Computer Sciences Department

Content-Based Routing for Continuous Query-Optimization

Pedro Bizarro
Shivnath Babu
David DeWitt
Jennifer Widom

Technical Report #1511

July 2004

UNIVERSITY OF
WISCONSIN
MADISON

Content-Based Routing for Continuous Query-Optimization

Pedro Bizarro
University of Wisconsin - Madison
Department of Computer Sciences
pedro@cs.wisc.edu

Shivnath Babu
Stanford University
Computer Science Department
shivnath@cs.stanford.edu

David DeWitt
University of Wisconsin - Madison
Department of Computer Sciences
dewitt@cs.wisc.edu

Jennifer Widom
Stanford University
Computer Science Department
widom@cs.stanford.edu

Abstract

Current Data Stream Management Systems do not fully exploit their adaptive nature to handle complex queries. To date, such systems route stream tuples to operators or operator paths based only on operator-level statistics. Their optimizers ignore non-independent distributions, attribute correlations, and tuple content. In this paper, we propose a content-based *tuple routing approach which, together with histogram-like statistics, allows a stream query processing system to exploit non-independent distributions and correlations instead of being hurt by them. We present a framework for content-based routing in a stream query processing system and an algorithm for learning content-based routes automatically and efficiently. We present an extensive experimental evaluation of content-based routing based on a prototype implementation in TelegraphCQ. Our results clearly indicate that good content-based routes can be learned quickly and efficiently to improve query performance significantly. We believe that any system that processes complex queries over possibly non-uniform data, even in a non-stream environment, can profit by being simultaneously adaptive and content-aware.*

1. Introduction

One of the biggest advantages of database systems is the use of declarative languages. As such, the system, not the user, is responsible for devising the plans to execute queries. Since the beginning, almost all commercial systems use a plan-first, execute-next approach [34] using an optimizer based on dynamic programming to determine a "best" plan. These optimizers rely heavily on statistics to determine the size of intermediate tables and decide join orderings and access methods. A key limitation of the current approach is that errors for estimates in intermediate statistics grow exponentially [22]. One solution is to use some form of query re-optimization: modifying query plans during execution [6, 23, 24, 27], or Eddies [3], a tuple-routing approach to query optimization. Eddies provide the most flexible approach possible as the plan is chosen on a tuple-by-tuple basis. That is, it is possible to have the simultaneous execution of different plans for different tuples of the same query. Eddies work as follows: every time a tuple arrives at the Eddy (from either a data source or the output of an operator), it decides where to route the tuple next. Routing decisions employ a greedy approach and rely on simple operator statistics like selectivity, cost, and queue length.

As data streams applications are developed, we expect that these applications will have streams with increasing tuple rates, and that users will pose complex decision support queries. Consider the following example. Imagine that we want to select those stocks from a stream of stocks that are doing better than the NASDAQ average over both the past week and the past year. To evaluate these predicates, we need to route tuples to operators that evaluate the week predicate and the year predicate. The routing decision requires choosing which operator to evaluate first. However, there seems to be a strong relationship between the stock type and the performance of the stock over time. For example, gold stocks did much better than tech stocks just after the dot-com crash. If the optimizer knows about this relationship, it can route tuples from tech stocks to the weekly operator first (where they are likely to fail the selection) and gold stock tuples to the yearly operator first. That is, a *content-based routing policy* can exploit the differences across tuples and generate plans tailored to different input classes to eliminate tuples sooner, decrease average tuple delay, and improve overall system performance.

This paper is organized as follows. In the rest of this section we describe work related to data streams and query optimization. Section 2 introduces and motivates content-based routing. Section 3 describes a framework for content-

based routing, Section 4 describes the Content-Learns algorithm for learning content routes automatically and efficiently, and Section 5 discusses the impact of content-based routing on adaptivity. Section 6 presents simulation results and Section 7 presents experimental results from a prototype implementation of content-based routing in TelegraphCQ. Conclusions and future work are presented in Section 8.

1.1. Related Work

Approaches for estimating the sizes of intermediate tables accurately include some form of re-optimization [6, 23, 24, 27], join synopses [1], intermediate statistics [9], multi-dimensional histograms [31], finding correlated attributes [21], adaptive sampling [25], and automatic statistics maintenance [13, 36]. Dynamic optimization [2] proposed starting competing plans in parallel but considered only single table accesses. The work of [19] was probably the first to employ a specific operator to choose plans. Later, query scrambling [40] (changing plans on the fly to cope with unexpected delays) was developed. Urhan and Franklin developed algorithms for dynamic pipeline scheduling [39]. These systems require new types of joins: partially preemptive hash joins [30], symmetric hash joins [41], Xjoins [38], ripple joins [20], etc. Related to this work is Juggle [33], a re-order operator that produces important results faster. Other fundamental operators are Eddies [3] and SteMs [32]. Examples of stream management systems include Aurora [10], NiagaraCQ [14], STREAM [29], TelegraphCQ [11], etc. Several authors have proposed improvements to data stream systems regarding operator scheduling [4], distributed data streams [37], and sharing of work among several continuous queries [26]. [12] proposes an interesting symmetry between data and queries where it is possible to pose new queries to old data and feed new data to old queries. [18] contains an excellent survey of data streams systems. [17] addresses several of the criticisms directed at Eddies, namely, the cost of keeping a state vector per tuple and the cost of invoking the data-flow optimizer every time a tuple enters the Eddy.

2. Content-Based Routing

In this section we introduce and motivate content-based routing. For ease of exposition, we will focus on data stream processing in the Eddies context throughout this paper. The reader should bear in mind that Eddies handle traditional relational queries as well, so our techniques apply to such queries. Also, we restrict ourselves initially to continuous filter queries over a single stream processed using a set of filter operators which either pass or drop an input tuple. Non-filter operators are considered in Section 4.4.

Correlated attributes and predicates pose a huge challenge to modern query optimizers [15, 36]. It is expensive to track correlation in large, update-intensive databases [21]. Furthermore, query optimization time increases significantly if we try to account for all possible correlations. Consequently, optimizers often ignore correlations by assuming independence, resulting in query plans that may be far from optimal when correlations are present [15]. This problem is particularly severe in data stream systems where stream characteristics may vary over time, sometimes very rapidly. An effective solution to this problem is *re-optimization*, that is, interleaving the optimization and the execution phases of query processing [23, 24, 27]. Re-optimization leverages the idea that we can compute accurate statistics on query subexpressions cheaply during query execution and detect when the optimizer's assumptions or estimates are incorrect. For example, if the size of a query subexpression computed during execution differs significantly from its size as estimated by the optimizer, then the rest of the query could be re-optimized taking this accurate new information into account. The Eddies approach [3], which we adopt in this paper, interleaves the optimization and execution stages continuously at the granularity of tuples. Thus, Eddies adapt fast to changes in stream and system conditions as well as converge to very efficient plans in the presence of correlated attributes and predicates.

An Eddy processes a query by routing input stream tuples through operators specific to that query. While there is no explicit notion of a query plan in an Eddy, the routes followed by tuples effectively simulate query plans. We will use an example continuous query from the network intrusion detection domain [7, 35] to illustrate the Eddies approach. We will use the same example later to motivate content-based routing. The goal in intrusion detection is to track packets in a network which signify a potential attack. An example continuous query in an enterprise network is:

> Track packets with destination address matching a prefix in table T, and containing the 100-byte and 256-byte sequences "0xab23...98b" and "0x76853...786ab" respectively as subsequences.

The lookup table T may contain addresses of subnetworks with access to the critical data in the enterprise. The byte sequences may represent patterns common to a specific type of network attack [7]. An Eddy for this query uses three operators—O_1, O_2, and O_3 corresponding to the filter 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. (O_1 could be a user-defined join as in [16].) Operators O_2 and O_3 perform the 100-byte and 256-byte sequence matches respectively. The Eddy will forward incoming S tuples to one of O_1–O_3. If the tuple is not dropped by that operator, the Eddy will forward it to one of the remaining operators, and so on. Tuples that pass all operators are streamed in the query result. Most tuples are routed through the most

selective operator path based on current statistics (*exploitation*). A small fraction of the tuples is routed through alternate paths to estimate the performance of these paths (*exploration*).

Let c_i denote the current expected processing cost per tuple for operator O_i, and let σ_i, $0 \leq \sigma_i \leq 1$, denote the current expected selectivity of O_i. (Selectivity refers to the fraction of input tuples passed by the operator.) Suppose the following conditions hold: $c_1 > c_3 > c_2$ and $\sigma_1 > \sigma_3 > \sigma_2$. Given these statistics, a traditional optimizer which ignores correlations will pick a plan for this query containing the operators in the order O_2, O_3, O_1. If the filter conditions are truly independent, the Eddy's routing will *converge* to the ordering O_2, O_3, O_1, i.e., most tuples will follow this route. However, the conditions corresponding to O_2 and O_3 could be correlated since they correspond to the same type of network attack, e.g., most tuples that pass (fail) O_2 may also pass (fail) O_3. The Eddy will observe this correlation when it explores the performance of different orderings, and converge to the optimal O_2, O_1, O_3 which outperforms O_2, O_3, O_1 in this case.

Suppose the attack monitored by our example query is underway on a noncritical subnetwork whose prefix is not in T. 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, we can distinguish between the attack and non-attack packets based on the source address. Therefore, we can route packets through O_1, O_2, O_3 or through O_2, O_1, O_3 based on the source address to ensure that both attack and non-attack packets are processed with high probability by the corresponding optimal ordering.

Currently, an Eddy makes routing decisions based exclusively on operator selectivity and cost estimates that represent each operator's performance over all tuples the operator has processed recently. Tuple content is ignored completely while estimating these statistics and making routing decisions. Specifically, the Eddy does not differentiate among tuples based on content while estimating operator selectivity or cost. Our example above illustrates a scenario where it is most efficient to route tuples through different operator orderings based on tuple content. This form of routing is called *content-based routing* (*CBR*). Without CBR, an Eddy ends up routing most tuples through the current best single ordering. CBR gives an Eddy the ability to detect and exploit multiple operator orderings simultaneously for a query when the input contains multiple tuple classes with different behavior with respect to operators, and therefore, different optimal operator orderings. However, the following questions need to be answered before we can judge whether CBR is viable or not:

Value of A	σ_1	σ_2	σ_3
$A = a$	0.32	0.10	0.55
$A = b$	0.31	0.20	0.65
$A = c$	0.27	0.90	0.60
Overall	0.30	0.40	0.60

Table 1. Content-specific selectivities

- **Q1:** Under what scenarios does CBR apply?
- **Q2:** How can we automatically and efficiently learn and use good content-based routes?
- **Q3:** How does CBR affect adaptivity?
- **Q4:** How does CBR perform in practice?

We address these questions in the next four sections. Section 3 addresses Q1, Section 4 addresses Q2, Section 5 addresses Q3, and Section 7 addresses Q4.

3. A Framework for CBR

3.1. Tuple Classes

Informally, CBR applies when the input tuples can be partitioned into two or more *tuple classes* where each class has a different optimal operator order for processing. To implement CBR efficiently, these tuple classes must be distinguishable easily from one another. In this paper we will consider tuples classes that can be distinguished from one another based on tuple content, namely, the attributes in the tuples. When a discrete-valued attribute A is used for CBR, we use hash-partitioning on A to partition the input tuples into different classes. Similarly, when a continuous-valued attribute A is used for CBR, we use range-partitioning on A. (Details of implementing hash and range partitioning are given in Section 4.1.) Attributes used to distinguish tuple classes are called *classifier attributes*.

3.2. Classifier Attributes

CBR applies when each input tuple class has an optimal (single) ordering for processing that is different from the optimal ordering of each other class. Informally, such a case arises when the selectivity of some operator is correlated with the content of some input attribute. The following example illustrates such correlation and its effect.

Example 3.1 Consider an input stream S processed by three operators O_1, O_2, and O_3. Let A be an attribute 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 tuples with $A = a$, $A = b$, and $A = c$, and the overall selectivity of each operator on S tuples. Assuming O_1–O_3 have equal costs, if we only take overall selectivities into account, 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 $A = c$. □

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. The degree of correlation between two distributions may be specified in a number of ways [28]. In this paper we use a specification from Information Theory which is based on the concept of *gain ratio* [28], described next.

Let R be a random sample of tuples processed by an operator O. 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* [28] of R, which is an information-theoretic metric used to capture the information content of R, is defined as:

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

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:

$$Entropy(R) = -\sigma \log_2 \sigma - (1 - \sigma) \log_2 (1 - \sigma) \quad (2)$$

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 [28]:

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

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 [28]:

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

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

Gain ratio is used widely in decision-tree learning algorithms (e.g., ID3 [28]) 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* $\text{GainRatio}(R, A) > \gamma$, *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 result 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 classifier attributes and their combinations. While some of our techniques extend directly to classifier attribute sets, we defer a detailed exploration of this issue to future work.

3.3. Run-time Overhead of CBR

Even when the input data has multiple tuple classes and optimal orderings that differ across these classes, CBR may not always improve overall performance because of the extra run-time overhead that CBR incurs. 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 content-based routes automatically. Therefore, CBR should be used opportunistically. Because of the extra routing overhead, CBR will not be useful for queries which contain a small number of cheap operators only, e.g., simple predicate evaluators. However, there are a variety of more expensive operators in stream systems where the benefits of CBR apply: joins with stored relations or windowed streams, user-defined functions that perform regular-expression-based pattern matching, multimedia or GIS functions, etc.

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 attributes C_1, C_2, ..., C_k that is processed by operators $O_1, O_2, \ldots, O_n$, and describe our *Content-Learns* algorithm to learn good content-based routes automatically in this setting. For now we will assume that all of attributes C_1, C_2, ..., C_k and operators $O_1, O_2, \ldots, O_n$ are candidates for CBR. In Section 4.3 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:

- **Optimization:** In this step, for each operator $O_l \in O_1, \ldots, O_n$, if one or more attributes in $C_1, \ldots, C_k$ are classifier attributes for O_l, then we keep track of the best classifier attribute for O_l. Informally, we identify the attribute in $C_1, \ldots, C_k$ 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. If the selectivity of O_l is not correlated with the contents of any attribute, then we do not perform CBR with respect to O_l. Details of the optimization step are described in Section 4.1.
- **Routing:** In this step we perform CBR using the current operator-attribute combinations identified by the optimization step. Our routing algorithm for CBR, which extends the original Eddies routing algorithm, is described in Section 4.2.

4.1. The Optimization Step of Content-Learns

The goal of optimization is to identify for each operator $O_l \in O_1, \ldots, O_n$, 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 3.2. 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. This technique requires the specification of two parameters: a probability 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 $\mathit{GainRatio}(R, C_j)$ 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 $v \in D_l$ 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 two types of run-time data structures:

1. **Selectivity Matrix:** Content-Learns maintains a $d \times k$ selectivity matrix, denoted M_l, for each profiled operator O_l. The rows of M_l correspond to the d partitions $v_1, \ldots, v_d$ of each attribute domain. The columns of M_l correspond in order to the k potential classifier attributes $C_1, \ldots, C_k$. Informally, $M_l[i, j]$ represents the selectivity of O_l for tuples which map to the ith partition of the domain of C_j. Each column of the matrix is implemented as a hash table of selectivities. Once a random sample R of tuples have been collected while profiling O_l, we can compute $M_l[i, j]$ as the fraction of tuples with $f_j(t.C_j) = v_i$ which are not dropped by operator O_l. In our implementation, we maintain $M_l[i, j]$ incrementally as new tuples are added to R, which has the advantage that we do not need to store the sample R.
2. **Weight Matrix:** Content-Learns maintains a $d \times k$ weight matrix, denoted W_l, for each profiled operator O_l. The rows and columns of W_l correspond to the rows and columns of M_l. For the random sample R of tuples collected when O_l is profiled (which is used to compute $M_l[i, j]$) $W_l[i, j]$ maintains the fraction of tuples in R with $f_j(t.C_j) = v_i$. Informally, $W_l[i, j]$ represents the fraction of tuples that map to the ith partition of the domain of C_j when O_l is profiled. $W_l[i, j]$ is maintained incrementally similar to $M_l[i, j]$.

Once we have collected the random sample R of tuples processed by operator O_l while profiling O_l, we can compute $\mathit{GainRatio}(R, C_j)$ (Equation (5)) for all $C_j \in C_1, \ldots, C_k$ using matrices M_l and W_l. From Equation (2), $\mathit{Entropy}(R)$ depends only on the overall selectivity of O_l over R, which is the weighted sum $\sum_{i=1}^{d} W_l[i, j] M_l[i, j]$ for any j. (Alternatively, the overall selectivity of O_l over R is simply the fraction of tuples in R passed by O_l.) Similarly, $\mathit{Entropy}(R_i)$ in Equation (3) for $\mathit{InfoGain}(R, C_j)$ depends only on $M_l[i, j]$. Finally, $\frac{|R_i|}{|R|}$ in Equations (3) and (4) for $\mathit{InfoGain}(R, C_j)$ and $\mathit{SplitInformation}(C_j)$ is equal to $W_l[i, j]$.

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 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. The routing algorithm implemented in the current TelegraphCQ release [11] does not take operator costs into account. This algorithm routes tuples to operators according to a probability which is inversely proportional to the operators' selectivities.[1] We call this algorithm *Selectivity*.

We designed Content-Learns to account for operator costs along the lines of [17] and [5]. Furthermore, we separated the statistics gathering part so that operator selectivity and cost statistics are now measured and updated for only a sample of the tuples processed instead of for every single tuple as done by the original implementation. In addition to reducing the run-time overhead of Eddies, this modification ensures that statistics are maintained in an unbiased manner, e.g., independent of the dominant operator routes. Again, the ideas are borrowed from [17] and [5]. Next, we describe our improved version of the Eddies algorithm and then describe our simple extension to incorporate CBR.

When an Eddy has to route a tuple t to one of operators $O_1, \ldots, O_n$, the Eddy routes t to the operator which has minimum value of $(1 - \sigma)/cost$, where σ is the expected overall selectivity of the operator and *cost* is the expected cost of the operator to process a tuple. When CBR is being used, some operators may be tagged with a classifier attribute. With CBR, when an Eddy has to route a tuple t to one of operators $O_1, \ldots, O_n$, the Eddy routes t to the operator which has minimum value of $(1 - \sigma)/cost$, 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 $M_l[i, j]$ where $f_j(t.C_j) = v_i$. (We have used the same notation as in Section 4.1.)
- If O_l is not tagged with a classifier attribute, then σ is the expected overall selectivity of O_l, which is the same value as that used always by the original Eddies algorithm.

Although *cost* is defined in a content-specific manner just like selectivity, we ignore this aspect for simplicity of presentation. 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 the original Eddy routing algorithm. Furthermore, CBR defaults to the overall selectivity estimate for attribute values for which the selectivity matrix does not contain a content-specific estimate.

4.3. 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. That is, the relative positions of operators differ across these orderings. However, if an operator is very cheap or very selective with respect to the other operators, or both, then it's position will mostly remain unchanged across these orderings. This intuition translates into an effective pruning heuristic where we do not consider very cheap (e.g., simple predicate evaluators) or very selective operators for CBR. Similarly, we can ignore operators that are very expensive or very unselective 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 for CBR to keep routing

1 Note that these selectivities represent the overall percentage of tuples that pass an operator. They do not represent the percentage of some class of tuples that passes an operator.

overhead low.[2]) While it is not hard to detect such attributes automatically, the required information often is available from the schema definitions.

4.4. 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, e.g., 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, which is used widely in decision-tree learning [28]. There are other techniques to quantify this skew, 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.

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. CBR increases both the learning overhead and the routing overhead of Eddies. Fundamentally, reducing run-time overhead is at odds with improving adaptivity [5, 11]. The approach we have adopted in this paper is to keep run-time overhead as low as possible while settling for slower adaptivity. So, we profile only one operator at a time, which may fail to adapt fast if the classifier attributes of an operator change in between two of its profiling phases. One possible extension to our approach is to continuously maintain all entries $M_l[i,j]$, $W_l[i,j]$, $1 \leq i \leq d$, for each O_l-C_j operator-attribute combination being used for routing. This extension will enable us to track the corresponding gain ratio values so that we can stop using O_l-C_j immediately if this correlation reduces significantly. There are various other techniques here with different tradeoffs between run-time overhead and adaptivity. We defer a detailed exploration of this spectrum to future work.

2 Note that this heuristic does not contradict the intrusion detection example in Section 2. In that example, the long attributes are analyzed only by the operators, while CBR checks only the four-byte source address attribute.

6. Simulation Results

A major focus of this work was to add two content-based routing policies to TelegraphCQ. However, we started by adapting the simulator used in [37] to include content-based routing. The simulator allowed us to compare content-based routing against more advanced policies not available in TelegraphCQ. It also allowed us to measure the average tuple delay and to include a routing-cost penalty to content-based policies. Note that the average tuple delay is a meaningless measure in the current implementation of TelegraphCQ because the system uses the stack model of operator scheduling. With this model, TelegraphCQ processes each input tuple completely before starting with another tuple. As such, if all operators have a selectivity of less than or equal to 100%, then the inter-operator queues will never contain more than one unprocessed tuple. Therefore, bad routing decisions will not affect the average delay time that tuples wait in queues because no tuples are waiting in any operator queue. Instead, bad routing decisions affect how long it takes to drop a tuple, the total number of routing calls, and the total execution time. Note that output tuples need to satisfy all operators and therefore take more or less the same time regardless of the routing policy used. Note also that it is not possible to implement policies Q, T, SCQ and WSCQ[3] [37] in TelegraphCQ because the operator queues will have at most one tuple, and because these policies all need queue length to represent the load level of operators.

6.1. Simulator Setup

The goal of the simulations was to quickly explore scenarios where CBR outperforms other policies. In this section, our routing policy does not learn correlations and does not use sampling. These advanced features where left for the next section. Instead, for the simulations, the Eddy with CBR knows which attribute is the classifier. Furthermore, the Eddy with CBR updates statistics for every tuple instead of using sampling. In fact, the simulation results lead us to implement these features in the real implementation of CBR in TelegraphCQ (Section 7) as a way to keep the CBR overhead small.

As for other routing policies, in [37], WSCQ was shown to have the best overall performance. Therefore, we compare only WSCQ and content-based routing.

3 With policy Q, the Eddy routes tuples to the operator with the shortest queue. With policy T, the Eddy routes tuples to operator with biggest ticket number. T reverts to Q when the operators are overloaded. With SCQ tuples are routed to the operator with greater benefit where the benefit takes into account selectivity, cost, and queue length. WSCQ is a variation of SCQ where tuples are routed to an operator with a probability proportional to the square of the benefit.

	Op1	Op2	Op3	Class total
Class 1	5%	100%	100%	5%
Class 2	100%	5%	100%	5%
Class 3	100%	100%	5%	5%
Overall	68.3%	68.3%	68.3%	

Table 2. Selectivities used for simulation's base experiment.

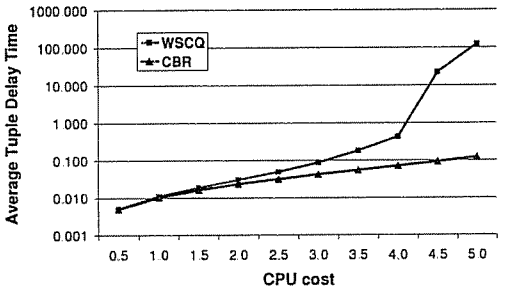

Figure 1. Average tuple delay for selectivities of base experiment.

The base case was a query with three operators over a stream with 100,000 tuples. The interval between tuples was set at 0.01 time units. Following the values used in [37], the available CPU per time unit was set at 300. The available CPU per time unit limits how many tuples can be processed by an operator. The CPU cost (the cost to process one tuple in one operator) was varied from 0.5 to 5 units. As an example, if the CPU cost to process an operator was 5 units, then the CPU could process 60 tuples per time unit.

[37]'s stream generator was changed such that there were three classes of tuples, with all classes equally likely to be generated and with the selectivities shown in Table 2. Although all operators had an overall selectivity of 68.3%, for one class of tuples the true selectivity was 5% while for the other two classes of tuples the true selectivity was 100%. All classes of tuples had a 5% chance of passing all three operators.

6.2. Average Tuple Delay

We used the simulator with the previous setup and varied the CPU cost from 0.5 to 5. The average tuple delay is shown in Figure 1.

As the CPU cost is increased, the number of tuples in the operator queues also increases. As such, the average tuple delay grows exponentially. It grows faster with WSCQ because this policy is unable to learn that the underlying stream is composed of three classes of tuples, each of which has different selectivities with respect to the operators. The content-based policy sends tuples to the right operator. That is, it sends tuples of class 1 to operator 1 first, it sends tuples of class 2 to operator 2 first, etc. Thus, the content-based policy is able to drop tuples sooner, maintain shorter queues, and keep the average tuple delay small. Note that WSCQ overloads the system when the CPU cost reaches 4.5 units. Using WSCQ, the average tuple delay jumps from 0.4 units to 22.1 units to 123.8 units as the cost to process one tuple increases from 4 to 4.5 to 5 units. Using the content-based policy, the average tuple delay grows from 0.07 units to 0.09 units to 0.12 units as the CPU cost increases from 4 to 4.5 to 5 units.

	Op1	Op2	Op3	Class total
Skew 5/100	5%	100%	100%	5%
Skew 15/58	15%	58%	58%	5%
Skew 25/45	25%	45%	45%	5%
Skew 35/38	35%	38%	38%	5%

Table 3. Selectivities for tuples of Class 1 for different skews. Selectivities for tuples of Classes 2 and 3 are set similarly.

The selectivities of Table 2 are highly favorable for a content-based routing policy because the penalty of making the incorrect decision is very big. That is, when a tuple of, say, Class 1 arrives, if the Eddy makes the right decision and routes it to operator 1, there is a 95% chance that the tuple will be dropped. However, if the Eddy routes it to operators 2 or 3, there is a 0% chance that the tuple will be dropped. In this case, the tuple will consume CPU time when it is processed, increasing the queue length, which in turn, increases the average tuple delay time. On the other hand, if the difference between operator selectivities is not so skewed, content-based routing provides less benefit. The following experiment explores other skews.

The selectivities in Table 3 were chosen so that the overall selectivity was kept constant for the different skews. The average tuple delays corresponding to these selectivities are shown in Figure 2.

The improvement in the average tuple delay of using the content-based routing policy is shown in Figure 3. Note, however, that with a content-based routing policy, a system is likely to incur higher routing costs. To model that cost, an extra routing cost equal to 50% of the processing cost was added[4]. That is, if the operator CPU cost is 3 units per tuple processed, the extra routing cost for the content-based

4 We verified this value through experimental results (shown in Section 5) that content-based routing has a routing overhead 50% higher than selectivity routing. Since selectivity routing is actually simpler to implement than WSCQ, the 50% extra routing overhead considered here is an upper limit.

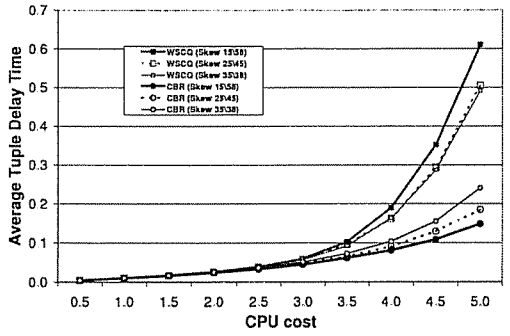

Figure 2. Average tuple delay for different skews.

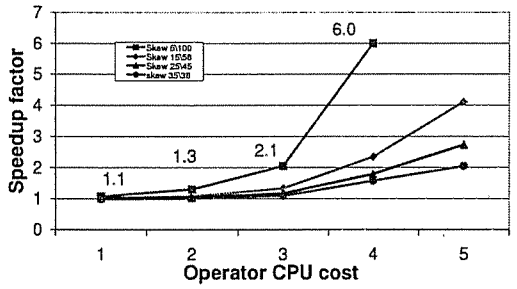

Figure 3. Speedup due to Content (no extra routing cost).

routing policy will be 1.5 units per tuple routed. These results are shown in Figure 4.

As can be seen, even with extra routing overhead, content-based routing provides very substantial improvements over WSCQ. The more skewed the selectivities are, and the more costly it is to process a tuple, the larger the improvement provided by content-based routing.

In one extreme, when the skew is very high, content-based is between a factor of 2 to a factor of 5.5 faster than WSCQ. On the other extreme, when there is no skew, content-based routing is between 8% and 17% worse than WSCQ because of the extra routing overhead and because WSCQ takes into account the queue length in the operators. We note that this was the result that lead us to develop a CBR algorithm with very low routing overhead (described in Section 4 and evaluated in Section 7). In the experimental results of Section 7 we show that the overhead of content-based routing in practice is very low and content-based routing is never worse than the TelegraphCQ policy.

7. Experimental Results

We now describe an experimental evaluation of our CBR techniques. We have studied CBR using both a simulator

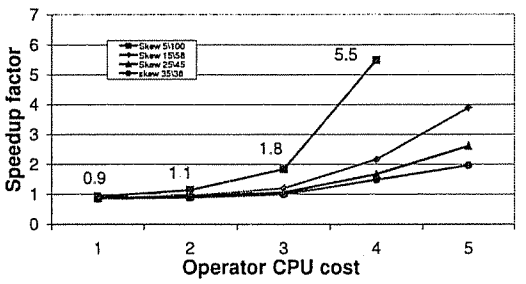

Figure 4. Speedup due to Content (with extra routing cost).

and a prototype implementation in TelegraphCQ [11].

In our prototype, some features of our CBR algorithm from Section 4 have not been completed yet. The relevant features among these are: (i) operator costs are not being measured, so costs are assumed to be the same for all operators; (ii) we do not compare performance with and without using a chosen operator-attribute combination before using it for routing; (iii) pruning heuristics are not used. The implementation of these features is straightforward as we have described, and is scheduled as immediate future work.

7.1. Experimental Setup

Section 4.2 described most details of our implementation of CBR in TelegraphCQ, and the non-content-based *Selectivity* algorithm in TelegraphCQ. To bring out the differences between the learning overhead and the routing overhead of CBR, our experiments include a routing algorithm called *Content-Knows* which does not need to learn classifier attributes automatically. Instead, Content-Knows is "told" which attribute is the best classifier for each operator. Effectively, Content-Knows is Content-Learns without the learning overhead.

To speedup the computation of entropies, we use a table with 101 precomputed entries. Each entry corresponds to a selectivity value $\sigma \in [0.00, 0.01, \ldots, 1.00]$ and stores the corresponding entropy computed from Equation 2.

For our experiments, we created a synthetic benchmark based on a star schema. Instead of the central fact table, we used a data stream S. 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          // Op1
         AND s.fkd2 = d2.pk      // Op2
         ...
         AND s.fkdN = dN.pk      // OpN
```

A star schema was chosen for two reasons. First, it seems that the majority of queries over streams refer to one single stream source that joins with zero or more normal tables,

	Operator 1	Operator 2	Operator 3
Class 1	100%	50%	0%
Class 2	50%	50%	50%
Class 3	10%	20%	30%
Overall	53.3%	40%	26.7%

Table 4. Content-specific selectivities

which is similar to what queries over star-schemas do. Second, it seems that the majority of data streams applications have streams that represent facts (traffic information; online purchases; search engine requests; stock trading; etc.) which then join with dimensions (speed sensors and carID; productID, costumerID, and date; keyword, client IP, and browser version; and stockID, stockType, and date; etc.). For each query, a stream of 100,000 tuples was generated. Depending on the query, between two to eight dimension tables containing 10,000 tuples each are used.

As each new stream tuple arrives, the Eddy decides to which of N SteMs, or join operators, to route it to.[5] Our stream generator is able to generate tuples with any kind of non-independence between the classifier attribute *attrC* and the selectivity of the join operators. For instance, it can generate a stream with the characteristics in Table 4. In this stream, tuples of Class 1 have 0% chance of finding a matching tuple when probing operator 3's SteM and have a 100% chance of finding a matching tuple when probing operator 1's SteM. Therefore, a clever algorithm for this stream will send Class 1 tuples to Operator 3 first, Class 3 tuples to Operator 1 first, then to Operator 2, and finally to Operator 3, and Class 2 tuples to any operator. In addition to the running time, we use the number of routing calls as the performance metric in some experiments. The justification for using routing calls is that it shows a clearer picture of the quality of the routing algorithm regarding its ability to find the most selective operators. 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. On the other hand, the running time represents overall performance, including the overhead of learning and routing.

7.2. Benefits of CBR

In this section we present results from experiments where we studied the performance of CBR with respect to the amount of correlation between operator selectivity and attribute content. We added additional attributes to the input stream apart from the foreign keys corresponding to the dimension tables. One such attribute, denoted *attrC*, is the one whose content is varied in the experiments in this section to vary the correlation between input content and the selectivity of the N join operators. The content of the other additional attributes is chosen randomly, so they are uncorrelated with operator selectivities. These attributes enable us to test the effectiveness of Content-Learns at identifying the right classifier attribute. Based on this setup, *attrC* is the only potential classifier attribute in the input stream. While this information is provided to the Content-Knows algorithm, the Content-Learns algorithm must learn the classifier attribute to be able to use CBR.

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

Table 5. Selectivities for class/operator pairs

	N=2	N=4	N=6	N=8
A	B	B	B	B
5%	**100%**	**100%**	**100%**	**100%**
15%	**33%**	**69%**	**80%**	**85%**
25%	20%	**58%**	**72%**	**79%**
35%	14%	**52%**	**68%**	**76%**
45%	11%	**48%**	**64%**	**73%**
55%	9%	45%	**62%**	**71%**
65%	8%	43%	60%	**69%**
75%	7%	41%	58%	68%
85%	6%	39%	57%	67%
95%	5%	37%	55%	66%

Table 6. Selectivities A and B for different number of operators N in the experiments

In the following experiments, streams with as many tuple classes as joins were used. For example, for a 5-way join query, the domain of *attrC* had five values. The motivation is to evaluate situations where the complexity of data and the complexity of queries grow in a similar way. For each stream, tuple classes at random were generated with equal probability. The selectivities of the operators for tuples in one class were different from the selectivities of operators for tuples in another class. Selectivities were assigned as shown in Table 5. That is, for each class, there is one operator whose selectivity (A) is different from the selectivities of the other operators (B). The aggregate selectivity of Op1 $\wedge$ Op2 $\wedge \ldots$ OpN was kept constant at 5% by choosing the values for A and B as shown in Table 6. (Aggregate selectivity is varied in Section 7.5.)

5 Each hash join is composed of two SteMs. Tuples from one relation in a join build into one of the SteMs and probe into the other. Although the query plan contains a total of 2N SteMs (2 per join), half of them will be build SteMs for incoming stream tuples.

In the upper-right triangular region in Table 6 where $A < B$ (marked in **bold**), a clever routing algorithm can exploit the *selectivity skew* by routing tuples first to the lower selectivity operator corresponding to A. In the lower-left region where $A > B$, a clever routing algorithm will avoid the A operator and route tuples through B operators first. The N-way join query was run for two, four, six, and eight join operators, using selectivities as given by Tables 5 and 6. The results for two and six joins are shown in Figure 5 and Figure 6 respectively. The results for the other queries are similar in the sense that the curve for Content-Learns is below[6] the one from Selectivity, and having a similar number of routing calls only when the values of A and B are close (and therefore, when routing decisions become less relevant). Also, the difference in number of routing calls between Content-Learns and Selectivity is largest in the left part of the graph, i.e., when the selectivity skew is high. As skew decreases, both algorithms require essentially the same number of routing calls. Finally, as the skew increases again, Content-Learns incurs fewer routing calls than Selectivity, although by a much smaller margin. Furthermore, in all cases, Content-Learns closely follows the curve of Content-Knows. This shows that Content-Learns is learning the classifier attribute correctly and quickly.

Overall, the higher the skew between the selectivity values A and B, especially when $A < B$, the greater the extent by which Content-Learns outperforms Selectivity. At most, Content-Learns outperforms Selectivity by performing 65.0% fewer routing calls (with eight operators and the largest skew). Across all experiments, when $A < B$, Content-Learns required on average 20.6% fewer routing calls, and when $A > B$, Content-Learns required 9.3% 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 by being less selective. Overall, Content-Learns required 14.7% fewer calls.

Why do the initial skew ($A < B$) and the final skew ($A > B$) yield different results? This difference is accounted for by the likelihood of finding the most selective operator and the difference between the most and the least selective operators. For instance, for the 6-way join query, initially there is a 1/6 chance of finding the A operator which has a selectivity of 5%, while at the end of the curve, there is a 5/6 chance of finding any of the five B operators that have a selectivity of 55%.

7.3. Quality of Learning

To study the quality of Content-Learns, we ran the N-way join query with four, six, and eight joins. Each query was run over 50 streams of 100,000 tuples each and with

6 There were a few statistically insignificant points were Selectivity had slightly less (at most 0.3% less) routing calls than Content-Learns in the region where the two curves meet.

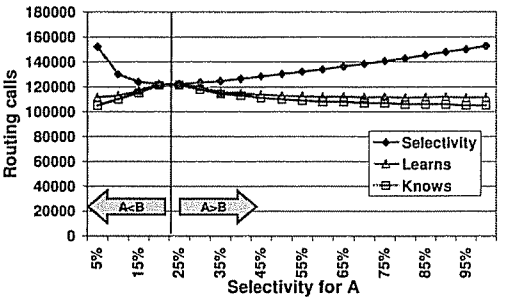

Figure 5. Routing calls in 2-way join query

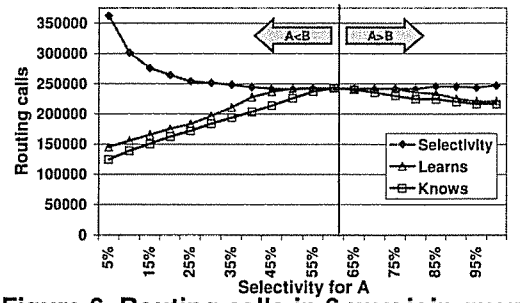

Figure 6. Routing calls in 6-way join query

random amounts of correlation between values of attribute *attrC* and the selectivities of each join operator. We included additional attributes (a constant, a sequence number, and random numbers as the foreign keys) whose content was not correlated with any of the selectivities of the operators. Therefore, Content-Learns has to learn that the right classifier attribute is *attrC* among all attributes in the input stream. There were 6, 8, and 10 extra attributes not correlated with any operator for the queries with 4, 6, and 8 joins respectively. The results, reported in Figures 7 and Figure 8, show 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 1.2% of the time, yielding on average 20%, 28%, and 33% fewer routing calls than Selectivity for the queries with 4, 6, and 8 joins respectively.

7.4. Run-time Overhead of Learning

At first, we expected Content-Learns to have higher run-time overhead—extra learning and routing—than both Content-Knows and Selectivity. This overhead has three components: the storage overhead for content-based selectivities and weights, the cost of gathering and updating these statistics in addition to operator statistics, and the cost of computing gain ratio to identify classifier attributes. However, as seen in Figure 9, in practice, maintaining these extra statistics and computing the gain ratio does not incur a significant amount of run-time overhead. The reason is that the extra statistics are only updated during operator profil-

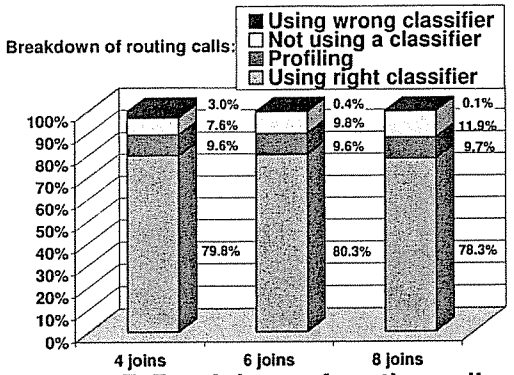

Figure 7. Breakdown of routing calls

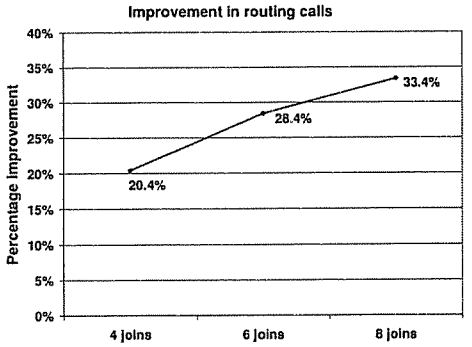

Figure 8. Improvement in routing calls by Content-Learns over Selectivity

ing (on average, one out of 8 tuples is sent to the profiled operator first) and the gain ratio is computed only after 500 samples have been collected (or around 4000 tuples).

For all policies and for all queries, the overhead corresponding to routing and updating statistics was around 1.75%. For the Selectivity policy, this represented 2.2 microseconds per routing decision and for Content-Learns it represented 2.4 microseconds per routing decision.

With respect to storage overhead, Selectivity uses one double value per operator to store the operator's selectivity. Content-Knows stores one hash table of selectivities per operator-classifier combination used for CBR. Since we have at most one classifier attribute (*attrC*) in our experiments, the number of selectivity hash tables needed by Content-Knows is at most the number of operators (= N). All attributes in our experiments are discrete-valued, so a selectivity matrix in Content-Knows stores pair-wise selectivities of the form (*attrC*=X, $\sigma(O_l)$) in a table hashed by the value X of *attrC*. Content-Learns requires everything that Selectivity requires, plus a hash table of selectivities for each potential classifier attribute while profiling an operator. Each hash table contains 32 buckets and each bucket stores one selectivity value of type double. Furthermore, Content-Learns requires also one hash table per operator O_l for which an O_l-attribute combination is being used for CBR. Thus, for Content-Learns, the total storage overhead grows at most as O((#Potential_classifiers + #Ops) * Hash_table_size). An N-way join in our experiments has N operators and N+3 potential classifier attributes (including *attrC*). For $N = 8$ and for hash tables containing 32 4-byte doubles, this overhead comes to 2432 bytes. The corresponding weight matrices require other 2432 bytes totalling an overhead of 4864 bytes.

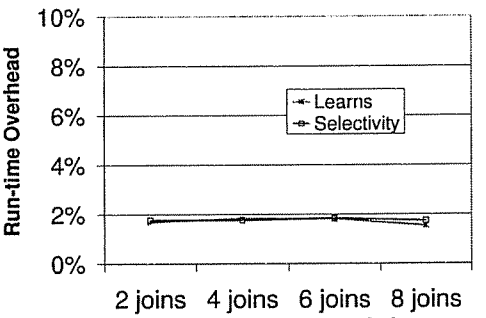

Figure 9. Percentage of time doing routing decisions and updating statistics

7.5. Robustness of Performance

In Section 7.2, the choice of selectivities made routing tuples to operators difficult for the Selectivity routing algorithm 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, the operators appeared to have a selectivity of $(A + B(N-1))/N$, for the N-way join. In the following experiments, selectivities are random multiples of 10 between 0 and 100. The N-way join query was run with 2, 4, 6, and 8 operators using 50 different input streams. Figure 10 shows the improvement in running time of Content-Learns versus Selectivity.

It is clear from Figure 10 that a content-based routing algorithm can improve performance significantly compared to an algorithm that ignores tuple content. Note that the larger the number of operators involved, the more opportunities are available for improvement.

The selectivities used in the experiments so far were all low. In Section 7.2, the overall aggregate selectivity was kept at 5%. In Section 7.3 and in the previous experiment, the operator selectivities were random without any guarantee on the aggregate selectivity. On average, that aggregate selectivity was very close to the expected aggregate selectivity of 0.5^N for each stream (with $N = 2, 4, 6$, and 8 operators) or about 8% on average across all streams. This section explores the space of aggregate selectivities from 5% to 35%. For this experiment, the 6-way join query was run with "random" selectivities. These selectivities were chosen with the restriction that the overall aggregate selectivity was kept at some pre-determined value of 5%, 15%, 25%, or

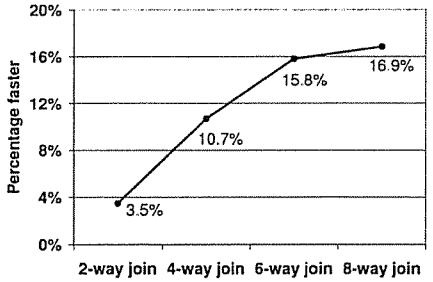

Figure 10. Random selectivities in all operators: percentage of total time that Content-Learns is faster than Selectivity

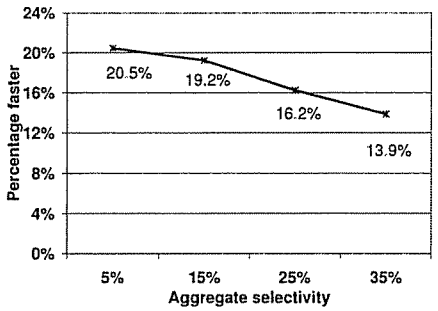

Figure 11. Varying aggregate selectivity: percentage of total time that Content-Learns is faster than Selectivity (for 6-way join queries)

35%. In each case, 50 streams of 100,000 tuples were processed. The results appear in Figure 11.

8. Conclusions and Future Work

In this paper we showed how the adaptive architecture of a Data Stream Management System can be extended with content-based learning and routing to enable the system to exploit non-independent distributions and correlations instead of being hurt by them. Our most important contribution was to show that this content-based learning and routing can be done very cheaply and still achieve relevant performance improvements. We presented the Content-Learns algorithm which learns good content-based routes automatically and showed that the overhead of maintaining the extra statistics and computing gain ratio is negligible when compared to the Selectivity algorithm.

Our prototype implementation indicates that CBR can improve execution time by up to 20% when compared with routing based on operator statistics alone. Whenever skew was present, Content-Learns yielded better results than Selectivity in all experiments, both in the number of routing calls as well as in absolute running time. When no skew was present, Content-Learns was at most 0.3% worse than Selectivity. In addition, the performance comparison between Content-Learns and Content-Knows showed that Content-Learns learns classifier attributes correctly in real time.

While CBR seems to be a promising approach in stream 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 3.2. 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 learned automatically yet. These include the hash and range partitioning functions, the schedule for profiling operators, and the sampling rate and sample size for computing gain ratio.
- Does CBR apply to adaptive plan-based stream systems like STREAM [5] and to adaptive database systems like Leo [36]? Based on initial studies, we expect most of our techniques to apply directly to plan-based stream systems, but further investigation is needed.

References

[1] S. Acharya, P. B. Gibbons, V. Poosala, and S. Ramaswamy. Join synopses for approximate query answering. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 275–286, June 1999.

[2] G. Antoshenkov. Dynamic query optimization in Rdb/VMS. In *Proc. of the 1993 Intl. Conf. on Data Engineering*, pages 538–547, Apr. 1993.

[3] R. Avnur and J. Hellerstein. Eddies: Continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 261–272, May 2000.

[4] B. Babcock, S. Babu, M. Datar, and R. Motwani. Chain: Operator scheduling for memory minimization in data stream systems. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003.

[5] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom. Adaptive ordering of pipelined stream filters. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, June 2004.

[6] S. Babu and J. Widom. StreaMon: An adaptive engine for stream query processing. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, June 2003. Demonstration proposal.

[7] J. Beale et al. *Snort 2.1 Intrusion Detection*. Syngress Publishing, 2004.

[8] P. Bizarro, S. Babu, D. DeWitt, and J. Widom. Content-based routing for continuous query-optimization. Technical report, Wisconsin Database Group, June 2004. Available at http://www.cs.wisc.edu/~ pedro/icde05_cbr_full.pdf.

[9] N. Bruno and S. Chaudhuri. Exploiting statistics on query expressions for optimization. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 263–274, June 2002.

[10] D. Carney et al. Monitoring streams–a new class of data management applications. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, Aug. 2002.

[11] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. First Biennial Conf. on Innovative Data Systems Research*, Jan. 2003.

[12] S. Chandrasekaran and M. Franklin. Streaming queries over streaming data. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, pages 203–214, Sept. 2002.

[13] S. Chaudhuri and V. Narasayya. Automating statistics management for query optimizers. *IEEE Trans. on Knowledge and Data Engineering*, 13(1):7–20, Jan. 2001.

[14] J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, pages 379–390, May 2000.

[15] S. Christodoulakis. Implications of certain assumptions in database performance evaluation. *ACM Trans. on Database Systems*, 9(2):163–186, 1984.

[16] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk. Gigascope: A stream database for network applications. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, pages 647–651, June 2003.

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

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

[19] G. Graefe and K. Ward. Dynamic query evaluation plans. In *Proc. of the 1989 ACM SIGMOD Intl. Conf. on Management of Data*, pages 358–366, May 1989.

[20] P. J. Haas and J. Hellerstein. Ripple joins for online aggregation. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 287–298, June 1999.

[21] I. Ilyas, V. Markl, P. Haas, P. Brown, and A. Aboulnaga. CORDS: Automatic discovery of correlations and soft functional dependencies. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 647–658, June 2004.

[22] Y. Ioannidis and S. Christodoulakis. On the propagation of errors in the size of join results. In *Proc. of the 1991 ACM SIGMOD Intl. Conf. on Management of Data*, pages 268–277, May 1991.

[23] Z. Ives, D. Florescu, M. Friedman, A. Levy, and D. Weld. An adaptive query execution system for data integration. In *Proc. of the 1999 ACM SIGMOD Intl. Conf. on Management of Data*, pages 299–310, June 1999.

[24] N. Kabra and D. DeWitt. Efficient mid-query re-optimization of sub-optimal query execution plans. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 106–117, June 1998.

[25] Y. Ling and W. Sun. An evaluation of sampling-based size estimation methods for selections in database systems. In *Proc. of the 1995 Intl. Conf. on Data Engineering*, pages 532–539, Mar. 1995.

[26] S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, pages 49–60, June 2002.

[27] V. Markl, V. Raman, D. Simmen, G. Lohman, and H. Pirahesh. Robust query processing through progressive optimization. In *Proc. of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, pages 659–670, June 2004.

[28] T. Mitchell. *Machine Learning*. McGraw-Hill, New York, 1997.

[29] R. Motwani, J. Widom, et al. Query processing, approximation, and resource management in a data stream management system. In *Proc. First Biennial Conf. on Innovative Data Systems Research (CIDR)*, Jan. 2003.

[30] H. Pang, M. Carey, and M. Livny. Partially preemptive hash joins. In *Proc. of the 1993 ACM SIGMOD Intl. Conf. on Management of Data*, pages 59–68, May 1993.

[31] V. Poosala and Y. Ioannidis. Selectivity estimation without the attribute value independence assumption. In *Proc. of the 1997 Intl. Conf. on Very Large Data Bases*, pages 486–495, Sept. 1997.

[32] V. Raman, A. Deshpande, and J. Hellerstein. Using state modules for adaptive query processing. In *Proc. of the 2003 Intl. Conf. on Data Engineering*, Mar. 2003.

[33] V. Raman, B. Raman, and J. Hellerstein. Online dynamic reordering for interactive data processing. In *Proc. of the 1999 Intl. Conf. on Very Large Data Bases*, 1999.

[34] P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie, and T. G. Price. Access path selection in a relational database management system. In *Proc. of the 1979 ACM SIGMOD Intl. Conf. on Management of Data*, pages 23–34, June 1979.

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

[36] M. Stillger, G. Lohman, V. Markl, and M. Kandil. LEO - DB2's LEarning Optimizer. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, pages 9–28, Sept. 2001.

[37] F. Tian and D. DeWitt. Tuple routing strategies for distributed eddies. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, Sept. 2003.

[38] T. Urhan and M. J. Franklin. Xjoin: A reactively-scheduled pipelined join operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.

[39] T. Urhan and M. J. Franklin. Dynamic pipeline scheduling for improving interactive performance of online queries. In *Proc. of the 2001 Intl. Conf. on Very Large Data Bases*, Sept. 2001.

[40] T. Urhan, M. J. Franklin, and L. Amsaleg. Cost based query scrambling for initial delays. In *Proc. of the 1998 ACM SIGMOD Intl. Conf. on Management of Data*, pages 130–141, June 1998.

[41] A. N. Wilschut and P. M. G. Apers. Dataflow query execution in a parallel main-memory environment. In *Proc. of the 1991 Intl. Conf. on Parallel and Distributed Information Systems*, pages 68–77, Dec. 1991.