Efficient Execution of Continuous Aggregate Queries over Multi-Source Streaming Data

Hrishikesh Gupta*, Manish Bhide†, Krithi Ramamritham‡, Prashant Shenoy§
*Dept. Of Computer Science And Engg.
University of California, San Diego, CA 92093-0114
Email: hgupta@cs.ucsd.edu
†IBM India Research Lab,
Block-1 IIT Delhi, New Delhi India 110016 Email: abmanish@in.ibm.com
‡Computer Sc. and Engg Dept, IIT Bombay India 400076
Email: krithi@cse.iitb.ac.in
§Computer Science Dept, University of Massachusetts, Amherst, MA 01003-4610
Email: shenoy@cs.umass.edu

Abstract— **On-line decision making often involves query processing over time-varying data which arrives in the form of data streams from distributed locations. Examples of time-varying data include financial information such as stock prices and currency exchange rates, real-time traffic, weather information and data from process control applications. In such environments, typically a decision is made whenever some function of the current value of a set of data items satisfies a threshold criterion. For example, when the traffic entering a highway exceeds a prespecified limit, some flow control measure is initiated; when the value of a stock portfolio goes below a comfort level, an investor might decide to rethink his portfolio management strategy. In this paper we present data dissemination and query processing techniques where such queries access data from multiple sources. Key challenges in supporting such *Continuous Aggregate Queries with Thresholds* lie in minimizing network and source overheads, without the loss of *fidelity* in the responses provided to users. Using real world data we demonstrate the superior performance of our techniques when compared to alternatives based on periodic independent polling of the sources.**

I. Introduction

A. Motivation

An increasing fraction of data on the web is time-varying (i.e., varies continuously with time) and is presented in the form of data streams. Examples of time-varying data include financial information such as stock prices and currency exchange rates, real-time traffic, weather information and data from process control applications. They are frequently used for online decision making, and in many scenarios, such decision making involves *multiple* time-varying data items, from multiple independent sources. Examples include a user tracking a portfolio of stocks and making decisions based on the net value of the portfolio. Observe that computing the overall value of the portfolio at any instant requires up-to-date values of each stock in the portfolio. In some scenarios, users might be holding these stocks in different (brokerage) accounts and might be using a third-party data aggregator, such as yodlee.com [26], to provide them a unified view of all their investments. In general, third-party aggregators assemble a unified view of the information of interest to the user by periodically gathering and refreshing information from multiple independent sources. If the user is interested in certain events (such as a notification every time the value of the portfolio crosses $10000), then the aggregator needs to periodically and intelligently refresh the value of each stock price to determine when these user-specified thresholds are reached. More formally, the aggregator runs a *continuous query* over the aggregate of data items by intelligently refreshing the individual data items from their distributed sources in order to determine when user-specified thresholds are exceeded. We refer to such queries as *Continuous Aggregate Queries with Thresholds (CAQTs)* (pronounced as KAKTs).

Much of the prior work on continuous queries (see Section VI) has assumed that queries are handled directly by the server (data source). Such an approach has two important limitations. First, since continuous threshold queries are compute-intensive (due to the need to continuously evaluate the query condition), the source could become a bottleneck, and thereby, limit the *scalability* of the system. Second, a pure source-based approach makes CAQTs on data items from multiple independent sources impossible (for example, the source-based approach is unfeasible in the case where a user specifies a query on data items held in multiple independent accounts). Executing queries at intermediate data aggregators (e.g. proxies in case of the web) eliminates both restrictions—the approach is scalable, since computations are off loaded from the source to aggregators, and queries on data items from multiple sources become feasible. In addition, a proxy which acts as a data aggregator can improve user response times [6], [7] by being placed closer to the eventual clients. However, such an approach raises new research challenges. The key challenge is to ensure that the results of the query are no different from the case where the query is handled by an idealized aggregator which has the current version of the data available all the time –i.e., without delays– (or correctness of the results should be ensured). To do so, the aggregator should ensure that the values of time-varying data items are *temporally coherent* with the source.

In the rest of this section, we provide a precise definition

of CAQTs and then outline the research contributions of this paper.

B. Continuous Aggregate Queries with Thresholds (CAQTs): An Introduction

A CAQT $Q(\mathcal{D},\mathcal{N},\mathcal{R})$ operates on N data items $d_1, \ldots, d_N \in \mathcal{D}$ with $n_1, n_2, \ldots, n_N \in \mathcal{N}$ as the weights attached to these data items and informs the invoker whenever the threshold $\mathcal{R}$ is crossed. The set $D = \{S, P\}$ where $\mathcal{S}$ is the set of data values at the source and the $\mathcal{P}$ is the set of data values at the proxy. Let, $s_i(t)$ and $p_i(t)$ be the values of the data item $d_i(t)$ at time t at the source and proxy respectively. Then the CAQT can be formally defined as

$$If \quad f(\mathcal{S},\mathcal{N}) \geq \mathcal{R} \ then \ f(\mathcal{P},\mathcal{N}) \geq \mathcal{R} \tag{1}$$

and if

$$f(\mathcal{S},\mathcal{N}) \leq \mathcal{R} \ then \ f(\mathcal{P},\mathcal{N}) \leq \mathcal{R} \tag{2}$$

Here f can be any function such as sum, min, max, etc. In this paper, we focus on Sum Based Queries although our techniques can be generalized for other aggregations V as well. Sum based CAQTs are a type of CAQT in which the user should have the correct knowledge of the position of the portfolio of data items with respect to the specified threshold $\mathcal{R}$. This can be stated as

$$If \ \sum_{i=1}^{i=N} s_i(t) \cdot n_i \geq \mathcal{R} \ then \ \sum_{i=1}^{i=N} p_i(t) \cdot n_i \geq \mathcal{R} \tag{3}$$

and if

$$\sum_{i=1}^{i=N} s_i(t) \cdot n_i \leq \mathcal{R} \ then \ \sum_{i=1}^{i=N} p_i(t) \cdot n_i \leq \mathcal{R} \tag{4}$$

The aggregator needs to dynamically track changes in the stock price to successfully execute the CAQT and notify the portfolio value to the user when it crosses the specified threshold.

C. Metrics used to characterize the algorithms for executing CAQTs

1) Fidelity of CAQTs: We use *Fidelity* to measure the accuracy of the CAQT executed by a aggregator with respect to that of an idealized one, that is, a aggregator which has current data available all the time which in turn implies that the aggregator has the correct knowledge of the position of the CAQT with respect to the value of the CAQT at the source. To quantify these misperceptions, we define fidelity as the percentage of total time duration for which the aggregator knows the correct state of the CAQT.

$$fidelity = (1 - \frac{Total\ out\ of\ sync\ time}{Total\ trace\ duration}) \ \times \ 100$$

where the *Total out of sync time* is the time duration for which the aggregator was oblivious of the right state of the CAQT.

Notice that if Equation 3 and 4 are satisfied at a particular instant t, then the exact values of data items at the aggregator, $p_i(t)$, *need not correspond to* $s_i(t)$*, the values at the source.* It is sufficient for the aggregator to know that the threshold is not exceeded. In other words, the aggregator needs to be aware of only those updates to the data items that cause the CAQT value to cross the threshold. This key observation allows us to reduce the number of updates to data items that need to be propagated from the source to the aggregator.

2) Network Overhead: We have chosen the number of network messages as the parameter to measure network overhead. If we are able to minimize the number of messages itself, irrespective of the network delays, we will have better performance. Moreover network delays also vary over different networks. Therefore, we define network overhead as the number of network messages incurred for every 100 changes at the source.

In a push-based approach, there are push messages from the source involving update of the values at the aggregator and from the aggregator to the sources involving update of windows. However, a pull-based approach requires *two messages*—a request, followed by a response—per poll. Clearly, given a certain number of updates received by a aggregator, pull-based approaches incur larger message overheads.

3) Scalability: The data dissemination set up needs to be scalable to a large number of CAQTs or to CAQTs which involve a larger number of data items. Computational overhead as well as space overhead can be a bottleneck and hamper the scalability of the solution.

Computational overheads can occur at the aggregator, source, or both. Computational overheads for a pull-based approach result from the need to deal with individual pull requests at the aggregator. After getting a pull request from the aggregator, the source just needs to look up the latest data value and respond. On the other hand, when the source has to push changes in the data value to the aggregator, for each change that occurs, the source has to check if the change must be pushed. The latter might involve evaluating specific conditions.

A pull-based source is stateless and does not have any overhead at the source. In contrast, a push-based source must maintain the coherency requirement of a client for each data item, the latest pushed value, along with the state associated with an open connection. Since this state is maintained throughout the duration of connectivity, the number of clients the source can handle may be limited resulting in *scalability* problems. If there are N data items and c communication channels then the state space needed is:

$$N \times (B(d,w)) \ + c \times (B(cs)) \tag{5}$$

where d is the data item, w is the coherency requirement, $B(d,w)$ is the number of bytes needed for a (d,w) pair and B(cs) is the number of bytes needed for a connection state.

4) Resiliency: If a source fails and a aggregator expects the source to push changes of interest, a aggregator might incorrectly believe that it has temporally coherent data and possibly take some action based on this. On the other hand, if a aggregator explicitly pulls data from a source, it will soon enough, say at the end of a time out period, know about the failure of the source and take appropriate action. Once the

aggregator realizes that the source has crashed, it can either refrain from taking any decisions or connect to a different source if possible. Clearly, the latter could be done transparent to the actual user.

The above discussion clearly shows that pull is preferable when we consider the computational overheads, state space overheads and resiliency problems associated with push. On the other hand, fidelity and communication overheads of pull depend on the nature of the data updates and how frequently the aggregator refreshes the data. We make use of the complementary properties of the canonical pull and push based approaches to develop efficient techniques to deal with query processing over streams originating from multiple sources.

D. Contributions of this paper

In this paper, we present:

1) A suite of data dissemination algorithms to efficiently disseminate time-varying data items from sources to aggregators (observe that a naive approach that disseminates every update to a frequently changing data item can have a prohibitive overhead). To overcome this overhead, we face the following issues.
   a) Since every CAQT is executed over a group of data items, how should one derive the coherency requirement of each of the data items used by the CAQT?
      The coherency requirement associated with a data item depends on (a) the overall contribution of the data item to the CAQT's result and (b) the threshold value associated with the CAQT. In this paper we propose techniques to determine a data items' coherency requirements to ensure correct execution of the CAQTs.
   b) How should the (derived) coherency requirement associated with each item be ensured?
      To maintain coherency of the data, each data item must be periodically refreshed with the copy at the source. In this paper we demonstrate that, in the case of CAQTs, using standard techniques, such as pull and push, for individually retrieving each of the data items is not efficient - *additional mechanisms must be employed which exploit the fact that the data items that are needed to execute a CAQT form a semantic unit*.
2) How our model for data aggregation allows (a) query processing to be moved from sources to aggregators, (b) permits the presence of multiple aggregators and (c) judiciously combines push and pull based approaches to improve scalability and resiliency.

Note that a user is interested only in the fidelity of the CAQT and not the exact values of the d_i's. Hence as long as a aggregator is correctly informed about the state of a CAQT, even if an individual data item is changing very rapidly there is no need to follow these changes frequently. This is a key observation that is exploited by our novel techniques.

The *Pull-Based CAQTs Execution Approach ($PullCEA$)* outlined in Section II makes use of this fact to reduce the network overhead without loss of fidelity . Here, a aggregator tries to estimate the expected change in the value of a data item and based on the position of the CAQT relative to the threshold, it estimates the time to refresh for each data item so that the fidelity of the CAQT is maintained.

Given sources that are capable of pushing changes, an alternative is to use a *Push-Based CAQTs Execution Approach ($PushCEA$)*, outlined in Section III. Here a aggregator calculates the coherency Requirement of data items and communicates them to the data sources. Sources push only those changes needed to meet user coherency requirements. In contrast with pull-based techniques, this approach offers better fidelity for individual data items and hence for the CAQTs.

Unfortunately, a push based approach requires that a source should maintain the coherency requirements of the data items that it serves. As mentioned earlier, this consumes state space at the source and is also prone to *scalability* and resiliency problems. Consequently, it is imperative to consider an intelligent combination of $PullCEA$ and $PushCEA$ as is done in Section IV. We develop a high performance *Hybrid CAQTs Execution Approach ($HyCEA$)* that dynamically categorizes CAQT's data items into those needing pull vs. those needing push. These are aimed at reducing the state space requirements and resiliency and *scalability* problems of a push based approach while retaining its fidelity maintenance properties. We describe as well as experimentally evaluate the performance of the three approaches using CAQTs defined over real-world data streams of dynamically changing data (specifically, stock prices) in Sections II through IV. The performance of the algorithms was evaluated using real-world data streams. The presented results are based on stock price streams (i.e., history of data values) of around 150 companies (Table I lists a few of them). Streams were constructed through repeated polling of http://finance.yahoo.com and all the experiments were done on the local intranet. These experiments were done for CAQTs between 3 and 15 data items.

All algorithms were evaluated using a prototype server/proxy that employed trace replay. Our evaluations show that by dynamically tracking the changes to data item values and by judicious combination of push and pull approaches, one can achieve high fidelity and *scalability* while maintaining low overheads.

The algorithms proposed for dealing with CAQTs are compared to the related work in Section VI and a summary of this paper is presented in Section VII.

II. The Pull-Based CAQTs Execution Approach ($PullCEA$)

$PullCEA$ is entirely pull-based and does not need any special support at sources (and hence, is compatible with any HTTP server). This algorithm dynamically computes the coherency requirements of each data item based on its "contribution" to the overall value of the CAQT—data items that are likely to change the value of the CAQT by a larger

TABLE I

SOME OF THE TRACES USED FOR THE EXPERIMENT

Company	Date	Time	Maximum Value	Minimum Value	Average Value
ABC	Jun 2, 2000	22:14-01:42 IST	135.75	134.5	135.144
Dell	Jun 1, 2000	21:56-22:53 IST	43.75	42.875	43.434932
UTSI	Jun 1, 2000	22:41-23:15 IST	22.25	21.00	21.731243
CBUK	Jun 2, 2000	18:31-21:57 IST	8.625	8.25	8.505309
Intel	Jun 2, 2000	22:14-01:42 IST	134.500	132.500	133.462484
Cisco	Jun 6, 2000	18:48-22:20 IST	65.000	63.0625	63.971584
Oracle	Jun 7, 2000	00:01-01:59 IST	79.3750	76.6250	78.576186
Veritas	Jun 8, 2000	21:20-23:48 IST	137.000	133.500	134.859644
Microsoft	Jun 8, 2000	21:02-23:48 IST	69.6250	68.0781	69.046370

amount are given a tighter coherency requirement, which enables the aggregator to track those data item with greater accuracy. In addition, the coherency requirement depends on the change in the data value vis-a-vis the changes in the other data items constituting the CAQT. The algorithm uses coherency requirements to determine how frequently to refresh (pull) the new value from the source—data items with tighter (smaller) coherency requirements are refreshed more frequently. We now describe the various components of $PullCEA$ and thereafter evaluate its performance.

The aggregator makes use of two kinds of coherency requirements: one is the static coherency requirement (c) which is the the potential amount by which each data item needs to change in order to contribute to an overall change of $\mathcal{T}$ (relative threshold given by equation 7) in the value of the CAQT. The other type of coherency requirement is the dynamic coherency requirement (d) which is the estimated change in the value of the data item depending on the dynamics of the data item. Using these two values the aggregator estimates the TTR (Time to refresh of each data item) and polles the data items based on the TTR values.

The c represents the potential amount by which each data item should change so that overall the portfolio crosses the threshold. Thus, given the absolute threshold $\mathcal{R}$, the $PullCEA$ algorithm apportions a part, c_i, to each data item such that

$$c_1 \times n_1 + c_2 \times n_2 + \ldots + c_N \times n_N = \mathcal{T} \tag{6}$$

where

$$\mathcal{T} = |\mathcal{R} - initial\ CAQT\ value| \tag{7}$$

The parameter c_i is the c of the ith data item. Intuitively, if each data item changes by amount c_i, the value of the CAQT changes by $\mathcal{T}$. The challenge then is to determine an appropriate c_i for each data item such that Equation 6 is satisfied. We then determine the c_i based on the weight attached with the data items to the value of the CAQT. If the weight attached with the data item is more then even a small change in the data value will have a large effect on the value of the CAQT. Hence data items that have a larger weight are given a smaller coherency requirement, implying that the aggregator tracks those data items with greater accuracy. Assuming a weight n_i for data item i,

$$c_i = \frac{\sum_{j=1}^{n} n_j - n_i}{\sum_{j=1}^{n} n_j} \times \frac{\mathcal{T}}{n_i \times (N-1)} \tag{8}$$

The larger the contribution of data item i (i.e., n_i), the smaller the quantity $\frac{\sum_{j=1}^{n} n_j - n_i}{\sum_{j=1}^{n} n_j}$, and the smaller the resulting tolerance c_i. The second factor $\frac{\mathcal{T}}{n_i \times (N-1)}$ ensures that the computed c_is satisfy Equation 6.

Whenever the aggregator is informed of a change in the value of any data item constituting the CAQT, the aggregator finds the relative threshold (equation 7) using the data values cached at the aggregator. It calculates the c (using equation 8) and tries to estimate the expected changes in the value of all data items constituting the CAQT during the next interval. To do so, we have extended the core idea of Asset Pricing Model for Stocks [24]. Though this model was developed specifically for stocks and our experiments were also conducted on stock values (solely due to the easy availability of stock data), it is generic enough to represent the behavior of any data item whose values exhibit brownian motion. It tries to model the behavior of a dynamically changing data item by decomposing the changes into drift components which are the expected changes in its value (based on the history of its behavior) and a diffusion component (changes caused by the external processes outside the system). The diffusion component accounts for the brownian behavior in the value of the data item.

In order to find the expected change, the aggregator starts building up the history of the data items constituting the CAQT. With the available history, we then use the following equation (derived from [24]) based on the drift and diffusion components of changes to estimate the future changes in the prices of data items:

$$d = \mu dt + \sigma dX \tag{9}$$

Here,

- d is the estimated change in the value of the data item during dt
- μ is the expected change in the price of the stock
- dt is the time interval for which we are calculating the estimated change
- σ is the volatility in the value of the data item
- dX is the measure of external factors

In this equation, the first term μdt helps us estimate the drift and the second term σdX helps us estimate the diffusion. We fix dt as 1 so as to be able to calculate the estimated change in data value over the next interval (further also referred to as S). To calculate μ we use the following equation:

$$\mu_i \ = \frac{(\mu_{i-1} \times S_{i-1}) + change_i}{S_i} \tag{10}$$

Here, μ_{i-1} is the previous μ, S_i is the count of intervals elapsed since the CAQT was registered with the aggregator, S_{i-1} is the count when the value of this data item was last updated at the aggregator and $change_i$ is the change in the data value since S_{i-1}. Thus, we are incrementally calculating the value of μ by storing minimal information at the aggregator (the values of S_{i-1} and μ). σ is the volatility in the value of the data item and it is calculated using the standard formula for volatility.

The value of dX is calculated as:

$$dX_i \ = \frac{L \times change_{i-1}}{(S_i - S_{i-1}) \times d_{i-1}} + (1 - L) \times dX_{i-1} \tag{11}$$

In this equation, $\frac{change_{i-1}}{(S_i - \ S_{i-1}) \times d_{i-1}}$ gives us an estimate of the unexpected change over the last update as $\frac{change_{i-1}}{S_i - S_{i-1}}$ is the change in the data value per interval over the last update and when divided by d_{i-1} (which was the estimated change per interval during the last update) we get an estimate of the new external forces that have been active in determining its value over the period of last update. We take an exponentially smoothed value of dX so that the entire randomness is not just attributed to the last change. The value of L ($0 \leq L \leq 1$), which is the smoothing constant, is kept greater than 0.5 to account for the recency effect (which states that changes in the recent past are reflective of the changes in the near future). This helps in capturing small and sudden trends in the values of dynamically changing data items.

Using this value of d, we calculate the polling interval (TTR) as follows. First we address two special cases:

- Case 1: If the c is zero, it implies that the CAQT value is equal to that of the threshold. In such a situation the aggregator should track the data item very accurately and hence its Time to refresh (TTR) is set to 1 time units.
- Case 2: If the dynamic cr of the data item is zero, it implies that the data item is changing very slowly or the expected change in the data value is zero. Hence the aggregator can afford to relax the polling frequency of the data item. The aggregator therefore increases the TTR of the data item and sets it to TTR$_{i-1}\times$ GROWTH FACTOR LIMIT.

In the unlikely event of both the cases being true, priority is given to the first case. If the above two cases do not apply to a data value (i.e., the CAQT value is not equal to the threshold and the data item is changing with non-zero rate) then there can be the following two cases:

- Case 3: If the data item moved away from the threshold during the last two pollings, then the possibility of the portfolio exceeding the threshold due to the stock is less. Hence the aggregator increases the TTR of the data item and sets it to TTR$_{i-1}\times$ GROWTH FACTOR LIMIT
- Case 4: If case 3 does not apply, then it implies that the data value is changing and it is taking the portfolio towards the threshold. The c is an estimate of total change required in the data value, so that the threshold is reached. Further, the d denotes the expected change in the data value in the next time interval. Hence the $PullCEA$ sets the TTR to $\frac{c}{d}$ which is nothing but the time required for the data item to change by c.

Once the new value of TTR is calculated, it is ensured that the value is within static bounds of (TTR$_{min}$,TTR$_{max}$), so that the TTR is not set to very high or very low values.

The TTR calculated by the above method assigns a portion of the threshold to each of the stocks and calculates the TTR required by each stock to achieve its apportioned threshold. However the rate of change of the data items in the portfolio can be very diverse, i.e., some of the data items might be changing very rapidly and at the same time some of the data items might be idle. In such a case, the $PullCEA$ will keep on polling the data items which are changing very rapidly with small TTR values and the TTR of the slow data items will gradually approach the TTR$_{max}$ value. However, under these circumstances, it was observed that the portfolio, does not cross the threshold, and the algorithm results in unnecessary pollings for the data items which are changing rapidly. In order to avoid these extra messages, we do an optimization in the $PullCEA$ algorithm and we distribute the static cr of those stocks that are changing slowly to the rest of the stocks. With this optimization, the TTR of the fast moving stocks increases, thereby reducing their polling frequency. This will ideally lead to a reduction in the network overhead without significantly affecting the fidelity of the CAQT.

The $PullCEA$ algorithm makes use of the direction of the data value change while calculating the cr, and the TTR. If one of the data items is taking the CAQT away from the threshold and at the same time if there is another data item that is changing very rapidly in a direction towards the threshold, then the $PullCEA$ will poll this data value very frequently. However, due to the first data item, the CAQT might not cross the threshold and such a case leads to unnecessary pollings. In order to avoid these extra pollings, we can distribute the static cr of the data items that are moving away from the threshold to the rest of the data items. Thus the polling frequency of the rest of the data items can be reduced without a significant loss in fidelity. This redistribution of the static cr is done only for those data item who has shown a history (during the last 2 polls) of moving away from the threshold.

A. Performance of PullCEA

In this section we show the experimental results for the $PullCEA$ algorithm. We simulated a CAQT of four data items each of which can potentially be on an independent server. The $PullCEA$ algorithm was run at the proxy that tracked the CAQT value by polling the data sources based on the

$PullCEA$ algorithm. We found out the range in which the CAQT value varied and then evaluated the algorithm by varying the threshold for this entire range of values. In practice there is no way to find out the fidelity of the CAQT at the proxy as the proxy is not always aware of the exact value of the CAQT at the data source. In our simulations we modified the data sources so that they kept track of the CAQT value at the proxy and kept track of the fidelity offered to the proxy. The constants that we used in our experiments are given in table II.

The results were also compared with an approach in which the coherency requirements were calculated only initially and were kept unchanged irrespective of the relative changes in the values of the data items. Once the (static) coherency requirement was calculated the, Adaptive TTL algorithm [19] was used to compute the TTR of the data items constituting the CAQT. The adaptive TTL algorithm computes the TTR based on the cr allocated to the data item. Thus this is the normal approach which does not take into consideration the relative performance of the data item in a CAQT.

Figure 1 shows the variation of the fidelity and the network overhead with varying values of average distance of the CAQT value from threshold. The graphs show that as the average distance from the threshold decreases, the network overhead increases and the fidelity decreases. As the average distance of the CAQT value from the threshold decreases, the number of times that the CAQT crosses the threshold increases. Hence there is an increase in the network overhead and a drop in fidelity. The important point over here is that the method in which the crs were statically computed performs very badly as compared to the $PullCEA$ algorithm. The Adaptive TTL algorithm does not take cognizance of the distance of the CAQT from the threshold. In this algorithm the cr value is calculated initially and it is kept fixed. As a result this method results in a larger network overhead without gaining in fidelity. The graph shows that the $PullCEA$ algorithm outperforms the static cr algorithm both in terms of network overhead as well as the fidelity.

Figure 2 shows the performance of the $PullCEA$ algorithm with the optimization when the static cr of slow moving data item is distributed amongst the rest of the stocks. It is evident from the figure that the network overhead of the optimized algorithm is less than that without the optimization: there is a saving of nearly 15% in the network overhead. An interesting point to note over here is that this reduction in the network overhead does not result in a significant decrease in the fidelity. In the optimization, if a data item is moving slowly then its static cr is distributed amongst the rest of the data items. This increases the TTR of the rest of the fast moving data items. This results in reduced polling frequency which might lead to a violation if (1) there is a sudden change in a slow moving data item or (2) there is a sudden large change in the fast moving data item which is missed due to the increased TTR. However the figure clearly shows that there is a very small decrease in the fidelity which validates the efficiency of our optimization.

We also found out the performance of the algorithm by varying the value of L. We have not included the figures due to space constraints but our results showed that there was a drop in the fidelity for values of L close to 1 and to 0. For values of L close to 0, the dX is based only on the past history and it ignores the immediately preceding change. The latest change is often reflective of the changes to the data value in the near future, and hence a value of 0 might lead to an incorrect value of dX resulting in reduced fidelity. With an L value of 1, the value of dX is only dependent on the last change in the data value. This also did not lead to good results, which signifies that the changes in the data value are dependent on the last few changes in the data value.

In the next section we describe how a push based approach can be used to execute a CAQT.

III. PUSH-BASED CAQTS EXECUTION APPROACH ($PushCEA$)

In *PushCEA* coherency requirements are represented in terms of coherency windows. A coherency window (w_i) denotes the upper and lower limit within which the value of data item may vary at the source without being updated to the aggregator. In $PushCEA$ the aggregator computes the coherency windows of each of the data items in the CAQT and informs them to the corresponding data sources. The data source stores the coherency window of each data item that it servers and informs the aggregator whenever the value of the data item goes outside the coherency window. The coherency window of the ith data item (w_i) can be formally stated as

$$w_i = < l_i, u_i >$$

where l_i is the lower part of w_i and u_i is the upper part of w_i with the w_i centered around the value of the data item at the aggregator.

The main issue in $PushCEA$ is to find the right window size for each of the data items in the CAQT such that the value of the CAQT should cross the threshold, if and only if atleast one of the data item value moves outside its window. Ideally, the window size should be dependent on the following issues: (1) The contribution of the data item to the CAQT with respect to the rest of the data items and (2) the rate of change of the data item as compared to the rest of the data items. In order to accomplish this goal, the d is calculated as given in equation 9. This d is the expected change in the data value in the next time unit. This expected change in the data value is then biased based on the contribution of the data item to the CAQT. This biased expected change $c_i^{'}$ is calculated as follows:

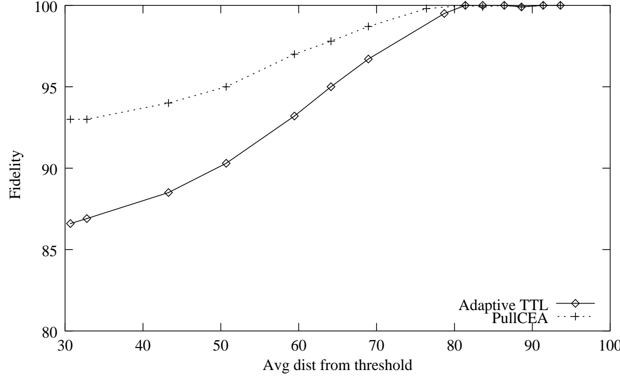
$$c_i^{'} = \frac{d_i \times ((\sum_{j=1}^{N} n_j) - n_i)}{\sum_{j=1}^{N} n_j \times (N-1)} \qquad (12)$$

If the weight of a data item is large, then even a small change in the data value will have a large effect on the overall value of the CAQT. Hence such a data item should be tracked much more accurately and should be monitored closely. This is captured by the above equation and expected change is

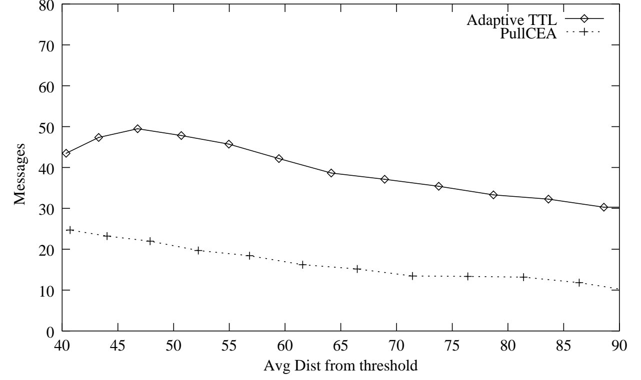
Symbol	Meaning	Value
GROWTH FACTOR LIMIT	The maximum factor by which the TTL can increase	2
TTR_{min}	The minimum value of TTR	1 sec
TTR_{max}	The maximum value of TTR	60 secs

TABLE II

VALUE OF PARAMETERS USED IN THE EXPERIMENTS

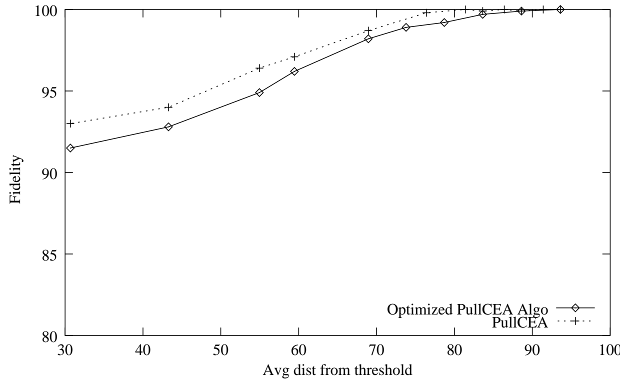


(a) Variation of Fidelity with Threshold

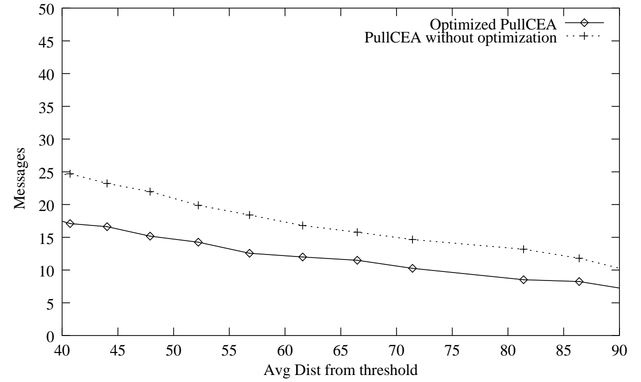


(b) Variation of Network Overhead with Threshold

Fig. 1. Comparison of static cr and $PullCEA$



(a) Variation of fi delity with Threshold



(b) Variation of network overhead with threshold

Fig. 2. Performance of $PullCEA$ with optimization

modified based on relative weight of the data item with respect to the weights of the rest of the data items in the CAQT. Now these biased cr ($c_i^{'}$) is independent of the distance of the CAQT from the threshold i.e., the relative threshold. We are interested in window sizes such that the CAQT value should cross the threshold if and only if atleast one of the data item value moves outside its window. In order to ensure this, the weighted cr ($wc^{'}$) is calculated which apportions the relative threshold value amongst the N data items.

$$wc_i^{'} = \frac{{c_i}' \times T'}{\sum_{j=1}^{N} c_j^{'} \times n_j} \quad (13)$$

where T' is the relative threshold (calculated as per equation 7.

The $PushCEA$ makes use of the weighted cr value to find the windows for each data item. For a coherency window, w_i, its l_i is initialized to $p_i - wc_i^{'}$ and u_i is initialized to $p_i + wc_i^{'}$ where p_i is the value of the data item cached at the proxy. Note that the windows are centered around the data value cached at the aggregator. Such a window size ensures that if the CAQT crosses the threshold, then atleast one of the data values will move outside its window. This can be proved as follows: The maximum difference between the value of the ith data item at the proxy and at the source is $wc_i^{'}$. Hence the maximum difference between the value of the CAQT at the source and that at the proxy is given by: $\sum_{i=1}^{N}(wc_i^{'} \times n_i)$ which is equal to the relative threshold.

These coherency windows are then communicated to the corresponding data sources. Whenever the value of a data item moves outside its coherency window at the source, the source pushes the new value to the aggregator. When a new data value is pushed by a source, the aggregator recalculates the $wc^{'}$ of all the data item based on the cached values of the data items at the aggregator. However if all these windows are informed to the data sources then this will lead to a lot of extra messages. In order to avoid this the aggregator does the following:

1) It informs the coherency window for the data item that caused this recalculation of windows i.e., the data item that was pushed, to the data source.
2) Out of the rest of the data items, the aggregator finds that data item that has not been pushed for the longest amount of time in the history. Such a data item is the most passive data item in the CAQT. Any increase or decrease in the window size of this data item is least likely to affect the network overhead.
3) The aggregator determines the new window size of this data item such that the following equation is satisfied

$$\sum_{i=1}^{N}(w_i \times n_i) = T' \tag{14}$$

w_i and n_i are the window size (weighted cr value at data source) and weight of the ith data item and T' is the relative threshold. The aggregator has information of the window size that are present at the server. Using this information, the aggregator calculates the new window w_s of the slowest data item as:

$$w_s = \frac{T' - (\sum_{i=1}^{N} w_i \times n_i - w_j \times n_j)}{n_s} \tag{15}$$

where w_j is the window size of the data item that was pushed (that caused this recalculation of windows).

Thus two windows: one for the data item that was pushed and the one for the data item that is the slowest; are informed to the corresponding data sources. However, in this technique, if the data value that was pushed, has changed by a very large amount, then it can happen that the w_s calculated using equation 15 can be negative. This implies that with the given window sizes, mearly changine two window sizes might not ensure that equation 14 is satisfied. In such a case the windows of all the data items (i.e., the weighted cr values) are sent to the corresponding sources.

The windows in $PushCEA$ are double sided, with a lower part and and upper part. An optimization that can be done in case of $PushCEA$ is to have single sided windows. If the CAQT is below the threshold, then the aggregator is only interested in those changes that make the CAQT cross the threshold i.e., changes that increase the value of the CAQT. Hence instead of having double sided windows, the source can store only single sided windows. A single sided window can be assumed to be a window having one of the parts as ∞ or - ∞. Thus there can be the following two cases:

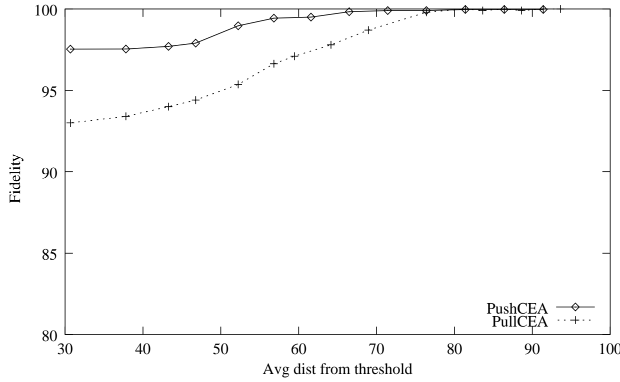
1) If the lower bound is closer to the threshold the bounds are set as $< p_i - wc_i^{'}, \infty >$.
2) If the upper bound is closer to the threshold the bounds are set as $< -\infty, p_i + wc_i^{'} >$.

This will lead to savings in space at the source as well as less computation overhead at the source. Another advantage to this is that this approach will reduce the network overhead. However the flip side to this is that this might lead to a drop in the fidelity which will be explained in the performance section.

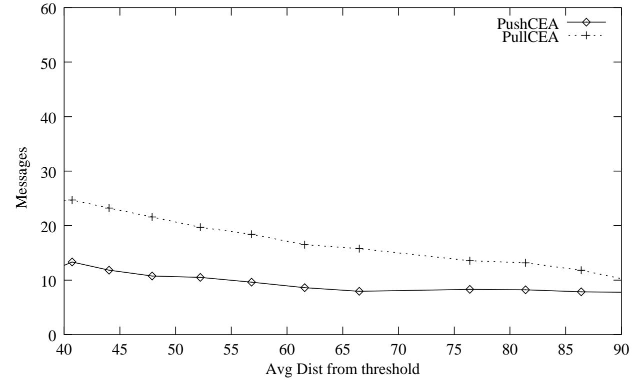
A. *Performance of PushCEA*

In this subsection we evaluate the performance of $PushCEA$. Fig 3 compares the fidelity and network overhead of $PushCEA$ with that of $PullCEA$. It can be observed that the network overhead of $PushCEA$ is significantly less than that of $PullCEA$. This is due to the fact that $PushCEA$ pushes only those data values that cross the coherency windows. $PullCEA$ on the other hand guesses the next time when the data value will change by static cr. This estimate is not always accurate and hence it leads to a lot of unnecessary pollings as shown by the graphs. However, note that the network overhead of $PushCEA$ is affected by the size of the CAQT (explained later in section IV-C). In these experiements the CAQT consisted of four data items. One interesting thing shown in the graph is that the fidelity offered by $PushCEA$ is not 100 %. Ideally as shown in the previous section, the fidelity should have been 100 %. This loss of fidelity can be explained with the following example: Consider a CAQT with two data items. Let the window of the first data item be $< 10, 20 >$ and that of the second be $< 20, 30 >$. The value cached at the proxy is 15 and 25 respectively. Let the relative threshold be 100 and the weights of the two data itmes be 10. Now if data item 2 changes to 34, then this will be pushed to the proxy. The proxy calculates the new value of the CAQT using the cached value of data item 1. Now with this cached value, the new relative threshold is 10. But when this push happened, if the value of data item 1 was 18 then the CAQT had already crossed the threshold and the proxy was unaware of this resulting in a violation. The proxy will become away only after the new windows are sent to the server and there will be push for data item 1. However such occurrences are rare as is shown by our experimental results.

Figure 4 compares the performance of $PushCEA$ with optimization (i.e., with single sided windows) and $PushCEA$ without optimization. The graphs show that the optimization succeeds in reducing the network overhead by a significant amount. However, this also leads to a drop in the fidelity. The reason for this drop in fidelity can be attributed to the fact that, in the absence of a push from the source, the aggregator assumes that the cached data value is equal to the data value at the source. In the case of a single sided window, if the data value changes in the direction where the window limit is ∞, then there is a lot of difference between the data value cached at the proxy and the value at the source. Lets consider a case of a CAQT of two data items. If the value of the first data item moves in the direction where the limit is ∞, then the

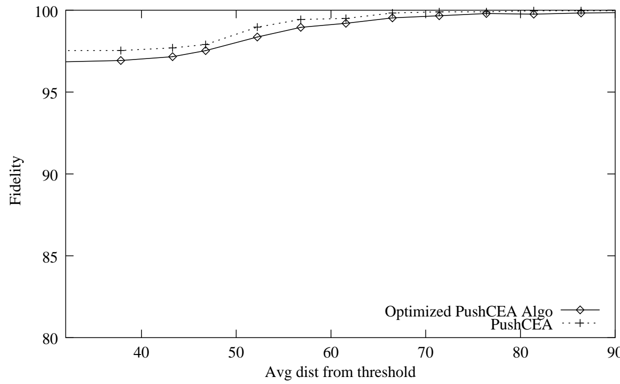


(a) Fidelity

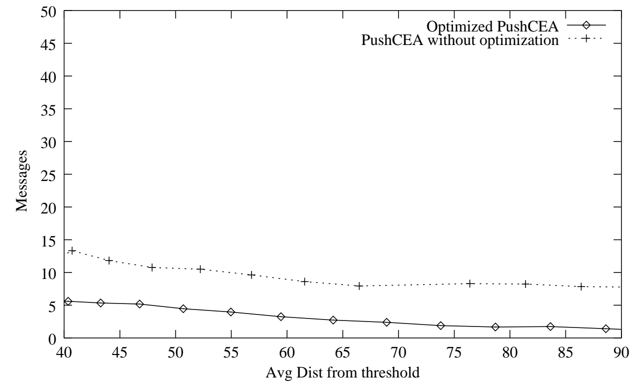


(b) Network Overhead

Fig. 3. Comparison of network overhead and fidelity for $PushCEA$ and $PullCEA$



(a) Fidelity



(b) Network Overhead

Fig. 4. Comparison of the $PushCEA$ with and without optimization

aggregator is not informed about this change. Now, in such a case if the value of data item 2 moves towards the threshold by a very large value, then the aggregator will calculate the new value of the CAQT based on the cached value of data item 1. Due to this the aggregator might conclude that the CAQT has crossed the threshold, which might not be the case. Hence there is a drop in the fidelity of the CAQT in case of one sided windows.

IV. A HYBRID CAQTS EXECUTION APPROACH ($HyCEA$)

$PushCEA$ has an edge over the $PullCEA$ because of the following:

- It can offer very high fidelity very close to 100%.
- It can be deployed over a multiple layer network where there are multiple aggregators, one aggregator serving the other. If we try to do the same with a pull based approach, the resulting fidelity is much lower than that of push. If the expected fidelity over a push connection is f then the expected fidelity over a series of n push connections would be f^n. Now if we have fidelity for a push connection as f_1 and fidelity for a pull connection as f_2 then $f_1 > f_2 \Rightarrow f_1^n >> f_2^n$. Thus, in case of pull where fidelity for individual connection is f_2, the fidelity loss over a series of connections, (1 - f_2^n), would be much larger.

However, this approach is not scalable due to the load it imposes on the source in terms of the number of the data items that the source needs to keep track of as well as the computation overhead. To overcome the *scalability* problem, the aggregator can resort to the use of pull based service for some of the data items and push for the rest. This helps in reducing the amount of state space at the source and shifting some of the computation overhead to the aggregator. Considering Equation 5 if we are able to reduce the number of data items requiring push we will save on the state space required by these data items. Thus the state space required then would be

$$u \times (B(d,w)) \; + c \times (B(cs)) \tag{16}$$

where u (the number of data items having push connection) is less than N (total number of data items in the CAQT). This leads to the idea of combining pull and push based approaches to produce a Hybrid CAQTs Execution Approach ($HyCEA$).

A. Calculation of Coherency constraints for $HyCEA$

A straight forward way of using push and pull connections for a single CAQT is to divide the threshold into two parts

based on the contribution of the push and pull data items to the CAQT and to run the two algorithm independently of each other. However such an approach does not perform well, as it looses out the information about the data items being a part of a semantic unit. In such a case it can happen that the data items using $PushCEA$ might be below its apportioned threshold, and at the same time the data items using $PullCEA$ might be above the threshold. Therefore, $PullCEA$ will reduce the polling frequency of the pull data items and $PushCEA$ will assign very large coherency windows to the push stocks. However the CAQT as a whole might be very close to the threshold and there might be a loss in fideltiy. Hence the push and pull data items should be combined in an intelligent way such that there is no loss of information about the actual position of the CAQT.

In our technique, we calculate the TTR and the weighted cr of all the data items irrespective of the type of connection given to the data item. In $HyCEA$ the TTR for the pull stocks is calculated in the same manner as given in section II and the coherency windows are calculated as given in section III. In $HyCEA$ the aggregator keeps track of the coherency windows of each of the data items irrespective of the type of the connection of the data item. Every time that the aggregator pulls a new value for a pull data item, it compares the new value with its coherency windows. If the new value lies outside the window given to the pull data item, then there is a possibility that even if the data value of the push data items are within their windows, a fidelity violation can occur. The coherency windows for the push stocks were computed using the relative threshold value at the computation time. If the new value of the pull stock moves outside the window, then it signifies that there is significant shift in the value of the relative threshold and hence the aggregator recomputes the new coherency windows and informs them to the respective data sources. The important point to note over here is that the aggregator does this recomputation only if the new value of the data item lies outside its window.

B. Distribution of data items between $PullCEA$ and $PushCEA$

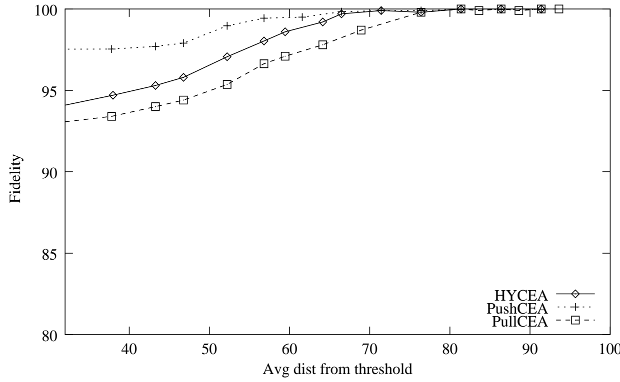
In the simplest approach, the allocation of the data items that use $PushCEA$ can be done statically depending on the contribution of the data item to the entire CAQT; those data items that have a higher initial value or whose weights are larger have a greater influence on the CAQT. Such data items should be maintained more accurately and hence are given push connections. In addition to this, the connection type ($PushCEA$ or $PullCEA$) also needs to be varied according to the dynamics of the data. The Push connections will have a higher fidelity and hence it makes sense to give this service to the data items with maximum fluctuation (hot data items). This helps in reducing the network overhead as the hot data items make a greater contribution to the network overhead. The aggregator keeps track of the average change in the data value for all the data items and the data items with the maximum change in their data value are given a push connection. The connection type is periodically reviewed and based on the dynamics of the data, the connection type of each data item is changed after fixed intervals of time. Thus this approach adapts to the dynamics of the data, but it requires additional history of the data items to be maintained at the aggregator.

C. Evaluation of $HyCEA$

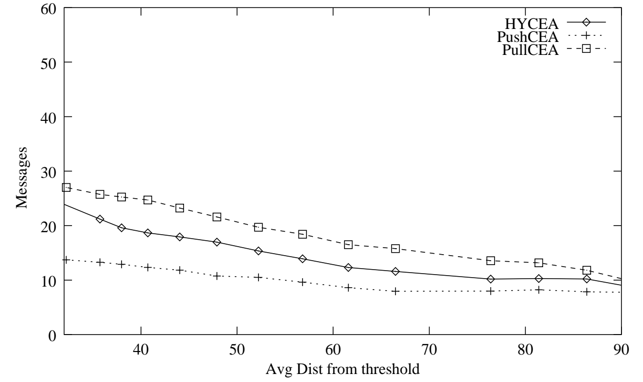
Figure 5 compares the fidelity and network overhead of the three algorithms namely $PullCEA$, $PushCEA$ and the $HyCEA$. In $PushCEA$, because of the state information at the source none of the interesting changes are missed by the aggregator. Hence the fidelity offered by $PushCEA$ is maximum. $PushCEA$ pushes only those changes that are required to maintain 100% *fidelity*. At the other extreme $PullCEA$ incurs a large number of unnecessary polls, as is evident from its high network overhead. $HyCEA$ is a judicious combination of the two approaches and hence has fidelity and network overhead between that of $PushCEA$ and $PullCEA$. Note that these graphs are for small size CAQTs. As the size of the CAQT increases, the network overhead for $PushCEA$ increases.

An interesting phenomenon similar to 'Thrashing' was observed in case of Push data items which were very volatile. In this, if the data item is very volatile and if the CAQT value is near to the threshold, then there is a very significant drop in the fidelity accompanied with a large increase in the network overhead. As explained in section III, contrary to expectations, there can be some loss of fidelity in $PushCEA$. If there is a change in the data value that takes the value outside its window, then it is pushed to the aggregator. At the same time if the value of the other data items are not centered around the window center, then this leads to a fidelity violation, which is detected when the new windows are sent to the data source. If such a case happens when the CAQT value is close to the threshold, then the possibility of violation recurring is more due to the reduced threshold. With the reduced window sizes, even a small change will give rise to a push and might possibly lead to a violation and further reduction in window size. Thus, one fidelity violation triggers the other and this continues till the CAQT is close to the threshold. The solution to this problem is a bit counter intuitive: reduce the window sizes further than what is required to maintain 100 % fidelity. This reduction is done as a fixed percentage of the window sizes in the ideal case (i.e., the one required to maintain 100 % fidelity). With this reduced window sizes, changes that otherwise would not have been pushed are now pushed to the aggregator. So consider a CAQT with 2 data items. A fidelity violation can occur if there is a push for data item 1 and if the data value 2 does not lie around the center of its window. If the window sizes are less, then if the data value of data item 2 is far away from its center then it will be pushed to the aggregator. In the earlier case, with large windows, this push would not have happened and it results in the loss of fidelity. Now even if there is a push for data item 1 the value of data item 2 wont be far off from its center and hence the fidelity is maintained.

Figure 6 shows the effect of the size of the portfolio on



(a) Fidelity



(b) Network Overhead

Fig. 5. Performance of $HyCEA$ with respect to $PullCEA$ and $PushCEA$

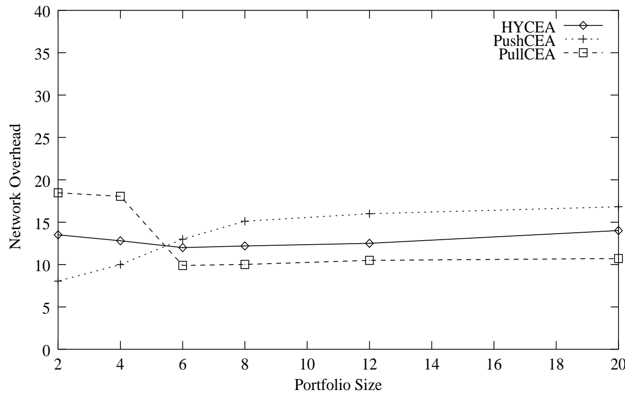


Fig. 6. Performance of the three algorithms

the network overhead. The figure shows that the network overhead of $PushCEA$ increases as the size of the portfolio increases. As the size of the portfolio increases, every time that a window size changes a minimum of two new windows have to be communicated to the data sources. In this simulation experiment, we assumed that all the data items are from different data sources. Hence with increasing size of the CAQT the number of messages to change to window sizes increases which contributes to the increase in the network overhead. An interesting result that is shown by the graph is that the network overhead of $PullCEA$ decreases with the increase in the size of the CAQT and later on it becomes more or less stagnant. When the size of the CAQT is small, the chances of a small change in the data value making an effect on the overall value of CAQT is very high. Hence even a small change in the data value might change the CAQT value by a large amount thereby reducing the relative threshold and in effect reducing the TTL of all the data items. Hence the network overhead for small CAQT is high. However for larger size CAQT, small data value changes do not have any effect on the overall value of the CAQT and hence the network overhead is less. The network overhead of the $HyCEA$ algorithm does not deteriorate as rapidly as the $PushCEA$ algorithm. This is due to the presence of the pull component in the CAQT. The experiment was conducted with 50% push connections. Hence there was no return message from the aggregator to the data source for 50% of the data items (that had a pull connection) which is responsible for the reduction in the network overhead. Thus the figure clearly shows that push does not scale for large CAQT size. $HyCEA$ scales well with large CAQT size and it offers better fidelity than $PullCEA$. The fidelity was not affect much by the size of the CAQT.

V. Extension to Other Aggregate Queries

Till now, we have focused only on sum-based CAQT. In this section, we will show how other queries like count, min, max, avg can also be computed using the existing setup. For computing a count based CAQTs, the data values can be replaced by the counts of the individual items and the weights of the data items depend on the preferences to these data items. In a simple case, these weights would be 1 for every data item. Similarly, average based CAQT involve computing the sum based CAQT and dividing the consequent value by the count of data items in the query. In other words, what this implies is that we just need to blow up the precision of the average based CAQT by the count of data items in the query and run a sum based CAQT with the blown up constraint. Since each of the queries use the same set up and are themselves based on the sum based CAQT, the performance results are bound to be same.

Min and Max queries can also be computed using the same setup. For example, let us take a Min query. In this case, the data items closer to the threshold or in other words having minimum value or close to minimum value can be assigned push connection to be tracked more precisely and the others can be assigned pull connections. In this case, the weights in the sum based CAQT will be just 1 and the sum function at the aggregator would be replaced by the min function.

VI. Related Work

The problem of consistency maintenance between a data source and cached copies of the data was first studied in [1]. The paper discusses techniques where a source pushes updates to users based on expiration times associated with the data. Since then numerous efforts have investigated push-based dissemination techniques [2], [3], [5], [4], [12], [16], [21], however, either they are not modeled for continuously changing data or if so, they do not consider the semantics of the query (like the direction in which the aggregate query is moving). [21] is a recent work in this direction but like any other push based approach, it is susceptible to scalability and resiliency issues. There is no provision to offload the server load to th aggregator.

The problem of consistency maintenance has also been studied in the context of web caching and several techniques such as client polling [10], adaptive time-to-live (TTL) values [19], source invalidation [14] and leases [27], [25] have been proposed. These efforts typically assume that cached data is modified on slow time scales (e.g., tens of minutes or hours) and are less effective at maintaining consistency of rapidly changing data cached at proxies. Caching of *dynamic* content has been studied in [13] wherein a scheme based on push-based invalidation and dependence graphs is proposed, while another effort has focused on availability and *scalability* by adjusting coherency requirement of data items [28]. Neither effort has explicitly addressed coherency maintenance for efficiently executing queries at a aggregator.

Mechanisms for disseminating fast changing documents are proposed in [18], [20], [17]. The difference between these approaches and ours is that they disseminate *all* updates to the document using Multicast, while we selectively disseminate updates based on the coherency requirements of a data item. The concept of approximate data at the users was studied in the context of stock price dissemination in [23]; the approach focuses on individual data items and does not address the additional mechanisms necessary to track an aggregate of data items. A push based approach for the execution of aggregation queries over databases has been proposed in [22].

Finally, the CAQTs are a subset of the general class of continuous queries over dynamically changing data [15], [8]. Continuous queries in the Conquer system [15] are tailored for heterogeneous data, rather than for real time data, and use a disk-based database as its back end. NiagaraCQ [8] focuses on efficient evaluation of queries as opposed to temporally coherent data dissemination to proxies (which in turn can execute the continuous queries). A aggregator-based approach leads to better *scalability*.

In summary, none of the research efforts have focused on the infrastructure necessary for the temporally coherent dissemination of time-varying data for efficient execution of CAQTs at a aggregator. Specifically, our approach

1) is executable at the aggregator,
2) dynamically assigns coherency requirements to continuously changing data items based on both direction as well as the speed of changes and
3) uses adaptive dissemination mechanisms [9] like the ability to to switch between pull and push for executing the CAQT with high fidelity and a low cost.

VII. Conclusions and Future Work

On-line decision making often involves processing significant amounts of time-varying data. In such environments, decisions are typically made whenever a continuous query over a set of data items satisfies a threshold criterion. In this paper, we investigated adaptive data dissemination techniques where such *Continuous Aggregate Queries with Thresholds* access data from multiple sources. Key challenges in supporting such queries lie in meeting users' consistency requirements while minimizing network and source overheads, without the loss of fidelity in the responses provided to users. Meeting these required us to solve two subproblems: (1) deriving the coherency requirement of each of the data items used by the aggregator, and (2) ensuring that the (derived) coherency requirement associated with each data item is satisfied. We showed that to address the former, the coherency requirement associated with a data item should be derived from the overall contribution of the data item to the CAQT's result and the threshold value associated with the CAQT. With regard to the latter, we demonstrated that, in the case of CAQTs, using standard techniques, such as pull and push, for individually retrieving each data item are not efficient - *additional mechanisms must be employed which exploit the fact that the data items comprising a CAQT form a semantic unit*. A hybrid approach was developed that *pulled* some of these items from their sources, and resorted to *push* for the rest. This hybrid approach strikes a balance between the trade offs of *scalability*, fidelity , and network overheads (Table III). The algorithms achieve good fidelity at low cost by considering the data items in a CAQT as a semantic unit, since this reduces the dissemination overheads if the data values are changing such that there is very little chance of a CAQT's threshold being exceeded.

References

[1] R. Alonso, D. Barbara, and H. Garcia-Molina. Data caching issues in an information retrieval system. *ACM Trans. Database Systems*, September 1990.

[2] S. Acharya, M. J. Franklin, and S. B. Zdonik. Balancing push and pull for data broadcast. In *Proceedings of the ACM SIGMOD Conference*, May 1997.

[3] E. Amir, S. McCanne, and R. Katz. An active service framework and its application real-time multimedia transcoding. In *Proceedings of the ACM SIGCOM Conference*, September 1998.

[4] G. Banavar, T. Chandra, B. Mukherjee, J. Nagarajarao, R. E. Strom, and D. C. Sturman. An effi cient multicast protocol for content-based publish-subscribe systems. In *International Conference on Distributed Computing System*, 1999.

[5] A. Bestavros. Speculative data dissemination and service to reduce source load, network traffi c and service time in distributed information systems. In *International Conference on Data Engineering*, March 1996.

[6] P. Cao and S. Irani, Cost-Aware WWW Proxy Caching Algorithms., *Proceedings of the USENIX Symposium on Internet Technologies and Systems, December 1997.*

[7] A. Chankhunthod, P. B. Danzig, C. Neerdaels, M. F. Schwartz and K. J. Worrel A Hierarchical Internet Object Cache., *Proceedings of the 1996 USENIX Technical Conference, January 1996.*

TABLE III

COMPARISON OF DIFFERENT ALGORITHMS

Algorithm	Resiliency	Fidelity	Network Overhead	Computation Overhead (on Source)	State Space (on Source)
$PullCEA$	Graceful Degradation	Low	High	Low	Low
$PushCEA$	Low	High	Low-High	High	High
$HyCEA$	Delayed Graceful Degradation	Medium	Medium	Medium	Medium

[8] J. Chen, D. Dewitt, F. Tian, and Y. Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data*, May 16-18 2000.

[9] P. Deolasee, A. Katkar, A. Panchbudhe, K. Ramamritham, and P. Shenoy, Adaptive Push-Pull: Dissemination of Dynamic Web Data *10th International World Wide Web Conference , Hong Kong, May 2001.*

[10] A. Dingle and T. Partl. Web cache coherence. In *Proc Fifth Intnl WWW Conference*, May 1996.

[11] V. Duvvuri, P. Shenoy and R. Tewari, *Adaptive Leases: A Strong Consistency Mechanism for the World Wide Web. InfoCom March 2000.*

[12] J. Gwertzman and M. Seltzer. The case for geographical push caching. In *Proceedings of the Fifth Annual Workshop on Hot Operating Systems*, May 1995.

[13] Arun Iyengar and Jim Challenger. Improving web server performance by caching dynamic data. In *USENIX Symposium on Internet Technologies and Systems*, 1997.

[14] C. Liu and P. Cao. Maintaining strong cache consistency in the world wide web. In *Proceedings of ICDCS*, May 1997.

[15] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knowledge and Data Engg.*, July/August 1999.

[16] G. R. Malan, F. Jahanian, and S. Subramanian. Salamander: A push based distribution substrate for internet applications. In *Proceedings of the USENIX Symposium on Internet Technologies and Systems*, December 1997.

[17] P. Rodriguez and E. Biersack. Continuous multicast push of web documents over the internet. *IEEE Network Magazine, vol. 12, 2, pp. 18–31, Mar-Apr*, 1998.

[18] Pablo Rodriguez, Keith W. Ross, and Ernst W. Biersack. Improving the WWW: caching or multicast? *Computer Networks and ISDN Systems*, 1998.

[19] R. Srinivasan, C. Liang, and K. Ramamritham. Maintaining temporal coherency of virtual warehouses. In *Proceedings of of the 19th IEEE Real-Time Systems Symposium*, December 1998.

[20] P. Rodriguez, E. Biersack, and K. Ross. Automated delivery of web documents through a caching infrastructure. *Technical Report, EURECOM, June*, 1999.

[21] C. Olston, J. Jiang and J. Widom Adaptive Filters for Continuous Queries over Distributed Data Streams. In *Proceedings of the ACM SIGMOD Conference*, May 2003.

[22] C. Olston, and J. Widom Offering a Precision-Performance Tradeoff for Aggregation queries over Replicated Data. In *Proceedings of the Very Large Databases*, September 2000.

[23] C. Olston, B. T. Loo, and J. Widom Adaptive precision setting for Cached Approximate Values. In *Proceedings of the ACM SIGMOD Conference*, May 2001.

[24] P. Wilmott, S. Howison, and J. Dewynne The Mathematics of Financial Derivatives. New York, *Cambridge University Press, 1995*.

[25] Jian Yin, Lorenzo Alvisi, Mike Dahlin, and Calvin Lin. Hierarchical cache consistency in a WAN. In *USENIX Symposium on Internet Technologies and Systems*, 1999.

[26] Yoodlee.com launches a powerful, new consumer service to help you take control of your life online. *Press Release*, September 28, 1999.

[27] Haobo Yu, Lee Breslau, and Scott Shenker. A scalable web cache consistency architecture. In *SIGCOMM*, pages 163–174, 1999.

[28] H.Yu and A.Vahdat. Design and evaluation of a continuous consistency model for replicated services. In *Proceedings of OSDI*, October 2000.