and A. Y. Wu. An optimal algorithm for approximate nearest neighbor searching fixed dimensions. *Journal of the ACM*, 45(6):891–923, 1998.
3. B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *SIGMOD*, pages 253–264, 2003.
4. D. Carney et al. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.
5. CISCO. Quality of Service (QoS). On-line. http://www.cisco.com/univercd/cc/td/doc/cisintwk/ito_doc/qos.htm, 2003.
6. A. Das, J. Gehrke, and M. Riedewald. Approximate join processing over data streams. In *SIGMOD*, pages 40–51, 2003.
7. A. Dobra, M. N. Garofalakis, J. Gehrke, and R. Rastogi. Processing complex aggregate queries over data streams. In *SIGMOD*, pages 61–72, 2002.
8. P. Ferguson and G. Huston. *Quality of Service: Delivering QoS on the Internet and in Corporate Networks*. John Wiley & Sons, 1998.
9. S. Ganguly, M. N. Garofalakis, and R. Rastogi. Processing set expressions over continuous update streams. In *SIGMOD*, pages 265–276, 2003.
10. L. Gao, M. Wang, X. S. Wang, and S. Padmanabhan. A learning-based approach to estimate statistics of operators in continuous queries: a case study. In *DMKD*, pages 66–72, 2003.
11. A. C. Gilbert, Y. Kotidis, S. Muthukrishnan, and M. Strauss. Surfing wavelets on streams: One-pass summaries for approximate aggregate queries. In *VLDB*, pages 79–88, 2001.
12. D. McDysan. *QoS and Traffic Management in IP and ATM Networks*. McGraw-Hill Osborne Media, 1999.
13. N. Tatbul, U. etintemel, S. B. Zdonik, M. Cherniack, and M. Stonebraker. Load shedding in a data stream manager. In *VLDB*, pages 309–320, 2003.
14. X. S. Wang, L. Gao, and M. Wang. Condition evaluation for speculative systems: a streaming time series case. In *STDBM*, pages 65–72, 2004.