Journal of Computer and System Sciences 79 (2013) 312–329

Contents lists available at SciVerse ScienceDirect

Journal of Computer and System Sciences

www.elsevier.com/locate/jcss

Multi-route query processing and optimization

Rimma V. Nehme [a], Karen Works [b,*], Chuan Lei [b], Elke A. Rundensteiner [b], Elisa Bertino [c]

[a] Microsoft Research Lab., WI, United States
[b] Worcester Polytechnic Institute, MA, United States
[c] Purdue University, IN, United States

ARTICLE INFO

Article history:
Received 4 April 2011
Received in revised form 18 January 2012
Accepted 11 September 2012
Available online 22 October 2012

Keywords:
Data stream database systems
Adaptive query processing

ABSTRACT

A modern query optimizer typically picks a *single* query plan for all data based on overall data statistics. However, many have observed that real-life datasets tend to have non-uniform distributions. Selecting a single query plan may result in ineffective query execution for possibly large portions of the actual data. In addition most stream query processing systems, given the volume of data, cannot precisely model the system state much less account for uncertainty due to continuous variations. Such systems select a single query plan based upon imprecise statistics. In this paper, we present "*Query Mesh*" (or *QM*), a practical alternative to state-of-the-art data stream processing approaches. The main idea of *QM* is to compute multiple routes (i.e., query plans), each designed for a particular subset of the data with distinct statistical properties. We use terms "plans" and "routes" interchangeably in our work. A *classifier* model is induced and used to assign the best route to process incoming tuples based upon their data characteristics. We formulate the *QM* search space and analyze its complexity. Due to the substantial search space, we propose several cost-based query optimization heuristics designed to effectively find nearly optimal *QM*s. We propose the *Self-Routing Fabric* (*SRF*) infrastructure that supports query execution with multiple plans without physically constructing their topologies nor using a central router like Eddy. We also consider how to support uncertain route specification and execution in QM which can occur when imprecise statistics lead to more than one optimal route for a subset of data. Our experimental results indicate that *QM* consistently provides better query execution performance and incurs negligible overhead compared to the alternative state-of-the-art data stream approaches.

1. Introduction

1.1. Single versus multiple execution plans

Most modern query optimizers determine a *single* "best" query plan at compile time. Their objective is to find the *query plan* with the cheapest execution cost, which typically is estimated based on the average statistics for all data tuples. Since the cost is estimated using the statistics of the data as a whole, it largely relies on the uniformity of attribute values. Significant statistical variations of different subsets of data may lead to overall poor query execution performance. The main drawback here is that such a "monolithic" approach misses important opportunities for effective query optimization [1,2], as none of the individual tuples may be served well by such an "average" plan.

* Corresponding author.
E-mail addresses: rimman@microsoft.com (R.V. Nehme), kworks@cs.wpi.edu (K. Works), chuanlei@cs.wpi.edu (C. Lei), rundenst@cs.wpi.edu (E.A. Rundensteiner), bertino@cs.purdue.edu (E. Bertino).

0022-0000/$ – see front matter
http://dx.doi.org/10.1016/j.jcss.2012.09.010

Motivating example. Consider monitoring stocks that exhibit "bullish" patterns and appear in recent news and "street research", e.g., blogs, web sites, etc. See the query below.

```
SELECT S.company_name, S.symbol, S.price
FROM Stock as S, News as N, StreetResearch as SR
WHERE matches(S.data, BullishPatterns)              /*op1*/
AND contains(S.sector, News[1 hour])                /*op2*/
AND contains(S.company_name, StreetResearch[3 hours]); /*op3*/
WINDOW 60 seconds
```

The lookup table *BullishPatterns* contains "bullish" patterns of stock behavior, e.g., "symmetrical triangle" [3]. Operator op_1 performs a similarity-based join on the latest financial data (i.e., incoming stock data tuples from the last 60 seconds) with the *BullishPatterns* table. Operators op_2 and op_3 perform the matches on the stock's sector and the company name with the news and street research data. Here c_i and δ_i respectively denote the current processing cost per tuple and the current selectivity of op_i respectively.

Suppose it is a bullish market (i.e., stocks are doing well) and the following conditions hold: $c_1 > c_2 > c_3$ and $\delta_1 > \delta_2 > \delta_3$. Given these statistics, the best query processing order is op_3, op_2, op_1. Now suppose the news reports poor crop harvests. This may hurt companies in the agricultural sector. The price of a company's stock is often driven by the health of an industry and may change due to the reaction of investors towards a given industry. Thus for agricultural sector stocks the news of a poor harvest will likely result in fewer matches with the *BullishPatterns* table and with more references in the news and blogs. In this case, δ_1 will be relatively lower than δ_2 and δ_3 for such data tuples. So, op_1, op_2, op_3 may now become the most efficient ordering for processing agricultural sector stock tuples. However, other sectors may remain unaffected, and hence for them op_3, op_2, op_1 will remain the best ordering. Query execution performance would be significantly limited if the system continues to use the overall statistics across all data and processes all data using the same single plan, i.e., op_3, op_2, op_1.

1.2. Additional real-life examples

Some other real-life examples where different subsets of data exhibit rather distinct statistical properties include:

Stock market data. As demonstrated by the above example, stock market data is known to be not uniform [4]. The prices of stocks quickly reflect information concerning current events and future expectations. While some stocks may go up, others may go down as a result of the same event.

Internet and communication networks. Network traffic tends to vary depending on its destination or type, e.g., voice or multimedia. Some destinations may be more popular than others, e.g., certain web sites get higher visitation rates. Different network traffic may also have different characteristics. In the case of a network congestion, traffic packets will be discarded by routers with different probabilities. Clearly, utilizing a single plan may lead to highly inefficient query processing for some or possibly huge fractions of the actual data.

Scientific and environmental data. Most earth science data has a non-uniform spatial distribution, that is, similar values are found in neighboring areas, while significantly different values often exist in distant areas. In addition, the phenomena of interest are not uniformly distributed [5].

1.3. Query optimization approaches

We now classify the state-of-art query optimization techniques along two dimensions namely, *optimization time* and *optimization granularity*. Optimization time denotes to when optimization decisions are made. While optimization granularity defines if the optimization decisions are based upon optimizing 1) all tuples, 2) groups of tuples, or 3) individual tuples processed. Fig. 1 provides an intuition of where our solution fits compared to the state-of-the art techniques.

In terms of optimization time, some database systems determine query plans in advance at compile time. While others forego pre-computed plans and "route" tuples on-the-fly at runtime. We observe that these approaches closely resemblance network communication methods namely, *connection-oriented* or *connection-less*. In connection-oriented, a connection with the receiver is established in advance before passing any data. While in connection-less, data is sent without establishing an a priori connection, and the next hop of a packet is determined by a router at runtime. The parallel between route optimization in networking and query optimization in databases is evident. Query operators can be viewed as a "network" and a query plan as a specific "route" through all operators in the "network". Given this parallel, we classify the state-of-the-art techniques according to the optimization time dimension as "*route-oriented*" and "*route-less*" solutions. "Route-oriented" means that routes (i.e., query plans) are established in advance.

Database systems including major commercial DBMSs that determine a query plan a priori employ the *route-oriented* paradigm. Systems like Eddy [6] and its descendants [7,8] decide at runtime for every data tuple or batch of tuples which operator should process it next. Hence they fall under the *route-less* category.

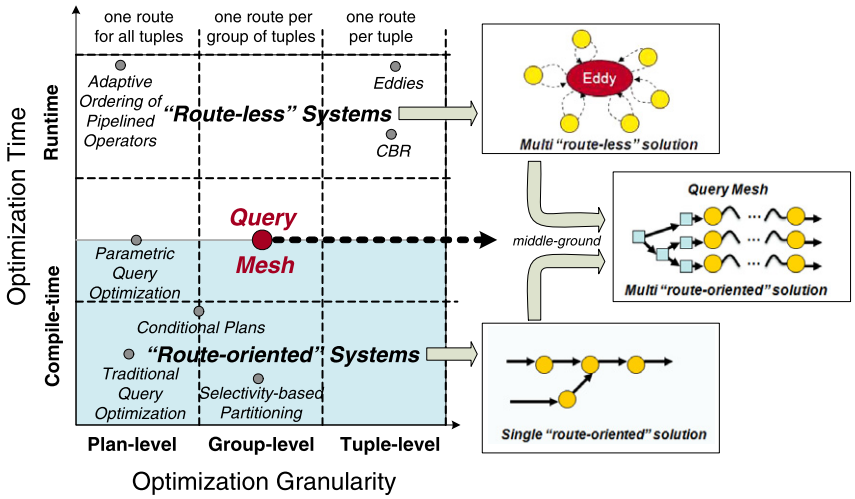


Fig. 1. *QM* versus other optimization techniques.

Most major commercial DBMSs determine a query plan a priori. Almost all *route-oriented* solutions pre-compute a *single route*. This then leads to their main disadvantage, namely optimization coarseness [9,10]. Having a fully established plan also comes with the advantage that all tuples follow the same execution plan and the execution is thus nearly "overhead-free". Furthermore, the size of data tuples does not need to be extended to store processing-related metadata, such as which operators have been visited.

Some systems at runtime decide for every data tuple which operator to visit next, e.g., Eddy [7,6,8]. Such systems discover different execution routes for tuples on-the-fly [7,6]. *Route-less* systems by design are inclined to use multiple routes. Unfortunately the "optimization decision" (i.e., which operator should process a tuple next) is made continuously, and in the worst case for every individual tuple, thus resulting in an unavoidable per-tuple route discovery overhead. This approach ignores the fact that stable conditions tend to dominate query execution time, and that tuples with identical content or similar statistical properties are likely to be best served by the same route [1]. Furthermore, the size of individual tuples respectively is relatively larger, as tuples must now carry processing status related metadata.

In summary, optimizing too frequently (i.e., *route-less*) may discover several plans but likely will waste resources. However, optimizing too coarsely (i.e., *route-oriented*) may miss critical opportunities to improve query execution performance. We thus propose a practical middle-ground approach between these extremes in the form of a "*multi route-oriented*" solution called *Query Mesh* or *QM* (see Fig. 1).

In addition compared to traditional relational databases where the entire dataset and complete statistics about it are available, data stream management systems (DSMS) work with incomplete datasets and imprecise statistics. In DSMS, the knowledge about the environment such as data input rates, operator selectivities, attribute values and their distributions is typically incomplete and is in fact continuously changing. Thus, uncertainty naturally arises during query optimization in the streaming context. Most query processors in DSMS (similar to their relational counterparts), however, consider all knowledge to be certain and complete during the optimization phase, which may significantly limit query performance at runtime [11]. Existing solutions dealing with uncertainty, e.g., [11,8], are primarily single-plan-based and focus on cardinality estimations for the data as a whole. A more complex structure and a different execution paradigm (that employs unique plans for distinct subsets of data) makes these solutions for tackling uncertainty insufficient for a multi-route solution like query mesh.

1.4. Our proposed solution: query mesh

The main idea of *QM* is to determine *multiple execution routes*, each optimized for a subset of data with distinct statistical properties. Then, a *classifier* model is created based on the computed set of routes and the data characteristics (Fig. 2). Henceforth we refer to the set of multiple execution routes as a *multi-route configuration*. At runtime, the classifier determines the best processing route for incoming tuples. Thereafter each data tuple will travel only on its designated optimal route. While many classification models could be used, e.g., neural networks, naive bayes, etc. [12], we employ a *decision tree* (*DT*) classifier. Our experiments demonstrate that our *DT* classifier approach very quickly "zeros-in" on the sought-after route with typically a small number of comparisons. Additional beneficial features of *DT* classifiers are covered in Section 4. Lastly, when there is uncertainty in the statistics collected and for a subset of data more than one route is deemed to be suitable, then QM supports uncertain route specification by the classifier and execution by the QM executor.

Research challenges. Several practical considerations make this multi-route problem challenging: 1) Finding an optimal solution is complex, because there is a combinatorial explosion of all possible execution routes for all possible subsets of

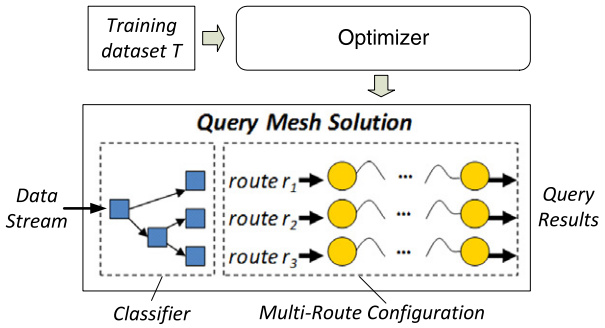


Fig. 2. Optimizer producing logical *QM* solution.

training data to consider. 2) In addition, the classifier model, the number of execution routes, and the choice of particular execution routes are strongly dependent on each other. A change in one may cause a modification in the other, subsequently affecting the cost of the overall *QM* solution. This raises the proverbial chicken and the egg question. Should the training data get partitioned first, and then the optimizer finds the routes for the different partitions? Or should it be the other way around? 3) Query execution with multiple concurrent routes is a challenge. If the execution infrastructure is not well-designed, the benefits of using multiple distinct routes for different subsets of data may be completely lost. 4) Finally, QM must be able to support uncertain routes when statistics are known to be based upon unpredictable, dirty, and unreliable streaming data.

This paper now addresses the above challenges. In particular, the contributions of this paper include:

1) We introduce the *Query Mesh* model, a general middle-ground query processing approach. *QM* consists of a classifier and a multi-route configuration, where each customized route serves data with distinct statistical properties.
2) We employ machine learning measures to induce an efficient classifier operator. Then online the classifier operator inspects incoming tuples to determine the best routes for each tuple with minimum overhead.
3) We analyze the *QM* search space and propose an algorithm to find optimal *QM*s.
4) Due to the large complexity of the search space, finding the optimal *QM* is not feasible in practice. To address this issue, we propose a family of innovative cost-based search heuristics for compile-time query optimization that locate near optimal *QM*s.
5) We propose the *Self-Routing Fabric* (or *SRF*) infrastructure to execute multiple routes in parallel without physically constructing the multiple query plans nor utilizing a router like Eddy [13].
6) We also discuss possible extensions to the core QM infrastructure to support the ability to select uncertain routes based upon imprecise statistics and adjust route decisions mid-execution.
7) Our extensive experiments compare *QM* to the state-of-the-art techniques, namely *route-oriented* (i.e., the traditional "single plan for all data" system [14]) and *route-less* systems (i.e., the Eddy framework with CBR routing [1]). Our results show that compared to other systems, *QM* substantially improves performance. Thus they establish that *QM* is indeed a promising paradigm for query optimization.

In this work, we focus in depth on the compile-time QM optimization problem – the characterization of the QM search space and strategies for finding the best QM configuration at compile time. While our work in [15] centers on the design of runtime assessment tuning of QM to address the problem of concept drift – a well-known subject in machine learning. More details on the differences between these two works can be found in Sections 5.3 (discussion on adapting query mesh) and 2 (related work).

This paper is organized as follows. The *QM* framework is covered in Section 3. Section 4 discusses the *QM* optimizer. While Section 5 describes the *QM* executor and outlines how we may handle imprecision in routes in QM. Our experimental study is in Section 6. Finally Sections 2 and 7 cover related work and conclusions, respectively.

2. Related work

Query optimization is a well-studied area, with most effort concentrating on optimizing a *single execution plan* for all data [16,17]. *QM* approach is related to the concept of *horizontal partitioning* [2], where the main idea is to partition data so that different partitions can be processed along different physical query plans. In contrast, QM partitions and processes data along a single physical query plan that supports multiple routes.

In the context of sensor networks, *conditional plans* [5] 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*. The goal is to reduce the communication and acquisition costs to minimize the sensor battery consumption. Conditional plans focus on selecting a single and cheap sensor partitioning attribute. Such an attribute is not necessarily the "best" splitting attribute in a general

query optimization context. In that respect, *QM* is a more general model selecting (often several) best splitting attributes in the classifier based upon machine learning knowledge gained by examining the data distribution. A conditional plan is typically computed on a powerful computer (a base station) and then the 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 another key difference from the *QM* execution approach.

Although adaptivity is orthogonal to the ideas presented in this paper, several techniques from *adaptive query processing* [18] are related to *QM*. *Eddies* [6], which can adapt at the tuple granularity, is observed to mostly be using a single plan for nearly all tuples as was indicated in [1] and confirmed by our experiments. We borrow the concept of the SteM (a State Module) from [8]. SteM is used with an Eddy to allow flexible adaptation of join algorithms, access methods, and the join spanning tree. The STAIR operator [7] encapsulates the state typically stored inside the join operators and exposes it to the Eddy. [13] adds batching to Eddies to reduce the tuple-level routing overhead. What differs from Eddy's batching and our rusters is that in the former the batching is naive: continuous chunks of k tuples that happened to arrive together in time are batched and routed together. In *QM* instead, the tuples are grouped together into the same *ruster* using the classifier and share the same customized optimized route.

Related to *QM* is the *content-based routing* (CBR) extension of Eddies [1]. CBR focuses on continuously profiling operators and identifying "classifier attributes" to partition the underlying data into tuple classes that may be best routed differently by Eddy. One distinguishing characteristic between *QM* and CBR is that CBR considers only a single-attribute decisions. We take a more general approach in *QM* and build a classifier model that implicitly takes multiple attributes (their values' correlations and statistics) into account to identify distinct data subsets with respect to their execution routes. Although [1] states 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. 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 may not change frequently [1]. 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. In contrast, *QM* has a much wider scope of applicability as it addresses query processing using multiple routes in plan-based systems – the standard in database systems. We have shown experimentally that the *QM* approach has outperformed Eddies and CBR by a substantial margin.

While in Section 5.3 we argue that our QM architecture naturally lends itself to adapting an existing query plan with little effort, in a twin work [15] we dug further into the subtitle challenges specific to the adaptivity of the QM multi-route system. The concepts presented in [15] are orthogonal and complimentary to the fundamental work in this paper which focuses on QM optimization and QM execution. More precisely, the contributions of [15] are all related to runtime assessment tuning. In particular they dealt with QM monitoring, QM concept drift detection, and actuation of the QM migration. In this work, we instead focus in depth on the compile-time QM optimization problem – how to produce a high quality QM assuming relatively stable statistics. In particular we explore in detail the QM search cost model, QM search heuristics, and QM architecture.

Some works have proposed uncertainty-aware solutions for query optimization e.g., [11,8]. They primarily focus on a single-plan strategy for query execution. While in this work we focus on the impact of imprecise data streams on multi-route solutions.

In Babcock and Chaudhuri [11], using the probability distribution, the optimizer selects the appropriate query plan after considering the relative importance of predictability vs. performance preference of the user. Prior to optimization, the user selects the trade-off between the two goals of predictability and performance (which could be at odds sometimes) to find the appropriate query plan.

While Raman et al. [8] addresses query optimization by introducing the STAIRS operator, an extension to the ripple join operator. Beyond the standard join processing, the STAIRS operator allows the system to dynamically adjust the intermediate tuples stored in the cache to optimally process tuples based upon changes in the selectivity, data arrival rates, and/or performance of operators in the query plan.

3. The query mesh framework

The *QM* framework consists of two core components, namely the QM *optimizer* and QM *executor* (Fig. 3). For a given query, the optimizer locates the *QM* solution (i.e., the classifier and multi-route configuration) with the best query execution performance. The *QM* executor instantiates the *QM* solution as the *QM* physical runtime infrastructure. The *classifier* at runtime assigns the routes for the processing of the incoming tuples based upon their characteristics. The *QM* executor using the *Self-Routing Fabric* executes the query by processing each incoming tuple along the route selected by the *classifier*. Full details on the QM *optimizer* and QM *executor* are in Sections 4 and 5 respectively.

4. Query mesh optimizer

The optimizer, given a query Q and training dataset T, finds a multi-route *QM* solution (R, C) where R is a set of routes and C is a classifier that results in the lowest execution cost for tuples in T (Fig. 3).

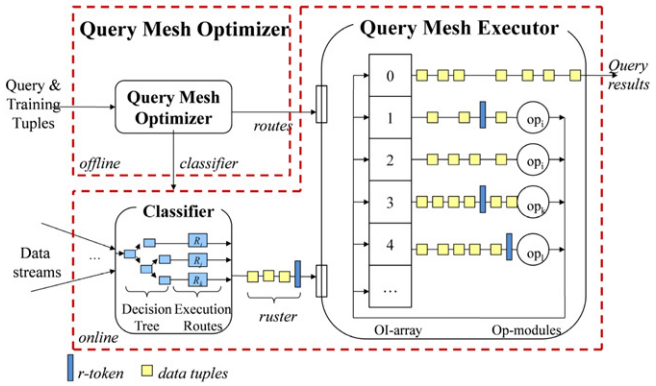


Fig. 3. Query mesh framework.

4.1. Data model and preliminaries

Our optimizer extends the relational database model to support streaming data environments. In general, we assume a stream S is a possibly infinite data source of elements $\langle s, t\rangle$, where s is a tuple belonging to the schema of S and t is the timestamp of the element. A query, expressed by the CQL query language [19] as shown in our motivating example in Section 1, describes the information needs over data streams submitted by a user. A time-based sliding window on a stream S is a moving window extending to the specified time interval T specifying a computable period of time. Intuitively, a time-based sliding window defines its output stream over time by sliding an interval of size T time units capturing the latest portion of a stream. It enables us to limit the number of tuples considered by a query. In this paper, we focus on select-project-join (SPJ) queries. However, queries containing additional operators such as group by or aggregation can also be handled by QM. Selection and projection queries are stateless, and thus can be seamlessly inserted into any pipeline in any order. Thus we focus our discussion on the join operator. We use join operators that are similar to SteM operators [8] that correspond to a half of a traditional binary join operator. Such join operator is a stateful operator such that it stores all tuples that have been processed thus far from one input stream so to be able to join them with future incoming tuples from the other input stream. They are formed over a base stream, and support build (insert), search (probe), and delete (eviction) operations.

4.2. Training dataset

Using a *training dataset* T that represents the data and its expected distribution is a common approach in both database systems [1,20] and data mining prediction models [21]. The dependence of streaming databases on data samples is unavoidable, as it is impossible to "see" all of the streaming data a priori.

Selecting a good quality training dataset is challenging. A training dataset's size and accuracy directly affect the size of the *QM* search space and the quality of the resulting *QM* solution. With a smaller training dataset, we may be able to enumerate all possible *QM* solutions, but may not accurately represent the real data. Whereas with a larger training set, the accuracy may improve, but at the cost of an extremely large search space, making it impossible to enumerate all solutions.

The selection of a training dataset to accurately depict the data distribution via a sampling technique is a research topic in its own right. We explored several techniques from statistics, including random sampling with cross-validation and sampling with bootstrapping [21]. These methods allow the system to estimate how well the selected dataset represents future as-yet-unseen data, and to re-sample until the desired accuracy is achieved.

In practice, training data can be collected using statistics from previous execution runs of the query or by employing a similar approach to *plan staging* [18], where optimization and execution are interleaved. The first stage of query processing may use a single route for query execution, while simultaneously collecting data statistics and training data. Using the samples from the first stage as the training dataset, the optimizer then refines the *QM* solution into multiple routes in a subsequent phase.

4.3. Query mesh search space

To find an optimal *QM* solution we must estimate the execution cost for the optimal route for each possible data subset in the training dataset, i.e., to explore the *QM* search space. The cardinality of the training dataset directly impacts the size of the *QM* search space. When the cardinality of the training dataset is large, a larger number of data subsets are possible. This causes more differences among statistics and routes which expands the number of QM configurations to be evaluated by the optimizer. Hence, the training dataset must be selected wisely to be compact yet sufficiently representative of the real

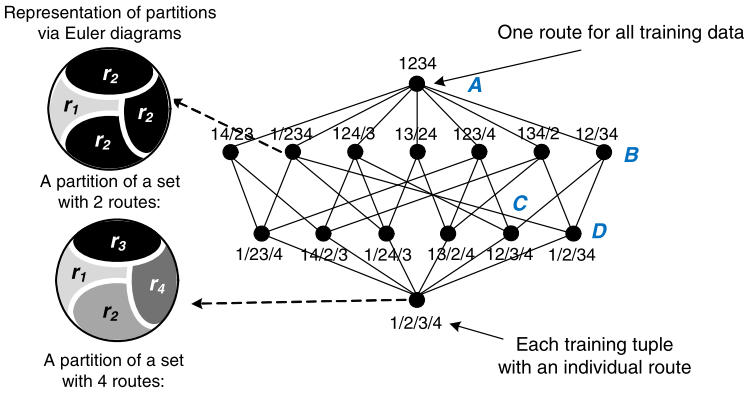


Fig. 4. Lattice-shaped query mesh search space.

data. To reduce the size of the search space, we perform *data condensing* on the set of sampled data tuples. The condensing step aims to select a subset of tuples without degrading its representativeness.

Condensing techniques can be categorized into two main groups, namely, to select a subset of the original tuples or to modify or generate them [22]. In both cases original data "densities" (i.e., value frequencies) [23] are associated with the condensed tuples. We implemented the second approach. For more details on the condensing algorithms, we refer the reader to [23]. Henceforth we refer to a condensed tuple as a *training tuple* t and the reduced dataset composed of condensed tuples as a *training dataset* T. We developed a clear abstraction of the complete QM search space to express the QM search problem as a movement in the search space.

Example 1. Consider multi-route configuration S_1, represented as "xy/zw/stv", where {x, y}, {z, w}, and {s, t, v} denote three subsets of training tuples that are respectively being processed by a distinct route. The optimizer may explore the neighborhood of the multi-route configuration S_1 by for instance merging the two subsets {z, w}, and {s, t, v} within multi-route configuration S_1 resulting in multi-route configuration S_2: "xy/zwstv". The improvement of S_2 over S_1 is measured by our cost model (Section 4.5).

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 *one single route for all tuples* in T. Let n denote the cardinality of the training tuple set T, i.e., $n = |T|$. The number of possible multi-route configurations explored in the worst case equals the number of distinct ways of assigning n tuples to k routes. The *Bell number* B_n [24], the number of different *partitions* of a set of n elements, thus describes the size of *QM* search space. The problem is challenging, as B_n's complexity is exponential (see Eq. (1)). The Bell number [24] can be represented as the sum of *Stirling numbers* $S(n, k)$ where $S(n, k)$ is the number of ways to partition a set of cardinality n into exactly k nonempty subsets.

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

Example 2. Given 4-element training data set representing the Stock stream, namely, data set in Stock stream {Apple, Amazon, Google, Ford}. For brevity, we denote Apple as 1, Amazon as 2, Google as 3, and Ford as 4. Thus, {{Apple, Amazon, Google}, {Ford}} is denoted by "123/4" where "123/4" represents a multi-route configuration containing two routes, namely, {{Apple, Amazon, Google} and {Ford}}. Then this set could be partitioned in 15 distinct ways: 1234, 1/234, 2/134, 3/124, 4/123, 12/34, 13/24, 14/23, 12/3/4, 1/23/4, 1/2/34, 13/2/4, 14/2/3, 24/1/3, and 1/2/3/4. Hence, in this case, we have $B_4 = 15$. Fig. 4 illustrates the lattice-shaped *QM* search space for a set of training tuples of size $n = 4$. Node "123/4" corresponds to a multi-route configuration with two routes; one subset of tuples uses route r_1 and the rest use route r_2. The total number of different multi-route configurations here equals 15 ($B_4 = 15$). A single basic transformation (e.g., a split of a subset, or merge of two subsets) would transition from a multi-route configuration (a node in the search space) to its neighbor along an edge in the space, e.g., "24/13" → "24/1/3".

Besides determining which configuration of data subsets best suits the given query Q and training dataset T, the optimizer must select the best processing route for each data subset in our chosen solution. The best route is the optimal order in which operators should be processed. Finding an optimal ordering of operators to optimize the query performance for a given set of data is a well-studied topic [16,17]. Our QM optimizer can employ any of the state-of-the-art techniques [25,9,26] to locate the single best route for a subset of data based on available statistics for that data partition subset. For example the single best route can be found using operator rank [17], dynamic programming [26], or transformation-based [9] techniques.

4.4. Creating a classifier

Once a desired *QM* solution (R, C) has been selected by the optimizer, a classifier is implemented that associates each computed route with the specific characteristics of each data subset (Fig. 3). There are many classifiers in the literature, each with its strengths and weaknesses. Determining a suitable classifier for a given problem is difficult as a classifier's performance and quality depends greatly on the characteristics of the data to be classified [21]. While many classification models could be used, e.g., neural networks, naive bayes, etc. [12], we employ a *decision tree* (*DT*) classifier. *DT* is attractive for several reasons: First, complex decisions can be approximated by combining simpler local decisions at various levels of the tree. Second, in contrast to other classifiers, where each tuple is tested against all classes, in a *DT* classifier, a tuple is tested against only certain subsets of test conditions so as to eliminate unnecessary computations. The efficiency and effectiveness of our *DT* was confirmed by our experimental study.

Many algorithms for *DT* induction can be found in the literature [12]. Our *DT* induction algorithm executes in a top–down recursive manner. At the start, all training tuples and their assigned routes are at the root. Then, they get partitioned recursively by the tree induction algorithm based on selected test attributes. Test attributes are selected on the basis of the *entropy*-based measure, called *information gain* [12]. Finally, the leaf nodes of the *DT* contain the route identifiers for the execution routes that will be assigned to the tuples that reach those leaf nodes after classification. Conditions for stopping *DT* growth are either that all training tuples for a given node belong to the same route, no attributes for further partitioning remain (when this occurs *majority voting* [12] is employed to assign a class label to the leaf), or no training tuples are left.

4.5. Query mesh cost model

We now describe the cost model we utilize to determine the best *QM* solution. The best *QM* solution is the solution with the lowest estimated execution time for the selected training dataset. This consists of the cost to both classify and execute each incoming tuple along its classified route.

(1) **Cost of routes.** Each execution route r_q in query q has a per-tuple cost $c(r_q)$ to process a tuple. $c(r_q)$ represents the expected time to process a single tuple to completion using r_q, meaning either to output or drop the tuple. The cost of r_q is commonly calculated using two characteristics: 1) *cost* $c_i(op_i)$ represents a per-tuple cost of $op_i \in r_q$, and 2) *selectivity* $\delta(op_i)$ denotes the fraction of tuples that are expected to satisfy op_i.
(2) **Cost of classification.** The classification cost for processing an incoming tuple is defined as the cost $c(DT|r_q)$ of traversing a single path in the classifier decision tree p_{dt} which starts at the *DT* root and ends at a leaf node with a route label r_q. $c(DT|r_q)$ is a function of the number of nodes in the path $|p_{dt}|$, multiplied by the cost $c(node_i)$ of computing each test node in the path $c(DT|r_q) = \sum_{i=1}^{|p_{dt}|} c(node_i)$.
(3) **Multi-route overhead.** Maintaining multiple execution routes introduces system overhead (e.g., memory, processing, etc). For simplicity of presentation, we abstract all overhead associated with a route into a single variable *OVH* representing the average overhead per route.

Thus, the total cost of a *QM* is estimated by

$$cost(QM) = \left(\sum_{r_q=1}^{|R|} f_{r_q} * \big(c(r_q) + c(DT|r_q)\big) \right) + |R| * OVH \tag{2}$$

where r_q represents a leaf node (i.e., a route) in the *DT* of the QM solution, $|R|$ the number of leaf nodes in the *DT* of the QM solution, and f_{r_q} the expected fraction of tuples from the training dataset T that at runtime will be mapped via the *DT* to r_q.

4.6. Optimal query mesh search

As a baseline, we will use the *Opt-QM* algorithm in Fig. 5 that is guaranteed to find an optimal *QM* solution. *Opt-QM* traverses all multi-route configurations in the search space to locate the optimal *QM* solution.

Opt-QM. First, *Opt-QM* computes the *power set* $P(T)$ for the given training dataset T (Line 2). The power set of T is the set of all subsets of T. Using training dataset, the statistics for each subset in the power set is calculated (Line 4). A route is computed for each subset similar to the methods discussed in Section 4.4 (Line 5). Then different training tuples are run through the operators to estimate the operators' costs and selectivities. Next the best routes for each subsets are computed. Finally the algorithm iterates through all possible *partitions* of T by composing the subsets, each representing a multi-route configuration for T. For each partition p composed of the subsets $\{S_i \dots S_j\}$ (Line 7), a union of all routes R associated with it's subsets is collected (Line 8) and then the classifier C is induced (Line 9). A *QM* solution is constructed with the routes R and the classifier C (Line 10), and its cost estimated (Section 4.5). If the new *QM* has the smallest cost compared to the QM solutions seen so far, it is kept (Line 12), otherwise it is discarded (Line 14). After exhaustive enumeration of all possible configurations, the algorithm returns the *QM* solution with the smallest cost as a result.

Algorithm *Opt-QM* (T – training dataset)

```
1: bestQM ← null; bestQMCost ← ∞
2: P(T) ← ComputePowerSet(T) //power set of T
3: for each set S ∈ P(T)
4:    S.stats ← stats(S)
5:    S.r ← best route for S based on S.stats
6: repeat
7:    let p ← {S_i ... S_j} //p is a partition of power set P(T)
8:    R ← BestRoutes(p) //union of best routes of {S_i ... S_j}
9:    C ← InduceClassifier(p) //classifier based on data and routes
10:   QM ← NewQMSolution(C, R)
11:   if (QM.cost < bestQMCost) then
12:      bestQM ← QM; bestQMCost ← QM.cost
13:   else
14:      discard QM //better QM has been found earlier
15:   end if
16: until (all partitions p ∈ P(T) enumerated) //B_n of them
17: return bestQM;
```

Fig. 5. Optimal *QM* search algorithm.

Complexity analysis. The complexity of *Opt-QM* is $O(B_n\ ^*E)$. B_n (Section 4.3) represents the cardinality of all multi-route configurations for the set T. E denotes the time complexity of the particular route computation algorithm used, e.g., $E = O(n2^n)$ for dynamic programming [27]. Clearly with large training datasets, *Opt-QM* algorithm is not scalable. The problem of finding one optimal route alone is known to be *NP*-hard in the general case [17]. Looking for multi-route configurations further increases the complexity of the problem. Consequently, given both the exponential running time and the space requirements efficient search heuristics are required in practice to tackle this problem.

4.7. Query mesh search heuristics

We thus now propose a series of cost-based heuristics for finding a good quality *QM* solution in reasonable time without enumerating the entire search space. The heuristics have the following three main steps:

Step 1. A *start solution* QM_s is chosen and its cost is computed. It is set as the best solution found so far, $bestQM = QM_s$.
Step 2. A *search strategy* is iteratively employed by traversing the *QM* search space to find another solution *QM'*.
Step 3. *QM'*s cost is computed and compared to the *bestQM* cost. If *QM'* has a smaller cost, *bestQM* is replaced with *QM'*. We repeat Steps 2–3 until a *stop condition* is met.

All three steps in unison have a significant impact on the quality of the final *QM* solution. We thus design the most appropriate strategy for each step. Typically, after a start solution is chosen, the search strategy performs walks in the search space via a limited number of moves. If a poor start solution is picked, the search strategy may never reach a quality *QM*. We now investigate various schemes to address the following questions: 1) How to pick a promising *start QM solution*? 2) What *search strategies* are effective at improving the start solution? 3) When should the search terminate (i.e., *stop condition*)?

Selecting a start solution. One approach is to pick a start solution based on the similarity of data content. Given a training dataset T, a *content-driven approach* (or *CDA*) partitions training tuples based on the respective similarity of their values. The motivation is that similar content likely means similar selectivities, and thus the same best routes. Each attribute domain is partitioned based on a pre-defined threshold that prescribes how "close" the content of training tuples are to one another. For a discrete domain it could be a set of values. For a continuous domain it could be a range. This is akin to data clustering. After the content-based groups are determined, the best route for each group is computed. This corresponds to the starting QM solution. The quantity and the quality of routes selected for this starting solution depend on the partitioning function. Clearly during the search process, the optimizer will consider other solutions in the QM search space, which means that these initially content-based groups may still be merged into larger groups as dictated by the particular search strategy and the associated estimated costs of the new QM solution. The advantage of *CDA* is that it is intuitive, rather simple, and can produce distinct subsets quickly.

An alternative *route-driven approach* (or *RDA*) is to first compute the routes for each of the training tuples separately. Thereafter, tuples are "grouped" by similarity of their respectively assigned routes, i.e., groups of "route-equivalent" tuples. Then the classifier model is induced. The motivation here is that tuples with different values may still share the same best route. In other words, *RDA* is capable of finding groups of arbitrary "shape", i.e., tuples with very different content may still be found to belong to the same group and thus may help create a more efficient *QM* solution.

Other *QM* start solution approaches include *random-pick* (*RP*), where a *QM* with the smallest cost out of x randomly selected solutions is chosen, and *extreme*-1-*route* (or *E*1*R*), which picks a *QM* solution where all data is processed using the

Algorithm *II-QM* (*bestQM* – start *QM* solution)

```
1: bestQMCost ← bestQM.cost
2: while (not stop condition) do
3:    QM ← start solution (using methods from Section 4.7)
4:    while (not local_minimum(QM)) do
5:       QM' ← random solution in NEIGHBORS(QM)
6:       if (cost(QM') < cost(QM)) then
7:          QM ← QM'
8:       end if
9:    end while
10:   if (QM.cost < cost(bestQM)) then
11:      bestQM ← QM
12:   end if
13: end while
14: return bestQM;
```

Fig. 6. *II* search strategy for *QM*.

same (single) route and the classifier is empty. *E*1*R* relies on the search strategy to divide the data into particular subsets to determine the next *QM* solution.

Selecting a stop condition. The *stop condition* largely depends on the *QM* search strategy used. In general, query mesh search may stop when either k iterations have gone by, or the solution did not improve in the last several rounds, i.e., the search process plateaued. Alternatively, the search can be bounded by time or resources, e.g., when memory limits are reached.

Search strategy. Many search algorithms can be found in the literature – each with its own respective strengths and weaknesses. We chose randomized search strategies, in particular, iterative improvement and simulated annealing. Such approaches are guaranteed to find a good *QM* solution in a reasonable amount of time [28]. They thus serve as viable alternatives to the prohibitively expensive optimal *QM* search (Section 4.3).

Iterative improvement *QM* search (*II-QM*). Fig. 6 sketches the iterative improvement *QM* search strategy or *II-QM*. The inner loop of *II-QM*, also called a local optimization, starts at a given start *QM* solution and improves it by repeatedly accepting random downhill moves (i.e., *QM*s with decreasing costs) until it reaches a local minimum. *II-QM* repeats such local optimizations until a stop condition is met. Then it returns the local minimum with the lowest cost found. The procedure is repeated various times, each time starting at a new *QM* start solution, until a stop condition is met. Then the algorithm compares the local minima found and chooses the solution with the lowest cost. As time approaches ∞, the probability that *II-QM* will visit the global minimum approaches 1. Thus with enough repetitions of the first steps, the algorithm locates a solution close to the global minimum.

Example 3. We now explain the above *II-QM* search method using an example. For this, consider the scenario from Example 1. Assume the optimizer selects 1/234 (i.e., {Apple}, {Amazon, Google, Ford}) as the start QM solution for the Stock stream. Then it tries to improve the initial QM solution by randomly considering neighbors in the search space, which in this case, would correspond to taking downhill moves. Specifically, 1/23/4, 1/24/3, 1/2/34 are the neighboring states in the search space. Assume now that 1/23/4 is the local minimum among the three alternative solutions, because Amazon, and Google, belonging to the same sector in the stock market, tend to have similar data characteristics. Consequently, the cost of such partition may be lower than that of the other two QM solutions. The optimizer repeats the above search process now from the current best solution, say 1/23/4, again considering its neighbors. This process continues until a local minimum is found, i.e., the neighboring QM solutions have more costs than the currently best QM solution. Beyond that, the optimizer repeats the complete search process by starting over with a different start solution QM each time. For instance, it selects 1234 as its start solution, and then finds out 123/4 is the local minimum as Apple, Amazon, and Google all belong to the same sector in the stock market. The optimizer compares the cost of the new local minimum solution 123/4 with the current best solution 1/23/4 and updates the best solution if the newly found solution has lower cost. The optimizer repeats the above search process until a stop condition is met.

Simulated annealing QM (*SA-QM*). In simulated annealing (*SA-QM* in Fig. 7), initially the "temperature" τ parameter is set to high to allow a great deal of random movement in the search space. 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. *SA-QM* accepts a worsening move with a certain probability. This probability declines as τ declines, by analogy the randomness in the movements decreases as the temperature falls. When τ is small enough the algorithm accepts only the improving moves.

Algorithm *SA-QM* (*bestQM* – start *QM* solution)

1: $QM = bestQM$
2: Choose a random neighbor of *QM* and call it *QM'*
3: $\delta = cost(QM') - cost(QM)$ *//Decide to accept the new query mesh or not*
4: **if** ($\delta \leqslant 0$) **then**
5: $QM = QM'$ *//QM' is better than or same as QM*
6: **else**
7: $QM = QM'$ with probability $e^{\frac{-\delta}{\tau}}$
8: **end if**
9: **if** (*stop condition* is met) **then**
10: exit with *QM* as final solution
11: **else**
12: reduce temperature by setting $\tau = \rho^{*}\tau$, and go to Step 1
13: **end if**
14: return *bestQM*;

Fig. 7. Simulated annealing *QM* search strategy.

Example 4. This example of the *SA-QM* search algorithm is based on the scenario in Example 1. Assume the optimizer chooses 1/234 as its start QM solution. It then chooses its random neighbor 1/23/4 as QM'. The cost of 1/23/4 would be lower than 1/234 as 2 (i.e., Amazon) and 3 (i.e., Google) share the similar data characteristics. Thus δ is less than 0. Therefore, 1/23/4 is considered as the current best QM solution. Thereafter, we again select a neighbor of the current best QM solution, 1/23/4, which in this case is 1/2/3/4 as the new QM'. Assume here that the new QM' does not have a better cost than QM, but it is similar. In that case, with a probability (driven by the current "temperature" setting), the optimizer may still accept this new QM'. This allows the search in the beginning to be relatively generous in exploring the space. Over time, the temperature would be reduced, meaning, that the probability that a new solution QM' would be considered as next search step becomes smaller and smaller over time. In other words, later in the search, the optimizer will tend to only search directly neighbors in the local minimal space being examined. As a result, 1/234 → 1/23/4 → 1/2/3/4 → 12/3/4 → 123/4 could be one of the possible movements that reach the optimal solution in the lattice-shaped search space.

5. Query mesh executor

5.1. Physical QM instantiation

The *QM* executor takes the *QM* solution, namely, the classifier and set of routes, found by the optimizer and instantiates a *QM* runtime infrastructure (Fig. 3). The runtime infrastructure consists of the *Online Classifier* operator and the *Self-Routing Fabric* (*SRF*) in which operators *self-route* tuples according to best routes (Fig. 9). *SRF* infrastructure consists of two main structures. 1) The *Operator Index Array* (*OI-array*) stores pointers to all query operators. Each index *i* corresponds to a unique operator op_i. Index "0" is reserved for the *SRF* global output queue, where query result tuples are sent. 2) The *Operator Modules* (*Op-modules*) are the actual query operators, e.g., *selection*, *join*, etc.

Arriving tuples are classified into "routable clusters" – groups of tuples with the same route, using a *classification window* W_{TC}, where W_{TC} is a *tumbling* window. 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 the classification overhead can be minimized by classifying one tuple per window and then sending the rest of the tuples along 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 1 *(Ruster).* Let S_i be a data stream. A routable cluster *RC* is a window-bounded set of data tuples $\{s_1 \dots s_k\} \subseteq S_i[W_{TC}]$ assigned to the same route r_i by the classifier. Tuples in the ruster have timestamps in the time interval $[RC.ts - W_{TC}, RC.ts]$, where *RC.ts* is the time of tuple classification and W_{TC} is the classification window size.

In this paper, we focus on select-project-join (SPJ) queries. However, queries containing additional operators such as group by or aggregation can also be handled by *QM*. Selection and projection queries are stateless, and thus can be seamlessly inserted into any pipeline in any order. Thus we focus our discussion on the join operator. We use *one-way-join-probe* (*OJP*) operators, for joins, which are similar to SteM operators [8] that correspond to a half of a traditional join operator. They are formed over a base stream, and support the *insert*, *search*, and *delete* operations. The choice for our join implementation is influenced by our desire to support all possible routes. Such an approach ensures that each data subset can be executed along its optimal route – without any special treatment of states. Traditional join operator implementations are binary, i.e., they are formed over rigidly encapsulate pairs of states of two different streams within one atomic unit, i.e., the operator. The composition of the incoming streams (and hence their state) is directly coupled to the route taken by the tuples thus far. Furthermore, when composing two such binary operators into a pipeline, the states in the down-stream binary operators may now also be holding intermediate tuples produced by up-stream join operators. To support

Algorithm *ConstructSRF* (*QM* – logical *QM* solution)

```
1: OPT ← new operator translation hash table
2: for each r ∈ QM.R do
3:     for each op ∈ r do
4:         O ← (op) Union O
5:     end for
6: end for
7: SRF.OI-Array ← new Array[1 + |O|]
8: op_index ← 1 //0-th index is reserved for result tuples
9: for each op ∈ O do
10:    OPT[op] ← op_index
11:    SRF.OI-Array[op_index++] ← new PhysOp(op)
12: end for
13: ReplaceLogicalLabelsInOnlineClassifier(OPT)
14: return;
```

Fig. 8. Physical instantiation of *SRF*.

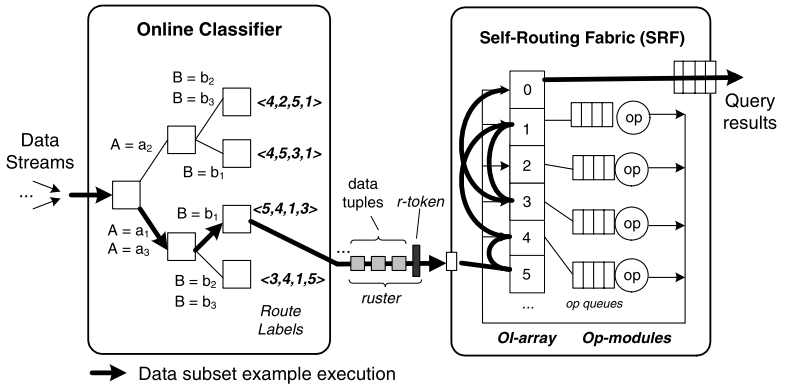


Fig. 9. *QM* physical runtime infrastructure.

all possible routes using traditional join operators, the system would need to support careful treatment of these states, including their reuse across different pipelines and their migration as done in [29]. Thus to avoid this additional complexity and overhead we adopt a unary join solution [8] and focus on the core ideas of query mesh infrastructure and optimization.

Before starting query execution, the *SRF* infrastructure is instantiated (see pseudo-code in Fig. 8). The algorithm takes as input a logical specification of *QM* from the optimizer. First it iterates through the operators in the routes of *QM* (i.e., *QM.R*), and computes a union of all operators denoted by *O* (Lines 2–6). Thereafter, the *OI-Array* is instantiated (Line 7) with the "0-th" index being reserved (Line 8), and with a physical instance of each operator assigned to one position in the array (Lines 9–12). The *OPT* hash table stores the mapping between the logical operators and their physical instances (Line 10). It is used during the instantiation to update the logical routes with their physical counterparts (Line 13). Label replacement simply updates the classifier labels with operators' physical ids in the *SRF*. This step is done only once at instantiation time to eliminate the burden of continuous logical-to-physical operator translation at runtime.

5.2. Physical execution

When new tuples arrive, they are first processed by the online classifier operator. To keep memory and CPU overhead minimal, the routing decisions are applied to groups of tuples, denoted as "*route clusters*" (or short "*rusters*"), rather than individual tuples. Thereafter, the rusters are forwarded into the *SRF* for query evaluation (Fig. 9). In contrast to simple tuple batches, *rusters* are based on the semantics of sharing the same execution route. Hence, a *ruster* conceptually is different from both a batch of tuples (e.g., grouping tuples that arrive contiguously in time) [13], as well as from a traditional cluster (e.g., grouping tuples based on similar values). The pre-computed route for a *ruster* is stored in a *route token* (short *r-token*). *R-tokens* are metadata tuples, similar to streaming punctuations [30], embedded inside data streams, thus partitioning data streams into *rusters*. What distinguishes *r-tokens* from other streaming metadata is that 1) they are "self-describing" as they carry *routing instructions* and 2) that routes in the *r-tokens* are specified in the form of an *operator id stack*.

A *ruster* is routed to the operator that is currently the top node in its routing stack. *OI-array* enables the knowledge of the "location" of other operators. After an operator is done processing the *ruster*, the operator "pops" the top node in its routing stack, and then forwards the *ruster* to the next operator. If *all* tuples from a *ruster* are dropped by an operator, then the *r-token* can also be discarded. When the *r-token* operator stack is empty, the *ruster* tuples are forwarded to the global output queue for content dismissal.

Example 5. A runtime execution is depicted by a thick black arrow in Fig. 9. Consider the scenario from Example 1. When new tuples from the Stock stream arrive, the online classifier operator first processes them based on their domain values. In this case, the routing decision for {Apple, Amazon, Google} is op_3, op_2, op_1. Namely, the new tuples should join with the news and street research news and then join with *BullishPatterns*, as these stocks from IT sector exhibit better performance on profitability. Consequently, op_1 has higher selectivity (i.e., producing more results) than the other two operators. On the other hand, the routing decision for {Ford} is op_1, op_2, op_3 as the company exhibits *BearPatterns* in recent stock market and thus op_1 is likely to produce less results. As a result, a route, $r_1 = \langle op_3, op_2, op_1 \rangle$, for {Apple, Amazon, Google} will be encoded in an *r-token* as a stack $\langle 3, 2, 1 \rangle$, where '3' and '1' are the first and last operators in the route respectively. And a second route, $r_2 = \langle op_1, op_2, op_3 \rangle$, for {Ford} will be encoded in an *r-token* as a stack $\langle 1, 2, 3 \rangle$.

For r_1, it is first routed to op_3 as op_3 is the top node in its routing stack. op_3 joins the new arrival stock tuples with the street research tuples held in its state, and then sends the intermediate results to next operator on r_1's routing stack (i.e., op_2). Consequently, op_2 uses the incoming tuples from op_3 to probe and join with its state (i.e., news), and sends the updated intermediate results to the last operator op_1. In the end, op_1 receives the tuples from op_2 and uses them to join with *BullPattern* so as to produce the final outputs. r_2 would follow the same process, except the join ordering is different.

The QM architecture supports concurrent execution of multiple routes. Furthermore, *SRF* eliminates the need for any central router. This removes the "backflow" bottleneck problem present in Eddy where tuples are continuously sent back to the router to determine which operator to use next [13]. The runtime strategy of *SRF* provides an architectural query processing approach that naturally supports *multi-query execution*. An encoding for route shared possibly amongst multiple queries may also be stored in an *r-token*. This enables *SRF* to process such *rusters* just as it would for a single query.

5.3. Discussion of adapting query mesh

The beauty of our *QM* design detailed in this work is that, unlike classical hard coded route solutions, it now enables adaptivity in routes within a single physical operation, namely the change of the classifier. The optimizer is run on a separate thread offline. The optimizer starts with the current *QM* solution (i.e., classifier model and multi routes) and searches for a better *QM* solution. If a better *QM* solution is located only a simple pointer re-assignment to the new classifier object is needed. As an additional benefit, at runtime, both an older and a newer *QM* solution can concurrently be executed within the same infrastructure.

Due to the routing flexibility of our proposed architecture, there is no migration delay. Although routes (i.e., query plans) are pre-computed, their topology is not physically constructed. Instead, the *SRF* infrastructure provides distributed routing by query operators based on the plan specification assigned by the classifier to particular batches of tuples, i.e., rusters. This physical separation between the component that determines which plans should be used for execution (the optimizer) and the component that actually executes the plans based on the specifications (the executor) makes *QM* adaptivity so light-weight. In other words we get adaptivity for free in this network.

Lastly, subtle issues of adaptivity of QM are studied in [15], where the goal is to address the concept drift problem – a well-known subject in machine learning. Namely [15] addresses the problem of how to efficiently decide when to tune and what subcomponent of the overall QM solution to tune. See the related work section (Section 2) for more discussion.

5.4. Discussion of imprecise-routing in query mesh

Compared to a single-plan solution, where uncertainty may be present in only one route, a multi-route solution may have uncertainty in several routes. Route uncertainty occurs when there is uncertainty in operators' selectivities and costs – two key factors used by the classifier to select the routes. By uncertainty in routes we mean uncertainty in the costing of routes. This leads to ambiguity about the best operator ordering. To resolve this issue, we now consider how to best support uncertain route specification and execution in query mesh.

For this, we now relax the query mesh infrastructure and allow the classifier to select more than one route for a given subset of data. More precisely, if for a given subset of data more than one route is estimated to have a similar execution costs (Section 4.5) then these routes will be combined to form an uncertain route specification. To enable uncertain route specification, we introduce a small modification to the route encoding: the r-token $\langle 2, 3|1, 4 \rangle$ indicates that '2' is the first operator in the route, '3' or '1' is the next, etc. The encoding 3|1 depicts the uncertainty about the order of these two operators.

The QM executor must be adjusted to support execution of these uncertain routes. More precisely, when operator op_i is done processing a *ruster*, if there is uncertainty about which should be the next operator then op_i selects one of the possible subsequent operators in the ruster and update the ruster accordingly. In order to select the sequence of operators in the uncertain route to actually follow, we employ a simple feedback mechanism similar to lottery scheduling. In this approach, each operator would randomly select the next operator to follow on the uncertain route from among the set of possible next destination operators in the uncertain route. Initially, all possible next destination operators in the uncertain route are given an equal probability slice (in the so-called lottery wheel). A random coin toss, using that given probability model, would then be used to select which operator to actually utilize at a given time. In addition, we would keep track of the effectiveness of this choice made by counting the number of output tuples produced for this tuple by the chosen operator. This information would then be recorded for this operator with respect to the subset of tuples that this current

Table 1
Defaults used in the experiments.

Parameter	Value	Description
D	*Poisson*	Default data distribution
$\|A\|$	6	# of attributes in tuple schema
$\|S\|$	1000 tuples	Size of sample tuple set
$\|T\|$	10 tuples	Size of training tuple set (after data condensing)
Start Solu.	*Route-Driven*	*QM* start solution strategy
Search Strat.	*II-QM*	*QM* search strategy
Stop Cond.	$k = 3$	# of iterations in the search
W^{TC}	1000 tuples	Classification window size
Ruster size	100 tuples	*ruster* size

tuple belongs to. That is, we would incrementally adjust the subset specific estimated selectivity recorded for that operator. Subsequently, the probability slice assigned to this destination operator would be increased, if the selectivity was low. Hence the next time a choice would need to be made at this location for this uncertain route, the random selection of the destination operators in the uncertain route would now be somewhat biased based on the adjusted relative sizes of probability slices (the adjusted lottery wheel).

To update the ruster when the top of the stack contains uncertainty encoding, the operator removes the chosen next operator from the set of possible operators in the ruster. If the uncertain set now contains only one operator then the uncertainty encoding is removed (i.e., there is only one choice for the next operator) otherwise the uncertainty encoding remains. Reconsider the example above. If 3 is selected as the next operator then the ruster will be updated to $\langle 1, 4\rangle$. However, if the r-token was $\langle 3|1|4\rangle$ and 3 was selected as the next operator then the ruster would be updated to $\langle 1|4\rangle$.

6. Experimental evaluation

Experimental setup. All our experiments are run on a machine with Java 1.6.0.0, Windows Vista with Intel(R) Core(TM) Duo CPU @1.86GHz processor and 2GB of RAM. Similar to [7,1,8], our experiments use both real and synthetic data sources which we describe in more detail below.

We compare the performance of *QM* against the traditional "single plan for all data" system [14] (or SP) and the "multi-route-less" system [6] (or MR). For SP, we use a multi-way join (MJoin) [14] operator. MJoin is a generalization of symmetric binary join algorithms shown to provide the best plan for each stream, and thus it is our choice for SP. For MR, we use the Eddy framework with CBR routing [1], which is the closest approach in the literature to *QM*. CBR continuously profiles operators and identifies "classifier attributes" to partition the underlying data into tuple classes used by the router to determine which operator to send tuples to next. To ensure an even comparison, all systems were implemented in the same platform, namely, CAPE [31], a Java-based continuous query engine, and each implementation used as much of the same codebase as possible.

Our experiments include the measurement of several metrics: 1) the average output rate and execution time, 2) the execution-runtime overhead, 3) the total number of tuples produced, 4) the effect of training set size, and 5) the effect of the start solution. Besides comparing our QM solution against alternative strategies, we also study the impact of different features of our solution on the effectiveness of our optimizer, most notably metrics 4 and 5 above.

6.1. Data sets and queries

Stocks-News-Blogs-Currency dataset. We have implemented a data polling application that collects NYSE stock prices, foreign currency exchange rates (provided via web service by Yahoo Finance), news and blogs on different subjects (provided via RSS feeds).

Lab dataset. This dataset contains readings from sensors in the Intel Research, Berkeley Lab. The original dataset [1] consists of a single stream sensor readings. We have partitioned this dataset by sensor locations into several data streams. Each stream corresponds to a group of sensors in close proximity to each other. The sensor readings are sent to CAPE in generation order, as they would have been were the tuples submitted by the sensors in real-time.

Synthetic data sets. To establish data skew, we employ the *Uniform* and *Poisson* data distributions. These distributions model many real-life phenomena (Table 2). The default data properties, distribution parameters and system parameters used are depicted in Tables 1 and 2.

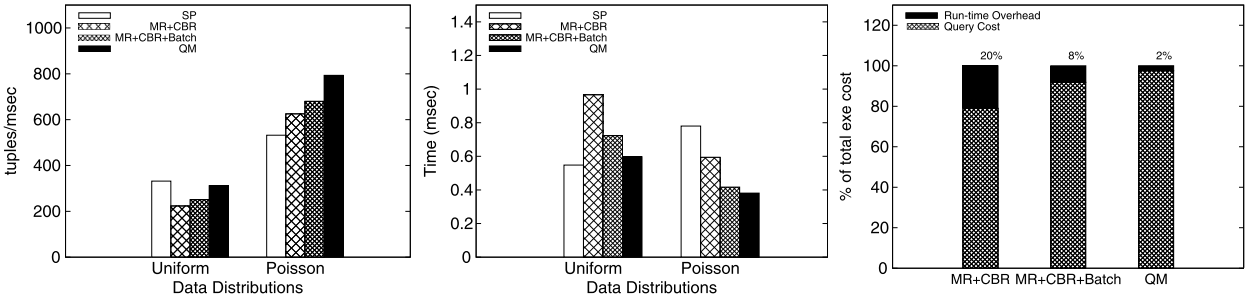
Queries. We deploy N-way join queries, as N-way join queries are among the core and most expensive queries in database systems. Such queries are commonly used to discover correlations across data coming from different sources and to compose complex results. In the context of *Stocks-News-Blogs-Currency* dataset, such a query may be useful for making a decision on a stock purchase, e.g., to estimate expected fall or rise in stock prices based on the latest news. For sensor data, this type of

Table 2
Distribution statistics.

Data distributions		
Name	Parameters	Application examples
Uniform	$\alpha \in \{\ldots, \beta-1, \beta\}$ $\beta \in \{\alpha, \alpha+1, \ldots\}$ $X \in \{\alpha, \ldots, \beta-1, \beta\}$	• Long-term patterns of data • Distribution of moving objects in some geographic areas
Poisson	$0 < \lambda < \infty$ $X \in \{0, 1, \ldots\}$	• Service times in a system • # people at a counter • # of times web server accessed per minute
Uniform ($\alpha = 0$, $\beta = 100$): *min*: 0.0, *max*: 100.0, *med*: 49.0, *mean*: 49.7, *ave.dev*: 25.2, *st.dev*: 29.14, *var*: 849.18, *skew*: 0.05, *kurt*: −1.18		
Poisson ($\lambda = 1$): *min*: 0.0, *max*: 7.0, *med*: 1.0, *mean*: 0.97, *ave.dev*: 0.74, *st.dev*: 1.01, *var*: 1.02, *skew*: 1.17, *kurt*: 1.89		

Table 3
Defaults used in the experiments.

Parameter	Value	Description
Data Arrival	*Poisson*	Data arrival distribution
μ	500 msec	Mean inter-arrival rate
$\lvert T_{dq} \rvert$	1000	Maximum # of tuples dequeued by an operator at a time
p	3	# of skewed partitions per stream. Partition selectivities result in a different best operator ordering compared to if the overall stream statistics were used
$\lvert T \rvert$	100 tuples	Size of training tuple set
H	*SA*	Default heuristic used to find query meshes
Start Solution	*Content-Driven*	Query mesh start solution strategy
W^{TC}	1000 tuples	Classification window size
Ruster size	100 tuples	Minimum *ruster* size



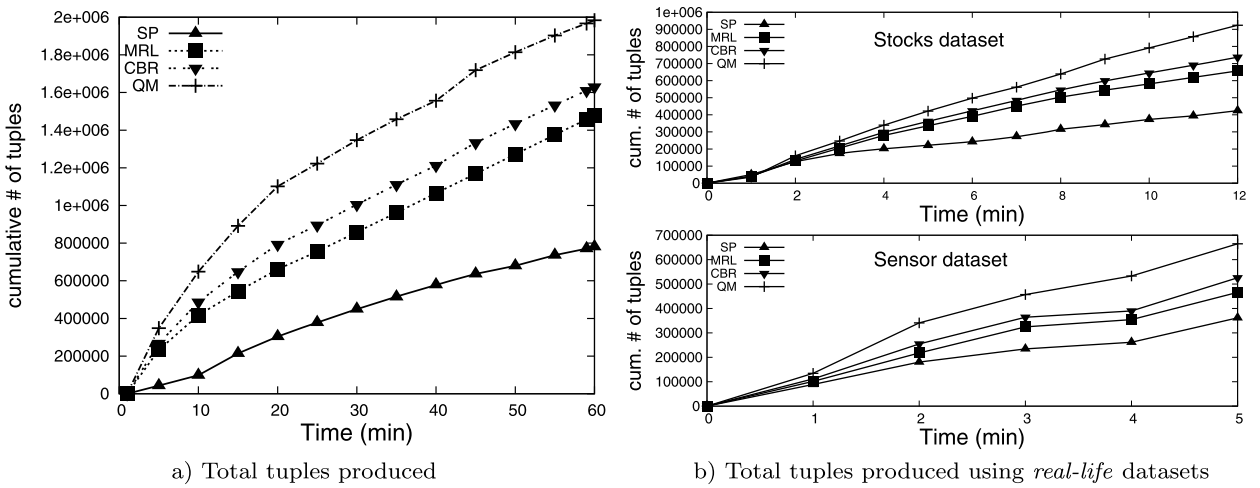
a) Average output rate b) Average execution time c) Comparison of runtime overheads

Fig. 10. Experimental results.

query may help to detect fire "hotspots" or for automatic temperature control inside buildings. The sliding windows on the timestamps present in the data (as opposed to the clock times when tuples arrived to the system during a particular test run are used). This ensures 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. For the experiments run using synthetic data, the default values used in the experiments are listed in Table 3. The query is an equi-join of 10 streams, i.e., $S_0 \bowtie S_1 \ldots S_9 \bowtie S_{10}$.

6.2. Results and analysis

Average output rate and execution time. We now compare *QM* to SP and MR using the synthetic data. We execute MR in two modes: with and without batching streams to reduce execution overhead [13]. The batch size is set to 100, which is similar to the max *ruster* size parameter in *QM* (Table 1). Each query is run for 25 minutes several times using these different solutions. The results averaged over all runs for the average output rate and the average execution time per tuple are depicted in Figs. 10 a and b respectively.



a) Total tuples produced

b) Total tuples produced using *real-life* datasets

Fig. 11. Experimental results.

We now consider the worst case scenario for QM, namely, when the data is uniform. In this case, *QM* tends to default to a single route per stream solution. Occasionally, due to sampling, *QM* uses two routes to process tuples from a stream. This is expected as the dataset is not skewed and can be optimally processed by a single query plan. No benefit can be gained from trying to find distinct routes during optimization and then processing *rusters* of tuples that all have the same route. In this case, we observe on average *QM* is 2.2% worse than SP in terms of their output rate, better by 39% than MR + CBR without batching, and better by 24% than MR + CBR with batching (Fig. 10 a).

For Poisson distribution, we observed that *QM* on average has 27% higher output rate than MR + CBR without batching, 18% higher than MR + CBR with batching and 44% higher output rate than SP. The average execution time per tuple (Fig. 10 b) follows a similar trend. Our results show that *QM* has a low overhead as there is no significant degradation in performance if datasets are not skewed, and has significant improvements in performance if they are.

Execution-runtime overhead. Now we compare runtime overhead of *QM* and MR.[1] Fig. 10 c reports the overheads per workload of tuples relative to the total execution cost (in black). A workload is a set of data tuples received and processed over a 1 minute time interval. For both *QM* and MR, we considered any execution cost that are not the actual query processing to be a runtime overhead. Our code calculates the time spent on such overhead versus regular query processing.

In *QM*, tuples are grouped together into the same *ruster* based on the classification. The classifier model considers the data values and the similarity of statistics when assigning routes. Thus all tuples in a *ruster* are guaranteed to share the same best route. Furthermore, *QM*'s runtime infrastructure based on *SRF* enables the query operators to route data tuples nearly overhead-free, thus eliminating the rerouting overhead associated with Eddies. The only runtime overhead incurred in *QM* is the initial classification to determine the execution plan for arriving data, which was on average measured to be very small, less than 2% of the query execution cost (Fig. 10 c). The classifier in our case is small in height (maximum 2–3 levels). Thus the *DT* traversal is cheap similar in cost to inexpensive select conditions. Additional system overheads include scheduling of the online classifier.

MR suffers from continuous re-optimization and re-learning overheads. In MR, the router continuously profiles operators and identifies "classifier attributes" to determine which operator to send tuples to next [6]. The learning overhead may be unnecessary as the best classifier attribute for an operator does not change very often, as was indicated in [1]. Furthermore, MR experiences the "backflow" overhead, where tuples get continuously routed back to the Eddy operator that has to re-examine the tuples and forward them to the next best deemed operator for processing. Batching attempts to reduce MR overhead. However, even with batching in Eddy [13] the runtime overhead is not significantly improved as "backflow" is still expensive [13]. Without any batching, the runtime overhead in MR + CBR algorithm amounted to nearly 20% of the total execution cost. These overheads limit query performance and the benefit obtained from Eddy.

Total number of tuples produced. Fig. 11 indicates the total number of tuples produced by the three execution models over time. We ran queries for over an hour and show the average output for the first 60 minutes. We can observe that over time the total number of tuples output by *QM* is significantly larger than those by either SP or MRL solutions. In the SP approach, 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-route-less system suffers from the "backflow" overhead, where tuples get

[1] We did not measure SP runtime overheads, as all data is processed using one route, and thus no overhead regarding how the data should be processed is incurred.

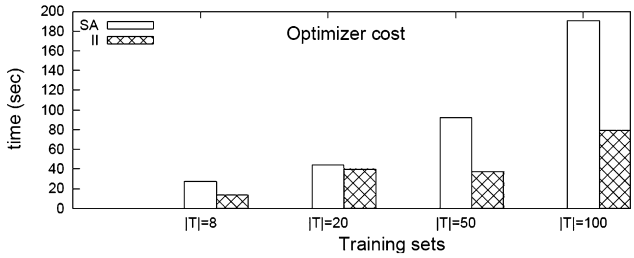


Fig. 12. Effect of training set.

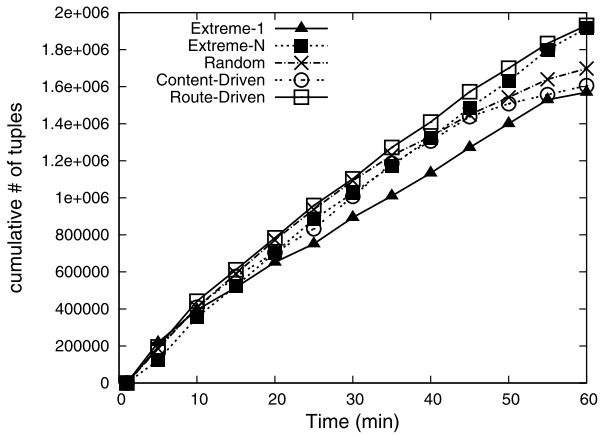


Fig. 13. Effect of start solution.

continuously routed back to the central router operator that has to re-examine the tuples and forward them to the next operator for processing.

The trend is similar to the synthetic dataset experiment, namely, query mesh outperforms the other two alternatives, although not by as much. Fig. 11 b shows the total number of tuples produced using the real-life datasets (Section 6.1).

Effect of training set size. We now study the effect of the training set size T on the cost of the *QM* optimizer and the resulting quality of *QM*s. Fig. 12 shows the optimizer search time with various sizes of T (i.e., from 8 to 100 tuples) using *SA-QM* and *II-QM* search heuristics (Section 4.7). These results support our hypothesis that larger training sets increase the *QM* optimization cost.

Effect of start solution. Here, we evaluate the effect of the *start solution* (Section 4.7) on the quality of the resulting *QM* configuration with the proposed heuristic-based search. We chose tuple output rate as our measure to estimate the quality of query meshes. Fig. 13 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*-1 which is a single-plan start solution strategy). The *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*-1 *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.

6.3. Summary of experimental results

The main findings of our experimental study can be summarized as: 1) *QM* improves execution time and output rate by up to 44% better than that experienced from state-of-art i.e., SP. 2) In the worst case when the data is uniform and thus a traditional single route solution would have been ideal, *QM*'s performance is only 2% slower than SP. The key here is that in this case QM succeeds to default to a single-plan setup. 3) The runtime overhead of *QM* is very small (2–3% at most) relative to the overall query processing cost. In all our experiments we show actual measurements of system runs, i.e., all overheads are included in the numbers shown. These experiments show that *QM* can achieve significant performance improvements over alternative solutions, and thus presents a promising new paradigm for query processing.

7. Conclusion

In this paper, we have proposed a novel query processing approach called *Query Mesh* (or *QM*). *QM* is a middle-ground solution between plan-based systems using a single plan and the continuously re-optimizing solutions like Eddies. *QM* offers

numerous advantages over the state-of-the-art techniques. First, to find the best processing strategy for different subsets of data, *QM* uses machine learning techniques to discover relationships between the data and processing routes. Second, *QM*, being comprehensive, provides innovative solutions to both query optimization and processing. Third, *QM* execution infrastructure facilitates shared operator processing and has near-zero route execution overhead. Fourth, we sketch how QM could be extended to handle query optimization issues that arise when there are imprecise statistics by supporting the selection and execution of uncertain routes. Fifth, our most important contribution is to show that *QM* can significantly improve performance over alternative solutions, and thus is a promising paradigm for query optimization. This result opens opportunities for exploration into this novel multi-route paradigm.

Acknowledgments

We thank our WPI peers for CAPE [31] and feedback. This work is supported by the following grants: NSF 0917017 & 0414567 & 0551584 (equipment grant), and GAANN.

References

[1] P. Bizarro, S. Babu, D. DeWitt, J. Widom, Content-based routing: Different plans for different data, in: VLDB, 2005, pp. 757–768.
[2] N. Polyzotis, Selectivity-based partitioning: a divide-and-union paradigm for query optimization, in: CIKM, 2005, pp. 720–727.
[3] TradingMarkets http://www.tradingmarkets.com/.
[4] L. Harris, Stock price clustering and discreteness, Rev. Financ. Stud. (1991) 389–415.
[5] A. Deshpande, C. Guestrin, W. Hong, S. Madden, Exploiting correlated attributes in acquisitional query processing, in: ICDE, 2005, pp. 143–154.
[6] R. Avnur, J.M. Hellerstein, Eddies: Continuously adaptive query processing, in: SIGMOD, 2000, pp. 261–272.
[7] A. Deshpande, J. Hellerstein, Lifting the burden of history from adaptive query processing, in: VLDB, 2004, pp. 948–959.
[8] V. Raman, A. Deshpande, J. Hellerstein, Using state modules for adaptive query processing, in: ICDE, 2003, pp. 353–365.
[9] G. Graefe, W. McKenna, The volcano optimizer generator: Extensibility and efficient search, in: ICDE, 1993, pp. 209–218.
[10] M. Astrahan, M. Blasgen, D. Chamberlin, K. Eswaran, J. Gray, P. Griffiths, W. King, R. Lorie, P. McJones, J. Mehl, G. Putzolu, I. Traiger, B. Wade, V. Watson, System R: relational approach to database management, ACM Trans. Database Syst. (1976) 97–137.
[11] B. Babcock, S. Chaudhuri, Towards a robust query optimizer: a principled and practical approach, in: SIGMOD, 2005, pp. 119–130.
[12] T.M. Mitchell, Machine Learning, McGraw–Hill, 1997.
[13] A. Deshpande, An initial study of overheads of eddies, SIGMOD Rec. (2004) 44–49.
[14] S. Viglas, J. Naughton, J. Burger, Maximizing the output rate of multi-way join queries over streaming information sources, in: VLDB, 2003, pp. 285–296.
[15] R. Nehme, E. Rundensteiner, E. Bertino, Self-tuning query mesh for adaptive multi-route query processing, in: EDBT, 2009, pp. 803–814.
[16] R. Krishnamurthy, H. Baral, C. Zaniolo, Optimization of non-recursive queries, in: VLDB, 1986, pp. 128–137.
[17] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, J. Widom, Adaptive ordering of pipelined stream filters, in: SIGMOD, 2004, pp. 407–418.
[18] A. Deshpande, Z. Ives, V. Raman, Adaptive query processing, in: Foundations and Trends in Databases, 2007.
[19] A. Arasu, S. Babu, J. Widom, The CQL continuous query language: semantic foundations and query execution, VLDB J. 15 (2006) 121–142.
[20] R. Lipton, J. Naughton, D. Schneider, S. Seshadri, Efficient sampling strategies for relational database operations, Theoret. Comput. Sci. (1993) 195–226.
[21] D. Hand, P. Smyth, H. Mannila, Principles of Data Mining, MIT Press, Cambridge, MA, USA, 2001.
[22] C. Chen, A. Jozwik, A sample set condensation algorithm for the class sensitive artificial neural network, Pattern Recogn. Lett. (1996) 819–823.
[23] M.T. Lozano, Data reduction techniques in classification processes, PhD thesis, Jaume University, Spain, 2007.
[24] M. Klazar, Bell numbers, their relatives, and algebraic differential equations, J. Combin. Theory Ser. A (2003) 63–87.
[25] S. Chaudhuri, An overview of query optimization in relational systems, in: SIGMOD, 1998, pp. 34–43.
[26] P. Selinger, M. Astrahan, D. Chamberlin, R. Lorie, T. Price, Access path selection in a relational database management system, in: SIGMOD, ACM, 1979, pp. 23–34.
[27] K. Ono, G. Lohman, Measuring the complexity of join enumeration in query optimization, in: VLDB, 1990, pp. 314–325.
[28] P.J.M. Laarhoven, E.H.L. Aarts, Simulated annealing: theory and applications, 1987.
[29] A. Deshpande, J.M. Hellerstein, Lifting the burden of history from adaptive query processing, in: VLDB, 2004, pp. 948–959.
[30] P.A. Tucker, D. Maier, T. Sheard, L. Fegaras, Exploiting punctuation semantics in continuous data streams, IEEE Trans. Knowl. Data Eng. (2003) 555–568.
[31] E. Rundensteiner, L. Ding, T. Sutherland, Y. Zhu, B. Pielech, N. Mehta Cape, Continuous query engine with heterogeneous adaptivity, in: VLDB, 2004, pp. 1353–1356.