Tuple Routing Strategies for Distributed Eddies

Feng Tian David J. DeWitt

Department of Computer Sciences
University of Wisconsin, Madison
Madison, WI, 53706
{ftian, dewitt}@cs.wisc.edu

Abstract

Many applications that consist of streams of data are inherently distributed. Since input stream rates and other system parameters such as the amount of available computing resources can fluctuate significantly, a stream query plan must be able to adapt to these changes. Routing tuples between operators of a distributed stream query plan is used in several data stream management systems as an adaptive query optimization technique. The routing policy used can have a significant impact on system performance. In this paper, we use a queuing network to model a distributed stream query plan and define performance metrics for response time and system throughput. We also propose and evaluate several practical routing policies for a distributed stream management system. The performance results of these policies are compared using a discrete event simulator. Finally, we study the impact of the routing policy on system throughput and resource allocation when computing resources can be shared between operators.

1. Introduction

Stream database systems are a new type of database system designed to facilitate the execution of queries against continuous streams of data. Example applications for such systems include sensor networks, network monitoring applications, and online information tracking. Since many stream-based applications are inherently distributed, a centralized solution is not viable. Recently the design and implementation of scalable, distributed data stream management systems has begun to receive the attention of the database community.

Many of the fundamental assumptions that are the basis of standard database systems no longer hold for data stream management systems [8]. A typical stream query is long running -- it listens on several continuous streams and produces a continuous stream as its result. The notion of running time, which is used as an optimization goal by a classic database optimizer, cannot be directly applied to a stream management system. A data stream management system must use other performance metrics. In addition, since the input stream rates and the available computing resources will usually fluctuate over time, an execution plan that works well at query installation time might be very inefficient just a short time later. Furthermore, the "optimize-then-execute" paradigm of traditional database systems is no longer appropriate and a stream execution plan must be able to adapt to changes of input streams and system resources.

An eddy [2] is a stream query execution mechanism that can continuously reorder operators in a query plan. Each input tuple to an eddy carries its own execution history. This execution history is implemented using two bitmaps. A done bitmap records which operators the tuple has already visited and a ready bitmap records which operators the tuple can visit next. An eddy routes each tuple to the next operator based on the tuple's execution history and statistics maintained by eddy. If the tuple satisfies the predicate of an operator, the operator makes appropriate updates to the two bitmaps and returns the tuple to the eddy. The eddy continues this iteration until the tuple has visited all operators. Figure 1.1 shows an eddy with three operators. The major advantage of an eddy is that the execution plan is highly adaptive with the routing decision for each individual tuple deciding the execution order of the operators for this tuple. [2][18] demonstrate that this technique adapts well to changes in input stream rates.

However, a centralized eddy cannot be directly employed in a distributed data stream management system without incurring unnecessary network traffic and delays and would almost certainly end up being a bottleneck.

**Proceedings of the 29th VLDB Conference,
Berlin, Germany, 2003**

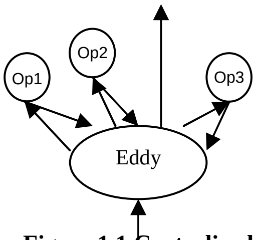

Figure 1.1 Centralized Eddy

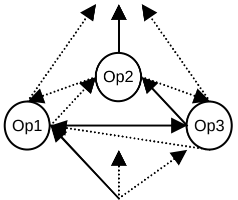

Figure 1.2 A distributed query plan

In this paper we study the design, implementation, and performance of the following distributed eddy algorithm. After an operator processes a tuple, instead of returning the tuple to a centralized eddy, the operator makes a routing decision based on the execution history of the tuple and statistics maintained at the operator. Figure 1.2 shows a distributed plan with three operators. The dashed arrows indicate possible routes between operators. The four solid arrows indicate one possible execution order that a tuple might actually take. The routing policy at each operator decides the execution order of the operators for each tuple, therefore, dynamically optimizing the distributed stream query plan. The purpose of this paper is to study the effectiveness of different routing policies.

As discussed earlier, query response time is not an appropriate metric to evaluate a data stream management system. Instead we propose the following two metrics:

ART - the average response time measured as the average time between when a tuple enters and leaves the operators that form the distributed eddy.

MDR - the maximum data rate the system can handle before an operator becomes a bottleneck.

The formal description of the system and rigorous definitions of these metrics will be given in Section 3.

Section 4 examines the impact of the routing policy on system performance. The distributed query plan is modelled using a queuing network and a solution technique is described. We also study several practical routing policies that have straightforward implementations and compare their performance.

A distributed stream processing system must be able to dynamically adapt to configuration changes such as adding or removing computing resources. Changes in input data rates may also require the system to re-allocate resources via load sharing techniques. Aurora* [6] implements box sliding and box splitting to enable load sharing across nodes. The natural way of applying these load sharing techniques is to split the workload of an overloaded node and to merge workloads of lightly loaded nodes. The routing policy is an important factor in determining which node is likely to be overloaded. In Section 5, the effect of routing policy on the system throughput and resource allocation when computing resources can be added to or removed from a node is examined. Conclusions and future research directions are contained in Section 6.

2. Related Work

There are a number of research projects currently studying issues related to streaming data [1][2][3][4][5] [6][7][8][12][16][18][22][26]. Those that are most closely related to our work are the Aurora* [6][8], STREAM [3][4][22], Telegraph [2][9][18] and Cougar [7][12] projects.

The original eddy paper [2] introduced the concept of routing tuples between operators as a form of query optimization. This paper extends the idea of an eddy to a distributed environment. The routing policies described in [2] and [18] are compared against several other routing policies in Section 4 and 5.

Aurora [8] describes the architecture of a data stream management system. Aurora* [6] extends Aurora to a distributed environment and discusses load sharing techniques. Aurora also uses routing as a mechanism to reorder operators. The routing mechanism is similar to that of an eddy and our results can be adapted to Aurora*.

STREAM [3] describes a query language and precise semantics of stream queries. [5][22] describe both operator scheduling and resource management in a centralized data stream management system, focusing on minimizing inter-operator queue length or memory consumption. In [22] a near-optimal scheduling algorithm for reducing inter-operator queue size is presented. In addition, [22] explores using constraints to optimize stream query plans.

Cougar [7][12] is a distributed sensor database system. Cougar focuses on forming clusters out of sensors to allow intelligent in-network aggregation to conserve energy by reducing the amount of communication between sensor nodes.

[27] asserts that execution time is not an appropriate goal for optimizing stream queries and proposes the use of output rates as more appropriate. The output rate metric proposed in [27] is essentially equivalent to our **MDR**.

Several approaches have been proposed on how to gather statistics over a stream [4] [11] [13] [16] [19] [20] [21] with the primary focus being how to obtain good estimates over streaming data with limited amounts of memory and minimal CPU usage. These results will be critical to the design of accurate routing policies to any distributed eddy implementation.

There are many papers that describe the use of queuing networks to analyze computer system. [14][15] are the standard texts on this subject.

3. Overview of the System Model and Performance Metrics

We model a distributed stream query plan as a set of operators ***Op_i, i=1,..,n*** connected by a network. Input tuples to an operator are added to a first-come, first-served (FCFS) queue, as shown in Figure 3.1. ***Op_i.R*** resources (i.e. CPU, memory and network bandwidth) are assumed to be available to each operator **$Op_{i,}$**. We further assume that each input tuple to **Op_i** consumes, on average, **$Op_i.r$**, resources. Thus, **Op_i** can process at most **$Op_i.R/Op_i.r$** input tuples per time unit and the average service time **Ts** for each individual tuple is **$Op_i.r/Op_i.R$**. Later we will see that only **Ts** will appear in solutions of the queuing model and in fact, many queuing network literature normalize **$Op_i.R$** to 1. We do not use such normalization technique because in Section 5 we examine the case that computing resources at each operator can be dynamically reallocated among the operators. The **residence time *Tr*** of a tuple at **Op_i** is the sum of the queuing delay and the processing time of the tuple.

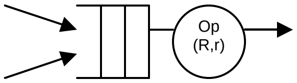

Figure 3.1 Operator

A data source operator is a special kind of operator that generates tuples. As a tuple **t** is generated it is time-stamped with the system time *ts*. As in [2], each tuple also carries its own execution history *h*. The execution history records the operators that the tuple has already visited. The operators that the tuple needs to visit next are implicitly determined by the operators that it has already visited. A tuple with timestamp *ts* and history *h* is denoted as **t^{ts}_h**. A data source operator does not receive input tuples from other operators and we assume that the residence time of a tuple (generated) at a data source operator is zero.

In average, for every input tuple **t^{ts}_h** to operator **Op_i**, **Op_i** will produce **ρ** output tuples. **ρ** is called the selectivity of **Op_i**. In the case of a selection operator, **ρ** is a constant less than 1. In the case of a join, **ρ** may be greater than 1[1]. For each output tuple, **Op_i** selects the next operator to send the tuple to with probability proportional to a routing weight function **$Op_i.W(h)=(w_{i1}(h),\ w_{i2}(h),\ \ldots,\ w_{in}(h))$**, where **$n$** is the number of operators in the distributed plan. Each output tuple is time-stamped with the same *ts* as the input tuple and its execution history is set appropriately before it is sent to the next operator. This process continues until the tuple reaches a special operator termed the Data Sink (**Sink**). The **Sink** operator models the result of the distributed plan. Tuples sent to a **Sink** operator correspond to tuples that have left the system; therefore we assume the tuple consumes no resources at a **Sink** operator and that the residence time of the tuple at **Sink** is zero. When a **Sink** receives an input tuple, the operator can compute the **response time of the tuple** as the system time minus the timestamp of the tuple. The operators that the tuple has visited, in order, is called **execution path of the tuple**.

Operators in this model have only one input queue. We briefly explain how to implement the join operator, which logically has two input streams. Our treatment of join is very much like a distributed version of SteMs described in [18]. We assume that all the join operators use sliding window semantics so that a tuple will not have to be joined with an infinite amount of data. Also, we assume all join operators are implemented using a symmetric hash join algorithm [28].

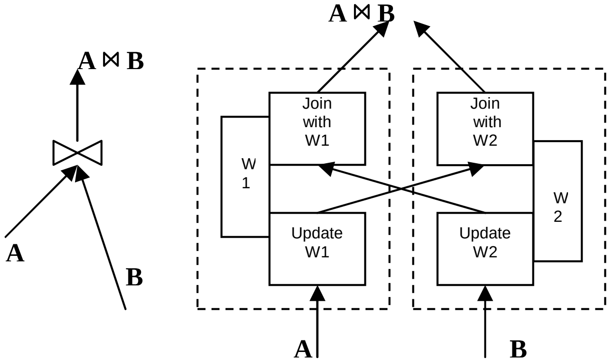

Figure 3.2 The distributed symmetric algorithm for A⋈B

Figure 3.2 shows a distributed symmetric join algorithm for A⋈ B. An input tuple from stream A is first processed by an operator that maintains a sliding window for stream A. The tuple is then sent to an operator that joins the tuple with the window of stream B (on perhaps a different physical node). Tuples from stream B are processed in a symmetric way. Dashed boxes in Figure 3.2 indicate components of the join algorithm that are likely to reside on a single node.

We propose the following two metrics to measure the performance of a distributed query plan.

- The **Average Response Time (ART)** of tuples that reach the **Sink** operator. Notice tuples that are filtered out during query processing do not contribute to this metric.
- The **Maximum Data Rate (MDR)** at the data source operators before the system is overloaded. We say that the system is overloaded if the length of any operator's input queue exceeds a pre-defined maximum length. When there are multiple data source operators, it is reasonable to optimize for some function of the data rates of each data source operator. For simplicity, we will optimize for the sum of all data rates of the data source operators.

The corresponding optimization problems for these two metrics then can be stated as

Optimization Problem 1 (ART): Given the network configuration (including the selectivity of each operator) and the data generation rates at the data source operators, choose a routing weight function **W** for each operator **Op_i**,

[1] For most operators **ρ** is a constant. However, there are situations that **ρ** depends on the execution history of input tuples. We will see such an example (a three way join) in Section 4.

such that the **Average Response Time** at **Sink** is minimized.

Optimization Problem 2 (MDR): Given the network configuration, choose a routing weight function $\boldsymbol{W}$ for each $\boldsymbol{Op_i}$, such that the **Maximum Data Rate** is maximized.

4. ROUTING POLICY OF A PLAN WITH FIXED RESOURCE ALLOCATION

4.1 Numerical solutions for ART and MDR

In this section, we consider the routing policy for a plan in which the computing resource $\boldsymbol{R}$ of operator $\boldsymbol{Op_i}$ is fixed. We further assume that the routing weight function $\boldsymbol{W}$ of $\boldsymbol{Op_i}$ has the following form. $\boldsymbol{Op_i.W(h)=(w_{i1}(h), w_{i2}(h), \ldots, w_{in}(h))}$, where $\boldsymbol{w_{ik}(h)=0}$ if a tuple with history $\boldsymbol{h}$ cannot be routed to operator $\boldsymbol{Op_k}$ for processing next and $\boldsymbol{w_{ik}(h)=cik}$ if a tuple with history h can be routed to $\boldsymbol{Op_k}$. This means that if two tuples can be routed to $\boldsymbol{Op_j}$ and $\boldsymbol{Op_k}$, the probability of being routed to each operator remains proportional, regardless of the histories of the two tuples.

The algorithm **compute_rate** in Figure 4.1 computes the total input rate $\boldsymbol{Op_i.\lambda}$ (as a function of $\boldsymbol{cik}$ and the data generation rates of the data source operators) to the operator $\boldsymbol{Op_i}$. If we treat operator $\boldsymbol{Op_i}$ as an M/M/1 server with input rate $\boldsymbol{Op_i.\lambda}$[2], we can compute the average resident time $\boldsymbol{Op_i.Tr}$ of a tuple at $\boldsymbol{Op_i}$ and the average queue length $\boldsymbol{Op_i.qlen}$ at $\boldsymbol{Op_i}$. We have

$$\boldsymbol{Op_i.Tr = Op_i.Ts/(1-Op_i.\lambda * Op_i.Ts)}$$

$$\boldsymbol{Op_i.qlen = (Op_i.\lambda * Op_i.Ts)/(1-Op_i.\lambda * Op_i.Ts)}$$

where $\boldsymbol{Op_i.Ts = Op_i.r/Op_i.R}$ is the average service time of a tuple at $\boldsymbol{Op_i}$.

The algorithm in Figure 4.1 also computes the arrival rates at the **Sink** operator of each different execution history (or execution path), denoted by $\boldsymbol{Sink.\lambda h}$. The response time $\boldsymbol{Th}$ for a tuple with the execution history $\boldsymbol{h}$ is the sum of residence time at all operators along its path[3], that is, $\boldsymbol{Th = \Sigma_{Opi\ in\ h} Op_i.Tr}$. We can compute the average response time of the system as

$$\boldsymbol{ART=\Sigma_h(Sink.\lambda h * Th)/\Sigma_h Sink.\lambda h}.$$

Branch is a data structure with fields *{op, rate, history}*
active_branches is a list of Branches
The total input rate to each operator ***Opi*** will be stored in ***Opi.λ***. The arrival rate at **Sink** with history ***h*** will be stored in ***Sink.λh***

Line 1-11 initializes the data rates at the data source operators. Line 12-33 recursively computes data rates of the following branches of an execution path till all the paths reach the data sink operator.

Algorithm compute_rate:

```
1  for each operator op:
2     if op is not data source
3          op.λ = 0
4     end if
5     if op is data source:
6          new_branch.op = op
7          new_branch.rate = op.λ
8          new_branch.history = {op}
9          active_branches.add(new_branch)
10    end if
11 end for
12 while active_branches is not empty:
13    curr = active_branches.first()
14    active_braches.remove(curr)
15    (w1, w2, ..., wn) = curr.op.W(curr.history)
16    total_w = $\Sigma_{1<=k<=n}$wk
17    for i in 1,2,...n:
18          if wi!= 0:
19               p= wi/total_w
20               r = curr.rate * curr.op.ρ * p
21               if Opi is not Sink:
22                    Opi.λ += r
23                    new_branch.op = Opi
24                    new_branch.flow = r
25                    new_branch.history =
26                              curr.history.append(Opi)
27                    active_branches.add(new_branch)
28               else:
29                    Sink.λcurr.history += r
30               end if
31          end if
32    end for
33 end while
```

Figure 4.1 Algorithm compute_rate

Equipped with the queuing network model, we can reformulate the two optimization metrics of Section 3 as the following constrained optimization problem for $\boldsymbol{cik}$.

Optimize ART: Solve

$$\boldsymbol{argmin_{\{cik\}}(\Sigma_h(Sink.\lambda h * Th)/\Sigma_h Sink.\lambda h)}\ ,$$

under the constraints $\boldsymbol{Op_i.\lambda < Op_i.R/Op_i.r}$. Notice here that the data rate $\boldsymbol{\lambda}$ at data source nodes are given.

Optimize MDR: Solve

$$\boldsymbol{argmax_{\{cik,\ Opi.\lambda\ where\ Opi\ is\ data\ source\}}\ (\Sigma_{Opi\ is\ data\ source} Opi.\lambda)}$$

under the constraints $\boldsymbol{Op_i.\lambda<Op_i.R/Op_i.r}$ and $\boldsymbol{Op_i.qlen<Op_i.MaxQ}$, where $\boldsymbol{Op_i.MaxQ}$ is the given maximum queue length at $\boldsymbol{Op_i}$ before $\boldsymbol{Op_i}$ is considered overloaded.

[2] In general, the arrival rate at an operator is not exponential. For example, if a join operation produces more than one output tuples, the arrival time at next operator of these tuples are not independent. Other reasons in practice include that the network delay between two physical nodes are not independent. This exponential assumption, however, enables us to analyze the network and because of the averaging effect, still provides a good approximation [14][15].

[3] The formula does not include terms that represent the network delay because we assume the available network bandwidth and the cost of sending the tuple through the network is accounted for in the computing resource.

4.2 Examples

We give two examples to illustrate the application of the solution technique.

Example S3

The first example is a plan consisting of three selection operators, Op_1, Op_2 and Op_3. Op_0 is the data source operator. The three selection operators can be evaluated in any order. Relevant parameters such as the computing resources available to each operator are given in Table 4.1.

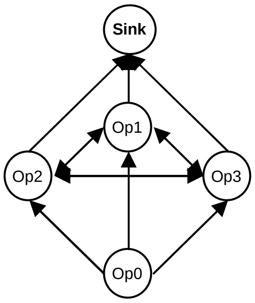

Figure 4.2 Example S3

	R	*r*	*Ts=r/R*	*ρ*
Op_1	10	.1	.01	.5
Op_2	20	.1	.005	.2
Op_3	20	.5	.025	.4

Table 4.1 S3 Parameters

P1	$Op_0,Op_1,Op_2,Op_3,Sink$	P4	$Op_0,Op_2,Op_3,Op_1,Sink$
P2	$Op_0,Op_1,Op_3,Op_2,Sink$	P5	$Op_0,Op_3,Op_1,Op_2,Sink$
P3	$Op_0,Op_2,Op_1,Op_3,Sink$	P6	$Op_0,Op_3,Op_2,Op_1,Sink$

Table 4.2 Possible Execution Paths in Example S3

There are six possible execution paths (P1-P6 shown in Table 4.2) that tuples can take from Op_0 to $Sink$. The data generation rate at Op_0 is $Op_0.\lambda$. Without loss of generality, we can assume $\Sigma_j cij = 1$. We can compute the total arrival rate and resident time at Op_1 as

$$\begin{aligned} Op_1.\lambda = {} & Op_0.\lambda * c01 && \text{--- part contributed by P1, P2} \\ & + Op_0.\lambda * c02 * Op_2.\rho * c21 && \text{--- by P3} \\ & + Op_0.\lambda * c02 * Op_2.\rho * c23 * Op_3.\rho && \text{--- by P4} \\ & + Op_0.\lambda * c03 * Op_3.\rho * c31 && \text{--- by P5} \\ & + Op_0.\lambda * c03 * Op_3.\rho * c32 * Op_2.\rho && \text{--- by P6} \end{aligned}$$

$$Op_1.Tr = Op_1.Ts/(1-Op_1.\lambda * Op_1.Ts)$$

Similar computations can be applied at Op_2 and Op_3. Since a tuple travelling from Op_0 to $Sink$ will visit Op_1, Op_2 and Op_3 exactly once regardless of the path taken, the average response time at **Sink** is,

$$ART = Op_1.Tr + Op_2.Tr + Op_3.Tr.$$

Now we can solve for both **ART** and **MDR** numerically. For example, if given $Op_0.\lambda = 200$, the solution of **ART** is shown in Table 4.3. Suppose $Op_i.MaxQ = 10000, i=1,2,3$, the solution of **MDR** is shown in Table 4.4.

ART	c01	c02	c03	c12	c13	c21	c23	c31	c32
0.14	.21	.78	.01	.98	.02	.99	.01	.15	.85

Table 4.3 Solution of ART of S3 when $Op_0.\lambda$ = 200

MDR	c01	c02	c03	c12	c13	c21	c23	c31	c32
243	.31	.68	.01	.77	.23	.50	.50	.16	.84

Table 4.4 Solution of MDR of S3 when $Op_i.MaxQ$=10000

The cij in the solutions above specify the probability weight of Op_i routing a tuple to Op_j. The results obtained numerically match the intuition that more tuples should be routed to operators which have lower service times (faster, Op_1, Op_2 in the example) and which are more selective (Op_1). Also, observe that several execution paths exist simultaneously in the solution. This is an important difference between the optimal distributed routing policy and the optimal routing policy for a centralized Eddy. For a centralized eddy considered in [18], the optimal policy orders selection operators from most selective to least selective and always applies them in that order. Later in this section, we will show that practical routing policies that are designed to approach the optimal solution for a centralized eddy do not perform well in the distributed environment.

Example J3

The second example is a three way join A⋈ B⋈ C. Stream A, B and C are generated at Op_0, Op_1 and Op_2 with rates $Op_0.\lambda$, $Op_1.\lambda$ and $Op_2.\lambda$ respectively. As discussed in Section 3, the symmetric join algorithm can be carried out in two phases – a sliding window update phase and a join phase. Since the cost of window update operators for stream A, B and C are the same regardless of the routing policy used, we assume that the window update is performed at Op_0, Op_1 and Op_2 and do not consider the delay of updating the window in the optimization problem. Notice that two different join operations will be performed on the sliding window of stream B – the right side join of A⋈ B and the left side join of B⋈ C. Because both operations must access the same sliding window, it is a reasonable requirement to model that both operations compete for the same physical computing resource and share a single queue, shown as Op_4 in Figure 4.3. However, we should note the selectivity of streams A and B at Op_4 can be very different. We assume that Op_4 has selectivity ρ_A on tuples from stream A and ρ_C on tuples from stream C. The parameters for Example J3 are given in Table 4.5.

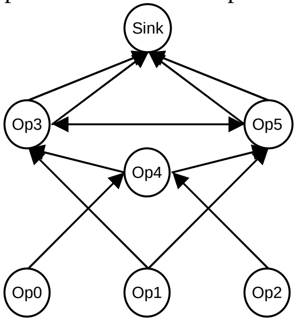

Figure 4.3 Example J3

<table>
<tr><th></th><th><i>R</i></th><th><i>r</i></th><th><i>Ts=r/R</i></th><th colspan="2"><i>ρ</i></th></tr>
<tr><td>Op_3</td><td>10</td><td>.2</td><td>.02</td><td colspan="2">.8</td></tr>
<tr><td rowspan="2">Op_4</td><td rowspan="2">20</td><td rowspan="2">.2</td><td rowspan="2">.01</td><td>ρ_A</td><td>ρ_C</td></tr>
<tr><td>.1</td><td>.4</td></tr>
<tr><td>Op_5</td><td>40</td><td>.4</td><td>.01</td><td colspan="2">.2</td></tr>
</table>

Table 4.5 Parameters for J3

P1	$Op_0,Op_4,Op_5,Sink$	P3	$Op_1,Op_3,Op_5,Sink$
P2	$Op_2,Op_4,Op_3,Sink$	P4	$Op_1,Op_5,Op_3,Sink$

Table 4.6 Possible paths in Example J3

It is reasonable to require that tuples from stream A (and C) must first be joined with stream B in order to avoid having to form the cross product of A and C. There are four possible execution paths (P1-P4) in example J3. The only routing decision is made at Op_1 and the routing weights ***c13*** and ***c15*** must be determined. Again, without loss of generality, we can let $c13 + c15 = 1$. The arrival rates and resident time at each operator can be computed as

$Op_3.\lambda = Op_1.\lambda * c13$ -- contributed by P3
$\quad + Op_1.\lambda * c15 * Op_5.\rho$ -- contributed by P4
$\quad + Op_2.\lambda * Op_4.\rho_C$ -- contributed by P2
$Op_4.\lambda = Op_0.\lambda + Op_2.\lambda$ -- contributed by P2 and P4
$Op_5.\lambda = Op_1.\lambda * c15$ -- contributed by P4
$\quad + Op_1.\lambda * c13 * Op_3.\rho$ -- contributed by P3
$\quad + Op_0.\lambda * Op_4.\rho_A$ -- contributed by P1
$Op_i.Tr = Op_i.Ts/(1 - Op_i.\lambda * Op_i.Ts)$

At the **Sink** operator, the arrival rates and response times along each path are:

$Sink.\lambda_{P1} = Op_0.\lambda * Op_4.\rho_A * Op_5.\rho$
$ART_{P1} = Op_4.Tr + Op_5.Tr$
$Sink.\lambda_{P2+P3} = Op_1.\lambda * Op_3.\rho * Op_5.\rho$
$ART_{P2} = ART_{P3} = Op_3.Tr + Op_5.Tr$
$Sink.\lambda_{P4} = Op_2.\lambda * Op_4.\rho_C * Op_3.\rho$
$ART_{P4} = Op_4.Tr + Op_3.Tr$

The **ART** of the plan is the weighted average of the **ART**s along paths P1 through P4.

$$ART = (Sink.\lambda_{P1} * ART_{P1} + Sink.\lambda_{P2+P3} * ART_{P2} + Sink.\lambda_{P4} * ART_{P4}) / (Sink.\lambda_{P1} + Sink.\lambda_{P2+P3} + Sink.\lambda_{P4})$$

Assuming that the input rates are $Op_0.\lambda$=20, $Op_1.\lambda$=80, $Op_2.\lambda$=20, that $Op_i.MaxQ$ = 10000, and that $Op_0.\lambda : Op_1.\lambda : Op_2.\lambda$ = 1 : 4 : 1, numerical solutions for the **ART** and **MDR** can be found in Tables 4.7 and 4.8, respectively.

ART	c13	c15
.08	.0	1.0

Table 4.7 ART solution of J3

MDR	c13	c15
153	.24	.76

Table 4.8 MDR solution of J3

In example J3, Op_5 is both more selective and faster (smaller ***Ts***) than Op_3. For low data rates, the optimal routing decision for **ART** at Op_1 is to route all tuples to Op_5. For this case an algorithm that is designed for a centralized eddy is likely to perform very well. For the **MDR** problem, however, the optimal solution routes about one fourth of the tuples to Op_3. It is important to observe that the solution for **MDR** balances the workloads between Op_3 and Op_5. One might assume that this solution can be achieved by a policy that routes as many tuples as possible to Op_5 until it is close to the point of being overloaded and then routes the remaining tuples to Op_3. This is not correct because 80% of the tuples routed to Op_3 from Op_1 will be sent to Op_5 for further processing. These tuples will end up overloading Op_5 if Op_1 has already sent too many tuples to Op_5.

4.3 Practical Routing Policies

The numerical solution to the **ART** or **MDR** problems provides the "optimal" routing policy for a distributed stream query plan. This optimal policy cannot, however, usually be achieved in practice because:

1. The solution requires the global configuration information of the distributed network, which is often unavailable.
2. The optimization process is very expensive. The number of possible execution paths is roughly the factorial of the number of operators in the plan.
3. Often parameters such as stream rates, computing resources, and operator selectivity are likely to change during plan execution.

In this section we present some "practical" routing policies. These policies share the following characteristics. Operators in a distributed plan learn statistics during execution, which are exchanged among the operators periodically. Based on these statistics, each operator makes its own routing decision without consulting any central authority or any other operators. This "autonomous" property is desirable in a distributed system.

Before describing the routing policies, we first examine the statistics that can be used to make a routing decision. Table 4.9 shows the statistics that are gathered at each node. The input queue length indicates the load at each operator. Ticket and Selectivity are different measures of learned selectivity of an operator over its input streams.

	Symbol	Meaning and how to measure
Q length	**Q**	Average input queue length, can be obtained through a resource monitor at each operator.
Ticket	**T**	Increased by 1 when an input tuple is received and decreased by 1 when a tuple is placed on the operator's output stream. T is set to 0 when it becomes negative or when the server is overloaded (indicated by queue length that grows greater than a pre-defined threshold).
Selectivity	**S**	Selectivity over a specific time interval $\mathbf{S_{interval}}$ can be measured by a monitor, operator selectivity is updated every time interval using the formula $\mathbf{S_{new} = 0.8*S_{old} + 0.2*S_{interval}}$ This formula allows S to adapt to changes in selectivity over time and avoids sudden changes of S due to spikes in $\mathbf{S_{interval}}$.
Cost	**C**	Average service time ***Ts*** to process a tuple, can be obtained through monitor at each operator.

Table 4.9 Statistics information used by routing policies

Next we introduce six practical routing policies.

Routing Policy Q (Q length)

The first policy is the “back-pressure” policy described in the original Eddy paper [2]. The idea is that a more heavily loaded operator will apply a larger “back-pressure” to its input stream, forcing tuples to be routed to more lightly loaded operators. When an operator outputs a tuple, the tuple is routed to the operator that has the shortest queue length Q among all possible operators that can process the tuple next. Notice that policy Q does not route an equal number of tuples to each operator. An operator with smaller ***Ts*** can consume tuples at a faster rate and thus more tuples are routed to it.

Routing Policy T (Ticket)

This policy is the smart eddy algorithm described in [2][18]. A ticket T is used as a rough indicator of the learned selectivity at each operator. A tuple is routed to the most selective operator (i.e. the largest ticket). The reason behind this policy is that more tuples are dropped at more selective operators, thus saving unnecessary processing at operators that are less selective. Note that an operator sets its ticket to 0 when it becomes overloaded (as indicated by the length of its input queue). Our results indicate that this is very important, for otherwise the most selective operator becomes the bottleneck of the system and the system becomes congested at rather low data rates. Following the policy in [18], if the tickets of possible output operators are all zeros, the routing policy reverts to policy Q.

Routing Policy SC (Selectivity-Cost)

Policy Q does not consider the selectivity of an operator; therefore tuples are likely to be routed to an operator with low selectivity but which is “fast” (low ***Ts***). The policy T on the other hand routes tuples to highly selective operators without considering the cost of processing a tuple at the operator. Policy SC uses the following “benefit” to make routing decisions. The benefit of routing a tuple to operator Op_i is defined as

$$Benefit\_SC_i = (1-Op_i.S) / Op_i.C$$

If $Op_i.S$ is larger than 1, the benefit is set to zero. Intuitively, this benefit is the possibility that a tuple be eliminated by an operator divided by the cost of processing the tuple. An output tuple at each operator is routed to the next eligible operator with the biggest benefit.

Routing Policy WSC (Weighted Selectivity-Cost)

The WSC policy is inspired by the fact that several execution paths exist simultaneously in the optimal plan derived from the analytic model. In the analytic solution, operator Op_i routes tuples to the next operator Op_j with a certain probability according to weight ***cij***. Policy WSC mimics the analytic solution by routing a tuple to Op_j with probability proportional to weight W_SC_j, which is computed as

$$W\_SC_j = (Benefit\_SC_j)^2$$

At Op_i, W_SC_j corresponds to the ***cij*** in the analytical model.

Routing Policy SCQ (Selectivity-Cost-Qlength)

Policy SC considers the per-tuple processing cost at an operator. The cost does not consider the queuing time that a tuple incurs while waiting to be processed. Tuples will be routed to an operator with large benefit regardless of the load at the operator. In policy SCQ, we define

$$Benefit\_SCQ_i = (1-Op_i.S) / ((1+Op_i.Q) *Op_i.C)$$

$Benefit\_SCQ_i$ takes queue length into consideration. The cost of processing a tuple is the sum of the processing and queuing time. When an operator is more heavily loaded, routing a tuple to it will have less benefit.

Routing Policy WSCQ (Weighted Selectivity-Cost-Qlength)

Like the WSC policy, policy WSCQ routes tuples to the next possible operator with certain probability. The weight of routing a tuple to Op_i is

$$W\_SCQ_i = (Benefit\_SCQ_i)^2$$

Here we give a short justification of using the square in computing the weight. When a tuple t is routed to Op_i, the resident time (service time plus queuing time) of t at Op_i is $(1+ Op_i.Q)*Op_i.C$. The routing decision, however, not only affects tuple t, as t also contributes to the queue length at Op_i. Thus, tuples sent to Op_i after t will incur a longer queuing time than they would have had t not been routed to Op_i. Our experiments showed that Benefit alone is not a good weight function for either WSC or WSCQ while $Benefit^2$ shows very good results for both WSC and WSCQ.

4.4 Simulation Results

We simulated each routing policy using CSIM [24], a discrete event simulator. The system is simulated as a number of physical nodes connected by a network. Each operator of a distributed plan executes on one physical node and one physical node may have several operators running simultaneously. Tuples are modelled as messages exchanged between operators. If several operators reside on one physical node, tuples sent to these operators are multiplexed over one physical link of the network and de-multiplexed into the message queue of each operator. Tuples transferred on one network link are grouped into pages for transmission. A tuple may be buffered for at most one time unit and half full pages maybe transferred if this timer expires. Tuples are generated at data source operators with an exponential rate. Each operator gathers the statistics listed in Table 4.9 and broadcasts its learned statistics to other operators after 5 time units or after processing every 40 input tuples[4]. Messages containing statistics have priority over data. They are not queued with data packets and are processed immediately when received. The important parameters of our simulation model are:

1. The computing resources at each physical node. We used values from 500 to 1000 in our simulation.
2. The bandwidth of the network link between two physical nodes. We used values randomly selected from 1MB to 100 MB per time unit.
3. Computing resources required by each operator to process one tuple. We used values from 1 to 20.
4. The page size used for network transfers (4Kbytes).
5. The average tuple size is 100 bytes; therefore, there are 40 tuples per page.

We first report results for example S3 (Figure 4.2) from Section 4.2 and compare the results of the different policies with the solutions derived using the analytical model. Table 4.10 shows the **ART** simulation results for example S3. If the “overloaded” condition for an operator is defined as having a queue length of 10,000, the optimal solution of the analytic model can handle a maximum data rate of λ=243. Shaded entries with bold fonts indicate major deterioration in **ART** for a routing policy.

Data Rate	**200**	**210**	**220**	**230**	**240**
Model	.14	.18	.25	.43	1.57
T	.35	***22***	***26***	***35***	***45***
Q	1.5	2.1	2.6	3	3.8
SC	.11	2.3	***23***	***34***	***44***
WSC	.12	.49	9.0	***20***	***29***
SCQ	.54	.95	1.2	1.4	2.2
WSCQ	.22	.44	.61	0.8	1.6

Table 4.10 ART of Example S3

Not surprisingly, no practical policy has a better average response time than the numerical solution from the analytical model. Policies Q, SCQ, and WSCQ all can handle a data rate of λ=240 without incurring a major deterioration in **ART**. These results demonstrate that incorporating queue length as a routing parameter is critical in maximizing the rates at which tuples are processed. Policy T exhibits the worst **MDR** results because it depends only on tickets. Once the most selective operator (***Op*$_2$**) becomes saturated, it sets its ticket to 0 but this quickly induces congestion at the next most selective operator. After all operators are congested the routing policy falls back to policy Q – only worse because of the congestion.

Policy Q balances the load between operators but too many tuples are sent to low selectivity operators and cause too much unnecessary processing. Compare **ART** of policy Q to those of policies SCQ and WSCQ, we can see it is important to consider selectivity and cost in order to reduce **ART**. SCQ and WSCQ show good results across low and high data rates. Both WSC and WSCQ outperform their un-weighted counterparts in **ART**. Policy WSC also shows better **MDR** than policy SC.

4.4.1 Single Query Simulation Results

We also ran simulations of 100 distributed plans running on a network consisting of 10 physical nodes. The number of operators in each plan is uniformly randomly chosen between 20 and 30 (not including data source operators and the **Sink** operator). The average number of operators per plan was set at 25. The input rates (data generation rates at the data source operators) of the 100 distributed plans are then varied to generate a total of 1000 queries to evaluate the **ART** of the different policies. We choose relatively high data rates to demonstrate that the different routing policies will exhibit different queuing behaviours[5]. Each query is run by itself (i.e. multiprogramming level of 1).

	T	**Q**	**SC**	**WSC**	**SCQ**	**WSCQ**
GM(ART)	158	66.6	6.0	4.7	3.7	1.92

Table 4.11 Geometric Means of ART

Table 4.11 shows the geometric mean of the **ART** of each policy for the 1000 queries. We can see that policy WSCQ out-performs the other policies by a significant margin. WSCQ considers selectivity, execution cost, and the load on each operator (i.e. queue length) and performs better than those policies that only consider one or two factors. Comparing the performance of WSCQ with that of SCQ (WSC with SC), we conclude that using weighted probability in routing is an effective technique to reducing the **ART**.

In our simulation, we treat a physical node as overloaded if more than 10,000 messages are queued at that node. Table 4.12 shows the geometric mean of the **MDR** of the different scheduling policies for the 1000 test queries. Policies SCQ and WSCQ have much higher throughputs than the other policies. From the simulation results for example S3 in Section 4.2, we concluded that queue length is a very important factor in routing policy to achieve higher data rates. The results in Table 4.12 demonstrate that queue length alone is not adequate.

	T	**Q**	**SC**	**WSC**	**SCQ**	**WSCQ**
GM(MDR)	35	150	194	277	349	347

Table 4.12 Geometric Means of MDR

[4] The system is easily overloaded if operators exchange statistics too infrequently. On the other hand, the statistics information is noisy if operators exchange statistics in very short intervals. Our experience shows that exchanging statistics information after processing 10 to 50 tuples is appropriate for our simulation.

[5] The queuing time will be low if the data rate is too low and the time that tuples are buffered in network to fill a page will be a major factor of **ART**.

4.4.2 Multiple Query Simulation Results

In a real data stream management system, it is highly likely that many long running queries will be running concurrently. Using our simulation model we also conducted simulations with 20 simultaneous queries (each has 10 to 40 operators with an average of 25 operators per plan) running on a network of 40 nodes[6]. Tables 4.13 and 4.14 contain the **ART** and **MDR** results respectively.

	T	Q	SC	WSC	SCQ	WSCQ
GM(ART)	331	3.3	21	26	1.52	1.3

Table 4.13 Geometric Mean of ART for multiple plans

	T	Q	SC	WSC	SCQ	WSCQ
GM(MDR)	22	94	61	63	96	97

Table 4.14 Geometric Mean of MDR for multiple plans

Again the WSCQ routing policy minimizes the **ART**. Compared with the single plan cases, the **ART**s of policies SC and WSC relative to policy WSCQ are much worse plan while the **ART** of policy Q relative to WSCQ is much better. There is no significant difference between the **MDR**s of policies Q, SCQ and WSCQ and they are much better than policies that do not consider queue length. We conclude that queue length is the most important factor in making routing decisions when there are many plans in the system.

The effect of using weight in WSCQ is not as obvious as the single plan cases because each physical node may have several operators from different plans and the workload is automatically balance across physical nodes.

5. ROUTING POLICY OF PLAN WITH DYNAMIC RESOURCE ALLOCATION

Load sharing is an important technique to achieve scalability and efficiency in a distributed system. [6] describes several techniques to facilitate load sharing across several cooperating physical nodes. Here we briefly describe these techniques. If more than one operator (box in Aurora*) resides on an overloaded physical node, one or more of the operators can be moved to neighbouring physical nodes through **box sliding**. If one operator on a physical node consumes a large amount of resources, this operator can be split into two copies through **box splitting**. One copy of the operator can then be moved to another physical node. Figure 5.1 illustrates these techniques. The dashed line boxes indicate a physical node.

These load sharing techniques raise many important and difficult issues. [6] points out a number of problems including when to apply these techniques, which operator these techniques should be applied to, and possible trade offs when splitting one operator is much more expensive than splitting another. We do not try to attack the problem in such a general form. Instead, we focus on the impact of routing policies on load sharing in the system. We assume the following simple and natural load sharing policy. If an operator consumes a large amount of resources at a physical node and the node is overloaded, the operator is split and sent to nodes that are under utilized. It is easy to see that the routing policy used in the system will have an impact on which operators are more likely to be overloaded. Therefore, the routing policy will affect the resource allocation between operators.

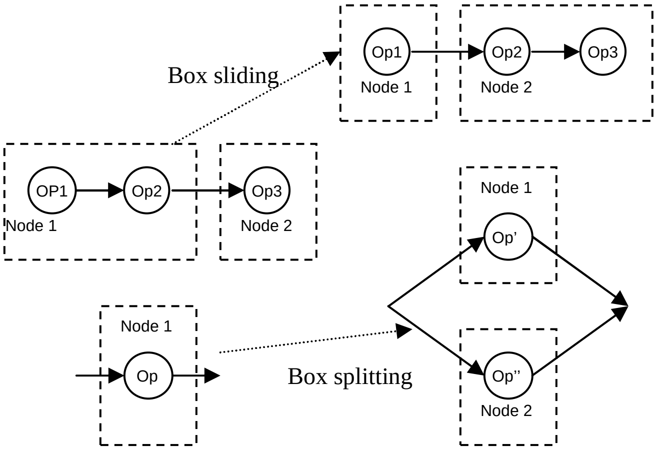

Figure 5.1 Box Sliding and Box Splitting

Physical nodes that share a workload using box sliding and box splitting are cooperative in the sense that a node receiving an operator must have enough computing resources to execute the operator. We model box splitting and box sliding using the following paradigm. Like the static resource allocation case, a distributed stream query plan consists of n operators $\boldsymbol{Op_1}$, $\boldsymbol{Op_2}$, …, $\boldsymbol{Op_n}$. The total computing resources (aggregated across all nodes on which the operators are running) available to the query are denoted as ***Total_R***. The computing resource allocated to operator $\boldsymbol{Op_i}$ from ***Total_R*** is denoted as $\boldsymbol{Op_i.R}$. Splitting and sliding an operator $\boldsymbol{Op_i}$ is modelled as increasing the resources allocated to it. Resources of operators on the physical node that receives a copy of the operator will be reduced accordingly. Copies of an under-loaded operator may be merged later to release resources back to the ***Total_R***. Next we consider the problem of optimizing for the maximum data rate the system can handle.

Optimize MDR: Given a distributed plan with operators $\boldsymbol{Op_1}$, $\boldsymbol{Op_2}$, …, $\boldsymbol{Op_n}$ and a total amount of resources ***Total_R***, compute the routing weight function ***cij*** and computing resources $\boldsymbol{Op_i.R}$ of each operator for the maximum data rate **MDR** under the constraints:

$$\boldsymbol{Op_i.\lambda < Op_i.R/Op_i.r}$$

$$\boldsymbol{Op_i.qlen < Op_i.MaxQ}$$

$$\boldsymbol{\Sigma Op_i.R < Total\_R}$$

[6] We have also experimented with different numbers of physical nodes in the network. Our conclusions hold across simulations with different number of nodes.

The techniques for solving this problem are similar to those for the static network described in Section 4.1 so we do not repeat them here. The only difference is that the computing resources ***R*** at each operator become variables and there is an additional constraint on the sum of these resources.

We implemented box splitting and box sliding in our simulation model by creating a copy of the operator and shipping the copy to the destination physical node. Some of the tuples queued at the original copy of the operator are also shipped to the new copy. This implementation is simpler than a real system because splitting an operator that has state (such as a join operator) may also need to ship its state to the destination physical node. Our simulation ignored this cost because we expect that box sliding does not occur very frequently in real systems and some real system implementations simply allow tuples to be dropped during splitting; tolerating a decrease in the QoS for a very short period of time [6].

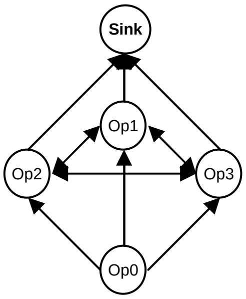

Figure 5.2 Example DS3

	r	$Ts=r/R$	ρ
Op_1	.1	.1/$Op_1.R$	.5
Op_2	.1	.1/$Op_2.R$	.2
Op_3	.5	.5/$Op_3.R$	.4

$Op_1.R+Op_2.R+Op_3.R$ <= 100

Table 5.1 DS3 Parameters

We use example DS3 in Figure 5.2 to illustrate dynamic resource allocation in a distributed plan. DS3 has a similar network topology to that of example S3 in Section 4.2 except that Op_1, Op_2 and Op_3 share a total of 100 units of computing resources. The numerical solution of **MDR** for DS3 is 570. Table 5.2 shows the numerical solution to the **ART** for the analytic model and the simulation results of **ART** for the different routing policies. Shaded entries indicate a major deterioration in **ART** for a routing policy. We can see that policy SCQ and WSCQ achieve much higher maximum data rate than other polices and are close to the solution derived using the analytical model.

Data Rate	**200**	**300**	**400**	**500**
Model	.04	.06	.09	.23
T	.06	.27	.56	***20***
Q	.99	***81***	***512***	***760***
SC	.04	.74	***208***	***332***
WSC	.04	.07	***50***	***124***
SCQ	.05	.18	.29	0.6
WSCQ	.05	.12	.45	0.8

Table 5.2 ART of Example DS3

Figure 5.3 shows the actual allocations of resources among the three operators when the system begins to be overloaded for each routing policy.

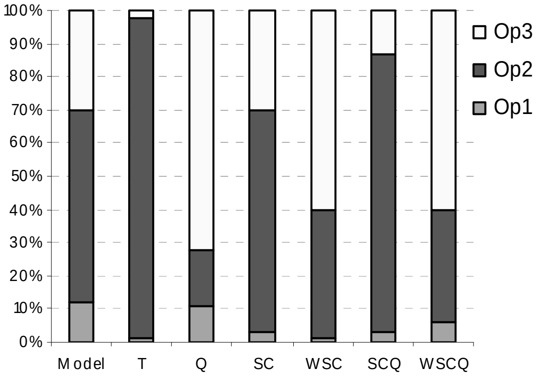

Figure 5.3 Actual Resource Allocations for Example DS3

The solution derived using the analytic model allocates most of the resources to Op_2, which has high selectivity and is efficient at processing a tuple. Intuitively, this allocation strategy routes tuples early to more efficient operators. Routing tuples early to high selectivity operators automatically reduces the workload for subsequent operators in the query. Policy SC and SCQ route tuples to more efficient and more selective operators first; therefore, they are more aggressive in allocating resources to those operators. Policy T considers selectivity only and can be over aggressive in allocating resources to more selective operators. The probability weights in WSC and WSCQ attempt to balance the workload between operators. These policies are more conservative in allocating resources. Policy Q attempts to balance queue lengths across operators. Thus, most resources are allocated to the most expensive operators; this is usually an inferior resource allocation strategy.

We have conducted simulation experiments of each of the routing policies with dynamic resource allocations. The plans used are the same as the plans in Section 4.4.1 except that the operators in each plan can be split and moved to another physical node. A physical node will accept a copy of an operator only if the node is not overloaded (when less than 2000 tuples are queued at the node). Table 5.3 shows the **MDR** results of each policy with and without dynamically adjusting resources between operators. The MDR values for each routing policies in the static case are taken from Table 4.12. We observe that dynamically adjusting resources between operators can improve the **MDR** of policy SCQ and WSCQ by a factor of two to three. We conclude that box splitting and box sliding are very effective techniques to improve **MDR** in a distributed stream management system. Of the six policies, SCQ and WSCQ have better results than the others.

	T	**Q**	**SC**	**WSC**	**SCQ**	**WSCQ**
MDR(static)	35	150	194	277	349	347
MDR(dynamic)	311	175	250	835	1315	1297

Table 5.3 Geometric Mean of MDR

Higher data rates are used to study the **ART** of these policies because the rates used in Table 4.11 are too low to show queuing behaviour of policy WSC, SCQ and WSCQ. Table 5.4 shows the **ART** of the six policies.

	T	Q	SC	WSC	SCQ	WSCQ
GM(ART)	607	1247	248	34	17	31

Table 5.4 Geometric Mean of ART

The routing policy plays an important and complex role in determining the system's throughput. As discussed in Section 4, given a fixed resource allocation, balancing the workload between nodes is an important factor in making routing decisions to achieve good average response time and system throughput. However, balancing the workload also turns out to be conservative in allocating resource to more selective and more efficient operators. Our simulation results indicate that the **ART** of policy SCQ is about one half of that of policy WSCQ due to more efficient resource allocation.

We also simulated systems with multiple plans using the same test cases from Section 4. The results are shown in Table 5.5 and Table 5.6.

	T	Q	SC	WSC	SCQ	WSCQ
GM(ART)	71	323	3030	89	21	19

Table 5.5 Geometric Mean of ART for multiple plans

	T	Q	SC	WSC	SCQ	WSCQ
MDR(static)	22	94	61	63	96	97
MDR(dynamic)	561	428	99	279	691	657

Table 5.6 Geometric Mean of MDR for multiple plans

Table 5.6 shows the **MDR** results with dynamic resource allocation are also much higher than **MDR** results of static resource allocation (taken from Table 4.14), which demonstrates box splitting and box sliding are also very effective in systems with multiple plans. Like the single plan cases, the SCQ and WSCQ polices perform better than other policies in terms of both the **ART** and the **MDR** metrics. Having multiple plans in the system automatically balances the workload. Therefore, the impact of the weight used by the WSCQ policy is diminished. Policy SCQ and WSCQ have similar results in both **MDR** and **ART**.

6. CONCLUSION AND FUTURE WORK

In this paper, we have studied the impact of routing policies on the performance of a distributed data stream management system. We used a queuing network to build an analytic model for a distributed query plan and defined two performance metrics, the average response time and maximum data rate. We studied six practical routing policies and compared their relative performance using a discrete event simulator. We conclude that in a distributed plan in which each operator has a fixed amount of computing resources allocated to it, routing policies that consider operator selectivity, execution cost and operator load outperforms simpler policies that only consider one or two factors. Routing policies that are designed for centralized eddy [2][18] do not perform well in a distributed environment. Routing tuples using weighted probabilities is an effective technique to achieve shorter **ART**. Overall, the policy WSCQ performs well compared to other policies. For a distributed plan that can dynamically allocate computing resources between operators, our simulation results demonstrate that adjusting resources between operators (for example, using box splitting and box sliding) is very effective in improving system throughput. We observed that there are two factors that affect system throughput: balancing the load between operators to avoid congestion and allocating resources efficiently. Our experiments showed that SCQ and WSCQ outperform other policies by a significant margin.

In terms of future work, there are many research opportunities in studying routing policies in a distributed data stream management system. We list some of them here.

1. An overloaded node may drop tuples as "load-shedding" technique [6]. Because the routing policy is an important factor in deciding which node will be overloaded, the interaction between routing policies and the policy used to drop tuples will affect the QoS (as defined in [6][8]) of the system.
2. There are many open issues in "splitting" or "sliding" operators between physical nodes. We have only considered one simple strategy, that is, to add resources to overloaded (indicated by queue length) operators. It is worthwhile to investigate the relationship between the more complex load sharing policies and the routing polices. Especially, we discovered that under our simple load sharing policy, balancing the load between operators and efficient allocation of resources are two competing factors. A routing policy that is "cooperative" with load sharing policy is needed to achieve both efficient resource allocation and efficient execution.

ACKNOWLEDGEMENT

This work was supported by the National Science Foundation under grant ITR 0086002.

REFERENCES

[1] A. Arasu, B. Babcock, S. Babu, J. McAlister, and J. Widom. Characterizing memory requirements for queries over continuous data streams. In *Proc. of the ACM Symp. on Principles of Database Systems (PODS)*, 2002.

[2] R. Avnur, J. Hellerstein. Eddies: Continuously Adaptive Query Processing. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2000.

[3] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and Issues in Data Stream Systems Invited paper in *Proc. of the ACM Symposium on Principles of Database Systems (PODS)*, 2002.

[4] B. Babcock, M. Datar and R. Motwani. Sampling From a Moving Window Over Streaming Data. In *Proc. of Annual ACM-SIAM Symp. On Discrete Algorithms (SODA),* 2002.

[5] B. Babcock, S. Babu, M. Datar, R. Motwani. Chain: Operator Scheduling for Memory Minimization in Stream Systems. *In Proc. Of ACM SIGMOD International Conference on Management of Data*, 2003

[6] H Balakrishnan, D. Carney, et al. Aurora*: A Distributed Stream Processing System. Submitted for publication.

[7] P. Bonnet, J. E. Gehrke, and P. Seshadri. Towards Sensor Database Systems. In *Proceedings of the Second International Conference on Mobile Data Management*, 2001.

[8] D. Carney, U. Cetintemel, M. Cherniack, et al. Monitoring Streams: A New Class of Data Management Application. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002.

[9] S. Chandrasekaran, M. J. Franklin. Streaming Queries over Streaming Data. In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB)*, 2002

[10] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, NiagaraCQ: A Scalable Continuous Query System for Internet Databases. *In Proc. of ACM SIGMOD International Conference on Management of Data*, 2000.

[11] M. Datar, A. Gionis, P. Indyk and R. Motwani, Maintaining Stream Statistics over Sliding Windows, In *Proc. Of Annual ACM-SIAM Symp. On Discrete Algorithms (SODA),* 2002.

[12] A. Dobra, M. Garofalakis, J. Gehrke, R. Rastogi., Processing Complex Aggregate Queries over Data Streams. In *Proc. of ACM SIGMOD International Conference on Management of Data*, 2002.

[13] S. Guha, N. Koudas, K. Shim. Data-Streams and Histograms. In *Proc. of ACM Symposium on Theory of Computing (STOC),* 2001

[14] L. Kleinrock, Queueing Systems, Volume I: Theory. New York, Wiley, 1975

[15] L. Kleinrock, Queueing Systems, Volume II: Computer Applications, New York, Wiley, 1976.

[16] Flip Korn, S. Muthukrishnan, D. Srivastava. Reverse Nearest Neighbor Aggregates Over Data Streams, In *Proceedings of the 28th International Conference on Very Large Data Bases (VLDB),* 2002

[17] L. Liu, C. Pu, W. Tang. Continual Queries for Internet Scale Event-Driven Information Delivery. *IEEE Transactions on Knowledge and Data Engineering (TKDE) 11(4),* 1999

[18] S. Madden, M Shah, J. Hellerstein, V. Raman. Continuously Adaptive Queries over Streams. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002

[19] G. Manku, S. Rajagopalan, and B. G. Lindsay. Approximate medians and other quantiles in one pass and with limited memory. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1998.

[20] Y. Matias, J. Vitter, and M. Wang. Wavelet-based histograms for selectivity estimation. In *Proc. of the ACM SIGMOD Intl. Conf. on Management of Data*, 1998.

[21] Y. Matias, J. Vitter, and M. Wang. Dynamic maintenance of wavelet-based histograms. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases (VLDB)*, 2000.

[22] R. Motwani, J. Widom, A. Arasu, et al. Query Processing Approximation and Resource Management in a Data Stream Management System. In *Proceedings of the 2002 Conference on Innovative Data System Research (CIDR),* 2002

[23] P. Roy, S. Seshadri, S. Sudarshan and S. Bhobe. Efficient and extensible algorithms for multi query optimization. In *Proc. of the ACM SIGMOD International Conference on Management of Data*, 2000

[24] H. Schwetman, CSIM--18 the simulation engine. In *Proc. Of the 1996 Winter Simulation Conference*, 1996.

[25] T. Sellis, Multiple Query Optimization. *ACM Transactions on Database Systems*, 1986

[26] P. Tucker, D. Maier, Exploit Punctuation Semantics in Data Streams, In *18th International Conference on Data Engineering (ICDE)*, 2002

[27] S. Viglas and J. F. Naughton. Rate-Based Query Optimization for Streaming Information Sources. In *Proceedings of the ACM SIGMOD International Conference on Management of Data*, 2002.

[28] A. N. Wilschut, P. M. G. Apers. Dataflow Query Execution In A Parallel Main-Memory Environment. *Proc. of the First International Conference on Parallel and Distributed Information Systems (PDIS),* 1991.