Content-Based Routing: Different Plans for Different Data

Pedro Bizarro[†] Shivnath Babu[‡] David DeWitt[†] Jennifer Widom[‡]

[†]University of Wisconsin
Madison, WI, USA
{pedro, dewitt}@cs.wisc.edu

[‡]Stanford University
Stanford, CA, USA
{shivnath, widom}@cs.stanford.edu

Abstract

Query optimizers in current database systems are designed to pick a single efficient plan for a given query based on current statistical properties of the data. However, different subsets of the data can sometimes have very different statistical properties. In such scenarios it can be more efficient to process different subsets of the data for a query using different plans. We propose a new query processing technique called *content-based routing* (CBR) that eliminates the single-plan restriction in current systems. We present low-overhead adaptive algorithms that partition input data based on statistical properties relevant to query execution strategies, and efficiently route individual tuples through customized plans based on their partition. We have implemented CBR as an extension to the Eddies query processor in the TelegraphCQ system, and we present an extensive experimental evaluation showing the significant performance benefits of CBR.

1. Introduction

The conventional approach to query optimization is to pick a single efficient plan for a query, based on statistical properties of the data along with other factors such as system conditions. In many application domains, different partitions of the overall data accessed by a query may have very different statistical properties. For example, statistical properties of the observations collected by different sensors in a sensor network environment may be very different [14]. In such cases it can be more efficient to process the different partitions using different plans. In this paper we propose a new general-purpose query processing technique called *content-based routing* (CBR) that eliminates the single-plan restriction in current systems. CBR automatically identifies *tuple classes*—partitions of the input data that differ in relevant statistical properties—and processes the query using multiple plans, each of which is customized for an individual tuple class. CBR is low-overhead and it is adaptive, revisiting its decisions as changes in data characteristics are detected.

Adaptive approaches to query optimization have received a great deal of attention recently, with a focus on handling data properties and system conditions that may change while a query is running, e.g., [2, 8, 9, 10, 18, 28]. Our problem is different: We do not focus on adapting a single plan as data characteristics change, but rather on detecting classes of data characteristics that can be used to route different data to different plans. Note that even Eddies [2], which can potentially adapt at the tuple granularity, still uses a single plan for (nearly) all tuples at any point of time.

Our CBR algorithms are implemented as an extension to Eddies [2]. However, our approach applies to any query processing environment where the data movement can be modeled as streams, e.g., stream systems, regular database systems using iterators [19], and "pull" systems like acquisitional query processors [25]. An Eddy processes a query by routing input stream tuples through operators specific to that query. Without CBR, an Eddy makes routing decisions based on the selectivity of each operator over all tuples the operator has processed recently. Tuples are not differentiated based on content, so all tuples from the same stream source are routed identically. We denote this type of routing as *source-based routing* (SBR).

When CBR is added to Eddies, correlations between tuple content and operator selectivity are detected, and they are exploited during routing to eliminate tuples sooner, reduce latency, and improve overall system throughput relative to SBR. Next we motivate CBR using two examples.

Example 1.1. Figure 1(a) is an intrusion detection query for an enterprise network [6, 29]. The lookup table *T* may contain addresses of subnetworks in the enterprise that are exposed to the public Internet. The byte sequences represent patterns common to a specific type of network attack [6]. Figure 1(b) shows an Eddy for this query with three filter operators–O_1, O_2, and O_3–corresponding to the

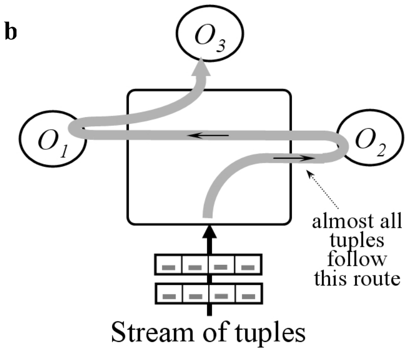
Proceedings of the 31st VLDB Conference,
Trondheim, Norway, 2005

a)

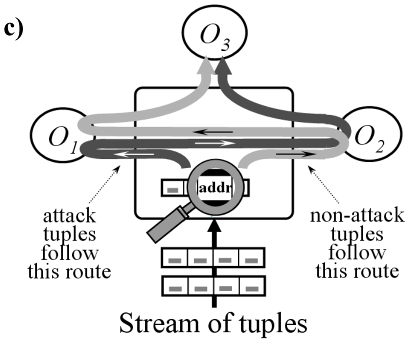
Query: "Track packets with destination address matching a prefix in table *T*, and containing the 100-byte and 256-byte sequences "0xa...8" and "0x7...b" respectively as subsequences"

SELECT * FROM packets
WHERE matches(destination, T) $\leftarrow O_1$
AND contains(data, "0xa...8") $\leftarrow O_2$
AND contains(data, "0x7...b"); $\leftarrow O_3$

b



c)



O_3
O_1
O_2
addr
attack tuples follow this route
non-attack tuples follow this route
Stream of tuples

Figure 1. (a) A Continuous Query; (b) the Eddies approach; and (c) Eddies with Content-Based Routing

three conditions, over an incoming stream *S* of network packets. Operator O_1 performs a prefix-based join on the destination address attribute of incoming *S* tuples with *T*. Operators O_2 and O_3 perform the 100-byte and 256-byte sequence matches respectively.

Let c_i denote the current average processing cost per tuple for operator O_i, and let σ_i, $0 \leq i \leq 1$, denote the current expected selectivity of O_i.[1] Suppose the following conditions hold for the example: $c_3 > c_1 > c_2$ and $\sigma_3 > \sigma_1 > \sigma_2$. Given these statistics, the Eddy's routing will converge to the ordering O_2, O_1, O_3, i.e., most tuples will follow this route as shown in Figure 1(b).

Now suppose the monitored attack is underway on a subnetwork whose prefix is not in *T*. (The subnetwork may be secured separately by a firewall.) In this case, σ_2 and σ_3 will be very high, and σ_1 will be very low for packets (tuples) coming from the attacker(s). So, O_1, O_2, O_3 will be the most efficient ordering for processing these "attack packets". For other packets, O_2, O_1, O_3 will remain the best ordering as before. Since an attack happens typically from some group of compromised hosts, CBR can distinguish between the attack and non-attack packets based on the source address, and use the appropriate ordering (Figure 1(c)). Without CBR, the Eddy will continue using the O_2, O_1, O_3 ordering, limiting performance. □

Example 1.2. Consider the following query over a distributed sensor network in a large warehouse building:

```
SELECT * FROM sensors
WHERE light < 1000 lux[2] AND temperature > 20°C;
```

To answer this query, data must be acquired from sensors. However acquiring readings from sensors is a power-consuming operation. Since sensors are power-constrained, one of the main goals of acquisitional systems is to minimize power consumed by data acquisitions [14]. Note however, that sensors that are placed close to windows receive more natural light and likely report higher temperatures than sensors located in interior rooms. Therefore, for those sensors close to windows, the probability that the predicate on light will fail may be higher than that for the temperature predicate. On the other hand, for sensors that are placed in interior locations, the probability that the predicate on light will fail may be lower than that for the temperature predicate. Therefore, instead of using a single fixed order for evaluating the two predicates across all sensors, we may want to use CBR: use different operator evaluation orders depending on the sensor location. For each sensor location, CBR chooses an operator evaluation order that evaluates the most selective operator first. On average, CBR will reduce the number of predicates evaluated per sensor and the number of data acquisitions required, resulting in significant power consumption savings in this setting. □

1.1. Contributions and Outline of Paper

Implementing CBR using Eddies introduces several challenges that we address in this paper:

- In Section 3 we define classifier attributes, an important concept in CBR.
- In Section 4 we present algorithms to automatically and efficiently learn classifier attributes, to partition the underlying data into tuple classes, and to route tuples from these classes optimally through the operators in an Eddy.
- In Section 5 we discuss the adaptive nature of our algorithms to handle changes in input data properties and system conditions while the query is running.
- Finally, in Section 6 we present an extensive experimental evaluation of CBR using a prototype implementation in TelegraphCQ. Our results show good performance improvements over not using CBR.

2. Related Work

Work related to CBR can be grouped into four categories: exploiting correlations among attributes during query processing, adaptive query processing, identifying correlations in large datasets, and computing complex statistical information over data streams.

[1] Cost is the time spent by the operator processing the tuple. Selectivity refers to the fraction of input tuples passed by the operator.

[2] A value of 1 Lux corresponds to moonlight, 400 Lux to a bright office, and 100,000 Lux to full sunlight.

The work most closely related to CBR is identifying *conditional plans* in an acquisitional query processing system [13, 14]. Like CBR, a conditional plan partitions the input data and processes each partition with a different plan. The approach taken in [14] is to learn a single good conditional plan based on an initial training sample of the data, and then to use this plan unchanged throughout query execution. That initial training is done offline, requires a large amount of collected training data, and learns the conditional plans using complex decision tree building algorithms. On the other hand, CBR uses light-weight machine learning techniques over the streaming tuples that enable a continuously adaptive approach to query processing. Thus, CBR does not require previous knowledge of the data and is not dependent on previous learned models of the world.

While many adaptive query processing systems have been built to date, the most of them use a single plan for almost all tuples at any point of time [2, 4, 5, 22, 24, 26]. Some of these systems, including Eddies, on which we have implemented CBR, process almost all of the input tuples using the current best plan, and the remaining tuples are processed using other plans to track the performance of these plans (to identify plans to change to) or to collect run-time statistics [2, 4]. Reference [23] describes a technique that combines hash join and merge join operators to take advantage of mostly-ordered inputs. Tuples following the expected order are routed to the merge join, remaining tuples to the hash join. A final phase joins tuples across the two operators to produce the complete join result. This technique, complementary to ours, can be seen as providing adaptivity within a single join operator while CBR provides adaptivity in a query plan by allowing different join orders.

There has been some recent work on identifying correlations in large datasets. None of this work has been used to identify different plans for processing different partitions of the data for a query. Reference [21] identifies sets of attributes that are correlated. Reference [11] uses the lack of correlation (independence) among attributes to build compact multi-dimensional histograms. Reference [16] uses probabilistic models like Bayesian networks to capture the statistical relationship among attributes so as to compute cardinalities accurately for intermediate results in query plans.

There has been work on computing complex statistical information over data streams, for example, decision trees [15], correlated aggregates [17], and histograms [20]. None of this work includes computing correlations between tuple content and selectivities of operators, identifying tuple classes, or finding different plans for different subsets of data.

3. Classifier Attributes

Our goal is to identify *tuple classes* where each class has a different optimal operator order for processing. CBR considers tuples classes that can be distinguished from one another based on tuple content, namely, the attributes in the tuples. In this context, different tuple classes may have different optimal operator orders if the selectivity of one or more operators is correlated with the content of one or more input attributes. Attributes used to distinguish tuple classes are called *classifier attributes*. Informally, an attribute A is called a *classifier attribute* for an operator O if the content of A is correlated with the selectivity of O. As illustrated by Example 3.1, CBR is based on identifying and exploiting such classifier attributes.

Example 3.1. Consider an input stream S processed by three operators O_1, O_2, and O_3. Let A be an attribute of tuples in S which takes one of three values a, b, or c with equal probability. Table 1 shows the respective selectivities of O_1–O_3 for the tuple classes with $A=a$, $A=b$, and $A=c$, and the overall selectivity of each operator on S tuples. Assuming O_1–O_3 have the same execution costs, if only overall selectivities are considered, then the best ordering for S tuples is O_1, O_2, O_3. However, note that the selectivity of O_2 is correlated with the value of A: the selectivity of O_2 for $A=a$ and $A=b$ is much lower than O_2's overall selectivity, and it is much higher for $A=c$. Therefore, for tuples with $A=a$ or $A=b$, the ordering O_2, O_1, O_3 will outperform O_1, O_2, O_3, while O_1, O_3, O_2 will outperform O_1, O_2, O_3 for tuples with $A=c$.

Value of A	σ_1	σ_2	σ_3
A=a	32%	10%	55%
A=b	31%	20%	65%
A=c	27%	90%	60%
Overall	30%	40%	60%

Table 1. Content Specific Selectivities □

The degree of correlation between two distributions may be specified in a number of ways [27]. In this paper we use a specification from Information Theory which is based on the concept of *gain ratio* [27], described next.

Let R be a random sample of tuples processed by an operator O. (In this paper we assume all operators are filters; an extension to non-filter operator is discussed in Section 4.5.) Let σ be the overall selectivity of O for tuples in R. Each tuple in R belongs to one of two *classes*: tuples that O passes and tuples that O drops. The *entropy* [27] of R, which is an information-theoretic metric used to capture the information content of R, is defined as:

$$\text{Entropy}\ (R) = -\sum_{i=1}^{c} p_i \log_2 p_i \qquad (1)$$

where c is the number of classes in R and p_i is the fraction of R belonging to class i. In our case $c=2$, corresponding to the tuples passed and dropped by O, so $p_1=\sigma$ and $p_2=1-\sigma$ respectively. Therefore:

$$\text{Entropy}\ (R) = -\sigma \log_2 \sigma - (1-\sigma)\log_2(1-\sigma) \qquad (2)$$

Let A be an attribute of tuples in R. Let $v_1, v_2, \ldots, v_d$ be the distinct values of A in R. The information gain of A with respect to R, which represents the increase in information about R gained by knowledge of A, is defined as [27]:

$$\text{InfoGain}(R, A) = \text{Entropy}(R) - \sum_{i=1}^{d} \frac{|R_i|}{|R|} \text{Entropy}(R_i) \quad (3)$$

Here, R_i is the subset of R with $A=v_i$, and $|R|$ ($|R_i|$) is the number of tuples in R (R_i). *Gain ratio* is a normalized representation of information gain [27]:

$$\text{SplitInformation}(A) = -\sum_{i=1}^{d} \frac{|R_i|}{|R|} * \log_2 \frac{|R_i|}{|R|} \quad (4)$$

$$\text{GainRatio}(R, A) = \frac{\text{InfoGain}(R, A)}{\text{SplitInformation}(A)} \quad (5)$$

Gain ratio is used widely in decision-tree learning algorithms (e.g., ID3 [27]) to determine the attribute that best classifies a given data set. Since classifier attributes serve a similar purpose in our case, our formal definition of a classifier attribute is based on gain ratio.

Definition 3.1 (Classifier Attribute) *An attribute* A *is a classifier attribute for an operator* O *if for any large random sample* R *of tuples processed by* O*, we have* GainRatio(R,A) > γ*, for some threshold* γ.

Example 3.2. We revisit Example 3.1. Let Table 1 now represent the selectivities computed from random samples R_1, R_2, and R_3 of tuples processed by operators O_1, O_2, and O_3 respectively. Since A takes one of values a, b, or c with equal probability, the samples will contain tuples with $A=a$, $A=b$, and $A=c$ in roughly equal proportion. We can use Equations (2) – (5) to compute the gain ratio of attribute A with respect to R_1, R_2, and R_3: $GainRatio(R_1, A) = 0.33$, $GainRatio(R_2, A) = 0.63$, and $GainRatio(R_3, A) = 0.37$. Notice that GainRatio(R_2, A) dominates the others because of the strong correlation between the selectivity of O_2 and the content of A.

Our definition of classifier attributes extends to *classifier attribute sets* where the selectivity of an operator is correlated with a set of attributes instead of with any single attribute in that set. That is, tuple classes in the input may be determined by a set of attributes instead of a single attribute. We do not consider classifier attribute sets in this paper; instead we focus on single-attribute classifiers. Note however that CBR considers multiple single-attribute classifiers when making routing decisions. While some of our techniques extend directly to classifier attribute sets, we defer a detailed exploration of this issue to future work.

4. Learning Routes Automatically

We are now ready to consider the problem of learning good content-based routes automatically for the CBR framework introduced in Section 3. We will consider a single input stream S with tuples having attributes C_1, C_2, …, C_k that are processed by operators O_1, O_2, …, O_n, and describe our *Content-Learns* algorithm to learn good content-based routes automatically in this setting. For now we will consider all operators O_1, O_2, …, O_n, and for each operator, we consider all attributes C_1, C_2, …, C_k as potential classifier attributes for CBR. In Section 4.4 we will present heuristics to prune the space of attributes and operators that we consider for CBR. Content-Learns consists of two continuous, concurrent steps:

1. **Optimization**: In this step, for each operator $O_\ell \in O_1, ..., O_n$, if one or more attributes in $C_1,...,C_k$ are classifier attributes for O_ℓ, then we keep track of the best classifier attribute for O_ℓ. Informally, we identify the attribute in $C_1,...,C_k$ based on whose content we can make the best routing decisions with respect to O_ℓ. The operator-attribute combinations identified during optimization are used for CBR by the routing step as described in Section 4.2. Details of the optimization step are described in Section 4.1.
2. **Routing**: In this step we perform CBR using the current operator-attribute combinations identified by the optimization step. If the selectivity of operator O_ℓ is not correlated with the contents of any attribute, then we do not use any O_ℓ-attribute combination but instead make routing decisions regarding O_ℓ using the selectivity of O_ℓ alone. Our routing algorithm for CBR is described in Section 4.2.

4.1. The Optimization Step of Content-Learns

The goal of optimization is, for each operator $O_\ell \in O_1, \ldots, O_n$, to identify the best classifier attribute for O_ℓ in $C_1, \ldots, C_k$. We cycle through the operators in a round-robin fashion, so each operator is considered periodically. When we consider operator O_ℓ, which we call *profiling* O_ℓ, we identify the best classifier attribute for O_ℓ. To identify the classifier attributes for O_ℓ, we have to measure the gain ratio of $C_1, \ldots, C_k$ based on a random sample of tuples processed by O_ℓ; recall Section 3. To collect this random sample R when O_ℓ is profiled, the Eddy routes a fraction of input tuples to O_ℓ before they are routed to any other operator, and notes whether O_ℓ dropped each such tuple or not. (Note that we profile operators using tuples straight from the input stream. However, in some scenarios it may make sense to profile tuples after they have been filtered by some operators. We can extend our profiling to track such conditional selectivities as in [4] which we intend to do as future work.)

Our profiling technique requires the specification of two parameters: a probability P for sampling an input tuple so that it will be routed first to O_ℓ, and a sample size to fix $|R|$. Once R has been collected, we can compute $GainRatio(R, C_j)$ for each $C_j \in C_1, \ldots, C_k$, to determine the classifier attributes for O_ℓ. If there are two or more such attributes, then the attribute with maximum gain ratio is the best classifier attribute for O_ℓ. Details of our implementation for profiling O are outlined next.

Let D_j denote the domain of potential classifier attribute C_j. For each C_j we choose a *partitioning function* f_j that partitions D_j into d partitions. If C_j is a discrete-valued attribute, we choose a hash function that maps any $v \in D_j$ to one of d buckets. If C_j is a continuous-valued attribute, we maintain running estimates of max(D_j) and min(D_j) and use a range-partitioning function to map any $v \in D_\ell$ into one of d partitions. Without loss of generality, let $v_1, v_2, \ldots, v_d$ denote the d partitions of each domain. (Note that, e.g., partition v_1 of domain D_1 is not the same as partition v_1 of domain D_2.)

Content-Learns maintains the following run-time data structures, as shown in Figure 2:

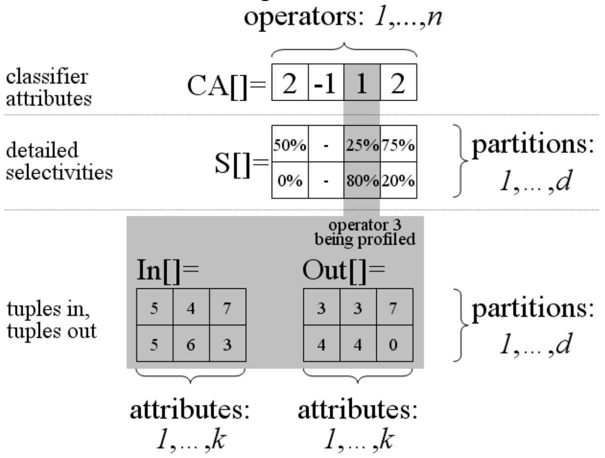


Figure 2. Run-time Data Structures

1. **Classifier Attribute Matrix, *CA*[].** CBR keeps an array that, for each operator O_ℓ, stores the attribute index of the best classifier attribute, i.e., the attribute with highest gain ratio for O_ℓ. If O_ℓ has no classifier attributes, CBR assigns $CA[\ell] = -1$. CBR recomputes $CA[\ell]$ after R random sample tuples are used to profile operator O_ℓ. In Figure 2, the classifier attribute for operator 3 (marked in gray) is attribute 1.
2. **Tuples In, *In*[] and Tuples Out, *Out*[] Matrices**: These matrices track which tuples in which partitions of all attributes pass (increments both *In*[] and *Out*[] entries) or fail (increments only *In*[] entries) the operator being profiled. For each one of the R random sample tuples, k entries are updated in each one of these matrices. The entries to be updated are (j,i), with $j = 1,\ldots,k$, and $v_i = f_j(t.C_j)$.
3. **Detailed Selectivities Matrix, *S*[].** Each column in this matrix stores the running selectivities for an O_ℓ–C_j operator–classifier-attribute pair. Entries in the matrix are updated at two different times:
   (i) *Run-time*: Each time a tuple passes or fails an operator, one entry in this matrix is updated.[3] For a tuple t being processed by O_ℓ, the column to update in the matrix is ℓ, and the row is $v_i = f_j(t.C_j)$, with j being the index of the classifier attribute for O_ℓ, i.e., $j = CA[\ell]$.
   (ii) *Initialization*: After completing profiling operator O_ℓ and finding its classifier attribute C_j, CBR updates O_ℓ's column: $S[\ell,i] \leftarrow Out[j,i]/In[j,i]$, with $i = 1,..,d$. If $In[j,i]=0$, then $S[\ell,i] \leftarrow W[\ell]$, where $W[\ell]$ is the overall selectivity of operator O_ℓ as described next.
4. **Overall Operator Selectivities, W[].** This matrix (not shown in Figure 2) is non-CBR specific information and it is kept both by CBR and by the non-CBR implementation in TelegraphCQ. $W[\ell]$ tracks the recent overall selectivity of operator O_ℓ over all tuples processed by O_ℓ.

Once we have collected the random sample R of tuples processed by operator O_ℓ while profiling O_ℓ, we can compute $GainRatio(R,C_j)$ (Equation (5)) for all $C_j \in C_1,\ldots,C_k$ using matrices I and O. From Equation (2), $Entropy(R)$ depends only on the overall selectivity of O_ℓ over R, which is the number of output tuples over all tuples profiled: $\left(\sum_{i=1}^{d} Out[j,i]\right)/|R|$ for any j.

Similarly, $Entropy(R_i)$ in Equation (3) for $InfoGain(R,C_j)$ depends only on I and O. Finally, $|R_i|$ in Equations (3) and (4) for $InfoGain(R,C_j)$ and $SplitInformation(C_j)$ is equal to $In[j,i]$.

So far we have seen how the classifier attributes for O_ℓ can be determined by profiling O_ℓ. If there are one or more such attributes, then the attribute with maximum gain ratio, denoted C_{max}, is the best classifier attribute for O_ℓ. Even though C_{max} is the best classifier for O_ℓ, using the O_ℓ-C_{max} combination for CBR may not improve overall performance. (Details of using operator-attribute combinations during routing are given in Section 4.2.) The reason it may not improve performance is that we may already be using some other operator-attribute combinations for CBR. The additional benefit that O_ℓ-C_{max} gives in this context may be lower than the extra routing overhead that it incurs. We use a simple yet accurate technique to estimate the overall benefit of adding O_ℓ-C_{max} for CBR in the current context. We simply start using O_ℓ-C_{max} for CBR alongside the other operator-attribute combinations being used already, and measure the overall performance with and without O_ℓ-C_{max}. We characterize overall performance in terms of the rate at which the Eddy can process input tuples, which can be measured at negligible overhead. If the overall performance improves when we start using O_ℓ-C_{max} for CBR, then we stick with it until the next time O_ℓ is profiled. (Just before we start profiling an operator O_ℓ, we stop using any O_ℓ-attribute combination being used for CBR.) Otherwise, we stop

[3] The formula used to update selectivity after a tuple is known to pass or fail an operator is: selectivity = selectivity * α + pass * (1- α), where selectivity is a percentage between 0 and 100, pass is 100 if the tuple passes the operator or 0 if it is dropped, and $\alpha = 0.95$.

using O_ℓ-C_{max}. In either case, we move on to profile the next operator in our round-robin schedule. Note that after computing gain ratio values for $C_1,\ldots,C_k$ while profiling O_ℓ, we may realize that O_ℓ has no classifier attributes. Then, we move directly to profile the next operator.

4.2. The Routing Step of Content-Learns

In this section we describe how we extend the original Eddy routing algorithm to incorporate the operator-attribute combinations identified in the optimization step for CBR. This algorithm routes tuples to operators according to a probability that is inversely proportional to the operators' selectivities (stored in matrix W in our implementation). We call this algorithm *Source-Based Routing* (*SBR*).[4]

When an Eddy using Content-Learns has to route a tuple t to one of operators $O_1,\ldots,O_n$, the Eddy routes t to the operator with minimum value σ, where σ is defined as follows for an operator O_ℓ:

- If O_ℓ is tagged with classifier attribute C_j, then σ is the expected selectivity of O_ℓ for tuples t' with $f_j(t'.C_j)=f_j(t.C_j)$, which is equal to $S[\ell,i]$ where $f_j(t.C_j)=v_i$ and $j=CA[\ell]$. (We have used the same notation as in Section 4.1.)
- If O_ℓ is not tagged with a classifier attribute, then σ is $W[\ell]$, the expected overall selectivity of O_ℓ, which is the same value as used by the SBR algorithm.

Intuitively, for operators that have a classifier attribute, CBR uses the content-specific selectivity of the operator while making routing decisions. The content-specific selectivity is available from the selectivity matrix for the operator. For operators that do not have a classifier attribute, CBR uses the overall selectivity of the operator across all tuples as done by SBR.

4.3. Overheads and Benefits of CBR

There are two forms of overhead associated with CBR: the *routing overhead* of evaluating content-based conditions while making routing decisions, and the *learning overhead* of learning and maintaining good routes automatically. The routing overhead was designed to be very low, as it is incurred each time a tuple is routed by the Eddy. The learning overhead is amortized across a large number of tuples as this overhead is incurred once after |R| sample tuples are observed. Section 6.8 presents experiments where the overheads of CBR can be observed to be very low.

The benefit of CBR comes from finding routes that drop tuples sooner. As such, the benefit of CBR is proportional to the percentage of time that a query spends evaluating operators. In Section 6.7 we explore the performance of CBR while varying operator costs.

4.4. Pruning Operators and Attributes

So far we considered all attributes and all operators as potential candidates for CBR. We now describe some heuristics to prune this space. These heuristics often reduce the learning overhead of CBR significantly without any noticeable effect on the quality of content-based routes.

CBR applies when optimal operator orderings differ across input tuple classes. If an operator is very cheap or very selective relative to the other operators, or both, then its position will mostly remain unchanged across the orderings. This intuition translates into an effective pruning heuristic where we do not consider very inexpensive or very selective operators for CBR. Similarly, we can ignore operators that are very expensive or not very selective with respect to the other operators because their position is likely to remain unchanged across those orderings as well.

Similar to pruning operators, there are some effective heuristics to prune the attributes considered for CBR. For example, we can ignore monotonically increasing (or decreasing) attributes such as timestamps or sequential identifiers which typically are generated synthetically. Discrete-valued attributes with large domains, e.g., a comments string attribute, may be ignored. (It is advisable to ignore long attributes as classifier attributes for CBR to keep routing overhead low.) While it is not hard to detect such attributes automatically, the required information often is available from the schema definitions.

4.5. CBR for Non-Filter Operators

We have focused so far on filter operators that either pass or drop an input tuple. This class does not capture, for example, non-foreign-key join operators, limiting the scope of our techniques. However, our techniques apply to non-filter operators with one simple modification. We have used the filter property of an operator only to compute entropy in Equation (2) which contributed to the gain ratio value used to identify classifier attributes. The two-class notion of passed and dropped tuples is meaningless for non-filter operators whose "selectivity"– the expected number of tuples produced per input tuple– can be any non-negative real number. Our real purpose here is to quantify the skew in content-specific operator selectivities with respect to the overall selectivity. Gain ratio is one proven technique to quantify this skew. There are other techniques, e.g., variance, which apply to non-filter operators. Therefore, our techniques for CBR apply to non-filter operators provided the gain-ratio-based test for classifier attributes is replaced by an appropriate test that applies to non-filter operators.

[4] We call this algorithm Source-Based Routing because without looking at the content, an Eddy treats all tuples coming from the same source the same way.

5. Adaptivity

Since the Eddies architecture has been designed to support adaptive processing, a relevant question to ask is how our extensions to support CBR in Eddies affect adaptivity. Adaptivity refers to the ability of the system to find an efficient plan quickly for the new data and system characteristics when these change. The changes in the data stream characteristics that can affect routing decisions are changes in operator selectivities and changes in correlations between attributes and operators' selectivities.

CBR increases both the learning overhead and the routing overhead of Eddies. Fundamentally, reducing run-time overhead is at odds with improving adaptivity [9]. The approach we have adopted in this paper is to keep run-time overhead as low as possible while being as adaptive as the SBR routing policy in TelegraphCQ.

To be as adaptive as SBR, CBR keeps the operator selectivity matrix W up to date. Note that W is common across both policies. In exchange, CBR settles for slower adaptivity with respect to changes in classifier attributes by profiling only one operator at a time. This design decision may fail to detect a new correlation if the classifier attribute for an operator changes between two of its profiling phases. However, in spite of this decision, CBR is designed to never be less adaptive than SBR. Example 5.1 illustrates why.

Example 5.1: CBR as adaptive as SBR. Consider that CBR finds C_j to be the classifier attribute for O_ℓ. Then, when routing tuple t, CBR assumes the selectivity of O_ℓ to be $S[\ell,i]$, with $v_i=f_j(t.C_j)$. However, it may be the case that the correlation between C_j and O_ℓ no longer holds since O_ℓ was last profiled due to one of two reasons:

- *No attribute is correlated with* O_ℓ. If this is the case, then the selectivity of O_ℓ is given by $W[\ell]$ and not $S[\ell,i]$. However, if C_j is not actually correlated to O_ℓ, then all entries $S[\ell,i]$, with $i=1,\ldots,d$ will quickly converge to $W[\ell]$ (because CBR updates entries in $S[]$ as frequently as those in $W[]$).
- *Another attribute is correlated with* O_ℓ. If this is the case, then we have an argument for more aggressive content-based routing statistics tracking (e.g., profiling multiple operators simultaneously as done in [3]), not less. In any case, given that C_j is not correlated with O_ℓ, all entries $S[\ell,i]$, with $i=1,\ldots,d$ will still quickly converge to $W[\ell]$. □

The assumption behind the current CBR design is that operators' selectivities change more frequently than the correlations between operators and tuple content. As such, selectivity is tracked continuously (quick to detect changes) while profiling is performed only for a sample of the tuples (slower to detect changes). For example in the real-life dataset that we worked with we observed changes in selectivity from 1% to 96% in one operator while the best classifier attribute for that operator stayed the same (Section 6.6).

6. Experimental Results

We now describe an experimental evaluation of our CBR techniques using a prototype implementation in TelegraphCQ [9]. We evaluate the CBR prototype using both synthetic and real life datasets. The synthetic dataset is used to evaluate CBR by varying parameters hard to control in a real-life dataset: skew, selectivity, and aggregate selectivity. The real-life dataset is used to evaluate CBR's adaptivity and performance under varying operator costs and overhead.

6.1. Datasets

The prototype implementation of CBR was evaluated with both a synthetic and a real-life dataset, described below:

- ***Stream-Star***: We created a synthetic benchmark, Stream-Star, based on a star schema. Instead of a central fact table, we used a data stream S.[5] Our experiments use N-way join queries of the following form which join incoming S tuples with N dimension tables d1, d2, …, dN:

```
SELECT * FROM stream S, d1, d2, …, dN
 WHERE s.fkd1 = d1.pk      // Operator Op1
   AND s.fkd2 = d2.pk      // Operator Op2
   …
   AND s.fkdN = dN.pk;     // Operator OpN
```

  Each stream consisted of 100,000 tuples. Depending on the query, between two and eight dimension tables containing 10,000 tuples each are used. Stream S contains tuples with a single classifier attribute, *attrC*, which is correlated with the selectivities of all operators. (We note that in the real-life dataset described next, different operators can have different correlated attributes and these correlations can change, appear, or disappear with time. CBR worked equally well in both settings.). Our stream generator is able to produce tuples with any kind of non-independence between the classifier attribute *attrC* and the selectivity of the join operators. For example, it can generate a stream with the characteristics shown in Table 1.

- ***Lab***: The *Lab* dataset is a trace of readings from 54 sensors in the Intel Research, Berkeley Lab. The readings were taken between end of February and beginning of April of 2004. The schema consists of one single stream, *sensors*. Tuples in the stream have attributes *light*, *humidity*, *temperature*, *voltage*, *sensorID*, and timestamp information (year, month, day, hour, minute, and second) [14]. We cleansed this

[5] A star schema was chosen for two reasons: (i) queries over streams normally refer to one single stream source that joins with zero or more local tables; and (ii) data stream applications have streams that represent facts, e.g., traffic information, which then join with dimensions, e.g., speed sensors and cars.

dataset by removing tuples with missing values or impossible values (e.g., negative humidity) that sometimes happen when the sensor batteries run low. There are 2.2 million records in the cleansed dataset. For this dataset the readings are sent to TelegraphCQ in generation order, as they would if the tuples were being collected from the sensors in real-time.

6.2. Algorithms, Metrics, and Default Values

Section 4.2 described most of the details of our implementation of CBR in TelegraphCQ, *Content-Learns* (*Learns* in the figures), and the non-content-based *SBR* algorithm in TelegraphCQ. To illustrate the differences between the learning overhead and the routing overhead of CBR, in the Stream-Star experiments we include a routing algorithm called *Content-Knows* (*Knows* in the figures) which does not need to learn classifier attributes automatically. Instead, Content-Knows is a theoretical bound that simulates a routing policy that is "told" which attribute is the best classifier for each operator and what is the best routing order for each class.

In addition to the running time, we also use the number of routing calls as a performance metric. The number of routing calls shows a clear picture of the quality of the routing algorithm: a bad routing algorithm will miss opportunities to route a tuple to the most selective operator, e.g., a tuple may be routed several times before being dropped. In addition, the improvement in routing calls due to using Content-Learns instead of SBR acts as a ceiling in the improvement we can expect in total running time.

Unless otherwise stated, the default values used in the experiments are the ones listed in Table 2.

Parameter	Defaults	Comment
P	6%	Tuple sampling probability
|R|	150 tuples	Sample size to compute GainRatio
d	24	Number of buckets in hash partitions
Confidence	95%	Confidence intervals in graphs

Table 2. Defaults used in experiments and graphs

6.3. Varying Skew

In this section we use the Stream-Star dataset to show how CBR performs in the presence of skew among the content-specific selectivities of operators. We set the stream to have as many tuple classes as joins. (Each tuple class is identified by a unique value of attribute *attrC*.) Skew was created by setting the selectivity of one operator to *A*, and setting the selectivity of the all other *N-1* operators to *B*, as shown in Table 3.

A was varied from 5% to 95% with *B* varying accordingly such that the overall aggregate selectivity remained constant at 5%. (Section 6.4 reports experiments where selectivities are chosen randomly and Section 6.5 reports experiments where the aggregate selectivity is varied.) There were 8 other attributes in tuples in the stream not correlated with the selectivities of the operators. Thus, Content-Learns must learn that, among all these attributes, *attrC* is the best classifier attribute for all operators. The *N*-way join query was run for two, four, six, and eight join operators. Due to space constraints, we only show results for two and six joins in Figure 3 and Figure 4.

	Op1	Op2	...	OpN
Class 1	*A*	*B*	...	*B*
Class 2	*B*	*A*	...	*B*
...	...	...	...	...
Class N	*B*	*B*	...	*A*

Table 3. Selectivities for class/operator pairs

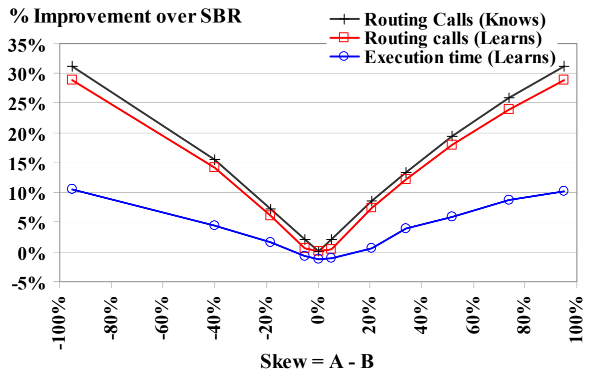


Figure 3. Improvement with varying skew (2 joins)

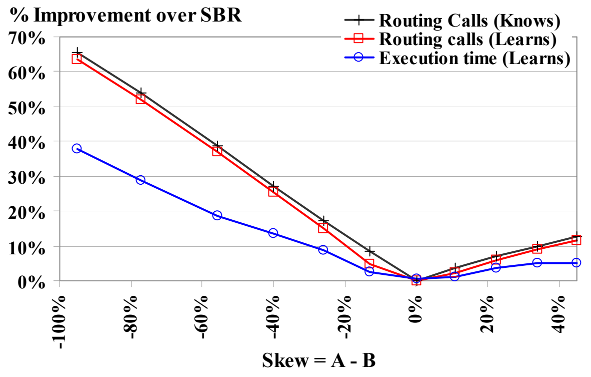


Figure 4. Improvement with varying skew (6 joins)

Note that when *A*<*B* (negative skew), a good routing policy should exploit the selectivity skew by routing tuples first to the lower selectivity operator corresponding to *A*. When *A*>*B*, a good routing algorithm will avoid the operator with selectivity *A* and route tuples through all the other operators first.

Overall, the higher the skew between *A* and *B*, especially when *A*<*B*, the greater the extent by which Content-Learns outperforms SBR. At most, Content-Learns outperforms SBR by performing 67.8% fewer routing calls (with eight operators and the largest skew). Across all experiments, when *A*<*B*, Content-Learns required on average 26.9% fewer routing calls and when *A*>*B*, Content-Learns required 10.2% fewer routing calls. That is, it is more useful to know which operator is different by being more selective than it is to know which operator is different for being less selective. This happens because more selective operators will appear earlier in operator orderings affecting more tuples and having greater performance impact than less selective operators that appear later in the operator order.

6.4. Varying Selectivities

In Section 6.3, the choice of selectivities made routing tuples to operators difficult for SBR because all operators appeared to be equally selective. Each operator had selectivity *A* for one class of tuples and *B* for all other classes. Thus, in all cases, to SBR, all operators appeared to have a selectivity of (A + B * (N-1))/N, for the N-way join query.

We continue to use the Stream-Star dataset in the following experiments. Each query was run against 50 different streams. Attribute *attrC* was correlated with the selectivities of the operators. However, this time we assigned random selectivities to each operator. As before, we included additional attributes (constants, sequences, and foreign keys) whose content was not correlated with any of the selectivities of the operators. Figure 5 shows that Content-Learns is very effective at learning the right classifier; out of the 16 million routing calls, Content-Learns used the wrong classifier only 6.4% of the time.

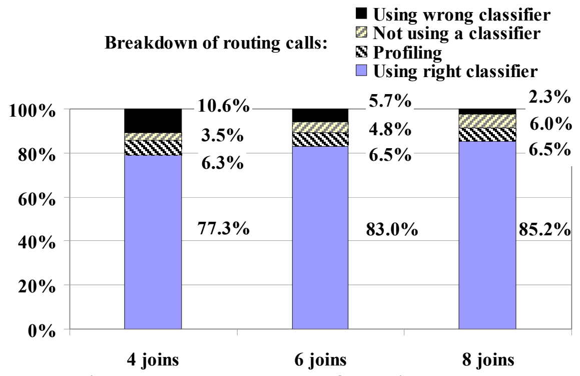


Figure 5. Breakdown of routing calls

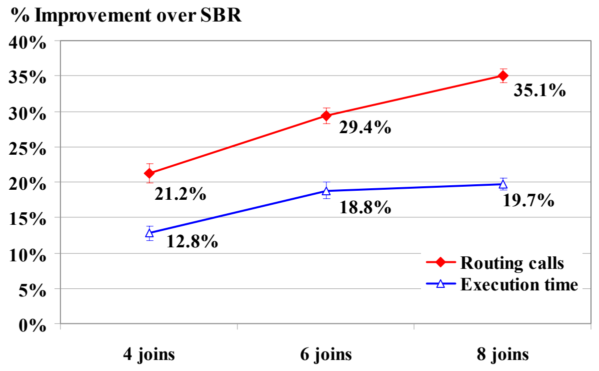


Figure 6. Improvement with random selectivities

Figure 6 shows the improvement of Content-Learns over SBR both in terms of routing calls and total execution time. Note that the larger the number of operators involved, the more opportunities are available for improvement.

6.5. Varying Aggregate Selectivity

In Section 6.3 the overall aggregate selectivity was kept at 5%. In Section 6.4 the operator selectivities were randomly selected without any guarantee on the aggregate selectivity. On average, the aggregate selectivity was 8% across all streams. This section explores the space of aggregate selectivities from 5% to 35%. For this experiment, we ran a 6-way join query over Stream-Star with the operators having random selectivities under the restriction that the overall aggregate selectivity was kept at some pre-determined value. The aggregate selectivity is varied in Figure 7. Each point in the plot represents the average improvement of CBR over SBR for 50 streams of 100,000 tuples each.

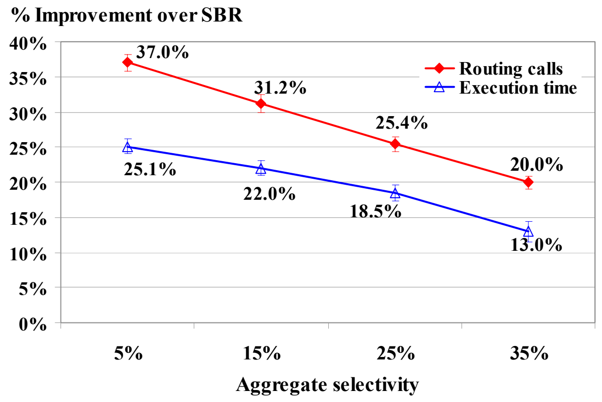


Figure 7. Improvement with varying aggregate selectivity (6 joins)

6.6. Adaptivity Experiments

In this and subsequent sections, we will use the real-life *Lab* dataset. In the *Lab* dataset the best classifier attributes for operators change, appear, and disappear as time progresses. Query Q1 is used to illustrate how CBR adapts in the presence of variations of selectivity and variations of correlation.

```
SELECT * FROM sensors WHERE light>500
```
(Q1)

For example, the amount of light varies with the time of day in the obvious way: during the day there is more light than during the night. However, the predicate that evaluates “light>500” may actually be correlated with *sensorID* and not with, say, *hours*. This happens because some sensors are placed in illuminated areas like windows or in offices, while others are placed in hallways with less human activity and light. Furthermore, if the operator that checks if light>500 evaluated to true for, say, sensor 7, at 12h34pm, then it is very likely that it will evaluate to true for the same sensor 1 minute later. During the night, when it is dark and when people have left the building, the operator that tests for light will almost always have zero selectivity. When that is the case, no attribute can be found to be correlated with the operator; that is, if the selectivity of an operator is 0% (or 100%), then all attributes have zero information gain ratio.

Figure 8 shows the result of running query Q1 for three days and nights of data. The top part of the figure shows the selectivity of the predicate; note that during the day the selectivity does not reach 100%, thus, some sensors are in darker areas than others. In the middle of the figure, we show what attribute is most correlated with the selectivity of the operator for each moment in time. *sensorID* is almost always the best classifier attribute. Sometimes, especially during transitions night-day or day-night, the attribute *hours* is the best classifier attribute. In three other moments, one of the other attributes was found to be the best classifier. In all other periods not covered by any of the black lines from “sensorID”, “hours”, and “All others”, CBR could not find any attribute correlated with the selectivity of the operator (because its selectivity was 0%). Finally, the lower part of the Figure 8 shows how the information gain of attribute *sensorID* varies with time. Although Figure 8 is indicative that data characteristics in the stream change dramatically and that CBR is able to adapt to them, queries with only one operator (like query Q1) do not require good routing policies.

To evaluate the adaptivity of CBR on the *Lab* dataset, we ran queries like query Q2 below:

```
SELECT * FROM sensors                                (Q2)
 WHERE light         BETWEEN lowL AND highL
   AND temperature BETWEEN lowT AND highT
   AND humidity    BETWEEN lowH AND highH
   AND voltage     BETWEEN lowV AND highV;
```

For each attribute, the parameter lowX was randomly chosen from among the lowest 25% values in the attribute’s domain and the parameter highX was randomly chosen from the highest 25% values in the domain.

For 50 different random Q2 queries, we obtained on average an improvement of 8% in routing calls, 5% in total execution time, 7% in time spent evaluating operators, and 18% in routing calls needed until a tuple is dropped. The results are positive but modest. Two reasons explain why CBR does not provide greater improvements:

(i) There are overheads in TelegraphCQ unrelated to routing or operator execution [12], for instance, the IO required to get the stream tuples from the network and the overhead of queuing those tuples before they get to the Eddy. These overheads limit the benefit we can obtain from a better routing policy. In Section 6.7 we explore operators with higher execution costs and show that as operator costs increase, CBR’s performance improves.

(ii) CBR can only obtain improvements when the selectivities of the operators are not close to 0% or 100%. As seen in Figure 8 there are large intervals in the dataset where the selectivities of operators stay very close to 0% or very close to 100%. The selectivity graphs for the other operators (not shown) have similar intervals very close to 0% or to 100%. For Q2, this happened 57.2% of the time, CBR yields improvements only on the other 42.8% of the time.

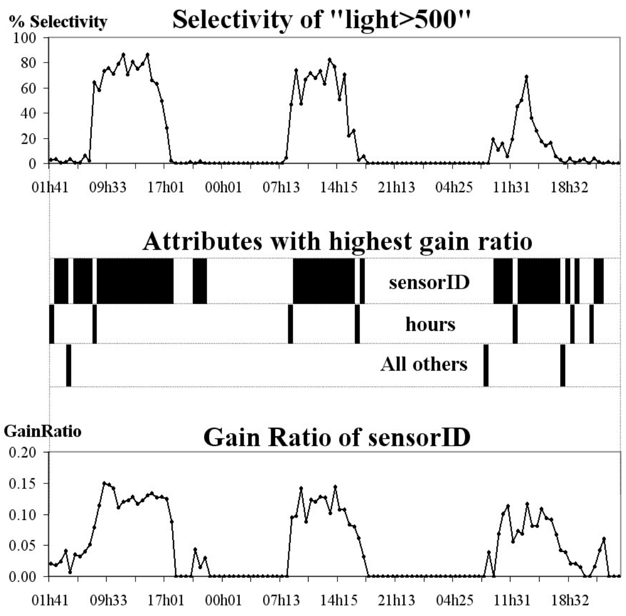


Figure 8. Change in selectivity, best classifiers, and gain ratio

6.7. Varying Operator Cost

In this section we vary the time it takes an operator to process a tuple and report the corresponding CBR’s performance improvements. There are two motivations for exploring the space of higher operator costs: (i) there are applications where operator costs can be very high (for example, [14] reports operator costs, expressed in terms of power consumption, with cost differences of two orders of magnitude between operators) and (ii) the implementation of TelegraphCQ we used has overheads [12] that overshadow operator costs. By increasing the operator costs, we decrease the weight of these overheads in the overall execution time.

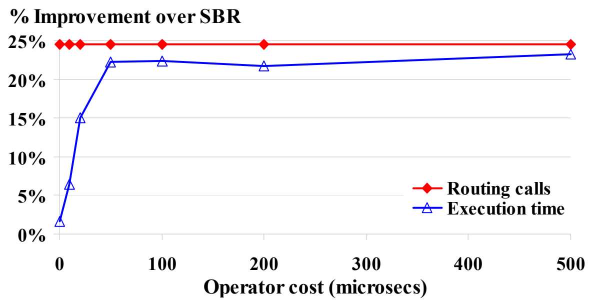


Figure 9. Improvement with varying operator cost

Figure 9 shows the improvement in performance from using Content-Learns in queries like Q2. The improvement in the number of routing calls remains constant throughout and is shown only for reference. The improvement in execution time improves as the operator cost increases. The increase in operator cost was obtained by running CPU intensive computations every time a tuple had to be processed by an operator.

6.8. Run-time Overhead of CBR

As mentioned in Section 4.3, CBR has two overheads: routing overhead and learning overhead. We instrumented the code to determine the time spent by each of these overheads. The routing overhead was measured as the time taken by the function that performs routing decisions (the algorithm of Section 4.2). The learning overhead was measured as the time taken for updating the data structures described in Section 4.1 together with the time spent computing the best classifier attributes for each profiled operator. We also instrumented the SBR version to report its routing and updating overheads (although SBR does not determine classifier attributes, it spends time updating statistics as well). Figure 10 reports, per routed tuple[6], these overheads, in microseconds, for both SBR and CBR policies for the experiments of Section 6.4 (*Stream-Star* dataset). For both policies, the total overhead (routing together with learning and updating statistics) was around 4-5% of the total execution time.

In addition, we also measured the worst case scenario for CBR: when the routing policy is irrelevant, as is the case for queries with one operator only. If there is just one operator, no benefit can be gained from different routing policies. Thus, differences in total execution time must be from overhead and not from better decisions. For this experiment we run query Q1 from Section 6.6 over the *Lab* dataset (without using the operator delays mentioned in the previous section) for both CBR and SBR. The average over 10 runs of query Q1 shows that, when no benefit is possible, CBR is about 0.8% worse than SBR in total execution time.

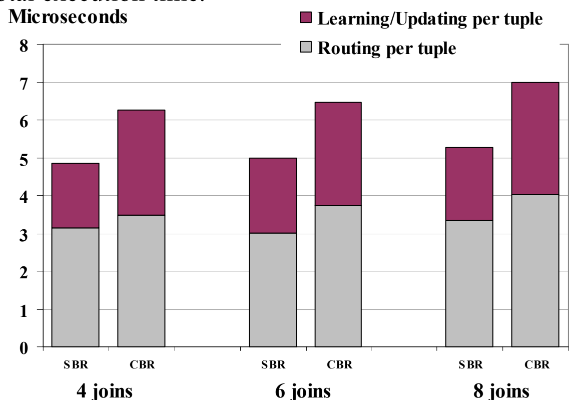


Figure 10. Per tuple overhead

[6] Per tuple overhead is computed as total overhead divided by the number of *routing calls*. Note that the number of routing calls is equal to the number of times the Eddy has to route tuples.

7. Conclusions

In this paper we proposed a new concept: assigning different query execution plans for subsets of data with different statistical properties. As such, we developed a new query processing technique called content-based routing that eliminates the single-plan restriction in current systems. We showed how the adaptive architecture of a data stream management system, TelegraphCQ, can be extended with content-based routing to enable the system to exploit correlations between tuple content and operator selectivities.

Our most important contribution was to show that content-based learning and routing can be simultaneously inexpensive and adaptive while still achieving significant performance improvements. We presented the Content-Learns algorithm which learns good content-based routes automatically, and we showed that the overhead of maintaining the extra statistics and computing classifier attributes is negligible when compared to a non-CBR algorithm.

Our prototype implementation indicates that CBR can improve execution time by up to 35% when compared with routing based on operator statistics alone. For all queries with more than one operator, CBR yielded better results than SBR, both in the number of routing calls as well as in absolute running time. In addition, the performance comparison between Content-Learns and Content-Knows showed that Content-Learns learns classifier attributes correctly in real time.

8. Future Work

While CBR appears to be a promising approach for query processing, many issues remain to be explored:

- In this paper we considered only operator-attribute combinations as the basis for CBR. This approach could be extended to consider combinations of operator sets (or lists) and attribute sets. The relevance of classifier attribute sets was discussed briefly in Section 3. Operator sets for CBR are useful in the presence of non-commutative operators and also to reduce routing overhead.
- Some run-time parameters in our implementation of CBR are not yet learned automatically. These include the number of partitions used by the hash functions, the sampling rate, and the sample size for computing gain ratio.
- Although our work is not strictly comparable with [13] it is useful to contrast some high level design decisions. In [13] the goal is to minimize power consumption over of a large network of sensors. This is achieved by collecting large amounts of data before running queries, processing the data with heavy machine learning algorithms to learn conditional plans, and distributing those plans to sensors. Our work, though not covering all sensor acquisitional scenarios, is much more

adaptive: it uses lightweight techniques to detect correlations and produce the different plans for different data (a form of conditional plans) on the fly. An interesting avenue of future work is trying to combine the light-weight adaptive nature of our techniques with the distributed nature and power-consumption minimization of acquisitional systems.

Acknowledgements

We would like to thank Sam Madden and Amol Deshpande for providing us the Lab dataset. We also would like to thank Sailesh Krishnamurthy, Amol Deshpande (again), Joe Hellerstein, and the rest of the TelegraphCQ team for providing TelegraphCQ and answering all our questions.

References

[1] A. Arasu, B. Babcock, S. Babu, M. Datar, K. Ito, R. Motwani, I. Nishizawa, U. Srivastava, D. Thomas, R. Varma, J. Widom. STREAM: The Stanford Stream Data Manager. *IEEE Data Eng. Bull*. 26(1): 19-26 (2003).

[2] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD '00.*

[3] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *ACM SIGMOD '04*.

[4] S. Babu and J. Widom. StreaMon: An adaptive engine for stream query processing. In *ACM SIGMOD '04*. Demonstration proposal.

[5] S. Babu, P. Bizarro, and D. DeWitt, Proactive Re-optimization. In *ACM SIGMOD '05*.

[6] J. Beale. *Snort 2.1 Intrusion Detection*. Syngress Publishing, 2004.

[7] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *ACM SIGMOD '02*.

[8] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams – a new class of data management applications. *In VLDB '02*.

[9] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR '03.*

[10] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *ACM SIGMOD '03*.

[11] A. Deshpande, M. N. Garofalakis, and R. Rastogi. Independence is Good: Dependency-Based Histogram Synopses for High-Dimensional Data. In *ACM SIGMOD '01*.

[12] A. Deshpande. An initial study of overheads of eddies. *SIGMOD Record*, 32(4), Dec. 2003.

[13] A. Deshpande, C. Guestrin, S. Madden, J. Hellerstein, and W. Hong. Model-Driven Data Acquisition in Sensor Networks. In *VLDB '04.*

[14] A. Deshpande, C. Guestrin, S. Madden, W. Hong. Exploiting Correlated Attributes in Acquisitional Query Processing. In *ICDE '05*.

[15] P. Domingos and G. Hulten. Mining high-speed data streams. In *SIGKDD '00*.

[16] L. Getoor, B. Taskar, D. Koller. Selectivity Estimation using Probabilistic Models. In *ACM SIGMOD '01*.

[17] J. Gehrke, F. Korn, and D. Srivastava. On computing correlated aggregates over continual data streams. In *ACM SIGMOD '01*.

[18] L. Golab and T. Ozsu. Issues in data stream management. *SIGMOD Record*, 32(2):5–14, June 2003.

[19] G. Graefe. Query Evaluation Techniques for Large Databases. *ACM Comput. Surv*. 25(2): 73-170. 1993.

[20] S. Guha, N. Koudas, and K. Shim. Data-streams and histograms. In *Proc. of the 2001 Annual ACM Symp. on Theory of Computing*, 2001.

[21] I. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *ACM SIGMOD '04*.

[22] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *ACM SIGMOD '99*.

[23] Z. Ives, Alon Y. Halevy, Daniel S. Weld. Adapting to Source Properties in Processing Data Integration Queries. In *ACM SIGMOD '04*.

[24] N. Kabra and D. J. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *ACM SIGMOD '98*.

[25] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong. The Design of an Acquisitional Query Processor for Sensor Networks. In *ACM SIGMOD '03*.

[26] V. Markl, V. Raman, D. Simmen, G. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *ACM SIGMOD '04*.

[27] T. Mitchell. Machine Learning. McGraw-Hill, 1997.

[28] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *CIDR '03*.

[29] Snort: The Open Source Network Intrusion Detection System. http://www.snort.org.

[30] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO -DB2's LEarning Optimizer. In *VLDB '01.*