

Self-tuning Query Mesh for Adaptive Multi-Route Query Processing

Rimma Nehme
Elke Rundensteiner
Elisa Bertino

CSD TR #08-018
July 2008

Self-Tuning Query Mesh for Adaptive Multi-Route Query Processing

Rimma V. Nehme
Purdue University
West Lafayette, IN 47906
rnehme@cs.purdue.edu

Elke A. Rundensteiner
Worcester Polytechnic Institute
Worcester, MA 01608
rundenst@cs.wpi.edu

Elisa Bertino
Purdue University
West Lafayette, IN 47906
bertino@cs.purdue.edu

ABSTRACT

In real-life applications, different subsets of data may have distinct statistical properties, e.g., various websites may have diverse visitation rates, different categories of stocks may have dissimilar price fluctuation patterns. For such applications, it can be fruitful to eliminate the commonly made *single execution plan* assumption and instead execute a query using several plans, each optimally serving a subset of data with particular statistical properties. Furthermore, in dynamic environments, data properties may change continuously, thus calling for adaptivity. The intriguing question is: can we have an execution strategy that (1) is plan-based to leverage on all the benefits of traditional plan-based systems, (2) supports multiple plans each customized for different subset of data, and yet (3) is as adaptive as “plan-less” systems like Eddies? While the recently proposed *Query Mesh (QM)* approach provides a foundation for such an execution paradigm, it does not address the question of adaptivity required for highly dynamic environments. In this work, we fill this gap by proposing a *Self-Tuning Query Mesh (ST-QM)* – an adaptive solution for content-based multi-plan execution engines. *ST-QM* addresses adaptive query processing by abstracting it as a *concept drift problem* – a well-known subject in machine learning. Such abstraction allows to discard adaptivity candidates (i.e., the cases indicating a change in the environment) early in the process if they are insignificant or not “worthwhile” to adapt to, and thus minimize the adaptivity overhead. A unique feature of our approach is that all logical transformations to the execution strategy get translated into a *single inexpensive physical operation* – the classifier change. Our experimental evaluation using a continuous query engine shows the performance benefits of *ST-QM* approach over the alternatives, namely the non-adaptive and the Eddies-based solutions.

1. INTRODUCTION

1.1 Motivation

Many modern applications deal with data that is updated continuously and needs to be processed in real-time [2, 6, 38,

40]. Examples include network monitoring, financial monitoring, fraud detection, etc. Even if given a highly effective query execution strategy at the start, data and system characteristics may change considerably during the query lifetime, making it necessary to adapt the execution strategy. This pressing problem of adaptivity has become an important and active area of research in recent years [3, 21, 31, 36, 39]. Moreover, real-life datasets typically tend to be non-uniformly distributed [22], e.g., sensor networks, moving objects, etc. Enforcing a single query plan execution strategy, as is the defacto standard for most database technology, may lead to serious performance deterioration in situations where subsets of data may have very different statistics [8].

Motivating Example. Consider the following continuous query used in a financial monitoring application to correlate stock prices with current events: `SELECT * FROM stocks, news, currency, blogs WHERE blogs.subject = stocks.industry AND stocks.region = news.region AND news.country = currency.country AND stocks.change% > 15`. To answer this query, the data may be acquired from several stock exchanges, geographically dispersed news sources and blogs that may be updated at various rates, e.g., based on the location or the time zone. Arriving from various data providers, the respective data subsets are likely to have different statistical properties, such as their data values, their frequency, and arrival rates. To complicate matters, in reaction to the same real-life events, prices of stocks may fluctuate rather differently over time. News about political instability in certain geographical regions may affect positively the stocks of defense-related companies while having an opposite or no effect on other sectors. Change in data values and their frequencies may lead to the disappearance of existing and the emergence of new statistically similar data subsets, consequently leading to changes in query execution statistics. To ensure good performance at all times, a database system must be capable to continuously identify such distinct data subsets and to adapt the execution strategy accordingly.

Unlike most adaptive solutions, e.g., [3, 31, 36], our work does not focus on adapting a *single execution plan* for a query, but rather on adapting the *multi-plan-based* (or we refer to it as *multi-route-based*) *execution strategy*¹ [35].

1.2 Multi-Route Query Processing

Multi-route query processing is an effective approach,

Permission to copy without fee all or part of this material is granted provided that the copies are not made or distributed for direct commercial advantage, the ACM copyright notice and the title of the publication and its date appear, and notice is given that copying is by permission of the ACM. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires a fee and/or special permissions from the publisher, ACM.

EDBT 2009, March 24–26, 2009, Saint Petersburg, Russia.
Copyright 2009 ACM 978-1-60558-422-5/09/0003 ...\$5.00

¹We use terms “plan” and “route” interchangeably. Both mean the same thing in the context of this paper. To prevent any confusion with Eddies-based systems [12, 33], a route in our work is a fully pre-computed query plan.

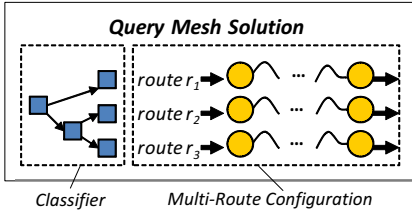


Figure 1: Overview of a query mesh solution.

especially when datasets are non-uniformly distributed. Instead of forcing all data to be processed by the same single plan, a multi-route solution effectively supports the concurrent usage of multiple plans to evaluate a query. The recently proposed approach called *Query Mesh (QM)* [35] supports such execution paradigm very efficiently.

QM employs a practical middle-ground strategy between the two query optimization extremes – the solutions that employ a “monolithic” single execution plan strategy for all input data, e.g., nearly all commercial DBMSs [1, 11, 29], and the systems like Eddies that employ a fine-grained “plan-less” approach, where instead of predetermined plans, at runtime the Eddy operator determines, one-at-a-time, the next operator, that the tuples must visit for processing [33]. The former strategy may miss critical opportunities to improve query execution performance when data has distinct data subsets. The latter strategy may incur the frequent and often unnecessary re-optimization overhead [12]. Since there are no pre-computed plans, the system continuously “re-discovers” the best routes for the arriving tuples.

QM provides the middle-ground by using *multiple pre-computed plans*, each optimized for a subset of data with certain statistical properties, and the *classifier* component to determine which data subsets should be processed by which of the pre-computed routes (Figure 1)². The *QM* framework, implemented in a continuous query processing engine [13], has been shown to be very effective for both real and synthetic data compared to the single plan and the Eddy-based query processing alternatives. Experimental evaluation in [35] has shown that the *QM*-based optimizer can find a good *QM* solution in a reasonable amount of time, and for skewed data, it typically performs much better in both response time and throughput than alternative systems, namely the single plan and the Eddy-based systems. For more details on *QM* framework we refer the reader to Section 2.

1.3 Adaptive Multi-Route Query Processing

The open question now arises, if a multi-plan based execution strategy, such as *QM*, can be as adaptive as “plan-less” systems like Eddies? The need for adaptivity is evident. Even with an initial good choice of a *QM* solution, after some time, data characteristics, e.g., data values, their frequencies and execution statistics, such as operators’ costs and selectivities, may change considerably requiring to adapt the execution strategy. The fundamental challenge for *QM* adaptivity is the problem of determining the discrepancy between the previously constructed *QM* model³ and the currently most suitable *QM* solution based on the new

²In the rest of the paper, we will refer to the combination of two components: classifier and a set of execution routes – a *QM solution*.

³A *QM* solution represents a particular “model” of execution, as determined by the classifier and the set of execution routes. In machine learning, this term is commonly used to refer to classifier-based systems.

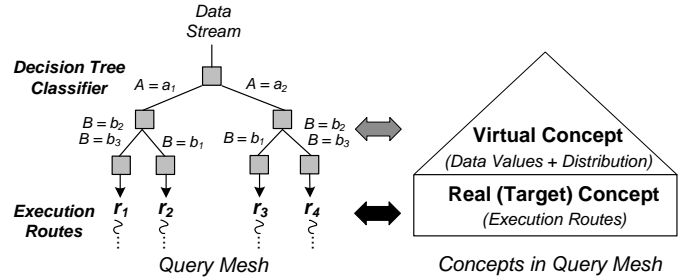


Figure 2: Virtual and real concepts in *QM*.

data characteristics, i.e., its values and its statistics. In machine learning, such discrepancy is called a *concept drift* [27]. Concept drifts happen when a model built in the past is no longer applicable to the current data.

In the context of *QM*, the change may occur at either the *target concept* level, i.e., the routes in the multi-route configuration, or at the underlying *data distribution* level, i.e., the data values and their frequencies (see Figure 2). The necessity to change the current model due to changes in the data distribution is called a *virtual concept drift* [14]. A *real concept drift* may occur for instance when more accurate statistics become available during execution and the routes in *QM* should be adapted based on this new information. Virtual and real concept drifts often occur together. We refer to such case as *hybrid concept drift* [27]. From a practical point, a concept drift (real, virtual, or both) gives a good indication that the current *QM* solution needs to be adapted. A concept drift implicitly indicates that either the data values, their frequencies or execution statistics have changed. Thus the predictions made by the current *QM* solution become less accurate as the time passes, e.g., data may be assigned to “wrong” subsets and less efficient execution plans may be used for processing of those data tuples. Hence, detecting concept drifts can serve as a good signal indicating a possible need to adapt.

Multi-route adaptivity is a more complex problem compared to a single plan adaptivity and brings several new challenges. First, we must continuously find and deploy the best execution solution where multiple plans are used concurrently. The majority of current adaptive solutions [3] are inapplicable here, as these methods are designed to support only a single plan. Second, *QM* employs a classifier as a component of query processing infrastructure. Therefore the classification cost must be taken into account by the *QM* optimizer. Furthermore, a *QM* may need to be adapted not only when statistics change, but also when data values change (even if statistics stay the same), because such change has a direct impact on the classifier accuracy⁴ and the overall performance of *QM* solution. Therefore, monitoring data values is as important as monitoring statistics. Finally, the physical execution of *QM* adaptation itself must be inexpensive to make it practical for dynamic environments where query results must be produced in near-real time. In summary, the key challenges include: (1) how and when to determine that the current *QM* solution is no longer adequate, (2) how to determine the new “best” *QM* solution based on the new data values and the updated statistics, and (3) how to efficiently execute the physical migration from the current *QM* to a new *QM* solution while the query is being executed.

⁴The classifier is constructed based on data values.

1.4 Our Proposed Solution: ST-QM

We address the above-mentioned challenges by proposing a self-tuning framework for QM called $ST-QM$. The techniques presented in this work are discussed in the context of stream environments and multi-plan query processing, however, in principle, they can be applicable to other systems as well. In summary, the contributions of this paper are:

1. We abstract the adaptivity of a multi-plan solution QM as a *concept drift* problem. Our approach, based on monitoring and detection of concept drifts, can discard many insignificant adaptivity cases early, and thus minimize the adaptivity overhead.
2. We present algorithms to efficiently determine *virtual* and *real* concept drifts in QM used to determine if and how the execution strategy should be adapted.
3. The key feature of our adaptive method is that all *logical* transformations to the current execution solution are translated into a *single physical operation* – the change of the classifier, without effecting the rest of the execution infrastructure. This makes physical adaptivity extremely lightweight.
4. We thoroughly evaluate the $ST-QM$ approach through experiments. Our results show that $ST-QM$ is very effective in adapting to different kinds of concept drifts, its overhead is minimal, and the physical actuation of adaptivity has nearly negligible cost.

The rest of the paper is organized as follows. Section 2 gives the background on the core QM framework. Sections 3-6 describe the details of $ST-QM$ design. Section 7 presents our experimental evaluation of $ST-QM$ framework. Section 8 discusses the related work and we conclude in Section 9.

2. BACKGROUND: QUERY MESH

The QM framework consists of two primary components: *query mesh optimizer* and *query mesh executor* (Figure 3⁵). For a given query, the optimizer computes the QM solution offline using available statistics and samples of data (the training data). The training data accurately represents the distribution of the data expected to come in the near future – a common assumption in most database systems [30, 34] and prediction models in data mining and machine learning alike [16]. For streaming databases, relying on samples of data is unavoidable, since it is impossible to “see” all of the data a priori. For a QM solution generated by the optimizer, the classifier model is inferred and the routes are computed. For details of these steps, we refer the reader to [35].

The query executor takes the QM solution produced by the optimizer and instantiates the physical runtime infrastructure, which consists of two main elements: the *Online Classifier Operator* and the *Self-Routing Fabric (SRF)*. The online classifier at runtime classifies arriving data tuples and assigns the best routes for their processing. While many classification models could be plugged into QM , e.g., neural networks, naive bayes, etc., in our work we use a *decision tree (DT)* [27]. Decision trees are considered as very effective classifiers: (1) DT generation algorithms do not require additional information besides that already contained in the training data, i.e., they are non-parametric [32]. (2) Compared to neural networks or bayesian classifiers, DT s can be easily understood and interpreted by a human user [19].

⁵ A thick black arrow in Figure 3 illustrates an example of a QM -based query execution for a subset of data.

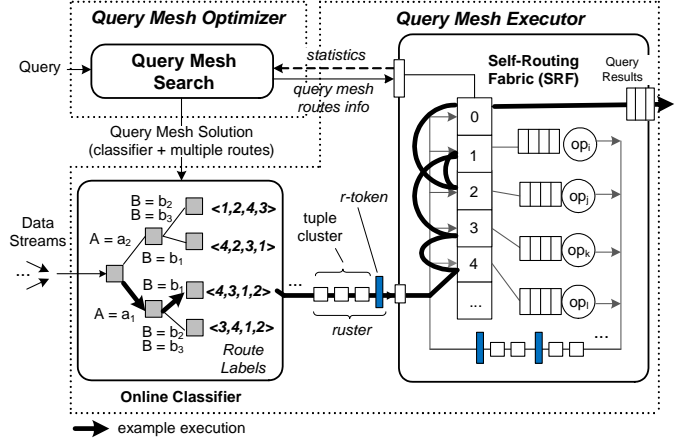


Figure 3: Core QM framework.

This can be useful when a DBA wants to analyze the QM performance and relate the classifier model to the actual input data. (3) DT s display similar and sometimes better accuracy compared to other learning techniques [10].

For efficient query processing, the QM -based executor uses an infrastructure, called the *Self-Routing Fabric (SRF)*, which implements query processing via multiple routes with near-zero route execution overhead. In contrast to current adaptive systems, SRF eliminates the expensive central data router, such as the Eddy operator [12, 33, 41]. Instead, route specifications are encoded in meta-data tuples, called *routing tokens* (or short r -tokens). R -tokens are then embedded inside data streams along with their data tuples by the *online classifier operator*. This allows de-centralized self-routing of data and eliminates the “backflow” bottleneck [12, 30, 33] present in the Eddy-based systems.

To keep memory and CPU overheads minimal, the tuples are assigned to an existing route in groups called “*routable clusters*” or short “*rusters*” rather than individual tuples. *Rusters* distinguish themselves from traditional batching, e.g., [12, 21], in that they are formed by probing the classifier. Hence, only the tuples that share the same best route get assigned to the same *ruster*. To enable de-centralized routing, routes in the r -tokens are specified in the form of an *operator stack* based on the design of SRF . The stack nodes represent the indexes of the operators in the SRF , e.g., the r -token $\langle 2,3,1,4 \rangle$ indicates that ‘2’ is the first operator in the route, ‘3’ is the next, etc. A *ruster* is always sent to the operator that is currently the top node in the routing stack. After an operator is done processing the *ruster*, the operator “pops” the top of the routing stack – its unique identifier in the r -token, and then puts the *ruster* into the next (now the top) operator’s input queue. When the operator stack is empty, the *ruster* tuples are forwarded to the global output queue reserved by index “0” and then to the application(s).

3. OVERVIEW OF SELF-TUNING QM

3.1 The Main Idea

The following is the problem we tackle in this paper:

Multi-Route AQP Problem: For a given query Q and its multi-plan solution QM computed at time t_i based on the representative dataset T and statistics H , continuously detect a concept drift when a new sample dataset T' and statistics H' become available at time $t_j > t_i$. If a concept drift has occurred, find a new solution QM' based on H' that results in the lowest execution cost for tuples in T' . If the

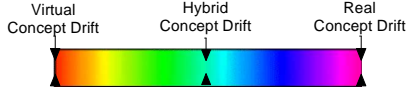


Figure 4: Concept drift spectrum.

$estimated\ cost(QM') < cost(QM)$, replace QM with QM' .

The goal of *Self-Tuning Query Mesh* framework (or short *ST-QM*) is to detect QM concept drifts and to adapt the current QM solution correspondingly to best suit the observed drift. Our approach is unique in that we view the problem of adaptive query processing (AQP) as a *concept drift* problem from machine learning [42]. This abstraction of the AQP gives several advantages to the adaptive system. First, if we discard an adaptivity case due to an absence or a presence of a small concept drift, it is likely not going to lead to a better QM solution, because we've discarded insignificant changes in the environment. If we do not discard a case, then there is a high chance that it is worthwhile to analyze further. In the end, there are fewer cases *ST-QM* has to analyze and the ones that do get analyzed further are all promising. Second, techniques from machine learning and data mining fields addressing the concept drift detection and analysis can be leveraged here to determine if adaptation is needed and how to best adapt to the observed drift.

3.2 Query Mesh Concept Drifts

Given the two kinds of concepts in QM (*virtual* and *real*) described in Section 1.2, the following three cases may occur:

Case 1: Virtual Concept Drift. This indicates that data values and/or their frequencies have changed, but the execution statistics of the new data subsets stay the same, thus making the previously computed routes still applicable. One example when such scenario may occur is when a better quality (i.e., more representative) training dataset is collected over time. In this case, the execution statistics of the subsets might not change significantly, yet the QM classifier can be further fine-tuned by integrating new data values. For example, a new DT sub-tree can be added or the nodes can be “pushed-up” or “down” for faster classification. Another example of this case (based on the application mentioned in Section 1.1) is when a stock exchange opens and starts streaming its data. The streaming data values from the recently opened stock exchange get combined with the streaming data from other previously streaming stock exchanges (e.g., from other regions). Here the new stock data values, e.g., symbols, location, etc., will appear in the data streams, yet the underlying distribution and the statistically similar data subsets are likely to stay unchanged.

Case 2: Real Concept Drift. This case means that the data values stay unchanged, but their execution statistics (e.g., selectivities or operator costs) begin to vary, thus requiring the execution routes to be adapted. This scenario tends to be less frequent, but may arise when the optimizer used a rough approximation of data subsets' statistics, and then more accurate statistics become available as a result of the query execution feedback. The updated statistics enable the “tune-up” of the execution routes. Using the financial application example, this case may happen when a sole stock market is being monitored. Here, the data values, e.g., stocks being sold on this stock exchange, become available as soon as it opens and are unlikely to change significantly during the day. Yet, the statistics for the new data might not be very accurate at the beginning of the execution. However, the longer the query runs, the more accurate estimations

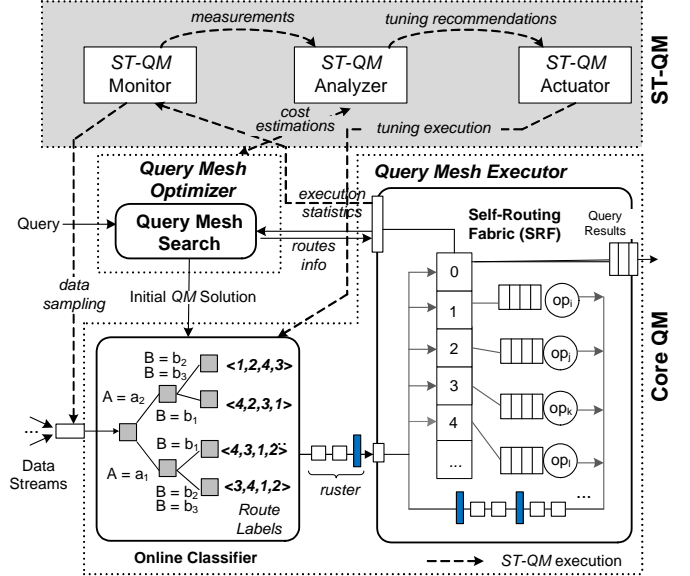


Figure 5: Self-tuning QM framework.

can be made. Another example when this case may occur is when better routes are found through *route exploration* described in Section 4.2.

Case 3: Both Virtual and Real Concept Drifts. We refer to this case – the hybrid concept drift, and it happens when both the data distribution and the execution statistics change, consequently leading to alterations in the execution routes and the classifier. Using financial application example, this case may happen when during the after-market trade hours, important news become public, which may have a significant impact on the stock prices of some industries. Since not all stocks participate in the after-hours trading, the data distribution changes after the markets close. Furthermore, the real-life news may impact the prices of only certain types of stocks. In this case, both the data distribution and the execution statistics may change significantly, thus requiring both the classifier and the set of execution routes in QM to be adapted.

The three cases described above are not independent. Virtual and real concept drifts are the special cases of the hybrid concept drift. The three cases compose a comprehensive “spectrum” of changes that may occur in a system (Fig. 4): specifically, a change in data values and their frequencies, a change in execution statistics and a change in both.

3.3 ST-QM Architecture

ST-QM adds three new components to the core QM framework: *ST-QM Monitor*, *ST-QM Analyzer* and *ST-QM Actuator* (shaded grey in Figure 5). We have designed *ST-QM* to be highly modular, enabling adaptivity functionality to be turned on/off with complete transparency to the core QM framework (bottom of Figure 5). The architecture is easily extensible: new algorithms and metrics can be added without much disturbance to the rest of the system. We describe the functionality of each *ST-QM* component next.

ST-QM Monitor continuously samples data and execution statistics that will be used to determine if a concept drift has occurred. Monitored parameters include data values, their frequencies, and the operators' costs and selectivities. Our monitoring approach is comparable to that of the existing systems, e.g., [36, 37] with a few distinct characteristics (see Section 4). Given the measurements from the *ST-QM Mon-*

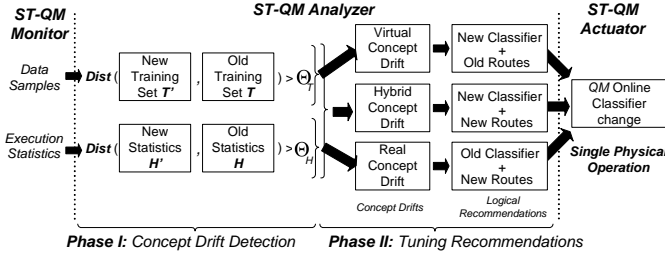


Figure 6: ST-QM process flow.

itor, the *ST-QM Analyzer* determines if a concept drift has actually occurred, how well the current *QM* solution is meeting its estimated costs and performance goals, and what (if anything) is going wrong. Based on the analysis, the *ST-QM Analyzer* makes recommendations if and how the *QM* solution should be adapted. *ST-QM Actuator* takes these recommendations and physically adapts the *QM* solution. Figure 6 graphically depicts the flow of the entire process.

Monitoring, analysis, and actuation of adaptivity in *ST-QM* add overhead to query processing. Thus, to minimize the overhead, the following system requirements must be met: (1) monitoring must be light-weight, and only if significant changes are detected should the more expensive analysis process be invoked, (2) adaptivity candidates corresponding to insignificant changes in the environment must be discarded early, e.g., during monitoring or in the early analysis, without invoking the optimizer, (3) the decision to adapt should be made only if significant improvement in the performance is expected, and (4) the physical execution of adaptivity must be fast and inexpensive to be done online.

4. ST-QM MONITOR

Monitoring aims to identify if the current *QM* solution is no longer consistent with the current data and its characteristics. What sets apart our monitoring goals from the existing systems, e.g., [36, 37] is that: (1) we monitor not only the change in data distributions and execution statistics but also in the data values, and (2) we focus not only on assuring the overall representativeness of a sample but also on ensuring that new, i.e., the never seen before data values are not gone undetected. *ST-QM Monitor* employs two complimentary techniques, namely the input data and the execution statistics monitoring.

4.1 Input Data Monitoring

For data monitoring, we sample the arriving to the server data to collect a *new training dataset*. This new dataset is analyzed to see if changes in the data values and their distributions have occurred. Monitoring data values (in addition to the distributions and execution statistics) has the advantage that the adaptive system can exploit this extra information, which is collected inexpensively, to minimize the overhead of the more expensive execution statistics monitoring, e.g., when profiling operators or exploring for new execution routes alternatives. Often changes in data values implicitly indicate changes in data distributions. Consequently, this leads to a possible change in the execution statistics, since virtual and real concept drifts frequently occur together. If the system detects a change in data values, it may then employ a more expensive and detailed execution statistics monitoring to see if the routes may need to be adapted. Simple random and systematic sampling techniques can be used here for data sampling [34]. However they can miss potentially “important” training data trying

to uniformly cover the entire sampling window. Thus, we’ve designed the following techniques:

Classifier-driven sampling. This type of sampling is based on the “importance” of tuple attributes. In *QM*, some attributes are naturally more important than others, e.g., when the decision tree (*DT*) classifier is constructed, a split criterion is used to select the best splitting attribute at each node. *Information gain*, *entropy*, or *gini index* measures of impurity can be used for this purpose [10, 27]. Comparing the impurity value of a split attribute in the *DT* classifier for the old and the new samples of data can be a good indicator if the data distribution possibly changed. If the differences between the old and the new impurity measures for the same attributes are significantly different, then the new sample is considered “interesting to analyze further” and thus is not discarded. The decision of how many impurity measures to compute, i.e., for how many *DT* nodes, and their relative importance in the overall sampling is parameterized.

Route-driven sampling. This sampling method resembles a *biased* sampling approach. It is guided by the *QM* execution routes and the expected percentage (% *expected*) of tuples to be processed by those routes. Here, each tuple from the new sample first probes the current *QM* classifier (Stage 1). After probing, tuple groups are formed, with each group being assigned to a particular route. If the difference between the actual and the expected route assignment fraction of tuples is less than the system-set threshold, then a random selection of *k* members from those groups is performed. If the difference is greater than the threshold, these tuple groups get a high “priority”, because they contain the different (from before) data and $(k + k * (\% \text{ expected} - \% \text{ actual}))$ tuples are sampled from each such tuple group (Stage 2). The sub-sample size here is directly proportional to the observed frequency difference.

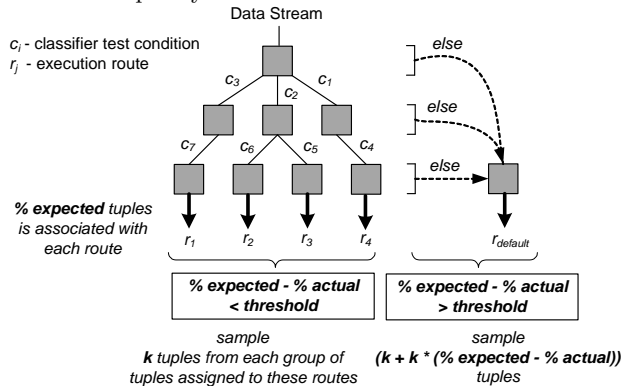


Figure 7: Route-driven sampling.

The motivation behind this method is the following: if the same fraction of tuples were assigned to the same routes, then the data distribution is unchanged. However, if the difference is significant, e.g., in the case of a special route called the *default route*⁶, then more tuples should be sampled, as this could be an indication of a virtual concept drift and possibly the classifier may need to be updated. Since this type of sampling is “biased” towards collecting previously unseen data, the new sample is treated as a complement to the old training data set and the two sets are combined (unioned)

⁶A *default route* $r_{default}$ (illustrated in Figure 7) is an execution route based on the overall statistics of the data. It is used by the data that has similar statistics as the overall data statistics, as well as by the “new” data with properties (values and frequencies) that may have not been present when the *QM* was originally computed.

to improve the overall quality of the training data and the resulting execution model.

4.2 Execution Statistics Monitoring

The statistics collected during execution are used to detect the presence of the *real* concept drift. Execution statistics monitoring consists of two complementary sub-parts: exploitation and exploration statistics monitoring.

Exploitation Statistics. Exploitation statistics monitoring tracks the selectivities and costs of operators when using the established execution routes. We instrument query operators to collect three types of statistics: (1) *independent* selectivities, (2) *correlated* selectivities and (3) operator costs (measured by wall-clock time). To compute independent selectivities, a *statistics bit* is turned on in the *r-token* of a randomly selected *ruster*, thus making it a special-purpose (*statistics*) *ruster*. If a tuple from a *statistics ruster* does not satisfy operator predicate, the tuple is not physically discarded, but rather marked as a “ghost” to be able to compute independent selectivities for other operators en-route. For correlated selectivities, the selectivity is computed using only the regular (“non-ghost”) tuples in the *statistics ruster*, i.e., the tuples that have not been discarded by any previous operators in the route. All three types of statistics are collected for each individual route by the operators.

Exploration Statistics. The motivation for the exploration statistics lies in the fact that the only way to know precise costs of alternative strategies is through competitive execution [33]. For this purpose, we use *exploration rusters* – a small fraction of the input *rusters* that are randomly selected and assigned different from their current “best” routes, while monitoring the statistics along these routes. The exploration routes are determined by the *exploration policy*. *ST-QM* employs two exploration policies: (i) *random existing route*, where chosen *rusters* are sent on another randomly picked *existing* route; (ii) *random new route*, where *rusters* are sent on a randomly generated and currently *non-existing* in the *QM* solution route.

Devoting resources to exploration to obtain information about thus-unknown costs may help in finding better routes, but in the short term it detracts from exploitation – producing results with the current best routes. This is a classic exploration versus exploitation dilemma [3]. To address this problem, *ST-QM* adaptively determines the number of *rusters* used for exploration. The total number of exploration *rusters* (*TER*) depends on the value of a distance measure (described in Section 5) and is computed as: $TER = DER + (\alpha * \mu)$, where *DER* is the default number of exploration *rusters*, μ is the value of the distance measure and α is the fraction of *rusters* per distance unit. The larger the distance, the larger the number of *rusters* used for exploration. Exploration may also be applied selectively to only some *rusters*, to put more focus on exploring routes for certain subsets of data.

5. ST-QM ANALYZER

The *ST-QM Analyzer* takes the data samples and the statistics from the *ST-QM Monitor* and based on them determines if any concept drifts have occurred. It then gives tuning recommendations based on the analysis. The execution consists of two phases: (1) *concept drift detection*, and (2) *tuning recommendations*.

5.1 Phase I: Concept Drift Detection

5.1.1 Virtual Concept Drift Detection

The concept drift detection algorithm *CD-Detect* (in Figure 8) maps the problem of virtual concept drift detection to the problem of comparing two data samples T and T' .

Algorithm **CD-Detect**(T old training set, T' new tuple sample, H old statistics, H' new statistics)

```

1:  $dist_{data} = ComputeDataDistance(T, T')$ 
2:  $dist_{routes} = ComputeRoutesDistance(H, H')$ 
3: if ( $dist_{data} > \theta_{data}$ ) and ( $dist_{routes} > \theta_{routes}$ ) then
4:   return Hybrid Concept Drift
5: else if ( $dist_{data} > \theta_{data}$ ) then
6:   return Virtual Concept Drift
7: else if ( $dist_{routes} > \theta_{routes}$ ) then
8:   return Real Concept Drift
9: end if
```

Figure 8: QM concept drift detection.

The algorithm requires a distance measure $dist_{data}$ which quantifies the difference between data samples T and T' . If $dist_{data} > \theta_{data}$, where θ_{data} is the adjustable distance threshold, virtual concept drift is reported. The key to the change detection is the intelligent choice of the distance function to compute $dist_{data}$, which must accurately quantify a data change that may impact the current *QM*. The choice for the threshold θ_{data} value defines the balance between the sensitivity and the robustness of the detection. The smaller θ_{data} , the more likely we are to detect small changes in the data, but the larger is the risk of a false positive.

One common approach to measuring data differences is to first estimate the probability distributions of the data, and then compute the distance, such as the *Kullback-Leibler Divergence* or the *Jensen-Shannon Divergence* [9], between the estimated distributions. However, this approach is computationally impractical for large and high dimensional data. The problem becomes even more challenging in streaming data environments, as the high speed makes it difficult for such expensive algorithms to keep up with the data [45]. To tackle this issue, we have designed two efficient methods:

Misclassification Rate. Misclassification rate or error rate \mathcal{E} , described as $\mathcal{E} = (1 - \mathcal{A})$ where \mathcal{A} is the classifier accuracy, represents the fraction of total cases “misclassified” by the current *QM* classifier for the new data sample. The main idea here is to assign the execution routes to the tuples from the new data sample. Then the tuples from the new sample probe the current classifier, and the classifier’s misclassification rate, e.g., mean absolute error, is computed. The reason we assign the new sample tuples to the existing routes (even though we could possibly find better plans for their processing) is because we are checking for virtual concept drift with respect to the current target (i.e., the current set of execution routes).

Signature-Based Method. This method regards the decision tree classifier as a summarization of the distribution of data. Each leaf node contains a route label and the fraction of tuples expected to be processed by that route. Together, all the leaf nodes can be thought of forming a special “histogram” of route assignment frequencies. Then after probing the classifier, a signature is assigned to each data sample that depicts the route assignments frequencies. This way we evaluate data distribution changes by comparing these signatures. This method is extremely efficient, since all it requires is a quick probe of the classifier.

5.1.2 Real Concept Drift Detection

Real concept drift occurs when execution statistics change significantly, consequently implying that the execution routes

Algorithm *TR- Produce*(*CD* detected concept drift, *T'* new training dataset, *H'* latest execution statistics)

```

1: QM = current query mesh solution used in execution
   /* VIRTUAL CONCEPT DRIFT RECOMMENDATIONS */
2: if (CD.Type == Virtual Concept Drift) then
3:   Compute new classifier C' based on the training set T'
4:   Let QM' = new query mesh solution with classifier C'
5:   if (cost(QM') < cost(QM)) then
6:     Recommend New Classifier C'
7:   end if
   /* REAL CONCEPT DRIFT RECOMMENDATIONS */
8: else if (CD.Type == Real Concept Drift) then
9:   Compute new set of routes R'
10:  Let C' = current classifier QM.C
11:  Update the target level of the classifier C' with routes R'
12:  if (the target R' requires modification of classifier C') then
13:    Compute a new classifier C'' based on the new target R'
14:    Let QM' = new query mesh solution with classifier C''
15:    if (cost(QM') < cost(QM)) then
16:      Recommend New Classifier C'' and New Routes R'
17:    else
18:      Let QM' = new query mesh with routes R'
19:    end if
20:    if (cost(QM') < cost(QM)) then
21:      Recommend New Routes R'
22:    end if
23:  end if
   /* HYBRID CONCEPT DRIFT RECOMMENDATIONS */
24: else if (CD.Type == Hybrid Concept Drift) then
25:   Compute new QM' solution based on the training set T' and
   the new statistics H'
26:   if (cost(QM') < cost(QM)) then
27:     Let C' = classifier QM'.C
28:     Let R' = set of routes QM'.R
29:     Recommend New Classifier C' and New Routes R'
30:   end if
31: end if

```

Figure 9: QM tuning recommendations.

may need to be altered as well. Given the updated execution statistics, a new set of routes is computed and compared to the old set of routes. The goal here is *not* to estimate whether the *QM* solution with the new set of routes would necessarily be “better” (remember, the cost of a *QM* solution depends on the combination of both the classifier and the routes’ costs), but rather that the new routes are different (see Algorithm in Figure 8). Using such simple route difference approach allows *ST-QM* to minimize its overhead: since route computation is a fraction of the entire *QM* recomputation [35]. Next we discuss several possible choices for the route distance measure $dist_{routes}$.

Number of Affected Routes. This distance measure counts the number of routes that are different when comparing the old and the new sets of routes. Let R denote the old set of routes, and R' be the new set of routes. Then $dist_{routes} = |R_{diff}| = |R' - R|$, where $\forall r \in R_{diff}, r \in R' \text{ and } r \notin R$. For example, if a route r has a different operator ordering or if a new route r exists in the new set as a result of exploration – all these changes contribute to the route distance measure. If a more fine-grained measure is needed, the approach can be extended to consider the count of the operators with significantly different selectivities and execution costs.

Route Edit Distance. This distance measure is based on the *edit distance* approach [46]. Here, the old and the new routes are mapped respectfully to the same data subsets, meaning these routes were considered as the best execution strategies for processing of the same data subset at different times. Routes represent operator sequences and can be described by the strings composed of operator identifiers. The edit distance between any two routes is then the number of operations required to transform one of them into the other.

The examples of edit distances that can be used here include *Hamming distance*, *Levenshtein distance*, etc.

5.2 Phase II: Tuning Recommendations

After *QM* concept drifts have been detected, the *ST-QM Analyzer* determines how to address them. In response to the concept drifts, *ST-QM Analyzer* may do the following: (1) ignore the concept drifts, if they are small or the benefits of adapting the current *QM* is not expected to give much performance improvement; (2) incrementally tune a sub-part of the *QM* solution, e.g., a classifier sub-tree or a route; (3) compute a new *QM* solution based on the updated statistics and consider to replace the current *QM* solution.

5.2.1 Recommendation Algorithm

Figure 9 illustrates the pseudo-code for the *TR- Produce* algorithm⁷ employed by the *ST-QM Analyzer* to produce tuning recommendations. Similar to many adaptive solutions, *ST-QM* uses the *QM optimizer cost model* [35] to compare the current execution to what was originally expected or what is estimated to be possible [3]. The recommendation algorithm has the following cases:

Case 1: Virtual Concept Drift Recommendation. If a virtual concept drift is detected, first a new classifier C' for the new training set T' is computed. Then the cost of the new query mesh (with the new classifier C') QM' is determined and compared to the cost of the current QM . If the new QM' has a smaller cost, the new classifier C' is recommended.

Case 2: Real Concept Drift Recommendation. If a real concept drift has been detected, the target level (i.e., the routes) in the QM classifier are updated. If this update does not require the modification of the rest of the classifier, and if the QM' solution with new routes R' has a smaller cost than the current QM solution, then the new routes R' are recommended. If the classifier needs to be adjusted (e.g., if some routes are now shared by several groups or if some routes are removed), this case is then handled as a hybrid concept drift.

Case 3: Hybrid Concept Drift Recommendation. If a hybrid concept drift has been detected, a new QM solution with the new classifier and the new set of routes is computed, its cost is estimated and compared to the current QM solution’s cost. If the newly computed QM' has a smaller cost, then both the new classifier C' and the new routes R' are recommended⁸.

To evaluate the benefit of a recommendation, *ST-QM Analyzer* uses the metric, called improvement I :

$$I(QM, QM', T', H') = 100\% * \left(1 - \frac{cost(QM', T', H')}{cost(QM, T', H')} \right)$$

where QM is the initial and QM' the recommended solution, and $cost(QM', T', H')$ is the expected cost of evaluating a query under the QM' solution based on the training data set T' and the statistics H' . The *ST-QM Analyzer* computes the expected improvement value, and if the value is deemed as substantial, only then the recommendation is outputted.

6. ST-QM ACTUATOR

6.1 Physical Execution of Adaptivity

⁷“*TR*” is the abbreviation for “Tuning Recommendations”.

⁸If either the classifier or the new set of routes have been computed in the earlier stages of analysis, e.g., during concept drift detection, they are cached and not recomputed in this phase.

ST-QM Actuator physically adapts the *QM* solution in the execution framework based on the recommendations received from the *ST-QM Analyzer*. As described in Section 5.2, *ST-QM Actuator* may receive the following three kinds of recommendations:

- **R₁.** *New Classifier + Old Routes*
- **R₂.** *Old Classifier + New Routes*
- **R₃.** *New Classifier + New Routes*

The key characteristic of the *ST-QM* is that all three recommendations get translated into a *single physical operation* in the execution infrastructure, namely the change of the classifier in the online classifier operator. To accomplish this, only a simple pointer re-assignment to the new classifier object is needed (Figure 10). This single step is the actual execution of *QM* adaptivity and the implementation is trivial. What makes this possible is the architecture of *QM* framework. Although routes (i.e., query plans) are pre-computed, their topology is not physically constructed. Instead the *Self-Routing Fabric (SRF)* infrastructure provides distributed routing (i.e., forwarding of tuples to the operators in the plan) based on the plan specifications assigned by the classifier (Section 2). This physical separation between the component that determines which plans should be used for execution and the component that actually executes the plans based on specifications, makes the *QM* adaptivity so light-weight. To change the execution strategy, all the system needs to do is modify the specification of the plans (in the classifier).

Figure 10 illustrates an example of physical execution of *QM* adaptivity. The old classifier, marked by lighter grey, is replaced by the new classifier, and the *rusters* with new routes are sent into the self-routing fabric instantaneously. The attractiveness of our design is that we can easily switch

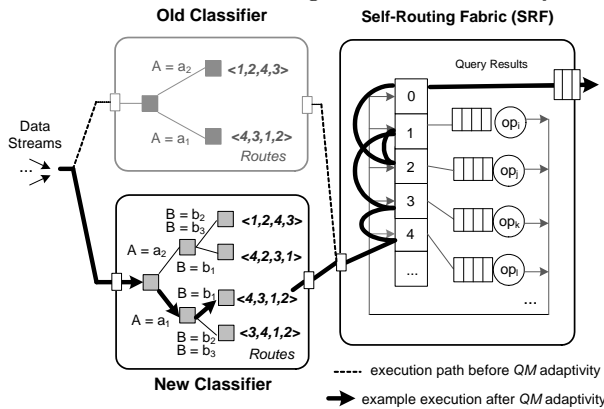


Figure 10: Physical execution of *QM* adaptivity.

between different multi-plan solutions. If the desired performance improvements after adaptivity are not gained, *ST-QM* can easily switch back to the previous *QM* solution. The architecture makes such behavior very flexible.

6.2 State Management and Adaptivity

One of the key questions that must be answered in adaptive systems is the problem of state management for stateful operators. We consider select-project-join (SPJ) queries. For joins, we employ *one-way-join-probe (OJP)* operators [35], similar in spirit to *SteMs* [43], which correspond to a half of a traditional join operator. There is one *OJP* associated with each stream attribute that participates in the join. The *OJP* keeps track of the window of attribute values that

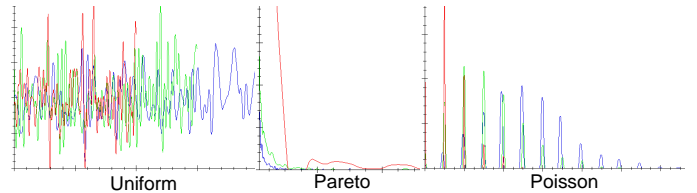


Figure 11: Experimental distributions: different colors illustrate how the distribution changes when the parameter values vary as described in Table 2.

have arrived on the stream and allow subsequent tuples from the other streams to probe these stored attribute values to search for a match. In the case of the join operator, the order in which tuples probe the *OJP* is irrelevant as long as each tuple passes through each *OJP* exactly once. This holds from the associativity and the commutativity property of the join operator [21, 43]. Without adaptive functionality, the core *QM* framework already supports concurrent plans with different operator ordering. Hence, adding adaptivity does not require any additional support. We plan, however, to investigate new state management techniques in our future work, to extend support to other types of queries.

7. EXPERIMENTAL EVALUATION

We now describe our experimental evaluation of *ST-QM* implemented inside Java-based continuous query engine called CAPE [13]. To evaluate *ST-QM*'s design, we compare its relative performance against competitor systems, namely the non-adaptive *QM* (described in Section 2) and the adaptive “plan-less” Eddies [33] with CBR-based routing policy [30] – the closest solutions to *ST-QM*. To ensure even comparison, all three systems were implemented in CAPE, and their implementation used as much of the same codebase and data structures as possible. We also demonstrate the effectiveness of *ST-QM* by measuring its overheads and the benefits of its concept drift abstraction approach.

7.1 Experimental Setup

All our experiments are run on a machine with Java 1.6.0.0 runtime, Windows Vista with Intel(R) Core(TM) Duo CPU @1.86GHz processor and 2GB of RAM. Our experiments use *N*-way join queries which join incoming $S_1 \dots S_N$ streams. The specific query we use is an equi-join of 10 streams, i.e., $S_0 \bowtie S_1 \dots S_9 \bowtie S_{10}$. *N*-way join queries are one of the core queries in database systems used to discover relationships across data or events coming from different data sources.

We use synthetic data sources for our experiments, similar to [4, 30, 43]. Using synthetic data allows us to manage data properties that are hard to control in real-life data. We employ several known data distributions to determine the skew of the data. Specifically, we use well-known distributions: *Uniform*, *Pareto* and *Poisson* [7] (see Figure 11). These distributions model many real-life phenomena (see Table 1 for examples). The default data properties, system parameters and distribution parameters used in the experiments are shown in Table 1 and Table 2.

Each stream's schema is composed of five attributes and a timestamp. For every join attribute column, integer-based values are generated using one of the above-mentioned distributions. The values of other attributes are correlated to the join attribute values, e.g., in a stream $S(col_1, col_2, col_3, col_4, col_5)$, if col_1 is a join attribute, the values of $col_2 \dots col_5$ are correlated to the values in the join attribute column according to the specified generator correlation parameters. The default values are 50%, 30%, 15%, 5%. To make this

Table 1: Default experimental parameters.

Parameter	Value	Description
<i>Ruster size</i>	100 tuples	Average <i>ruster</i> size
<i>Sample size</i>	100 tuples	Average sample size per stream
<i>Data monitoring</i>	Route-driven sampling	Data monitoring method. $k = T / R $, $\theta_{diff} = 0.2$
<i>Execution monitoring</i>	Exploitation statistics	No exploration is used
<i>dist_{data}</i>	Signature-based	Virtual concept drift detection method. $\theta_{data} = 0.1$
<i>dist_{routes}</i>	Number of affected routes	Real concept drift detection method. $\theta_{routes} = 0.2$
<i>Impr. I</i>	$I = 0.1$	Improvement parameter

Data Distributions

Name	Parameters	Application Examples
<i>Uniform</i>	$\alpha \in \{\dots, \beta-1, \beta\}$ $\beta \in \{\alpha, \alpha+1, \dots\}$ $X \in \{\alpha, \dots, \beta-1, \beta\}$	• Long-term patterns of data
<i>Pareto</i>	$0 < \alpha < \infty$ $0 < \beta < \infty$ $\alpha \leq X < \infty$	• Animal migration • Word frequencies
<i>Poisson</i>	$0 < \lambda < \infty$ $X \in \{0, 1, \dots\}$	• Service times in a system • # of phone calls at a call center per minute • # of times a web server is accessed per minute

more concrete, consider an example: value 100 is generated in the join attribute column based on the chosen distribution, then in another attribute column, 50% of the time value 99 will appear next to 100, 30% value 98, and so on⁹. For other attributes in the stream, the values are generated similarly. We decided against generating random values in the non-join attribute columns, to avoid short and wide decision tree classifiers (e.g., a decision tree with height 1 and the test conditions based on all possible random values). The explanation for this is the following: if an attribute contains a lot of unique random values, the entropy value for this attribute column approaches 0. Since many splitting criteria in *DT* construction algorithms are *entropy*-based [27], the attribute with the most distinct values gets picked first, and the algorithm stops right there, thus resulting in a short and wide decision tree.

To simulate dynamic changes, the generation of data was managed as follows: the data generator starts with a data distribution and its initial distribution parameters; over time, the distribution parameters values are varied, e.g., for Poisson distribution, the transition: $(\lambda = 1) \rightarrow (\lambda = 3) \rightarrow (\lambda = 5)$ (see Table 2), means that the initial distribution parameter value was 1, after some time it was changed to 3, and then to 5. This process is repeated continuously for infinite data streams. The values of distribution parameters are changed every 10K tuples across all streams.

The execution of *ST-QM* in CAPE [13] is split into two execution threads. The monitoring and the adaptivity actuation are interleaved with the query execution on one thread. The analysis of *ST-QM* (i.e., concept drift detection, optimizer calls and generation of tuning recommendations) is executed on another thread. The analysis and the optimizer search can sometimes be extensive [35], thus blocking the query executor from processing the arriving data tuples, while the system is being analyzed by adaptive component, is not practical. Hence, we separated *ST-QM* analysis into a separate thread, to prevent blocking of the query executor and to ensure that results are produced at all times.

7.2 Results and Analysis

⁹Values ‘99’ and ‘98’ were picked arbitrarily here to convey the example.

Table 2: Distribution statistics and parameters.

Uniform ($\alpha = 0, \beta = 100$): <i>min</i> : 0.0, <i>max</i> : 100.0, <i>med</i> : 49.0, <i>mean</i> : 49.7, <i>ave.dev</i> : 25.2, <i>st.dev</i> : 29.14, <i>var</i> : 849.18, <i>skew</i> : 0.05, <i>kurt</i> : -1.18. <i>Distr. trans</i> : $(\alpha=0, \beta=100) \rightarrow (\alpha=0, \beta=150) \rightarrow (\alpha=0, \beta=200) \dots$
Pareto ($\alpha = 1, \beta = 1$): <i>min</i> : 10.0, <i>max</i> : 6833.0, <i>med</i> : 19.0, <i>mean</i> : 73.56, <i>ave.dev</i> : 86.22, <i>st.dev</i> : 341.25, <i>var</i> : 116455.33, <i>skew</i> : 14.26, <i>kurt</i> : 240.2 <i>Distr. transitions</i> : $(\alpha=1, \beta=1) \rightarrow (\alpha=1, \beta=1.5) \rightarrow (\alpha=1, \beta=2) \dots$
Poisson ($\lambda = 1$): <i>min</i> : 0.0, <i>max</i> : 60.0, <i>med</i> : 10.0, <i>mean</i> : 10.0, <i>ave.dev</i> : 7.2, <i>st.dev</i> : 9.8, <i>var</i> : 97.59, <i>skew</i> : 0.96, <i>kurt</i> : 0.88 <i>Distribution transitions</i> : $(\lambda = 1) \rightarrow (\lambda = 3) \rightarrow (\lambda = 5) \dots$

7.2.1 Comparison Against Alternative Systems

In this experiment, we compare *ST-QM* design against the closest competitors, specifically the non-adaptive *QM* execution and the Eddy-based system with CBR-based routing [30]. The main difference between the implementations is that the non-adaptive *QM* evaluates the query using the same classifier and routes for the duration of the entire query execution. If data characteristics change, and the classifier does not have a sub-tree for the new data values, the “default plan” ($r_{default}$) is used for processing of that data. $r_{default}$ plan is based on the overall statistics of the data and is computed by the optimizer prior to query execution, just like in traditional query optimization. CBR-based execution is done in the context of Eddies. Eddy operator continuously profiles operators and identifies “classifier attributes” to partition the data into tuple classes that may be routed differently [30]. We execute Eddy with CBR routing in two modes: (i) with batching and (ii) without batching [12]. The batch size is set to 100, which is similar to *ruster* max size parameter in *ST-QM* (see Table 1), and is designed to reduce execution overhead.

We ran the query processor for 25 minutes several times, employing these different execution strategies, and show the results, averaged over all those runs. Figure 12(a) compares the average output rate, the average execution time per tuple is presented in Figure 12(b), and the run-time execution overheads present in these systems are in Figure 12(c)¹⁰.

From Figure 12(a), we can observe that for Uniform distribution, on average, *ST-QM* has 39% higher output rate than CBR without any batching, 24% higher than CBR with batching, and 6% lower than non-adaptive *QM*. In Uniform distribution, most of the time, the streams tend to have a single route. Occasionally, due to sampling, we have noticed two routes per stream in *ST-QM*. However, even with changes in the environment, the routes based on average statistics of the “old” data tend to be the same best routes for the “new” data. This explains the close output rate of *ST-QM* compared to non-adaptive *QM* for Uniform distribution. For Poisson distribution, *ST-QM* on average has 13% higher output rate than CBR without batching, 0-0.5% smaller rate than CBR with batching, and 43% higher output rate than non-adaptive *QM*. Here the simple batching of Eddies incidentally plays out very well, thus resulting in an average performance of *ST-QM* and Eddies being really close. For Pareto distribution, we observe that *ST-QM* on average has 27% higher output rate than CBR without batching, 18% higher than CBR with batching and 44% higher output rate than non-adaptive *QM*. The average execution time per tuple (in Figure 12(b)) follows a similar trend.

For Pareto and Poisson distributions, when a concept drift

¹⁰“QM” in the charts refers to the non-adaptive *QM* execution.

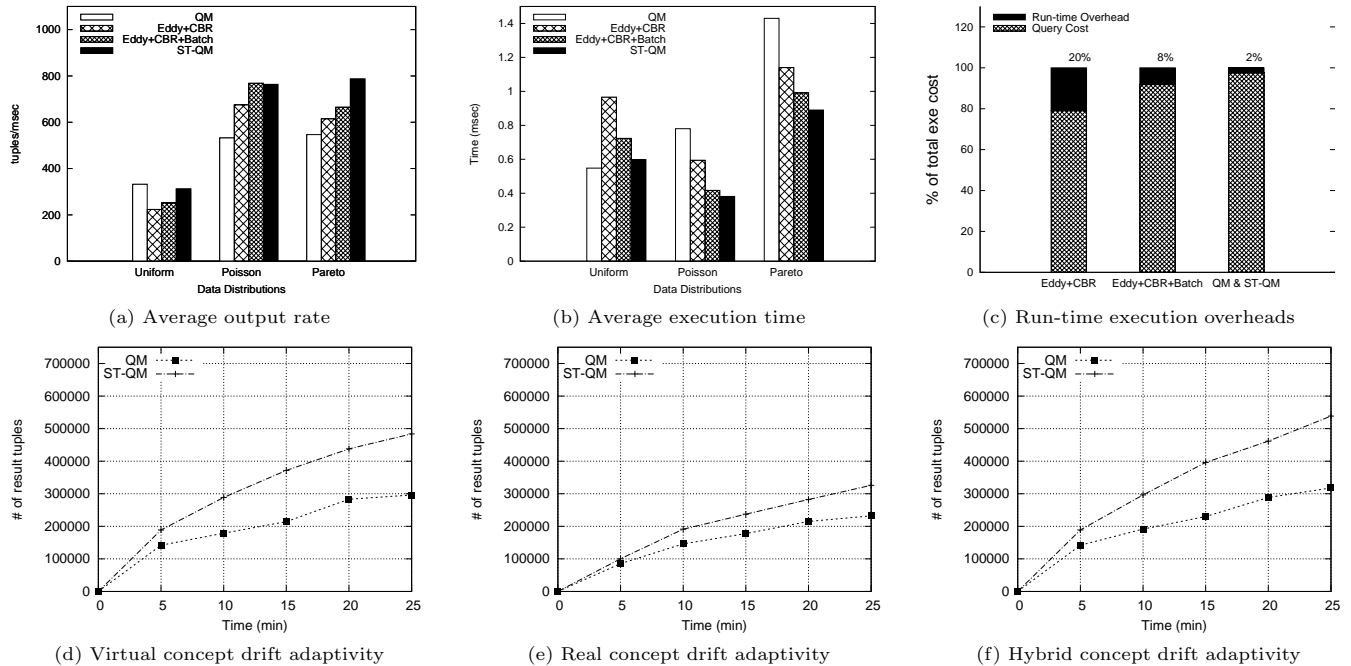


Figure 12: Experimental results.

occurs, and most of the data gets processed by the default routes in non-adaptive *QM* system, this results in poor execution strategy, since the data properties have changed and the execution could be improved by determining the new data subsets and customizing the routes for them, as is done in *ST-QM*. CBR, on the other hand, suffers from continuous re-optimization and re-learning overheads (the relative overhead is depicted in black in Figure 12(c)). Implemented in the context of Eddies, CBR continuously experiences the “backflow” overhead, where tuples get continuously routed back to the Eddy operator that has to re-examine the tuples and forward them to the next operator for processing. The overhead is $O(n+1)$ time, where n equals the number of operators and 1 accounts for the first time a tuple from an input stream gets processed. Without any batching, Eddy processing with CBR algorithm amounted to nearly 20% of the total execution cost.

Batching attempts to reduce Eddy overhead. However, batching in Eddy [12] is still very naive: every b tuples, i.e., a continuous chunk of tuples that happened to arrive together in time are batched and routed together. Without batching, the Eddy “backflow” overhead per workload of tuples W is $O((n+1)*|W|)$. With batching, the overhead gets reduced by the batch size b , resulting in the total overhead $O((n+1)*|W|/b)$. In practice, the batches might be smaller, depending on the arrival rates of the tuples. In *QM*, on the other hand, tuples are grouped together into the same *ruster* based on the classification, i.e., the data values and the similarity of statistics, and are thus guaranteed to share the same best route.

Eddies employing CBR also experience continuous overhead of re-computing classifier attributes based on runtime information, even though the best classifier attribute for an operator does not change very often [30]. These overheads limit the benefit that can be obtained from a better adaptive policy in Eddy. Static *QM* and *ST-QM* also have a small runtime overhead, namely the probing of the online classi-

fier to determine the execution plan for arriving data. The classification overhead, however, was measured to be very small, only 2% of the query execution cost (Figure 12(c)).

7.2.2 Adaptivity to Concept Drifts

This experiment evaluates how *ST-QM* adapts to different concept drifts. We use non-adaptive *QM* execution as a base case to compare *ST-QM* results.

A *virtual concept drift* means that the data values change, but the distributions of the new content groups stay the same, thus affecting the classifier component but not the target routes. To simulate only virtual concept drifts, we generate data using one of the experimental distributions, and then over time replace the data values with different values, while maintaining the same distribution of data values. Thus, the content of data changes, but their frequencies stay the same. A real life example when this scenario may happen is the variation between the number of times a web server is accessed per minute. Depending on the day (e.g., work day or weekend), the hour (e.g., morning or evening) the values may be different, but the overall distribution typically tends to follow Poisson distribution [24]. Due to space constraints, we only show the results for the Poisson distribution here, but similar trends have been observed for other distributions as well. Figure 12(d) shows the results for *ST-QM* compared to non-adaptive *QM*. *ST-QM* gives, on average, between 24% to 38% improvement over static *QM* execution.

In *real concept drift*, the data values stays constant, but the execution routes change. Real concept drift may occur due to changes in either the selectivities, the costs of query operators, or both. Typically, a change in selectivity indicates a change in the data distribution, and thus most likely a hybrid concept drift. Therefore, to simulate only real concept drifts, we vary the time it takes an operator to process a tuple over time (with non-changing data values) and report the effects on *ST-QM*’s performance. To motivate the

exploration of the space of higher operator costs, consider the following example: [5] describes multilingual query operators, e.g., *LexEQUAL* and *SemEQUAL*, for matching multilingual names and concepts, respectively. If over time, the user is not happy with the results produced by the queries composed of such operators, the user may increase the quality threshold [5], which may result in more detailed computations by such operators for certain phonemically close words. In our experiments, the increase in operator cost is obtained by running CPU intensive computations every time a tuple has to be processed by an operator, and varying this cost depending on the tuple’s data values. Figure 12(e) shows that *ST-QM* is quite effective at detecting and adapting to real concept drifts. On average, *ST-QM*’s approach results in 15 to 28% faster output rate than the non-adaptive *QM* case.

For hybrid concept drift, we varied both data values and operator costs. Figure 12(f) shows the results for continuous hybrid concept drift occurrence, i.e., when both virtual and real concept drifts take place together. We can observe, that *ST-QM* outperforms non-adaptive *QM* by 24% to 41% in hybrid concept drift case.

7.2.3 Run-time Overhead of *ST-QM*

ST-QM has three overheads: monitoring, analysis and actuation. We instrumented the code to determine the time spent by each of these overheads. Figure 13 reports the overheads per workload of tuples relative to the total execution cost. A workload in this experiment is a set of data tuples received and processed during time interval between any two *ST-QM* invocations.

The monitoring overhead per tuple was measured as the time taken by the function that performs sampling and makes the decision whether to discard or keep the sample (see Section 4.1). For execution statistics monitoring, we have instrumented each operator to measure the time spent computing the statistics (selectivities and execution cost) for each “statistics” *ruster* (Section 4.2). The analysis overhead was measured as the time taken by the function that performs concept drift detection, to invoke the optimizer, and to produce tuning recommendations (see Section 5). The actuation overhead was measured as the time taken to replace the current classifier with a new classifier (described in Section 6).

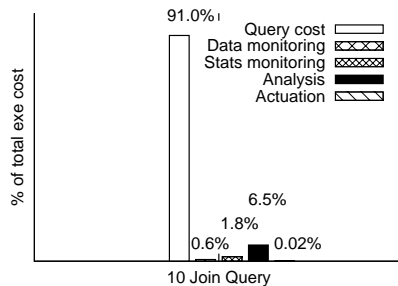


Figure 13: *ST-QM* overhead.

The total overhead (monitoring together with analysis and actuation) is 2.42% of the total execution time without optimizer invocation, and 8.92% with optimizer invocation. One important parameter to control the overhead of *ST-QM* is the size of the training tuple set or the new tuples’ sample size (due to space constraint, the chart is not shown). The

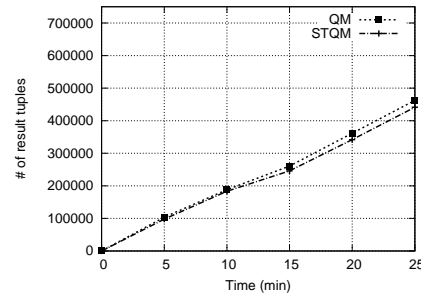


Figure 14: Overhead when no adaptation is needed.

more tuples get collected, the larger is the analysis overhead and the optimizer overhead. The optimizer overhead is especially sensitive to the size and type of training tuples collected, as was previously reported in [35]. A balance must be kept between the size and the quality of the training data.

In addition, we also measured the worst case scenario for *ST-QM*: when no concept drift occurs and the adaptation is not needed. If there are no changes in the environment, no benefit can be gained from changing to a different *QM* solution. Thus, differences in the output rates must be due to extraneous overhead (and not due to better decisions). For this experiment, we ran our experimental query over the Poisson-distributed dataset without any changes to the data and with *ST-QM* functionality enabled. Figure 13 displays the average over 5 runs of the query. When no benefit is possible, *ST-QM* is on average between 2.2 - 4.8% worse than static *QM* in the total number of results produced. This result confirms that *ST-QM* approach has detected that changes were insignificant, based on its monitoring and concept drift detection and did not invoke the optimizer. By discarding such insignificant adaptivity cases early, it minimized its adaptivity overhead. This overhead can be further reduced in the system by minimizing the monitoring frequency of both data and execution statistics.

7.3 Summary of Experimental Conclusions

The main points of our experimental study can be summarized as follows:

1. *ST-QM* can give up to 44% improvement in execution time and output rate.
2. *ST-QM* is highly adaptive to virtual, real and hybrid concept drifts and can result in some cases in up-to 41% improvement compared to non-adaptive *QM*.
3. The runtime overhead of *ST-QM* relative to query execution is small (at most 7%) . The actuation cost of physical adaptivity is nearly negligible resulting in 0.02% of total execution cost.
4. Even if no adaptivity is needed, *ST-QM*’s performance in the worst case will be at most 2-4% slower than of static *QM*.

8. RELATED WORK

Previous work on adaptive query processing primarily considers traditional *single execution plan* query processing strategies [3]. One approach is to collect statistics about query sub-expressions during execution and use the accurate statistics to generate better plans for future queries [26]. Other approaches [18, 28] reoptimize parts of a query plan following a materialization point based on accurate statistics of the materialized sub-expression. We are unaware of any solutions in database optimization that abstract adaptive query processing as a concept drift problem.

The Eddies architecture [12, 21, 33, 39, 43] enables very fine-grained adaptivity by eliminating query plans entirely, instead tuples are adaptively routed by the central Eddy operator, to all the operators, one-at-a-time, thus determining routes at runtime. This approach, although highly adaptive, suffers from a continuous re-optimization overhead and does not exploit stable conditions that often are prevalent in a system for some time. Our approach is more coarse-grained than Eddies, since at a given point in time a particular route (plan) is fully pre-computed and may be used by groups of tuples for query evaluation.

Current methods to concept drift detection in machine learning generally involve constant relearning, either by decaying the importance of older instances [20], by block re-training [44], or by sub-tree insertion for decision tree classifiers [17]. Adaptation in changing context continues to be an active research area [15, 17, 23] in data mining and machine learning. For a survey of data mining techniques in streaming environments, we refer the reader to [25].

9. CONCLUSION

This paper addresses the problem of *adaptivity* in the multi-plan-based query processing engines. We have presented a *Self-Tuning Query Mesh (ST-QM)* architecture that uses multiple plans for processing different subsets of data, and yet is as adaptive as the “plan-less” systems. *ST-QM* increases the efficiency of query processing in highly dynamic environments, by adapting the multi-plan solution, so that different subsets of data may benefit from different execution plans over time. *ST-QM* approach is unique in that it abstracts the problem of adaptive query processing as a concept drift problem. Such abstraction allows *ST-QM* to discard adaptivity candidates early in the process, if the changes are insignificant to adapt to and thus minimize the adaptivity overhead. The key characteristic of the *ST-QM* approach is that all logical changes to the current *QM* solution get translated into a simple physical operation, namely the classifier change. Our most important contribution is that we have shown in our prototype implementation that *ST-QM* approach can be simultaneously inexpensive and adaptive. Our experimental study indicates that *ST-QM* can adapt to different types of concept drifts very efficiently. Furthermore, the run-time overhead of *ST-QM* execution is fully amortized by the performance benefits of the better multi-plan-based query processing.

10. REFERENCES

- [1] Ms sql server. <http://www.microsoft.com/sql/default.mspix>.
- [2] A. Arasu et.al. Characterizing memory requirements for queries over cont. data streams. In *PODS*, pages 221–232, 2002.
- [3] A. Deshpande et. al. Adaptive query processing. In *Foundations and Trends in Databases*, 2007.
- [4] A. Deshpande et.al. Lifting the burden of history from adaptive query processing. In *VLDB*, pages 948–959, 2004.
- [5] A. Kumaran et.al. On pushing multilingual query operators into relational engines. In *ICDE*, pages 98–110, 2006.
- [6] B. Babcock et. al. Models and issues in data stream systems. In *PODS*, pages 1–16, 2002.
- [7] I. M. Chakravarti and et.al. *Handbook of Methods of Applied Statistics*, volume I. John Wiley & Sons, 1967.
- [8] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *TODS*, 9(2):163–186, 1984.
- [9] T. M. Cover and J. A. Thomas. *Elements of information theory*. Wiley-Interscience, New York, NY, USA, 1991.
- [10] D. Michie et.al. *Machine learning, neural and statistical classification*. Ellis Horwood, 1994.
- [11] DB2. <http://www.ibm.com/software/data/db2/>.
- [12] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Rec.*, 33(1):44–49, 2004.
- [13] E. Rundensteiner et.al. Cape: Cont. query engine with heterogeneous-grained adaptivity. In *VLDB*, pages 1353–1356, 2004.
- [14] G. Widmer et.al. Effective learning in dynamic environments by explicit context tracking. In *ECML*, pages 227–243, 1993.
- [15] Haixun Wang et.al. Mining concept-drifting data streams using ensemble classifiers. In *KDD*, pages 226–235, 2003.
- [16] D. J. Hand, P. Smyth, and H. Mannila. *Principles of data mining*. MIT Press, Cambridge, MA, USA, 2001.
- [17] G. Hulten, L. Spencer, and P. Domingos. Mining time-changing data streams. In *KDD*, pages 97–106, 2001.
- [18] Z. Ives. *Efficient Query Processing for Data Integration*. PhD thesis, University of Washington, 2002.
- [19] J. Gehrke et.al. Rainforest - a framework for fast decision tree construction of large datasets. *Data Min. Knowl. Discov.*, 4(2-3):127–162, 2000.
- [20] Jeffrey C. Schlimmer et.al. Beyond incremental processing: Tracking concept drift. In *AAAI*, pages 502–507, 1986.
- [21] K. Claypool et.al. Teddies: Trained eddies for reactive stream processing. In *DASFAA*, pages 220–234, 2008.
- [22] K. Hua et. al. Considering data skew factor in multi-way join query optimization for parallel execution. *The VLDB Journal*, 2(3):303–330, 1993.
- [23] M. Kubat and G. Widmer. Adapting to drift in continuous domains. In *ECML*, pages 307–310, 1995.
- [24] J. F. Kurose and K. Ross. *Computer Networking: A Top-Down Approach Featuring the Internet*. Addison-Wesley Longman Publishing Co., Inc., Boston, MA, USA, 2002.
- [25] M. Gaber et.al. Mining data streams: a review. *SIGMOD Rec.*, 34(2):18–26, 2005.
- [26] M. Stillger et.al. Leo - db2’s learning optimizer. In *VLDB*, pages 19–28, 2001.
- [27] T. M. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.
- [28] N. Kabra et.al. Efficient mid-query re-optimization of sub-optimal query execution plans. In *SIGMOD*, pages 106–117, 1998.
- [29] Oracle. <http://www.oracle.com/index.html>.
- [30] P. Bizarro et.al. Content-based routing: Different plans for different data. In *VLDB*, pages 757–768, 2005.
- [31] Q. Li et.al. Adaptively reordering joins during query execution. In *ICDE*, pages 26–35, 2007.
- [32] R. Agrawal et.al. An interval classifier for database mining applications. In *VLDB*, pages 560–573, 1992.
- [33] R. Avnur et.al. Eddies: Continuously adaptive query processing. In *SIGMOD*, pages 261–272, 2000.
- [34] R. Lipton et.al. Efficient sampling strategies for relational db operations. *Theor. Comput. Sci.*, 116(1):195–226, 1993.
- [35] R. Nehme et.al. Query mesh: An efficient multi-route approach to query optimization. Technical Report CSD TR #08-009, Purdue University, April 2008.
- [36] S. Babu et. al. Adaptive ordering of pipelined stream filters. In *SIGMOD*, pages 407–418, 2004.
- [37] S. Chaudhuri et. al. Sqlcm: A continuous monitoring framework for relational database engines. In *ICDE*, pages 473–485, 2004.
- [38] S. Guha et.al. Data streams and histograms. In *STOC*, pages 471–475, 2001.
- [39] S. Madden et. al. Continuously adaptive continuous queries over streams. In *SIGMOD*, 2002.
- [40] S. Viglas et.al. Rate-based query optimization for streaming information sources. In *SIGMOD*, pages 37–48, 2002.
- [41] F. Tian and D. J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.
- [42] A. Tsymbal. The problem of concept drift: definitions and related work. Technical Report TCD-CS-2004-15, The University of Dublin, Trinity College, 2004.
- [43] V. Raman et.al. Using state modules for adaptive query processing. In *ICDE*, pages 353–365, 2003.
- [44] Venkatesh Ganti et.al. Mining data streams under block evolution. *SIGKDD Explor. Newsl.*, 3(2):1–10, 2002.
- [45] H. Wang and J. Pei. A random method for quantifying changing distributions in data streams. In *PKDD*, pages 684–691, 2005.
- [46] Z. Bar-Yossef et.al. Approximating edit distance efficiently. In *FOCS*, pages 550–559, 2004.