

# **Query Mesh: An Efficient Multi-Route Approach to Query Optimization**

Rimma Nehme  
Karen Works  
Elke Rundensteiner  
Elisa Bertino

CSD TR #08-009  
March 2008

# Query Mesh: An Efficient Multi-Route Approach to Query Optimization

Rimma V. Nehme <sup>#1</sup>, Karen Works <sup>\*2</sup>, Elke A. Rundensteiner <sup>\*3</sup>, Elisa Bertino <sup>#4</sup>

<sup>#</sup>Purdue University, West Lafayette, IN 47906 USA

<sup>1</sup>rnehme@cs.purdue.edu, <sup>4</sup>bertino@cs.purdue.edu

<sup>\*</sup>Worcester Polytechnic Institute, Worcester, MA 01608 USA

<sup>2</sup>kworks@cs.wpi.edu, <sup>3</sup>rundenst@cs.wpi.edu

**Abstract**—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. It has however been repeatedly observed that real-life datasets are non-uniform. Selecting a single execution plan may result in a query execution that is ineffective for possibly large portions of the actual data. In this paper, we present a practical alternative to the current state-of-the-art query optimization techniques, termed a *multi-route query mesh* model (or short *QM*). The main idea of *QM* is to compute multiple routes (query plans), 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. The classifier is used for efficient partitioning of the new data to assign the best route for query processing. We formulate the *QM* search space and analyze its complexity. To find optimal query meshes, we design the *Opt-QM* algorithm. Faced with a dilemma – whether to determine distinct data subsets or to compute a set of execution routes first, we design several heuristics that can effectively find good quality query meshes very efficiently. For runtime query processing, we employ a *Self-Routing Fabric (SRF)* infrastructure which supports shared operator processing and has near-zero routing overhead. Results of our experimental study with real-life and synthetic data indicate that *QM*-based approach consistently provides better query execution performance for skewed datasets compared to the state-of-the-art alternatives, namely both the traditional systems that employ a single pre-computed plan execution and also the systems that determine different routes on-the-fly.

## I. INTRODUCTION

### A. Single versus Multiple Execution Plans

Most modern query optimizers determine a *single* “best” plan at compile time for executing a given query [1]. The execution cost for alternative plans is estimated and the one with the overall cheapest cost is chosen. The cost typically is estimated based on the average statistics of the data as a whole as the objective is to find *one* plan for all data. However, significant statistical variations of different subsets of data may result in poor query execution performance [2]. The main drawback is the very coarse optimization granularity: *a single execution plan is chosen for all data*. Such “monolithic” approach can miss important opportunities for effective query optimization [3], [2], [4].

**Network Traffic Data:** In Internet and telecommunication networks, traffic non-uniformity is inherent [5]. Some destination(s) may be more popular than others, e.g., certain web sites get higher visitation rates. Different network traffic may also

have different characteristics. For example, multimedia packets may be discarded by some routers due to congestion and due to multimedia applications being tolerant to missing data. Voice packets, however, transmitted via reliable protocols, will be guaranteed to travel through all routers. For queries monitoring network traffic, query processing could potentially benefit by tailoring plans for different network traffic types.

**Stock Market Data:** Stock market data is known to be not uniform [6]. The prices of stocks quickly reflect information concerning current events and the expectations in the future. Some categories of stocks may go up, others may go down as a result of the same event. Thus, query plans for a financial monitoring application could be customized for different stock types, based on either the region or the industry or both.

We can observe from the examples above that real-life data tends to be *not uniform*. Clearly, a single execution plan often is not likely to be able to serve well many of these rather diverse subsets of data, leading to seriously inefficient query processing for some or possibly huge fractions of data [2].

An extreme solution to tackle this problem is the Eddy system [7] and its CBR extension [3]. Here, individual tuples are routed through operators one-by-one, and query plans are modified on-the-fly at the tuple-granularity level by changing the order of query operators to which tuples are routed. Conceptually, every tuple may be processed via a unique query plan, resulting in a *multiple plans* solution. However, such *extremely fine-grained* and eager re-optimization may incur impractically large costs [8], [9]. Further, Eddy does not exploit the observation that tuples with identical content or similar statistical properties are likely to be best served by the same query plan, thus missing the opportunity to share the processing and reduce the execution overhead [3], [10].

### B. Spectrum of Query Optimization Techniques

Query optimization techniques can be classified along two dimensions: the timing of optimization decision and the granularity of optimization (Figure 1<sup>1</sup>). Some database systems determine query plans<sup>2</sup> in advance (at compile time), while others forego pre-computed plans and “route” tuples on-the-fly (at runtime). We observe that runtime versus compile-time

<sup>1</sup>Figure 1 is not an exhaustive survey of all query optimization techniques. It merely illustrates where our proposed solution fits among the existing approaches.

<sup>2</sup>We use terms “plans” and “routes” interchangeably in our work.

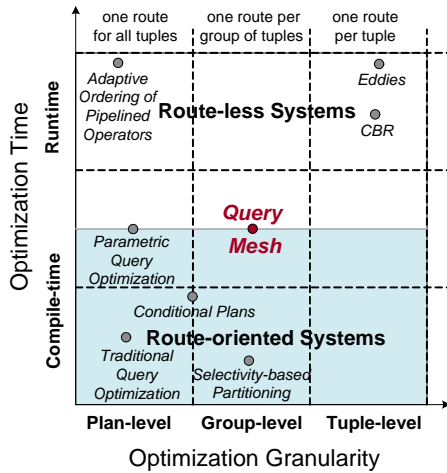


Fig. 1. Overview of the spectrum of different query optimization techniques.

query optimization solutions have a close resemblance to network communication methods which can be: (1) *connection-oriented*, also known as circuit-switched, where a connection with the receiver is established in advance before passing any data, or (2) *connection-less*, also known as packet-switched, where data is sent without establishing an a priori connection, and the next hop of a packet is determined by a router during the transmission at runtime [11]. The parallel between route optimization in networking and query optimization in databases is evident. Query operators can be viewed as a “network of routers” and the query plan as a “route” through all operators in the “network”. In databases, the goal is to find the “cheapest” route through the “network” of query operators.

Given this parallel, we classify the state-of-the-art query optimization techniques according to optimization time dimension as *route-oriented* and *route-less* solutions. By “oriented”, we mean that routes are established in-advance. Database systems, including all commercial DBMSs [12], [13], [14] that tend to establish a query plan before runtime execution, employ the *route-oriented* paradigm. Recent systems, like Eddies [7], which determine for each tuple at runtime which operator should process it next, fall under the *route-less* category [7], [15], [16]. Advantages of route-oriented query processing include: (1) faster routing, since routes are computed in advance, (2) savings on the route computation, since routes are computed only once, and (3) savings in tuples’ sizes, since all tuples are processed using the same route, and hence individual tuples do not need to carry their “itineraries” (i.e., lineage) as it is done in Eddies. In practice, however, almost all route-oriented solutions pre-compute a *single route* [17], [18]. The main disadvantage of such *single route-oriented* solutions, as was mentioned in Section I-A, is their optimization coarseness. On the other hand, route-less solutions tend to be “multi-route” by default, hence we term them *multi route-less* solutions [7], [16]. Their advantages are that they employ multiple routes for processing of different data subsets, thus improving query execution performance. Since such solutions make decisions at runtime, they can also adapt quickly to changes in the environment. However, the disadvantages here include: (1) high per-tuple state, (2)

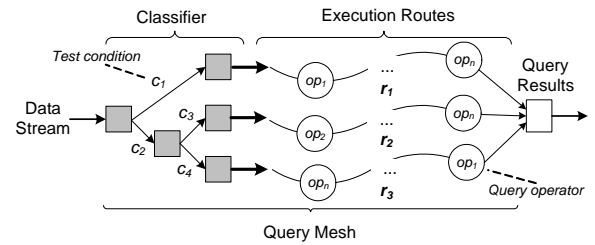


Fig. 2. Conceptual view of query mesh.

unavoidable per-tuple processing overhead, and (3) variable sometimes unpredictable tuple latencies, due to different individual tuple routes being determined at runtime.

In summary, optimizing too frequently as in the multi route-less solutions may result in wasted resources, but optimizing too coarsely as in the single route-oriented solutions may miss critical opportunities to improve query execution performance. We thus propose a practical middle ground between these two extremes in the form of a *multi route-oriented* query optimization solution called *Query Mesh* (or *QM*, for short).

### C. Our Proposed Solution: Query Mesh

The main idea of *QM* is to compute multiple routes, each optimized for different subsets of data, and then infer a classifier model based on the computed set of execution routes and the data subsets’ characteristics. At runtime, the new data is effectively classified (i.e., partitioned) using the classifier into distinct subsets, and each subset is processed using a route customized for its respective local statistics. The different routes in query mesh are executed concurrently. We compare our proposed solution against the state-of-the-art query optimization techniques in detail in Section VII.

A query mesh solution *QM* is composed of the *classifier* and the *multiple routes*<sup>3</sup> (see Figure 2). While many classification models could be plugged into query mesh, e.g., neural networks, naive bayes classifier, etc. [19] in this paper, we employ a *decision tree (DT)* classifier. In our experiments, we have observed that using *DT* classifier approach can “zero-in” on the sought-after route very quickly with a small number of comparisons.

Finding an optimal query mesh solution for a given query is an expensive process due to the combinatorial explosion in the size of the optimization space (Section III-E). We formulate the complexity of the query mesh search space and the algorithm *Opt-QM* which can find optimal query meshes (Section III-B). *Opt-QM*, however, may be not feasible in practice due to its exhaustive nature in enumerating the search space. As viable alternatives, we propose efficient search heuristics to find good quality query meshes very quickly. To further optimize these heuristic-based searches, we describe several approaches to find a good start *QM* solution which can help in finding a high-quality *QM* faster (Section IV).

For the efficient *QM*-based query execution, we design an infrastructure, called the *Self-Routing Fabric (SRF)*, which provides a near-zero route execution overhead. *SRF* eliminates

<sup>3</sup>We also refer to the set of execution routes in a *QM* solution as a *multi-route configuration*.

the expensive central dataflow router, such as the Eddy operator [7]. The query operators route data tuples in a distributed fashion, thus eliminating the “backflow” problem associated with having a central router operator [8]. The routing is based on the route specifications, encoded inside meta-data tuples, which are interleaved with the data. To keep memory and CPU overhead minimal, the routing decisions are applied at the granularity of groups of tuples, which we denote as “*routable clusters*” or short “*rusters*” rather than individual tuples. The *ruster* approach minimizes the overhead and guarantees that only the tuples that share the same best route are batched together for shared processing.

#### D. Our Contributions

The contributions of our work are summarized as follows:

- 1) We introduce a multi-route query optimization approach, called *Query Mesh* (short *QM*). *QM* is a general model composed of the learning classifier and the multi-route structure, where each route is customized for different subsets of data.
- 2) *QM* employs machine learning metric to generate an efficient classifier used for data partitioning and assigning the most suitable routes with minimum overhead.
- 3) We present the *Opt-QM* algorithm for optimal query mesh solution search and analyze its complexity.
- 4) Due to the complexity of optimal *QM* search, we devise a repertoire of alternative cost-based search heuristics that efficiently find high-quality query meshes.
- 5) We describe a *Self-Routing Fabric (SRF)* infrastructure which efficiently implements concurrent multi-route execution. The novelty of *SRF* is its support of multi-route routing by the operators without constructing physical execution plans, i.e., operator pipelines.
- 6) We thoroughly evaluate the *QM* approach through experiments comparing it to the state-of-the-art solutions, including the single route-oriented and the multi-route-less approaches. Our results show that for skewed datasets, *QM* gives substantial performance improvements over the alternatives with minimal overhead.

The rest of this paper is organized as follows. Section II provides the preliminaries and overviews the *QM* architecture. Sections III and IV discuss *QM*-based optimization. Section V describes *QM*-based query execution. Section VI presents our experimental evaluation. Section VII discusses the related work, while Section VIII concludes the paper.

## II. THE QUERY MESH: MAIN IDEA

### A. The Query Mesh Problem

For a given query, we determine a multi-route configuration that can give the best overall query execution performance. In addition to multiple concurrent routes employed for execution, a *QM* solution also includes the classifier component which at runtime assigns the routes for processing to the incoming data tuples. Clearly the classifier cost must be considered during query optimization, as classification is now a part of the overall query execution process. In order to determine the best query mesh, the query optimizer uses a *training dataset*

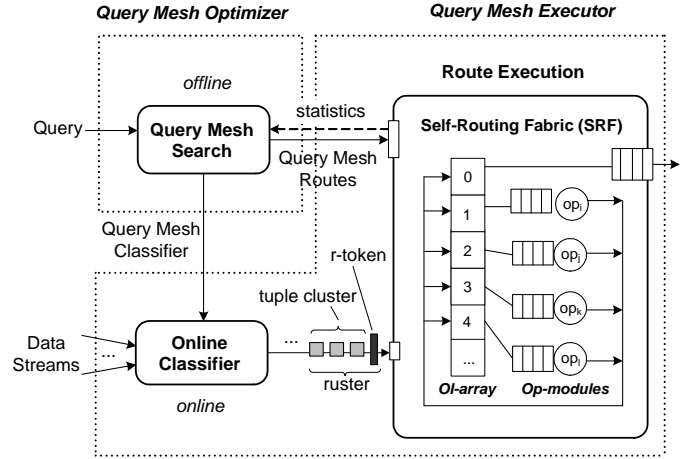


Fig. 3. Query mesh framework.

that accurately represents the distribution of the data expected to come in the future – a common approach in many database systems [3], [20] and in prediction models in data mining alike [21]. For streaming databases, relying on samples of data is unavoidable, since it is impossible to “see” all of the streaming data a priori. The query mesh optimization problem can be stated as follows.

**Query Mesh Optimization Problem:** For a given query  $Q$  and a representative dataset  $T$ , find a query mesh solution  $QM$  consisting of multi-route configuration  $R$  and classifier  $C$  that results in the lowest execution cost for tuples in  $T$ .

Several practical considerations make this problem challenging: (1) Finding such optimal solution is complex, because there is a combinatorial explosion of route configurations for all possible subsets of training data to consider. (2) Selecting a good training dataset is challenging, as its size and quality directly affect the *QM* search space. With smaller training data, we may be able to enumerate all possible *QM* solutions, but the training set may represent the real data with low accuracy. Whereas with a larger dataset, we may be able to more accurately represent the actual data, yet at the cost of an extremely large search space, making it impossible to enumerate all solutions. (3) Moreover, the structure of the classifier 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 to the other, subsequently affecting the cost of the overall *QM* solution. This introduces a dilemma – how should a query mesh be computed? Both scenarios below can be exploited: first, training data gets partitioned based on the similarity of attribute values – one criterion among several possible alternatives for partitioning of data, and then the routes are computed for the different data partitions. Alternatively, some effective routes are computed first, using the training data statistics, and then the data partitions and the classifier model are induced based on the computed routes. Clearly, query mesh design is a complex problem.

### B. Our Assumptions

In this paper, we consider select-project-join queries, however, ideas presented are general. We focus on dynamic

data stream management systems (DSMS), yet the underlying execution system could either be a static DBMS or a dynamic DSMS. We consider optimization of a single query and assume stable conditions. The problem of adaptive processing when either the environment or the data characteristics change during query execution is outside the scope of this paper.

### C. Query Mesh Framework Overview

The *QM* framework consists of two primary components: *query mesh optimizer* and *query mesh executor* (see Figure 3). For a given query, using training data and statistics, the *QM*-based optimizer computes a *QM* solution offline. The query mesh executor takes the *QM* solution (the classifier and the multiple routes) generated by the optimizer and instantiates the *QM* physical runtime infrastructure. We describe each of these components in detail next.

## III. QUERY MESH OPTIMIZER

### A. Data Sampling

The selection of the training data, i.e., which sampling technique should be employed to accurately depict the distribution of the data, is a research topic in its own right. For the purpose of this work, we have explored several techniques from statistics, including random sampling with cross-validation [21] and bootstrapping [22]. Using these methods, we can estimate how well the selected training dataset is going to represent future as-yet-unseen data, and re-sample the data until the desired accuracy is achieved [21]. In practice, the training dataset may also be collected using statistics from previous (historical) execution runs of the query or by employing a similar approach to *plan staging* [23], where optimization and execution are interleaved. The first stage of query processing may use a traditional single plan for execution while simultaneously collecting data statistics and training data. Using the samples from the first stage as the training set, the query mesh can be computed by the optimizer and then used for the execution in the next (*QM*-based) stage.

### B. Query Mesh Search Space

Given a query and training data, we now study how many possible multi-route configurations exist. These configurations would have to be evaluated in order to find the optimal *QM* solution, thus they comprise the *QM* optimization space.

The training dataset size has a direct impact on the size of the query mesh search space. Hence, the training dataset must be selected wisely to be compact yet sufficiently representative of the real data. To reduce the size of the search space, we can perform a “*compression*” on the set of sampled data tuples by clustering the tuples based on the similarity of data values. Any clustering algorithm from the literature can be applied here<sup>4</sup> [24], [25]. After clustering, each tuple cluster serves as an abstraction for a subset of the sample dataset. Then a *centroid* tuple [26] is chosen to represent each tuple cluster. For simplicity of discussion, in the rest of the paper, we will refer to such centroid tuple as a *training tuple*  $t$  and a dataset that consists of such centroid tuples, i.e., the

<sup>4</sup>In our implementation, we use  $k$ -means clustering – one of the most commonly used algorithms.

tuples summarizing their respective tuple clusters, as a *training dataset*  $T$ .

Since routes are computed based on the training dataset  $T$ , the spectrum of possible multi-route configurations ranges from *an individual route per each training tuple* in  $T$  to a *single route for all tuples* in  $T$ . Let  $n$  denote the cardinality of the training tuple set  $T$ , i.e.,  $n = |T|$ . The upper-bound for all possible multi-route configurations corresponds to the number of distinct possible ways of assigning  $n$  distinguishable tuples to one or more routes. The number that describes this value is the *Bell number* ( $B_n$ ), which represents the number of different *partitions* of a set of  $n$  elements [27]. In our work, a multi-route configuration, which represents the set of execution routes, is a *partition* of the training tuple set  $T$  defined as a set of non-empty, pair-wise disjoint subsets of  $T$  whose union is  $T$  (see Figure 4). For example,  $B_3 = 5$  because the 3-element set  $\{1, 2, 3\}$  can be partitioned in 5 distinct ways:  $\{\{1\}, \{2\}, \{3\}\}$ ,  $\{\{1\}, \{2,3\}\}$ ,  $\{\{2\}, \{1,3\}\}$ ,  $\{\{3\}, \{1,2\}\}$  and  $\{\{1,2,3\}\}$ <sup>5</sup>. The *Bell number* describes the size of the query mesh search space, i.e., the total number of all possible partitions for an arbitrary training dataset  $T$ . Mathematically, the Bell number is represented as the sum of *Stirling numbers* of the second kind [27]:

$$B_n = \sum_{k=1}^n S(n, k) = \sum_{k=1}^n \left( \frac{1}{k!} \sum_{j=1}^k (-1)^{k-j} \binom{n}{k} j^n \right). \quad (1)$$

The Stirling number  $S(n, k)$  is the number of ways to partition a set of cardinality  $n$  into exactly  $k$  nonempty subsets. Figure 4 depicts the lattice-shaped query mesh search space for a set of training tuples of size  $n = 4$  and the two examples of two different partitions with 2 and 4 routes respectively. The total number of different partitions here equals 15 ( $B_4 = 15$ ).

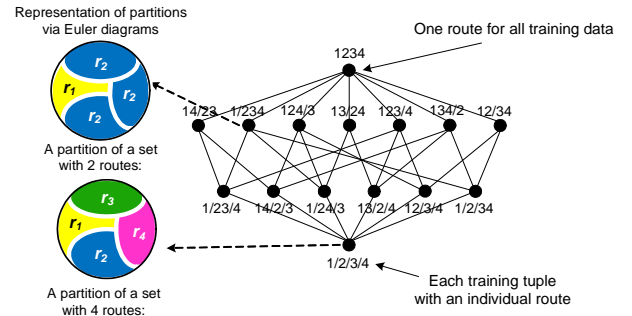


Fig. 4. Lattice-shaped query mesh search space.

### C. Query Mesh Optimizer Sub-problems

In this section, we first characterize the two sub-problems of query mesh selection. We then proceed with the cost model and the search algorithms used by the *QM*-based optimizer.

**Classifier Selection Sub-problem:** One of the sub-components of *QM* is the classifier, which is computed based on the existing routes and the training data attribute values. There is a wide range of classifiers available in the literature [19], each with its strengths and weaknesses. Determining a suitable classifier for a given problem is still more an art than a science. This is due to the fact that

<sup>5</sup>For brevity, we denote  $\{\{1\}, \{2,3\}\}$  as “1/2/3”.

classifier performance and quality depend greatly on the characteristics of the data to be classified [21].

In our work, we employ a *decision tree* (DT) for the classifier component of query meshes. DT is attractive for the following reasons [28]: (1) Complex decisions can be approximated by the conjunction of simpler local decisions at various levels of the tree. (2) In contrast to other classifiers, where each data tuple is tested against all classes, thereby reducing efficiency, in a DT classifier, a data tuple is tested against only certain subsets of test conditions, thus eliminating unnecessary computations. As we require, our DT-based classifier thus is very efficient, because most tuple features are deterministic and often common to a group of tuples.

The algorithm for the decision tree induction has the following steps. The tree is constructed in a top-down recursive divide-and-conquer manner. At the start, all training tuples are at the root. Training tuples are partitioned recursively by the tree induction algorithm based on selected test attributes. Test attributes are selected on the basis of a common in machine learning statistical measure, called *information gain* [19]. The information gain  $I(Y, X)$  for a given data tuple attribute  $X$  with respect to the target route  $Y^6$  is the reduction in uncertainty about the value of  $Y$  when we know the value of  $X$ . The uncertainty about the value of  $Y$  is measured by its *entropy*  $H(Y)$ . The uncertainty about the value of  $Y$  when we know the value of  $X$  is given by the *conditional entropy* of  $Y$  given  $X$ ,  $H(Y|X)$ .

$$I(Y, X) = H(Y) - H(Y|X). \quad (2)$$

When  $Y$  and  $X$  are discrete variables that take values in  $\{y_1 \dots y_k\}$  and  $\{x_1 \dots x_l\}$ , then the entropy of  $Y$  is given by:

$$H(Y) = - \sum_{i=1}^{i=k} P(Y = y_i) \log_2(P(Y = y_i)). \quad (3)$$

The conditional entropy of  $Y$  given  $X$  is:

$$H(Y|X) = - \sum_{j=1}^{j=l} P(X = x_j) H(Y|X = x_j). \quad (4)$$

If the predictive variable  $X$  is not discrete but continuous then in order to compute its information gain with respect to the route identifier attribute  $Y$  we consider all possible attributes,  $X_\theta$ , that arise from  $X$  when we choose a threshold  $\theta$  on  $X$ .  $\theta$  takes values from all the values of  $X$ . Then the information gain is simply:

$$I(Y, X) = \arg \max_{X_\theta} I(Y, X_\theta). \quad (5)$$

For more details on the above metrics, we refer the reader to [19]. Conditions for stopping tree growth are: (1) all training tuples for a given node belong to the same route, (2) there are no remaining attributes for further partitioning, then *majority voting* [29] is employed for classifying the leaf, or (3) there are no training tuple samples left. The leaf nodes of the decision

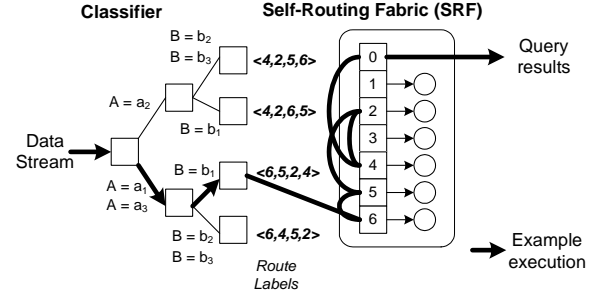


Fig. 5. Example of  $QM$ -based execution.

tree contain the labels of the routes to be used for processing of the tuples that reach those leaf nodes after classification.

**Route Selection Sub-problem:** The second sub-component of  $QM$  solution is the set of execution routes, i.e., the multi-route configuration. Let  $O = \{op_1, \dots, op_n\}$  be the set of operators in a query, where  $op_i \in O$  ( $1 \leq i \leq n$ ) is  $\sigma$ ,  $\pi$  or  $\bowtie$  operator. A route  $r_i$  denotes an operator ordering,  $r_i = \langle op_1, \dots, op_n \rangle$ . The computation of a single best route for set of data is viewed as a “black box” in  $QM$  framework. That is, the optimizer invokes an existing procedure to compute a route for a given data subset based on its statistics using any of the state-of-the-art techniques [30]. For example, similar to [31], the best sequence of operators can be determined by ordering operators in an increasing order of operator *rank*, where the rank of an operator  $op_i$  is defined as  $rank(op_i) = \frac{c(op_i)}{1-s(op_i)}$ , where  $c(op_i)$  is the cost of operator  $op_i$  and  $s(op_i)$  is its selectivity. Alternatively, the optimizer can use common dynamic programming [32] or transformation-based [18] solutions.

Putting all of the above together, Figure 5 illustrates an example of a query mesh solution, where an example of classification with subsequent route execution for a data subset is depicted by a thick black arrow.

#### D. Query Mesh Cost Model

Next, we describe the cost model used by the  $QM$ -based optimizer to compare query mesh solutions when searching for the best  $QM$ . The cost of a  $QM$  consists of three main components described below:

(1) **Cost of routes:** Each execution route  $r_i$  composed of operators has a per-tuple cost  $c(r_i)$  to process a tuple using that route.  $c(r_i)$  represents the expected time to process a single tuple to completion, meaning either to output the tuple as a result or to drop it using  $r_i$ . The cost of  $r_i$  is commonly calculated using two quantities: (i) *cost of operator*, which represents a per-tuple cost of  $op_i$ , and (ii) *selectivity of operator*, which is defined as the fraction of tuples that are expected to satisfy  $op_i$ .

(2) **Cost of classification:** Since each arriving tuple must be processed by the classifier first, the classification cost must be included in the overall execution cost. The classification cost of decision tree (DT) classifier is the cost of traversing the decision tree to reach a leaf node, denoted as  $c(DT|r_i)$ , i.e., the cost of a DT traversal from the root to the leaf node with a label for the route  $r_i$ . The cost is a function of the number

<sup>6</sup>The targets in the context of our work are the execution routes. The value of the target attribute  $Y$  represents a route label (illustrated Figure 5).

of nodes in the path, combined with the cost of computation at each test node of the classifier.

**(3) Multiple routes overhead:** Maintaining multiple execution routes introduces system overhead such as memory, processing and scheduling costs. For simplicity of presentation, we abstract all overhead associated with maintaining a route into a single parameter, denoted by  $c(r_{ovh})$ .

The total cost of the query mesh  $cost(QM)$  thus is:

$$cost(QM) = \sum_{p=1}^{k=|P|} f_p * [c(DT|r_p) + c(r_p|p)] + k * c(r_{ovh}).$$

where  $p$  represents a different subset in a partition  $P$ ,  $k$  is the number of routes in the  $QM$ ,  $k = |P| \leq |T|$ ,  $f_p$  is the expected fraction of tuples from the training dataset  $T$  to be processed by a particular route  $r_p$ , and  $c(r_p|p)$  is the cost of the route for the subset  $p$ .

#### E. Optimal Query Mesh Search Algorithm

As a baseline, we now introduce *Opt-QM* algorithm which is guaranteed to find the optimal query mesh solution (Figure 6). *Opt-QM* traverses all the points in the lattice-shaped search space (Figure 4) starting from the extreme where training tuples representing all possible data subsets have individual routes, to the other extreme where all data is processed using a single route. *Opt-QM* applies a two-step process. First, *Opt-QM* computes the *power set*  $P(T)$  for the given training dataset  $T$ . The power set of  $T$  is the set of all subsets of  $T$ . Given the power set  $T$ , the algorithm estimates the route execution cost for each subset in  $P(T)$  using its statistics. Second, the algorithm puts together partitions composed out of these subsets and aggregates their costs to derive the overall cost of  $QM$ . The query mesh with the smallest cost out of all possible solutions is returned as a result. By using the two-step approach above, the computations can be re-used and the results memoized.

**Complexity Analysis:** The complexity of *Opt-QM* is  $O(B_n * E)$ , where  $B_n$  is the *Bell number* (see Section III-B) and represents the upper-bound of all possible partitions and  $E$  is the time complexity of the algorithm used to find the execution route. The complexity  $E$  depends on the actual algorithm employed by the optimizer for route computation, e.g.,  $E = O(n2^n)$  for dynamic programming [33], or  $E = O(n^2)$  for rank-based ordering algorithm [31], [34]. Clearly with large training datasets, *Opt-QM* algorithm is not scalable in practice. The problem of finding optimal routes alone is already known to be *NP-hard* [31]. By adding a multi-route factor, we increase the complexity of the problem further. Consequently, both the exponential running time and the space requirements provide a strong motivation to design efficient search heuristics.

### IV. QUERY MESH SEARCH HEURISTICS

#### A. Main Idea

We propose a series of cost-based heuristics that guarantee to find a good  $QM$  solution in reasonable time without exhaustive enumeration of the search space. The heuristics have the following three main steps:

---

#### Algorithm *Opt-QM*

**Input:**  $T =$  training dataset  
1: form a *power set*  $P(T)$  based on the set of tuples  $T$   
2: for each set  $S \in P(T)$   
3:   compute  $stats(S)$   
4:   compute  $best\ route(S)$  using  $stats(S)$   
5: **repeat**  
6:   put together a *partition*( $T$ ) out of several sets (from steps 1-4) and construct a query mesh  $QM$  solution (classifier and routes)  
7:   **if** ( $cost(QM)$  is the smallest so far) **then**  
8:      $bestQM = QM$   
9:   **else**  
10:    discard  $QM$   
11:   **end if**  
12: **until** (all possible partitions  $B_n$  have been enumerated)

Fig. 6. Optimal  $QM$  search algorithm

---

- 1) A **start  $QM$  solution** is chosen and its cost  $cost(QM)$  is computed. The starting  $QM$  is set as the best solution considered so far, i.e.,  $bestQM = QM$ .
- 2) A **search strategy** is iteratively applied to traverse the query mesh search space to find another solution  $QM'$ .
- 3) The cost  $cost(QM')$  is computed and compared to the cost of the  $bestQM$  found so far. If  $QM'$  has a smaller cost, the  $bestQM$  is replaced with  $QM'$ . Steps 2-3 are repeated until a **stop condition** is reached.

Next, we propose different strategies to address the following questions: (1) How to pick a promising *start solution*  $QM$ ? (2) How to *improve* the start solution by employing an effective *search strategy*? (3) Finally, when should the search for the best  $QM$  terminate, i.e., what should be the *stop condition*?

#### B. Selecting a Start Solution

Selecting a good *start solution* is essential for  $QM$  quality when using a search heuristic. A search algorithm typically performs walks in the solution space via a series of moves. The number of moves is limited. Hence, if a poor start solution is chosen, the search algorithm might not be able to reach a good quality  $QM$ . In this section, we propose several approaches.

**Content-Driven:** The content-driven approach first groups training tuples based on the similarity of their content<sup>7</sup>. The query mesh optimizer may partition continuous domain data based on the pre-defined thresholds (e.g., in the form of simple ranges) that define how “close” the training tuples are to one another based on their content. Such scheme is similar in spirit to the content-based partitioning technique in CBR [3]. The motivation is that similar content means similar selectivities and thus the same preferred route. Then the best route is computed for each group. Based on the tuple groups and their routes, a decision tree is computed to complete the  $QM$  solution.

**Route-Driven:** The route-driven method first computes the routes for each of the training tuples separately. Thereafter the tuples are *grouped-by* their respective routes, thus forming groups composed of *route-equivalent* tuples. Lastly, the decision tree induction is performed over the route-equivalent groups of tuples. This method is the reverse of the *Content-Driven* approach. The motivation behind this approach is that

<sup>7</sup>Since the training tuples already represent the summaries of their respective clusters of real data tuples, this step resembles hierarchical clustering.

tuples with different content may still share the same best route.

**Other:** Other approaches include *Random-Pick*, where the start solution is a query mesh with the smallest  $cost(QM)$  out of  $x$  randomly selected solutions. Other possible start solutions may include *Extreme-N-Routes*, where every training tuple representing a cluster of real sampled tuples has its own unique route (the bottom of the lattice in Figure 4), or *Extreme-1-Route*, where all training tuples, thus all data, have the same route (the top of the lattice in Figure 4). The latter corresponds to a single plan execution strategy, just like in traditional query optimization systems.

### C. Selecting a Search Strategy

For a query mesh *search strategy*, we adopt and adapt two well-known randomized search algorithms: Iterative Improvement and Simulated Annealing. Both guarantee to find a good  $QM$  solution in reasonable amount of time [35], and thus are viable alternatives to the exhaustive query mesh search.

**Iterative Improvement  $QM$  (II- $QM$ ):** Figure 7 depicts the iterative improvement  $QM$  algorithm pseudo-code. The inner loop of *II- $QM$*  is called a local optimization. A local optimization starts at a random state and improves the solution by repeatedly accepting random downhill moves (i.e.,  $QMs$  with decreasing costs) until it reaches a local minimum. *II- $QM$*  repeats these local optimizations until a stop condition is met. Then it returns the local minimum with the lowest cost found. As time approaches  $\infty$ , the probability that *II- $QM$*  will visit the global minimum approaches 1 [35]. However, given a finite amount of time, the algorithm’s performance depends on the start solution, the cost model and the connectivity of the search space determined by the neighbors of each state.

---

#### Algorithm II- $QM$

```

Input:  $bestQM$  - start solution query mesh
1: while (not stop condition) do
2:    $QM$  = start solution (e.g., chosen at random, or using heuristics from Section IV-B)
3:   while (not local minimum( $QM$ )) do
4:      $QM'$  = random solution in  $NEIGHBORS(QM)$ 
5:     if ( $cost(QM') < cost(QM)$ ) then
6:        $QM = QM'$ 
7:     end if
8:   end while
9:   if ( $cost(QM) < cost(bestQM)$ ) then
10:     $bestQM = QM$ 
11:   end if
12: end while
13: return  $bestQM$ ;

```

Fig. 7. Iterative improvement  $QM$  search strategy

**Simulated Annealing  $QM$  (SA- $QM$ ):** In simulated annealing (SA- $QM$ ), initially the “temperature”  $\tau$  parameter is set to high. Thus a great deal of random movement in the search space is tolerated. Later the “temperature” parameter is lowered, and thus less and less random movement is allowed, until the solution settles into a final “frozen” state. This allows the heuristic to sample the solution space widely when the “temperature” is high, and then gradually move towards simple steepest ascent/descent as the “temperature” cools. Thus the search can move out of local optima during the high temperature phase. The SA- $QM$  algorithm accepts a worsening

move with a certain probability. This probability declines as  $\tau$  declines, by analogy the randomness in the movements decreases as the temperature falls. When  $\tau$  is small enough the algorithm accepts only the improving moves. Figure 8 sketches the pseudo-code for SA query mesh search.

---

#### Algorithm SA- $QM$

```

Input:  $bestQM$  - start solution query mesh
1: Start-up:
2:  $QM = bestQM$ 
   Choose an initial (high) temperature  $\tau > 0$ 
   Choose a value for  $\rho$ , the rate of cooling parameter
3: Choose a random neighbour of  $QM$  and call it  $QM'$ 
   Calculate the cost difference in the query meshes:
4:  $\delta = cost(QM') - cost(QM)$ 
   //Decide to accept the new query mesh or not
5: if ( $\delta \leq 0$ ) then
6:    $QM = QM'$  // $QM'$  is better than or same as  $QM$ 
7: else
8:    $QM = QM'$  with probability  $e^{-\frac{\delta}{\tau}}$ 
9: end if
10: if (stop condition is met) then
11:   exit with  $QM$  as final solution
12: else
13:   reduce temperature by setting  $\tau = \rho * \tau$ , and go to Step 3
14: end if

```

Fig. 8. Simulated annealing  $QM$  search strategy

---

### D. Selecting a Stop Condition

The *stop condition* largely depends on the search strategy employed. The query mesh search may stop when either  $k$  iterations have gone by (e.g., in *II- $QM$* ), or the solution did not improve in the last several rounds (e.g., in *SA- $QM$* ) indicating that the search process has reached a plateau. Alternatively, the search can be time-bounded or resource-bounded, e.g., when memory or CPU utilization limits are set.

## V. QUERY MESH EXECUTOR

### A. Query Mesh Execution Infrastructure Overview

The query mesh executor takes the  $QM$  solution computed by the optimizer and instantiates the  $QM$  runtime infrastructure. The runtime system consists of the *online classifier operator* and the query operators which are instantiated inside the *Self-Routing Fabric (SRF)* infrastructure (Figure 3). When new tuples arrive, they first get processed by the online classifier operator to determine the routes that would be used for their processing. Thereafter, the tuples are forwarded into the *SRF* for the actual query evaluation according to their routes.

### B. Forming Routable Tuple Clusters

Arriving tuples are classified into “routable clusters” – groups of tuples with similar routes, using a *classification window*  $W^{TC}$ , where  $W^{TC}$  is a *tumbling window* [36]. We use a tumbling window, because it partitions a stream into non-overlapping consecutive windows, so that a tuple is classified only once. If tuples within a time window are known to be correlated, then classification overhead can be minimized by classifying one tuple per window and then sending the rest of the tuples on the same route as the classified tuple. We denote a set of tuples that due to classification are assigned to the same route a *routable cluster*, or short “*ruster*”.

**Definition 5.1: (Ruster)** Let  $S_i$  be a data stream. A *routable cluster*  $RC$  is a window-bounded set of data tuples  $\{s_1 \dots s_k\} \subseteq S_i[W_i^{TC}]$  assigned to the same route  $r_i$  by the classifier.

Tuples in the ruster have timestamps in the time interval  $[RC.ts - W_i^{TC}, RC.ts]$ , where  $RC.ts$  is the time of tuple classification and  $W_i^{TC}$  is the size of the classification window.

The pre-computed route for a ruster is stored in a route token (or short  $r$ -token).  $R$ -tokens are metadata tuples, similar in spirit to streaming punctuations [37], embedded inside data streams, thus partitioning the data tuples into rusters. The distinct characteristics of  $r$ -tokens include: (i)  $r$ -tokens are “self-describing” as they carry routing instructions for streaming data to convey to query operators, (ii)  $r$ -tokens precede the data tuples they are applicable to, and (iii) routes in the  $r$ -tokens are specified in the form of an operator stack based on the design of the  $SRF$ , which we describe next.

### C. Self-Routing Fabric and Route Encoding

$SRF$  has the following two components (see Figure 3):

- **Operator Index Array (OI-array):**  $OI$ -array stores the pointers to the operators’ input queues. Each index  $i$  corresponds to a unique operator  $op_i$ . Index “0” is reserved for the  $SRF$  global output queue, where the result tuples are placed to be sent to the applications.
- **Operator Modules (Op-modules):** Operator modules are the actual operators processing the tuples. We focus on select, project and join queries in this paper. *Selection* and *projection* operators are simple: when an operator receives an input ruster, it filters the tuples based on the selection predicate or projects out unwanted tuple attributes. For joins, we have designed *one-way-join-probe* operators, similar in spirit to *SteM* operators [16], which essentially correspond to a half of a traditional join operator. Such operators are formed over a base stream, supporting the *insert* (build), *search* (probe), and *delete* (eviction) operations for window purging. Such *SteM*-like operators eliminate the burden of state management, when different routes are executed concurrently. For more details, we refer the reader to [16].

**Example:** Consider an  $SRF$  with the operator index array as follows:  $OI$ -array[1] =  $op_i$ ,  $OI$ -array[2] =  $op_j$ ,  $OI$ -array[3] =  $op_k$ , and  $OI$ -array[4] =  $op_l$ . Then a route  $r = \langle op_j, op_k, op_i, op_l \rangle$  will be encoded in an  $r$ -token as a stack  $\langle 2, 3, 1, 4 \rangle$ , where ‘2’ is the first operator in the route and ‘4’ is the last. The items in the stack represent the indexes of the operators in the  $SRF$ . A ruster is always routed to the operator that is currently the top node in the routing stack. After an operator is done processing the ruster, the operator “pops” its index from the top of the routing stack in the  $r$ -token, and then puts the ruster into the next (now the top) operator’s queue. If all tuples from a ruster are dropped by an operator  $op_i$ , then the ruster is not processed any further and its  $r$ -token is discarded. When the  $r$ -token operator stack is empty, the ruster tuples are forwarded to the global output queue reserved by the index “0” and thus to the application(s).

The novelty of the  $SRF$  infrastructure is in that it completely eliminates a central router operator, thus removing a “back-flow” bottleneck problem present in the multi-route systems [8]. Another key advantage (although not explored in this

paper) is that the  $SRF$  infrastructure makes  $QM$  adaptivity a very inexpensive process, since routes operator pipelines are not physically created like in traditional query execution plans. If a  $QM$  solution needs to change, the only thing that needs to be modified is the  $DT$  classifier without affecting the rest of the execution infrastructure.

## VI. EXPERIMENTAL STUDY

We have implemented our  $QM$  approach in the DSMS CAPE [38] written in Java with Java 1.6.0.0 runtime, running on Windows Vista with Intel(R) Core(TM) Duo CPU @1.86GHz processor and 2GB of RAM. The objective of our experimental study is twofold: (1) to compare our  $QM$  model with traditional single-plan and multiple (route-less) plan approaches in terms of output rate and overhead, and (2) to examine the  $QM$  optimization costs and its possible tuning choices. In our experiments, we used both synthetic and real-life datasets described below:

**Stocks-News-Blogs-Currency dataset:** We have implemented a Web application that continuously collects NYSE stock prices, currency exchange rates (provided via webservice by Yahoo Finance), news and blogs from different geographic regions and on different subjects (provided via RSS feeds).

**Lab dataset:** This dataset contains readings from sensors in the Intel Research, Berkeley Lab. The original dataset [3] consists of a single stream sensor readings. We have partitioned this dataset by sensor locations into several streams. The sensor readings are sent to CAPE in generation order, as they would if the tuples were collected from the sensors in real-time.

**Synthetic dataset:** We have also implemented a synthetic stream generator to produce tuples with a variety of parameters including skewness, selectivity, etc. The generator takes as parameters the number of streams to generate, the number of columns in the schema of the streams and the number of tuples in each stream. A desired number of content-based partitions for a tuple attribute, their skewness, selectivity and correlation to other column(s) can additionally be specified.

Our experiments use  $N$ -way equi-join queries which join incoming tuples from  $N$  streams  $S_1 \dots S_N$  of the form: `SELECT * FROM S1, S2, S3, ... SN WHERE S1.col1 = S2.col1 and ... SN-1.col2 = SN.col1`.  $N$ -way join query is one of the core queries in database systems and can be used to discover correlations across different sources. The sliding windows in the queries are based on the timestamps present in the data (as opposed to the clock times when tuples arrived to the system). In this way, we ensured that the query answers are the same regardless of the rate at which the dataset is streamed to the system or the order of tuple processing. Due to lack of space, the majority of results presented in this section involve synthetic dataset. However, similar trends were observed with real-life data as well. The query is an equi-join of 10 streams, i.e.,  $S_0 \bowtie S_1 \dots S_9 \bowtie S_{10}$ . Unless otherwise stated, the default values used in the experiments are the ones listed in Table I.

The  $QM$  execution framework is compared against alternative execution models, namely the single route-oriented (SRO), the multi-route-less (MRL) and the content-based routing

TABLE I  
DEFAULTS USED IN THE EXPERIMENTS.

Parameter	Value	Description
Arrival Distribution	Poisson	Data arrival distribution
$\mu$	500 msec	Mean inter-arrival rate
$ T_{dq} $	1,000	# of tuples dequeued by an operator at a time for processing
$p$	3-10	# of skewed partitions per stream
$ T $	100 tuples	Size of training tuple set (sample size = 1,000 tuples)
$H$	SA	Default heuristic used to find query meshes
Start Solution	Route-Driven	Query mesh start solution strategy
$W^{TC}$	1,000 tuples	Classification window size
Ruster size	100 tuples	Average ruster size

(CBR) [3] solutions<sup>8</sup>. All four systems were implemented inside CAPE. We have used a multi-way join operator (short MJoin) [39] as a representative of SRO. MJoin is a generalization of symmetric binary join algorithms, providing the best plan for each stream, and thus it became our choice for a single route-oriented solution. In SRO, data from a stream follows the same best route computed based on the overall statistics. For MRL, we used Eddy execution framework with lottery routing policy [7]. Lottery routing is one of the better routing policies as it favors not only fast operators but also operators with low selectivity. CBR was implemented as an extension to the Eddy system based on the original paper [3]. We use a Round-Robin scheduler in all four systems, which cycles over the list of active operators and schedules the first operator ready to execute. When scheduled, an operator runs for a fixed amount of time bounded by  $|T_{dq}|$ , the number of tuples that an operator may dequeue from its input queue in each execution epoch. Round-Robin was chosen as it has a desirable property of avoiding starvation – no operator with tuples in its input queue goes unscheduled for an unbounded amount of time.

#### A. Experimental Results

**Total Number of Tuples Produced:** Figure 9(a) shows the total number of result tuples produced by the four execution models over time. We ran the query processor using different solutions for over an hour and show the average output for the first 60 minutes. We observe that over time the total number of tuples outputted by  $QM$  is significantly larger than by alternative solutions. In SRO, single plan optimization coarseness leads to producing a lot of intermediate results filling up the queues and hindering the performance of the system. Whereas the multi-routeless system suffers from the “backflow” overhead, where tuples get continuously routed back to the central router operator that has to re-examine the tuples and forward them to the next operator for processing. CBR suffers from the same overhead as the MRL, plus the overhead of continuous re-learning. However, through better routing policy it can achieve better performance than MRL. Figure 9(b) shows the total number of tuples produced using the real-life datasets. Although not by as much as in the synthetic dataset experiment, still the trend is similar – query mesh outperforms the other three alternatives.

<sup>8</sup>In the rest of this section, we will refer to these alternative solutions using their abbreviated names, SRO, MRL and CBR, respectively.

**Average Output Rate:** Figure 9(c) illustrates the comparison of the average output rates during different time intervals for the four execution models. We have computed this statistic as follows: we partitioned the execution time into non-overlapping time intervals, each 10 minutes long. For each time interval, the average output rate was computed, and then we took the average of the output rates for all execution intervals. The output rate is  $R = N / T$ , where  $R$  is the output rate,  $N$  is the total number of result tuples produced during the interval and  $T$  is the length of the time interval (in min). Each bar plotted in Figure 9(c) represents the average output rate value over a particular time interval, and the last bar is the overall average. To prevent the average value from being skewed, we have removed the warm-up time interval (the first 10 minutes). Figure 9(c) shows a solid trend that  $QM$ ’s output rate on average can be up-to 60-100% higher than that of SRO, up-to 10-40% higher than that of MRL and 8-28% than that of CBR. This trend was confirmed by multiple runs of the same query on different datasets.

**Memory Utilization:** Figure 9(d) shows the memory overhead by the three execution models. Memory cost (measured in MB) was periodically computed as follows:  $UM = TM - FM$ , where  $UM$  is the amount of memory used,  $TM$  is the total amount of memory in the system and  $FM$  is the amount of free memory in the Java Virtual Machine, respectively. Figure 9(d) shows that SRO model has the least memory overhead. This is expected, as all tuples follow the same route, so there is no extra multi-route overhead. The memory overhead in MRL and CBR is associated with the bitmask attached to each tuple to serve as its lineage, plus the delay in processing (due to the backflow back to Eddy’s queue) which may contribute to the increase in memory requirements and a few additional data structures in CBR. The memory overhead in  $QM$  is due to the presence of the routing tokens to store the *rusters*’ routes. The smaller the parameter limiting the number of tuples per *ruster*, the larger the number of *r-tokens* in the streams. But as can be seen, the percent of overhead in  $QM$  is still not noticeably larger compared to the SRO (between 5-16%) and is significantly smaller than that of MRL and CBR (between 10-38%).

Next we describe a set of experiments measuring  $QM$ -specific costs.

**Overhead of Runtime Classification:** We have evaluated the overhead of the online classifier relative to the overall query execution cost. Figure 9(e) shows the classifier overhead for 4-way, 6-way, 8-way and 10-way join queries. As can be seen, online classification has a very low relative overhead ranging from 2% for a 10-join query up to 4% for a 4-join query. We have observed that the classifier tends to be small in height (maximum 2-3 levels high) and  $DT$  traversal is thus quick and cheap. Additional system overhead of the online classifier corresponds to scheduling an additional operator by the scheduler. Since the processing of the tuples by the classifier is fast, the operator, when scheduled, completes its work very quickly giving majority of the execution time to other operators.

**Effect of Classification Window:** Figure 9(f) shows the effect

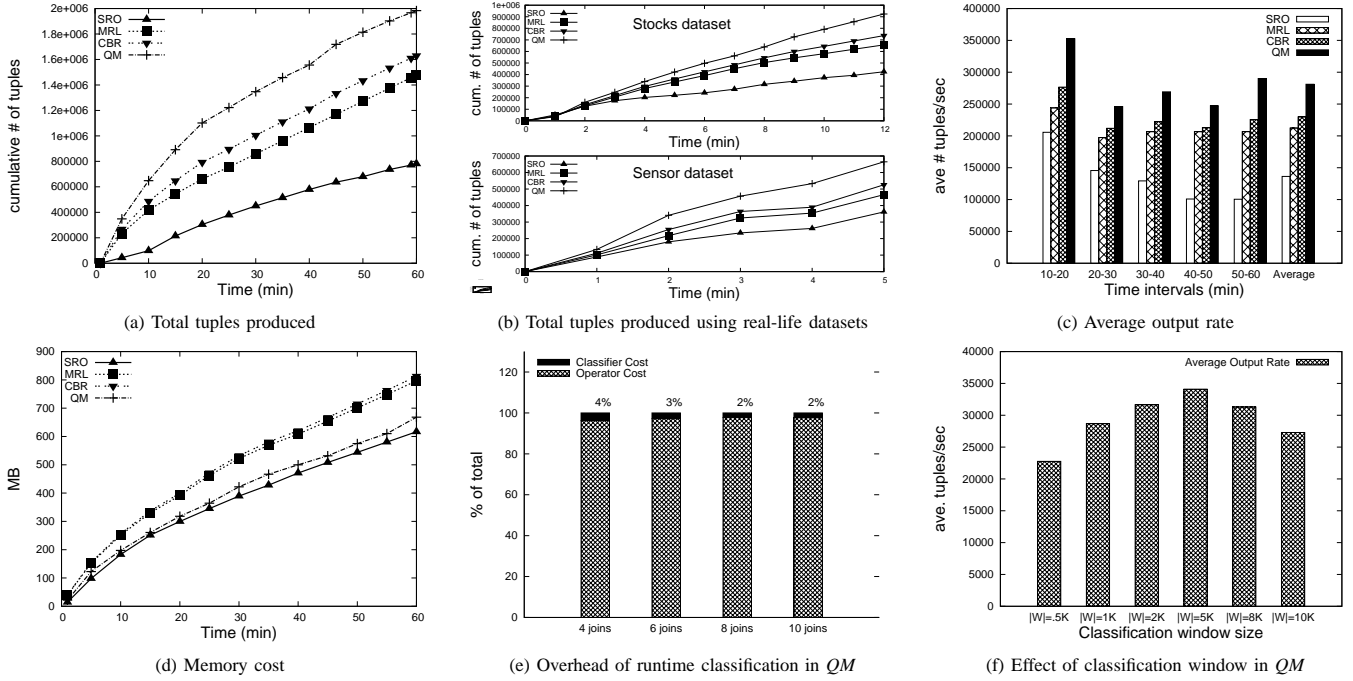


Fig. 9. Experimental results.

of the size of the classification window on the output rate of the  $QM$ . The classification window size parameter is one of the tuning choices in the  $QM$  system, since every arriving tuple has to be classified first, before the actual query processing. If the classification window is set to be too small, many tuples would be waiting in the input queue of the online classifier to be classified to be sent to query operators. However, if the parameter is set to be too large, it will increase the time of the classifier execution relative to the actual query execution cost. It would also increase the number of tuples that get forwarded to other operators. Since query operators have a bound on how many tuples they can process per execution epoch, those tuples may be waiting in the operator queues for a long time before they would be processed.

**Effect of Ruster Size:** Figure 10(a) shows the effect of the *ruster* size parameter on the  $QM$  performance. This parameter controls the number of tuples in a *ruster*, i.e., minimum number of tuples that follow an *r-token*. It is also one of the tunable parameters in  $QM$  execution. Not surprisingly, we observe that with smaller *rusters* we tend to add more overhead and reduce the output rate. This is due to the larger number of *r-tokens* present in the streams that take up more memory and CPU resources. Note also that this parameter affects how many of the actual tuples get processed by query operators, when operators get scheduled. The number of *r-tokens* contributes to the total number of tuples that an operator can dequeue (controlled by  $|T_{dq}|$ ). Moreover, a *ruster* size is also bounded by the size of the  $W^{TC}$  and depends on the arriving content distribution of data. After classification, if there are some *rusters* that have a smaller size than the maximum *ruster* size parameter, the operator still sends off these “incomplete” *rusters* for processing without waiting for the next execution epoch.

**Effect of Training Set Size:** In this experiment, we study the effect of the training set size on the cost of the query mesh optimizer and the resulting quality of query meshes. For this purpose, we have varied the training set size from 8 to 1,000 tuples. The sampling method stayed unchanged, only the upper-bound for the training set size was varied. Figure 10(b) (the bottom chart) shows the optimizer search time with various training set sizes using *SA* and *II* search heuristics<sup>9</sup>. Figure 10(b) (the top chart) shows the total number of result tuples produced – our method of measuring  $QM$  performance. The results support our hypothesis that (1) larger training sets increase the  $QM$  optimization cost, and (2) if compact, yet accurate, training sets are chosen the quality of  $QM$  does not suffer, keeping the optimization time very practical. As can be seen, the distribution is accurately depicted by  $|T| = 100$  and its performance is no worse than for  $|T| = 1000$ .

**Effect of Start Solution:** Here, we evaluated the effect of the *start solution* (Section IV-B) on the quality of resulting  $QM$  with heuristic-based searches. We chose tuple output rate as our measure to estimate the quality of query meshes. Figure 10(c) shows the performance for different query meshes, computed when the optimizer employed different *start solution* approaches. As can be seen, all  $QM$ s are pretty close in quality (except for the *Extreme-I* which is a single plan start solution strategy). But still *Route-Driven* approach results in a slightly better  $QM$  that is 6% better than *Extreme-N*  $QM$ , 10% better than *Content-Driven*  $QM$ , 12% better than *Random*  $QM$  and 19% better than *Extreme-I*  $QM$ . It is no surprise that *Extreme-N*'s quality is close to *Route-Driven*'s  $QM$ , as the former is a special case of the latter approach.

**Experimental Conclusions:** The main findings of our exper-

<sup>9</sup>Optimization cost for  $|T| = 1000$  is not shown for the relative visibility of the costs for  $8 \leq |T| \leq 100$ .

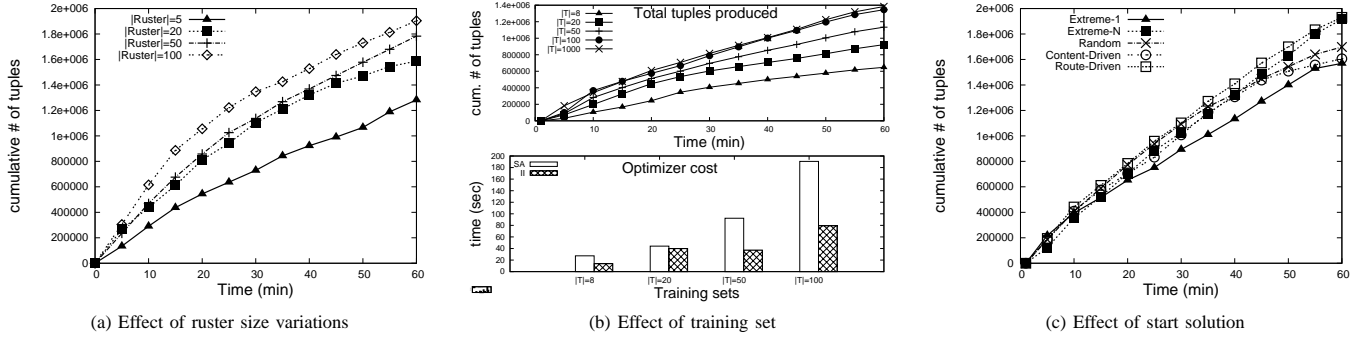


Fig. 10. Experimental results (cont.).

imental study can be summarized as follows:

- 1) *QM* can give up-to 60-100% higher output rate than SRO, up-to 10%-40% higher than MRL and 8%-28% higher than CBR approaches for skewed datasets.
- 2) Memory overhead of query mesh is lower, 10-38% less, than for MRL and CBR solutions and only 5-16% higher than for SRO solution.
- 3) The runtime overhead of classification is very small (2-4%) relative cost to the overall query processing cost. The decision tree probe is fast, and on average only 2-3 test checks are needed to traverse a *DT*.
- 4) *Route-driven* start solution strategy results in higher quality *QMs*.

## VII. RELATED WORK

Query optimization is a well-studied area, with most efforts, however, primarily concentrating on optimizing a *single* plan for all data [31], [40], [41], [42], [39].

Our proposed query mesh model is related to the concept of *horizontal partitioning* [43]. Conceptually, the main idea is to partition data so that different partitions can be processed using different plans. For example, selectivity-based partitioning scheme [4] adopts a divide-and-union approach. A relation is partitioned according to selectivities, and subsequently the query is rewritten as a union of constituent queries over the computed partitions. The approach presented in [4] is orthogonal to *QM*, as it primarily focuses on the partitioning algorithm rather than a complete systematic approach to multi-route query optimization and query processing – the focus of our work. In fact, the selectivity-based partitioning algorithm can be employed by *QM*-based optimizer to find a good start solution in the query mesh search described in Section IV-B.

*Conditional plans* [44] generalize serial plans by allowing different predicate evaluation orders to be used for different tuples based on the values of attributes and the cost of their *acquisition*. Since the main goal is to minimize the communication and acquisition costs in order to minimize the sensor battery consumption, conditional plans primarily focus on selecting a single and very cheap to acquire partitioning attribute. Such attribute is not necessarily the “best” splitting attribute in a more general query optimization context. In that respect, query mesh is a more general model selecting the best splitting attributes (in the classifier) based on data distribution-based measure from machine learning to assign data to routes so that tuples would be discarded as early

as possible. A conditional plan is typically computed on a powerful computer (a basestation) and then appropriate plan is sent to the different sensor nodes in the network. Thus, conceptually, still a single plan strategy is employed locally at each sensor node for execution, which is different from *QM*-based execution approach.

Several techniques from adaptive query processing [23], [36], [45] are related to query mesh. Most adaptive query processing works, however, still focus on adapting a single query plan. *Eddies* [7], which can potentially adapt at the tuple granularity, is observed to mostly be using a single plan for nearly all tuples as was also indicated in [3]. [8] adds batching to the *Eddies* routing to reduce the tuple-level routing overhead which is close in spirit to our *ruster* concept. What differs *Eddy* batching from ours is that in the former, the batching is very naive: every  $k$  tuples, i.e., continuous chunk of tuples that happened to arrive together in time are batched and routed together. In *QM*, the 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.

Related to *QM* is the content-based routing (CBR) extension of *Eddies* [3]. CBR focuses on continuously profiling operators and identifying “classifier attributes” to partition the underlying data into tuple classes that may be routed differently by *Eddy*. The key distinguishing characteristic between *QM* and CBR is that CBR considers only single-attribute classifiers. We take a more general approach in *QM* and build a classifier model that implicitly takes multiple attributes, values’ correlations and statistics into account to identify distinct data subsets with respect to execution routes. Although the authors state in [3] that CBR approach does not require “previous knowledge” of the data, the *gain ratio* metric in CBR is based on a historic profile of an operator which is similar to our approach. Finally, CBR inherits several problems associated with *Eddies*, such as continuous and often unnecessary re-optimization and re-learning overhead. The classifier attributes are re-computed continuously, even though the best classifier attribute for an operator does not change very often [3]. Extending CBR to non-*Eddy*-based systems, i.e., systems that pre-compute plans prior to execution, is non-trivial, as CBR does not compute full routes and instead makes its decisions locally and continuously for each operator. *QM* contribution has a higher significance for two reasons: (1) *QM* has a much

wider scope of applicability as it addresses multi-route query processing in plan-based systems - the standard in database systems, and (2) experimentally, *QM* approach has shown to outperform Eddies and CBR by a substantial margin.

*QM* has some characteristics that are close in spirit to *Parametric Query Optimization (PQO)* [46], where a set of plans appropriate for different situations is found and the decision of which one to use is deferred until runtime. This work differs substantially from ours in three essential ways: First, in PQO, the execution plan is computed for *all* data, thus not exploiting the partitioning of data into distinct data subsets, which we have seen to be prevalent and widely exploitable. Second, in PQO, the choice of an execution plan is typically query parameter-driven rather than data characteristics-driven. Third, PQO does not exploit machine learning to identify the relationships between the data properties and the execution routes as our approach.

### VIII. CONCLUSION

In this paper, we have proposed a multi-route query optimization and execution model, called *Query Mesh (QM)*. *QM* is general and applicable to static DBMSs as well as streaming engines, offering numerous advantages. First, *QM* employs efficient machine learning techniques to learn the relationship between the data and the resulting routes to find the best processing strategy for different subsets of data. Second, we present a complete *QM*-based approach for query optimization and query processing applicable to arbitrary data and queries. Third, *QM*-based query processing uses very efficient multi-route execution infrastructure, which facilitates shared processing and has near-zero route execution overhead.

Our most important contribution was to show that *QM* implemented in a real database system can achieve significant performance improvements over other alternative solutions. Our experimental results demonstrate *QM* potential as a paradigm for efficient query optimization.

Although routes in *QM* are fully computed, the physical operator pipelines are not constructed, which makes the *QM* infrastructure very amenable to adaptivity – the subject of our future work. The problem of adaptive *QM* is orthogonal to the issues addressed in this paper. Here, we have provided the foundation of the *QM* problem and the algorithms to find a good *QM* solution efficiently, without which the adaptive aspect cannot be addressed.

### REFERENCES

- [1] R. Ramakrishnan and J. Gehrke, *Database Management Systems*. McGraw-Hill Higher Education, 2000.
- [2] S. Christodoulakis, "Implications of certain assumptions in database performance evaluation," *TODS*, vol. 9, no. 2, pp. 163–186, 1984.
- [3] P. Bizarro and et al., "Content-based routing: Different plans for different data." in *VLDB*, 2005, pp. 757–768.
- [4] N. Polyzotis, "Selectivity-based partitioning: a divide-and-union paradigm for effective query opt." in *CIKM*, 2005, pp. 720–727.
- [5] N. Mir, "Analysis of nonuniform traffic in a switching network," in *IC3N*. IEEE Computer Society, 1998, p. 668.
- [6] L. Harris, "Stock price clustering and discreteness," *Review of Financial Studies*, vol. 4, no. 3, pp. 389–415, 1991.
- [7] R. Avnur and J. M. Hellerstein, "Eddies: Continuously adaptive query processing," in *SIGMOD*, 2000, pp. 261–272.
- [8] A. Deshpande, "An initial study of overheads of eddies," *SIGMOD Rec.*, vol. 33, no. 1, pp. 44–49, 2004.
- [9] S. Madden, M. Shah, and et al., "Continuously adaptive continuous queries over streams." in *SIGMOD*, 2002.
- [10] Z. G. Ives, A. Halevy, and et al., "Adapting to source properties in processing data integration queries." in *SIGMOD*, 2004, pp. 395–406.
- [11] J. F. Kurose and K. Ross, *Computer Networking: A Top-Down Approach*. Addison-Wesley, 2002.
- [12] "Microsoft sql server. <http://www.microsoft.com/sql/default.mspx>."
- [13] DB2, "<http://www.ibm.com/software/data/db2/>."
- [14] Oracle, "<http://www.oracle.com/index.html>."
- [15] A. Deshpande, J. Hellerstein, and et al., "Lifting the burden of history from adaptive query processing." in *VLDB*, 2004, pp. 948–959.
- [16] V. Raman, A. Deshpande, and et al., "Using state modules for adaptive query processing." in *ICDE*, 2003, pp. 353–365.
- [17] M. Astrahan, M. Blasgen, and et al., "System r: relational approach to database management," *TODS*, vol. 1, no. 2, pp. 97–137, 1976.
- [18] G. Graefe and W. McKenna, "The volcano optimizer generator: Extensibility and efficient search," in *ICDE*, 1993, pp. 209–218.
- [19] T. M. Mitchell, *Machine Learning*. New York: McGraw-Hill, 1997.
- [20] R. Lipton and et al., "Efficient sampling strategies for relational database operations," *Theor. Comput. Sci.*, vol. 116, no. 1, pp. 195–226, 1993.
- [21] D. J. Hand, P. Smyth, and H. Mannila, *Principles of data mining*. Cambridge, MA, USA: MIT Press, 2001.
- [22] B. Efron and R. Tibshirani, *Introduction to Bootstrap*. London: Chapman & Hall/CRC, 1994.
- [23] A. Deshpande, Z. Ives, and et al., "Adaptive query processing," in *Foundations and Trends in Databases*, 2007.
- [24] J. A. Hartigan, *Clustering Algorithms*. New York, NY, USA: John Wiley & Sons, Inc., 1975.
- [25] S. G. et al., "Clustering data streams," in *FOCS*, 2000, p. 359.
- [26] A. K. Jain, A. Topchy, M. H. C. Law, and J. M. Buhmann, "Landscape of clustering algorithms," in *ICPR*, 2004, pp. 260–263.
- [27] M. Klazar, "Bell numbers, their relatives, and algebraic differential equations," *J. Comb. Theory Ser. A*, vol. 102, no. 1, pp. 63–87, 2003.
- [28] P. Swain and H. Hauska, "The decision tree classifier design and potential," in *IEEE Trans. Geosci.*, 1977, pp. 142–147.
- [29] S.-B. Oh, "On the relationship between majority vote accuracy and dependency," *Pattern Recogn. Lett.*, vol. 24, no. 1-3, pp. 359–363, 2003.
- [30] S. Chaudhuri, "An overview of query optimization in relational systems," in *SIGMOD*, 1998, pp. 34–43.
- [31] S. Babu, R. Motwani, and et al., "Adaptive ordering of pipelined stream filters," in *SIGMOD*, 2004, pp. 407–418.
- [32] P. G. Selinger and et al., "Access path selection in a rel. database management system," in *SIGMOD*. ACM, 1979, pp. 23–34.
- [33] K. Ono and G. M. Lohman, "Measuring the complexity of join enumeration in query optimization," in *VLDB*, 1990, pp. 314–325.
- [34] J. M. Hellerstein, "Predicate migration: optimizing queries with expensive predicates," in *SIGMOD*, 1993, pp. 267–276.
- [35] Y. E. Ioannidis and Y. Kang, "Randomized algorithms for optimizing large join queries," in *SIGMOD*, 1990, pp. 312–321.
- [36] D. Carney and et al., "Monitoring streams - a new class of data management applications." in *VLDB*, 2002, pp. 215–226.
- [37] P. A. Tucker, D. Maier, and et al., "Exploiting punctuation semantics in continuous data streams," *TKDE*, vol. 15, no. 3, 2003.
- [38] E. A. Rundensteiner and et al., "Cape: Continuous query engine with heterogeneous-grained adaptivity." in *VLDB*, 2004, pp. 1353–1356.
- [39] S. Viglas and et al., "Maximizing the output rate of multi-way join queries over streaming inf. sources." in *VLDB*, 2003, pp. 285–296.
- [40] Y. E. Ioannidis and et al., "Left-deep vs. bushy trees: An analysis of strategy spaces and its implications for query optimization," in *SIGMOD*, 1991, pp. 168–177.
- [41] R. Krishnamurthy and et al., "Optimization of non-recursive queries," in *VLDB*, 1986, pp. 128–137.
- [42] A. Swami and et al., "A polynomial time algorithm for optimizing join queries," in *ICDE*, 1993, pp. 345–354.
- [43] P. D. Bra and et al., "Horizontal decompositions and their impact on query solving," *SIGMOD Rec.*, vol. 13, no. 1, pp. 46–50, 1982.
- [44] A. Deshpande and et al., "Exploiting correlated attributes in acquisition query processing." in *ICDE*, 2005, pp. 143–154.
- [45] S. Babu, P. Bizarro, and D. DeWitt, "Proactive re-optimization," in *SIGMOD*, 2005, pp. 107–118.
- [46] Y. E. Ioannidis, R. T. Ng, K. Shim, and T. K. Sellis, "Parametric query optimization," in *VLDB*, 1992, pp. 103–114.