# Lifting the burden of history in adaptive ordering of pipelined stream filters

Efthymia Tsamoura, Anastasios Gounaris, Yannis Manolopoulos

*Department of Informatics, Aristotle University of Thessaloniki, Greece*
{etsamour|gounaria|manolopo}@csd.auth.gr

***Abstract*— Ordering of commutative and correlated pipelined stream filters in a dynamic environment is a problem of high interest due to its application in data stream scenarios and its relevance to many query optimization problems. Current state-of-the-art adaptive techniques continuously reoptimize filter orderings utilizing statistics that are collected during the execution of the query; however, both the up-to-date and the out-of-date collected statistical data may be considered with the consequence of not always taking effective reoptimization decisions. In this work, we propose a technique for adaptive filter ordering that tries to lift the burden of out-of-date statistical data after detecting changes in the filter drop probabilities. To this end, we propose a novel drop probability change detection algorithm. The experimental evaluation shows that the proposed technique can yield significant performance improvements while decreasing the runtime overhead.**

## I. Introduction

Nowadays, an increasing number of applications deals with streaming data, e.g., network monitoring, online processing of sensor data, etc. Since the queries that are submitted to a data stream management system are usually long-running or even continuous, the characteristics of the underlying execution environment, as well as the characteristics of the streaming data may be significantly different from those when the query was initiated. This necessitates the development of adaptive query processing techniques [1], which can be deemed as resulting to self-optimizing query processors. The adaptive techniques employ a three phase procedure, called adaptivity loop: (i) they collect runtime statistics (statistics collection or measurement update phase); (ii) they analyze the efficiency of the current execution plan with respect to the collected statistics (analysis phase); and (iii) they perform reoptimization if the analysis indicates that the previously selected plan is inefficient (reoptimization phase).

A problem of high interest deals with the ordering of commutative filters in a dynamic environment; this problem relates to additional query optimization problems (e.g., multi-way join query optimization [2], [1]). In this work, we deal with a flavor of the problem where a single data source streams the input tuples. The input tuples are pipelined among the filters, while the filters are commutative and associated with (i) processing costs and (ii) drop probabilities. In the generic case, the drop probabilities are correlated, i.e., a filter's drop probability depends on the filters upstream in the ordering. The goal is to find a filter ordering, such that the total processing cost of the input tuples is minimized. Moreover, since the execution environment is dynamic, a filter ordering must adapt to the changes in the execution environment.

In order to build filter orderings that are consistent with the varying characteristics of the execution environment, an adaptive technique, called A-greedy, is proposed in [2]. Although this technique collects runtime statistics, it may not always take effective reoptimization decisions, because it considers both up-to-date and out-of-date statistical data when investigating filter reordering.

In this work, we propose a novel technique for adaptive filter ordering that lifts the burden of out-of-date statistical data when performing adaptive query optimization. The technique first learns the filter drop probabilities based on runtime collected statistical information. It then checks whether the filter drop probabilities have changed. If this is the case, the out-of-date statistical data that have been collected so far is dropped, the filter drop probabilities are re-learned using only up-to-date statistics, and the algorithm checks whether the changes have rendered the current filter ordering suboptimal in order to reoptimize it. As such, adaptivity decisions are based on updated data.

The proposed technique utilizes a change detection algorithm in order to learn and check for changes the filter drop probabilities. Although several state-of-the-art change detection algorithms can be employed for change detection, our technique performs change detection through a novel algorithm, called BCS, which is tailored to drop probability learning and change detection. Its high accuracy in detecting drop probability changes and its low run-time overhead compared with state-of-the-art change detection algorithms (e.g., [3], [4]) render BCS more suitable for general query processing purposes.

We focus on abrupt changes. In many real life applications, malfunction or malicious system behavior phenomena (e.g., network intrusion) exhibit abrupt change patterns ([5]). For example, regarding network intrusion detection applications ([5]), many network attacks, such as denial of service, lead to a high and abrupt increase in the total number of packets that are transmitted to the victim machine. As such, any filter with respect to the number of packets may experience abrupt changes in its drop probability. The evaluation results show that, in scenarios where abrupt changes in filter drop probabilities occur, the proposed technique can significantly improve the performance of the resulting filter ordering compared to A-greedy while decreasing the runtime overhead.

The paper is organized as follows. Section II provides background information related to A-greedy ([2]), defines the problem that we deal with and gives a motivating example. Section III presents the new change detection algorithm BCS and Section IV presents the complete technique into which BCS is encapsulated. The experimental evaluation is done in Section V. Section VI briefly presents the related work and, finally, Section VII concludes the paper.

## II. Background and Problem Formulation

Let the query to be evaluated be a conjunction of $N$ commutative filters $F_1, ..., F_N$, to be applied to the tuples from a streaming relation. The execution plan forms a linear ordering of all filters. We denote the per tuple cost of the filter placed in the $i$-th position of the current ordering by $c(i)$, and the probability that a tuple does not satisfy the condition of the filter in the $i$-th position, and thus drops the tuple, by $d(i)$. Since, in most real applications, the drop probabilities are correlated, we also define the conditional probability $d(i|i-1)$, which corresponds to the probability that the filter placed at the $i-th$ position will drop a tuple from the input stream, given that this tuple was not dropped by any of the filters that are placed up to the $i-th$ position in the current ordering. For simplicity, we assume that the per tuple processing costs $c(i),\ 1 \leq i \leq n$ are the same; extensions of our technique to the generic case where costs differ are straightforward and are omitted due to space restrictions.

As shown in [2], minimizing the total cost is recast as the problem of building an ordering that, for each position $i$ in the ordering, satisfies the so-called greedy invariant given by:

$$d(i|i-1) \geq d(j|i-1), 1 \leq i \leq j \leq N$$

Since the actual filter drop probabilities are not known a priori, they are estimated after collecting runtime statistics. In particular, in A-greedy, which is the most advanced solution to this problem to date, every input tuple that is rejected by at least one of the input filter is probabilistically chosen for profiling [2]. If an input tuple is chosen for profiling, it is virtually processed by the input filters and it is recorded in an $N$ attribute tuple, called *profile tuple*, which denotes which of the filters eventually drop the specific tuple and which not; if the $i-th$ input filter drops the profiled tuple, then the $i-th$ attribute is true; otherwise it is false. The profile tuples are stored in a fixed-size sliding window $W$, called *profile window*. A-greedy utilizes the contents of $W$ to estimate the conditional drop probabilities of the input filters. For example, the drop probability $d(j|i),\ 1 \leq i < j \leq N$ is approximated by the total number of the profile tuples that are currently stored in $W$ which are rejected by the $j-th$ filter in the ordering and are not rejected by any of the filters that are placed up to the $(i+1)-th$ position in the current ordering (the reader is referred to [2] for full details).

A problem encountered when selecting a profile window of static size is that data (i.e., profile tuples) may co-exist for which the same input filters have significantly different drop probabilities. The coexistence of such inconsistent, in terms of statistics, profile tuples can lead to wrong filter drop probability estimates and subsequently to wrong or delayed re-optimization decisions.

*Example 1:* Assume that there exist two independent filters $F_1$ and $F_2$, for which the drop probabilities change abruptly at one point (e.g., they depend on a day period). Also assume that the input stream consists of 100,000 tuples, every input tuple is used for statistics collection and the profile window is large enough to store all tuples. During the first 50,000 tuples, filter $F_1$ unconditionally drops a tuple with constant probability $0.925$, while $F_2$ drops a tuple with constant probability $0.05$. A-greedy converges to the optimal ordering $F_1F_2$, since exploiting the profile tuples that have been added to $W$ so far, it can easily be derived that the drop probability of $F_1$ is higher than that of $F_2$. For the last 50,000 tuples the drop probability of $F_1$ becomes 0.15 and the drop probability of $F_2$ becomes 0.6. Although the optimal ordering now becomes $F_2F_1$, A-greedy cannot proceed to reoptimization, because the out-of-date contents of $W$ bias the maintained drop probability estimates towards the initial ordering. $\square$

In the above example, if we were able to detect the stream points where the drop probabilities of the filters have changed and subsequently erase the tuples of the profile window that arrived before the change, then an adaptive ordering algorithm would be capable of converging to the optimal ordering. This is exactly what our proposal is capable of doing.

## III. The BCS change detection algorithm

In this section, we present a new online algorithm dedicated to drop probability change detection, called BCS. We first divide the profile window into non-overlapping segments of size $k$, and for each new segment, we estimate the drop probabilities taking into account only the segment contents. Since we assume fixed sized sliding profile windows, these estimates are produced periodically and they are fed to the change detection algorithm. We denote the $j$-th such estimate as $\hat{d}_j$.

We start from considering an existing method, namely the CUSUM one, which is one of the first methods introduced in the literature for change detection [6]. CUSUM assumes that we know the probability distributions of the data prior and after a change in drop probability, and these probabilities distributions are denoted as $P_0$ and $P_1$, respectively. CUSUM relies on the observation that, upon receiving a new drop probability estimate $\hat{d}_j$, if the current distribution is $P_0$, then the probability that $\hat{d}_j$ is produced under $P_0$ is higher than $P_1(\hat{d}_j)$ and the vice versa, and thus 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. Thus, every time a new item $\hat{d}_j$ arrives, the CUSUM algorithm updates a cumulative sum $S_j$ as follows:

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

where $S_0 = 0$. If a probability distribution change from $P_0$ to $P_1$ occurs, then the values of the log-likelihood ratios that are estimated as new data items arrive are positive, and thus the cumulative sum $S_j$ continuously increases. The CUSUM method 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 [6] 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 [7] that is based on the assumption that the $P_0$ and $P_1$ distributions are normal.

However, in our case, we have strong evidence that drop probabilities better fit a beta distribution [8]. Thus we develop an online version of CUSUM that assumes beta distributions, and we term this method as BCS (for online Beta distribution-based Cumulative Sum). In BCS, a training step is firstly adopted to derive the parameters of the base (i.e., prior to change) beta distribution. After that, the original test in Eq. (1) is employed. The exact phases of the BCS algorithm are as follows:

*Train step*: The training phase requires a set $\mathcal{D}$ of drop probability estimates of size $m = |\mathcal{D}|$. As the number of data that is used during the training phase increases, the actual filter drop probability distribution is learned more accurately at the expense of a longer training period. In that phase we derive for the parameters $\alpha$ and $\beta$ of the base beta distribution, their single-value estimates $\alpha_{|\mathcal{D}|}$ and $\beta_{|\mathcal{D}|}$ and their associated confidence intervals $[\alpha^{lo} \ \alpha^{up}]$ and $[\beta^{lo} \ \beta^{up}]$, assuming a confidence level of $\zeta$. More details are given in Section V-A. After finishing the training phase, the cumulative sum in Eq.(1) is set to 0.

*Change detection step*: Every time a drop probability estimate $\hat{d}_j$ is supplied to BCS, the mean value $\mu_j$ and the standard deviation $\sigma_j$ of the $j > m$ drop probability estimates seen so far (including those of the training set) are computed:

$$\mu_j = \mu_{j-1} + (\hat{d}_j - \mu_{j-1})/j \tag{2}$$

$$\sigma_j = \sqrt{\frac{(j-1)\sigma_{j-1}^2 + (\hat{d}_j - \mu_{j-1})(\hat{d}_j - \mu_j)}{j}} \tag{3}$$

and the $\alpha_j$ and $\beta_j$ parameters of the beta distribution are then computed by

$$\alpha_j = (\mu_j(1-\mu_j)/\sigma_j - 1)\, \mu_j \tag{4}$$

$$\beta_j = (\mu_i(1-\mu_j)/\sigma_j - 1)\, (1-\mu_j) \tag{5}$$

If $\alpha_j$ or $\beta_j$ do not lie within the confidence intervals estimated during the training step, then it is assumed that the item $\hat{d}_j$ is produced by a different beta 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. In order for the BCS algorithm to be operational again, a training phase must be applied with a new training set, since the previously found confidence intervals of the beta distribution are estimated using past data items.

Neither CUSUM nor BCS can determine the most probable changepoint, i.e., although they are capable of detecting changes, they cannot provide any information regarding the most probable changepoint. In order to overcome this limitation, we assume that the change is initiated at the $\nu - th$ most recent drop probability estimate that has been supplied, where $\nu > 0$ is an empirically configured parameter. Finally, threshold $h$ is empirically set in order to maximize the ability in detecting changes, while keeping the faulty detected changes as low as possible. The impact of these parameters is evaluated in Section V-A.

The runtime computational complexity of BCS is $O(1)$, as each time a new drop probability estimate is fed to the algorithm, the mean value and the standard deviation of the estimates seen so far are incrementally computed (see Eq.(2) and Eq.(3)). As such, BCS's runtime overhead is significantly lower with respect to other state-of-the-art change detection methods, e.g., [3], [4].

## IV. The proposed adaptive technique

The rationale behind the proposed technique is that, when at least one filter drop probability changes, then all of the profile tuples that are currently stored in the profile window and correspond to data that has arrived before the occurred change is considered to be out-of-date. For this reason, every time BCS detects a drop probability change, the technique erases the out-of-date data of the profile window and the filter drop probabilities are relearned utilizing the data tuples left in $W$. Each drop probability $d(j|i), 1 \leq i \leq j \leq N$, is checked for changes with the help of a different instance BCS.

As mentioned already, the drop probability estimates that are fed to the BCS algorithm to perform change detection are estimated after dividing the profile window into non-overlapping segments of size $k$. The estimate $\hat{d}$ of $d(j|i)$ is equal to the proportion of the profile tuples in the $k$-size segment that are rejected by the $j - th$ filter in the ordering and are not rejected by any of the filters that placed up to the $(i+1) - th$ position. To summarize, every time a batch of $k$ profile tuples is added to the profile window, the drop probability estimates are derived from this batch of tuples and are subsequently fed to the BCS algorithm to check the filter drop probabilities for changes.

The adaptivity loop of the proposed technique is given below. In the measurement update phase, the conditional drop probabilities are checked for changes. If a change has occurred the most probable timepoint where the change in the data characteristics has started is estimated, as described in the previous section. In addition, the out-of-date profile tuples are removed from the profile window and, second, the change detection algorithms are trained to learn the new conditional filter drop probabilities from the up-to-date contents of $W$. Then the analysis phase is triggered. In the analysis phase, the framework checks for violations of the greedy invariant. When the invariant is violated, then the reoptimization phase is executed and the filters are reordered. Compared to A-Greedy [2], our approach shares the same analysis and reoptimization phases, however it enters those phases less often and only under the condition that the monitored drop probabilities have changed; note that the analysis phase in A-Greedy is performed continuously, i.e., for each update in the profile window [2], [1].

## V. Experimental evaluation

The conducted experiments aim, first, to compare the performance and the runtime overhead of the proposed technique with the performance and the runtime overhead of A-greedy and, second, to study how does the above criteria change with respect to (i) the algorithm that is employed for drop probability change detection –to this end, we consider the martingale test (MT) [3] and ADWIN2 [4]) change detection algorithms–, and (ii) the frequency of the filter drop probability changes. The key observations are: (i) the proposed technique yields significant performance improvements (up to $55\%$) over A-greedy; (ii) the runtime overhead of the proposed technique when employing BCS is up to 37 $\%$ lower than that of A-greedy; and (iii) BCS outperforms MT and ADWIN2 in terms of precision and recall, and incurred overhead. All experiments were performed on an Intel i5 Linux machine with 3.7 GB memory.

### *A. Parameter tuning and comparison of change detection alternatives*

In this section, we present the methodology that we adopted to tune the $h, k, \nu$ and $m$ parameters of the BCS and the MT algorithms. The rest of the parameters of MT and ADWIN2 are set according to [3] and [4], respectively. The parameter tuning procedure consists of two steps. In the first step, we perform change detection employing the BCS (or the MT) algorithm with different values for the $h$ (cumulative sum threshold), $k$ (segment size), $\nu$ (used in changepoint detection) and $m$ (train set size) parameters. The characteristic that is being checked for changes is the selectivity of an attribute. From the change detection results that are derived, we compute precision and recall. Precision is the probability that a detected change is actually correct, while recall is the probability that an actual change is detected. In the second step, we select the $h, k, \nu$ and $m$ parameter values that optimize precision and recall.

The data tuples that are supplied to the BCS and the MT algorithms for change detection have the following characteristics: (i) there is only one attribute per tuple that takes only $\{0, 1\}$ values; (ii) the selectivity of the attribute is given by the proportion of tuples with attribute value equal to 1; (iii) the selectivity of the attribute changes every 400K tuples and there are 50 changes in total; and (iv) the selectivity changes are random and the smallest change is $10\%$.

In the following, we present the parameter tuning procedure for BCS. The different assignments of the $h, k, \nu$ and $m$ parameters are $h = \{5, 10, 25, 50, 100\}$, $k = \{10, 20, 30, 40, 50\}$, $\nu = \{5, 10, 20, 30, 50\}$, $m = \{5, 10, 20, 30, 50\}$, while $N = 15$, $|W| = 60K$ and $\zeta = 0.05$. The BCS algorithm performs change detection on the selectivity of the single attribute of the produced data tuples after sampling $1\%$ of the data tuples. It is executed totally $|h| \times |k| \times |\nu| \times |m|$ times, where $|.|$ denotes the number of the different values that a parameter can take. For every BCS execution, we compute its associated precision and recall measures taking into account all of the correctly, erroneously and missed detected attribute selectivity changes.

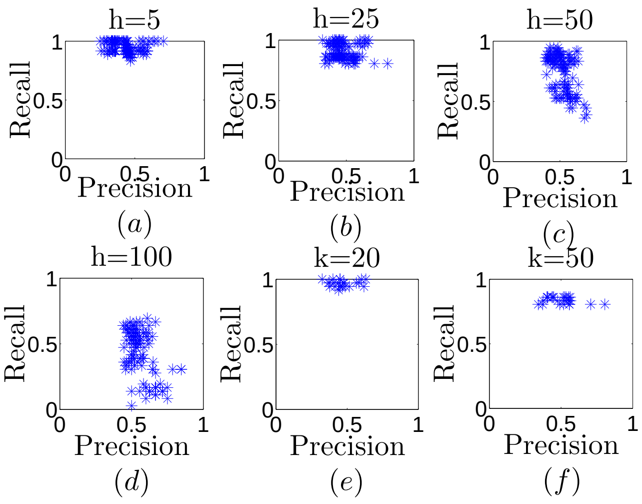
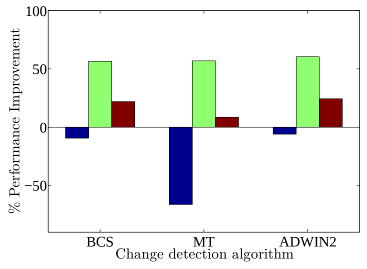


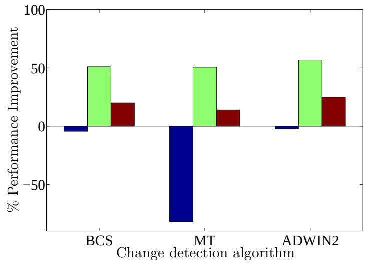
Fig. 1. (a)-(d)Precision and recall values obtained after employing BCS for attribute selectivity change detection with different values of the $k, \nu$ and $m$ parameters and (a) $h = 5$, (b) $h = 25$, (c) $h = 50$ and (d) $h = 100$. (e)-(f)Precision and recall values obtained after employing BCS for attribute selectivity change detection with different values of the $\nu$ and $m$ parameters and (e) $k = 20$, (f) $k = 50$. Parameter $h$ is set to 25.

Figure 1(a) shows the obtained precision and recall values for every possible value combination of the $k, \nu$ and $m$ parameters and $h = 5$. Similarly, Figures 1(b)-(d) show the obtained precision and recall values for every possible value combination of the $k, \nu$ and $m$ parameters and $h = 25$, $h = 50$ and $h = 100$, respectively. We observe that as the value of parameter $h$ increases, the precision of the BCS algorithm (slightly) increases too, while its recall decreases significantly. This is attributed to the fact that under high $h$ values, only large-scale drop probability changes are detected, ignoring the small-scale ones. We fix $h$ to 25, since, for that configuration, BCS has the highest recall and, approximately, equal precision values with respect to the ones derived when $h = 50$ and $h = 100$. Particularly, when $h = 5$, the precision and recall values range in $[0.25\ 0.70]$ and $[0.83\ 1.0]$, respectively. When $h = 25$ the precision is in $[0.32\ 0.80]$ and the recall is in $[0.80\ 0.92]$. The corresponding precision and recall intervals are $[0.39\ 0.70]$ - $[0.36\ 0.94]$ for $h = 50$ and, finally, $[0.44\ 0.84]$ - $[0.02\ 0.69]$ for $h = 100$.

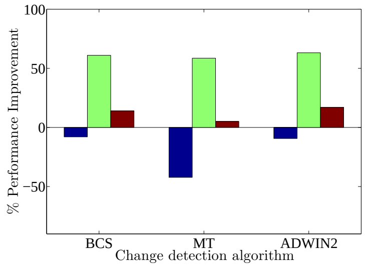
Figure 1 also shows the obtained precision and recall values for every possible value combination of the $\nu, m$ parameters and $k = 20$ (Figure 1(e)) and $k = 50$ (Figure 1(f)). When $k = 20$ the corresponding precision and recall value intervals are $[0.32\ 0.63]$ and $[0.89\ 0.92]$, respectively. When $k = 50$, these intervals become $[0.34\ 0.80]$ and $[0.80\ 0.86]$. For both cases, the highest precision and recall values are met when



(a)



(b)



(c)

Fig. 2. Maximum (middle bar) and median (rightmost bar) $\%$ performance improvement and maximum performance degradation (leftmost bar) of the change detection-based techniques over A-greedy. Results of (a) Experiment 1, (b,c) Experiment 2.

$\nu = 5$, which means that an actual change is being detected after $\nu = 5$ segment-wise estimates have been presented to the algorithm after a change has occurred. We fix $k$ to 20 and $\nu$ to 5, since this parameter combination leads to higher recall values.

The only parameter that has not been fixed yet is $m$. We fix parameter $m$ to 20, since this parameter assignment leads to the highest precision/recall values ($0.63$ and $0.92$, respectively) over the two other choices. To summarize, we have selected the value combination $h = 25$, $k = 20$, $\nu = 5$ and $m = 20$, which corresponds to $0.63$ and $0.92$ average precision and recall values, respectively.

After employing a similar procedure for tuning the parameters of the MT algorithm we select the $h = 25$, $k = 20$, $\nu = 10$, $m = 20$ parameter assignment, which leads to $0.52$ precision and $0.83$ recall values. Performing change detection through the ADWIN2 algorithm over the same testset of data tuples we derive the $0.35$ and $0.95$ precision and recall values, respectively. The relatively low precision that the ADWIN2 algorithm has is attributed to the fact that, from the time since a change in a filter's drop probability is first detected by ADWIN2 and for a short period, the algorithm is unstable and continuously detects changes every time new profile data arrives.

The above show that BCS is the most efficient change detection for detecting drop probability changes. To summarize, the average precision and recall values of the BCS algorithm for its optimal configuration are 0.63 and 0.92, respectively. Regarding the MT algorithm, the precision and recall values for its own optimal configuration are 0.52 and 0.83, while the precision and recall of ADWIN2 are 0.35 and 0.95, respectively.

## B. *Performance analysis*

In order to produce correlated input tuples, we adopt the methodology described in [2]. The filter drop probabilities change randomly every 250K, 500K, or 1000K tuples, while each filter drop probability changes 5 times totally when processing an input data stream. All changes are abrupt and new selectivities are at least 10% different than the previous ones. We keep $N = 15$, $|W| = 60K$, and $\zeta = 0.05$ and we set the rest of the parameters of the change detection algorithms to their optimal configuration in terms of precision-recall based on the results presented above.

In *Experiment 1*, we study the impact of the different change detection algorithms on the quality of the orderings compared to A-Greedy. Fifty different data streams have been produced, where in each stream the total number of attributes is fifteen (and thus fifteen filters are considered), the filter drop probabilities change every 500K tuples and each filter drop probability changes 5 times in total.

Figure 2(a) shows the maximum (middle bar) and median (rightmost bar) $\%$ performance improvement, as well as, the maximum performance degradation (leftmost bar) under the different filter ordering techniques. The performance improvement/degradation is estimated by $(\frac{Cost(\mathcal{O})-Cost(\mathcal{O}^{'})}{Cost(\mathcal{O}})$, where $Cost(\mathcal{O})$ is the cost of processing tuples with a filter ordering produced by A-greedy ($\mathcal{O}$) and $Cost(\mathcal{O}^{'})$ is the cost of processing tuples with a filter ordering produced by the techniques that detect changes ($\mathcal{O}^{'}$) when the drop probability change detection is done with the BCS, the MT, or the ADWIN2 algorithms. Note that the runtime overhead associated with the adaptivity loop of the change detection-based techniques is not considered, but will be examined later; the costs above take into account only the quality of the orderings selected.

A main observation is that the maximum performance improvement of all the change detection-based techniques over A-greedy is approximately $55\%$, while the median of the performance improvement over the different 50 input data streams is approximately $10\%$ when any of the BCS, MT, ADWIN2 algorithms is employed. The efficiency of an algorithm in detecting filter drop probability changes strongly affects the performance, since, when one or more filter drop probability change is not detected, the out-of-date data are not eliminated during query reoptimization. From Figure 2(a) we can see that the maximum performance degradation is $6\%$, $66\%$ and $5\%$ when the BCS, the MT, or the ADWIN2 algorithm is employed, respectively. The high performance degradation when employing the MT algorithm is because of the high number of missed filter drop probability change detections.

In *Experiment 2*, we study how the performance is affected by the frequency of the drop probability changes. To this end, we have repeated Experiment 1 with $freq = 250K$

and $freq = 1000K$. The performance results are shown in Figures 2(b) and 2(c), respectively. We can observe that as the frequency of changes increases, the performance improvements decrease, and, on the contrary, as the frequency of changes decreases, the performance improvements increase. This happens due to the fact that a filter drop probability change is not immediately detected after its occurrence, but after a specific delay, as discussed earlier. As the frequency of drop probability changes increases (decreases), the percentage of tuples that are processed before the next change occurs decreases (increases), with a corresponding decrease (increase) in the obtained performance improvements.

At the end of this section, and due to lack of space, we present a summary of the runtime overhead characteristics of the proposed technique. The first observation is that the runtime overhead of the proposed technique when employing the BCS algorithm is lower than that of A-greedy, since the number of conducted reoptimizations is up to $37\%$ lower than of A-greedy. Note that despite the increased number of reoptimizations, the performance of A-greedy is lower since it does not eliminate the out-of-date data from the profile window. On the other hand, the runtime overhead incurred when the other two change detection algorithms are employed (i.e., the MT or the ADWIN2 algorithms) is approximately ten times higher than that of A-greedy. The increased overhead is due to the higher complexity of the MT and ADWIN2 algorithms and to the relatively high number of unnecessary re-optimizations, because of their low accuracy in distinguishing the actual drop probability changes from temporal drop probability fluctuations.

In summary, the results above show that (i) BCS can incur significant performance improvements compared to A-greedy, while being characterized by lower overhead (up to 37 %); and (ii) BCS algorithm is more appropriate than the other two alternatives to change detection: the cost of the resulting filter orderings is approximately equal to the cost of the orderings produced when employing ADWIN2 and lower than that of MT, while the runtime overhead of BCS is much lower than that of ADWIN2.

## VI. Related work

The problem that is addressed in this work relates with the problems of operator ordering and adaptive query processing. In the area of operator ordering, the majority of the work assume independent operators (e.g., [9]), while the proposals that deal with correlated operators assume a static execution environment (e.g., [10]), with the exception of proposals such as [2]. Note that in a centralized environment, the problem of pipelined filter ordering can be optimally solved in polynomial time only if the filter drop probabilities are independent [11]. Several proposals have been presented in the literature during the last decade that introduce adaptive query processing techniques [1]. Their common characteristic is that they employ a three step adaptivity loop. However, the techniques proposed so far tend not to pay attention to the statistics collection phase, e.g., they do not evaluate the freshness of the selected statistical data, but mainly emphasize on responding as fast as possible to changes in the execution environment (e.g., [12]).

Regarding the problem of change detection in data streams, several other algorithms have been recently proposed in the literature (e.g., [13], [3], [4]). However, to the best of our knowledge, no algorithm is tailored to detecting changes in a drop probability distributions, which can be approximated with a beta distribution [8]. Additionally, the run-time overhead of the majority of the change detection algorithms is higher than that of the BCS algorithm, e.g., every time a new data item arrives a probability density function or a support vector machine may have to be adjusted, which is inappropriate for online settings.

## VII. Conclusions

In this work, we propose an adaptive technique for pipelined ordering of correlated filters. Our main contribution is twofold. First, we have presented an approach according to which only up-to-date statistical data are considered during adaptive filter ordering by employing an algorithm that learns and checks for changes the filter drop probabilities. Second, a novel algorithm is proposed that is tailored to drop probability distribution change detection. The evaluation results presented provide evidence that the proposed technique can improve the performance of the resulting filter orderings over state-of-the-art techniques such as A-Greedy while incurring lower runtime overhead. As a future work, we intend to conduct more thorough experiments and also evaluate our proposal against real evolving data streams, where drop probability changes may occur gradually rather than abruptly.

## References

[1] A. Deshpande, Z. Ives, and V. Raman, “Adaptive query processing,” *Foundations and Trends in Databases*, vol. 1, no. 1, pp. 1–140, 2007.

[2] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, “Adaptive ordering of pipelined stream filters,” in *SIGMOD*, 2004, pp. 407–418.

[3] S.-S. Ho and H. Wechsler, “A martingale framework for detecting changes in data streams by testing exchangeability,” *IEEE TPAMI*, vol. 32, pp. 2113–2127, 2010.

[4] A. Bifet and R. Gavalda, “Learning from time-changing data with adaptive windowing,” 2007, pp. 443–448.

[5] R. Lippmann, J. W. Haines, D. J. Fried, J. Korba, and K. Das, “The 1999 darpa off-line intrusion detection evaluation,” *Computer Networks*, vol. 34, pp. 579–595, 2000.

[6] M. Basseville and I. V. Nikiforov, *Detection of abrupt changes: theory and application*. Prentice-Hall, Incorporation, 1993.

[7] C. Alippi and M. Roveri, “Just-in-time adaptive classifiers-part i: Detecting nonstationary changes,” *IEEE TNN*, vol. 19, no. 7, pp. 1145–1153, 2008.

[8] B. Babcock and S. Chaudhuri, “Towards a robust query optimizer: A principled and practical approach,” in *SIGMOD*, 2005, pp. 119–130.

[9] J. M. Hellerstein and M. Stonebraker, “Predicate migration: Optimizing queries with expensive predicates,” in *SIGMOD*, 1993, pp. 267–276.

[10] K. Munagala, U. Srivastava, and J. Widom, “Optimization of continuous queries with shared expensive filters,” in *PODS*, 2007, pp. 215 – 224.

[11] R. Krishnamurthy, H. Boral, and C. Zaniolo, “Optimization of nonrecursive queries,” in *VLDB*, 1986, pp. 128–137.

[12] R. Avnur and J. M. Hellerstein, “Eddies: Continuously adaptive query processing,” *SIGMOD Record*, vol. 29, no. 2, pp. 261–272, 2000.

[13] F. Desobry, M. Davy, and C. Doncarli, “An online kernel change detection algorithm,” *IEEE TSP*, vol. 53, no. 8, pp. 2961–2974, 2005.