Query Mesh: Multi-Route Query Processing Technology

Rimma V. Nehme
Purdue University
rnehme@cs.purdue.edu

Karen E. Works
Worcester Polytechnic Institute
kworks@cs.wpi.edu

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

Elisa Bertino
Purdue University
bertino@cs.purdue.edu

ABSTRACT

We propose to demonstrate a practical alternative approach to the current state-of-the-art query processing techniques, called the "*Query Mesh*" (or *QM*, for short). The main idea of *QM* is to compute multiple routes (i.e., query plans)[1], each designed for a particular subset of data with distinct statistical properties. Based on the execution routes and the data characteristics, a *classifier* model is induced and is used to partition new data tuples to assign the best routes for their processing. We propose to demonstrate the *QM* framework in the streaming context using our demo application, called the "*Ubi-City*". We will illustrate the innovative features of *QM*, including: the *QM* optimization with the integrated machine learning component, the *QM* execution using the efficient "*Self-Routing Fabric*" infrastructure, and finally, the *QM* adaptive component that performs the online adaptation of *QM* with near-zero runtime overhead.

1. INTRODUCTION

In most database systems, traditional and stream systems alike, the optimizer picks a *single* query plan for all data based on the overall statistics of the data [5]. The execution costs for alternative plans are estimated and the one with the overall cheapest cost is chosen. Real-life datasets, however, tend to have non-uniform distributions and selecting a *single execution plan* may result in the execution that is ineffective for possibly large portions of the actual data. For example, in network communication, in case of a congestion, different traffic types (e.g., voice, multimedia or data) may have various probabilities of being discarded by routers. A query monitoring the traffic on a network may observe diverse frequencies for packets of different traffic types. Stocks from various sectors may exhibit different fluctuation patterns, and a query continuously correlating stocks' behavior with the latest news or blogs may encounter distinct statistics for various industry sectors. In a health application, sensors measuring patients' vital signs may observe different values depending on the patients' age, gender or geographic location.

[1]We use terms "plans" and "routes" interchangeably in our work.

VLDB '09, August 24-28, 2009, Lyon, France

Middle Ground in Query Processing

The most prevalent query processing approach in current systems and in the literature is using a single execution plan. Such approach has a major disadvantage: its optimization coarseness [1]. It does not focus on addressing the *intra-data* variations and often cannot serve well many of rather diverse subsets of data. Having a fully established plan a priori (at compile time), however, has several key advantages: all tuples follow the same plan, resulting in a nearly "overhead-free" execution. Moreover, data tuples' sizes do not need to be extended to store any optimization-related metadata (e.g., tuple lineage).

On the other hand, recent systems, like Eddies [1], tend to use multiple routes by default. Such systems at runtime decide which operator should process a tuple next, thus discovering different execution routes for tuples on-the-fly. Unfortunately the "optimization decision" is made continuously, and in the worst case for every individual tuple, leading to its main disadvantage – the unavoidable "route discovery" overhead. Furthermore, individual tuples' sizes tend to be larger, as they must now carry their individual "itineraries".

In summary, optimizing too frequently as in the Eddies approach may discover multiple plans, but may also result in wasted resources, especially if the environment is stable. However, optimizing too coarsely as in the systems pre-computing typically a single plan may miss critical opportunities to improve query execution performance. We thus have developed a practical middle ground solution between these two extremes called the *Query Mesh* (or *QM* for short) [7] model (see Figure 1). The main idea of *QM* system is to compute multiple execution routes a priori, each optimized for a subset of data with distinct statistical properties, and then infer a *decision tree-based classifier* model based on the computed set of routes and the observed data characteristics. At runtime, the new data is classified (i.e., partitioned) using the classifier into distinct subsets, and each subset is processed by the best route, customized for its respective local statistics. While the conditions are sta-

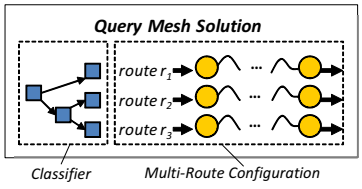

Figure 1: *QM* solution overview.

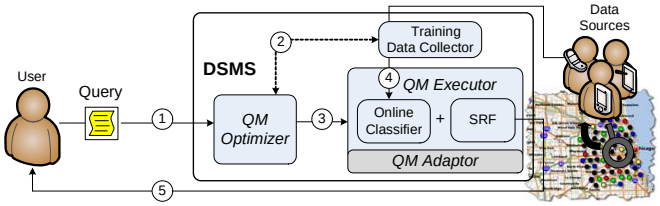

Figure 2: *QM* execution overview.

ble (which is monitored by the *QM Adaptor*), the classifier and the execution routes stay unchanged. For classification, we consider decision tree model as it helps to "zero-in" on the sought-after routes very quickly with typically a small number of comparisons (as will also be illustrated in our demonstration).

2. TECHNICAL DETAILS

2.1 Research Challenges

Several practical considerations make finding a good quality query mesh challenging. First, there is a combinatorial explosion in the search space. Finding a single optimal plan is already known to be *NP*-hard; by using multiple plans concurrently, the problem complexity increases multi-fold. Second, *QM* is computed based on the collected training dataset, and selecting a good representative dataset is a research topic in its own right. In our demo, we will demonstrate the various techniques for training data collection and their effect on the resulting query mesh (Section 3). Third, the classifier model and the number and the choice of particular execution routes are strongly dependent on each other. A change in one component may cause a modification in the other, subsequently affecting the cost of the overall *QM* solution. Such interplay between the routes and the classification introduces a dilemma regarding how a *QM* solution should be computed. Should the training dataset be partitioned first, and then the optimizer would compute the routes for the different partitions. Or alternatively, should some effective routes be determined first, and then the tuples from the training dataset would get assigned to one of the established routes[2]. Fourth, the execution of multiple routes must be done in parallel and must be very efficient, not to "cancel out" the benefits of using the multi-route strategy. This requires an effective runtime execution infrastructure to support the multi-route execution paradigm. Finally, given that we consider a streaming environment, the system conditions may change over the lifetime of a query execution, thus the *QM* system must be able to detect and adapt to runtime conditions in a light-weight manner.

2.2 The QM System and Technology

Overview

Our *QM* framework is implemented inside a Java-based DSMS prototype called *CAPE* [3]. Figure 2 gives a high level overview of *QM* execution. A user specifies a continuous query (Step 1), the *QM Optimizer* computes a logical *QM solution* – a set of execution routes and a classifier – based on the available training dataset that characterizes the latest streaming data (Step 2). Then the optimizer forwards the logical *QM* specification to the *QM Executor*, which instantiates the *QM* physical runtime infrastructure composed of the *Online Classifier Operator* and the *Self-Routing Fabric* (*SRF*) (Step 3). After the instantiation, the processing of the incoming streaming data begins (Step 4). The tuples are

[2]We will demonstrate both strategies in our demo.

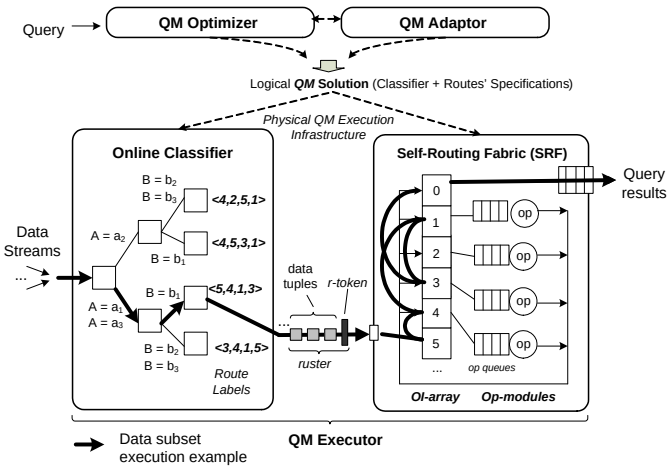

Figure 3: *QM* framework architecture.

partitioned into groups using the online classifier operator and are forwarded into the *SRF* for actual query execution. As *SRF* produces final tuples, they are returned to the user as query results (Step 5). Figure 3 illustrates the architecture of the key *QM* components in more detail.

QM Optimizer

QM optimizer computes a logical *QM* solution offline based on the available data samples (i.e., the training data) and their statistics. To find a good quality *QM* solution without exhaustive enumeration of the search space, *QM* optimizer uses a series of cost-based search heuristics. The heuristics address the following search sub-tasks: (1) selection of a promising *start QM solution*; (2) choice of an effective *search strategy*; and, (3) selection of a *stop condition* to terminate the *QM* search. To identify the relationships between the data and the resulting execution routes, a decision tree (or *DT*, for short) classifier model is induced to be used for fast and yet accurate classification of new data. A compact classifier is constructed by employing machine learning measures of data impurity based on *entropy* and *information gain*.

QM Executor

Based on the logical *QM* specification received from the optimizer, the *QM* executor instantiates the physical runtime infrastructure consisting of the *Online classifier Operator* and the *Self-Routing Fabric* (*SRF*) (see Figure 3). When new tuples arrive, they first get processed by the Online Classifier operator to determine the routes that will be used for their processing. The classification of data is cheap, largely due to decision trees typically being compact in size (which tends to be the norm). The classifier model stays unchanged during the execution, except when *QM* adaptation is required, and the current classifier is replaced by a new classifier. The benefits of the *QM* approach is that it classifies tuples cheaply, and there is no need for continuous online re-optimization as it is done in Eddies. After classification, tuples are forwarded into the *SRF* for the actual query evaluation according to their assigned routes. *SRF* has two key elements: (1) *Operator Index Array* (*OI-array*) that stores the pointers to all of the query operators in *SRF*. Here, each index i corresponds to a unique operator op_i[3], and (2) *Operator Modules* which are the actual operators processing the tuples. *SRF* enables concurrent multi-route

[3]Index "0" is reserved for the *SRF* global output queue, where the result tuples are placed to be sent to the applications.

execution without instantiating physical topologies of the execution plans and without any central router operator like Eddy [1]. Operators execute different routes based on the streaming metadata, called *r-tokens*, embedded inside data streams and encoded *routing instructions* for the streaming data. Routes in *r-tokens* are specified in the form of an *operator index stack* based on the design of *SRF*. An example of runtime execution is depicted by a thick black arrow in Figure 3. Consider an *SRF* with the operator index array as follows: *OI-array*[1] = op_i, *OI-array*[2] = op_j, *OI-array*[3] = op_k, *OI-array*[4] = op_l, *OI-array*[5] = op_m. Then a route $r = <op_m, op_l, op_i, op_k>$ will be encoded in an *r-token* as a stack <5,4,1,3>, where '5' is the first operator in the route and '3' is the last. The top of the stack represents the index of the operator in the *SRF*. A distinct group of tuples (called *ruster*[4]) is always routed to the operator that is currently the top node in the routing stack. *OI-array* enables the knowledge of the "location" of all operators. After an operator is done processing, the operator "pops" its index from the top of the routing stack in the *r-token*, and then forwards the tuples following it to the next (now the top) operator. The design of *SRF* and the embedded into streams route encodings enable the de-centralized self-routing of data by regular query operators.

QM Adaptor

Given that *QM* is deployed in dynamic data stream environments, *QM* must be able to adapt to changes in runtime conditions. The interesting question here is whether 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 statistics may change considerably requiring the adaptation of the execution strategy. The fundamental challenge for *QM* is the problem of determining the discrepancy between the previously constructed *QM* model[5] and the currently most suitable *QM* solution based on the new data characteristics, i.e., data values and their statistics. In machine learning, such disrepancy is called a *concept drift* [4]. 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* (or *real*) *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. The necessity to change the current model due to changes in the data distribution is called a *virtual concept drift* [7]. A *real concept drift* may occur when more accurate statistics become available during execution and the routes in *QM* must be adapted based on this new information [7]. Virtual and real concept drifts often occur together. We refer to such case as *hybrid concept drift*. From a practical point, a concept drift (real, virtual, or both) gives a good indication that the current *QM* solution needs to be adapted.

The key feature of *QM* physical adaptivity is that it requires only a *single online operation*, namely the classifier change, without affecting the rest of the infrastructure[6] [7].

[4] "*Ruster*" stands short for *routable tuple cluster*.

[5] A *QM* represents a particular "model" of execution, as de- termined by the classifier and the set of execution routes. In machine learning, this term is commonly used to refer to classifier-based systems.

[6] The optimizer determining a new *QM* based on the latest conditions is executed offline (on a separate thread).

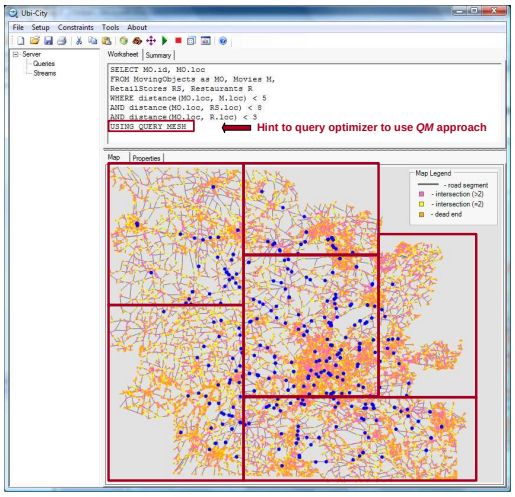

Figure 4: *Ubi-City* application.

To accomplish this, a pointer re-assignment to the new classifier (i.e., new decision tree) is the only step that is needed. The *QM* runtime infrastructure seamlessly enables the processing of tuples using both "old" as well as "new" routes concurrently. The physical separation between the component that determines which plans should be used for execution (the classifier) and the component that actually executes them (the *SRF*), enables *QM* adaptivity to be so light-weight and efficient.

3. DEMONSTRATION

Demo Scenario: Ubi-City

The real-life application we consider in our demo is a ubiquitous city or short the *Ubi-City* [8], representing a region, where virtually everything is linked to an information system through technologies such as wireless networking or RFID tags. As was mentioned before, *QM* addresses the problem of *intra-data variations* and employs the best execution strategy for each distinct subset of data. However, partitioning of data into distinct subsets is not limited to only data values. In our demo, we will present a case where distinct subsets can be identified based on the data *security policies* described by the streaming *security metadata* called "security punctuations" (or short *sps*) [6]. *Sps* are interleaved with streaming data and describe the access control policies for different portions of the streaming data. Here, *QM* execution paradigm is used in conjunction with *security-aware query processing*, and distinct subsets are determined based on how restrictive the data's access control policies are. Distinct security policies may lead to different best operator orderings for various portions of input data. In real-life this scenario may happen when people are moving, and their devices automatically adapt the security policies based on their location, the proximity of businesses and users' preferences, limiting to who would be allowed to "see" them. This helps to protect users from "context-aware spam" – information or services people don't know of or agree to.

Data and Queries

For data, we use the *Network-based Moving Objects Generator* [2] to generate continuously moving objects (e.g., cars, pedestrians with GPS devices) travelling in the city of Worcester, MA USA[7]. Moving objects continuously and

[7] The demo is not limited to this geographic location only. Our implementation supports Tiger line files as inputs, which can run the

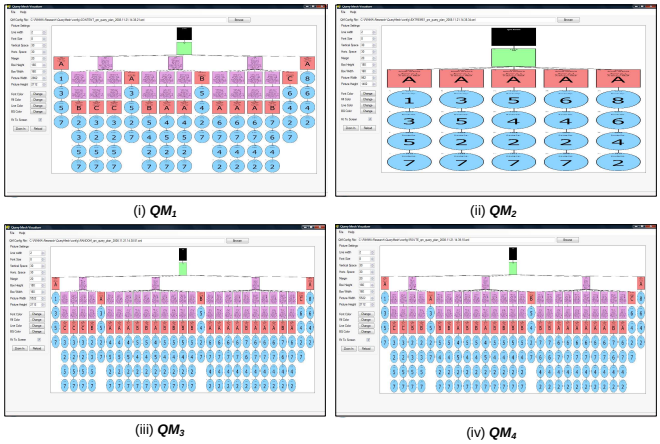

Figure 5: Visualization of logical *QM* solutions.

selectively restrict access to their current location using security punctuations. The examples of queries we will use in our demo include an n-way join and expensive filter queries of the form:

```
SELECT MO.id, Mo.loc
FROM MovingObjects as MO
WHERE Similar(MO.Interests[1
hour], Teenager) AND
Similar(MotionPattern(MO.loc[20
min]), Stationary) AND
!Similar(MO.Activities[1
hour], Lunch)
```

```
SELECT MO.id, MO.loc FROM
MovingObjects as MO,
Movies M, RetailStores
RS, Restaurants R WHERE
distance(MO.loc, M.loc) <
5 AND distance(MO.loc, RS.loc)
< 8 AND distance(MO.loc,
R.loc) < 3.
```

Such queries may be executed by a company providing location-based services to businesses in a certain area, e.g., [9].

Walkthrough

The audience of our demo will be able to see and perform the following actions using our *QM* system:

Distinct data generation: The attendees will be able to manipulate the parameters for generating the data with distinct characteristics (by modifying the access control policies) using our provided UI (see Figure 4).

Query specification: Second, the users will be able to submit a query to the DSMS using SQL language with a "hint" for the optimizer to employ *QM* optimization as shown in Figure 4 (top).

Sampling and training data collection: We have explored several techniques from statistics, including random sampling with cross-validation and sampling with bootstrapping for *QM* training data collection. Using these methods, the system can estimate how well the selected training dataset is going to represent the future yet-unseen data, and re-sample the data until the desired accuracy is achieved. We will illustrate how various techniques can make a difference on the training tuple mix and the resulting *QM*.

***QM* optimization**: After a user submits a query, the *QM* optimizer will be invoked to compute the best logical *QM* solution for the query. Given different search parameters, we will show how the *QM* optimizer may find a different *QM* solution and visualize it (see snapshots in Figure 5). Using the GUI, the audience will be able to see the distinct subsets of data, the structure of the classifier, and the best routes determined for execution. We will also illustrate a graphical respresentation of the *QM lattice-shaped* search space (see left in Figure 6) and the sub-space iterated by the optimizer when searching for a *QM* solution under different input parameters.

system in the context of other geographic areas as well.

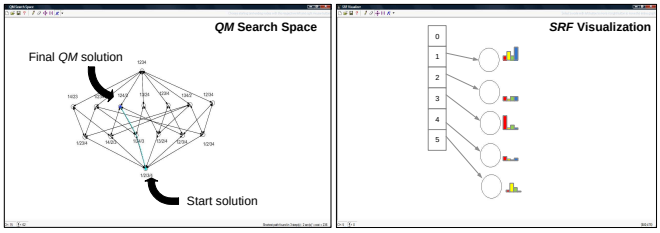

Figure 6: *QM* search space and *SRF* visualization.

***QM* execution**: After optimization, the *QM Executor* will be called to consume the logical *QM* solution (produced by the optimizer), to instantiate the physical runtime infrastructure. To highlight the uniqueness of *QM* runtime architecture, we will show the *stack-based* encoding of the routes in the *r-tokens* that allow the operators to "self-route" the data. Using our *SRF* visualizer (see Figure 6 on the right), we will show the *SRF* with the instantiated in it query operators and their runtime properties (operator statistics). The audience will thus observe the complete path of a query, from specification to optimization and finally to physical instantiation of query mesh runtime infrastructure.

***QM* adaptation**: To illustrate the *QM* adaptive component [7], we will periodically trigger changes in the security policies depicted by the streaming *sp*s. We will show how the *QM* system adapts to these changes during query execution. New *QM* solution will be computed offline and then a single and very cheap physical operation – the replacement of the classifier, will be performed online.

Performance showcase: Finally, our demo will present the performance benefits of our *QM* system compared to the existing approaches, namely the static single plan and the Eddies systems. Furthermore, we will also measure and visually illustrate the statistics for various *QM* overheads, including the *QM* optimization cost together with the runtime execution and the adaptivitiy overheads.

4. CONCLUSION

We propose to demonstrate a *Query Mesh* (*QM*) system which takes an innovative approach towards addressing the real-world problem that "*all data is not created equal*". *QM* provides a middle-ground approach compared to the single plan-based systems and the solutions that continuously rediscover different plans for different data at runtime. The key contribution of our demo is to show that *QM* is practical and versatile and implemented in a prototype DSMS can achieve great performance improvements.

5. REFERENCES

[1] A. Deshpande et. al. Adaptive query processing. In *Foundations and Trends in Databases*, 2007.
[2] T. Brinkhoff. A framework for generating network-based moving objects. *Geoinformatica*, 6(2):153–180, 2002.
[3] E. R. et. al. Cape: Continuous query engine with heterogeneous-grained adaptivity. In *VLDB*, 2004.
[4] T. M. Mitchell. *Machine Learning*. McGraw-Hill, NY, 1997.
[5] R. Ramakrishnan and J. Gehrke. *Database Management Systems*. McGraw-Hill Higher Education, 2000.
[6] R.Nehme et.al. A security punctuation framework for enforcing access control on stream. data. In *ICDE*, pages 406–415, 2008.
[7] R.Nehme et.al. Self-tuning query mesh for adaptive multi-route query processing. In *EDBT*, 2009.
[8] S.Kim et.al. Ubiquitous city technology & applications. *Int. Conference on Convergence IT*, 0:2342–2348, 2007.
[9] The Carbon Project. http://www.thecarbonproject.com/.