Rate-Based Query Optimization for Streaming Information Sources

Stratis D. Viglas   Jeffrey F. Naughton

Department of Computer Sciences
University of Wisconsin-Madison
1210 W Dayton St.
Madison, WI, 53706, USA

{stratis, naughton}@cs.wisc.edu

ABSTRACT

Relational query optimizers have traditionally relied upon table cardinalities when estimating the cost of the query plans they consider. While this approach has been and continues to be successful, the advent of the Internet and the need to execute queries over streaming sources requires a different approach, since for streaming inputs the cardinality may not be known or may not even be knowable (as is the case for an unbounded stream.) In view of this, we propose shifting from a cardinality-based approach to a rate-based approach, and give an optimization framework that aims at maximizing the output rate of query evaluation plans. This approach can be applied to cases where the cardinality-based approach cannot be used. It may also be useful for cases where cardinalities are known, because by focusing on rates we are able not only to optimize the time at which the last result tuple appears, but also to optimize for the number of answers computed at any specified time after the query evaluation commences. We present a preliminary validation of our rate-based optimization framework on a prototype XML query engine, though it is generic enough to be used in other database contexts. The results show that rate-based optimization is feasible and can indeed yield correct decisions.

1   INTRODUCTION

For the past twenty years, query optimization has been an intensively studied area of database system research. Most modern optimizers are cost-based in that they decide between execution plans by minimizing the estimated cost of evaluating the query. A fundamental technique used in cost estimation is cardinality estimation – optimizers take as input the cardinalities of tables at the leaves of a query tree, and then use selectivities of operators in the tree to estimate the cardinality of the input to operators further up in the tree. To convert cardinalities to costs, optimizers use functions that estimate the cost per tuple of each operator. While this approach is not perfect, it is very effective in most traditional DBMS applications. However, as we move to the Internet domain, this approach, in its current form, may not even apply. The reason for this is that if the leaves of the query tree correspond to incoming network streams, not only is their cardinality often not known, in some cases it may not even be well defined (e.g., in the case of infinite streams.)

To allow the optimization of queries in the presence of streaming data, a new approach is needed. In this paper we propose *rate-based optimization* for such applications. As we will describe, in rate-based optimization the fundamental statistics used are estimates of the rates of the streams in the query evaluation tree (rather than the sizes of intermediate results.) Cardinality estimates, when present, aid us in making better decisions but they are not required.

As a concrete example motivating our approach, consider a situation in which we have two selection operators σ_1 and σ_2 that have comparable selectivities but are very different in how quickly they can transmit an input to the output. To put numbers to the example, assume each predicate's selectivity is 0.1; inputs to the system arrive at a rate of 500 per second; and σ_1 is able to process 50 inputs per second, while σ_2 is so fast that for all practical purposes it can process data as fast as it receives it. Finally, assume we wish to run a query that applies the two selections in series to an infinite stream. Figure 1 presents a trace of the two possible execution plans.
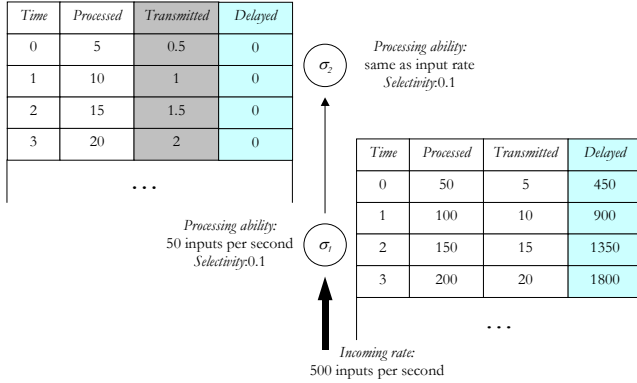
The interesting parameter to note in Figure 1 is the *Transmitted* column, which represents the output rate. Even in this simple a case, we obtain a ten-fold performance improvement in terms of output rate with a simple switch of their execution orders. Moreover, the first execution plan creates an ever-increasing backlog of delayed inputs at σ_1, while the second plan does not delay its inputs.

A traditional cardinality-based optimizer would not be in a position to distinguish one plan from the other. This can be seen as follows. Using a standard cardinality-based model, the cost of the first plan is $|I| \cdot c_1 + |I| \cdot f_1 \cdot c_2$, where $|I|$ denotes the input's size, f_i the predicate's selectivity factor and c_i the cost per tuple for predicate i. Similarly, the second plan's cost is $|I| \cdot c_2 + |I| \cdot f_2 \cdot c_1$. Unfortunately, since the size of the input is infinite, the cost for each plan is infinite, both will be of equal cost, and the optimizer will have no way to choose between them.
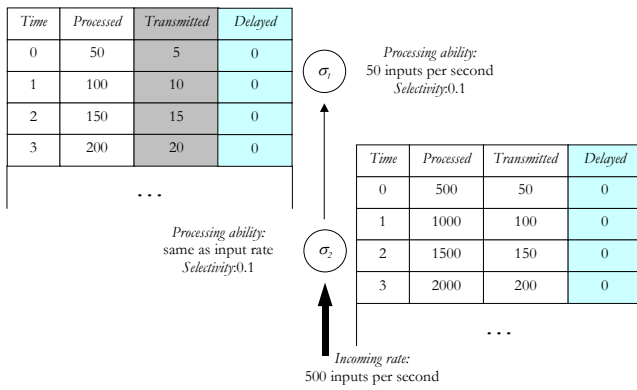
(a) Output rate = 0.5 outputs per second

(b) Output rate = 5 outputs per second

Figure 1: Two possible execution plans over time. The *Transmitted* column designates the output rate.

In this simple case, there is a way to "hack" existing technology to make it apply. We could start with an optimization framework for queries with expensive methods, like the one in [8], and choose an arbitrary constant K as the input cardinality. Then the optimizer will choose the right plan, since the correct plan is independent of K. However, in general, this approach will fail. In particular, in plans that combine multiple input streams, or combine local files and input streams, an arbitrary choice for K will not suffice, because combining multiple branches of a plan with arbitrary "faked" input sizes will only find a good overall plan by dumb luck.

Basing our optimization on rates rather than cardinalities solves this problem in a systematic way. Rather than asking, "What is the cost of evaluating this query plan?" we ask, "what is the expected output rate of this query plan?" By optimizing for output rate, we can choose the correct plan in the preceding example; as we will show, we can also optimize more complex plans with multiple branches.

A rate-based framework has other advantages in addition to working in the "infinite input stream" case. As one example, a rate-based optimizer can make better decisions than traditional optimizers if the inputs are finite streams that are accessed at widely differing rates. For another example, as we will show, for many query plans, even with fixed uniform input rates the output

rate of plans vary with time. Since our framework lets us estimate rates as a function of time, the rate-based approach allows us to optimize for any desired point during the execution – we can optimize for the most possible outputs in the first 10 seconds. Furthermore, since the number of tuples produced by time t is just the integral of the rate from time zero to t, we can also use rate-based optimization to optimize for the first k results, for arbitrary k. This ability may render rate-based optimization useful even in traditional database applications (since a scan of a disk-based table can certainly be viewed as a streaming input.)

We evaluate our rate-based framework in an experimental study, which yields satisfactory results regarding the validity of our approach. Moreover, our study demonstrates that when optimizing for rates in a stream processing environment, choosing one plan over another is non-trivial, since simple heuristics such as "put the fastest streams at the leaf levels of the plan", or "pull expensive operators higher in the execution pipeline" have situations where they fail. However, an unfortunate challenging aspect of our framework is that using rates and integrating them over time yields mathematical expressions that appear difficult to evaluate. To address this problem, we provide heuristics by which we can effectively approximate the cost of an evaluation plan, and show that these approximations appear good enough to be useful in query optimization.

The rest of the paper is organized as follows: Section 2 presents related work. Section 3 deals with the estimation of output rates for the most important operators appearing in a query execution plan, while Section 4 presents our general framework for rate-based query optimization. Section 5 presents an experimental study and validation of the optimization framework. Finally, Section 6 summarizes our conclusions and identifies future research directions.

2 RELATED WORK

The seminal paper on cost-based query optimization was [16]. Other optimization models have been proposed, especially in the areas of parallel query optimization, using cost models that are not cardinality-based but instead deal with resource scheduling and allocation [7], [13]. The Britton-Lee optimizer could optimize for the first result tuple [18], while in Mariposa [17] 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 have appeared in many contexts, including [3], 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 symmetric join algorithms [2], [22], [24], which aim at producing plans that do not block their execution because of slow input streams. Our framework indicates that with variable rate sources it is beneficial to employ such algorithms. In the same context, methodologies aiming at avoiding blocked parts of an execution plan at runtime [23] can benefit from our framework of rate optimization by starting with and/or switching to plans for which the predicted output rate is maximized.

The most closely related areas of work come from the adaptive query execution and dynamic re-optimization frameworks of [2] and [10]. In these frameworks, the main concern is to dynamically monitor an execution plan and identify points of sub-optimal performance. Once such points are identified, the system can choose to reorganize the plan in a way that is expected to yield better performance. In [10], such points are detected by incrementally measuring the cardinalities of partial outputs and comparing them to the optimizer's estimates. If the measured and estimated cardinalities differ by a substantial amount, the optimizer is called to generate a better execution plan under the new information. In [2], the objective is to dynamically adapt and improve performance by rerouting inputs to particular operators thus improving overall performance. They initially choose an execution plan through a heuristic pre-optimizer and then continuously monitor the executing plan's performance. They also use runtime deviations from selectivity estimates as a criterion to identify sub-optimal performance.

We view both of these frameworks as complementary to our work. Our rate-based optimization framework may be useful in conjunction with both these techniques – it could be used to choose an initial plan in cases where rate-based optimization is more appropriate than cost-based optimization. Then instead of switching plans when the runtime detects that the optimizer has botched a cardinality estimate, we would switch plans when the runtime detects that the optimizer has botched a rate estimate.

Work has also been done in the context of continuous queries over data streams in two directions: the first one aims at characterizing the behavior of these queries with respect to their memory requirements [1], [4]. Additionally, [6] deals with identifying and maintaining *stream statistics* for *sliding window* queries. We expect most queries over streams to fit into such a window-based operation framework. Our work is, again, complementary, since we deal with plan enumeration and selection. Window-based operation is conducive in choosing the amount memory the execution framework allocates to the various join operators. It can therefore provide better estimates for the cost of a join operator to our optimization framework.

Finally, our work fits also into the re-optimization frameworks of [12], [15], which focus on identifying performance bottlenecks of an already executing plan and ways to overcome them. In that direction, we provide insight into why these bottlenecks may appear when dealing with network-resident, streaming data, while our cost model could be used to make run time decisions when re-optimizing. Moreover, in a re-optimization framework like the one of [12], the performance crossing points our framework identifies can aid in scheduling when re-optimization should take place.

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. In this paper we will concentrate on some important and basic operations that are used in relational, object-relational, object-oriented and semi-structured database systems: (possibly generalized) selections, projections and joins.

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 with 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:

$$Output\ rate = \frac{Number\ of\ outputs\ transmitted}{Time\ needed\ to\ make\ the\ transmission} \qquad \textbf{Equation 1}$$

In what follows, we will use the cost variables of Table 1 to model an operation, expanding upon 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, we will refer to the pair of inputs with r_l for the left input and r_r for the right input.

Table 1: Cost variables used in the estimation of output rates

Cost Variable	Meaning
C_π	Cost of projecting parts of an input object
C_σ	Cost of performing a selection on an input object
C_l	Cost of handling an input coming from the left-hand side of a join
C_r	Cost of handling an input coming from the right-hand side of a join
T	Cost of making a single transmission

3.1 Projections

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

1. The cost of performing one projection is less than or equal to the inter-arrival time for input objects ($C_\pi \leq 1/r_i$).

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

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

Figure 2: Relation between cost and inter-arrival rate. Variable r_i models an input rate.

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_π 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 $r_o = 1/C_\pi$. In most cases we can safely assume, however, that the cost of making a projection is low, so for small values of C_π $r_o = r_i$.

3.2 Selections

For selections we need to incorporate the selectivity of the predicate under evaluation. Given the input rate, the number of input objects in one time unit will be r_i. Assuming a uniform distribution, the number of objects appearing in the output will be $f \cdot r_i$, where f is the predicate selectivity. We can calculate the output rate in a way analogous to the calculation of the projection output rate, with the only difference that we are using C_σ instead of C_π. The output rate will then be $r_o = f \cdot r_i$ if $C_\sigma \leq 1/r_i$ and $r_o = f/C_\sigma$ if $C_\sigma > 1/r_i$. Again, in most cases it is safe to assume that $C_\sigma \leq 1/r_i$, so $r_o = f \cdot r_i$.

3.3 Joins and Cartesian Products

Joins are more complex than projections and selections because they have two inputs. Before proceeding, 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 be the observed rate at time t. In particular, if the system
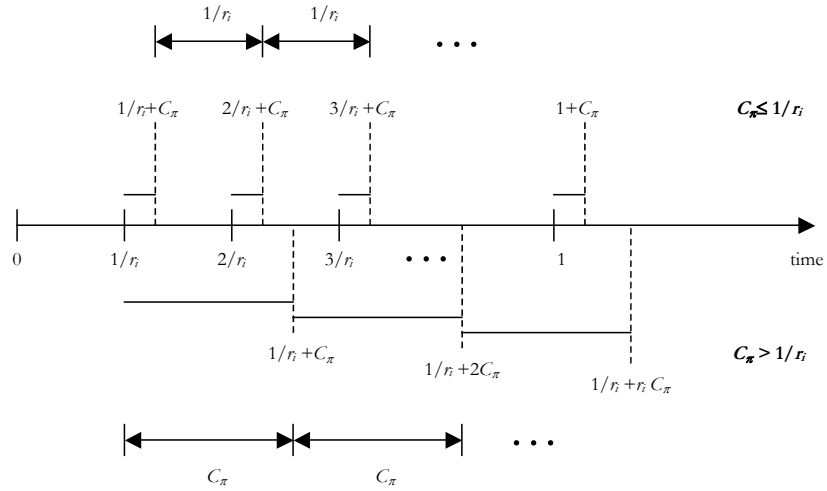
spends time processing these arrivals, the output tuples[1] corresponding to these arrivals may not appear until some point in the future. The rate our model predicts will be the rate at that point. Additionally, asking, "which tuples arrived at an arbitrary instant t" does not make sense, since time is continuous. So we instead ask about discrete time intervals, and then generalize from these discrete intervals 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. The number of result objects generated by the arrivals in one time unit, given we begin at time t, will be $f \cdot r_l \cdot r_r \cdot t^2$, where f is the selectivity factor. This can be seen as follows: 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 $r_r \cdot t$. 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 $f \cdot r_l r_r$. (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 $f \cdot r_l \cdot 2 \cdot r_r$ (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 $f \cdot r_r \cdot 2 \cdot r_l$. Thus the total number of output elements[2] generated by arrivals during the second time unit will then be $2 \cdot f \cdot 2 \cdot r_l r_r - f \cdot r_l r_r$ (the subtracted term avoids double counting). The total number of outputs produced by arrivals during the first two time units will then be the sum $f \cdot r_l r_r + 3 \cdot f \cdot r_l r_r$. Using the same logic, the total number of

---

[1] We use the term *tuple* to signify an output result, since the database reader is more familiar with that term. The result does not have to be a tuple in the relational sense, of course.

[2] In what follows, we will use the terms *tuple*, output *object*, *output* and *output element* interchangeably to avoid repetition.

outputs generated by arrivals during the first three time units will be $f \cdot r_l \cdot r_r + 3 \cdot f \cdot r_l \cdot r_r + 5 \cdot f \cdot r_l \cdot r_r$. 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 $r_o = \int f \cdot r_l \cdot r_r \cdot (2t-1) \, dt$. Solving the integral for time yields the total number of output objects produced by a join operation for the input that arrives at any time t, which is $f \cdot r_l \cdot r_r \cdot t \cdot (t-1)$.

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 $r_l \cdot t$ inputs from the left stream and $r_r \cdot t$ objects from the right stream. The time to handle each of the left inputs is by definition C_l while for each of the right inputs the cost is C_r. Then the time to process these input tuples will be $r_l \cdot t \cdot C_l + r_r \cdot t \cdot C_r = t \cdot (r_l C_l + r_r C_r)$. Substituting the above results into Equation 1 yields:

$$r_o = \frac{f \cdot r_l \cdot r_r \cdot t \cdot (t-1)}{t \cdot (r_l \cdot C_l + r_r \cdot C_r)} = \frac{f \cdot r_l \cdot r_r \cdot (t-1)}{r_l \cdot C_l + r_r \cdot C_r} \approx$$

$$\frac{f \cdot r_l \cdot r_r \cdot t}{r_l \cdot C_l + r_r \cdot C_r}$$

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 $t \cdot (r_l C_l + r_r C_r)$, is greater than t, which means that $(r_l C_l + r_r C_r) > 1$. 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 $f = 1$.

From Equation 2 it is clear that the output rate of a join operation is time-dependent. The time dependence is actually subtler than that formula indicates, because for some join operator implementations the constants C_l and C_r also depend on time (e.g., if the cost of handling an input depends upon the number of 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, or optimizing for specific time points of the operation. We will present such a framework in Section 4.

3.4 Cost Models for Join Operator Implementations

The purpose of this section is to derive 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 symmetric nested loops and the symmetric hash join. The cost expressions we devise are

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 the four cost variables given in Table 2.

A subtle issue that needs to be addressed here is that of the input sizes. For a join operation to generate the complete result, it has to compare each record in one of its inputs to each record in the other input. When inputs grow, so will the memory a join operator needs in order to ensure correct operation. If the inputs are unbounded, infinite storage is needed. Clearly, this is a problematic situation. We expect, however, most streaming applications to be accompanied by window predicates that will effectively bound memory requirements.

Table 2: Notation for cost formulas

Cost Variable	Meaning
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

3.4.1 Nested Loops Join

The nested-loops join algorithm traditionally needs all of the inner source's input present to execute properly. The outer source 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 straightforward 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 *partial results* architectures, like the one in [21].

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 of handling one left input arrival is then: $C_l = \text{move} + |S|_t \cdot \text{comp} = \text{move} + r_S t \cdot \text{comp}$, where $|S|_t$ is the number of inputs read from stream S at time t and is obviously equal to the stream's rate multiplied by time. For right input arrivals, a loop over all the left inputs has to be initiated. The cost is then: $C_r = \text{move} + |R|_t \cdot \text{comp} = \text{move} + r_R t \cdot \text{comp}$.

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 $C_l = C_r = \text{move} + \text{hash} + \text{probe}$: 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.
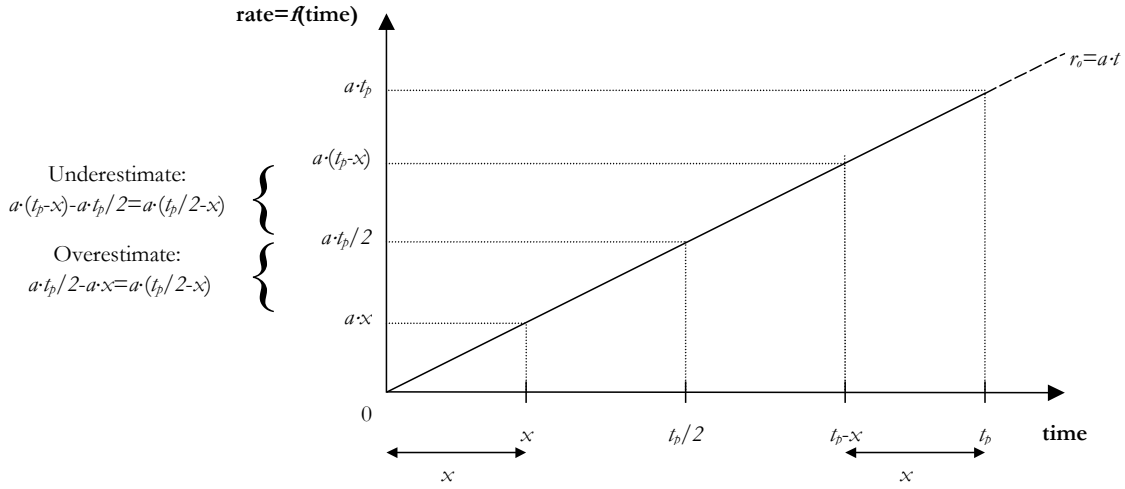
Figure 3: Assigning a constant value to the time variable as a means of approximation.

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.

Table 3: Cost formulas for the join algorithms

Algorithm	Left arrival cost (C_l)	Right arrival cost (C_r)
Nested loops	move+r_S·t·comp	move+r_R·t·comp
Symmetric hash join	move+hash+probe	move+hash+probe

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_p is given by the integral of the rate over time:

$$\#Outputs = \int_0^{t_p} r(t)dt \qquad \textbf{Equation 3}$$

The integral of Equation 3 provides the general framework for rate-based optimization. The problem then becomes: given a collection 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 that will reach the specified number of results the soonest and use it. In this situation we are asking: "which plan is the first one to reach N results?" Notice that N can be the total number of results, or the first result, or any result size in between.

The optimization opportunities we listed require computing the solution to an integral, which is inefficient for practical optimization purposes, as an optimizer can be expected to evaluate a large number of plans, and numerically integrating each plan will be costly. Accordingly, we need to explore means to approximate the integral of Equation 3. In the next section we will propose two such approximation heuristics for this purpose: 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, our intent is to illustrate a general class of heuristics rather than claim that these are the only heuristics that perform well in practice. Both of the heuristics we present aim at locally optimizing the plan under the premise that better local performance leads to better overall performance. We will concentrate on identifying plans with maximal rate over time, and plans that reach a specified number of results as soon as possible. For the first case, we propose a performance estimate for each join operation of the plan and we combine such estimates bottom-up to come up with an estimate for the whole plan. An optimizer will then aim at *maximizing* that estimate. In the second case, we use the same performance

estimate, working top-down, 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 of the query (i.e., joins that none of their inputs is the output of another join) we can devise an estimate for any given time interval t_p, by treating the time variable as a constant and assigning it a value of $t_p/2$, thus assuming that the overall rate for the whole time interval is, according to Equation 2, equal to $\frac{f \cdot r_l \cdot r_r}{r_l \cdot C_l + r_r \cdot C_r} \cdot \frac{t_p}{2}$. As Figure 3 shows, in making this choice we overestimate the rate for any time point x by as much as we underestimate it for a complementary time point t_p-x. We then use this rate estimate to evaluate the whole plan performance[3] in the fashion Figure 4 depicts. After coming up with a heuristic estimate for all candidate plans we choose the most promising one.

It is easy to incorporate this performance estimate in an optimizer. For instance, a dynamic programming optimizer dealing with streaming sources, instead of calculating the traditional cost for a join operation, would use the heuristic estimate as a performance indicator, proceeding as in Figure 4. Notice that the estimate allows the incorporation of any other CPU and I/O metric by inserting it into the calculation through C_l and C_r.
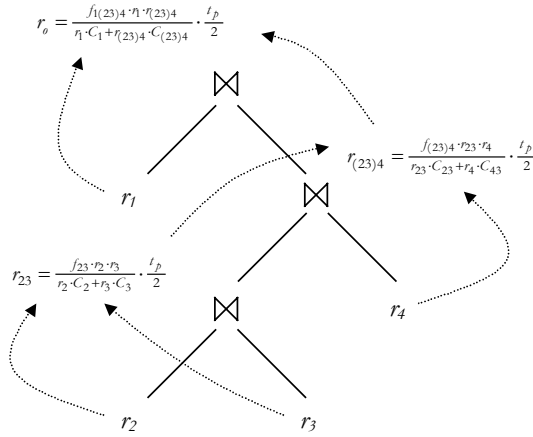

Figure 4: Local rate maximization for time t_p

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. The formula that connects output rate r, time t and number of outputs produced n is $n=r \cdot t$. 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, n/r.
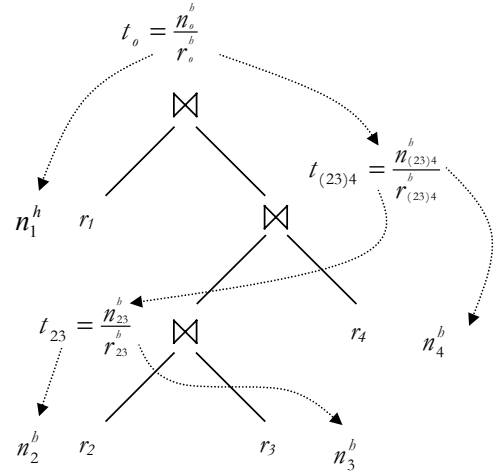

Figure 5: Required output size distribution for local time minimization

We can incorporate this strategy into a more general optimization framework: suppose we are facing the operation $A\bowtie B\bowtie C$. We wish to optimize for the time needed to reach 25% of the total output[4]. 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. This can be done as follows. We know that the number of overall outputs we optimize for is equal to $0.25 \cdot f_{AB} \cdot f_{BC} \cdot |A| \cdot |B| \cdot |C|$. Taking join sequence $A \bowtie (B \bowtie C)$ as an example, to reach our goal we approximately need to read $\sqrt{0.25 \cdot f_{AB} \cdot f_{BC}} \cdot |A|$ inputs from the base stream A and $\sqrt{0.25 \cdot f_{AB} \cdot f_{BC}} \cdot |B| \cdot |C|$ from $B \bowtie C$. Using this divide-and-conquer strategy we can handle 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 that will reach the desired number of outputs the soonest, 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 heuristic estimate of Section 4.2.1 is the following: We want an indication

---

[3] In fact it is possible, though not trivial, to come up with bounds as to how much we over- or underestimate each possible plan while heuristically evaluating it.

[4] Obviously, this strategy is not only applicable for fractions of the total output size, it can be used to optimize for the total number of outputs.

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, the value of its metric, and the fastest time for this number to be reached. For instance, given join strategy $A \bowtie (B \bowtie C)$, the information about it is encapsulated into $\{(A, n_A, r_A, t_A), (B \bowtie C, n_{B \bowtie C}, r_{B \bowtie C}, t_{B \bowtie C})\}$. The notation here is that we encapsulate each point of interest P (which can be a source stream, or the output of a join) in the structure (P, n_p, r_p, t_p), in 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 $\{(B, n_B, r_B, t_B), (C, n_C, r_C, t_C)\}$. 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 all the way 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. A grammar for the structures is then:

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

The input to our decision algorithm is a set of such structures. Solving the problem 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. The following algorithm presents a simple recursive 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.

$max(\varnothing, 0)$.

$max(Info \cup Tree, M) \quad \leftarrow \quad$ $Info = (S, n_S, r_S, t_S)$,
$\qquad max(Tree, M_t)$,
$\qquad t_S = n_S / r_S$,
$\qquad (M_t > t_S ? M = M_t : M = t_S)$.

$min(\varnothing, \perp, \infty)$.

$min(Tree \cup Forest, \quad \leftarrow \quad max(Tree, C_t)$,
$\quad BestTree, C_b) \qquad min(Forest, BestInForest, C_f)$,
$\qquad\qquad (C_t > C_f ?$
$\qquad\qquad [BestTree = Tree, C_b = C_t] :$
$\qquad\qquad [BestTree = BestInForest,$
$\qquad\qquad\quad C_b = C_f])$

The algorithm provides for substantial flexibility with regards to the cost of an operator.

5 EXPERIMENTAL VALIDATION OF THE FRAMEWORK

In this section we provide experimental results to explore the validity of our rate-based cost model. We focus our attention on two questions:

1. *Does the cost model correctly estimate individual plan performance?*

2. *Is the framework capable of providing correct decisions regarding the best choice among a set of plans?*

As is the case with traditional cardinality-based optimization, it would be unrealistic to expect the optimizer to be accurate to the granularity of seconds. We did expect it, however, to be correct in terms of identifying points of interest in an execution plan. For instance, if two plans "cross" in terms of which is best at some point, the optimizer should predict such a crossing point and roughly identify where it occurs.

5.1 Experimental Setup

For our experiments we used a synthetically generated XML data set for which we could vary the parameters of interest to the execution process, such as the various selectivities of the join predicates and the rates of the streaming sources. The sources were essentially flat structures in a way they would appear if exported from a relational database. To investigate streaming behavior, we simply view the files as prefixes of (potentially infinite) streams. The experimental data set consisted of five such sources, with sizes ranging from 0.7MB to 9.3MB.

Our experiments involved queries built out of four equijoin predicates, with selectivities ranging from 10^{-5} to $5 \cdot 10^{-3}$. Table 4 presents the specifics of our sources, while Table 5 presents the four join predicates and their respective selectivities. We treat the join predicate $A \bowtie C$ as an expensive one, assigning to it an additional transmission delay, while for the rest of the predicates their costs are equal to the cost of the evaluation algorithm. Because of its natural fit with streaming environments, we have used symmetric hash join as the evaluation algorithm for all join predicates.

Table 4: Parameters of the streaming sources

Source	Number of tuples	Size
A	5,000	0.7 MB
B	10,000	1.5 MB
C	20,000	1.8 MB
D	50,000	5.9 MB
E	100,000	9.3 MB

All experiments were performed using the publicly available Java prototype implementation of the Niagara Query Engine [14]. The hardware setup involved a Pentium-III processor operating at 500 MHz with 1GB of physical memory running RedHat Linux 6.2. The data were read from flat XML files using Apache's XML Parser and the parsing startup time was subtracted from the results. Byte-code was generated by IBM's *jikes* Java compiler, while the runtime environment was SUN's 1.3.1 Java Virtual Machine. The operators were run in an operator-per-thread fashion, using Linux's native threads. 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 [3], with a mean arrival rate equal to the stream's rate, which implies that the delays followed an exponential distribution with a mean equal to the inverse of the stream's rate (i.e., the inter-arrival delay.)

Table 5: Join predicates' parameters

Predicate	Selectivity	Handling cost
$A \bowtie B$	$2 \cdot 10^{-3}$	-
$A \bowtie C$	$5 \cdot 10^{-3}$	5 ms
$B \bowtie D$	10^{-4}	-
$D \bowtie E$	10^{-5}	-

5.2   Validation of the Cost Model

As a first step towards validating our optimization framework, we evaluated the performance of a three-way join query containing the predicates $A \bowtie B$ and $A \bowtie C$. We explored two execution plans: $(A \bowtie B) \bowtie C$ and $(A \bowtie C) \bowtie B$ (see Figure 6). We assigned an inter-arrival delay to each stream, with stream B being the fastest, having an inter-arrival delay of 2 milliseconds, while streams A and C were considerably slower with inter-arrival delays of 20 and 10 milliseconds. We then fed each plan's parameters into an estimator we developed using the rate-based optimization framework as the plan evaluation criterion. The issue was to estimate the performance of each plan as a plot of output size *vs.* time. We asked the estimator to generate the time, in seconds, of each of the two plans for output sizes between 0 and 90,000 in increments of 5,000 tuples. The estimator's prediction is shown in Figure 7, while Figure 8 depicts the measured performance. Observing that the two plans had comparable predicted performance for the first 5000 result tuples, we decided to zoom in on the first tuples, generating the predictions in Figure 9.
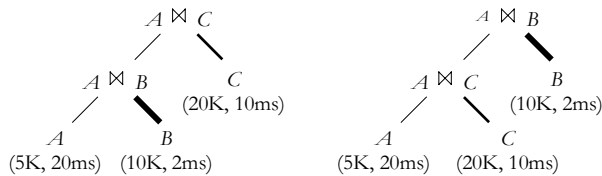

Figure 6: The two plans used for initial experimentation. A thicker line denotes a faster stream.

After predicting the performance of the plans, we ran them through the execution engine, keeping track of the time at which each result tuple appeared. Although not exactly matching the predictions (the actual performance curve was more ragged than the estimated curve) the general behavior of each plan was similar to the prediction. More importantly, the cost model not only predicted there were time intervals for which each one plan outperformed the other, it also predicted the point, in terms of number of output tuples, at which the performance would switch between the two plans. As Figure 9 shows, plan $(A \bowtie C) \bowtie B$ was predicted to start outperforming $(A \bowtie B) \bowtie C$ once roughly 2,000 result tuples appear, which is quite close to the real crossover point in Figure 10. The prediction, though, was farther off when predicting the actual time at which the switch occurred. As we have noted, however, at this point it is more important for the

optimizer to identify crossing points in the graphs, rather than be precise in the granularity of seconds of when these crossing points appear.
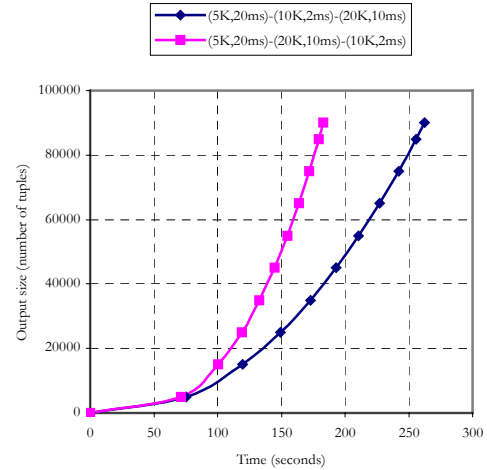

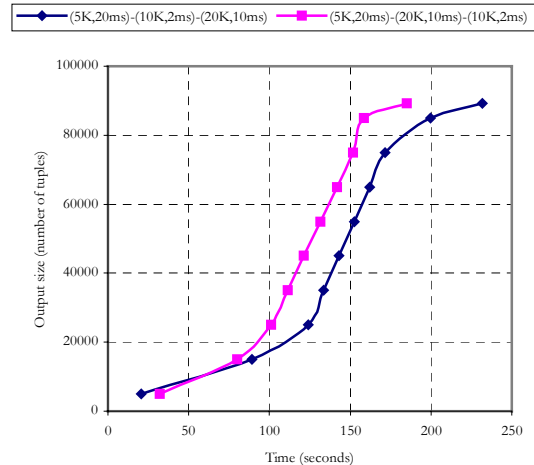**Figure 7: Estimated plan performance until the last result tuple**

Figure 8: Measure plan performance until the last tuple

5.3   Complex Plans

To further validate the optimization framework, we generated a five-way join query over all streaming sources. In this experiment we wanted to address two issues. The first was whether our framework was effectively capturing the order among different plans. That is, given a collection of more complex plans than those of Section 5.2, would it still correctly order the plans? The second issue we wished to explore was whether there were simple rules about stream placement in plans that could render our optimization unnecessary. For instance, is it sufficient to place fast streams at the lowest levels of the plan, or at the highest levels of the plan?
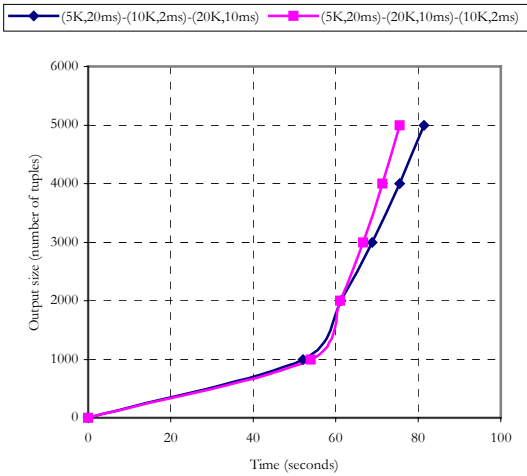
Figure 9: Estimated plan performance for the first few thousand tuples
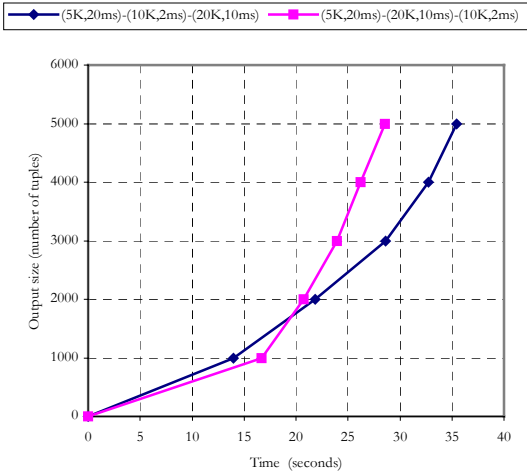

Figure 10: Measured plan performance for the first few thousand tuples

We organized the plans in three different ways, depicted in Figure 11, in which a thicker line denotes a faster stream. The first plan was a deep plan, with the fastest stream kept at the top end of the pipeline. The other two plans were bushy plans with one major difference. While in the first plan the fastest streams are at the lowest levels of the plan, the second plan keeps the fastest one in the middle of the join sequence, while the second fastest stream is kept at the lowest level of the pipeline. In what follows we will refer to the plans of Figure 11 (a), (b) and (c) as *Left Deep*, *Fast Leaves* and *Evenly Spread* respectively.

Figure 12: Estimated plan performance until the last result tuple for complex plans

As in Section 5.2, we first asked the estimator to predict the plans' performance in terms of output size vs. the time at which that particular output size appeared. Figure 12 shows the result. Again, we noticed intervals for which one plan outperformed the others, so we zoomed in to the first few results tuples, producing the plot of Figure 14. We then executed the plans, producing the graphs of Figure 13 and Figure 15 for the entire plan execution and for the first few tuples respectively.

The optimizer's predictions, as far as ordering the plans and crossing points are concerned, were again correct. The plan performance estimations were that *Left Deep* would be the slowest plan, while at the same time it would have comparable performance to *Fast Leaves*. *Evenly Spread* on the other hand, would clearly outpeform the other two, starting to do so from the initial stages of execution.

Figure 14 shows the optimizer predicting that *Left Deep* would have marginally better performance than *Fast Leaves* for the first 20,000 tuples, which is the actual case as Figure 15 depicts.

(a) Left Deep

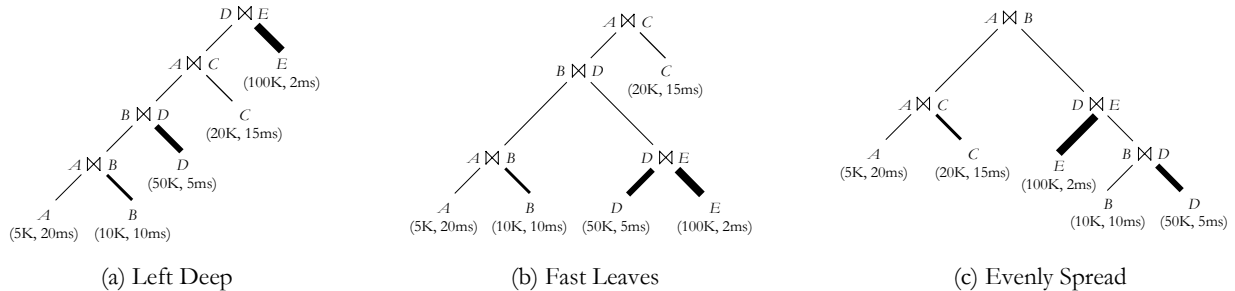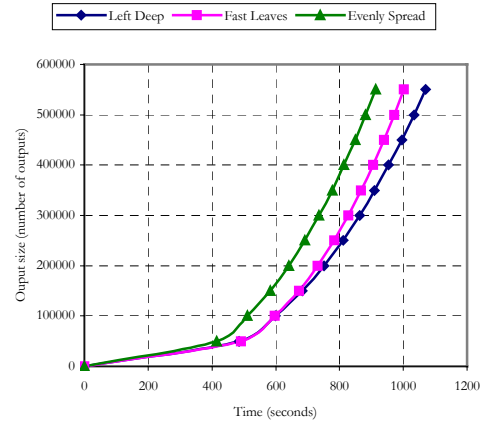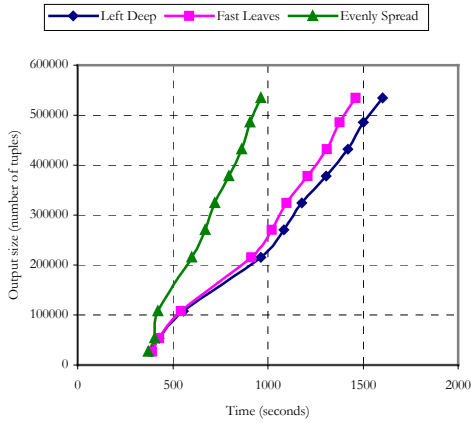(b) Fast Leaves

(c) Evenly Spread

Figure 11: The plans used for our experimentation. Each plan is annotated with the join predicates it computes. A thicker line denotes a faster stream.

Figure 13: Measured plan performance until the last result tuple for complex plans

What is also important is that there does not seem to emerge a clear heuristic. We cannot say that fast streams should be kept away from the highest levels of the plan, as is the case in *Left Deep*, nor that should be pushed to the lowest levels, as is the case in *Fast Leaves*. The best plan is *Evenly Spread*, which keeps the fastest in the middle of the plan, while the second fastest is at the leaf level. A lot more experimentation has to be carried out before a heuristic, if any exists, can be devised. Our intuition is that the nature of the problem is too complex for simple heuristics to uniformly affect performance. The reason is that the interaction between streams in rate-based optimization is not as simple as the one between input and output sizes in cardinality-based optimization is.
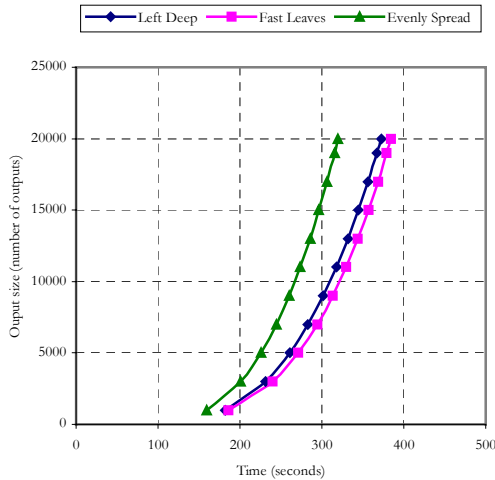

Figure 14: Estimated plan performance for the first few thousand tuples for complex plans
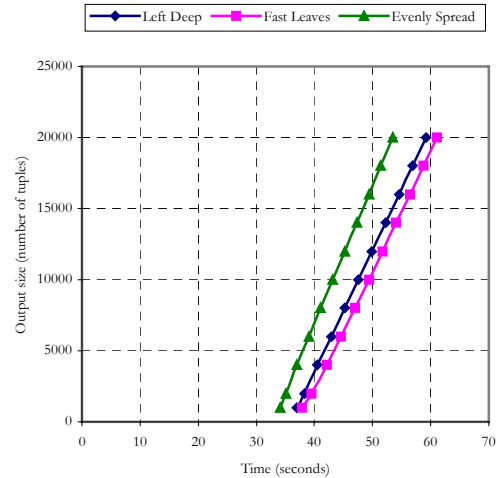
Figure 15: Measured plan performance for the first few thousand tuples for complex plans

5.4 Comparison to the Traditional Cost Model

We now turn to a final issue. In the presence of infinite input streams, it does not make sense to compare our rate-based approach with the traditional cardinality based approach, since the cardinality-based approach does not apply. However, even in cases where traditional optimization does apply (that is, over finite input streams or files) there may be cases where rate-based optimization is preferred. For this to be true, we need to find cases where, for some query and some optimization metric, the rate-based approach made different and better decisions than the cardinality based approach.

To explore this issue, we assumed finite input streams of known cardinality and generated a plan estimator using the traditional cost model. We then passed the three plans of Section 5.2 (see also Figure 11) through the traditional cost model's estimator. To compare, we asked the rate-based estimator to cost the plans in terms of final result output performance, i.e., time needed until complete results are produced. Table 6 summarizes the results. From that table it is clear that the rate-based estimator could distinguish between the plans, predicting which would be the first to reach the final result size. The traditional estimator, on the other hand, although it successfully managed to identify *Left Deep* as the most expensive plan, it failed to distinguish between the two bushy plans, costing *Fast Leaves* as the cheapest one.

Table 6: Comparison between the traditional and the rate-based cost model

Plan	Traditional Estimation	Rate-Based estimation
Left Deep	10^4	$1.3 \cdot 10^3$
Fast Leaves	$2 \cdot 10^3$	$9.7 \cdot 10^2$
Evenly Spread	$5 \cdot 10^3$	$8.8 \cdot 10^2$

In this case, the reason why the cost-based optimizer orders the plans incorrectly is that it assumes all of its input is present when execution commences. This is, however, not the case. The size of the input is time-dependent, which is essentially what the rate-based optimization framework captures by optimizing for output rate. In our example, and between the *Fast Leaves* and *Evenly Spread* plans, the earliest time point by which a single input's source will be entirely present will be after 100 seconds (sources *A* and *B* will have been completely read in by then.) At this time, though, only 33% of *C*, 40% of *D* and 50% of *E* will be present. The cardinality-based estimator fails to identify that and assumes all of the input is present at the same time, focusing on the handling cost of each input, disregarding the fact that this input might not be even present.

6   CONCLUSIONS AND FUTURE WORK

In this paper we propose rate-based optimization as a way to enable query optimizers to work with infinite input streams. For more traditional applications, rate-based optimization may be useful because it allows optimization for specific points in time during query evaluation – e.g., find the first plan to generate 1000 answers, or the one that generates the most answers in the first five seconds. We proposed a cost framework based upon rates, and gave examples of how this framework applies to select-project-join queries.

To evaluate our framework, we compared the predications an optimizer would make using our framework with measured execution times in a prototype version of the Niagara Query Engine. The results of this experiment indicate that rate-based optimization is indeed a potentially viable approach, worthy of further exploration.

A great deal of room for future work exists – in fact, we think that this initial work raises as many questions as it answers. In one direction, our cost models are quite simple, with rough heuristics to approximate integrals and naïve assumptions about the costs of various operators as a function of their inputs. Clearly these can be refined. In another direction, and the one we perhaps find most interesting, there are potentially powerful synergies between our rate-based approach and previous work on adaptive or dynamic query processing and re-optimization. We plan to explore both directions in future work.

BIBLIOGRAPHY – REFERENCES

[1]   A. Arasu, B. Babcock, S. Babu, J. McAlister and J. Widom, *Characterizing Memory Requirements for Queries over Continuous Data Streams*, Stanford Techinical Report, November 2001, http://dbpubs.stanford.edu/pub/2001-49.

[2]   R. Avnur and J. M. Hellerstein. *Eddies: Continuously Adaptive Query Processing*, Proceedings of the 2000 ACM SIGMOD Conference.

[3]   D. Bertsekas and R. Gallager. *Data Networks*, Prentice Hall, 2nd edition, 1991.

[4]   S. Babu, and J. Widom, *Continuous Queries over Data Streams*, SIGMOD Record, Sept. 2001.

[5]   J. Chen, D. J. DeWitt, F. Tian and Y. Wang. *Niagara-CQ: A Scalable Continuous Query System for Internet Databases*, Proceedings of the 2000 ACM SIGMOD Conference.

[6]   M. Datar, A. Gionis, P. Indyk and R. Motwani, *Maintaining Stream Statistics over Sliding Windows*, 2002 Annual ACM-SIAM Symposium on Discrete Algorithms (SODA 2002).

[7]   M. N. Garofalakis and Y. E. Ioannidis. *Parallel Query Scheduling and Optimization with Time- and Space-Shared Resources*, Proceedings of the 23rd International VLDB Conference.

[8]   J. M. Hellerstein, *Optimization Techniques for Queries with Expensive Methods*, TODS 23(2): 113-157 (1998).

[9]   W. Hong and M. Stonebraker. *Optimization of Parallel Query Execution Plans in XPRS*, Distributed and Parallel Databases, 1993, (1) 1:9-32.

[10]   Z. G. Ives, D. Florescu, M. Friedman, A. Levy and D. S. Weld. *An Adaptive Query Execution System for Data Integration*, Proceedings of the 1999 ACM SIGMOD Conference.

[11]   Z. G. Ives, A. Y. Levy and D. S. Weld. *Efficient Evaluation of Regular Path Expressions on Streaming XML Data*, University of Washington, Technical Report UW-CSE-2000-05-02.

[12]   N. Kabra and D. J. DeWitt. *Efficient Mid-Query Re-Optimization of Sub-Optimal Query Execution Plans*, Proceedings of the 1998 ACM SIGMOD Conference.

[13]   C. Lee, C.-H. Ke, J.-B. Chang and Y.-H. Chen. *Minimization of Resource Consumption for Multidatabase Query Optimization*, Proceedings of the 3rd IFCIS Conference.

[14]   J. Naughton, D. J. DeWitt, D. Maier et al. *The Niagara Internet Query System*, IEEE Data Engineering Bulletin 24 (2): 27-33 (2001), http://www.cs.wisc.edu/niagara.

[15]   K. W. Ng, Z. Wang, R. R. Muntz and S. Nittel. *Dynamic Query Re-Optimization*, Proceedings of the 11th SSDBM Conference.

[16]   P. G. Selinger, M. M. Astrahan, D. D. Chamberlin, R. A. Lorie and T. G. Price. *Access Path Selection in a Relational Database Management System*, Proceedings of the 1979 ACM SIGMOD Conference.

[17]   M. Stonebraker, P. M. Aoki, W. Litwin, A. Pfeffer, A. Sah, J. Sidell, C. Staelin and Andrew Yu. *Mariposa: A Wide-Area Distributed Database System*, VLDB Journal, 1996, (5) 1:48-63.

[18]   G. Schumacher. *GEI's Experience with Britton-Lee's IDM*, IWDM, 1983, pp. 233-241.

[19]   L. D. Shapiro. *Join Processing in Database Systems with Large Main Memories*, TODS, 1986, (11) 3:239-264.

[20]   J. Shanmugasundaram, E. J. Shekita, R. Barr, M. J. Carey, B. G. Lindsay, H. Pirahesh and B. Reinwald. *Efficiently Publishing Relational Data as XML Documents*, Proceedings of the 26th VLDB Conference.

[21]   J. Shanmugasundaram, K. Tufte, D. J. DeWitt, J. F. Naughton and D. Maier. *Architecting a Network Query Engine for Producing Partial Results*, WebDB 2000.

[22]   T. Urhan and M. J. Franklin. *Xjoin: A Reactively-Scheduled Pipelined Join Operator*, IEEE Data Engineering Bulletin, June 2000, (23) 2:27-33.

[23]   T. Urhan, M. J. Franklin and L. Amsaleg. *Cost Based Query Scrambling for Initial Delays*, Proceedings of the 1998 ACM SIGMOD Conference, Seattle, Washington, USA, June 1998, pp. 130-141.

[24]   A. N. Wilschut and P. M. G. Apers. *Pipelining in Query Execution*, Conference on Databases, Parallel Architectures and their Applications, Miami, 1991.

[25]   C. Zhang, J. F. Naughton, D. J. DeWitt, Q. Luo and G. M. Lohman. *On Supporting Containment Queries in Relational Database Management Systems*, Proceedings of the 2001 ACM SIGMOD Conference.