Ratio Threshold Queries over Distributed Data Sources

Rajeev Gupta *
IBM Research, India
grajeev@in.ibm.com

Krithi Ramamritham
Indian Institute of Technology, Mumbai, India
krithi@cse.iitb.ac.in

Mukesh Mohania
IBM Research, India
mkmukesh@in.ibm.com

ABSTRACT

Continuous aggregation queries over dynamic data are used for real time decision making and timely business intelligence. In this paper we consider queries where a client wants to be notified if the ratio of two aggregates over distributed data crosses a specified threshold. Consider these scenarios: a mechanism designed to defend against distributed denial of service attacks may be triggered when the fraction of packets arriving to a subnet is more than 5% of the total packets; or a distributed store chain withdraws its discount on luxury goods when sales of luxury goods constitute more than 20% of the overall sales. The challenge in executing such *ratio threshold queries* (RTQs) lies in incurring the minimal amount of communication necessary for propagation of updates from data sources to the aggregator node where the client query is executed. We address this challenge by proposing schemes for converting the client ratio threshold condition into conditions on individual distributed data sources. Whenever the condition associated with a source is violated, the source *pushes* its data values to the aggregator, which in turn *pulls* data values from other sources to determine whether the client threshold condition is indeed violated. We present algorithms to minimize the number of source condition violations (i.e., the number of pushes) while ensuring that no violation of the client threshold condition is missed. Further, in case of a source condition violation, we propose efficient selective pulling algorithms for intelligently choosing additional sources whose data should be pulled by the aggregator. Using performance evaluation on synthetic and real traces of data updates we show that our algorithms result in up to an order of magnitude less number of messages compared to existing approaches in the literature.

1. INTRODUCTION

Continuous aggregation queries over dynamic data are used in various scenarios for real time decision making and timely business intelligence. The aggregation function can take the form of addition, average, min, max [4,9], top-k [1], frequency measures [13], etc. In this paper we identify and focus on *ratio threshold queries* (RTQs) where a client wants to be notified if the ratio of two aggregates over distributed data crosses a specified threshold. In this scenario the client poses the ratio threshold query at a proxy or data aggregator which obtains the required data from geographically distributed data sources and executes the query.

1.1 Motivation behind RTQs

We describe a few practical scenarios where RTQ arise:
Example1: A store chain selling various items belonging to different categories like electronics, food, health-care, etc., integrates data from its inventory, marketing, and sales databases from different geographies for monitoring sales. During a sales campaign, the manager of the store chain wants to get notified whenever the sale of electronic goods for the day is more than 20% of the total sale so that, say, the discount percentage on electronics goods can be reduced. To know whether the 20% threshold is crossed, usually a multi-pass SQL query needs to be executed frequently, resulting in numerous data transfers, many of which are likely to leave the query result unchanged.
Example2: Various methods for defending against distributed denial of service (DDOS) attacks are reported in the literature including congestion based, anomaly based, source based, etc. In the congestion based method [6] a specific destination address is marked if the fraction of packets arriving at that address in a specified duration is above a threshold. As packets are handled by various edge routers, a distributed query can be launched to know whether the fraction has crossed the specified threshold for a particular recipient or subnet.
Example3: To incentivize residents to reduce energy consumption, we may want to send alerts to houses in a particular building whenever the average power consumption of a building is 5% above the overall average of a campus. This can be done by collecting energy consumption data from each home to get average energy consumption of each building as well as that of whole campus.

1.2 Definition of RTQs

As shown in Figure 1, let there be $|S|$ data sources whose data is aggregated at the data aggregator. We assume that data sources are independent and do not directly communicate with each other. An RTQ specified by the client has numerator (n) and denominator (d) data items along with a ratio threshold (μ) whose violation (i.e., the ratio becoming

*This work was done as part of PhD at IIT Mumbai.

Proceedings of the VLDB Endowment, Vol. 6, No. 8

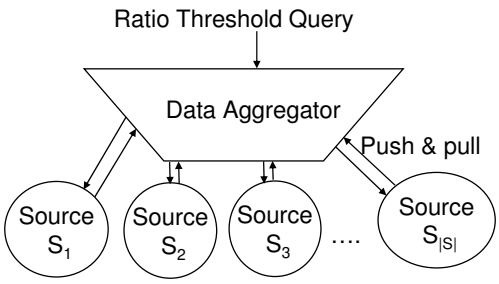

Figure 1: Data aggregator and sources.

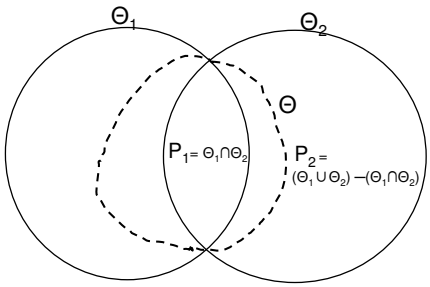

Figure 2: Deriving source conditions

larger (or smaller) than μ) should be notified to the client. At each data source S_i, data values at various time instances are aggregated to get numerator and denominator data values, n_i and d_i, respectively. These data values are further aggregated across distributed data sources to get the numerator and denominator over which the ratio threshold is specified. Specifically, the client threshold (μ) is specified over the ratio of the sum of n_is, denoted by N, and the sum of d_is, denoted by D. That is, the client is not notified as long as the following condition on the ratio N/D, called the *global ratio*, is satisfied:

$$\frac{N}{D} = \frac{\sum_{i=1}^{|S|} n_i}{\sum_{i=1}^{|S|} d_i} \leq \mu \quad (1)$$

In *Example1* the numerator is the dollar sales of electronics goods and the denominator is dollar sales of all items. Ratio threshold is 20%. Total dollar sales of electronics goods can be obtained by summing dollar sales of electronics from each source. The dollar sales of electronics at each source is obtained by adding dollar values of sales of all electronics goods at that source. The goal of this paper is to execute such ratio threshold queries with minimum communication between data sources and the data aggregator while ensuring that no instance of the client threshold violation is missed.

1.3 Categorizing RTQs

RTQs can be categorized along the following dimensions:

1. **Type of temporal aggregation**: An RTQ can have either start time aggregation (STA) or window based aggregation (WBA). In STA, the user requires RTQ to be executed using the data gathered from the start of the system (landmark window), e.g., *daily sales* = 0 when the store opens and it is likely to increase as further sales occur during the day. In case of STA, n_is and d_is are non-decreasing functions of time. In WBA, the RTQ is executed over the data gathered within a moving time window as in *Example2*.

2. **Type of threshold**: RTQ can have either upper bound threshold (UBT) or lower bound threshold (LBT). In case of UBT (LBT), the client desires notification only when the ratio N/D is greater (smaller) than the specified upper (lower) bound μ. In either case, we say that the threshold has been violated.

Although our work considers both the types of temporal aggregations and thresholds, for ease of exposition, in this paper, as a default we assume that the client wants notification when the ratio N/D goes above μ where N and D are obtained using start time aggregation (i.e., STA-UBT).

1.4 Executing RTQs

To check whether the global ratio has crossed the client specified threshold the aggregator needs the current values of the distributed data items. We have the option of evaluating the client query either by getting all the data at the data aggregator (data shipping), or by sending queries derived from the client query to all the data sources (query shipping). Data shipping is not a preferred option for continuously changing data items due to the high volume of the resulting data transfers. Thus, from the condition ($N/D \leq \mu$ as given by Equation 1), defined as *client condition* C, whose violation requires the aggregator to send notification to the client, we derive *source conditions* C_is (involving n_i and d_i) for each source i. Each source S_i needs to push data values to the data aggregator whenever C_i is violated. In response to a push message from a source, the aggregator judiciously pulls data from the other sources to evaluate the condition given by Equation 1, and if it is violated, the notification is sent to the client.

1.5 Deriving C_is from C: Correctness and Performance Objectives

The derivation of C_is from C is challenging because *all* threshold violations should be detected with the least communication overheads – resulting from pushes by sources to the aggregator and pulls by the aggregator. We explain these objectives formally now. We denote the set of time instances when C_i (C) is violated as Θ_i (Θ). Figure 2 shows an example scenario for two data sources where the dashed contour indicates Θ.

We have two considerations while deriving C_is from C:

1.5.1 Correctness objective

To ensure that no client threshold condition violation is missed (i.e., zero false negatives), we need to ensure that client's condition C is never violated without violation of at-least one of the source conditions (as aggregator pulls data and verifies C only when at-least one of the C_is is violated). This *correctness objective* can be denoted as:

$$\Theta \subseteq \cup\Theta_i \quad (2)$$

1.5.2 Performance objectives

To minimize the number of pushes from a data source, we aim to minimize the source condition violations ($\cup\Theta_i$) when

global ratio threshold violation (Θ) is unlikely. Thus, as the *first performance objective*, we need to minimize $|\cup\Theta_i - \Theta|$. As shown in Figure 2 we can divide $\cup\Theta_i$ into two parts:
1) $P_1 \equiv \cap\Theta_i$
2) $P_2 \equiv \cup\Theta_i - \cap\Theta_i$.
For minimizing pushes, we should be deriving source conditions, C_is, such that:

a There should not be any push resulting from P_1 without leading to a notification to the client, i.e., if all sources send the push message at a certain instance, it should imply the violation of Equation 1. If that is not the case, we can change the source conditions leading to lesser number of pushes without introducing any false negative.

b Number of push messages from P_2 is minimized.

Ideally, $\Theta \equiv \cap\Theta_i$ (ensuring $C_1 \wedge C_2 \wedge ... \wedge C_{|S|} \Rightarrow C$), i.e., we aim for the situation where the client condition is violated only if all the source conditions are violated. In a centralized setting, this will ensure that a source pushes only if the client condition is violated. But in the distributed scenario, although we aim for $\Theta \equiv \cap\Theta_i$, in practice $\cap\Theta_i \subseteq \Theta \subseteq \cup\Theta_i$. Specifically, we assign source conditions C_is such that when more C_is are violated, the client condition violation is more probable and when all C_is are violated, the client condition is indeed violated.

When the data aggregator receives a push message from any source, to determine whether the client threshold condition is violated, the data aggregator needs to pull data from other sources. Our *second performance objective* is to minimize the number of pull messages. In cases where push and pull costs are different for different sources (say, based on the number of hops from the aggregator), rather than minimizing the number of push and pull messages we should be minimizing the total push and pull costs.

1.5.3 Guiding Principles

We use the following principles to help achieve these objectives:

1. Each source condition should be associated with tuning parameters such that the aggregator can tune them based on last pulled values. These tuning parameters can be used to assign source conditions such that they are likely to be violated at the same time.

2. We should aim to minimize the number of conditions assigned to each source, as the larger the number of conditions, the larger the expected source condition violations ($|\Theta_i|$) and more difficult it would be to tune these conditions.

1.6 Contributions and Outline

In this paper, we introduce ratio threshold queries (RTQs) with their variations and present techniques for efficiently executing them. A review of related work in the areas of continuous queries and distributed and efficient execution of such queries (see Section 2) leads us to believe that ours is the first work specifically considering ratio threshold queries over distributed data sources while minimizing the number of source conditions violations (push messages) and the pulls required by the data aggregator. The closest related work [12] is based on a geometric approach to monitor threshold functions over distributed data values. We show that compared to our approach, the method based on [12] results in an order of magnitude more messages between the data sources and the aggregator.

Our contributions are:

- In Section 3 we develop a suite of algorithms to map the global client condition (C) into local source conditions (C_is) at individual data sources so that no client condition violation is missed (guaranteeing zero false negatives) while minimizing the number of push messages from the sources. While all these algorithms guarantee *correctness*, they have different characteristics and *performance*.

- Through performance evaluations using synthetic and real dynamic data traces, in Section 4, we demonstrate that our algorithms requires an order of magnitude lesser number of push messages compared to generic algorithms proposed in the literature [12].

- When any source pushes a message to the data aggregator, we show that it is not necessary to pull data from all the sources. Which data to pull is a question that needs to be addressed to reduce the number of pull messages. A judicious choice is made by the selective pulling schemes developed in Section 5.

- If the data sources have different push and pull costs, then we need to change (a) the way the source conditions are derived, and (b) the method to select the data sources to pull, to minimize the total push and pull costs. This is elaborated in Section 6.

- In Section 7, we consider the case when multiple queries with overlapping data items are executed at a data aggregator. We show how push and pull messages can be shared across queries.

- Conclusion and avenues for future work are the subject of Section 8.

2. RELATED WORK

The problem of answering a continuous query over distributed data sources has been studied extensively in the literature. In these queries data from multiple independent sources need to be aggregated at the data aggregator. These queries can be categorized based on the type of results desired by the user: *value based queries*, *entity based queries*, and *Boolean queries*. For these queries, various methods are proposed to reduce communication among the data sources, aggregators, and users. An imprecision bound may be specified for the value and entity based queries where the user is interested in the new value of the query only if it changes by more than the specified imprecision bound. Similarly, the threshold specified for the Boolean queries is used to reduce the number of refresh messages. In this paper, we use the threshold and the nature of aggregation to reduce the number of push and pull messages between the data sources and the aggregator.

In *value based queries* [3, 4, 7, 9, 13], the user may want to know values of individual data items or the result of some aggregation of the data items. For reducing the number of push messages from data sources to the data aggregator, authors of [9] proposed a method to divide the user specified

imprecision bound into imprecision bounds for individual data items. In [4] authors propose dividing the imprecision bound among sub-queries to be executed at distributed data aggregators to reduce the number of push messages from the data aggregators to the users.

In *entity based queries* [1, 2] the user wants to know the set of data items satisfying a certain selection condition. Authors of [2] consider range and rank based selection functions for the *entity based queries*. In a range query, the user is interested in knowing the entities which are in the user specified range of values whereas in a rank query, the user wants to know the *k-nearest neighbor* entities at a specific point in the data space. Authors use the tolerable false positives and false negatives to reduce the push messages from the data sources. In [1], authors present algorithms for distributed top-k monitoring with an imprecision bound. Their algorithm employs a coordinator which determines the initial set of top-k identifiers. The coordinator sends each node a set of constraints such that the list of top-k is guaranteed to remain unchanged unless one or more of constraints are violated. Our approach has one aspect common with these approaches: we also divide client's threshold condition into local source conditions for individual data sources. But, unlike other works, we are considering the situation where a client threshold is specified over the ratio of two aggregations.

In the third type of queries, *Boolean queries*, the user just wants to know whether a Boolean condition, expressed as a threshold over an aggregation of data items, is true [8, 12]. Authors of [12] present a geometric approach to monitor threshold functions over distributed data sources. In their setting each node collects a vector of real numbers derived from its data values; a global statistics vector is defined as the weighted average of vectors from the individual nodes; and a client is interested to know whether value of a given function over global statistics vector is above (or below) a given threshold. Even though their work is generic and can be adapted for ratio threshold queries, performance studies show that our algorithms outperform their approach for RTQs by an order of magnitude.

3. ALGORITHMS FOR DERIVING SOURCE CONDITIONS

In this section, we present algorithms for deriving conditions for each data source for a given RTQ. We assume that when a source condition is violated, the status of the condition does not change until the aggregator evaluates the condition using data from the sources. We assign source conditions based on the objectives and guiding principles outlined in Section 1.5.

Figure 3 shows aggregated values of numerator (N) and denominator (D) data items in a two dimensional space. As per Equation 1, the data aggregator need not send notifications to client as long as aggregated values of denominator and numerator are below the black continuous line with slope μ. We use this assertion to derive local conditions, i.e., conditions when the individual data sources need not send any notification to the data aggregator. Derivation is approached from different interpretations of Figure 3, leading to a variety of algorithms. All these algorithms are mathematically equivalent to Equation 1 (hence, satisfy the *correctness objective*), but, as we show in this section,

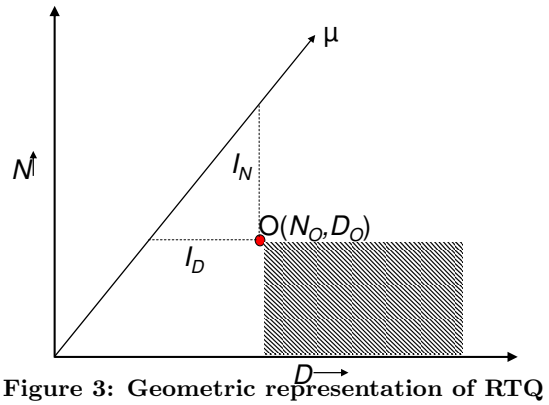

Figure 3: Geometric representation of RTQ

Table 1: Symbols used in the paper

Symbol	Description
$N_o(D_o)$	Aggregated numerator (denominator) data at point O
$n_i(d_i)$	Numerator(denominator) data item at i^{th} source
$N_i(D_i)$	Numerator (Denominator) value thresh. at i^{th} source
$D_i^u(D_i^l)$	Upper (lower) threshold for ratio algorithm
g	Value of global ratio
I_{Ni}	Value of numerator intercept at i^{th} data source
T_{Ni}	Value of intercept threshold
f_{i0}	Current value of fraction difference

they are different from a realization point of view, leading to different network overheads. Table 1 defines symbols used to describe these algorithms.

3.1 Value Algorithm

For any given point O below the black continuous line in Figure 3, we can have source conditions in the form of thresholds over n_is and d_is such that their aggregated values are below and to the right of O (i.e., in the shaded region). Thus, satisfaction of Equation 1 can be guaranteed if we can select any $O(N_o, D_o)$ such that: (a) N is less than or equal to numerator threshold, N_o; (b) D is greater than or equal to denominator threshold, D_o; and (c) ratio N_o/D_o equals the client threshold, i.e.,

$$\sum_{i=1}^{|S|} n_i \leq N_o;\ \sum_{i=1}^{|S|} d_i \geq D_o \text{ and } \frac{N_o}{D_o} = \mu \qquad (3)$$

We can ensure zero false negatives by assigning a source condition of the form $(n_i \leq N_i) \wedge (d_i \geq D_i)$ to each data source such that $\sum_{i=1}^{|S|} D_i = D_o$ and $\sum_{i=1}^{|S|} N_i = \mu D_o$. We need to find suitable values of N_is and D_is in such a way that the number of push messages can be minimized. For *start time aggregation*(STA), since both numerator and denominator are non-decreasing, we set D_is as the current denominator values (say, d_{i0}) for all the sources. That gives values of $D_o(= \sum D_i)$ and $N_o(= \mu D_o)$. We divide N_o among individual data sources in proportion to the rate of change of numerator values so that all the source conditions are likely to be violated at the same time. For *window based aggregation* (WBA), we assign D_i to be λd_{i0} for $0 \leq \lambda \leq 1$. Then we use the method outlined above to assign conditions of the form $(n_i \leq N_i) \wedge (d_i \geq D_i)$ to individual data sources. Reader is referred to [10] for further details of this algorithm.

Since the client is interested in the ratio of numerator and denominator data items, rather than their individual values, next we consider an algorithm which sets a threshold on their ratios.

3.2 Ratio Algorithm

For the global ratio to be greater than μ, at least one of the local ratios (n_i/d_i) has to be greater than μ. Thus, the source condition at source S_i can be:

$$\frac{n_i}{d_i} \leq \mu \tag{4}$$

It is easy to see that $n_i/d_i \leq \mu$ is a sufficient condition for ensuring zero false negatives. This solution is simple to understand and implement (with no parameter setting) but it has a drawback. For instance, in *Example1* if one store is selling lots of electronics items and very few other items, that store will push lots of messages to the data aggregator. Next we show that if we permit the ratio thresholds for a few data sources to be more than the client threshold μ these push messages can be avoided.

3.3 Modified Ratio Algorithm

It should be noted that if a data source has a ratio threshold greater than μ, we cannot always guarantee zero false negatives (as the data values at any source may not cross their thresholds when the global ratio crosses μ). Let us illustrate that with the help of an example with two data sources S_0 and S_1. We can assign a threshold $\mu_1 > \mu$ to the source S_1 only if the ratio at the other source $\mu_0 < \mu$. It is easy to see that if we assign threshold μ_1 ($> \mu$) to the source S_1, we can ensure zero false negatives only till its denominator value d_1 is less than or equal to some threshold D_1^u (depending on data values at the other source). Thus the local source conditions for the sources S_0 and S_1 can be written as:

$$\begin{aligned} S_1 &: (\frac{n_1}{d_1} \leq \mu_1) \wedge (d_1 \leq D_1^u) \\ S_0 &: (\frac{n_0}{d_0} \leq \mu_0) \wedge (d_0 \geq D_0^l) \end{aligned} \tag{5}$$

where $D_1^u(D_0^l)$ is the upper (lower) bound on denominator value of the source $S_1(S_0)$. Thus, in general, for ensuring zero false negatives, data sources need to have source conditions involving ratio thresholds (μ_i) and value thresholds (D_i^u or D_i^l). Thus,

$$\frac{\sum n_i}{\sum d_i} \leq \frac{\sum \mu_i d_i}{\sum d_i} \leq \frac{\sum_{i \in G_H} \mu_i D_i^u + \sum_{i \in G_L} \mu_i D_i^l}{\sum_{i \in G_H} D_i^u + \sum_{i \in G_L} D_i^l} \tag{6}$$

where G_H and G_L represent the set of sources having $\mu_i > \mu$ and $\mu_i \leq \mu$, respectively. The above equation can be understood by considering the global ratio as the weighted average of local ratios, and the fact that the weighted average increases if the weight of the higher local ratios ($i \in G_H$) increases or the weight of the lower local ratios ($i \in G_L$) decreases. Since value of *global ratio* should be less than μ, validity for a given set of local *ratio thresholds* and *value thresholds* can be checked using:

$$ratio\_bound = \frac{\sum_{i \in G_H} \mu_i D_i^u + \sum_{i \in G_L} \mu_i D_i^l}{\sum_{i \in G_H} D_i^u + \sum_{i \in G_L} D_i^l} \leq \mu \tag{7}$$

We use $ratio\_bound = \mu$ to assign value and ratio thresholds to data sources to minimize the number of push messages. First, the data aggregator pulls current values of data items from all the data sources and calculates current local ratios for individual data sources ($r_i = n_i/d_i$) and the global ratio ($g = \sum n_i / \sum d_i$). Local ratio thresholds to the data sources are assigned as $\mu_i = r_i + \frac{\alpha(\mu - g)}{|S|}$; where $|S|$ is the number of data sources and α is *adjustment factor* for adjusting the ratio and value thresholds for data sources. In Equation 7, $ratio\_bound = \mu$ is used to get a relation among value thresholds as:

$$\sum_{i \in G_H} (\mu_i - \mu) D_i^u \leq \sum_{i \in G_L} (\mu - \mu_i) D_i^l \tag{8}$$

For STA, *lower value thresholds* for the sources in G_L can be the same as the current value of the denominator d_{i0}. The *upper value thresholds* (for data sources in G_H) are assigned such that $(D_i^u - d_{i0})$, for i^{th} source, is proportional to rate of change of d_i, i.e., assuming that if d_is keep changing as per their past rate, all *source conditions* will be violated simultaneously. For WBA, we use a similar technique except that the *lower value thresholds* for the sources in G_L are assigned as λd_{i0} ($0 \leq \lambda \leq 1$). Through experimentation, we found that performance of this algorithm is a strong function of α and, as a thumb rule, value of α should be $|S|/2$. Similarly, a value of λ between 0.7 and 0.9 is observed to result in good performance.

In the *value* and *ratio* algorithms, multiple conditions are required to be maintained for each source. To reduce the number of source conditions, we next consider intercept based approaches.

3.4 Intercept Algorithms

As shown in Figure 3, any point $O(N_o, D_o)$ below the black arrow (μ line) can be represented with the help of *numerator intercept* (I_N) or *denominator intercept* (I_D) as $N_o = \mu D_o + I_N$ and $D_o = N_o/\mu + I_D$. As long as $I_N(I_D)$ is negative (positive) the data aggregator need not send any notification to the client. This *numerator* (*denominator*) *intercept* can be divided among data sources to get source conditions of the form $n_i \leq \mu d_i + T_{Ni}$ (or, $d_i \leq n_i/\mu + T_{Di}$); where T_{Ni} (T_{Di}) is the threshold over numerator (denominator) intercept for the i^{th} source. To understand the formulation of *intercept algorithms*, we rearrange Equation 1 to get:

$$\sum_{i=1}^{|S|} (n_i - \mu d_i) \leq 0 \Rightarrow \sum_{i=1}^{|S|} I_{Ni} \leq 0 \tag{9}$$

Thus, the condition given by Equation 1 can be converted into a set of local thresholds (T_{Ni}s) over the *numerator intercept* (I_{Ni}s) at individual data sources (similar formulation is possible for the *denominator intercept* also). To ensure zero false negatives, sum of these local thresholds should be 0 or less. Using the objectives outlined in Section 1.5, we present various methods to assign these thresholds.

3.4.1 Threshold depending on rate of change of intercept

In this algorithm the value of the threshold for the i^{th} source depends on current rate of change of I_{Ni}. Specifically, the difference between the threshold T_{Ni} for a particular source minus its current value of I_{Ni}, denoted by I_{Ni0},

should be proportional to the rate of change of I_{Ni}, denoted by γ_i, i.e.,

$$(T_{Ni} - I_{Ni0}) \propto \gamma_i$$

The proportionality constant, a positive value k_r, should be obtained to ensure zero false negatives. The value of k_r can be determined by setting the sum total of thresholds for all the sources, as given by Equation 9, to be less than or equal to zero. It can be seen that, if we assign $T_{Ni} = I_{Ni0} + k_r\gamma_i$ for sources having negative values of γ_i, threshold value will be less than I_{Ni0} which is not acceptable (source condition gets violated immediately after new threshold assignment). Thus, we use the following modifications to the approach: For sources having negative value of γ_i we keep $T_{Ni} = I_{Ni0}/\beta$ $(0 < \beta < 1)$ whereas for sources having non-negative γ_i we keep $T_{Ni} = I_{Ni0} + k_r\gamma_i$ such that the sum of all thresholds is zero.

3.4.2 Thresholds in proportion to fraction difference

In this algorithm we use the following intuition to assign thresholds: value of T_{Ni} should be more for a source which is contributing more to the total numerator value (N) and it should be less for a source contributing more to the total denominator value (D). Using this intuition T_{Ni}s should be assigned in proportion to the fraction a source is contributing to N minus the fraction a source is contributing to D, i.e.,

$$I_{Ni} \leq T_{Ni} = k_f f_{i0} = k_f\left(\frac{n_{i0}}{\sum_j n_{j0}} - \frac{d_{i0}}{\sum_j d_{j0}}\right) \tag{10}$$

where f_{io} is the fraction difference for the i^{th} source at the last pulling instance. This equation ensures that the sum of thresholds will be zero irrespective of the value of the proportionality constant k_f, thus ensuring zero false negatives. It should be noted that the value of k_f should be positive. At any pulling instance, since data aggregator knows the current values of data items, it can ensure that conditions specified by Equation 10 are satisfied for the current data values for all the sources, i.e.,

$$I_{Ni0} \leq k_f\left(\frac{n_{i0}}{\sum_j n_{j0}} - \frac{d_{i0}}{\sum_j d_{j0}}\right) \tag{11}$$

In Equation 11 the only unknown quantity is k_f. We present an algorithm to get value of k_f such that Equation 11 is satisfied for every data source, using the following theorems. First we prove that one can derive upper and lower bounds on k_f using Equation 11, and then give method to calculate k_f using these bounds.

THEOREM 1. *There exists a finite upper bound (k_{fupper}) on k_f.*

PROOF. As $\sum f_{i0} = 0$, there has to be some sources having positive values of f_{i0} while some having negative values of f_{i0} (or, all f_{i0}s are zero, which implies all the sources have the same ratio, i.e., value of k_f does not matter). If $f_{i0} < 0$, then it can be proved that either $I_{Ni0} < 0$ or *global ratio* $\geq \mu$. As we are assuming that the client condition is not violated, $f_{i0} < 0 \Rightarrow I_{Ni0} < 0$. For each source with $f_{i0} < 0$ and $I_{Ni0} < 0$ Equation 11 can be used to get upper bounds on k_f (as $k_f \leq |I_{Ni0}|/|f_{i0}|$). We use minimum of these, denoted by k_{fupper}, as the upper bound on k_f. □

THEOREM 2. *There exists a finite lower bound (k_{flower}) on k_f.*

PROOF. We have already proven the existence of sources having $f_{i0} > 0$. If none of the sources have $I_{Ni0} > 0$, we will have all sources having their local ratios less than μ. Hence, for a non-trivial case, we will have at least one source having positive I_{Ni0}. We get the lower bounds on k_f (as I_{Ni0}/f_{i0}) for each of such sources. Maximum of those, denoted by k_{flower}, is the lower bound on k_f. □

THEOREM 3. $k_{fupper} \geq k_{flower}$.

PROOF. We calculate k_{fupper} as minimum value of $\frac{|I_{Ni0}|}{|f_{i0}|}$ over all sources having $f_{i0} < 0$ (i.e., $\frac{n_{io}}{d_{io}} < \frac{\sum_j n_{j0}}{\sum_j d_{j0}} \leq \mu$). Using which we can show that $k_{fupper} \geq \mu \sum d_{i0}$. Similarly one can show $k_{flower} \leq \sum n_{i0}$. Since the client condition has not violated, $k_{fupper} \geq k_{flower}$. □

There may be cases where f_{i0} is positive and I_{Ni0} is negative, but those cases will not be deciding k_{fupper} or k_{flower} (as these will lead to very low lower bounds and no upper bounds). Any value of k_f between k_{fupper} and k_{flower} will satisfy Equation 11 for all the sources, but we need to select a value such that the number of push messages can be minimized. We use *bound parameter* θ to select the appropriate value of k_f such that:

$$k_f = \theta k_{fupper} + (1 - \theta)k_{flower} \tag{12}$$

θ can be between 0 and 1. In the next section, we examine the number of push messages for various values of θ to decide the most appropriate values for different categories of RTQs.

3.4.3 Time varying thresholds

From Equation 9 it is clear that we can assign any value of threshold to the intercepts at individual data sources as long as their sum is zero. At individual sources if we assign threshold values such that they are always slightly higher than their corresponding intercept values, the number of push messages can be minimized. Since the data aggregator refreshes values of data items only when it pulls them, using these values the aggregator should assign thresholds as a function of time so that the data sources can calculate the threshold values themselves. In general we can extend Equation 10 to assign thresholds as polynomial of time τ:

$$I_{Ni} \leq k_f f_{i0} + \sum_l C_{il}\tau^l \tag{13}$$

where the threshold value is a l^{th} order polynomial of time τ such that $\sum_i C_{il} = 0$ for all values of l (to ensure that the sum of thresholds is zero all the time). Specifically, we can use methods proposed in the previous section to assign the fixed part of threshold and use trends in historical data values to assign the time varying part of the threshold. In the next section, we use a first order polynomial to assign intercept thresholds for measuring performance of this algorithm.

3.5 Geometric Algorithm

Besides the algorithms proposed in this section, we also implemented the coordinator based solution proposed in [12], specifically for RTQs. In this algorithm, each source creates a vector of real numbers $\{n_i, d_i\}$. The current values of these vectors are used to create a *global statistics vector*, $\{N, D\}$, defined as the sum of vectors from the individual nodes. Except when all the data items are pulled from the sources,

the value of the *global statistics vector* is not known at any of the nodes (including DA). *Global estimate vector* is the estimate of the *global statistics vector* based on last known values of $\{n_i, d_i\}$. Individual data sources calculate the difference between the current values of $\{n_i, d_i\}$ and the values sent to the aggregator at the last pull (or push) instance. This difference vector is added to the *global estimate vector* to calculate a drift vector at each data source independently. It should be noted that as long as the *global estimate vector* and the *global statistics vector* are on the same side of the global threshold, the estimate vector can be used to answer the ratio threshold query. In [12], the *global statistics vector* is represented as the convex hull of the drift vectors. Further, the authors showed that the data region covering the convex hull of drift vectors can be represented as a subset of the union of regions covered by *balls* formed using the *global estimate vector* and the individual *drift vectors*. A *ball* formed using two vectors $\vec{u}$ and $\vec{v}$ has its center at $(\vec{u} + \vec{v})/2$ with radius $(\vec{u} - \vec{v})$. An individual data source need not send any notification to the aggregator as long as the ratio of numerator and denominator data items calculated at each point of the *ball* is below the global threshold. In our implementation we get equation of the *ball* and get the maximum ratio that any point in the *ball* may have. If this ratio is above the global threshold, notification is sent to the aggregator.

3.6 Summary

To summarize, we have developed a number of algorithms to assign source conditions at individual data sources:

1. Value algorithm (*Val*) is described in Section 3.1.
2. The ratio threshold algorithm, given in Section 3.2, is not expected to perform well in most of the cases; hence, instead, the modified ratio threshold algorithm (*Rat*) described in Section 3.3 is considered for performance evaluation.
3. *InterceptR* has thresholds depending on the *R*ate of change of the intercept, as given in Section 3.4.1.
4. *InterceptF* calculates thresholds that are dependent on *F*raction difference, as described in Section 3.4.2.
5. *Intercept* algorithm with *T*ime varying threshold *InterceptT*, is given in Section 3.4.3.
6. *Drift* algorithm is the solution proposed in [12] as explained in Section 3.5.

In the next section we compare the performance of these algorithms.

4. PERFORMANCE OF ALGORITHMS

For comparing the performance of various algorithms to assign source conditions at data sources, we use synthetic as well as real data.

Synthetic data: The synthetic data is generated to mimic the scenario described in *Example1*. Table 2 gives default values for data characteristics and RTQ parameters. By default, we assume that there are 10 sources whose data are being aggregated at an aggregator. For each source we consider 10000 update instances. For the i^{th} source ($1 \leq i \leq 10$) we simulated a total of i sales per update instance, out of which 1 sale is for electronics items. Dollar value for each individual sale is modeled using log-normal distribution [5] with a known mean (=1) and standard deviation ($0 \leq \sigma \leq 3$). Thus, mean local ratio at the i^{th} source is $1/i$. In these settings, default client threshold was kept at 0.25 (so there will not be any client threshold violation for $\sigma = 0$).

Table 2: Parameters for performance evaluation

Parameter Name	Nominal Value	Range
Number of data sources	10	10-50
μ (ratio threshold)	0.25	0.15-0.25
Number of sales per unit time	5	1-20
Electronic sales per unit time	1	1
Mean dollar value per sale	1	1
Std. dev. of dollar value/sale (σ)	1	0.1-3
Trace length	10000	10K-1M
θ (bound parameter)	0.5	0-1
α (adjustment factor)	4	1-10
λ (Section 3.1)	0.9	0.7-1
β (Section 3.4.1)	0.9	0.7-1
ϵ (Section 5)	0.05	0.03-0.1

Real-world backscatter data: We used CAIDA Backscatter 2004-2005 packet dataset [11] to simulate the scenario outlined in *Example2*. Such data is generated as side effects of spoofed denial of service (DOS) attacks. In this kind of attack the attacker forges the source address in IP packets sent to the host whose denial is attempted. The victim responds with normal packets. These response packets are known as backscatter. The backscatter data consists of traffic logs having time stamp, packet type, source IP address, destination IP address, and port numbers of the response packets.

We start with comparing performance of various algorithms, using synthetic data, for STA-UBT with varying data standard deviation. Real backscatter data is used to compare performance of WBA-UBT and corroborate the results obtained using synthetic data in Section 4.2. In both of these experiments *InterceptF* is proven to outperform all other algorithms. Effect of time varying threshold is presented in Section 4.3. In this experiment fixed part of the threshold is obtained using *InterceptF* whereas first order polynomial is used for the time-varying part as explained in Section 3.4.3. Next we measure performance of *InterceptF* for various categories of RTQs while varying value of the *bound parameter* θ to show that different values of θ are required in different categories for optimal performance.

We perform each of the experiments 10 times and report the average results.

4.1 Performance of STA-UBT

In the first set of experiments, we consider STA-UBT queries and compare the performance of *Val*, *Rat*, *InterceptR*, *InterceptF*, and *Drift*. Figure 4 shows that the number of push messages increases with increasing σ for all the algorithms. It should be noted that out of 10 sources 3 sources ($i = 1, 2, 3$) have their mean ratios (=1/1, 1/2, 1/3) above the ratio threshold (= 0.25). Thus if all the sources are assigned a local ratio threshold same as the client's threshold (Section 3.2) that is likely to result in more than 30000

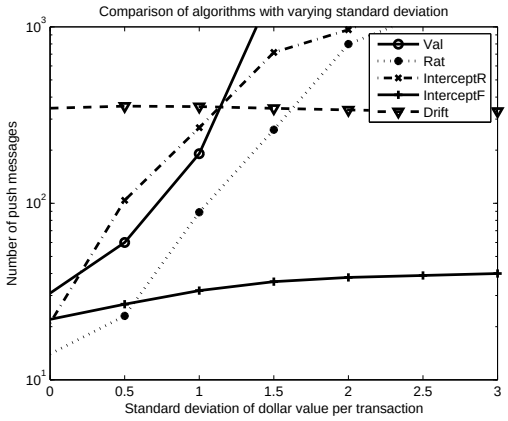

Figure 4: Performance comparison for STA-UBT

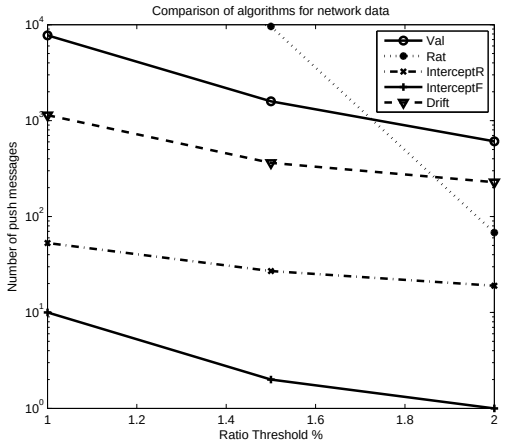

Figure 5: Performance of WBA-UBT with real traces

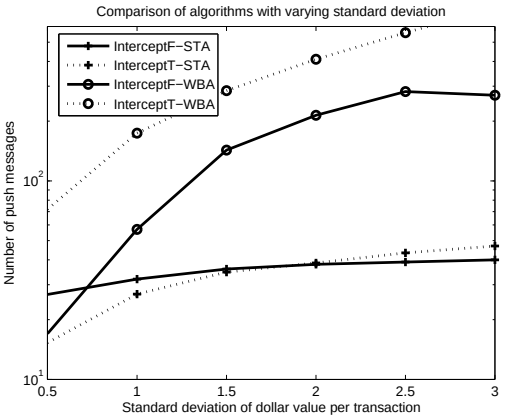

Figure 6: Effects of time varying threshold

(=3*10000) pushes. In comparison, the corresponding numbers for *Rat* and *InterceptF* are 89 and 32 respectively for $\sigma = 1$. The ratio threshold algorithm (*Rat*) performs well for very low values of standard deviation (with α between 4 and 6) whereas *InterceptF* gives good performance for all the values of σ. Surprisingly *InterceptR* performs very badly for larger values of σ. The poor performance is because of dependence between values of β and k_r. If β is higher (say, 0.9), sources whose intercept value is reducing (i.e., $\gamma_i \leq 0$) have threshold values close to the current value of the *intercept* and, with the higher standard deviation, these threshold values are violated very frequently. For lower values of β (say, 0.7), sources with positive values of γ_i cause a large number of pushes. The poor performance of *Drift* [12] can be explained by the fact that in *Drift* a source pushes a message if the absolute difference between expected data values and local data values crosses a threshold even if its local values are causing it to move away from the client threshold.

4.2 Performance of WBA-UBT

We use the backscatter data to measure the performance of our algorithms for WBA-UBT. To counter a denial of service attack, in the congestion based method [6], a specific destination P is "marked" if the fraction of packets coming to that address is above a specified threshold. We divided the traces collected from [11] to simulate packets incident on different routers. Each router maintains counts of total number of packets incident on it as well as the number of packets with destination address P. We assume that different routers are getting packets at different speeds. Specifically, we assumed that i^{th} router is receiving $50 * i$ packets per second for $1 \leq i \leq 10$. The network trace had more than 16 million packets enabling us to simulate 100 minutes. Client notification is generated if the fraction of packets, in a window of 1 minute, with IP addresses 61.0.0.0 (with subnet mask 255.0.0.0) crosses a specified threshold. Figure 5 shows the number of push messages for various algorithms. It can be seen that *InterceptF* performs best for WBA-UBT corroborating results obtained using synthetic traces.

4.3 Effect of Time Varying Thresholds

In section 3.4.3 we explained that the number of push messages can, probably, be reduced by having time varying thresholds. In this sub-section we use the synthetic data to investigate the effect of time varying thresholds by comparing *InterceptF* with *InterceptT*. For these experiments threshold at i^{th} source, as given by Equation 13, has a fixed part and a time varying part $C_i\tau$ where τ designates time elapsed since the latest threshold assignment to the source by the aggregator (we are considering 1^{st} order polynomial). Values of C_is are proportional to the change rates of the *numerator intercept*. Figure 6 shows that for STA *InterceptT* performs better than *InterceptF* for lower values of σ but worse as value of σ is increased. This happens because with increasing value of σ it becomes difficult to capture the trend in the values of the numerator intercept with time. Similarly, as there is no trend in the data values for WBA (window size=100), *InterceptT* performs poorly. Thus, if it is possible to get trends for data item values, using *InterceptT* is beneficial.

4.4 Performance of *InterceptF* with Different Parameter Settings

Now we present the performance of the *InterceptF* algorithm for various RTQ categories (Section 1.3) which will illustrate the need for different parameter values for the intercept algorithms in different scenarios. Figure 7 shows the performance of different categories of RTQs for various values of the *bound parameter* θ while keeping $\sigma = 1$: (a) STA-UBT with μ=0.25; (b) WBA-UBT with μ=0.25 and

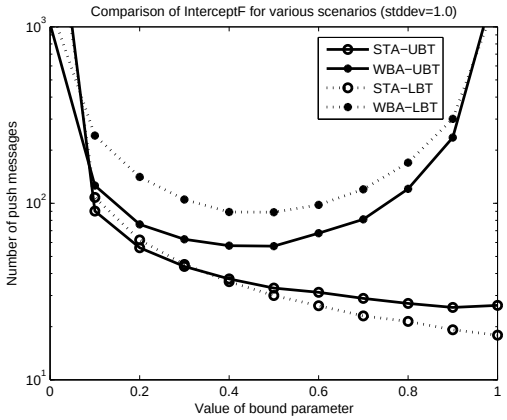

Figure 7: Performance of *InterceptF* with varying θ

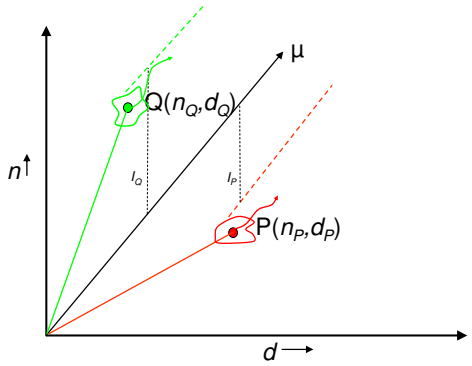

Figure 8: Explaining θ values for different RTQs

aggregation window size=100; (c) STA-LBT with μ=0.15; and (d) WBA-LBT with μ=0.15 and aggregation window size=100. It can be seen that for WBA algorithms, as we increase the value of θ, the number of push messages first decreases and then increases. Thus for WBA, we should keep θ in the range of 0.4-0.5 for UBT as well as LBT. In the case of STA, in general, the number of push messages reduces with increasing values of θ. For STA-UBT, minimum number of push messages were obtained at θ =0.9 whereas for STA-LBT the corresponding value was 1.0.

Reasons for the need for different values of θ for different scenarios can be explained as follows: Figure 8 shows typical numerators and denominator values at two sources P and Q. As long as the values of (n_i, d_i) at these sources are below the dashed red and green lines, respectively, there will not be any push message from these sources. As explained in Section 3.4.2 for UBT, lower bound for the proportionality constant k_f, i.e., k_{flower}, is calculated using data values from sources having positive values of fraction difference f_{i0} (like Q) and upper bound k_{fupper} from sources having negative value of f_{i0} (like P). In case data sources continue to behave as they have been behaving in the recent past (i.e., maintaining their local ratios approximately equals to the current ratios), then:

- In the case of WBA, depending on the value of standard deviation, numerator and denominator values at P and Q are likely to remain around their current values, thus possibly, within their corresponding red and green contours. In this case, both P and Q are equally likely to cross red and green dashed lines respectively. Thus we should keep the value of k_f equi-distant from k_{flower} and k_{fupper} which is achieved by keeping θ closer to 0.5.

- In the case of STA, data values at P and Q are likely to follow the corresponding (red and green, respectively) curved arrows. Thus, it can be seen that intercept at Q is more likely to cross its threshold than that at P. To keep the chances of intercept crossing similar for both the sources, we should have k_f closer to k_{fupper} which is achieved by having θ closer to 1.

- Different values of θ should be used for STA-UBT and STA-LBT as: In case of UBT (LBT), k_{flower} is obtained from sources having positive (negative) values of f_{i0} and k_{fupper} from sources having negative (positive) values of f_{i0}. We simulated the data at i^{th} source by aggregating dollar values of i sales for denominator and 1 sale for numerator. Thus, with a fixed value of mean and standard deviation of dollar value per transaction, denominator values of sources having positive f_i will have larger standard deviation compared to sources with negative f_i. Thus, for STA-UBT value of θ should be lesser (i.e., away from sources having more standard deviation) compared to that for STA-LBT. This explains why for STA-UBT we get best results for θ=0.9 whereas for STA-LBT we get best results for θ=1. Summarizing, in practice, θ=0.5 should be used for WBA whereas θ should be between 0.8-1 for STA depending on standard deviations of the dynamic data.

4.5 Summary of Performance Results

In this section we measured the number of push messages for various algorithms. We performed several other experiments details of which are not included for lack of space. We compared performance of various algorithms when standard deviation of the data changes with time. We found that *Drift* and *InterceptF* cope very well with changing data characteristics whereas *Val* and *Rat* have poor performances. Specifically, compared to the case of a constant standard deviation, in case of time varying standard deviation (we changed σ once at 5000^{th} update instance), performance of *InterceptF* worsens only by 10% compared to more than 50% for *Val*. To assess the scalability of our algorithms we measured the overhead at the aggregator for processing messages. Even for a scenario involving 10000 sources, 100000 updates per source, the total processing overhead at the aggregator was less than 1 millisecond. The total processing time at the aggregator included the time to read the data from all the sources, updating local conditions for them, and pushing the updated local conditions. This shows that our algorithms can work with a large number of data sources. To summarize, except for very low values of standard deviation, where *Rat* and *InterceptT* perform better, *InterceptF* performs best in almost all the cases.

5. SATISFYING CORRECTNESS WITHOUT PULLING FROM ALL SOURCES

In the previous section, we assumed that when the data aggregator receives a push message from any of the sources, it pulls data from all the sources to know whether the client needs to be notified and, if that is not the case, assign new source conditions to avoid unnecessary push messages. Can the data aggregator recognize that C is not violated without pulling data from all the sources? Can it assign new C_is without knowing data values at all the sources? In this section, we answer these questions with the goal of devising selective pulling techniques to reduce cost of handling each source condition violation which can significantly reduce the total number of messages, especially in cases when the number of data sources is large. We use *InterceptF* to illustrate our approach.

5.1 Adjusting Thresholds without Pulling from all the Sources

If any source S_1 sends a message to the aggregator, the data aggregator can know that it need not send notification to the client if the value of *numerator intercept* of S_1 can be brought within the threshold by assigning additional threshold to it. But, for getting the additional threshold, the aggregator needs to select one or more sources whose thresholds can be reduced without violating their own source conditions. It arranges sources in a priority order such that it is highly likely that the thresholds of higher priority sources can be reduced without violating their own source conditions. Various methods for assigning priorities are discussed in the next section. We employ the following scheme to assign modified thresholds to the data sources: For a notifying source S_1, we pull data from the source with highest priority (say, S_2) among the remaining sources. If the sum of the *numerator intercepts* of the notifying and the pulled sources is less than the sum of their thresholds, without pulling data from other sources, we know that the notification need not be sent to the client. Otherwise, we need to pull data from more sources. Specifically, we continue pulling data from highest priority sources among the remaining sources till $(I_{N1} + I_{N2})$ is less than $(1 - \epsilon)(T_{N1} + T_{N2})$. A small value of ϵ $(0 \leq \epsilon \leq 1)$ is used so that the values of individual thresholds are strictly more than the corresponding values of the intercepts. Using experiments we set $\epsilon = 0.05$. The data aggregator assigns modified thresholds to *numerator intercepts* of the notifying and the pulled sources using the method outlined in Section 3.4.2 (using f_{i0} values of the data sources involved).

If there are large number of data sources, pulling data from sources one-at-a-time may lead to high latency. Thus, while designing a selective pulling algorithm we may have to make a compromise between reducing the number of pull messages and reducing latency to determine whether a source notification should imply notification to the client. For example, we can selectively pull from a chosen subset of data sources rather than pulling from one source at a time. Algorithm described above can be easily extended to achieve this.

5.2 Deciding Pull Order

In this section we present two methods to assign priorities for *pulling*.

5.2.1 Random

In this scheme, the data aggregator randomly (by default, using uniform distribution) selects any of the sources, which have not notified their violation, for selective pulling. This scheme is based on the assumption that all the sources are equally likely to send push messages, hence, they are also equally likely to be useful for identifying whether the client condition is indeed violated.

5.2.2 Threshold based

If we assume that data sources (approximately) maintain their past ratios in the future, we can assign pulling priorities to sources in reverse order of threshold values, i.e., sources with lowest threshold values are pulled from first. However, this scheme suffers from a drawback: as data from a source (say S_2 of Section 5.1) is pulled, its threshold value is decreased (to increase threshold of S_1), thus, the next time a pull is required, S_2 is more likely to be pulled again. We need to correct this because when data from a source is pulled, its effectiveness to help in avoiding further pulls reduces. Once data is pulled from a source, to lower its priority of pulling, we propose a dynamic pull priority scheme. Specifically, after data is pulled from all the sources, we assign priorities to the sources as per their threshold values (lower the threshold higher the chances of pulling). We maintain a variable sf_i for each data source S_i as exponential average of the ratio of the number of data sources pushing at any particular instance divided by the number of sources (including S_i) whose data is pulled. A new value of sf_i is calculated whenever data is pulled from S_i. Then priorities are assigned in inverse order of *pfactor* defined as:

$$pfactor_i = \begin{cases} T_{Ni} \times sf_i & T_{Ni} \geq 0 \\ T_{Ni}/sf_i & \text{otherwise} \end{cases} \quad (14)$$

This ensures that the source with higher value sf_i is less likely to be selected for pulling.

Although we presented the *selective pulling* techniques for *InterceptF* algorithm similar techniques can be developed for other algorithms as well. For example, in the *Rat* algorithm value thresholds are assigned using Equation 8 and, pull priorities to the sources can be assigned based on their $(\mu - \mu_i)d_{i0}$ values.

5.3 Performance of Selective Pulling

Figure 9 shows the number of pulls for various values of σ with number of sources $|S| = 10$ & 20. Using selective pulling the number of pulls can be reduced by more than half. For $|S|$=20 and σ=2, selective pulling reduces the number of pulls from 600 to 276. Even random selection of data sources for selective pulling reduces the number of pulls by 40%. In comparison, selective pulling that uses the threshold based priority assignment requires lesser number of pulls as it selects a source more likely to reduce its threshold without incuring any of its own violations. It should be noted that we first found an efficient algorithm to minimize the number of push messages and then used selective pulling to minimize the number of pull messages. We believe that it is unlikely for an algorithm having higher number of push messages to incur lower pulling cost compared to *InterceptF*; as we saw in Section 4, *InterceptF* outperforms other algorithms comfortably. We verified these observations by designing selective pulling algorithm for the *Rat* algorithm and found its number of pulls (with selective pulling) much higher compared to *InterceptF*.

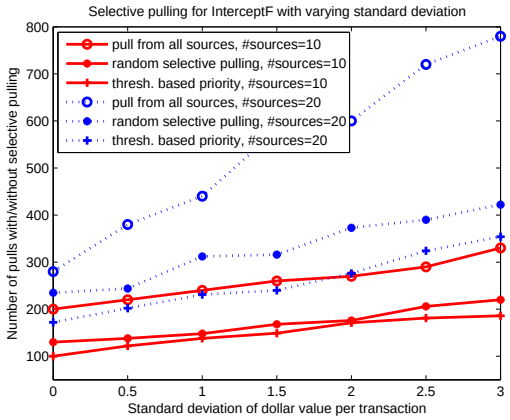

Figure 9: Comparison of *InterceptF* with and without selective pulling

6. MINIMIZING OTHER COST MEASURES

There are scenarios where different sources have different costs associated with pushing and pulling the data. The different costs may be due to different delays in getting the messages, load on the sources, etc. In such cases, instead of minimizing the number of pushes and pulls, our goal should be to minimize total cost of refreshing the data in a certain time interval. We assume that cost of each push and pull from data sources are specified in terms of scalar numbers, $pushcost_i$ and $pullcost_i$, respectively, i.e., the aggregator knows the cost metric corresponding to each data source.

6.1 Reducing Push and Pull Costs

To reduce the communication cost, we use the following methods:

1. **Reducing push cost**: In various algorithms presented in Section 3, we can assign tighter thresholds to the sources having lower cost while giving the extra *slack* to the sources having higher cost. This ensures that higher cost sources push fewer messages. Specifically we use the following algorithm to assign the local thresholds for *InterceptF*: first we assign the threshold as per the method outlined in Section 3.4.2. For each data source calculate *slack* l_i defined as the difference between its threshold and its current value of *numerator intercept*, i.e., $l_i = T_{Ni} - I_{Ni0}$. For sources having less than average *pushcost*s (denoted by *pushavg*), we reduce its *slack* in proportion to $pushcost_i/pushavg$. The additional *slack* is assigned to other sources in proportion to their $pushcost_i$ values.

2. **Reducing pulling cost**: While deciding on selective pulls, the aggregator gives higher priority to the sources having lower $pullcost_i$. We implement this by dividing the priority obtained by the method outlined in the previous section by $pullcost_i$

6.2 Performance of Cost Reduction Techniques

We calculate the total cost for five different optimization scenarios of the *InterceptF* algorithm:

1. Minimizing number of pushes using *InterceptF* without selective pulling (*minPushNum*).

Table 3: Total push and pull costs

Algo \ cost	Push Cost	Pull Cost
minPushNum	35.2	190.4
minPullNum	45.4	125.7
minPullCost	47.4	108.4
minPushCost	32.0	206.0
minPushPullCost	44.1	107.4

2. Minimizing number of pulls using selective pulling (*minPullNum*).

3. *Minimizing pull cost* using $pullcost_i$ values without any push cost reduction (*minPullCost*).

4. *Minimizing push cost* using $pushcost_i$ values without any pull cost reduction (*minPushCost*).

5. *Minimizing pull cost* and *reducing push cost* using $pullcost_i$ and $pushcost_i$ values, respectively (*minPushPullCost*).

In this experiment push and pull costs for different sources are: *pushcost*= {1.25, 1.25, 1.25, 1.25, 1.25, 0.75, 0.75, 0.75, 0.75, 0.75} and *pullcost*= {1.25, 1.25, 1.25, 1.25, 1.25, 0.75, 0.75, 0.75, 0.75, 0.75}. These experiments are done with 10 data sources and $\sigma = 2$ with 10 different random data generation keys. The *minPushNum* and *minPullNum* scenarios are implemented by assuming equal costs of messages for all sources. Selective pulling is used in *minPullNum*, *minPullCost*, and *minPushPullCost*. Table 3 shows that we can reduce the cost of pushing and/or pulling using methods given in Section 6.1. Using *minPushCost* we reduce the push cost from 35.2 to 32 compared to *minPushNum*. Similarly, compared to selective pulling (*minPullNum*), cost based pulling reduces cost by more than 10%. But if we try to reduce the pull cost then the push cost is more than that in the *minPushCost* case. Total cost of communication can be calculated by adding push cost and pull cost.

7. HANDLING MULTIPLE RTQs

Till now we have implicitly assumed that the execution of an RTQ does not have an impact on other queries. But if there are multiple RTQs, with overlapping data items, executing at an aggregator then the aggregator can use the data values pulled for one query to benefit the other queries. We present two schemes for this type of sharing.

Sharing messages: In this scheme, push and pull messages are shared across queries. First, we independently get local conditions for each RTQ. If any local condition is violated, the corresponding data source sends values of the data items along with the notification. The notification causes the aggregator to pull from other sources which contribute data items to that particular query. These data values are used by the data aggregator to reassign local conditions not only for that particular query but also for the other queries that use data from the same data sources. We use the algorithm described in Section 5.1 to adjust local conditions for these other queries. It should be noted that we cannot apply that algorithm unless we get data values from at least two data sources for a query, hence, this *sharing message* technique can be used only when queries have two or more data sources in common.

We conducted experiment with two RTQs having 10 data sources each, 5 of which are overlapping. Data traces are

generated using the technique described in Section 4. Global thresholds for the two queries are 0.25 and 0.28, respectively. If these queries are executed independently they require, on average, 18.9 and 13.4 notifications (push messages), respectively. But if we use the *sharing messages* technique, the total number of notifications is 25.1. Clearly, sharing helps in reducing the number of messages. There is an alternative technique involving sharing of local condition across RTQs. Due to space constraints we only provide an overview of this technique below.

Sharing local conditions: In this scheme, we not only share the messages to assign local conditions, we also assign a single local condition considering a number of queries.

- First, we get local conditions independently for each of the queries.
- Next we "merge" the local conditions involving the same set of numerator and denominator data items. For each local condition, we find the difference between the intercept threshold and the current value of the intercept. The local condition corresponding to the minimum of these differences is chosen as the local condition for that source. This tightest local condition is used for all the queries involving the same set of data items at that source.
- We again assign local conditions for all the queries, for the remaining sources, knowing the local conditions for the sources for which we have already assigned conditions.
- Whenever a particular local condition is violated we need to consider all the queries which may be affected by that violation. To know whether global condition of any of these queries is indeed violated, we need to pull at least one more data item for each of these queries. We can either pull all the common data items or selectively pull data items to know whether the global threshold is indeed violated.

There are various issues while designing this technique such as the order of considering data sources for local condition assignment, the order in which queries should be selected for data pulls, etc. We plan to investigate these issues as part of our future work.

8. CONCLUSION AND FUTURE WORK

In this paper we introduced ratio threshold queries in a distributed data sources setting. We motivated the problem using a number of practical examples. These ratio threshold queries are answered by a data aggregator which divides the client's threshold condition into conditions for individual data sources such that there are no false negatives and the number of message exchanges between data sources and the aggregator is minimized. We developed efficient algorithms to get the conditions for individual data sources. In general, we found that *InterceptF* algorithm gives better performance compared to other algorithms. We also presented various schemes for selectively pulling data from sources to efficiently identify whether notifying the client is warranted. Such schemes reduce the number of pulls by more than half. We have also done preliminary studies on non-linear RTQs involving non-linear aggregations and/or threshold being a function of dynamic data items [10]. Identifying compelling use cases for such non-linear aggregations and developing scenario specific efficient algorithms is our future research direction.

9. REFERENCES

[1] B. Babcock and C. Olston. Distributed top-k monitoring. In *SIGMOD '03: Proceedings of international conference on Management of data*, pages 28–39, 2003.

[2] R. Cheng, B. Kao, S. Prabhakar, A. Kwan, and Y. Tu. Filtering data streams for entity-based continuous queries. In *IEEE Transactions on Knowledge and Data Engineering*, pages 234–248, 2010.

[3] R. Gupta, A. Puri, and K. Ramamritham. Executing incoherency bounded continuous queries at web data aggregators. In *WWW '05: Proceedings of the 14th international conference on World Wide Web*, Chiba, Japan, 2005.

[4] R. Gupta and K. Ramamritham. Optimized query planning of continuous aggregation queries in dynamic data dissemination networks. In *WWW '07: Proceedings of the 16th international conference on World Wide Web*, pages 321–330, Banff, Alberta, Canada, 2007.

[5] Investopedia. `http://www.investopedia.com/terms/l/log-normal-distribution.asp`.

[6] J. Ioannidis and S. M. Bellovin. Implementing pushback: Router based defense against DDOS attacks. *Network and Distributed System Security Symposium*, 2002.

[7] A. Jain, J. M. Hellerstein, S. Ratnasamy, and D. Wetherall. A wakeup call for internet monitoring systems: The case for distributed triggers. In *SIGCOMM workshop on Hot Topics in Networks (HotNets III)*, 2004.

[8] R. Keralapura, G. Cormode, and J. Ramamritham. Communication efficient distributed monitoring of threshold counts. In *SIGMOD '06: Proceedings of international conference on Management of data*, 2006.

[9] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD '03: Proceedings of international conference on Management of data*, San Diego, 2003.

[10] Technical report: Executing ratio threshold queries over distributed data sources. `http://www.cse.iitb.ac.in/~krithi/papers/rtq.pdf`.

[11] C. Shannon, D. Moore, E. Aben, and kc claffy. The caida backscatter-2004-2005 dataset - may 2004 - november 2005. `http://www.caida.org/data/passive/backscatter_2004_2005_dataset.x`.

[12] I. Sharfman, A. Schuster, and D. Keren. A geometric approach to monitoring threshold functions over distributed data streams. *ACM Transactions of Database Systems*, 32(4):23, 2007.

[13] S. Zhu and C. V. Ravishankar. Stochastic consistency, and scalable pull-based caching for erratic data stream sources. In *VLDB '04: Proceedings of the Thirtieth international conference on Very large data bases*, 2004.