RESEARCH

Incorporating Change Detection in the Monitoring Phase of Adaptive Query Processing

Efthymia Tsamoura[1*], Anastasios Gounaris[2] and Yannis Manolopoulos[2]

Abstract

Recent Big Data research typically emphasizes on the need to address the challenges stemming from the volume, velocity, variety and veracity aspects. However, another cross-cutting property of Big Data is volatility. In database technology, volatility is addressed with the help of adaptive query processing (AQP), which has become the dominant paradigm for executing queries in dynamic and/or streaming environments. As the characteristics of the runtime environment may vary significantly along time, AQP techniques employ a three-phase adaptivity loop to process the input queries, comprising feedback collection, analysis and re-optimization. In the monitoring phase, the standard approach is to collect feedback in a fixed-size sliding window. However, several problems arise when the techniques adopt a fixed-size sliding window for maintaining runtime collected feedback. In this work, we tackle this limitation and we propose a novel monitoring phase, which assesses the collected feedback rendering an AQP technique capable of taking more informed decisions during the subsequent phases. The proposed approach is non-intrusive to the state-of-the-art adaptivity loop and can adopt any state-of-the-art online change detection algorithm through its plug-and-play abstraction. Another contribution of this work is a novel algorithm for detecting changes in a filter's drop probability, called β-CUSUM. The potential of the novel monitoring phase and of β-CUSUM is experimentally evaluated using both real-world and synthetic datasets.

Keywords: Adaptive query processing; Stream data processing; Stream change detection

Content

1 Main Text

Nowadays, an increasing number of Big Data applications deal with processing vast volumes of streaming data, e.g., network management and online processing of sensor data. Big Data applications typically emphasize on aspects, such as volume, velocity, variety, and veracity, which all require advances in the current state-of-the-art in data management [1, 2, 3]. Orthogonally to these aspects, there is significant volatility in data characteristics, such as bursty and unpredictable arrival rate, and evolving statistical properties. In this work, we focus on queries over data streams. Since the queries that are submitted to a data stream management system are usually long-running or even continuous, the runtime characteristics of the underlying execution environment may be significantly different from those when the query was initiated. This phenomenon may significantly deteriorate the performance of a query plan even in cases where that plan was considered optimal in the recent past.

To overcome this problem, a plethora of Adaptive Query Processing (AQP) techniques have been recently proposed in the literature aiming to adapt the runtime query plan in respond to changes in the execution environment or the characteristics of the streaming data [4, 5, 6, 7, 8]. The rationale followed by these AQP techniques can be condensed into a three-phase procedure, called adaptivity loop [9]. In the monitoring phase, the AQP techniques collect historic measurements from the underlying execution environment and keep them in a fixed-size (sliding) window, called profile window. Later, using the raw measurements or a synopsis of them, the AQP techniques derive estimates or compute statistics regarding the characteristics that majorly affect the performance of the query plan. In the analysis phase, they analyze the efficiency of the produced plan with respect to the derived statistics, and finally, in the reoptimization phase, they build a new query plan if the statistics indicate that the runtime plan is inefficient.

*Correspondence: efthymia.tsamoura@cs.ox.ac.uk
[1]University of Oxford, Parks Road, Oxford, UK
Full list of author information is available at the end of the article

Although the state-of-the-art AQP techniques have paid significant attention to efficiently and effectively analyze and reoptimize plans based on the statistics collected so far, they tend to overlook the backbone of the adaptivity loop, the monitoring phase. Thus, all of them suffer from the problems that arise when adopting a fixed-size sliding window.

In particular, in a small-sized profile window, the estimates or the statistics that are derived during the monitoring phase may vary significantly among different placements of the profile window, even in cases where the actual characteristics of the runtime environment do not change. As a result, the analysis and, subsequently, the reoptimization phases may be triggered quite often incurring overhead that outbalances the potential benefits of reoptimization.

In the opposite case where the profile window is large enough to keep out-of-date data (e.g., data that is not consistent with the characteristics of the runtime environment), the AQP technique may overlook changes in the characteristics that render the current plan suboptimal. The above are made clear through a real-world motivating example which deals with the filter ordering problem [6, 10] (see Section 2).

1.1 Contributions

In this work, we adopt the following rationale to face the limitations of a fixed-size sliding profile window discussed above. First, a characteristic that is relevant to an AQP technique (i.e., a characteristic that is monitored during the first phase of the adaptivity loop) is realized as a random variable or even as a collection of random variables. Second, an algorithm is employed to detect changes in the mean value of the distribution followed by that random variable (or the collection of random variables). Employing that rationale, each collected measurement is first added to the profile window and then used for change detection. If a mean value change is detected, then the out-of-date data that is kept in the profile window, i.e., the data that has been collected prior to change, is discarded. In this work, we focus on detecting changes in the filter selectivities and costs, although the above rationale can be adopted for other runtime characteristics, such as the network latency.

Through this idea, we can overcome the limitations inherent to a fixed-size sliding window: state-of-the-art change detection algorithms are sensitive enough to detect meaningful changes in the mean value of a characteristic and robust enough to overcome the problems faced by a small profile window. Moreover, the space overhead can be controlled by employing a change detection algorithm with a desirable space overhead.

The above rationale is incorporated in a novel monitoring phase which enhances the state-of-the-art one [9], while keeping the rest phases of the adaptivity loop unaltered. Moreover, neither the measurements/statistical estimates that are collected from the execution environment or the procedure that is employed for collecting them have to change when the novel monitoring phase is utilized. The novel monitoring phase can be employed as a "black box" by any AQP technique and can adopt any state-of-the-art algorithm for online change detection [15, 16, 17, 18, 19], through its plug-and-play abstraction. Furthermore, it provides the ability to perform other processing functions on the collected data (e.g., outlier detection). Finally, it works both when the characteristics of interest, e.g., filters' selectivity, are correlated or not. In the above, we discussed about changes occurred in the mean value of a characteristic. The same ideas can be also applied to other kinds of changes (e.g., a standard deviation change) given an appropriate change detection algorithm.

Another contribution of our work is the proposal of a novel algorithm, called β-CUSUM, for detecting changes in filter drop probabilities[1]. The execution of queries that consist of conjuncts of selective predicates is met quite often in modern network management and other types of applications, and their efficient execution heavily relies on updated filter drop probabilities.

There are several real-life examples, where our approach can be applied. A case study dealing with a network intrusion detection application is presented in Section 2. Another application is packet classification [20]. Packet classification is a router task during which multiple fields in the packet header are compared against rules (SQL-like queries) in a rule base. As these SQL queries are applied on a per packet basis and due to huge data volumes[2], the optimization of these queries is of primary importance. Given also the non-negligible space constraints [21], a drop probability change detection algorithm must have low computational and space overhead. β-CUSUM fulfills the above challenges having $O(1)$ both runtime and space complexity. Another field where our solutions can be applied is processing of environmental sensor data, where the filter drop probabilities depend on current weather conditions, such as wind speed and sunlight, which are continuously changing.

[1] A preliminary version of β-CUSUM has appeared in "E. Tsamoura, A. Gounaris and Y. Manolopoulos: Lifting the Burden of History in Adaptive Ordering of Pipelined Stream Filters, *Proceedings 7th IEEE International Conference on Data Engineering, Workshop on Self Managing Database Systems (ICDE-SMDB)*, 2012".

[2] Given packets of 40 bytes each at 40 Gbit/s speeds, a router has less than 10 nanoseconds to process each packet [21].

The proposed change detection algorithm is based on the well-known CUSUM algorithm [15], and operates on β-distributed data. We have chosen β distribution, since the latter was found to better model a filter's drop probability [22]. Experiments both on synthetic and real-world data prove its high accuracy in detecting drop probability changes. The latter along with its low run-time overhead compared with state-of-the-art change detection algorithms render β-CUSUM more suitable for general query processing purposes.

The novelty of the proposed monitoring phase is that it assesses the "quality" of the collected feedback. As a result, the decisions that are taken during the analysis and the reoptimization phases are shaped based on a high quality feedback. Another contribution of the proposed monitoring phase is that it provides the means to effectively control the tradeoff between the reoptimization frequency and the quality of the run-time plan. For example, a plan can be reoptimized only when the detected changes lead to performance deterioration more than a predefined threshold.

We experimentally study the benefits that accrue from the novel monitoring phase on the adaptive filter ordering problem [6], a problem of high interest due to its relevance to many query optimization problems and its applications in network management. Experiments were conducted both on real-world and synthetic datasets. Regarding the former experiments, we deal with an application of adaptive filter ordering in network intrusion detection [11].

The paper is organized as follows. In Section 2 we provide a real-world motivating example. In Section 3, we propose a plug-and-play abstraction for online change detection algorithms, while, in Section 4, we describe the novel monitoring phase that is based on the ideas discussed above. In Section 5, we present the β-CUSUM change detection algorithm, while in Section 6 we discuss about applying the novel monitoring phase on a state-of-the-art AQP technique, called A-greedy. The potential benefits of the novel monitoring phase are experimentally evaluated in Section 7. Finally, related work is discussed in Section 8 and the conclusions are drawn in Section 9.

2 Motivating example

The following example aims to provide insights into the problems stemming from fixed-size sliding window for collecting feedback measurements. Let the query

```
SELECT Packets P
FROM Network
WHERE P.PROTOCOL=TCP AND
byte_test(P,4,>32784,0,little)=TRUE AND
flow(P,to_server,established)=TRUE AND
P.DESTINATION_PORT=20031
```

This query selects the network packets that (i) follow the TCP protocol (P.PROTOCOL = TCP), (ii) are directed to the 20031 port of the receiver host (P.DESTINATION_PORT = 20031), (iii) are sent from a client to a server (flow(P, to_server, established)=TRUE) and, finally, (iv) their data payload has a specific content (byte_test(P, 4,>32784, 0,little) = TRUE). The query is taken from a real-world network monitoring application [11], which detects abnormal network behavior by running several queries over each captured network packet. The problem of finding the query plan that minimizes the total cost spent to process the input packets reduces to the *minimum cost filter ordering* problem as described in [10] and [6]. Assuming that the filters (selective predicates) have independent drop probabilities[3], i.e., the probability that a filter drops a tuple or not is independent of the outcome of the rest of the filters, and they have equal per-tuple processing costs, the optimal query plan is the one that orders the filters in descending drop probability order, thus acquiring accurate and updated drop probabilities is a key task. Moreover, as explained below, these probabilities

Figures 2(a) and 2(b) show how does the unconditional drop probability of each one of the above four predicates change in a time period of two weeks. The data packets of the specific example belong to the 1998 DARPA network intrusion detection dataset [12]. For presentation purposes, each figure shows the drop probability of two predicates. As shown in Figures 2(a)-(b), the filter drop probabilities change significantly during this two-week period. For example, the drop probability of the predicate P.DESTINATION_PORT=20031 is lower than the drop probabilities of the other predicates during the first week, while its drop probability reaches 1 (i.e., none of the network packets is directed to the 20031 of the receiver host) during Tuesday of week two.

From the above example, we can draw several interesting observations. As the filter drop probabilities show considerable changes during this two-weeks period (see Figure 2), we are interested in finding the ordering that has the lower cost. If we knew a-priori each filter's mean drop probability (considering the whole of this time period), we would be able to place the filters in descending mean drop probability order. However, this information is impossible to obtain as the network packets are dynamically created each timepoint. The only possible solution is to employ a sliding window approach to derive estimates of the filter mean drop probabilities. For example, we can track the filters that dropped each network packet and place this information in a sliding window.

[3] A filter's drop probability is defined as 1-filter's selectivity.

If we keep historic data relying on packets of the last two weeks, one week, or one day to collect statistics regarding the filter drop probabilities, then we build orderings with 30% approximately higher per-tuple processing cost (than the ones found when employing smaller size profile windows) due to out-of-date statistics. For example, when the profile window is large enough to hold statistics derived from packets that flowed during the first three days of the first week, then we converge to the ordering that first checks the P.PROTOCOL=TCP predicate even though its drop probability is significantly decreased the subsequent days. A solution towards producing orderings with lower cost is to adopt a smaller profile window, e.g., a window keeping statistics from packets captured during the last 10 minutes. By employing a smaller profile window the output filter orderings have on average lower per-tuple processing costs; however, the increased runtime overhead counterbalances the potential improvements. For the specific example, the runtime overhead increases 110 times approximately a phenomenon attributed to the high number of redundant re-optimizations that are performed.

Overall, the optimal profile window should have large enough size to smooth out transient fluctuations in the derived mean drop probability estimates that do not deteriorate the performance of the current ordering. At the same time, its size should be small enough in order not to contain out-of-date data which may hide changes rendering the current ordering suboptimal. We would be able to find the window size of the optimal profile only if (i) we knew the exact timepoints when the filter drop probabilities change and (ii) if we were able to arbitrarily increase the size of the profile window [13]. However, none of the above assumptions are realistic; the nature of modern streaming and/or distributed applications is totally unpredictable [14], and there exist usually constraints regarding the maximum length of the profile window.

3 A plug-and-play abstraction for online change detection algorithms

An initial aim of this work is to derive a generic description of online change detection algorithms, so that the proposed monitoring phase can interface with them in a technique-independent way. To create a plug-and-play abstraction we must first identify the major characteristics of the change detection algorithms. We derived these characteristics after considering several state-of-the-art online change detection algorithms (e.g., [15, 16, 17, 18, 19, 23, 24, 25]). The criteria of selecting them were their popularity and their diversity.

3.1 Characteristics of state-of-the-art change detection algorithms

The common characteristics of the state-of-the-art change detection algorithms could be summarized to the following: all of them take several assumptions regarding the distribution of the input data items. However, the "strictness" of these assumptions significantly differ among the change detection algorithms. For example, the newly introduced β-CUSUM operates on β distributed data items; the Martingale Test [19] realizes the input data items as a collection of exchangeable random variables. Second, all of the state-of-the-art change detection algorithms return the most probable changepoint, i.e., the timepoint where the change in the data characteristics is initiated [15]. The returned changepoint may be either the timepoint when the change is detected or an earlier point in time. Each algorithm employs different steps to estimate the most probable change point. For example, the Martingale Test returns the point in time when the change is detected as the most probable changepoint. On the other hand, ADWIN2 elaborates a more sophisticated technique (see Appendix B).

Regarding their differences, the state-of-the-art change detection algorithms are divided into two main categories (see Figure 3). In the first one we place the algorithms that require training in order to be operational. The algorithms of this category use feedback (a sample of input data items) to e.g., learn the current input data items' distribution (β-CUSUM, ChangeFinder [17]) or fill one or more baseline windows[4] (ADWIN2, Meta-algorithm). The algorithms of this category are called "feedback-full" throughout the rest of the paper. The algorithms that can perform change detection without requiring an input data item sample (e.g., Martingale Test) belong to the second category, the one of "feedback-less" algorithms. The overall aim of both types of algorithms is to use feedback in order to be capable of passing up-to-date and accurate metadata, such as filter drop probabilities, to the next phases in the adaptivity loop.

The class of "feedback-full" algorithms is further subdivided into two smaller categories: the "amnesic" and the "non-amnesic" one (see Figure 3). The algorithms of the former category must completely forget their runtime state as it has been modified so far after a change is detected. The state of a change detection algorithm is the runtime status of its data structures and variables and it is modified when either feedback is supplied to the algorithm or a data item for change detection. Those algorithms must transit to a state reflecting that no data (e.g., feedback) is ever presented

[4] A baseline line window is considered to store data items streamed prior a change has occurred.

to them. In contrast, the "non-amnesic" algorithms try to adjust their state, to reduce the impact of out-of-date data on it. Examples of "feedback-full" and "amnesic" algorithms are β-CUSUM, ChangeFinder and the Meta-algorithm. For example, ChangeFinder approximates the input data items' distribution through a mixture of normal distributions. When a change is detected, the previously learned data distribution model must be forgotten and the mixture of normal distributions must be relearned with newly incoming data items. On the other hand, ADWIN2 is a "non-amnesic" algorithm: ADWIN2 maintains a window where it places every input data item that is supplied while performing change detection. When a change is detected, ADWIN2 removes one or more items from this window starting from the ones appended earlier, but may not completely flush the content of this window. Of course, a "non-amnesic" algorithm can completely forget its runtime state and get new feedback after a change is detected. However, this is a less desirable action as additional delay is incurred to collect feedback and supply it to the algorithm.

The "feedback-full" algorithms also differ with respect to the way the feedback must be presented to them: the input data items must be supplied sequentially or in batch (see Figure 3). In the second case, the raw data that is collected is temporary buffered and is supplied to the change detection algorithm only after the buffer is full.

3.2 A Finite State Machine for change detection algorithms

In this section, we present the Finite State Machine (FSM) that models their operation. Based on the resulting FSM we will create the desirable plug-and-play abstraction. More specifically, the model states correspond to high-level functions, that are used as an interface between any change detection algorithm and our monitoring phase.

In Figure 4, we can see the FSM that corresponds to "feedback-full" and "amnesic" change detection algorithms. The first state is called *Tune* and corresponds to the procedure of initialising const-like parameters, i.e., parameters whose value does not alter during the execution of the algorithm. Examples of this kind of parameters is the number and size of the baseline windows that the Meta-algorithm employs [18], or the number of normal components that ChangeFinder employs to approximate the input data distribution.

The next state is called *Initialize*. There, the change detection algorithm adjusts its parameter values and data structures to reflect the case where no data items (e.g., feedback) have ever being presented. The value of the Boolean variable *hasFeedback* is also set at the *Initialize* state to false to express the lack of feedback.

After *Initialize*, the FSM transits to *Data source*. The FSM reaches this state every time a new data item arrives. From *Data source* the FSM can transit either to *Check*, where the algorithm can perform change detection, or to state *Collect feedback*, where the algorithm collects input data items for e.g., learning the input data's distribution or filling the maintained baseline windows. The transition depends on whether it is capable of performing change detection or not (*hasFeedback* is true or false, respectively).

In the *Collect feedback* state, the input data items are temporarily buffered to a data structure in cases where the "feedback-full" algorithm requires the data items to be supplied in batch; otherwise they are pipelined to the algorithm. Pipeline or batch mode is controlled by the value of the parameter $\theta_{feedback}$; a value greater than one corresponds to batching. At the *Use feedback* state, the algorithm uses the supplied data items. There, *hasFeedback* is set to true if the algorithm is ready to perform change detection. Independently of whether or not *hasFeedback* is set to true or false the FSM transits to state *Data source*; new data items must be presented to the algorithm either for collecting feedback or performing change detection.

From the *Check* state, the algorithm may or may not detect changes. When no change is detected, the algorithm gets the next data item to perform change detection (transition to *Data source* state). Otherwise, the algorithm reports the most probable changepoint (state *Get changepoint*) and then forgets entirely its runtime state (transition to *Initialize* state).

The FSM of "feedback-full" and "non-amnesic" change detection algorithms can be derived after transiting to a new state, called *Adjust feedback*, after *Get changepoint* (instead of transiting to *Initialize*) and adding a transition from *Adjust feedback* to *Data source*. In *Adjust feedback* state, the algorithm tries to eliminate the impact of out-of-date feedback on its state. Note that the algorithm, after performing the *Adjust feedback* procedure, may or not be able to perform change detection immediately. In the latter case, the *hasFeedback* value is set to false at that state. The above are explained through the following example. As described at the beginning of the section, ADWIN2, after detecting a change, it removes one or more items from its maintained window. If after the completion of this procedure the length of the maintained window is less than a threshold, then the algorithm must fill its window with newly arrived data items. To the best of our knowledge, the "feedback-full" and "non-amnesic" change detection algorithms sequentially utilize the data feedback. In cases where the feedback must be supplied in batch, then an extra data structure must buffer the input data stream items that have affected

```
1:  $pendingInitialize \leftarrow$ false; {/* Needed when employing "feedback-full" and
    "non-amnesic" algorithms */ }
2:  $detectedChange \leftarrow$ false;
3:  while A new (raw) measurement is taken from the execution environment do
4:    {Monitoring phase}
      { /* Steps performed when employing "feedback-full" and "non-amnesic" al-
      gorithms
5:    if $pendingInitialize$ is true then
6:      Initialize;
7:      $pendingInitialize \leftarrow$ false;
8:    end if*/ }
9:    Add measurement to the profile window;
10:   if $detectedChange$ is false then
11:     Preprocess the collected measurements;
12:     if hasFeedback is false then
13:       Collect feedback;
14:       if size of feedback = $\theta_{feedback}$ then
15:         Use feedback;
16:         Clear feedback; hasFeedback $\leftarrow$ true;
17:       end if
18:       goto Line 3; {/*Collect a new measurement from the execution environ-
          ment.*/}
19:     end if
20:     Check;
21:     if Change is detected then
22:       $detectedChange \leftarrow$ true;
23:       Get changepoint;
24:       Initialize; {The Adjust feedback function is called instead for "feedback-
          full" and "non-amnesic" algorithms.}
25:       Remove the out-of-date data from the profile window;
26:     end if
27:   end if
28:   {End of monitoring phase.}
29:   if $detectedChange$ is true $\wedge$ profile window has enough data items then
30:     $detectedChange \leftarrow$ false;
31:     Analysis phase;
32:     Reoptimization phase;
        {/* Steps performed when employing "feedback-full" and "non-amnesic"
        algorithms
33:     if The plan is reoptimized then
34:       $pendingInitialize \leftarrow$ true;
35:     end if*/ }
36:   end if
37: end while
```

Figure 1: AQP loop focusing on the novel monitoring phase details.

the algorithm's state up to the timepoint when the change was detected.

The FSM of "feedback-less" algorithms can be derived after removing the *Collect feedback* and *Use feedback* states from the FSM of Figure 4 (practically, they can be maintained as dummy states) and always setting at the *Initialize* state *hasFeedback* equal to true. The FSM must transit to *Initialize* state after a change is detected. This happens as, due to the best of our knowledge, no "feedback-less" algorithm is also "non-amnesic". In the opposite case, the FSM transits to *Adjust feedback* after a change is detected.

In Section 5 and Appendix B, we show how this plug-and-play abstraction is realized for the β-CUSUM, ADWIN2 and the Martingale Test algorithms.

4 The novel monitoring phase

Section 4 presents the novel monitoring phase. As stated in the introduction, it realizes the rationale of minimizing the impact of out-of-date data (e.g., CPU load measurements, filters' selectivity estimates) on the analysis and reoptimization phases of an AQP technique, while triggering re-optimization only when a meaningful change occurs. This is achieved through the incorporation of change detection into the AQP monitoring phase, which essentially modifies the effective size of the sliding window on-the-fly through discarding out-of-data data. The novel monitoring phase can be employed by AQP techniques that either maintain a (sliding) window of historic data or not.

In the following, it is assumed that the AQP technique periodically collects data regarding one characteristic of interest and buffers this data to a time-based or count-based (sliding) profile window. The size of this profile window must have a minimal length before the AQP technique applies its analysis phase. The AQP technique also associates each data item with a timestamp, which expresses the timepoint when a data item is retrieved from the execution environment. The generalization to multiple characteristics is straightforward and it is discussed at the end of this section.

We will first present a monitoring phase that is general enough to employ a change detection algorithm belonging to any of the categories presented in Section 3. Later, a refined monitoring phase is being developed for "non-amnesic" algorithms.

Figure 1 shows the generalized novel monitoring phase. Every time the AQP technique takes a (raw) measurement from the execution environment, this measurement is appended to the profile window (line 9). This measurement can be e.g., the CPU load taken at periodic time intervals or a bitmap which encodes which of the input filters rejected an input data tuple or not ([6]). Next, the collected measurements are being preprocessed (line 11). For example, we may remove outliers from the collected measurements. Next steps are similar to the ones presented in Figure 4 for "amnesic" change detection algorithms. When a change is detected, the *Initialize* function of a change detection algorithm is called and the latter entirely forgets its runtime state. After that, the AQP technique removes the out-of-date data values from the profile window (line 25). The values that will be removed are those with timestamp less than or equal to the returned changepoint.

The profile window may be empty after removing its out-of-date data. This happens when the returned changepoint is the current timepoint. As the profile window must contain a minimal number of data before invoking the analysis phase, the AQP loop may not be able to trigger the analysis phase. In this case, a procedure of collecting measurements from the runtime and adding them to the profile window begins (lines 3-9).

When the profile window has enough data (line 29), the AQP loop calls the analysis phase (line 31), which in turn, may call the reoptimization phase if it finds the current query plan suboptimal (line 32). When the analysis (and reoptimization) phases are finished the AQP loop transits to the monitoring phase and the flow proceeds as described in previous paragraphs.

Although the above described monitoring phase is general enough, it does not fully take advantage of the property of "non-amnesic" algorithms of not entirely forgetting their state – this property may reduce or nullify the time needed to collect and use feedback. For example, ADWIN2, after removing input data items from its window in response to a change in the runtime, its internal window may have enough items to immediately perform change detection.

The monitoring phase that targets the "non-amnesic" change detection algorithms has a few extensions/dissimilarities to the one presented above. The additional steps are enclosed into brackets. First, the *Adjust feedback* function is called after a change is detected, instead of *Initialize* (line 24). Second, if the query plan is reoptimized, the monitoring phase calls the *Initialize* function of the change detection algorithm, so as the change detection algorithm to entirely forget its runtime state (lines 5-8). Recall that after the *Initialize* state, a "feedback-full" algorithm must collect feedback prior to be operational.

This happens for the following reason: if the AQP technique estimates conditional statistics, then the state of the change detection algorithm prior to reoptimization is irrelevant is no longer useful. The above is explained through an example borrowed from the pipelined correlated filter ordering problem [6] (more details are given in Section 6). Suppose that we have three filters F_1, F_2, F_3 with correlated selectivity, i.e., the selectivity of one filter depends on the filters that precede it in the current ordering, and that the characteristic of interest is the conditional selectivity of the filter that is currently placed in the second position. Also suppose that ADWIN2 is employed for detecting changes in the filters' selectivity. If the filters are reordered from e.g., $F_1F_2F_3$ to $F_2F_3F_1$, then the state of ADWIN2 is related to the partial ordering F_1F_2 which is irrelevant to F_2F_3.

If the AQP technique does not estimate conditional statistics, then, the monitoring phase does not have to call the *Initialize* function of the change detection algorithm as the collected statistics may still be useful. Continuing with the previous example, if the filters do not have correlated selectivity, then the characteristics of interest are the filters' unconditional selectivity. As a result, the state of the ADWIN2 algorithm has been modified by only a specific filter's selectivity (the one whose selectivity is responsible to check for changes), which, in turn, is not affected by the current query plan but only by the characteristics of the input data items. Thus, the state of ADWIN2 does not have to be forgotten after the filters are reordered.

The above ideas can be straightforwardly applied when the AQP technique monitors two or more characteristics regarding the runtime execution environment. In that case, each one of the characteristics can be checked for changes employing a state-of-the-art change detection algorithm. The analysis phase is triggered when at least one change is detected. However, there is a point that must be disambiguated. Suppose that for each one of the characteristics of interest there exist a separate profile window that stores the corresponding data values collected during the monitoring phase of the AQP technique. If during change detection we find changes only for a subset of those characteristics, we can flush only the out-of-date contents of the corresponding profile windows. The analysis phase is then triggered as soon as the profile windows that were updated have enough data items.

5 The β-CUSUM algorithm

The main motivation behind the work in this section is that there is evidence that a filter drop probability better fits a β distribution [22]. To date, no change detection algorithm exists that is tailored to this type of probability distribution, and our proposal fills this gap. Our algorithm for detecting changes in a filter's drop probability is called β-CUSUM. It is a CUSUM-like algorithm [15], which works on β-distributed data motivated by the finding that. At the end of the section, we also discuss about how β-CUSUM can be plugged into the abstraction presented in Section 3.

CUSUM assumes that we know the probability distributions of the data prior and after a change has occurred. Let these probability distributions be P_0 and P_1, respectively, and $\hat{d}_j$ be an input data item. The algorithm relies on the observation that, upon receiving a new input data item (e.g., a drop probability estimate), if the current distribution is P_0, then the probability that $\hat{d}_j$ is produced under P_0 is higher than $P_1(\hat{d}_j)$. As a result, the log-likelihood ratio $\ln(P_1(\hat{d}_j)/P_0(\hat{d}_j))$ shows a negative drift before change, and a positive drift after the change. Based on the above rationale, every time a new item $\hat{d}_j$ arrives, the CUSUM algorithm updates a cumulative sum S_j as follows:

$$S_j = \left\{ \begin{array}{ll} S_{j-1} + \ln \frac{P_1(\hat{d}_j)}{P_0(\hat{d}_j)}, & S_{j-1} + \ln \frac{P_1(\hat{d}_j)}{P_0(\hat{d}_j)} > 0 \\ 0, & otherwise, \end{array} \right\} \tag{1}$$

where $S_0 = 0$. If a probability distribution changes from P_0 to P_1, then the log-likelihood ratios estimates that are derived as new data items arrive are positive – and thus the cumulative sum S_j continuously increases. CUSUM assumes that, if the sum of the log-likelihood ratios computed so far exceeds a certain threshold $h > 0$, then a change in the underlying data distribution is detected and P_1 becomes the new base probability distribution. Otherwise, the above procedure continues by cumulating the newly computed log-likelihood ratios. CUSUM is rather effective in detecting changes [15] but it requires the availability of the probability distributions P_0, P_1, which makes it inapplicable to online scenarios. An online variant of the original CUSUM is proposed in [26] that is based on the assumption that the P_0 and P_1 distributions are normal.

In our case, we have strong evidence that drop probabilities better fit a β distribution [22]. Thus, we develop an online version of CUSUM that assumes β distributions. In β-CUSUM, a training phase is first adopted to derive the parameters of the base (i.e., prior to change) β distribution. After that, the original test in Eq. (1) is employed.

Training phase: The training phase requires a set $\mathcal{D}$ of drop probability estimates. Given $\mathcal{D}$ and a confidence level of ζ we derive the single-value estimates $\alpha_{|\mathcal{D}|}$ and $\beta_{|\mathcal{D}|}$ and the associated confidence intervals $[\alpha^{lo}\ \alpha^{up}]$ and $[\beta^{lo}\ \beta^{up}]$ of the parameters α and β of the base β distribution. After finishing the training phase, the cumulative sum in Eq.(1) is set to 0. For presentation purposes we provide the details of the training phase in Appendix A.

Change detection phase: Every time a drop probability estimate $\hat{d}_j$ is supplied to β-CUSUM, the mean value μ_j and the standard deviation σ_j of the $j > |\mathcal{D}|$ drop probability estimates seen so far (including those of the training set) are incrementally computed. Then the α_j and β_j parameters of the β distribution are given by $(\mu_j(1-\mu_j)/\sigma_j - 1)\,\mu_j$ and $(\mu_i(1-\mu_j)/\sigma_j - 1)\,(1-\mu_j)$, respectively. If α_j or β_j do not lie within $[\alpha^{lo}\ \alpha^{up}]$ and $[\beta^{lo}\ \beta^{up}]$, then it is assumed that the item $\hat{d}_j$ is produced by a different β distribution $Beta(\alpha_j, \beta_j)$ and S_j is updated following Eq.(1), where $P_0 = Beta(\alpha_{|\mathcal{D}|}, \beta_{|\mathcal{D}|})$ and $P_1 = Beta(\alpha_j, \beta_j)$. If S_j exceeds the threshold h, then a change is detected and a changepoint is reported. For the β-CUSUM algorithm to be operational again, a training phase must be applied with a new training set, since the previously found confidence intervals of the β distribution are estimated using past data items.

Both the runtime computational and space complexity of β-CUSUM are $O(1)$; each time a new drop probability estimate is fed to the algorithm, we incrementally compute the mean value and the standard deviation of the estimates seen so far. The computational cost incurred during the training phase is also considered to be constant as the training set is relatively small ($|\mathcal{D}| < 500$ in practice) and the Nelder-Mead algorithm converges very fast in practice.

β-CUSUM is categorized as a "feedback-full" and "amnesic" change detection algorithm: it requires a sample training set D to learn the parameters of the prior to change β distribution and a new β distribution must be learned as soon as a change is detected. Being a "feedback-full" and "amnesic" algorithm we must implement the following functions. In *Tune*, we assign values for the h and ζ parameters, as well as, we choose the size of the sample set $\mathcal{D}$. In *Initialize*, we set to 0 the single value estimates and the confidence intervals associated with the α and β parameters of the base distribution. There, *hasFeedback* is also set to false. The sample set $\mathcal{D}$ must be supplied to the algorithm in batch and thus, $\theta_{feedback}$ is set to $|\mathcal{D}|$. The *Use feedback* and *Check* functions realize the training and the change detection phases of the algorithm, respectively. After *Use feedback* finishes, *hasFeedback* is set to true. Finally, *Get changepoint* returns the current time point when change is detected as neither CUSUM nor β-CUSUM can determine the most probable changepoint. We can optionally assume that the change is initiated at the $\nu - th, \nu > 0$ most recent drop probability estimate that has been supplied. The impact of the user specified parameters h and ν is evaluated in the Section 7.

6 Case study: The A-greedy AQP technique

This section deals with an example of applying the novel monitoring phase to a state-of-the-art AQP technique, called A-greedy, for the problem of commutative and correlated adaptive filter ordering [6][5].

Let $\mathcal{F} = \{F_1, F_2, \ldots, F_N\}$ be a set of N input filters. The input filters are commutative and associated with (i) processing costs and (ii) drop probabilities, which are correlated, i.e., a filter's drop probability depends on the filters upstream in the ordering. A single data source also streams tuples which are pipelined among the filters, i.e., tuples already processed by a filter may be processed by a subsequent filter in the pipeline, at the same time as the sender filter processes new data tuples. The goal is to find a filter ordering, such that the total processing cost of the input tuples by the filters is minimized. Since the execution environment is dynamic, e.g., either the characterizes of the input data

[5] A-greedy has also been used as an Eddies routing policy [29].

tuples or the per-tuple filter processing costs may vary along time, a three-phase AQP technique is proposed in [6].

The statistics that A-greedy collects during its monitoring phase are a set of $(N \times N)/2$ conditional drop probability estimates and the per-tuple processing costs of the input filters. Without loss of generality, let $\mathcal{O} = F_1F_2 \dots F_N$ be the current filter ordering. A-greedy collects estimates for $d(i|i-1)$, $1 \leq i \leq N$, where $d(i|i-1)$ is the conditional probability that F_i will drop a tuple from the input data stream, given that this tuple was not dropped by any of the filters that precede F_i in $\mathcal{O}$. For example, $d(1|0)$ is the unconditional probability that the first filter in the ordering (F_1) will drop a tuple. $d(2|1)$, in turn, is the conditional probability that the second filter in the ordering (F_2) will drop a tuple that was not dropped by the first one, and so on.

To estimate the desirable conditional drop probabilities, A-greedy maintains a fixed-size sliding profile window. Each row in the profile window keeps metadata extracted from a randomly chosen tuple t of the input data stream. The metadata that is being extracted from tuple t is a N-length bitmap; if the i-th input filter drops t, then the i-th entry in that bitmap is true and is false otherwise. Every time new metadata is added to the profile window A-greedy re-estimates the $(N \times N)/2$ conditional drop probabilities $d(i|i-1)$. The per-tuple processing costs of the input filters are estimated using a simple averaging technique. For each tuple that is randomly chosen to extract metadata from (regarding which of the input filters have dropped it or not), A-greedy also measures the time each filter spends to process that tuple. Then, the per-tuple processing cost of an input filter is the running average of the processing time measurements collected so far.

Every time the profile window is updated with new metadata, the analysis phase is triggered. There, A-greedy checks if the current ordering satisfies or not the greedy invariant [6]. If the greedy invariant is not satisfied, then the reoptimization phase is triggered and the input filters are reordered.

The modifications that are applied on A-greedy to adopt the novel monitoring phase are quite straightforward: each one of the $(N \times N)/2$ conditional drop probabilities $d(i|i-1)$, $1 \leq i \leq N$, is checked for changes utilizing either a state-of-the-art or the β-CUSUM algorithm (see Section 5). The drop probability values $\hat{d}(i|i-1)$, $1 \leq i \leq N$, that will be supplied for change detection are estimated as follows: the profile window is virtually divided into non-overlapping blocks of size k. For each new block, we derive the estimates $\hat{d}(i|i-1)$ taking into account only the block's contents. When at least one of the $(N \times N)/2$ drop probabilities changes, then the most recent changepoint is returned (invoking the *Get changepoint* function) and the metadata with timestamp prior to the reported changepoint is removed from the profile window. A-greedy adopts N profile windows to collect cost measurements and, subsequently, to derive cost estimates for the input filters (Section 4.1.2 of [6]). We can employ a similar rationale to detect changes in the filter processing costs. In this case, the incurred space and time overhead will depend on the characteristics of the adopted change detection algorithm.

7 Evaluation

The purpose of the Evaluation Section is two-fold. First, we study the advantages of employing the novel monitoring phase during AQP. We show the potential of the novel monitoring phase through the A-greedy technique (see Section 6) after performing a series of experiments both with real-world and synthetic datasets. Second, we qualify the robustness of the novel monitoring phase with respect to the algorithm that is employed to perform change detection in the mean value of a characteristic of interest (filter drop probability in our case). Four state-of-the-art algorithms (ADWIN2 [16], Martingale Test [19], Meta-algorithm [18], ChangeFinder [17]), as well as, the newly introduced β-CUSUM were considered to this end.

The key observations are summarized below: (i) When A-greedy employs the novel monitoring phase, the mean overall cost improvement reaches the 18% and 36% in real-world and synthetic datasets, respectively. (ii) The mean overall cost improvement (when the novel monitoring phase is employed by A-greedy) increases as the cost incurred during each reoptimization invocation increases, as well. On the other hand, it decreases as the per-tuple processing cost increases. (iii) The performance improves when any of the above change detection algorithms is employed. The maximum (mean) performance improvement is met when β-CUSUM algorithm is employed for drop probability change detection both in real-world and synthetic datasets. The experiments were conducted on an i5 Linux machine with 4GB memory.

7.1 Experiments on a real-world application

For our experiments, we took 111 queries of 4, 5, or 6 predicates each (similar to the one presented in the example in Section 2) that are executed on a per-packet basis for network intrusion detection. The input data that is supplied to each query is 6M network packets from weeks one and five from the 1998 DARPA Network Intrusion Detection dataset [12]. These queries were derived from the Snort query base[6] employing the following procedure:

[6]We consider the 2.9.2.1 query base snapshot.

- We removed the queries consisting of filters that either require more than one network packets to be checked or consider general characteristics from the packet flow, such as the total number of remote login failures. These filters are presented below: *stream_reassemble, same_ip, asn1, dce_iface, dce_opnum, dce_stub_data, sip_method, sip_stat_code, sip_header, sip_body, gtp_type, gtp_info, gtp_version, ssl_version, ssl_state, detection_filter, threshold.*
  We removed these filters for the following reason: in A-greedy the filter drop probabilities may be correlated. However, each filter drops or retains an input tuple after considering exclusively the contents of this tuple. The filters stream_reassemble, same_ip, etc., were dropped as they violate this assumption.
- We also removed the filters that follow constraints regarding the order that can be checked. Such filters are, for example, the ones that search for a specific pattern inside a packet's payload and the position where the search begins is controlled by previous pattern matches. Filters of this category are (i) *pcre, byte_test* and *isdataat* having the relative content modifier and (ii) *content* that relates to *within* or *distance* modifiers. For similar reasons, we also removed the queries that contain the filters *base_64_decode, byte_jump* and *byte_extract.* We removed these filters as A-greedy assumes that there are no constraints in the ordering of the filters, i.e., it assumes that filters can be positioned anywhere in the ordering.
- Finally, we removed from each of the remaining queries the filters that are either never satisfied or are satisfied at least two orders of magnitude less times from the rest of the filters in the same query when the 6M network packets dataset is supplied.

To find out if the filter drop probabilities are correlated or not, we created, for each query, all the possible filter orderings. Then, for each filter, we computed its maximum drop probability difference that is observed when its position in the ordering changes. We have observed that, for the majority of filters, the difference lies into [0.2,0.4], while for some filters the difference reaches the 0.95. For each query, we have seen that at least one conditional drop probability changes approximately 4.500 times, while 2.500 out of these changes lead to violations of the greedy invariant by 50% on the average. Regarding the evaluation time of each filter, it is in the order of 10 micro seconds.

The profile window of A-greedy (when it employs its current adaptivity loop) is realized as a time-based sliding window that maintains statistics computed from network packets captured during either of the following time periods: the last 2 weeks, the last 1 week, the last day, the last 12 hours, the last 6 hours, the last 1 hour and the last 10 minutes. When A-greedy employs the novel monitoring phase, we check for changes only the conditional filter drop probabilities, employing the simple averaging procedure to derive estimates of the per-packet processing costs. The conditional drop probabilities are checked each time employing a different state-of-the-art algorithm, while each algorithm is employed after considering a different parameter assignment. This is done, to assess the robustness of a change detection algorithm. For example, the possible parameter assignment combinations for the employed change detection algorithms are shown in Table 4 in C. For clarity, throughout the rest of this section, we use A-greedy* to refer to A-greedy when it employs the novel monitoring phase and Agreedy*/{Change detection algorithm}, e.g., A-greedy*/β-CUSUM, to further disambiguate the algorithm that is employed for filter drop probability change detection.

Table 1 shows both for A-greedy and Agreedy* (when either of the ADWIN2, Martingale Test, Meta-algorithm, ChangeFinder and β-CUSUM is employed for filter drop probability change detection) the mean overall cost considering all of the 111 queries described above. The overall cost is the total cost spent to process the input packets (second column) plus the runtime overhead of the adaptivity loop (third column)[7]. Regarding A-greedy (as the technique has been employed after considering seven profile windows of different sizes and, thus, for each query we got seven different solutions), we considered for each query the overall cost of each possible solution. For the same reason, we considered for each query the overall cost of each solution for A-greedy*/β-CUSUM, A-greedy*/ADWIN2, A-greedy*/ Martingale Test, A-greedy*/ Meta-algorithm and A-greedy*/ChangeFinder.

We can see that A-greedy* spends (on average) less time to process the packets of the dataset and to perform the phases of its adaptivity loop. In particular, the maximum overall cost improvement is met when A-greedy* employs β-CUSUM (18%). The minimum improvement, in turn, is met when the Meta-algorithm is employed by A-greedy* (12.4%). Both the overall packet processing cost and the runtime overhead improve. In particular, the cost spent by A-greedy to process the input packets is 81.64 sec, while the runtime

[7]We did not consider the cost spent during the monitoring phase to create the metadata for estimating the conditional filter drop probabilities, as it is the same both for A-greedy and A-greedy*.

overhead of its adaptivity loop is 0.9 sec. The corresponding cost values for A-greedy*/β-CUSUM are 67.41 sec and 0.05 sec respectively. The low runtime overhead of A-greedy*/β-CUSUM is due to the low runtime overhead of β-CUSUM for drop probability change detection and (ii) the low number of redundant reoptimization invocations (approximately one order of magnitude less than the number of query reoptimizations that A-greedy performs).

While conducting the above experiments we reached to the (somewhat expected) conclusion; the runtime overhead decreases inversely proportional to the overall cost spent to process the input data. However, A-greedy has the worst tradeoff between these two constants. Table 1 also shows the mean overall cost spent when A-greedy and A-greedy* build the filter orderings with the minimum and the maximum overall packet processing costs, respectively. To derive these results, we considered for each query the solutions having the lowest and the highest overall packet processing costs among the total solution set (e.g., for A-greedy we get seven different solutions for each query, one for a different profile window), respectively. A similar procedure is followed regarding A-greedy*. In the first case, the mean runtime overhead of A-greedy climbs to 8.77 sec (0.08 sec for A-greedy*/β-CUSUM), while in the second case, although the mean runtime overhead is 0.18 sec (0.04 sec for A-greedy*/β-CUSUM), the mean overall packet processing cost is 86.64 sec (70.45 sec for A-greedy*/β-CUSUM). The increased mean runtime overhead of A-greedy (that incurs when building the minimum packet processing cost plans) stems from the increased number of redundant reoptimization invocations. That above further support our intuition that although the data processing cost decreases when employing a small-sized profile window during the monitoring phase of an AQP technique, the reoptimization overhead outbalances the potential benefits of the lower cost plans.

In the middle and the lower part of Table 1 the maximum overall cost improvement is met when A-greedy* employs the β-CUSUM algorithm in both cases and it is 15.1% and 18.8%, respectively. On the other, the minimum overall cost improvement is met when employing the Meta-algorithm for drop probability change detection (12.8% and 9.5%, respectively).

Another observation that can be drawn from Table 1 is that A-greedy* is robust with respect to the runtime overhead and packet processing cost as the deviations among these constants when building the minimum and the maximum packet processing plans are lower than that of A-greedy. In particular, A-greedy*/β-CUSUM is the most robust solution since the packet processing cost and the runtime overhead deviations are 3% and 45%, respectively. The corresponding values for A-greedy are 17% and 97%, respectively.

7.2 Experiments on synthetic datasets

In the second part of the Evaluation Section, we present experiments performed with synthetic datasets. As in Section 7.1, we employ the novel monitoring phase on the A-greedy technique assuming correlated filters. We produced correlated input tuples adopting the steps described in [6]. Each query may consist of 4, 64 or 256 filters, respectively, while the per-filter processing costs lie in $\{10 \text{ micro seconds}, 10 \text{ seconds}\}$ and are constant. For a given query, at least one filter drop probability changes every 50K, 200K or 1000K tuples, while a query's conditional drop probability may change up to ten times in total. The occurred changes may be abrupt or gradual. In the latter case, there exists a transition period until a conditional drop probability reaches its new value. The length of this period (in tuples) may be 10K, 25K or 50K tuples. The model under which a conditional drop probability changes during a transition period is linear. Similar to Section 7.1, A-greedy is employed with profile window sizes of different lengths (106652, 34473, 6345, 3000, 1500, 250, or 41 tuples, respectively) and A-greedy* with different assignments of the parameters of the utilized change detection algorithms (see Table 4 in C). As the per-tuple processing costs are constant, A-greedy* checks for changes only the conditional filter drop probabilities.

Experiment 1. The goal of Experiment 1 is to study the behavior of the novel monitoring phase when a characteristic (filter drop probability in our case) changes abruptly. We considered 20 different queries of 4 filters having 10 micro seconds per-tuple processing cost each. Abrupt changes take place every 1000K tuples. Table 2 shows the % mean overall cost improvement over A-greedy. Similarly to the upper part of Table 1, the mean overall cost is computed taking into account the whole set of solutions for each query both for A-greedy and A-greedy*. Recall that for each query multiple solutions are produced, one for each possible assignment of the parameters of a change detection algorithm (when A-greedy* is employed) or for each profile window length (when A-greedy is employed). This is done to find out how robust a change detection algorithm is with respect to its parameter assignments. We can see that A-greedy* improves the overall cost by $\simeq$14% when employing β-CUSUM, by $\simeq$11% when employing ADWIN2, Martingale Test, and ChangeFinder algorithms, while the overall cost improvement when A-greedy* employs the Meta-algorithm is approximately 8.6%. To find out how does the performance of A-greedy* change with respect to the frequency of the occurred changes, we repeat the above experiment changing the filter drop probabilities abruptly every 200K and 50K tuples respectively. The results are also shown in Table 2. We can see that the overall cost improvement does not change significantly.

Experiment 2. In the second experiment, we study the impact of the number of filters on the performance of the novel monitoring phase. For this reason, we repeat the Experiment 1 with 64 filters this time. The results are shown in Table 2. We can see that the maximum (mean) overall cost improvement reaches $\simeq$36% when A-greedy* employs β-CUSUM or ADWIN2, while it is slightly lower for the rest of the algorithms. Experiment 2 was also performed considering 256 input filters and the (mean) overall cost improvement was about 80%. The results are however omitted due to lack of space. The outcome of Experiment 2 was somewhat expected; the runtime overhead of A-greedy is now higher due to the higher overhead of each query reoptimization (as the number of input filters increases, the overhead of each filter reordering increases, as well). The above, along with the fact that when the novel monitoring phase is employed, on average, one order of magnitude less query reoptimizations are performed, reason about the outcome of Experiment 2.

Experiment 3. Experiment 3 aims to investigate the correlation between the per-tuple processing cost and the performance of A-greedy*. Experiment 1 is repeated considering, however, filters having 10 sec per-tuple processing cost (see Table 2). We can see that the mean overall cost improvement of A-greedy* is somewhat lower, while when A-greedy* employs the Meta-algorithm we have performance loss, i.e., the mean overall cost is higher than that of A-greedy by 1%. The reason behind that phenomenon is the following: as the per-tuple filter processing cost increases, the data processing cost dominates over the runtime overhead. Thus, it is more beneficial to perform many query reoptimizations to keep the data processing cost as low as possible.

Experiment 4. The goal of the fourth experiment is to explore the potential benefits of the novel monitoring phase when the monitored characteristics change gradually. Similar to Experiment 1, we considered 20 different queries of 4 filters having 10 micro seconds per-tuple processing cost each. Gradual changes take place every 1000K tuples and the transition period lasts 10K tuples each time a change occurs. The % overall cost improvement results over A-greedy are shown in Table 3. The maximum and the minimum cost improvement is 23.90 % (under β-CUSUM) and 13.54 % (under the Meta-algorithm), respectively. We can see that under gradual changes the benefits of the proposed monitoring phase become clearer. The reason for which we observe a 9% higher overall cost improvement comparing with the case where the changes are abrupt (see the upper part of Table 3, column 2) is due to the fact that now A-greedy becomes more insensitive to the occurred changes. As a result it performs reoptimizations less often and, subsequently, it builds less efficient filter orderings. For completeness, we repeat the above experiment after changing the transition length to 25K and 50K tuples, respectively. The results are also shown in Table 3. The whole Experiment 4 is repeated after changing the frequency of the gradual changes from 1000K tuples to 200K tuples (see **Experiment 5** in Table 3). The conclusions that are drawn are similar to the ones of Experiment 4.

The final two experiments aim to study the behavior of the novel monitoring phase when the occurred changes are gradual and either the number of filters (**Experiment 6**) or the per-tuple filter processing cost (**Experiment 7**) changes. To this end we repeat Experiments 2 and 3, respectively, considering, however, gradual changes to occur every 1000K tuples. We study the cases where the transition period lasts 10K, 25K or 50K tuples. The maximum obtained performance improvement reach the 36.23% and the 21.10%, respectively, and in both cases appears when the β-CUSUM algorithm is employed (see Table 3).

The conclusions that are drawn from the above experiments are the following. First, A-greedy* has better performance over A-greedy in general. Second, β-CUSUM leads, in the majority of cases, to the maximum performance improvement, a fact that proves its robustness with respect to the type of the occurred change and the possible assignments of its parameters. The latter, along its $O(1)$ space complexity, render β-CUSUM a good solution for adaptive query processing.

8 Related Work

Our work relates to many scientific fields. In the following, we present the corresponding fields and stress out how our work differentiates with respect to state-of-the-art work. State-of-the-art AQP techniques (e.g., [6, 7, 8]) employ a three-phase adaptivity loop to produce plans that are consistent with the characteristics of the input data streams or the runtime environment. However, existing work does not assess the quality (e.g., freshness) of the feedback that is collected during their monitoring phase with the consequences described in Section 1. For example, in Eddies the selectivity and the cost of each operator is estimated considering the whole of the data streamed so far [29]. Similarly, the work in [7] estimates the load of each remote host at a specific timepoint utilizing a set of

CPU, memory and bandwidth measurements that has been collected since the submission of a query. The Borealis distributed stream processor employs a fixed-size sliding window technique to derive estimates of each distributed operator's load during a specific timeperiod [8].

Several techniques have been also proposed in the literature that aim to deal with the problems that the inaccurate statistics cause during database query optimization (e.g., [30, 31, 32]). For example, to derive a subquery's cardinality prior to query execution, a query optimizer must take several assumptions regarding the distribution of the base data (e.g., data uniformity assumption) and the query predicates (e.g., predicate independence assumption). However, these assumptions may cause significant cardinality estimation errors "deluding" a query optimizer towards suboptimal plans. The work that has been developed to overcome the above limitations cannot be applied when the data sources are streaming; it merely deals with the problems that arise when taking wrong assumptions to estimate a joint or conditional selectivity having database stored data.

Our work relates to the area of robust query optimization (e.g., [22, 33, 34]) too. However, state-of-the-art work on robust query optimization does not deal with streaming data sources or dynamic execution environments; the probability distributions or the confidence intervals that are adopted to approach a characteristic of interest are known a-priori and are not exploited during a runtime measurement collection procedure.

The problem of data stream sampling over a sliding window is also relevant to our work [35, 36]. Existing techniques, however, focus on maintaining either an unbiased sample of the data stream (a fixed size sample which has the same probability of being selected with any other sample of the same size) or a sample that preserves the characteristics of the data stream neglecting a potential stream evolution. The work in [37] comprises an exception, where stream evolution is considered during sampling through exponentially biasing the samples. This approach, however, assumes that the stream evolves under an a-priori known exponential model, which generally does not hold in modern streaming applications.

Our work could be considered as an alternative of ADWIN2, as ADWIN2 maintains a list of input data items which dynamically shrinks or enlarges depending on whether a change is detected or not. However, it is a generalization of ADWIN2. First, it can perform ADWIN2's functionality employing any online state-of-the-art change detection algorithm, controlling thus the runtime complexity. Another benefit of having the ability to employ any change detection algorithm is that we can deal with different kinds of changes (e.g., standard deviation changes), while ADWIN2 deals only with mean value changes. Third, ADWIN2 must store the entire set of the data items streamed so far until a change occurs. In contrast, our idea has the ability to work on limited space overhead.

9 Conclusions

The purpose of this work is two-fold. First, we address the problems that arise when adopting a fixed-size sliding window during the monitoring phase of an AQP technique. To the best of our knowledge, no work in the literature addresses that issue. Second, we propose a novel algorithm for detecting changes in a filter's drop probability. Experiments both on a real-world application and on synthetic datasets show the potential of our work. A possible direction for future work is to employ the proposed monitoring phase for the robust adaptive query optimization problem. An interesting topic might be also to employ a similar rationale to adjust the sampling rate that is used to collect measurements during the monitoring phase based on the characteristics of the runtime environment or the input data stream.

Appendix A

Training of β-CUSUM is done in two steps and requires a set $\mathcal{D}$ of drop probability estimates. Given $\mathcal{D}$ and a confidence level of ζ we derive the single-value estimates $\alpha_{|\mathcal{D}|}$ and $\beta_{|\mathcal{D}|}$ and the associated confidence intervals $[\alpha^{lo}\ \alpha^{up}]$ and $[\beta^{lo}\ \beta^{up}]$ of the parameters α and β of the base β distribution.

In the first step, we compute the single value estimates $\hat{\alpha}$ and $\hat{\beta}$ using a maximum-likelihood estimation (MLE) method ([27]). We compute the confidence intervals using the estimates derived during the previous step and the Fisher information matrix built from the likelihood function of the unknown α and β parameters. For a beta distribution in (0 1), Gnanadesikan et al. [27] have shown that the log-likelihood function of α, β given the sample set $\mathcal{D}$ is given by:

$$L(\alpha,\beta) = (\alpha-1)\ln(G_1)+(\beta-1)\ln(G_2)-\ln(Beta(\alpha,\beta)), \tag{2}$$

where G_1 and G_2 are given by

$$G_1 = \prod_{i=1}^{|\mathcal{D}|} d_i(.|.)^{1/|\mathcal{D}|} \tag{3}$$

$$G_2 = \prod_{i=1}^{|\mathcal{D}|} (1-d_i(.|.))^{1/|\mathcal{D}|} \tag{4}$$

and $Beta$ is the beta function [13]. The $\hat{\alpha}$ and $\hat{\beta}$ estimates can then be found by minimizing Eq.(2). This can be done using a multidimensional, unconstrained optimization algorithm, such as the Nelder-Mead [28].

In the second step, we build the 2×2 Fisher information matrix $\mathbf{I}(\alpha, \beta)$ of the log-likelihood function $L(\alpha, \beta)$ (see Eq.(2)). The expression $E[.]$ denotes the conditional expectation over L given α and β. The fisher information is a way of measuring the amount of information that an observable random variable, i.e., a random variable for which we have observations, carries about an unknown parameter upon which the probability of the random variable depends. Substituting in Eq.(5) the ML estimates $\hat{\alpha}$ and $\hat{\beta}$ and then inverting the matrix, we can approximate the covariance matrix of the $\hat{\alpha}$ and $\hat{\beta}$ estimates[8], i.e.,

$$\mathbf{I}^{-1}(\hat{\alpha}, \hat{\beta}) \simeq \begin{pmatrix} \widehat{\sigma^2_{\hat{\alpha}}} & \widehat{Cov(\hat{\alpha}, \hat{\beta})} \\ \widehat{Cov(\hat{\alpha}, \hat{\beta})} & \widehat{\sigma^2_{\hat{\beta}}} \end{pmatrix}, \qquad (6)$$

where $\widehat{\sigma^2_{\hat{\alpha}}}$ and $\widehat{\sigma^2_{\hat{\beta}}}$ are the variances of the $\hat{\alpha}$ and $\hat{\beta}$ estimates, respectively, while $\widehat{Cov(\hat{\alpha}, \hat{\beta})}$ is the covariance of $\hat{\alpha}$ and $\hat{\beta}$.

Utilizing the central limit theorem we can prove that the ML estimates $\hat{\alpha}$ and $\hat{\beta}$ follow asymptotically a normal distribution, i.e., for a large sample size $|\mathcal{D}|$, the distribution of $\hat{\alpha}$ and $\hat{\beta}$ would be very close to the normal distribution. The above provide us a way for estimating the confidence intervals for the estimates $\hat{\alpha}$ and $\hat{\beta}$. Thus, the $(1-\zeta)\%, \zeta \in (0,1)$, confidence intervals for the estimates $\hat{\alpha}$ and $\hat{\beta}$ are $\hat{\alpha} \pm z_{\zeta/2}\widehat{\sigma^2_{\hat{\alpha}}}$ and $\hat{\beta} \pm z_{\zeta/2}\widehat{\sigma^2_{\hat{\beta}}}$, respectively, where $z_{\zeta/2}$ is the upper $100\zeta/2$ percentage point of the standard normal distribution.

Appendix B

Appendix B presents how to realize the plug-and-play abstraction for the Martingale Test and ADWIN2 change detection algorithms.

The Martingale Test assumes that the input data stream items are produced from Exchangeable random variables and a change occurs when the variables' exchangeability property is violated [19]. The steps of the algorithm are given below:

[8]Since the estimates $\hat{\alpha}$ and $\hat{\beta}$ are computed using a sample set $\mathcal{D}$, each estimate may have a different value when computed using different samples from the set $\mathcal{D}$. The distribution of an estimate may be considered as the distribution of the values that it can take when computed using every possible sample of $\mathcal{D}$.

- *Check*: When a new data stream item $\hat{d}_i$ arrives, it is appended to list $\mathcal{D}$ and its "strangeness" s_i, with respect to the rest of the items in $\mathcal{D}$, is computed. To detect filter drop probability changes, we have used the strangeness measure $s_i = median_{\forall \hat{d}_j \in \mathcal{D}}\{|\hat{d}_i - \hat{d}_j|\}$. Next, the randomized power martingale M_i^{ϵ} is computed, with respect to a user-defined parameter $\epsilon \in [0,1]$. If its value is greater than a user-defined threshold $h > 0$, then a change is detected, otherwise the algorithm proceeds as described above.
- *Initialize*: When a change is detected, the data stream item list $\mathcal{D}$ is cleared and the randomized power martingale M_i^{ϵ} is set to 1.

The Martingale Test is a "feedback-less" and "amnesic" change detection algorithm. In *Tune*, the values of the parameters h and ϵ must be decided, while *Check* and *Initialize* are realized as described above. As the algorithm does not return a changepoint, we can adopt the approach followed by β-CUSUM.

The rationale of ADWIN2 is quite simple. The whole set of the items streamed so far is stored in a list $\mathcal{L}$. Every time a new data stream item is present, it is appended to $\mathcal{L}$. If $\mathcal{L}$ has enough data items it is partitioned into multiple pairs of non-overlapping sublists. If the estimated means of any sublist pair differ significantly, then it is assumed that the two sublists comprise data following distributions with different mean values and the data stream items belonging to the "oldest" sublist (the one holding the data items that appeared earlier) are removed from $\mathcal{L}$.

To reduce the runtime and space complexity, $\mathcal{L}$ is realized as a timeless variation of Exponential Histograms (EH) ([38]). The structure adopted by [16] is defined by a parameter $L > 0$ and consists of a list of buckets. Each bucket holds the total sum of the items that are stored in that bucket and the arrival time of the oldest element stored in that bucket. Employing the rationale described at the beginning of the section, when a data stream item $\hat{d}_i$ arrives:

- *Check*: It is added to the bucket list $\mathcal{L}$. Then the mean value difference of every sublist pair $(\mathcal{L}_0, \mathcal{L}_1)$ is computed with respect to a user-defined parameter $\delta > 0$. Note that the sublists are delimited by the borders of the buckets. If that difference is significant then a change is detected.
- *Adjust feedback*: When a change is detected, one or more buckets, starting from the oldest ones (i.e., the earliest created buckets), are dropped until there is not any sublist pair having mean values that differ significantly.
- *Get changepoint*: The returned changepoint is the timepoint of the oldest $\hat{d}_i$ item that resides on the oldest remaining bucket.

$$\mathbf{I}(\alpha,\beta) = \begin{pmatrix} E\big[\big(\frac{\partial}{\partial\alpha}\ln L(\alpha,\beta)\big)^2\big] & E\big[\big(\frac{\partial}{\partial\alpha}\ln L(\alpha,\beta)\big)\big(\frac{\partial}{\partial\beta}\ln L(\alpha,\beta)\big)\big] \\ E\big[\big(\frac{\partial}{\partial\alpha}\ln L(\alpha,\beta)\big)\big(\frac{\partial}{\partial\beta}\ln L(\alpha,\beta)\big)\big] & E\big[\big(\frac{\partial}{\partial\beta}\ln L(\alpha,\beta)\big)^2\big] \end{pmatrix} \tag{5}$$

As discussed in Section 3, ADWIN2 is a "feedback-full" and "non-amnesic" change detection algorithm. The values of parameters δ and L are set in *Tune*. In *Initialize*, the bucket list $\mathcal{L}$ is set to $\emptyset$, while *Check*, *Adjust feedback* and *Get changepoint* are realized as described above.

Appendix C

Appendix C presents details regarding the implementation of the change detection algorithms used to conduct the experiments of Section 7. The possible values that were checked when employing the algorithms are presented in Table 4. Regarding ChangeFinder, we employed a logarithmic loss scoring function both for outlier and change detection (see [17]). Similarly, we employed a mixture of three normal distributions to approximate the input data both during the outlier and the change detection phases of the algorithm. The possible assignments of the T averaging parameter are shown in Table 4. Regarding the Meta-algorithm [18], we employed the Kolmogorov-Smirnov statistic over intervals and tested different assignments for the parameter pair (n, p)[9](see Table 4). The different baseline windows that were employed by the Meta-algorithm had lengths 1000, 2400, 5000, 10000, 24900, 49900, 100000, 249900 or 499900 data items.

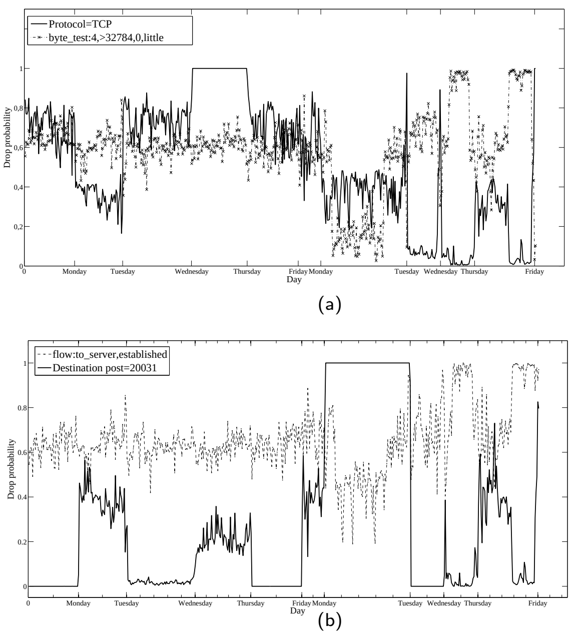

Figure 2: The drop probabilities of the example in Section 2.

Competing interests
The authors declare that they have no competing interests.

Author's contributions
E. Tsamoura conducted the main part of this research as part of her Ph.D. thesis. A. Gounaris closely collaborated with E. Tsamoura during all phases of the work described hereby. Y. Manolopoulos supervised the whole research activity.

Acknowledgements
None.

Figures

Tables

Author details
[1]University of Oxford, Parks Road, Oxford, UK. [2]Aristotle University of Thessaloniki, Thessaloniki, Greece.

[9]The latter pair has the following meaning: for a data stream, where no change occurs, the probability of detecting a change after the first n data stream points is at most p.

References
1. Abadi, D., Agrawal, R., Ailamaki, A., Balazinska, M., Bernstein, P.A., Carey, M.J., Chaudhuri, S., Dean, J., Doan, A., Franklin, M.J., Gehrke, J., Haas, L.M., Halevy, A.Y., Hellerstein, J.M., Ioannidis, Y.E., Jagadish, H.V., Kossmann, D., Madden, S., Mehrotra, S., Milo, T., Naughton, J.F., Ramakrishnan, R., Markl, V., Olston, C., Ooi, B.C., Re, C., Suciu, D., Stonebraker, M., Walter, T., Widom, J.: The beckman report on database research. ACM SIGMOD Record **43**(3), 61–70 (2014)
2. Armbrust, M., Xin, R.S., Lian, C., Huai, Y., Liu, D., Bradley, J.K., Meng, X., Kaftan, T., Franklin, M.J., Ghodsi, A., Zaharia, M.: Spark SQL: relational data processing in spark. In: Proceedings ACM International Conference on Management of Data (SIGMOD), pp. 1383–1394 (2015)
3. Singh, D., Reddy, C.K.: A survey on platforms for big data analytics. Journal of Big Data **2**(1), 1–20 (2015)
4. Gedik, B.: Partitioning functions for stateful data parallelism in stream processing. The VLDB Journal **23**(4), 517–539 (2014)
5. Elseidy, M., Elguindy, A., Vitorovic, A., Koch, C.: Scalable and adaptive online joins. Proceedings of the VLDB **7**(6), 441–452 (2014)
6. Babu, S., Motwani, R., Munagala, K., Nishizawa, I., Widom, J.: Adaptive ordering of pipelined stream filters. In: Proceedings ACM International Conference on Management of Data (SIGMOD), pp. 407–418 (2004)
7. Gu, X., Yu, P.S., Wang, H.: Adaptive load diffusion for multiway windowed stream joins. In: Proceedings IEEE International Conference on Data Engineering (ICDE), pp. 146–155 (2007)

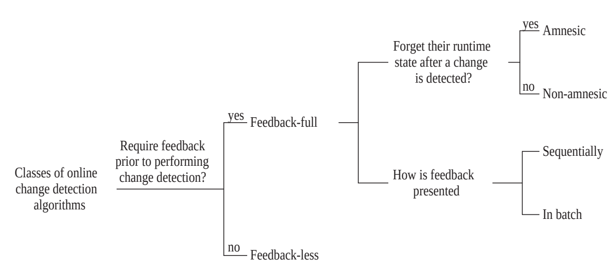
Classes of online change detection algorithms
Require feedback prior to performing change detection?
yes
Feedback-full
no
Feedback-less
Forget their runtime state after a change is detected?
yes
Amnesic
no
Non-amnesic
How is feedback presented
Sequentially
In batch

Figure 3: A taxonomy of state-of-the-art online change detection algorithms.

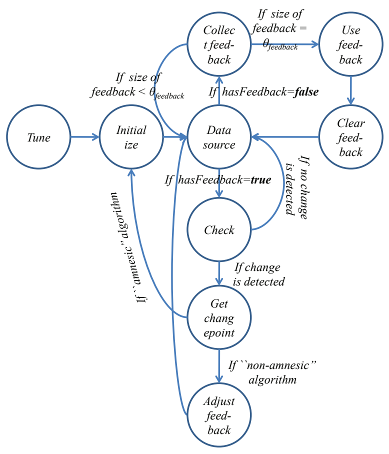

Figure 4: FSM for online change detection algorithms.

Table 1: Mean overall cost for A-greedy and A-greedy*.

For each query we considered all of the possible solutions		
	Packet processing cost (seconds)	Runtime overhead (seconds)
A-greedy*/ADWIN2	69.65	0.66
A-greedy*/Martingale Test	69.44	0.45
A-greedy*/ChangeFinder	69.59	0.25
A-greedy*/Meta-algorithm	72.08	0.20
A-greedy*/β-CUSUM	**67.41**	**0.04**
A-greedy	81.64	0.91
For each query we considered the minimum packet processing cost solution		
	Packet processing cost (seconds)	Runtime overhead (seconds)
A-greedy*/ADWIN2	69.40	0.72
A-greedy*/Martingale Test	69.37	0.48
A-greedy*/ChangeFinder	69.48	0.26
A-greedy*/Meta-algorithm	70.10	0.26
A-greedy*/β-CUSUM	**68.26**	**0.07**
A-greedy	70.93	8.77
For each query we considered the maximum packet processing cost solution		
	Packet processing cost (seconds)	Runtime overhead (seconds)
A-greedy*/ADWIN2	73.95	0.56
A-greedy*/Martingale Test	73.49	0.44
A-greedy*/ChangeFinder	73.68	0.23
A-greedy*/Meta-algorithm	78.61	0.06
A-greedy*/β-CUSUM	**70.45**	**0.04**
A-greedy	86.64	0.18

8. Xing, Y., Zdonik, S., Hwang, J.-H.: Dynamic load distribution in the borealis stream processor. In: Proceedings IEEE International Conference on Data Engineering (ICDE), pp. 791–802 (2005)
9. Deshpande, A., Ives, Z., Raman, V.: Adaptive query processing. Foundations & Trends in Databases **1**(1), 1–140 (2007)
10. Hellerstein, J.M., Stonebraker, M.: Predicate migration: Optimizing queries with expensive predicates. In: Proceedings ACM International Conference on Management of Data (SIGMOD), pp. 267–276 (1993)
11. Roesch, M.: Snort - lightweight intrusion detection for networks. In: Proceedings 13th USENIX Conference on System Administration (LISA), pp. 229–238 (1999)
12. DARPA: Intrusion Detection Evaluation Data Set (1998). http://www.ll.mit.edu/mission/communications/ist/corpora /ideval/data/1998data.html
13. Papoulis, A.: Probability, Random Variables, and Stochastic Processes, 3rd edn. McGraw-Hill, (1991)
14. Babcock, B., Babu, S., Datar, M., Motwani, R., Widom, J.: Models and issues in data stream systems. In: Proceedings 21st ACM Symposium on Principles of Database Systems (PODS), pp. 1–16 (2002)
15. Basseville, M., Nikiforov, I.V.: Detection of Abrupt Changes: Theory and Application. Prentice-Hall, (1993)
16. Bifet, A., Gavalda, R.: Learning from time-changing data with adaptive windowing. In: Proceedings 7th IEEE International Conference on Data Mining (ICDM), pp. 443–448 (2007)
17. Takeuchi, J., Yamanishi, K.: A unifying framework for detecting outliers and change points from time series. IEEE Transactions on Knowledge & Data Engineering **18**(4), 482–492 (2006)
18. Kifer, D., Ben-David, S., Gehrke, J.: Detecting change in data streams. In: Proceedings 30th International Conference on Very Large Data Bases (VLDB), pp. 180–191 (2004)
19. Ho, S.-S., Wechsler, H.: A martingale framework for detecting changes in data streams by testing exchangeability. IEEE Transaction on Pattern Analysis & Machine Intelligence **32**, 2113–2127 (2010)
20. Kencl, L., Schwarzer, C.: Traffic-adaptive packet filtering of denial of service attacks. In: Proceedings International Symposium on World of Wireless, Mobile & Multimedia Networks (WoWMoM), pp. 485–489 (2006)
21. Cormode, G., Muthukrishnan, S.: What's new: Finding significant differences in network data streams. IEEE/ACM Transactions on Networking **13**(6), 1219–1232 (2005)
22. Babcock, B., Chaudhuri, S.: Towards a robust query optimizer: A principled and practical approach. In: Proceedings ACM International Conference on Management of Data (SIGMOD), pp. 119–130 (2005)
23. Dasu, T., Krishnan, S., Venkatasubramanian, S., Yi, K.: An information-theoretic approach to detecting changes in multi-dimensional data streams. In: Proceedings Symposium on Interface of Statistics, Computing Science & Applications (Interface) (2006)
24. Kawahara, Y., Sugiyama, M.: Change-point detection in time-series data by direct density-ratio estimation. In: Proceedings SIAM International Conference in Data Mining (SDM), pp. 389–400 (2009)
25. Desobry, F., Davy, M., Doncarli, C.: An online kernel change detection algorithm. IEEE Transactions on Signal Processing **53**(8), 2961–2974 (2005)
26. Alippi, C., Roveri, M.: Just-in-time adaptive classifiers-part i: Detecting nonstationary changes. IEEE Transactions on Neural Networks **19**(7), 1145–1153 (2008)
27. Gnanadesikan, R., Pinkham, R.S., Hughes, L.P.: Maximum likelihood estimation of the parameters of the beta distribution from smallest order statistics. Technometrics **9**(4), 607–620 (1967)
28. Lagarias, J.C., Reeds, J.A., Wright, M.H., Wright, P.E.: Convergence properties of the nelder-mead simplex method in low dimensions. SIAM Journal on Optimization **9**, 112–147 (1998)
29. Avnur, R., Hellerstein, J.M.: Eddies: Continuously adaptive query processing. In: Proceedings ACM International Conference on Management of Data (SIGMOD), pp. 261–272 (2000)
30. El-Helw, A., Ilyas, I.F., Lau, W., Markl, V., Zuzarte, C.: Collecting and maintaining just-in-time statistics. In: Proceedings IEEE International

Table 2: Mean overall cost improvement of A-greedy* over A-greedy (%).

Experiment 1

	Abrupt 1000K	Abrupt 200K	Abrupt 50K
A-greedy*/ADWIN2	11.58%	10.99%	12.08%
A-greedy*/Martingale Test	11.70%	12.12%	14.08%
A-greedy*/ChangeFinder	11.56%	11.63%	12.19%
A-greedy*/Meta-algorithm	8.64%	4.79%	6.41%
A-greedy*/β-CUSUM	**14.06**%	**13.55**%	**14.89**%

Experiment 2

	Abrupt 1000K	Abrupt 200K	Abrupt 50K
A-greedy*/ADWIN2	**36.92**%	27.49%	20.75%
A-greedy*/Martingale Test	31.76%	29.83%	31.16%
A-greedy*/ChangeFinder	31.83%	30.48%	31.06%
A-greedy*/Meta-algorithm	31.19%	27.68%	30.76%
A-greedy*/β-CUSUM	36.58%	**34.54**%	**34.58**%

Experiment 3

	Abrupt 1000K	Abrupt 200K	Abrupt 50K
A-greedy*/ADWIN2	6.04%	6.79%	**9.80**%
A-greedy*/Martingale Test	6.57%	7.62%	9.60%
A-greedy*/ChangeFinder	6.39%	6.83%	7.25%
A-greedy*/Meta-algorithm	2.89%	-1.02%	0.002%
A-greedy*/β-CUSUM	**8.31**%	**8.62**%	9.73%

Conference on Data Engineering (ICDE), pp. 516–525 (2007)

31. Chaudhuri, S., Narasayya, V.: Automating statistics management for query optimizers. IEEE Transaction on Knowledge & Data Engineering **13**(1), 7–20 (2001)
32. Markl, V., Raman, V., Simmen, D., Lohman, G., Pirahesh, H., Cilimdzic, M.: Robust query processing through progressive optimization. In: Proceedings ACM International Conference on Management of Data (SIGMOD), pp. 659–670 (2004)
33. Chu, F., Halpern, J.Y., Seshadri, P.: Least expected cost query optimization: an exercise in utility. In: Proceedings 18th ACM Symposium on Principles of Database Systems (PODS), pp. 138–147 (1999)
34. Zadorozhny, V., Raschid, L.: Query optimization to meet performance targets for wide area applications. In: Proceedings 22nd International Conference on Distributed Computing Systems (ICDCS), pp. 271–279 (2002)
35. Braverman, V., Ostrovsky, R., Zaniolo, C.: Optimal sampling from sliding windows. In: Proceedings 28th ACM Symposium on Principles of Database Systems (PODS), pp. 147–156 (2009)
36. Chuang, K.-T., Chen, H.-L., Chen, M.-S.: Feature-preserved sampling over streaming data. ACM Transaction on Knowledge & Data Discovery **2**(4), 15–11545 (2009)
37. Aggarwal, C.C.: On biased reservoir sampling in the presence of stream evolution. In: Proceedings 32nd International Conference on Very Large Data Bases (VLDB), pp. 607–618 (2006)
38. Datar, M., Gionis, A., Indyk, P., Motwani, R.: Maintaining stream statistics over sliding windows: (extended abstract). In: Proceedings 13th ACM-SIAM Symposium on Discrete Algorithms (SODA), pp. 635–644 (2002)

Table 3: % Mean overall cost improvement of A-greedy* over A-greedy.

Experiment 4

	Transition length 10K	Transition length 25K	Transition length 50K
A-greedy*/ADWIN2	19.50%	20.38%	21.07%
A-greedy*/Martingale Test	20.65%	20.82%	21.35%
A-greedy*/ChangeFinder	20.25%	20.77%	21.06%
A-greedy*/Meta-algorithm	13.54%	14.73%	17.99%
A-greedy*/β-CUSUM	**23.90**%	**23.43**%	**23.53**%

Experiment 5

	Transition length 10K	Transition length 25K	Transition length 50K
A-greedy*/ADWIN2	11.62%	10.98%	10.46%
A-greedy*/Martingale Test	14.62%	14.26%	16.19%
A-greedy*/ChangeFinder	11.94%	13.15%	15.78%
A-greedy*/Meta-algorithm	-2.52%	1.88%	0.97%
A-greedy*/β-CUSUM	**16.71**%	**16.35**%	**17.75**%

Experiment 6

	Transition length 10K	Transition length 25K	Transition length 50K
A-greedy*/ADWIN2	27.69%	28.53%	33.56%
A-greedy*/Martingale Test	27.93%	26.30%	32.02%
A-greedy*/ChangeFinder	28.19%	27.49%	31.56%
A-greedy*/Meta-algorithm	22.92%	23.00%	30.56%
A-greedy*/β-CUSUM	**31.70**%	**32.15**%	**36.23**%

Experiment 7

	Transition length 10K	Transition length 25K	Transition length 50K
A-greedy*/ADWIN2	17.82%	18.70%	18.32%
A-greedy*/Martingale Test	19.15%	19.69%	19.00%
A-greedy*/ChangeFinder	18.63%	19.38%	18.74%
A-greedy*/Meta-algorithm	11.62%	11.81%	15.22%
A-greedy*/β-CUSUM	**21.10**%	**21.62**%	**20.73**%

Table 4: Possible parameter assignments for the β-Cusum, ADWIN2, the Martingale Test, the Meta-algorithm and the ChangeFinder algorithms.

Symbol	Value	Algorithm
k	$\{20, 30, 40, 50\}$	All
$\theta_{feedback}$	$\{10, 20, 30, 50\}$, ($\{10, 50, 100, 500, 1000\}$ for ADWIN2)	β-Cusum, ADWIN2, Meta-algorithm, ChangeFinder
L	$\{5, 10, 15\}$	ADWIN2
δ	0.01	ADWIN2
ϵ	0.9	Martingale Test
h	$\{5, 10, 25, 50, 100\}$	β-Cusum, Martingale Test, Meta-algorithm, ChangeFinder
ν	$\{0, 5, 10, 20, 30, 50\}$	β-Cusum, Martingale Test, Meta-algorithm, ChangeFinder
ζ	0.05	β-Cusum
(n, p)	{(5K,0.05),(50K,0.05),(100K,0.05),(500K,0.05),(1000K,0.05)}	Meta-algorithm
Baseline window lengths	{1000, 2400, 5000, 10000, 24900, 49900, 100000, 249900, 499900}	Meta-algorithm
T	$T \in \{5, 10, 20\}$	ChangeFinder