Towards collaborative data reduction in stream-processing systems

Ming Li* and David Kotz

Department of Computer Science,
Institute of Security Technology Studies,
Dartmouth College,
Hanover, NH 03755, USA
E-mail: ming.li.adv08@alum.dartmouth.org
E-mail: kotz@cs.dartmouth.edu
*Corresponding author

Abstract: We consider a distributed system that disseminates high-volume event streams to many simultaneous monitoring applications over a low-bandwidth network. For bandwidth efficiency, we propose a collaborative data-reduction mechanism, 'group-aware stream filtering', used together with multicast, to select a small set of necessary data that satisfy the needs of a group of subscribers simultaneously. We turn data-compressing filters into group-aware filters by exploiting two overlooked, yet important, properties of monitoring applications:

1 many of them can tolerate some degree of 'slack' in their data quality requirements
2 there may exist multiple subsets of the source data satisfying the quality needs of an application.

We can thus choose the 'best alternative' subset for each application to maximise the data overlap within the group to best benefit from multicasting. We provide a general framework that treats the group-aware stream filtering problem completely; we prove the problem NP-hard and thus provide a suite of heuristic algorithms that ensure data quality (specifically, granularity and timeliness) while collaboratively reducing data. The framework is extensible and supports a diverse range of filters. Our prototype-based evaluation shows that group-aware stream filtering is effective in trading CPU time for data reduction, compared with self-interested filtering.

Keywords: stream processing; data reduction; application-level multicast; group-aware filters; networked systems; wireless mesh networks; monitoring applications.

Reference to this paper should be made as follows: Li, M. and Kotz, D. (2009) 'Towards collaborative data reduction in stream-processing systems', *Int. J. Communication Networks and Distributed Systems*, Vol. 2, No. 4, pp.375–400.

Biographical notes: Ming Li studies problems in the general areas of distributed systems and information management. Specifically, she focused her research in pervasive computing, stream processing and networked systems during the past five years. She is a member of ACM, IEEE and USENIX. She received her PhD in Computer Science from Dartmouth College.

Copyright © 2009 Inderscience Enterprises Ltd.

David Kotz is a Professor of Computer Science at Dartmouth College in Hanover NH. During the 2008–2009 academic year, he is a Visiting Professor at the Indian Institute of Science, in Bangalore India and a Fulbright Research Scholar to India. At Dartmouth, he was the Executive Director of the Institute for Security Technology Studies from 2004–2007. His research interests include security and privacy, pervasive computing for healthcare and wireless networks. He has published over 100 refereed journal and conference papers. After receiving his AB in Computer Science and Physics from Dartmouth in 1986, he completed his PhD in Computer Science from Duke University in 1991 and returned to Dartmouth to join the faculty. He is an IEEE Fellow and a Senior Member of the ACM, and a member of the USENIX Association. For more information see http://www.cs.dartmouth.edu/~dfk//.

1 Introduction

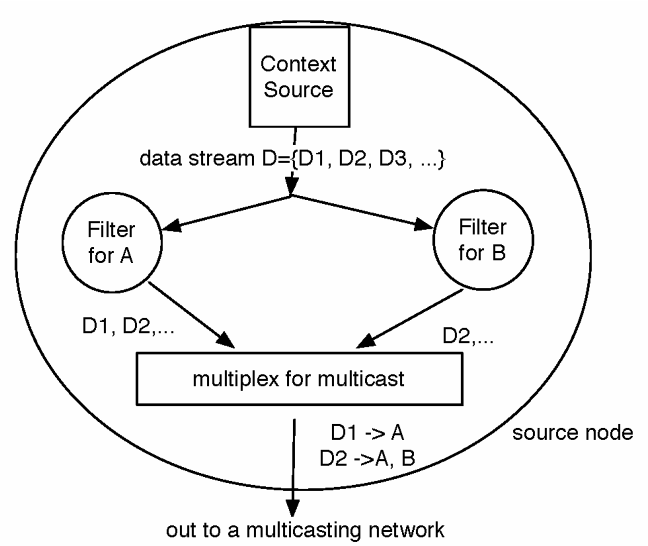
Recent years have seen data-intensive applications that feed on near-real time ‘context’ information, such as location, environmental status and surrounding resources, collected from distributed data sources leveraging sensor networks. Those data sources, such as click-streams, stock quotes and sensor data, are often characterised as fast-rate high-volume ‘streams’ (Babcock et al., 2002a). *Distributed stream-processing* systems acquire and aggregate high-resolution data for *monitoring applications* that come from many different domains; these applications range from RFID-based inventory management, pipeline monitoring for civil engineering, real-time stock-price analysis, mining of web click-streams, habitat monitoring, to vital-sign monitoring and medical triage.

In this paper, we consider a distributed stream-processing system that disseminates high-volume data streams to many simultaneous monitoring applications over a low-bandwidth network. At the scene of a large fire, we imagine, fire-spread prediction may require sub-second updates on temperature, wind speed and direction from the sensor networks deployed near the fire; command-and-control applications need frequent updates on first responders’ locations. Systems that disseminate data for those monitoring applications often use wireless networks for rapid deployment and cost-effectiveness. In the emergency-response scenario, such an infrastructure can be a wireless mesh network formed by computers on police cars or fire trucks on the scene. It is well recognised that the effective bandwidth of a wireless network is usually much lower (more than ten times less) than its link capacity (Akyildiz et al. 2005; Strix Systems, 2005), and that the high-volume data acquisition needs of monitoring applications may push the envelope of the bandwidth-constrained network. For bandwidth efficiency, we propose a collaborative data-reduction mechanism, *group-aware stream filtering*, used together with multicast, to select a small set of necessary data that satisfy the needs of a group of subscribers simultaneously.

Our group-aware stream filtering approach hinges on the fact that many applications that subscribe to the same data source may need different portions of the data. This may be due to applications’ different requirements on data granularity, varying capacity of their data-receiving nodes, or other factors. It thus reduces bandwidth consumption to deploy a filter on the source node that chooses only the data important to its corresponding application before transporting the data. (The output of a filter in our

consideration is a subset of its input data.) Further, if we multiplex and then multicast the outputs of those source-sharing filters, we can eliminate redundant communication in the network. Such a combination of data filtering and multicasting is illustrated in Figure 1: two applications, *A* and *B*, share the same data source *D*, but each application's filter selects a different subset on the source node. The multicast protocol allows us to label each tuple with the list of the applications that should receive that tuple; thus each tuple is transmitted at most once on any link. In this setting, the objective of our group-aware stream filtering is to make each filter 'group-aware' such that the combined outputs of the filters is minimised in size, while still satisfying the applications' needs simultaneously. This reduced total output can better reap the benefit of multicast.

Figure 1 Filtering for multicasting



The key characteristic of a group-aware filter is its capability to select an output from multiple quality-equivalent potential outputs that satisfy the requirements of its application. For many exploratory monitoring applications, data streams provide a series of state updates about an interested environment, and filters that compress the state updates based on applications' data granularity are of special interest for saving network bandwidth. Those compressing filters can be turned group-aware based on two overlooked, yet important, properties of the applications:

1 many applications can tolerate some degree of 'slack' in their data quality requirements

2 there may exist multiple subsets of the source data satisfying the quality needs of an application.

We can thus make filters group-aware and choose the 'best alternative' subset for each application, maximising the data overlap within the group to best benefit from multicasting. This paper makes the following key contributions:

- We thoroughly treat the group-aware stream filtering problem by formally proving its NP-hardness, and by providing a suite of heuristics-based algorithms, which ensure data timeliness and data granularity. Thus our approach is quality-managed.
- We provide a general framework for group-aware filtering, for a variety of data-selection operators, such as delta-compression (DC) filters and stratified sampling filters. We describe application scenarios where group-aware filtering may be beneficial.
- We built a prototype system for evaluation. Our results, based on real-world data sets, show that group-aware filtering can effectively save bandwidth with low CPU overhead when compared with self-interested filtering. We also evaluate the effect of each algorithm on temporal freshness of the data.

In the next section, we describe the foundation of group-aware filtering. In Section 3, we formally define the problem and prove its NP-hardness, and introduce the framework and algorithms for group-aware stream filtering. In Section 4, we evaluate our approach with a prototype system. In Section 5, we show that our framework is extensible to support diverse filters for different application scenarios. We discuss related work in Section 6 and summarise in Section 7.

2 Two key observations

We base our filtering approach on two key observations about data-quality requirements of monitoring applications. Data quality is normally measured as the *accuracy*, *granularity*, *timeliness* and *completeness* of the data. Implications of data quality at different parts of the data acquisition process may be different. For filtering, ensuring accuracy and completeness may mean that filters must not tamper with the input data (enforcing accuracy), and that filters must output all tuples in the input data stream that satisfy applications' needs (enforcing completeness). We assume that the chosen filters can always ensure these two qualities. The filters' main job is to select an appropriate subset of input data that meets the applications' data granularity requirement. For example, an application would like to get a temperature reading of a place whenever the reading has changed by n degrees. This n-degree data granularity requirement can be enforced by a *DC* filter that removes values that have changed less than n units from the filter's previous output, in effect compressing the stream data at 'delta', in this case n, units. (We consider other types of filters in Section 5.) The higher the data granularity (in the case of DC filters, the lower the 'delta' interval), the more output a filter should normally produce. Data granularity thus directly affects bandwidth consumption. The timeliness requirement at the filter can be measured by the amount of delay introduced by filtering. The faster a filter processes and outputs the data, the more timely is the data delivered to applications.

1. First observation. *Monitoring applications may tolerate some degree of 'slack' in their data quality*. Consider a temperature source and DC filtering, for example. Given a time-ordered nine-tuple sequence from the source, {0, 35, 29, 45, 50, 59, 80, 97, 100},[1] the output that satisfies compression at 50-unit

granularity is {0, 50, 100}. We recognise that applications may find it harmless to tolerate a small deviation from the ideal compression granularity in the output. For instance, the application may be able to tolerate a maximum of 10 degrees 'slack' with regard to its ideal 50 degrees granularity requirement. We denote such filters as a *(slack, delta) DC filter*, which selects data at delta-unit with slack-unit of quality deviation.

2 Second observation. *There may exist more than one sequence from a data source that can satisfy an application's approximate quality requirements*. In the previous example, if the application tolerates a maximum of 10 degrees slack in the 50 degrees compression granularity, it is easy to validate that the following sequences each satisfy the approximate granularity requirements as well: {0, 45, 100}, {0, 59, 100}, {0, 50, 97}, {0, 45, 97}, {0, 59, 97}, as 45, 59 are close by tuples within 10 degrees deviation from 'ideal' output 50 after initial output 0 and 97 from 'idea' output 100 as a third output.

3 Group-awareness. Let us call the above DC application *A*. Suppose application *B* shares the same source as *A* and tolerates a maximum of 5 degrees slack in a 40 degree compression granularity. By the above definitions, it is easy to validate that the following sequences satisfy *B*'s requirements: {0, 45, 97}, {0, 50, 97}, {0, 50, 100}, {0, 45, 100}.

Individually, *A* may choose {0, 50, 100} as its output; *B* may choose {0, 45, 97} as its output; there are thus five tuples to output when multiplexing the output streams for multicasting. If *A* and *B* are aware of each other's filtering needs, and both decide on, say, {0, 50, 97} as their individual output, then only three tuples need to be multicast to *A* and *B* to satisfy both filtering requirements. In effect, the 'group-awareness' reduces the bandwidth demand by two tuples.

3 Framework

In this section, we show that the group-aware stream filtering problem is NP-hard and provide a framework using a suite of heuristics-based algorithms to solve the problem approximately.

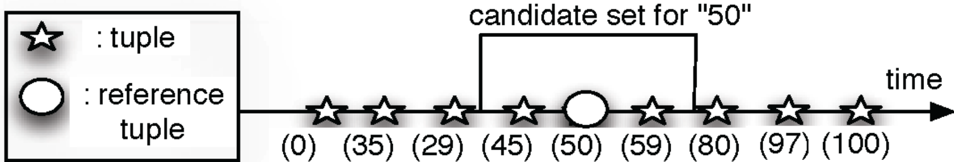
3.1 Problem definition

We assume that the input data stream is a time-ordered continuous sequence of tuples. Each tuple consists of several attributes, including a unique time stamp. We abstract our group-aware stream filtering method into two stages: In the first stage, *compute candidate outputs*, each filter processes the input stream and computes a set of candidate outputs; in the second stage, *decide on final outputs*, an output decider chooses a candidate output from each filter to be multicast. For a continuous stream, group-aware filtering iteratively goes through these two stages, processing one segment at a time.

3.1.1 Reference-based candidate sets

For the first stage, there exist many domain-specific ways for quality-slack tolerating filters to compute the candidate outputs. Here we introduce a *reference-based approach* to find candidate outputs for filters. The idea is for the filter to compute a candidate set for each tuple that the self-interested filter would select. We call the tuples that would be chosen by a self-interested filter *reference tuples*. Choosing any tuples from the candidate set corresponding to a reference tuple would be quality-equivalent to choosing the corresponding reference tuple for the output. Figure 2 shows how a DC filter that can tolerate ten units slack in 50-unit compression can select a vicinity of tuples around the reference tuple, here tuple 50, that are no more than ten units from the reference tuple, to form its candidate set. In this case, tuples whose values are 45, 50, 59 are within the ten units vicinity of, and contiguous with, the reference tuple 50 and thus make the candidate set. We assume for now that every candidate set is finite or *closable*.

Figure 2 Candidate set of a reference tuple (slack = 10)



Below, we define the group-aware filtering problem formally (In Appendix A, we prove its NP-hardness by reducing the problem to the minimum hitting-set problem). The minimum hitting-set problem has been studied extensively in the computer science literature. It is proved that the greedy algorithm produces a $\rho(n)$ approximation to the optimal solution (Cormen et al., 2001), where $\rho(n) = H(max\{|C|: C$ is a set in the hitting-set problem$\})$ and where *H* is a harmonic function and n is the total number of sets in the problem. We can apply this bounded approximation algorithm directly to the group-aware filtering problem.

Problem Definition 1. Given an input stream segment *S*, n filters F_1, F_2, .., F_n in the group, and a collection *C* containing all candidate sets produced by the filters. The objective is to pick a tuple o_j from each candidate set in *C*, such that the set $Output = \bigcup_j \{o_j\}$ has minimal size.

An important property of computing candidate sets based on reference points is that the selectivity (or compression ratio) of each group-aware filter is exactly the same as that of the corresponding self-interested filter, as the number of candidate sets of a group-aware filter is the same as the number of reference outputs selected by the corresponding self-interested filter. Yet, thanks to overlapping of candidate sets, the size of the overall outputs of group-aware filters is smaller or no bigger than that of the overall outputs of the corresponding self-interested filters of the group. In other words, group-aware filtering performs no worse than self-interested filtering, in term of reduction of bandwidth consumption.

3.2 Region-base stream segmentation

Notice the problem we defined assumes that the input is a finite-length time series. For a continuous event stream that is potentially infinite in length, we consider a group-aware filtering optimisation problem for all its finite prefixes of a time-ordered input data sequence.

For a long stream, it is not time-efficient, if not impossible, to collect all the data before applying filtering algorithms. So we consider the problem of whether there exists a way to segment the input time series in such a way that the segmentation does not affect the optimality of the solution. Here we propose *region-based segmentation* for applying minimum hitting-set algorithms.

The intuition behind region-based segmentation is that we would like to divide the collection of candidate sets in the problem into time-wise-clustered subsets such that there is no overlapping of any candidate set in a subset with any candidate set in another subset. This way, minimum hitting-set algorithm can be time-progressively applied to each subset to produce solutions that can be added to the overall solution without affecting the optimality of the total solution. Further, the segmentation scheme ensures that each subset is minimised in size, so that minimum hitting-set algorithm can be applied to each subset as early as possible. We call each such minimised subset a *region*. Appendix B shows the formal definition of region and two important properties of region-based stream segmentation:

1 region-based segmentation preserves solution's optimality, if optimal

2 it preserves the approximate ratio of sub-optimal solutions.

Figure 3 Two regions for three DC filters

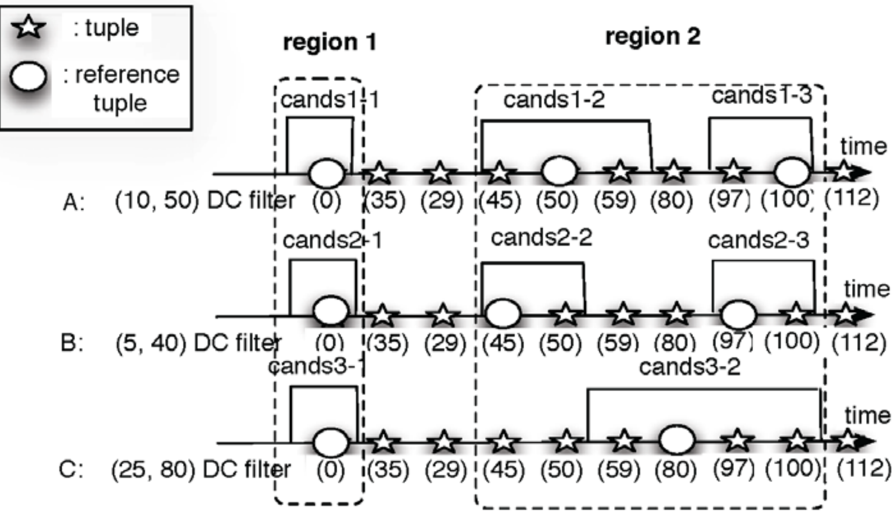


Figure 3 shows two regions for three DC filters' candidate sets: $region_1$ = {cands1-1, cands2-1, cands3-1}, $region_2$ = {cands1-2, cands2-2, cand3-2, cands1-3, cands2-3}. Each filter computes its candidate sets one after another. In this example, we assume that the closure of a candidate set is signalled by the first tuple that is not a candidate. Thus,

cands2-1 is closed when tuple 35 comes, as 35 is more than 5 units away from the reference tuple 0. Filter B now anticipates the next reference tuple to be at least 40 and it admits tuple 35 into its next candidate set, as the tuple is within 5 units away from 40. When tuple 29 comes, it is not qualified as a candidate tuple and it also invalidates tuple 35's candidacy, as in this example we assume that a candidate set is made of tuples that comes consecutively in time. When tuple 45 comes, it is at least 40 units away from the previous reference tuple and thus is admitted into B's candidate set. B admits tuple 50 and closes the candidate set when tuple 59 comes. Note that before a candidate set is closed, a filter has the ability to adjust its current candidate set by removing invalid candidates, for instance when a filter discovers the real reference tuple in a candidate set, or find unqualified tuples. It is easy to verify that adding any candidate set outside a region to the region will invalidate the region, as the added candidate set has no overlapping with the rest of the sets in the region.

Now we introduce REGION-BASED-GREEDY-FILTERING, a region-based greedy algorithm, for a continuous stream *S* in Figure 4. First, assume that we have instantiated each filter according to its specification from each application. A filter specification specifies the type and parameters of the filter, and how its internal state should be initiated and updated. We use a global object *globalState* to coordinate the filtering. The global state mainly consists of

1 the *group utility* of each tuple, which counts the number of filters that have included the tuple in their candidate set

2 the current *region* that keeps track of the connected candidate sets since the last region.

Figure 4 Region-based greedy algorithms for group-aware stream filtering.

```
REGION-BASED-GREEDY-FILTERING(S)
 1  while ( (currentTuple ← S.getNextTuple()) != null)
 2      do for each filter f in the group
 3            do if f.isAdmissible(currentTuple)
 4                 then ▷ first stage: admit candidates
 5                      f.candidateSet.add(currentTuple)
 6                      f.state.update(currentTuple)
                        ▷ increment group utility of currentTuple
 7                      globalState.groupUtility.increment(currentTuple)
 8               if f.candidateSet.closed(currentTuple)
 9                 then globalState.addClosedCandidateSet(f.candidateSet)
        ▷ second stage: if current region is ready, decide output based on globalState
10      if (region ← globalState.getCurrentRegion()) != null)
11        then ▷ apply greedy algorithm to the region to decide the output
12             output ← GREEDY-HITTING-SET(region, globalState.groupUtility)
               ▷ multicast output
13             multicaster.send(output)
```

Each filter uses its *isAdmissible* (line 3) method to decide whether a tuple is admissible to its candidate set. If so, the tuple is added to the filter's candidate set (line 5), and the tuple's group utility is incremented in the *globalState* (line 7). A filter's *isAdmissible* method may trigger the filter to find the next reference tuple as internal state for

admitting its candidate tuples. Next, if the filter finishes computing the current candidate set (line 8) when detecting that the current tuple does not belong to the current candidate set, the filter's current candidate set is *closed*. It then checks whether all connected candidate sets are closed. This check is done at the *globalState*, which keeps track of currently closed candidate sets not included in the previous region and tracks the group utilities of each tuple. If the utility of any tuple in a closed candidate set is greater than the number of currently closed candidate sets, then the region is not closed, as there must be a not-yet-closed candidate set admitting this tuple (line 10). When the current region is closed, it consists of all the closed candidate sets that are connected. Next, we apply a greedy hitting-set algorithm, GREEDY-HITTING-SET in Figure 5, to the current region (line 12) and send the solution for multicast (line 13). The solution contains a set of tuples chosen from the region that have high group utilities and hit all candidate sets in the region.

Figure 5 Greedy hitting-set algorithms

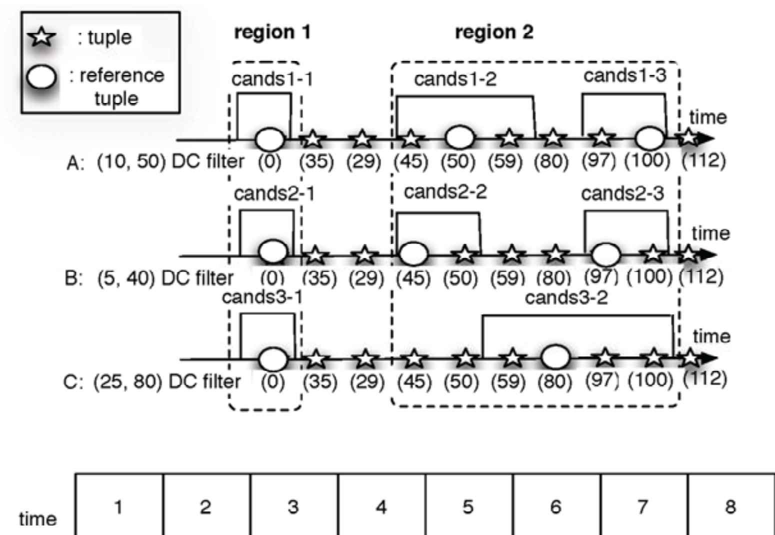
```
GREEDY-HITTING-SET(region, groupUtility)
1  resultSet.init(∅)
2  while (region.hasMoreCandidateSets())
   ▷ greedily pick the tuple with max groupUtility; use time stamp to break ties, if there are any
3      do maxUtilityTuple ← getMax(groupUtility)
4         resultSet.add(maxUtilityTuple)
          ▷ get all the candidate sets that are hit by maxUtilityTuple
5         hitCandidateSets ← region.removeAllCandidateSetsHitBy(maxUtilityTuple)
          ▷ decrement groupUtility for each tuple in the hitCandidateSets
6         groupUtility.decrUtilityFor(hitCandidateSets)
7  return resultSet
```

GREEDY-HITTING-SET (in Figure 5) picks the tuple with the highest group utility (line 3). If multiple tuples have the same highest utility, we use tuples' time stamps to break the ties and choose the tuple with the latest time stamp to favour time freshness. Then, remove all the candidate sets that contain the chosen tuple (line 5). The group utility of any tuple included in the removed candidate sets is decremented by the number of removed candidate sets containing the tuple (line 6). The algorithm then greedily picks the next tuple with the highest utility and the same hitting-set process continues until no candidate set is left to be hit. The chosen tuples constitute the solution.

Figure 6 shows how the region-based greedy algorithm is applied to the three group-aware filters *A*, *B* and *C* in the previous example. The upper part of the figure shows the candidate sets of the filters. The lower part shows the time-progressive computation of group utilities of the tuples, status of regions, and chosen outputs for the group.

At each time slot, each filter checks whether the newly arrived tuple is admissible. If so, the filter increments group utility of the tuple. At time slot 2, all three filters' candidate sets are closed, thus the region is closed to run greedy hitting-set algorithm. The output 0 is chosen for all filters. The next region closes at time slot 10, when all filters close their candidate sets. Tuple 100 is chosen as it is one of the tuples with the highest group utility. That is, cands1-3, cands2-3, cands3-2 are 'hit' by tuple 100. Next, tuple 50 is chosen, as it has the next highest group utility. Cands1-2 and cands2-2 are both hit by tuple 50. Now, all candidate sets have been hit in region 2. Thus, the outputs chosen at time slot 10 are tuple 100 for filter *A*, *B* and *C*, and tuple 50 for filter A and B.

Figure 6 Region-based greedy algorithm for three DC filters



time	1	2	3	4	5	6	7	8	9	10
tuple	0	35	29	45	50	59	80	97	100	112
group utility	3	0	0	2	2	2	1	3	3	0
region status		closed								closed
chosen outputs		0->{A, B, C}								100-> {A,B,C} 50-> {A,B}

3.3 Region's timely cuts

The region-based group-aware algorithm computes the smallest input stream segment to apply the hitting-set algorithm so that the optimality of the solution will not be affected. Beyond preserving bandwidth, we aim to ensure data timeliness as well.

Long candidate sets affect the timeliness of the output, because a region has to wait for all its member candidate sets to close before choosing outputs. In the case of a DC filter, after admitting a tuple in the candidate set, it waits for the first tuple that does not fall into the valid range for this candidate set to close the current candidate set. If the stream data changes little, and the filter's quality slack is relatively large, the candidate set can grow long, which affects the timely outputs of all its connected candidate sets.

Here we propose a mechanism, *cuts*, to curb the computation of long candidate sets according to filters' time constraints. We assume that each filter specifies a maximum tuple delay for group-aware filtering and we simply use the minimum of all the time specifications, we call it the *groupTime*, to enforce the data timeliness for the group. To derive the time cover of a region that satisfies *groupTime*, we build a latency model based on on-line observations of the most recent ten regions' performance, specifically the

correlation between region sizes and CPU time for computing the regions and choosing output for the regions. From our experiments, we found that a linear model was a pretty accurate fit. The last tuple in a region should not have a timestamp that exceeds (*groupTime* − *intercept*)/*slope*, where *intercept* and *slope* are coefficients of the current linear time model.

To enforce timely cuts to our previous algorithm, we extend it to check the time constraint after each filter finishes processing the new input tuple (after line 7 in Figure 4). Then, if the time constraints are about to be violated if we wait any longer, we force all open candidate sets to close. These closures will make the current region close automatically and then we can apply the GREEDY-HITTING-SET to choose the output, as before. Finally, after line 13 in Figure 4, we let the *globalState*, which keeps track of CPU time for computing a region; update the time model to compute a new group time constraint, which will be used in the next region.

It should be easy to see that cuts may reduce the sizes of candidate sets and thus reduce the likeliness of overlapping candidate sets, which may reduce the bandwidth-saving performance of group-aware filtering. Nevertheless, we can prove that the worst case for group-aware filtering algorithms with timely cuts is that each candidate set contains only one tuple and thus it is no different from self-interested filtering. Thus, although a timely cut may affect the bandwidth-saving performance of group-aware filtering, it performs no worse than self-interested filtering in terms of bandwidth consumption.

3.4 Stateful candidate sets

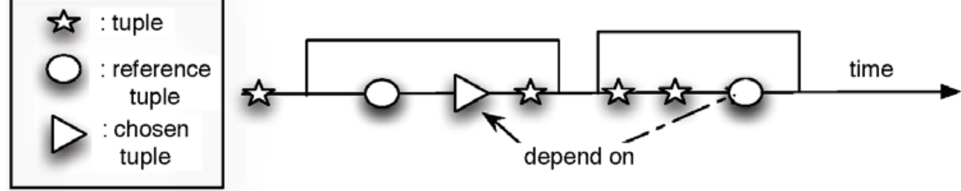
The above algorithms compute candidate sets based on reference tuples that are chosen by the self-interested filters with predicates. In other words, computing a filter's current candidate set does not depend on the chosen output of its previous candidate set. We call this *stateless computation of candidate sets* for a filter. For some applications, an alternative semantics for computing a candidate set is to base its reference on the chosen output of the previous candidate set. We call this *stateful computation of candidate sets*, and call the candidate sets *stateful candidate sets*.

For stateful candidate sets, the filter needs to choose the output as soon as its current candidate set closes, as the reference for the next candidate set depends on the chosen output (Figure 7). We use group state to track already-chosen tuples of each stateful candidate set in addition to the group utilities of tuples, and propose the following two heuristics for choosing output tuples from stateful candidate sets:

1 choose the tuple that has been chosen by other filters

2 choose the tuple that has the highest group utility.

Figure 7 Stateful candidate sets



The first heuristic takes precedence over the second heuristic. Both are subject to the tiebreaking rule, preferring the more recent tuple. After a filter chooses a tuple from a candidate set, the group utilities of all tuples in its candidate set are decremented by 1. Group state will keep track of the tuples chosen by each filter. If there are stateless filters in the group, identifying regions is still useful, as it is the earliest possible time to multicast decided tuples that have not yet been output in the region, when a region closes. In that case, we can still apply the greedy hitting-set algorithm upon the time the region is closed, only that the stateful filters' candidate sets become singleton sets with one chosen tuple in each. The logic for computing regions and timely cuts is the same as in the previous algorithm.

3.5 Output strategy

There are several output strategies we can use to enforce different output patterns. First, by computing regions, we get the *earliest possible* time for output tuples of a region without hurting the optimality of the solution. Second, by enforcing group time constraints, we get the *earliest possible subject to group time constraint* output pattern. Third, filters may opt for a *batched output* pattern, i.e., for a fixed-sized (time or tuple) batch of the input stream, select and output tuples.

In the case of a group with all stateful filters, it may be desirable to output tuples at the time each candidate set is closed, if the applications can tolerate disordered output within the predefined time frame. We call this *per-candidate set* output pattern. The benefit of using this pattern is that the delay of average tuple is less than that with a region-based earliest possible output strategy. The downside of it is that it may cause disorder in the output for the candidate sets in a region.

4 Evaluation

The goal of evaluation is to see how well group-aware filtering works in comparison to self-interested filtering, in terms of network bandwidth consumption and its effect on data timeliness.

4.1 Prototype system

We implement and integrate the group-aware filtering prototype with Solar (Chen et al., 2004), a general purpose data dissemination system developed at Dartmouth College. The core of Solar is a p2p overlay infrastructure in which each overlay node supports a suite of data-dissemination services, such as naming, data fusion and multicasting. We package group-aware filtering as a new service, working together with Solar's basic services on each overlay node.

Solar uses a content-based publish/subscribe model for flexible and scalable data dissemination. Publishers of context sources in Solar are called 'sources' and applications can 'subscribe' to sources in Solar to get the desired context information. Solar also allows an application to specify data operators, such as filters for pre-processing the source data.

For our testing, we replay real-world data traces as Solar sources and let a group of applications subscribe to the sources. Each subscribing application specifies a filter for its

processing needs. The group-aware filtering service then deploys, according to a filter's type and quality requirements, a group-aware filter object on the source node. The union of the output of all source-sharing filters is published via Solar's overlay multicasting service to the remote applications. To compute end-to-end latency based on time stamps, we deploy the subscribing applications on the same node as the data source to eliminate time skew in a network. Here we assume the real end-to-end latency is the time difference we will measure between a tuple published from the source and the time it arrives in an application, plus a constant number that captures overlay multicasting cost. In past deployments of Solar in a small (7-node) overlay network in Emulab[2], Solar's overlay multicasting delay was about 130 ms. This paper does not focus on the network aspects of group-aware filtering and we do not measure network behaviour while performing group-aware filtering. We thus measure the performance on the node where stream data were filtered. The source node was an Apple Powerbook with 1.67 GHz PowerPC G4 and 1 GB memory. Our code is written in Java and ran with Java 1.5.0 on Mac OS 10.4.9. Computing power of our testing nodes is reasonable for wireless mesh networks used for large-scale data dissemination in scenarios like emergency response.

4.2 Data sources

We chose data from real deployments of sensing devices for which the data stream has a sub-second data rate, so filtering is necessary and saving bandwidth for dissemination of the data is important. The networked aquatic microbial system (NAMOS) of the CENS project at UCLA[3] deployed embedded and networked sensors in Lake Fulmor for a marine scientific study during August 2006. The water was monitored by an array of thermistors and fluoro-meters, among others, installed on buoys of the lake. The data traces have data rates of 100 measurements per second and contain more than ten thousand measurements. These measurement traces make ideal data sources for our testing. Each NAMOS buoy trace tuple contains six temperature readings (we call them *tmpr* readings), one reading from a fluoro-meter (we call it the *fluoro* reading), a timestamp, and some other weather-related readings. We created a source in solar that replayed the NAMOS buoy trace at about 10 ms per tuple, observing the original time intervals of the trace data.

Table 1 Specifications for groups of DC filters (attribute, delta, slack)

Group name	*Filter*
DC_Fluoro	DC(fluoro, 0.0301, 0.0150)
	DC(fluoro, 0.0702, 0.0301)
	DC(fluoro, 0.0500, 0.0250)
DC_Hybrid	DC(fluoro, 0.0702, 0.0100)
	DC(tmpr2, 0.0460, 0.0153)
	DC(tmpr4, 0.0310, 0.0103)
DC_Tmpr	DC(tmpr4, 0.0310, 0.0155)
	DC(tmpr4, 0.0620, 0.0310)
	DC(tmpr4, 0.0480, 0.0240)

4.3 Filters for testing

The goal of the NAMOS buoy deployment was to help marine biologists to collect multi-scale high-resolution information, such as the spatial and temporal distribution of the chlorophyll level in the lake, for scientific analysis. Using DC filters or sampling filters is a valid way to enforce multi-scale granularity of the collected buoy data for these applications.

Due to limited space, we only show our experiments with DC filters (for experiments with other types of filters, see Lee (2008). Each DC filter has three parameters: the data attribute(s) that the filter is interested in, a delta value for compression and a corresponding quality slack it can tolerate. Table 1 shows the groups of filters we used for our testing. To set parameters for the *DC_Fluoro* and *DC_Tmpr* filter groups, we computed the average changes, *srcStatistics*, of two consecutive tuples in the source time series and then randomly picked delta values in the range of *srcStatistics* and 3**srcStatistics*, which ensured a reasonable data compression that had a non-trivial output data volume. Then we set slack values to be about 50% of the corresponding delta values. This approach prevented a tuple from being included in more than one candidate set for a filter, and also ensured large candidate sets for us to see the benefit of group-aware filtering. For the *DC_Hybrid* filter group, we randomly picked delta values from the range of *srcStatistics* and 20**srcStatistics* and randomly picked slack values that were less than 50% of corresponding delta values. Below, we also evaluate slack's effect on the performance of the DC filtering.

4.4 Metrics and results

4.4.1 Basic performance

The metric we use to measure the benefit of our group-aware filtering approach is the *O/I ratio*, that is, the output vs. input ratio defined as the total number of output tuples over the number of input tuples. A lower O/I ratio means low bandwidth consumption. It measures the bandwidth-saving benefit of group-aware filtering. We expect group-filtering should have an O/I ratio no more than that of self-interested filtering. We measured the filtering cost with *CPU time per tuple*, representing the CPU overhead of group-aware filtering. We also measured data timeliness with source-to-application *latency per tuple*, which shows the delay induced by group-aware filtering to each output tuple.

Table 2 Filter type notations

Abbreviation	*Meaning*
SI	Self-interested filter
RG	Region-based greedy filter
PS	Per-candidate-set greedy filter
+C	with timely cuts
+C(x)	with timely cuts, x is the name of a time spec.
(B)	with batched output strategy
(B)-x	with batched output strategy, x is input tuple window
(Pcs)	with per-candidate-set output strategy

Table 2 shows the notation we use for filters in the results. Figure 8 shows the O/I ratios for three groups of filters. All group-aware filtering algorithms consumed less than 80% of the bandwidth consumed by self-interested filters. PS filters had a performance comparable to RG filters, which in theory should have better performance guarantee. The addition of timely cuts had little impact on O/I ratio in this experiment, as we set the group time constraint big enough that few regions were cut.

Figure 8 O/I ratios for three groups of group-aware filters (see online version for colours)

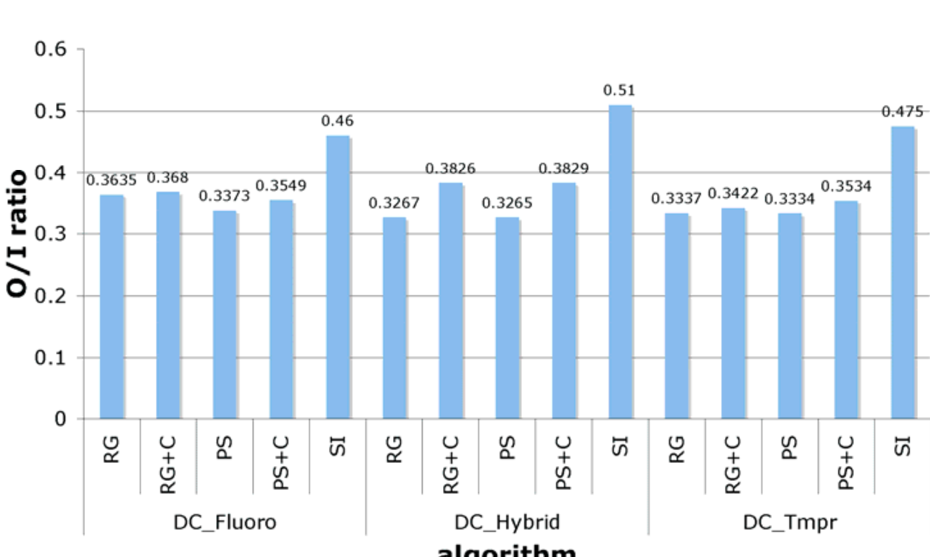


Figure 9 (a) CPU cost for DC_Fluoro (b) Latency for DC_Fluoro

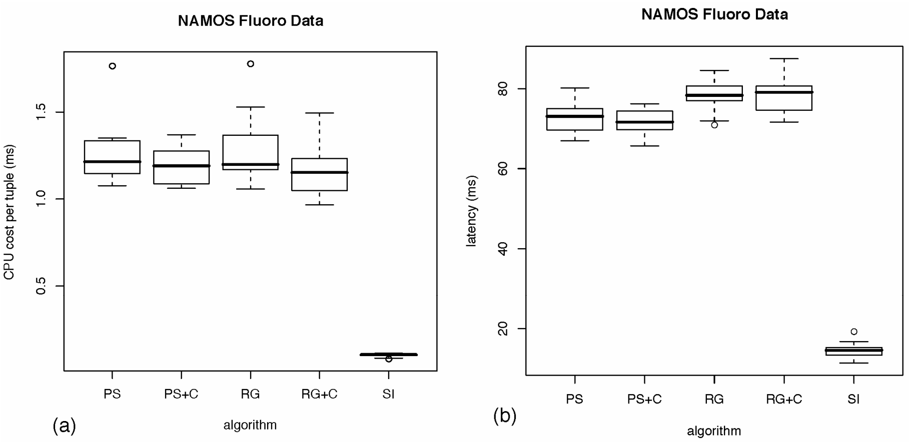


Figure 9(a) shows the CPU cost per tuple for the *DC_Fluoro* group. (The results are in a boxplot, which plots a summary of the minimum, 25% quartile, median, 75% quartile and maximum of the ten results. The circles represent outliers.) Group-aware filters were

more than 10 times more expensive than self-interested filters. This is understandable as the algorithms for group-aware filters are much complex than those of self-interested filters. However, it took only 1ms for processing each tuple for group-aware filters, which is fast enough for an input stream with a data rate of 100 tuples per second. Figure 9(b) shows the latency per tuple for the *DC_Fluoro* group. Since the group-aware filtering gathers tuples in a region before releasing output, it is understandable that the latency incurred for group-aware filters (about 70 ms per tuple) was much greater than that for self-interested DC filters (about 12 ms). The average region size of the filters was about 6 tuples; since tuples arrived at 10 ms intervals, it is clear that the 58 ms difference of latency was mainly due to waiting for the tuples to arrive for processing in segments. Due to limited space, we omit the CPU and latency results for the other two groups, but the conclusion was similar to that of *DC_Fluoro* group.

4.4.2 Performance of timely cuts

Next, we compare the performance of algorithms that enforce timely cuts. By decreasing the maximum time for closing a region from 125 ms in *RG+C*(01) filters, to a time 16-fold less in *RG+C*(05) (8 ms), the resulting average latency per tuple consistently dropped from above 70 ms/tuple to about 20 ms/tuple [see Figure 10(a)], thus proving that timely cuts were effective. Figure 10(b) shows that the CPU cost to enforce cuts, less than 1.5 ms, is acceptable for a fast stream with a 10 ms tuple interval. Cuts affected the O/I ratio only slightly (less than 3%), which is understandable because cutting a region will affect the optimality of the solutions found and it is a necessary trade-off for a latency-sensitive filter.

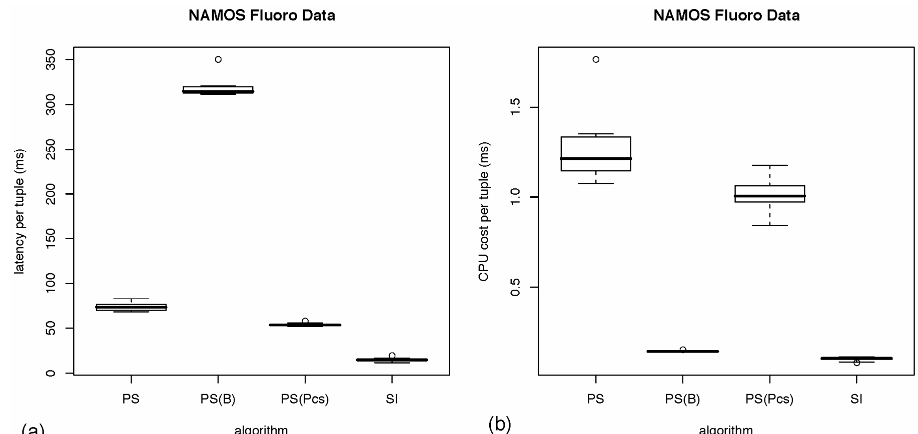
Figure 10 (a) Cuts affect latency for DC_Fluoro (b) CPU cost of cuts for DC_Fluoro

4.4.3 Evaluate output strategies

Finally, we evaluate the output strategies with *DC_Fluoro* filter group (Figure 11). The latency was affected mostly by the size of the average region a group-aware filter used before producing outputs. In the batched output pattern, when the batch size was much bigger than the size of a natural region, the latency increased dramatically due to

backlogging of the tuples in the filters until enough tuples were processed. The per-candidate-set output strategy helped to decrease the latency from above 70 ms to a little above 50 ms. In terms of CPU cost, the batched output pattern did not require sophisticated checking on whether a natural region is closed, which cut 1ms from the original 1.3 ms CPU time.

Figure 11 (a) Output strategy affects data timeliness (b) CPU cost of output strategies



4.4.4 Summary of results

To sum up, our prototype-based experiments validated the effectiveness of group-aware stream filtering in further saving of the bandwidth, compared with self-interested filtering. Worth noting, group-aware filtering is a tool to help data dissemination systems to explore further opportunities to save bandwidth consumption, but it does not guarantee the benefit, if the outputs of the filters intrinsically have no overlapping.

Its low CPU overhead indicates that group-aware stream filtering is suitable for fast stream processing (e.g., the incoming data rate in our experiment with DC filters was as fast as 10ms per tuple). The bottom line for using group-aware filtering is that the CPU overhead per tuple should not be consistently bigger than the arrival rate of the incoming data; that is, group-aware filtering should not cause congestion to the input buffers of the filters. Notice that the CPU cost is dependent on many factors, such as the complexity of the group-aware filters, the group size and the CPU power. It is thus hard to predict the performance of group-aware filtering well beforehand. It is advised to use online performance monitoring to dynamically decide whether or when to use group-aware filtering for long-running data-dissemination requirements. This leads to a direction of future work.

The increased delay in output tuples was due to the batch processing in group-aware filtering. Compared with the application-level multicast delay we measured, it is considered minor. Timely cuts were effective in curbing the latency in output tuples, yet its CPU overhead and its effect on O/I ratio were both small. Output strategies had the anticipated effect on output's timeliness, thus they provide knobs for the system to tune the performance of group-aware stream filtering for data timeliness.

We also evaluated some of the key factors that affect the performance of group-aware filters. Due to the space limit, we summarise the results in Figure 12. First, we decreased the slack values from 50% to 3% of the corresponding delta values, and the O/I ratios increased from 72% to 99%. Thus, slack values directly improved the data-reducing performance. Second, group size also showed positive effect on data reduction. This effect was due to the increased chance of overlapping among candidate sets. Finally, the characteristics of source data fundamentally determine the performances of the filtering. We summarise results for three other data sets: a cow's orientation changes (Schwager et al., 2007), seismic readings for the volcano (Werner-Allen et al., 2006) and the HRR(Q) reading changes in a fire experiment (Raghavan et al., 2007) respectively. The data sets' key values behave differently, and their plots are distinctive in the shape. The cow's orientation had brief changes over time. The seismic update pattern and HRR(Q) change pattern have relatively smoother curves. The difference in the features in the source data dictated the difference in their data-reduction performance.

Figure 12 Key factors that affect the performance of group-aware filters (see online version for colours)

Factor	Parameters	Results
Slack (DC)	3%Δ ~ 50%Δ	•O/I ratio: drop linearly (eg: 99% → 72%) •Little effect on CPU, latency
Group size	3 ~ 20 distinct specs	•O/I ratio: drop linearly (eg: 83% → 75%) •CPU: linearly increase (parallel to SI's)
Source data	Cow's orientation (MIT robotics)	•O/I reduction: 13.6% •CPU overhead: 33%
	Seismic (VolcanoNet)	•O/I reduction: 25% •CPU overhead: 21%
	HRR(Q) (Fire exps. WPI)	•O/I reduction: 36% •CPU overhead: 48%

5 Extensible framework

Here we discuss how group-aware filtering supports a variety of filters beyond simple DC filters. Filters in our consideration are selection-only operators that output a subset of tuples from an input source stream.

5.1 *Slack-based filters*

Slack-based filters concern a type of filters whose candidate sets are computed based on their tolerance to slacks in the quality requirements. The DC filters used in the previous

chapters are simple slack-based filters in that the quality slack is directly computed based on the raw values of an attribute in the time series. More generally, filters may apply functions to pre-process each tuple for computing candidate sets. For instance, if an application is interested in the changing rates or the 'trends' of temperature values, the filter may want to compute the ratio of the temperature change over a certain time span for each tuple. If the ratio is greater than some threshold, the application admits the tuple to the candidate set. Also, the functions may apply to more than one attributes of the data. For example, if a data stream consists of readings from multiple sensors of similar sensing capacities deployed in close vicinity, a filter may compute the 'average' readings over multiple attributes of the source data to see if it is above some threshold value. The distance functions used for computing the thresholds for admitting candidates can be more complex than numerical difference. For example, if a tuple contains two-dimension coordinates of a location, the Euclidean distance function will be used. Also, for classification-based candidate admission, domain-specific membership functions, such as fuzzy logic-based rules for 'safe' zones, may be used.

5.2 Sampling filters

For many exploratory data-analysis applications, *sampling* filters are of special interest. Sampling filters derive interesting properties by choosing a small set of data from a population. The notion of candidate sets is inherent in many commonly-used sampling methods, such as reservoir sampling, subset-sum sampling and stratified sampling (Johnson et. al, 2005; Thompson, 2002). For example, reservoir sampling chooses a fixed number of samples from a given population. Each tuple in the result can be replaced randomly by another tuple in the population. In this case, the candidate set of each output tuple is the whole data sequence in a predefined window. Reservoir sampling can be useful to bound the output bandwidth demands for some applications. For detection-oriented analysis, predicates that recognise interesting patterns can first be applied to the time series to distinguish important data sequences from less important ones, and then a higher sample rate can be applied to the more important data segments. This sampling theme belongs to *stratified sampling*, as it first decides strata of data with different characteristics and then samples each stratum with a different sample rate.

5.3 Rules for equivalence in quality

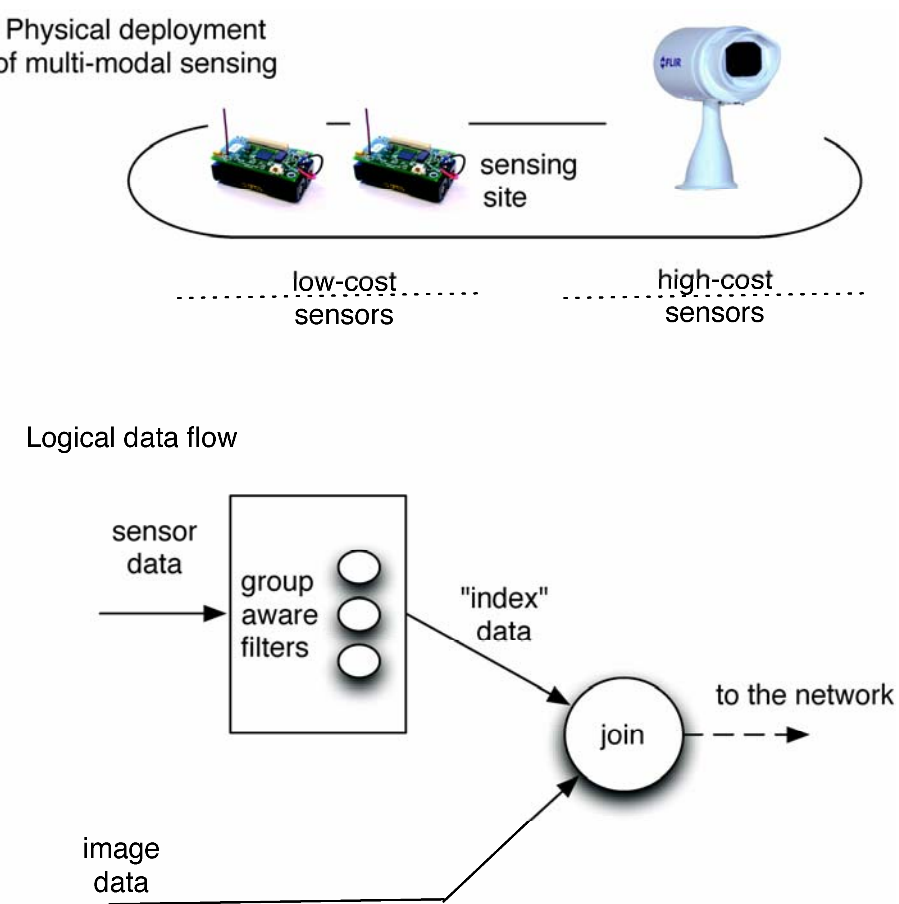
Some monitoring applications may employ sophisticated rules to define quality equivalence among tuples. For instance, given a data source that contains sensed information from many distributed sensing devices, the application may treat tuples equivalent in quality if they come from devices that are 'close' in physical and temporal distance and in sensing capacity. Reasoning on the closeness and similarity of the properties of the data is domain specific.

5.4 Framework extensions

We capture the diverse range of group-aware filters with a taxonomy (Li, 2008) of filters, based on how candidate sets should be computed, how outputs are selected and the dependency of candidate sets. Our framework provides many mechanisms for

applications to flexibly extend the two-stage process for group-aware filtering. First, applications can specify their filtering needs based on customisation of distance, membership and aggregate functions that system library provides, or based on self-defined domain-specific functions uploadable from uniform resource identifiers (URI) specified in the subscription file. We use object-oriented programming model to let applications extend the filtering procedures and interfaces flexibly. For example, we allow each filter to extend the basic group-aware filter's *isAdmisible* method to apply domain-specific functions in candidate admission. Second, our group-aware filtering service dynamically invokes pertinent group-aware filtering algorithms based on the applications' specifications. For instance, if the applications specify that the filter's candidate sets are dependent on one other, we invoke the per-candidate-set based algorithm, rather than the region-based algorithm. Finally, we expand the classic hitting-set problem and solutions for filters that need to select more than one output from each candidate set (Li, 2008).

Figure 13 Multi-modal sensing with group-aware filtering (see online version for colours)



5.5 Benefit for multi-modal sensing: an application scenario

The benefit of group-aware filtering may go beyond bandwidth savings. For example, Figure 13 depicts a scenario of a wireless surveillance system in which low-cost sensors, such as motion or seismic sensors, are bundled with a high-cost imager and deployed at the same site for remote monitoring. Such multi-modal sensing has shown an interesting balance between cost and functionality (Kulkarni et al., 2005; Ploetner and Trivedi, 2006).

The low-cost sensors can sample the targeting environment at a high rate to serve multiple surveillance applications. DC filtering and stratified sampling are both applicable for the surveillance application domain. Next, the output of the filters of the cheaper sensors is temporally correlated with the images taken by the high-resolution imagers to select the most informative images to send over the network to preserve network bandwidth. If we apply group-aware filtering to the filters, it will explore opportunities to reduce the output of the filters. The smaller the size of the output of the filters of the sensors, the smaller the number of the images selected to transport to remote applications. In this sense, we call the output from the filters of the low-cost sensors the 'index' for selecting data from the imagers. Imagine that we deploy such a sensor-imager bundle on a mobile robot for territory exploration; the indexing data may trigger cameras to take pictures. Thus it can potentially save the battery power needed for the robot to take the pictures and transmit them. It also saves space to store those images at the site in case of temporary network disconnection for delay-tolerant monitoring applications.

6 Related work

Our work exploits the semantics of a stream processing application to improve resource management in a distributed dissemination system. IBM's Gryphon (Strom et al., 1998) also leverages the semantics of subscribing applications to compress a sequence of data updates that have the same effect on applications' ultimate states. Zhao and Strom (2001) propose a special rule-based language to specify an application's sophisticated processing needs, specifically, the semantic equivalence of outputs to a remote application in face of retransmitted and disordered data. Rather than using a complicated language to describe the needs, our implementation provides a simple framework with customisable filters and functions to facilitate applications to describe the approximate nature of their filtering requirements.

Bandwidth-reduction mechanisms, such as sampling, summarising, and filtering, have been actively studied in recent years in the systems community (Babcock et al. 2002b; Chaudhuri et al., 1999; Mitchell, 1996). Most of the mechanisms are discussed in the context of a single streaming application. Only a few research efforts have looked into group optimisation for streaming applications, but these mechanisms are either based on traditional compiler rewriting techniques, or the simple grouping of stateless filters (Aryangat, 2004; Chen et al., 2000; Cheng et al., 2005; Madden et al., 2002; Olston et al., 2003). When data reduction is based on simple filters, grouping the filters for evaluation of common sub-expressions in the filters has been shown to save CPU time (Madden et al., 2002; Olston et al., 2003). We have different objectives for our filtering; the goal of our work is to trade computation time for savings in communication.

Johnson et al. (2005) summarised a general structure for sampling operators. The structure also contains candidate set admitting and output deciding stages, as we propose for the general group-aware filtering process. If we see the group-aware filtering from a sampling point of view, our algorithm is a special kind of sampler in that it picks an output from a candidate set of outputs 20 for each filter. But our process involves coordination across a group of applications, which never occurs in Johnson's single-application sampling.

7 Summary

This paper provides a general framework that gives a complete treatment to the group-aware filtering problem. We formally define the optimisation problem in group-aware filtering for continuous data streams, and prove its NP-hardness. We treat data quality management as the ultimate guidance to group-aware filtering: all our proposed heuristics-based algorithms for preserving bandwidth are subject to meeting the data granularity and timeliness requirements of the filters. We show that the group-aware filtering process is general enough to go beyond simple DC filters, and supports many sophisticated data filters such as stratified sampling. We demonstrate the effectiveness of our algorithms with an implemented system for disseminating real-world data sets. The encouraging results show that group-aware filtering is a quality-managed tool useful in exploring further opportunities to preserve bandwidth for data dissemination in low-bandwidth networks. More in-depth evaluations for group-aware stream filtering can be found in Li's (2008) thesis.

8 Acknowledgements

This research program is a part of the Institute for Security Technology Studies, supported under Award Number 2000-DT-CX-K001 from the US Department of Homeland Security, Science and Technology Directorate and by Grant Number 2005-DD-BX-1091 awarded by the Bureau of Justice Assistance. Points of view in this document are those of the authors and do not necessarily represent the official position of the US Department of Homeland Security or the US Department of Justice.

References

Akyildiz, I., Wang, X. and Wang, W. (2005) 'Wireless mesh networks: a survey', *Computer Networks*, pp.445–487.

Aryangat, S., Andrade, H. and Sussman, A. (2004) 'Time and space optimization for processing groups of multi-dimensional scientific queries', *Proceedings of the 18th Annual International Conference on Supercomputing (ICS)*, pp.95–105.

Babcock, B., Babu, S., Datar, M., Motwani, R. and Widom, J. (2002a) 'Models and issues in data stream systems', *Proceedings of the Twenty-First ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems (PODS)*, pp.1–16.

Babcock, B., Datar, M. and Motwani, R. (2002b) 'Sampling from a moving window over streaming data', *Proceedings of the Thirteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA)*, pp.633–634.

Chaudhuri, S., Motwani, R. and Narasayya, V. (1999) 'On random sampling over joins', *Proceedings of the 1999 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp.263–274.

Chen, G., Li, M. and Kotz, D. (2004) 'Design and implementation of a large-scale context fusion network', *Proceedings of the First Annual International Conference on Mobile and Ubiquitous Systems (MobiQuitous)*, pp.246–255, ACM Press.

Chen, J., DeWitt, D.J., Tian, F. and Wang, Y. (2000) 'Niagara CQ: a scalable continuous query system for internet databases'. *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp.379–390.

Cheng, R., Kao, B., Prabhakar, S., Kwan, A. and Tu, Y. (2005) 'Adaptive stream filters for entitybased queries with non-value tolerance', *Proceedings of the 31st International Conference on Very Large Data Bases (VLDB)*, pp.37–48.

Cormen, T., Leiserson, C., Rivest, R. and Stein, C. (2001) *Introduction to Algorithms*, 2nd ed., MIT Press.

Johnson, T., Muthukrishnan, S. and Rozenbaum, I. (2005) 'Sampling algorithms in a stream operator', *Proceedings of the 2005 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp.1–12, ACM Press.

Kulkarni, P., Ganesan, D., Shenoy, P. and Lu, Q. (2005) 'SensEye: a multi-tier camera sensor network', *Proceedings of the 13th Annual ACM International Conference on Multimedia (Multimedia)*, pp.229–238.

Li, M. (2008). 'Group-aware stream filtering', PhD thesis, Dartmouth College Computer Science, May 2008, available as Technical Report TR2008-621, Hanover, NH.

Madden, S., Shah, M., Hellerstein, J.M. and Raman.V. (2002) 'Continuously adaptive continuous queries over streams', *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp.49–60, ACM Press.

Mitchell, D.P. (1996) 'Consequences of stratified sampling in graphics', *Proceedings of the 23rd Annual Conference on Computer Graphics and Interactive Techniques (SIGGRAPH)*, pp.277–280, ACM Press.

Olston, C., Jiang, J. and Widom, J. (2003) 'Adaptive filters for continuous queries over distributed data streams'. *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data (SIGMOD)*, pp.563–574.

Ploetner, J. and Trivedi, M.M. (2006) 'A multimodal approach for dynamic event capture of vehicles and pedestrians', *Proceedings of the 4th ACM International Workshop on Video Surveillance and Sensor Networks (VSSN),* pp.203–210

Raghavan, V., Rundensteiner, E.A., Woycheese, J. and Mukherji, A. (2007) 'Firestream: sensor stream processing for monitoring fire spread', *Proceeding of the 23rd International Conference on Data Engineering (ICDE)*, pp.1507–1508, IEEE.

Schwager, M., Anderson, D.M., Butler, Z. and Rus, D. (2007) 'Robust classification of animal tracking data', *Computers and Electronics in Agriculture*, Vol. 56, No. 1, pp.46–59, March.

Strix Systems (2005) *Solving the Wireless Mesh Multi-Hop Dilemma*, available at http://www.strixsystems.com/products/datasheets/StrixWhitepaper_Multihop.pdf.

Strom, R, Banavar, G., Chandra, T., Kaplan, M., Miller, K., Mukherjee, B., Sturman, D. and Ward, M. (1998) 'Gryphon: an information flow based approach to message brokering', *International Symposium on Software Reliability Engineering (ISSRE)*

Thompson, S.K. (2002) *Sampling*, 2nd ed., John Wiley & Sons.

Werner-Allen, G., Lorincz, K., Welsh, M., Marcillo, O., Johnson, J., Ruiz, M. and Lees, J. (2006) 'Deploying a wireless sensor network on an active volcano', *IEEE Internet Computing*, Vol. 10, No. 2, pp.18–25.

Zhao, Y. and Strom, R. (2001) 'Exploiting event stream interpretation in publish-subscribe systems', *Proceedings of the 20th Annual ACM Symposium on Principles of Distributed Computing (PODC)*, pp.219–228.

Notes

1 Here we represent each tuple as a single integer; in reality, each tuple may have several fields, but for simplicity we represent each by the value of its 'temperature' field since it is that field that is used for filtering.

2 http://www.emulab.net is a cluster for distributed-systems research

3 http://cens.ucla.edu

Appendix A

NP-hardness proof for group-aware stream filtering problem

Here we prove that the group-aware stream filtering problem defined in Section 3.1 is NP-hard.

Theorem 1: Group-aware stream filtering is NP-hard.

Proof: We prove this property by reducing the problem to the minimum hitting-set problem, which is a classic NP-hard problem (Cormen et al., 2001).

Consider a special instance of the group-aware filtering problem in which each filter F_i has exactly one reference point and thus exactly one candidate set $cands_i$ for input stream *S*. Suppose we have n filters, so there are n candidate sets to choose output from. Since each candidate set is a subset of the tuples in *S*, this problem has a solution if and only if the minimum hitting-set problem with these n sets has a solution, that is, the output *O* of the minimum hitting-set problem makes sure that every of the n sets intersects with (or 'hits') *O* and *O*'s size is smallest among all solutions.

Appendix B

Definition and properties of region-based stream segmentation

The following definitions and properties lead to the definition of region. Later, we prove two important properties of region-based stream segmentation.

Definition 1: A time cover, TC_i, of a candidate set i is [$\min\{\forall t_j \mid t_j$ is time stamp of tuple j in candidate set $i\}$, $\max\{\forall t_j \mid t_j$ is time stamp of tuple j in candidate set $i\}$].

Axiom 1: Time covers of a filter's candidate sets do not intersect.

Axiom 2: Different regions' time covers do not intersect.

Proof: We prove it by contradiction. Suppose the time covers of two regions, *A* and *B*, intersect. It is easy to see that at least one candidate set, say $cands_0$ in *B*, is connected with a candidate set in *A*. Then adding a new candidate set $cands_0$ to A will still make A a region, which directly contradicts the assumption that A is a maximum collection of the connected candidate sets.

Definition 2: If A and B are candidate sets from two filters, and the time covers of A and B intersect, we say A and B are connected.

Definition 3: If A and B are connected candidate sets and B and C are connected candidate sets, we deem A and C to be connected.

Definition 4: A region is a maximum family of candidate sets such that each set is connected with every other in the family.

Definition 5: A time cover for a region is the union of all time covers of the candidate sets contained in the region.

Theorem 2: Given an input time sequence S, applying a divide-and-conquer approach for the group-aware filtering with region-based segmentation of S will not affect the optimality of the solution.

Proof: We need to prove that a set-union of the optimal solutions from each region on the input stream S is an optimal solution for S. We prove it by contradiction: that is, we suppose the opposite is true: given a total of n regions on S, and each region has an optimal solution O_i, the cardinality of the optimal solution O' of S is smaller than the size of the set-union U of all O_i, thus U is not an optimal solution for S. Now we divide O' into n distinctive subsets such that each subset is a group-aware filtering solution on each region, that is, each subset is a hitting set of a region.

To find such n subsets, we can first initialise n empty auxiliary sets, one for each region; then, for each tuple in O' that is contained by one of the candidate sets in a region, we put it in the auxiliary set of that region. We can see each tuple fall into exactly one auxiliary set; otherwise if a tuple belongs to two auxiliary sets, then there must be two candidate sets from two different regions containing the tuple, which means that the two candidate sets are connected and thus belong to the same region, which contradicts the assumption that they are from two different regions. In the end, we get n distinct subsets of O' in the auxiliary sets. We can prove that each auxiliary set is a hitting-set solution to its corresponding region. We prove it by contradiction. Suppose the opposite is true: that is, at least one candidate set in a region does not intersect with ('hit') the auxiliary set corresponding to the region. We know none of the other auxiliary sets hit this candidate set, otherwise there must be a candidate set in another region that intersects with this candidate set, which means that they are connected and are in the same region, which reaches a contradiction to our assumption.

As the size of O' is smaller than that of U, there must be at least one of the n subsets of O' whose size is smaller than that of the optimal solution O_j of that region, which contradicts the optimality of O_j for the region.

For heuristics-based algorithms, we prove that region-based segmentation preserves the approximation ratio of the sub-optimal solution.

Theorem 3: Region-based segmentation preserves the maximum approximation ratio of a heuristics-based algorithm. That is, given an input source segment S, which is segmented into n regions, if a heuristics-based algorithm has approximate ratio $r_1, r_2, ..., r_n$ on each region respectively, the approximation ratio r of the algorithm on the overall segment S satisfies the property that $r \le \max(r_1, r_2 ..., r_n)$.

Proof: Suppose $r_i = \max(r_1, r_2, ..., r_n)$, $1 \le i \le n$, that is, the approximate ratio of the algorithm in the ith region is the biggest among approximate ratios of all regions. According to the definition of approximation ratio, $r_j = \frac{O_j^{'}}{O_j}$, $1 \le j \le n$, where O_j is the size of the optimal solution on the jth region, and $O_j^{'}$ is the size of the sub-optimal solution produced by the heuristics-based algorithm on the jth region. According to the assumption that r_i is the bigger than the approximate ratio of any other region, we get $O_j^{'} \le \frac{O_j * O_i^{'}}{O_i}$, where $j \ne i$. The approximation ratio r on the overall data segment S satisfies

$$
\begin{aligned}
r &= \frac{O_1^{'} + O_2^{'} + ... + O_n^{'}}{O_1 + O_2 + ... + O_n} \\
&\le \frac{\frac{O_1 * O_1^{'}}{O_i} + \frac{O_2 * O_2^{'}}{O_i} + ... + \frac{O_n * O_i^{'}}{O_i}}{O_1 + O_2 + ... O_n} \\
&= \frac{\frac{(O_1 + O_2 + ... + O_n) * O_i^{'}}{O_i}}{O_1 + O_2 + ... + O_n} \\
&= \frac{O_i^{'}}{O_i} \\
&= r_i
\end{aligned}
$$