

**Title:** Rate-Based Query Optimization for  
Streaming Information Sources  
**Authors:** Efstratios Viglas, Jeffrey F. Naughton  
**Paper Number:** 233  
**Area:** Core Database Technology  
**Category:** Research  
**Relevant Topics:** Optimization and Performance  
**Contact Author:** Efstratios Viglas  
Department of Computer Sciences  
University of Wisconsin-Madison  
1210 W. Dayton Street  
Madison, WI 53706  
*e-mail:* [stratis@cs.wisc.edu](mailto:stratis@cs.wisc.edu)  
*phone:* (608) 262 2252

# Rate-Based Query Optimization for Streaming Information Sources

Efstratios Viglas

Jeffrey F. Naughton

Department of Computer Sciences  
University of Wisconsin-Madison,  
1210 W. Dayton Street, Madison, WI 53706  
E-mail: {stratis, naughton}@cs.wisc.edu

## Abstract

Query optimizers typically attempt to minimize response time. While this approach has been and continues to be very successful in traditional environments, in the presence of information sources of differing rates and non-blocking query execution something else is needed. In this paper, we propose a framework for rate-based optimization, in which the goal is to choose a plan that maximizes the rate at which answers are produced over a specified time interval. We demonstrate that different plans can have substantially different behavior with respect to the rate at which they produce tuples as a function of time. For example, it is possible that one plan is the fastest to produce the first 100 tuples, while another plan produces the most output tuples in the first 10 seconds, while yet another plan is the fastest to run to completion. We develop a rate-based cost model and study this new model in the context of variable rate input streams. We validate this model for the Niagara system. Our results show that optimizing using this cost model and using a greedy top-down search strategy can yield better decision than existing optimization techniques in this environment.

## 1 Introduction

Query optimization is an extensively investigated part of query execution. Cost models, optimization and plan generation strategies have been in the focus of the research community for more than twenty years, and each commercial database product has its own approach to tackle the problem. Most optimizers try to find the plan that minimizes total query execution time. There has been some work on variants such as minimizing the time until the first tuple has been produced (e.g.,

[10]), or minimizing some combination of resource utilization and response time (e.g., [13]), or optimizing for resource allocation in parallel systems [4, 7]. However, we are not aware of any optimizer that explicitly focuses on optimizing the rate at which answers are produced as a function of time.

A couple of trends are combining to render these existing optimization objectives insufficient. First, with the advent of the Internet, there is far greater interest in evaluating queries for which information sources can be remote, and which deliver information at different rates. Second, researchers have begun focusing on query evaluation techniques that are non-blocking and/or adaptive. The purpose of this paper is to re-address query optimization in the context of network-resident, differing rate information sources by optimizing for output production rate, based on estimates of the rates for streams appearing in a query, as well as on classic selectivity statistics.

When one is dealing with network-resident data, some basic assumptions about query execution have to be revisited:

- Execution often depends upon the input rates, and different sources will have different rates. Everyone who has ever downloaded information from the Internet knows that different sites transmit at different rates.
- Some evaluation algorithms are no longer pertinent. For instance, to evaluate a join using the classic implementation of sort-merge join, one has to have both inputs completely present at the beginning of execution. If the input sources are streaming, no answers can be produced before this point.
- When posing queries over the Internet, one is interested in a new set of properties regarding the time the result set appears. In traditional database systems, a user might be interested in seeing the first or the last result as soon as possible. For non-blocking evaluation plans over network resident data, a user might want to optimize for the first 100 results, or the first 1000; or, perhaps, to optimize the number of results that are produced in the first 5 seconds.

The focus of this paper is the introduction of rates into the optimizer cost model. We extract specific formulas to calculate output rates for various operators that appear in execution plans as a function of the input rates. We present a case for why it is better to use symmetric operators in query execution for streaming data by extracting cost formulas for common evaluation algorithms. We show what optimization opportunities arise and how, given the new cost model, we can perform

more detailed query optimization for specific time points in the query execution process. We validate this new framework through experiments with the Niagara system.

## 1.1 Motivation

In the context of the Niagara Project [3, 8, 12] we have implemented an Internet Query Engine over network-resident, streaming, XML information sources. We experimented with various ways of organizing plans involving multiple join operations, and measured their performance by keeping track of the exact time at which each individual output appeared. Each plan employed symmetric algorithms to evaluate the various join predicates. We generated deep as well as bushy plans, and upon examination of the results, an interesting pattern arose. Figure 1 depicts the performance of three different execution plans, for the same three-way join query, for the first eight seconds of query execution.

Clearly, the plans generate their results at different rates. Furthermore, the deep plan gives the most results over the first 6 seconds; bushy plan 1 is best from 6 to 8 seconds; and bushy plan 2 is somewhere in between most of the time. We tried to identify which factor was the one that made one plan faster than an other and how we could take advantage of that factor when implementing an optimizer. This paper reports on the results of this study.

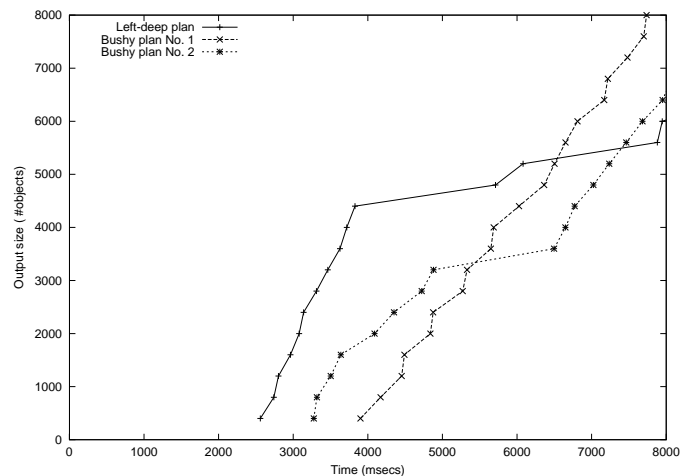


Figure 1: Comparison of three different execution plans for network-resident, streaming information sources

## 1.2 Our contribution

The starting point of this work is investigating the optimization and query execution process under different assumptions than those current database systems employ: information sources with different delivery rates, and optimization for rate of final output. The contribution regarding these aspects is three-fold:

- We first identify the problems and drawbacks of the current optimization process when applied to our problem domain. These mainly stem from the fact that standard query optimization techniques do not consider the rates of their input sources. We make a first step toward eliminating these problems by assigning a rate to each stream, thus adding another parameter for consideration by the optimization process. We show by using specific formalisms and metrics how this parameter influences query execution.
- We extract expressions for each operator used in non-blocking query plans. These expressions incorporate the new rate parameter. We combine all these expressions into a new cost model and provide a case study which proves that the new cost model can provide a better decision basis for the optimization process.
- We propose a unified framework to optimize for specific points of time during query execution.

As far as the last point is concerned, Section 5 shows that optimizing for rate in addition to resource utilization may require that the optimization process considers bushy plans as well. Considering bushy plans explodes the search space substantially. However, as we show in Section 3.4, when dealing with streaming information sources we are better off employing symmetric, non-blocking execution operators. Symmetric operators do not distinguish between outer and inner sources (i.e., they treat  $A \bowtie B$  and  $B \bowtie A$  as equivalent expressions with regards to cost) so it might be possible to take advantage of the trade-off: consider bushy plans with symmetric operators, at the expense of not considering plans not employing symmetric, non-blocking operators.

The rest of the paper is organized as follows: Section 2 presents the related work in this area. Section 3 deals with the estimation of output rates for the most important operators appearing in a query execution plan, while section 4 presents the general framework for rate-based query optimization which is based on the premises of output rate optimization, as well as estimates of plan performance. Section 5 presents a case study concerning different plans and what the new

cost model predicts about the rate at which they produce results. Finally, Section 6 summarizes our conclusions of this work and identifies future research directions.

## 2 Related Work

The seminal paper on cost-based plan optimization was [11]. Other optimization models, especially in the areas of parallel query optimization are not cost-based but deal with resource scheduling and allocation [4, 7]. The Britton-Lee optimizer could optimize for the first result tuple [10], while in Mariposa the optimization criterion was a combination of execution time and resource utilization. Modeling streaming behavior through input rates, and modeling network traffic as Poisson random processes, has appeared in many contexts, including [2], although to our knowledge it has not been applied in the context of query optimization.

A lot of work has been carried out in the areas of non-blocking algorithms [5, 14, 16], which aim at producing plans that do not block their execution because of slow input streams. These algorithms are symmetric in the sense that they do not assign different roles to the participating inputs (i.e. there is no distinction between outer and inner streams). Our framework employs such algorithms. In the same context, methodologies aiming at identifying blocked parts of an execution plan and isolating them [15] can benefit from our framework of rate optimization by dealing with and/or switching to plans for which the predicted output rate is maximized, in addition to their rescheduling and synthesis framework.

Our work fits also into the re-optimization frameworks of [6, 9]. All of these frameworks focus on identifying performance bottlenecks of an already executing plan and ways to overcome them. Our work provides insight into why these bottlenecks may appear when dealing with network-resident, streaming data, and our cost models could be used to make decisions at run time when re-optimizing.

Finally, the adaptive query execution framework of [1] can be used in the context of our work. The authors try to find specific points in their query execution strategy at which they should switch plans. These points are detected by continuously monitoring the execution plan’s performance. Our framework, by modeling output rates as a function of time, can provide estimates of when such time-points of sub-optimal performance will appear, and can predict which plan should be used at each point of the execution.

### 3 Estimation of Output Rates

We are interested in estimating the output rates of various operators as a function of the rates of their input. We will concentrate on the most important and basic operations, which are used in all of relational, object-relational, object-oriented and semi-structured database systems. This class of operators corresponds to a general class of queries, namely *conjunctive queries* consisting of selections, projections and joins. The challenging operators for rate estimation are filtering operators (selections and joins) since these are the ones that selectively qualify their inputs.

Throughout this section we will make a number of simplifying assumptions. Our experimental evaluation suggests that while these assumptions may reduce the absolute accuracy of our estimates, the estimates so derived are good enough to be useful for rate-based query optimization. However, it is certainly possible (indeed, likely) that future work will discover better approximations that can be used in our framework to further increase its accuracy.

All of our subsequent computations build on one simple observation: the rate of a stream is defined to be the number of data objects transmitted divided by the time needed to make this transmission. For clarity of exposition we will concentrate on the transmissions made within approximately one time unit. The general formula is as follows:

$$\text{Output rate} = \frac{\text{Number of outputs transmitted}}{\text{Time needed to make the transmission}} \quad (\text{Equation 1})$$

In what follows, we will use the cost variables of Table 1 to model an operation, refining them as we present the cost model. Whenever we need to refer to an input rate, we will refer to it by using the symbol  $r_i$ , while  $r_o$  refers to an output rate. In the case of joins, which need two inputs, we will refer to them by  $r_l$  for the left-hand side input and  $r_r$  for the right-hand side one.

<i>Cost variable</i>	<i>Meaning</i>
$C_\pi$	Cost of projecting specific parts of an input object
$C_\sigma$	Cost of performing a selection on an input object
$C_{L\bowtie}$	Cost of handling an input coming from the left-hand side of a join
$C_{R\bowtie}$	Cost of handling an input coming from the right-hand side of a join
$T$	Cost of making a single transmission

Table 1: Cost variables for estimation of output rates

### 3.1 Projections

Given that each input has a handling cost, there are two cases which we have to account for:

1. The cost of performing one projection is less than the inter-arrival time for input objects ( $C_\pi < 1/r_i$ ).
2. The cost of performing one projection is greater than the inter-arrival time for input objects ( $C_\pi > 1/r_i$ ).

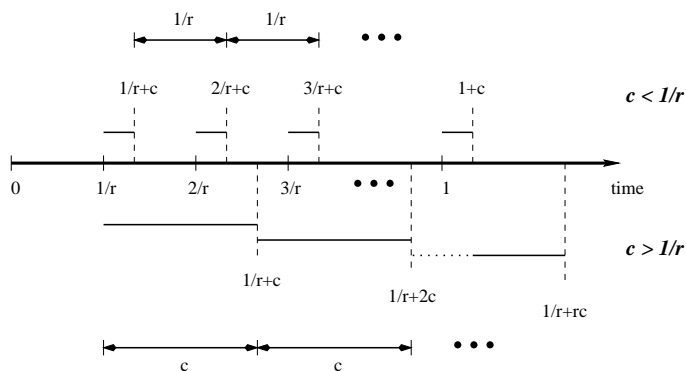


Figure 2: Relation between cost and inter-arrival rate. This applies to all cases, so  $c$  could be either the cost of a projection or a selection. Variable  $r$  models an input rate.

In this discussion, we incorporate the transmission cost  $T$  into the handling cost. If we want to distinguish between them, we can calculate the cost of handling an individual input as  $C_\pi + T$ . Figure 2 shows the consequences of each case. In the first case, the inter-transmission interval is equal to the inter-arrival interval, with the only difference being that the first output element appears after  $C_\pi$  time units, so  $r_o = r_i$ . In the second scenario, the situation is more complicated but we can figure out the inter-transmission interval from Figure 2 by observing that this interval has to be equal to the cost of handling one input. So, the transmission rate is the inverse of that, or  $r_o = \frac{1}{C_\pi}$ . We can safely assume, however, that the cost of making a projection is low, so for small values of  $C_\pi$ ,  $r_o = r_i$ .

### 3.2 Selections

Selections need to incorporate into the calculation the selectivity of the predicate under evaluation. Given the input rate, the number of input objects will be  $r_i$ . Assuming uniform distributions, the



number of objects appearing in the output will be  $\sigma \cdot r_i$  where  $\sigma$  is the predicate selectivity. The way to calculate the output rate is analogous to the calculation of the projection output rate with the only difference that we are using  $C_\sigma$  instead of  $C_\pi$ . The output rate will then be  $r_o = \sigma \cdot r_i$  if  $C_\sigma < 1/r_i$  and  $r_o = \frac{\sigma}{C_\sigma}$  if  $C_\sigma > 1/r_i$ . Again, it is safe to assume that  $C_\sigma < 1/r_i$ , so  $r_o = \sigma \cdot r_i$ .

### 3.3 Joins and Cartesian Products

Joins are more complex operations than projections and selections, in the sense that they operate over two distinct input sources. Before proceeding in our derivation, we must first be clear about what it is that we are trying to derive. Our model seeks to answer the following question: at any point  $t$  in the query execution, some left inputs and some right inputs may be arriving into the join. If  $r_l$  is the left input rate, and  $r_r$  is the right input rate, what is the output rate that will be observed for results generated by the arrival of these input tuples? Note that this rate may not in general be the observed rate at time  $t$ . In particular, if the system spends time processing these arrivals, the output tuples corresponding to these arrivals may not appear until some point in the future. The rate our model predicts will be the rate at that point in the future.

Asking “which tuples arrived at an arbitrary instant  $t$ ” does not make much sense, since time is continuous. So we instead ask about discrete time intervals, and then generalize from these discrete approximations to approximate the continuous case.

First, we need to compute the number of answer tuples that will be generated by the arrivals in some specified time interval. We will show that the number of result objects generated by the arrivals in one time unit, given we begin at time  $t$ , will be  $\sigma r_l r_r t^2$ . The logic behind this formula is the following: Assuming input rates  $r_l$  and  $r_r$  for the two input streams, at time  $t$  the number of elements read by the left stream will be  $r_l t$  while for the right stream  $r_r t$ . How many result tuples will be generated from these inputs? If we assume that we started at time zero, the number of result objects from these inputs will be  $\sigma r_l r_r$ . (This is just the number of tuples seen from the inputs times the selectivity of the join.)

Now consider the second time unit. At the end of this second time unit, the contribution from the left stream to the output will be  $\sigma r_l 2r_r$  (the selectivity times the number of objects read from the left stream in the time unit from  $t$  to  $2t$  times the total number of objects read from the right stream from time zero to  $2t$ ), while for the right stream it will be  $\sigma r_r 2r_l$ . Thus the

total number of output elements<sup>1</sup> generated by arrivals during the second time unit will then be  $2\sigma 2r_l r_r - \sigma r_l r_r$  (we need the deduction to account for duplicates). The total number of outputs produced by arrivals during the first two time units will then be the sum  $\sigma r_l r_r + 3\sigma r_l r_r$ . Using the same logic, the total number of outputs generated by arrivals during the first three time units will be  $\sigma r_l r_r + 3\sigma r_l r_r + 5\sigma r_l r_r$ . We can take these inductive steps to compute the total output elements for any time  $t$ .

We emphasize that we are talking about the elements generated due to arrivals during a time interval, not about the elements generated during the time interval. In particular, again, if the elements take longer to process than the inter-arrival rate, the outputs will be delayed, perhaps substantially, by outputs corresponding to tuples that arrived during previous time intervals.

Moving on to continuous time, we integrate these quantities over time. We need to solve the integral  $r_o = \sigma r_l r_r \int (2t - 1) dt$ . Solving the integral for time produces the total number of output objects produced by a join operation for the input that arrives at any time  $t$ , which is  $\sigma r_l r_r t(t - 1)$ .

Next, to calculate the rate that will be observed for these output objects, we need to compute how long it will take for them to be generated. Over a time interval  $t$  the join operator will have received  $r_l t$  inputs from the left stream and  $r_r t$  objects from the right stream. The time to handle each of the left inputs is by definition  $C_{L \bowtie}$  while for each of the right inputs the cost is  $C_{\bowtie R}$  (see also Table 1). Then the time to process these input tuples will be  $r_l t C_{L \bowtie} + r_r t C_{\bowtie R} = t \cdot (r_l C_{L \bowtie} + r_r C_{\bowtie R})$ . Substituting the above results into Equation 1 yields:

$$r_o = \frac{\sigma r_l r_r t(t - 1)}{t \cdot (r_l C_{L \bowtie} + r_r C_{\bowtie R})} = \frac{\sigma r_l r_r (t - 1)}{r_l C_{L \bowtie} + r_r C_{\bowtie R}} \approx \frac{\sigma r_l r_r t}{r_l C_{L \bowtie} + r_r C_{\bowtie R}} \quad (\text{Equation 2})$$

Finally, we note that we made the implicit assumption above that the time to process the tuples arriving during time  $t$ , which is  $t \cdot (r_l C_{L \bowtie} + r_r C_{\bowtie R})$ , is greater than  $t$ , which means that  $(r_l C_{L \bowtie} + r_r C_{\bowtie R}) > 1$ . If this is not the case, the denominator needs to be replaced by 1, since output tuples corresponding to a given input cannot be produced before the input itself arrives! The above holds for Cartesian products as well, with the only modification being that  $\sigma = 1$ .

From Equation 2 it is clear that the output rate of a join operation is time-dependent. The time dependence is actually more subtle than that formula indicates, because for some join operator implementations the constants  $C_{L \bowtie}$  and  $C_{\bowtie R}$  also depend on time (e.g., if the cost of handling an

---

<sup>1</sup>In what follows, we will use the terms *output object*, *output* and *output element* interchangeably to avoid repetition.

input depends upon the number previous inputs handled.)

Since the rate is a function of time there are optimization opportunities having to do with either maximizing the total output rate, and hence the time needed to produce the last result of the operation, or optimizing for specific time points of the operation. We will present such a framework in Section 4.

### 3.4 Cost Models for Specific Operator Implementations

The purpose of this section is to extract specific cost expressions for different join methods as a function of their input rates. The cost can be separated into two parts, namely the cost of handling an input from the left stream and the cost of handling a right stream input. We consider only non-blocking join algorithms, specifically the non-blocking nested loops and the symmetric hash join. The cost expressions we will extract, will be dependent on the number of input elements read up to the time point under consideration. The subsequent analysis assumes a join between streams  $R$  and  $S$ , with input rates  $r_R$  and  $r_S$  respectively. It also assumes the cost of each algorithm is further dependent on four cost variables, which Table 2 summarizes.

<i>Cost Variable</i>	<i>Meaning</i>
move	Cost of moving an input object from the input buffers into main memory for processing
comp	Cost of performing an in-memory comparison between two different objects
hash	Cost of hashing an object into a hash table
probe	Cost of probing a hash table in a lookup operation and producing the output

Table 2: Notation for cost formulas

#### 3.4.1 Nested Loops Join

The nested-loops join algorithm traditionally needs all of the inner input present to execute properly. The outer may be streaming, since late arrivals can be thought of as additional executions of the inner loop. If the inner stream is not bound at execution beginning, however, the algorithm has to be modified. A straight-forward non-blocking extension would be to insert all newly arrived inputs from the inner stream into a set, and whenever an inner loop ends, a second inner loop is executed

for all late arrivals. This fits into Niagara’s *partial results* architecture [12].

The algorithm needs to loop over all inputs of the outer stream moving them into memory, and for each input compare it against all inputs of the inner stream. The cost for handling one left input arrival is then:  $C_{L\bowtie} = \text{move} + |S|_t \cdot \text{comp} = \text{move} + r_{St} \cdot \text{comp}$ . For right input arrivals, a loop over all the left inputs has to be initiated. The cost is then:  $C_{\bowtie R} = \text{move} + |R|_t \cdot \text{comp} = \text{move} + r_{Rt} \cdot \text{comp}$ .

### 3.4.2 Symmetric Hash Join

Symmetric hash join is by definition non-blocking. It keeps two hash tables in memory and for each arrival it hashes it into the corresponding stream’s hash table, while at the same time using it to probe the other stream’s hash table. The cost is then the same for both streams of the operation and is equal to  $C_{L\bowtie} = C_{\bowtie R} = \text{move} + \text{hash} + \text{probe}$ : first move the input element into main memory, then hash it into the appropriate hash table and finally use it to probe the other stream’s hash table.

Table 3 summarizes the arrival cost formulas for the algorithms we have considered. Non-blocking nested loops has a time-dependent aspect to its cost, so, as time progresses, the cost increases. Symmetric hash join, on the other hand, has a constant cost to handle its inputs. In what follows and for the purposes of this study we will only concentrate on symmetric algorithms since they are the ones that lead to the least complex formulas.

<i>Algorithm</i>	<i>Left arrival cost (<math>C_{L\bowtie}</math>)</i>	<i>Right arrival cost (<math>C_{\bowtie R}</math>)</i>
Nested loops	$\text{move} + r_{St} \cdot \text{comp}$	$\text{move} + r_{Rt} \cdot \text{comp}$
Symmetric hash join	$\text{move} + \text{hash} + \text{probe}$	$\text{move} + \text{hash} + \text{probe}$

Table 3: Cost formulas for the different join algorithms

## 4 Using Estimates to Optimize Queries

In this section we first describe the general problem that arises when considering using our rate estimates to optimize queries. Next, we discuss two simple heuristics as examples of how the general problem might be simplified in practice.

## 4.1 General Framework for Rate-Based Optimization

Section 3 shows how to compute the output rate of an operator as a function of the rates of its inputs. In the case of join operators, the output rate is time-dependent. When asked to evaluate a plan, we can combine its various operations to come up with a function of time that models its output rate. Given the output rate of a plan  $r(t)$  then the number of results the plan will produce at any point in time  $t_O$  is given by the integral of the rate over time:

$$\# \text{ Outputs} = \int_0^{t_0} r(t)dt \quad (\text{Equation 3})$$

The integral of Equation 3 provides the general framework for rate-based optimization. The problem then becomes: given a collections of plans  $P_i$  and their output rate  $r_{P_i}(t)$  as a function of time, how do we decide which plan to employ? There are two important optimization opportunities.

**Optimize for a specific time point in the execution process.** The integral of Equation 3 can be treated as an equation. Given a collection of plans and a time point  $t_0$ , by solving the integral we can estimate how many output elements the plan will have produced by that time. We can then pick the plan with the highest number of output elements produced. The question we are asking is “which plan will produce the most results by time  $t_0$ ?”

**Optimize for output production size.** In this case we reverse the procedure. Given an output size  $N$  we want to identify the plan which will reach the specified number of results at the least time. We then pick the plan with the minimum such time. In this situation we are asking: “which plan is the first one to reach  $N$  results?” Notice that  $N$  can either be the total number of results, or the first result. The framework incorporates both of these notions.

The optimization opportunities we listed entail computing the solution to an integral, which is inefficient for practical optimization purposes. Optimizers have to manipulate a huge search space when enumerating possible plans, but none of the optimizers evaluate the whole search space. Instead, they employ heuristics that will limit the search space into the most promising enumerations. It is logical that we should follow the same tactic in trying to approximate the integral of Equation 3. Identifying ways to efficiently approximate the integral provides interesting research directions. In the next section we will propose two different kinds of approximation heuristics: *local rate maximization* and *local time minimization*.

## 4.2 Examples of Heuristics

Devising efficiently applicable heuristics that generate good plans is a rich area for future research. In this section we give two examples. Our intent is to illustrate the kinds of heuristics are possible (rather than to claim that these heuristics will perform well in practice.) Both of the heuristics we present aim at locally optimizing the plan under the premises that better local performance entails better overall performance of the execution plan. We will concentrate on identifying plans with maximum rate over time, and plans that reach a specified number of results as soon as possible. For the first case, we identify a performance estimate for each join operation of the plan and we combine such estimates to come up with an overall performance estimate. In the second case, we use the same performance estimate to locally minimize the time needed to produce the estimated number of results required at each join for a total number of results to be reached. Both of these heuristics can be used as performance metrics for existing optimizers.

### 4.2.1 Local Rate Maximization

A local rate maximization framework builds on a simple heuristic: the plan with the maximum overall rate is the one that will have the maximum constituent rates. What we propose to do in a local rate maximization framework, is to organize the plan in such a way that our rate estimates for each point of the plan are maximized. For join operations involving two primary sources for the query (i.e., joins that none of their inputs is the output of another join) we can determine the slope of the rate function, according to Equation 2: it is  $\frac{\sigma r_l r_r}{r_l C_{L \bowtie} + r_r C_{\bowtie R}}$ . We then use this as an estimate of how fast the join is and treat it as an input rate for subsequent join operations. This process will yield an estimate for each plan we consider. We choose the plan that has the maximum such overall estimate. Figure 3 shows how we calculate the estimate for a plan, by using only the primary rates and how we combine the rates to generate an overall heuristic estimate of the plan's performance.

It is easy to incorporate this performance estimate in any optimizer. For instance, a dynamic programming optimizer dealing with streaming sources, instead of calculating the traditional cost for a join operation, it uses the ratio  $\frac{\sigma r_l r_r}{r_l C_{L \bowtie} + r_r C_{\bowtie R}}$  as a performance indicator, proceeding as in Figure 3. Notice, that the estimate provides room to incorporate any other CPU and I/O metric by inserting it into the calculation through  $C_{L \bowtie}$  and  $C_{\bowtie R}$ .

As an example, suppose we are faced with three possible plans  $(A \bowtie B) \bowtie C$ ,  $(A \bowtie C) \bowtie B$  and  $(B \bowtie C) \bowtie A$ , for which there are join predicates between  $A$  and  $B$  with selectivity  $\sigma_{AB}$  and

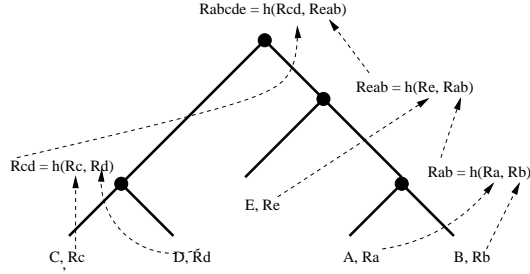


Figure 3: Local rate maximization process

$B$  and  $C$  with selectivity  $\sigma_{BC}$ . Figure 4 depicts the plans, while Table 4 shows the output rate as a function of time<sup>2</sup> and our estimate of the rate and hence the plan's performance. Our estimates closely approximate the performance of each plan, rate-wise. For any given time point, the factor which will differentiate one plan from an other is our estimate. In this case we will choose the plan that has the maximum value of our estimate. Choosing the plan with the highest value for the estimate entails choosing the plan with the highest output rate.

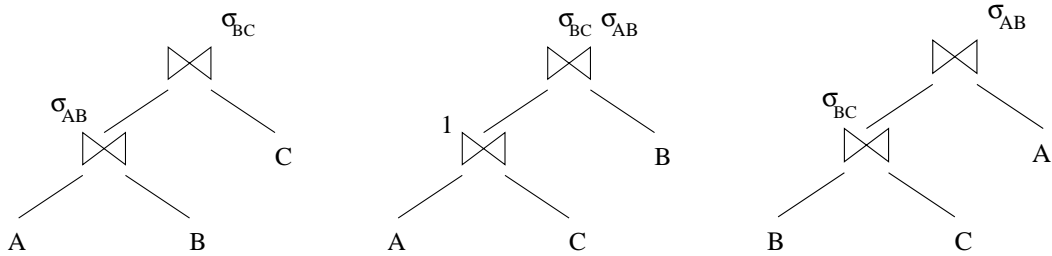


Figure 4: Three different execution plans for  $A \bowtie B \bowtie C$ . Each join operation is annotated with the estimated selectivity.

#### 4.2.2 Local Time Minimization

The local rate maximization heuristic identified an estimate of how fast a join operation produces results in general. We can use this estimate as a further estimation of how fast a join operation can produce a specific number of results, when we wish to identify the plan that will produce that number of results as soon as possible. To devise this estimate we are based on a simple observation.

<sup>2</sup>For simplicity of the table's formulas, we assume that  $C_{L \bowtie} = C_{\bowtie R} = c$ .

<i>Plan</i>	<i>Output rate as a function of time</i>	<i>Performance estimate</i>
$(A \bowtie B) \bowtie C$	$\frac{\sigma_{AB}\sigma_{BC}r_Ar_Br_Ct^2}{c(\sigma_{AB}r_Ar_Bt+r_Cc(r_A+r_B))}$	$\frac{\sigma_{AB}\sigma_{BC}r_Ar_Br_C}{c(\sigma_{AB}r_Ar_B+r_Cc(r_A+r_B))}$
$(A \bowtie C) \bowtie B$	$\frac{\sigma_{AB}\sigma_{BC}r_Ar_Br_Ct^2}{c(\sigma_{AB}\sigma_{BC}r_Ar_Ct+r_Bc(r_A+r_C))}$	$\frac{\sigma_{AB}\sigma_{BC}r_Ar_Br_C}{c(\sigma_{AB}\sigma_{BC}r_Ar_C+r_Bc(r_A+r_C))}$
$(B \bowtie C) \bowtie A$	$\frac{\sigma_{AB}\sigma_{BC}r_Ar_Br_Ct^2}{c(\sigma_{BC}r_Br_Ct+r_Ac(r_B+r_C))}$	$\frac{\sigma_{AB}\sigma_{BC}r_Ar_Br_C}{c(\sigma_{BC}r_Br_C+r_Ac(r_B+r_C))}$

Table 4: Output rates and rate-based estimates of plan performance for different execution plans

The formula which connects output rate  $r$ , time  $t$  and number of outputs produced  $n$  is  $n = rt$ . If we have an estimate of the results we need to produce and an estimate of the rate at which we can produce them, then an estimate of how soon we can generate them is the number of results divided by time, or in the previous formula,  $\frac{n}{r}$

We can incorporate this strategy into a more general optimization framework. Suppose we are again facing the operation  $A \bowtie B \bowtie C$ , as was the case in Figure 4. We wish to optimize for the time needed to reach 25% of the total output. We can tackle the problem by decomposing it into a number of equivalent sub-problems. To do so, we need to *push-down* the number of elements each input to the final join should produce for the desired number of outputs to be produced. The way to do so is simple. We know that the number of overall outputs we optimize for is equal to  $0.25 \cdot \sigma_{AB} \cdot \sigma_{BC} |A| |B| |C|$ . Taking the join sequence  $A \bowtie (B \bowtie C)$  as an example, in order to reach our goal we approximately need to read  $\sqrt{0.25 \cdot \sigma_{AB} \cdot \sigma_{BC}} |A|$  inputs from  $A$  and  $\sqrt{0.25 \cdot \sigma_{AB} \cdot \sigma_{BC}} |B| |C|$  from  $B \bowtie C$ . Using this *divide-and-conquer* strategy we can take care of arbitrarily complex join strategies. Figure 5 shows how we distribute the estimated number of required results between the various join operators of the execution plan. We finally transform the problem into a minimization/maximization one: the plan which will reach the desired number of outputs the fastest, is the one for which the latest time its constituents joins will reach their respective number of outputs is the smallest. The way we use the performance estimate is the following: We want an indication of how much time each sub-problem needs to be completed. An estimate of this time is the predicted number of outputs for the sub-problem, divided by the rate estimate.

We can provide the solution in a more formal fashion: Assume we have a recursive definition of possible join execution trees. This definition consists of a set of join strategies, with each join strategy annotated with the desired number of outputs to be reached as fast as possible,



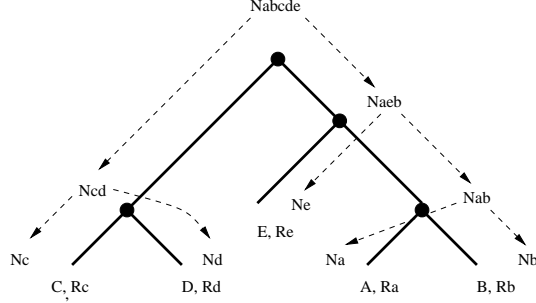


Figure 5: Required output size distribution for local time minimization

the value of its metric, and the fastest time for this number to be reached. For instance, given join strategy  $A \bowtie (B \bowtie B)$ , the information about it is encapsulated into  $\{(A, n_A, r_A, t_A), (B \bowtie B, n_{B \bowtie B}, r_{B \bowtie B}, t_{B \bowtie B})\}$ . The notation here is that for each point of interest  $P$  (which can be a source stream, or the output of a join), we encapsulate it in the structure  $(P, n_P, r_P, t_P)$  for which  $n_P$  is the number of results we wish to reach,  $r_P$  is the value of our rate estimate and  $t_P$  is the estimation of time we need to reach that number of results.  $B \bowtie C$  can be further decomposed into  $\{(B, n_B, r_B, t_B), (C, n_C, r_C, t_C)\}$ . Notice, that it is up to us to decide how far down the execution plan we wish to descend. For instance, we may not want to descend down to stream level, but rather stop at the joins of pairs of streams (i.e., not decompose  $B \bowtie C$  in  $B$  and  $C$  since we have an estimate for the rate of the join operation). The notation here is that for an input  $B$ , the fastest time to generate the desired number of  $n_B$  outputs is  $t_B$ . We can encapsulate such a notation by using logical rules as:

$$\begin{aligned}
 \text{Info} &\leftarrow (\text{Stream}, n_S, r_S, t_S) \\
 \text{Tree} &\leftarrow \{\text{Info}\} \mid (\{\text{Tree}\} \cup \text{Info}) \\
 \text{Stream} &\leftarrow S \mid \text{Tree}
 \end{aligned}$$

The input to our decision algorithm is a set of such structures. The solution to the problem then, involves three steps: (i) For each strategy in the set, find the maximum time needed to complete it, recursively going into the *Tree* structures (ii) Find the strategy with the minimum such time (iii) Choose the join strategy that corresponds to the minimum time. Algorithm 1 presents a simple program to perform the calculation over these structures, in which *min* returns the best execution tree while *max* returns the maximum execution time within a single tree.

---

**Algorithm 1** A simple decision algorithm to optimize for time needed to reach a specific part of the output

---

```

max ( $\emptyset$ , 0).
max (Info  $\cup$  Tree,  $M$ )  $\leftarrow$  Info = (S,  $n_S$ ,  $r_S$ ,  $t_S$ ),
    max(Tree,  $M_t$ ),
     $t_S = n_S/m_S$ ,
    ( $M_t > t_S$  ?  $M = M_t$  :  $M = t_S$ ).
min ( $\emptyset$ ,  $\perp$ ,  $\infty$ ).
min (Tree  $\cup$  Forest, BestTree,  $C_b$ )  $\leftarrow$  max(Tree,  $C_t$ ),
    min(Forest, BestInForest,  $C_f$ ),
    ( $C_t > C_f$  ? BestTree = Tree,  $C_b = C_t$  : BestTree = BestInForest,  $C_b = C_f$ )

```

---

## 5 Validation of the cost model

In this section we provide experimental results to prove the validity of the cost model we have proposed. We focus our attention on a specific equi-join query which we will use throughout the experimental section. This query involves five streaming sources, each with its own rate. We organized the plan for that query in three different ways and evaluated each one against our analytical framework, and against the metrics of plan performance we have devised.

### 5.1 Experimental Setup

The dataset we used was a synthetically generated XML dataset which represented a database of student information across various departments. Each department denoted a streaming input source to the query and it consisted of a number of graduate and undergraduate students. Figure 6 depicts the graph structure of each department file. Each file contains information about one department and each department contains a list of students. The purpose of the query was to find undergraduate students who were enrolled in all departments participating in the query. Assigning names  $A$ ,  $B$ ,  $C$ ,  $D$ , and  $E$  for the department, what we wanted was to compute the outcome of the query  $A.Undergraduate-Student = B.Undergraduate-Student = C.Undergraduate-Student = D.Undergraduate-Student = E.Undergraduate-Student$ . This setup allowed us to shape the plan in any way we wanted by choosing the appropriate equi-join predicates. Table 5 presents the rate and the size of each input stream. The cardinality of each student’s appearance in each department was fixed to a percentage of the total number of students appearing in the file, so each equi-join predicate had the same estimated selectivity. For our experiments, we kept this selectivity at 15%.

All experiments were performed using the current version of the Niagara Query Engine [3, 8, 12]

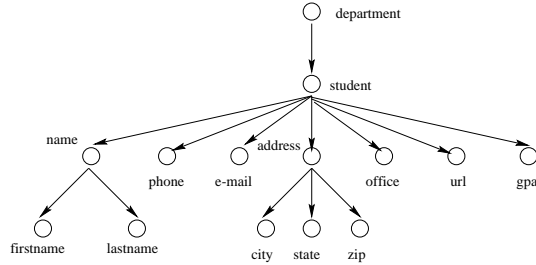


Figure 6: The XML schema used in our experiments

developed at the University of Wisconsin-Madison. The hardware setup involved a Pentium-III processor operating on 800 MHz with 256 MB of physical memory. The data were read from flat XML files, and the parsing startup time was subtracted from the results we report. We simulated network traffic by inserting random delays between element reads. The arrivals were modeled as a Poisson process, as is often the case for network traffic [2], with a fixed rate, equal to the stream’s rate. This meant that the delays followed an exponential distribution with a mean equal to the inverse of the stream’s rate. Throughout the experimentation we employed symmetric hash join as the evaluation algorithm.

<i>Stream</i>	A	B	C	D	E
<i>Rate (<math>\frac{ob}{sec}</math>)</i>	20	30	10	10	100
<i>Size (# objects)</i>	30	40	30	20	90

Table 5: Stream information for our experiments

We organized the query in three different plans, as Figure 7 presents. One of the plans is a left-deep plan, the other two are bushy plans. We have annotated each join operator of the plans with the join predicate it evaluates, while a thicker line represents a faster stream. All plans employ symmetric hash join as the evaluation algorithm. In what follows, we will refer to the plan of Figure 7(a) as *Deep Plan*, to the one of Figure 7(b) as *Bushy Plan #1*, while to the plan of Figure 7(c) as *Bushy Plan #2*.

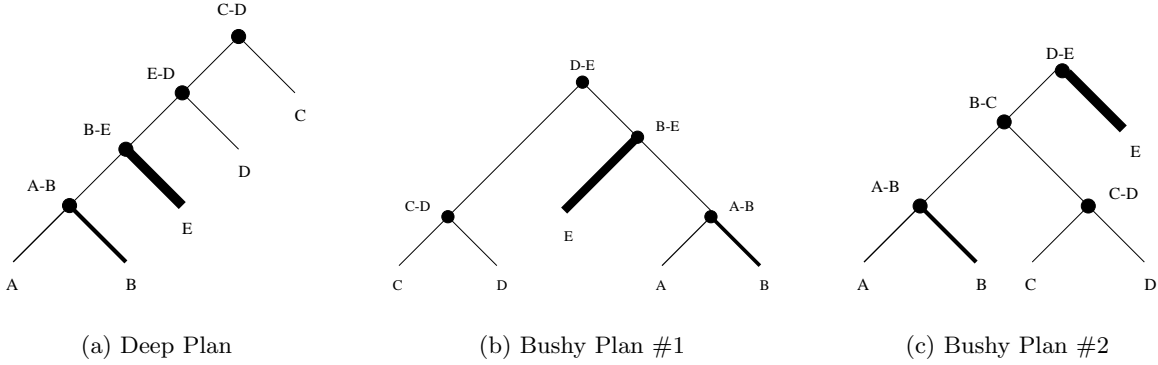


Figure 7: The three plans of our experimentation. Each plan is annotated with the join predicates it computes. A thicker line denotes a faster stream.

## 5.2 The Analytical Framework

Given the input rates of Table 5 we used the analytical framework of Section 3.3 to graph the expected performance of the plans. We defined the function  $R(\sigma, r_1, r_2, t) = \frac{\sigma r_1 r_2 t}{r_1 + r_2}$  to model the output rate of a join operation and used this function to compose the overall output rate of each execution plan<sup>3</sup> We then plotted the output rate as a function of time. Notice, that since we are only concentrating on symmetric algorithms, we do not treat the denominator as a function of time (as would have been the case if we were using a non-symmetric operator like nested-loops join). Figure 8 shows our prediction. At this point we are only interested in the shape of the curve and not the actual output size and time values. That is because we are only interested in verifying whether the plans exhibit the same behavior as the one we predict.

Looking at Figure 8 when we execute the three plans we expect Bushy Plan #2 to be the best for the initial stages of query execution, with Deep Plan being second in terms of performance. Both of these plans should have comparable performance throughout query execution. In the initial stages, Deep Plan is better than Bushy Plan #1 as well. As time progresses however, the situation should change. The second bushy plan should start performing better, until it bypasses both plans in the process.

<sup>3</sup>For instance, for the plan of Figure 7(a), the corresponding composition is  $R_t = R(\sigma_{CD}, R(\sigma_{ED}, R(\sigma_{BE}, r(\sigma_{AB}, r_A, r_B, t), r_E, t), r_D, t), r_E, t)$ .

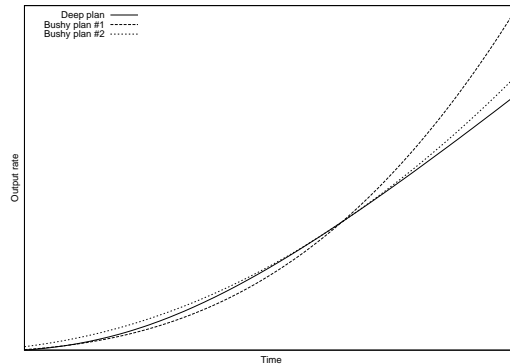


Figure 8: Analytical modeling of the plans depicted in Figure 7

### 5.3 Experimental Validation of the Analytical Framework

The next step after having the analytical model, is to execute the plans on our test-bed. Figure 9 presents the results. The plans exhibit the behavior we predict. The only difference between the two curves is that the “spread” between the different plans, is not as wide as the analytical model predicts. This could be, however, due to the fact that we are not using large streams as our input. The analytical model predicts that the performance between the plans will widen further as time progresses. Since we have finite inputs, we cannot validate this assumption.

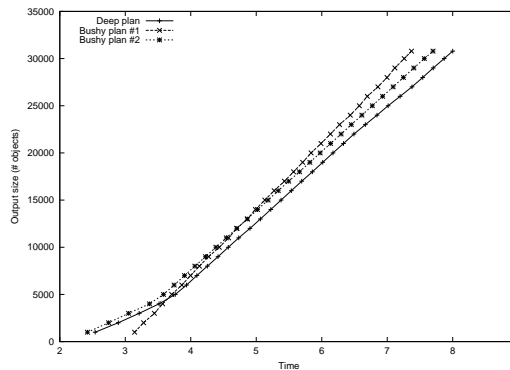


Figure 9: Execution of Figure 7’s plans on the Niagara Query Engine

We can, however, validate the fact that as the model predicts, in the initial stages of query execution, Bushy Plan #1 dominates, while for the same initial stages Deep Plan is better than Bushy Plan #2. Eventually, Bushy Plan #2 outperforms both other plans again in accordance to

the analytical model. So we can draw the conclusion that, at least for the query we experiment with, our model closely approximates the plans' behavior.

## 5.4 Heuristics Validation

The purpose of this section is to validate the heuristics we proposed in Section 4.2. If our heuristics are valid, they should predict that Bushy Plan #2 is the one with the higher performance among the three candidate plans. To verify this, we have to revisit Equation 2 which says that the output rate of a join operation is  $\frac{\sigma_l r_r t}{r_l C_{L\bowtie} + r_r C_{R\bowtie}}$ . For the purposes of this study we are only concerned with symmetric algorithms, so we can assume that  $C_{L\bowtie} = C_{R\bowtie} = c$  for some constant  $c$  (in the case of symmetric hash join, this constant is equal to `move + hash + probe`). We assume this estimates how fast a plan is. In Section 4.2.1 we presented a framework of measuring the performance of plans based on the estimate, by treating it as an actual rate for subsequent joins. Based on this observation, Table 6 presents the estimate's value for the join operations of the three plans we are considering<sup>4</sup>. The notation we used for the operations is the same as the one in Figure 7.

<i>Deep Plan</i>		<i>Bushy Plan #1</i>		<i>Bushy Plan #2</i>	
$A - B$	$\frac{1.8}{c}$	$A - B$	$\frac{1.8}{c}$	$A - B$	$\frac{1.8}{c}$
$B - E$	$\frac{1.35}{5c^2 + 0.09c}$	$C - D$	$\frac{0.75}{c}$	$C - D$	$\frac{0.75}{c}$
$E - D$	$\frac{2.025}{1000c^3 + 0.9c^2 + 1.35c}$	$B - E$	$\frac{27}{100c^2 + 1.8c}$	$B - C$	$\frac{0.2025}{2.55c^2}$
$C - D$	$\frac{3.0375}{500c^4 + 9c^3 + 13.5c^2 + 0.5c}$	$D - E$	$\frac{3.0375}{75c^3 + 28.35c^2}$	$D - E$	$\frac{3.0375}{255c^2 + 0.2025}$

Table 6: Performance estimates for the plans of Figure 7

The question now becomes what the value for  $c$  is. The only knowledge we have is that in order to use the formulas we used, the inequality  $c(r_l + r_r) > 1$  must hold (see Section 3.3) where  $r_l$  and  $r_r$  are the rates of the left and right input streams to a join operation. We can apply this formula to all joins using primary sources as inputs (i.e., one of  $A, B, C, D$  or  $E$ ). Looking at the plans, there are only two cases<sup>5</sup> :  $c(r_A + r_B) > 1 \Rightarrow c > 1/50$  and  $c(r_D + r_E) > 1 \Rightarrow c > 1/20$ . What we can have from both of these is that  $c > 1/20$ . Putting  $1/20$  into the formulas (or, in fact, any

<sup>4</sup>Due to space limitations we do not show the calculation.

<sup>5</sup>For the interested reader, using the metric as a rate estimate on the rest of the join operations, yields in all cases that  $c$  must be greater than some negative number, which obviously holds.

number between 1/20 and 3) yields that the denominator of what we predicted would be the fastest plan is the smallest, so our prediction is indeed correct. But what is the meaning of the value we chose for  $c$ ? Going back to Section 3.1 we saw that it can be treated as an estimate of how fast our system can produce results: in data networks terminology, we can regard it as the system's *mean service time* [2]. The value we gave it means that given our we should be capable of handling at least 20 inputs per unit time, something that is not an invalid assumption.

## 5.5 Comparison Between the Classic Cost Model and the Proposed One

The purpose of this section is to present the superiority of the proposed cost model to the classic one [11], that did not have any notion of rates. To calculate the cost of the operation according to the old cost model, we will use the cost formulas we extracted in Section 3.4. Notice that we are using symmetric hash join as the evaluation algorithm. When we are dealing with primary sources as inputs the cost of the algorithm for  $A \bowtie B$  will be  $|A|(\text{move} + \text{hash} + \text{probe}) + |B|(\text{move} + \text{hash} + \text{probe})$ , where again the notation is that  $|A|$  denotes the number of elements in stream  $A$ . We call the sum  $\text{move} + \text{hash} + \text{probe}$  as  $\text{mhp}$ . If we are not dealing with primary sources as inputs, we can deduct the cost of moving an element to the main memory. So the cost for  $A \bowtie B$  if both  $A$  and  $B$  have already been read becomes  $|A|(\text{hash} + \text{probe}) + |B|(\text{hash} + \text{probe})$ . We call the sum  $\text{hash} + \text{probe}$  as  $\text{hp}$ . The output size of a join is the selectivity of the predicate, times the product of the input sizes. So, the output of  $A \bowtie B$  is  $\sigma|A||B|$ . Since all predicates have the same selectivity, we will name this  $\sigma$  across the subsequent calculations. Having all these in mind we can proceed to calculate the cost of each plan according to the classic cost model. Table 7 presents the results.

<i>Plan</i>	<i>Expression</i>
Deep Plan	$ A \text{mhp} +  B \text{mhp} + \sigma A  B \text{hp} +  E \text{mhp} + \sigma^2 A  B  E \text{hp} +  D \text{mhp} + \sigma^3 A  B  E  D \text{hp} +  C \text{mhp}$
Bushy Plan #1	$ A \text{mhp} +  B \text{mhp} +  C \text{mhp} +  D \text{mhp} + \sigma A  B \text{hp} +  E \text{mhp} + \sigma C  D \text{hp} + \sigma^2 A  B  E \text{hp}$
Bushy Plan #2	$ A \text{mhp} +  B \text{mhp} +  C \text{mhp} +  D \text{mhp} + \sigma A  B \text{hp} + \sigma C  D \text{hp} + \sigma^3 A  B  C  D \text{hp} +  E \text{mhp}$

Table 7: Evaluation of the plans of Figure 7 according to the classic cost model

Looking at the cost expressions of Table 7 one sees that the determining factors are  $\sigma^2|A||B||E| + \sigma^3|A||B||E||D|$  for Deep Plan,  $\sigma|C||D| + \sigma^2|A||B||E|$  for Bushy Plan #1 and  $\sigma|C||D| + \sigma^3|A||B||C||D|$

for Bushy Plan #2, as the rest of the constituents are factored out. Putting numbers into the formulas yields that the cost of Deep Plan is 9720 hash and probe operations while the cost of the two bushy plans is 2520 operations for both. What we see here is that the classic cost model without taking into account the streaming rates of the inputs cannot distinguish between the two bushy plans. Our model however can distinguish between the two, knowing that as time progresses Bushy Plan #1 will outperform Bushy Plan #2. We are not claiming that our cost model is better than the classic model. We are claiming though that it makes sense to employ the new cost model since it can provide better decisions when faced with streaming sources.

## 6 Conclusions and Future Work

The contribution of this paper is a new cost model concerning streaming data. We looked into the nature of streaming behavior for network-resident inputs, identifying all those factors that transform the problem into a new research topic. We identified how streaming inputs can be incorporated into a new cost model and how this model can be used to produce faster execution plans. We identified metrics of plan performance and we showed how this metrics can be used to estimate the rate at which an execution plan with stream sources as inputs generates results. We validated our model with experiments run in our environment, using the current version of the Niagara Query Engine. The results prove the validity of this cost model.

The future work we plan to undertake focuses on further exploring the effects of streaming data on plan execution strategies. There are a few possible directions in this area, with the first one being to try and introduce data distributions into the cost model. The model so far assumes uniformity for the data values. It is our intuition, that if data are not uniformly distributed, selectivity is itself a function of time. This can be incorporated into the cost model.

Identifying good heuristics of approximating the integral of Equation 3 is a second possible research topic. We already identified some heuristics in Section 4.2 but coming up and further evaluating heuristics that let us approximate the values of the integral and provide ways to efficiently evaluate plans is of much interest.

Dynamic re-optimization is another possible direction. Since the output rate is time dependent, we can devise an optimizer that does not generate only one possible plan, but rather a family of plans and specific time points in the execution time period in which the plans should change. Such a strategy means always operating at the highest possible rate. For instance in Figure 8's graph,



there are estimated times for which a deep plan produces a higher output rate than that of a bushy plan. A “clever” optimization would be to employ the deep plan for as long as it is predicted to be faster, switching to the bushy plan at the estimated intersection time point.

Lastly, since dynamic programming algorithms generally generate deep plans, we can devise an algorithm in the style of dynamic programming that, although not exhaustive of the search space, it can generate bushy plans if the new model predicts they have better performance.

## References

- [1] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously Adaptive Query Processing. In Weidong Chen, Jeffrey F. Naughton, and Philip A. Bernstein, editors, *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, May 16-18, 2000, Dallas, Texas, USA*, volume 29, pages 261–272. ACM, 2000.
- [2] Dimitri Bertsekas and Robert Gallager. *Data Networks*. Prentice Hall, 2 edition, 1991.
- [3] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagara-CQ: A Scalable Continuous Query System for Internet Databases. In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data, Dallas, Texas, May 2000*, pages 379–390, 2000.
- [4] Minos N. Garofalakis and Yannis E. Ioannidis. Parallel query scheduling and optimization with time- and space-shared resources. In Matthias Jarke, Michael J. Carey, Klaus R. Dittrich, Frederick H. Lochovsky, Pericles Loucopoulos, and Manfred A. Jeusfeld, editors, *VLDB’97, Proceedings of 23rd International Conference on Very Large Data Bases, August 25-29, 1997, Athens, Greece*, pages 296–305. Morgan Kaufmann, 1997.
- [5] W. Hong and M. Stonebraker. Optimization of Parallel Query Execution Plans in XPRS. *Distributed and Parallel Databases*, 1(1):9–32, 1993.
- [6] Navin Kabra and David J. DeWitt. Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 106–117. ACM Press, 1998.
- [7] Chiang Lee, Chih-Horng Ke, J.-B. Chang, and Yaw-Huei Chen. Minimization of resource consumption for multidatabase query optimization. In *Proceedings of the 3rd IFCIS International Conference on Cooperative Information Systems, New York City, New York, USA, August 20-22, 1998, Sponsored by IFCIS, The Intn’l Foundation on Cooperative Information Systems*, pages 241–250. IEEE-CS Press, 1998.
- [8] Jeffrey Naughton, David DeWitt, and David Maier et. al. The Niagara Internet Query System. Submitted for Publication.

- [9] Kenneth W. Ng, Zhenghao Wang, Richard R. Muntz, and Silvia Nittel. Dynamic Query Re-Optimization. In *Proceedings of the 11th International Conference on Scientific and Statistical Database Management*, pages 264–273, 1999.
- [10] G. Schumacher. GEI’s Experience with Britton-Lee’s IDM. In *IWDM*, pages 233–241, 1983.
- [11] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access Path Selection in a Relational Database Management System. In Philip A. Bernstein, editor, *Proceedings of the 1979 ACM SIGMOD International Conference on Management of Data, Boston, Massachusetts, May 30 - June 1*, pages 23–34. ACM, 1979.
- [12] Jayavel Shanmugasundaram, Kristin Tufte, David J. DeWitt, Jeffrey F. Naughton, and David Maier. Architecting a Network Query Engine for Producing Partial Results. In *WebDB 2000*, Dallas, TX, May 2000.
- [13] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A Wide-Area Distributed Database System. *VLDB Journal*, 5(1):48–63, 1996.
- [14] Tolga Urhan and Michael J. Franklin. XJoin: A Reactively-Scheduled Pipelined Join Operator. *IEEE Data Engineering Bulletin*, 23(2):27–33, June 2000.
- [15] Tolga Urhan, Michael J. Franklin, and Laurent Amsaleg. Cost Based Query Scrambling for Initial Delays. In Laura M. Haas and Ashutosh Tiwary, editors, *SIGMOD 1998, Proceedings ACM SIGMOD International Conference on Management of Data, June 2-4, 1998, Seattle, Washington, USA*, pages 130–141. ACM Press, 1998.
- [16] A. N. Wilschut and P. M. G. Apers. Pipelining in Query Execution. In *Conference on Databases, Parallel Architectures and their Applications*, Miami, 1991.