Multi-Model Based Optimization for Stream Query Processing

Ying Liu and Beth Plale
Computer Science Department
Indiana University
\{yingliu, plale\}@cs.indiana.edu

Abstract

With recent explosive growth of sensors and instruments, large scale data-intensive and computation-intensive applications are emerging, especially in scientific field. Helping scientists to efficiently, even in real time, process queries over those large scale scientific streams thus has great demand. However, query optimization for such high performance stream applications— in particular its core component, the evaluation model— has not been systematically studied. We observe that evaluating stream query plans should consider two aspects: output rate and computation cost. However, existing research on evaluating stream query plans only uses one or the other. In this paper, we propose a new optimization goal which leverages both aspects and develop a multi-model based optimization framework to accomplish this goal. Specifically, we build a new Cost Model for stream query processing, to our knowledge, which is the first cost model under stream processing context. We also refine the Rate Model which is first brought by Viglas and Naughton [17]. Based on these two models, we search for an optimal plan satisfying the new proposed optimization goal. We experimentally justify our optimization framework by running simulated scientific data streams on Calder, a grid enabled stream processing system developed in our lab.

1 Introduction

In recent years, technological advancements that have driven down the price of handhelds, cameras, phones, sensors, and other mobile devices, have benefited not only consumers but the computational science community. As a result, data-driven computing is emerging, where computationally intensive applications often need real-time responses to data streams from distributed locations. These streaming sources can have vastly varying generation rates and event sizes. Responsiveness, i.e., the ability of a data-driven application to respond in a timely manner is critical. For instance, out-of-date analysis of beam luminosity data can be useless or worse yet may mislead particle physicists. Therefore, in practice, efficiency plays an important role in stream query processing.

Stream query processing has been an active research area in recent years [1, 2, 9, 6, 13], yet limited work has been done on query optimization for such high performance stream applications. Especially, to our knowledge, the core part of stream query optimization, i.e., the evaluation model, has not been systematically studied in this context.

As infinite event sequences, data streams introduce new challenges to query plan selection. First, since cardinality is not available for streams, the cardinality-based cost model loses its usefulness under the stream processing scenario. Second, data are not guaranteed to be fully processed. If a query processor does not process streaming data in time, the data will be lost forever once they are removed from a buffer. Unlike traditional query processing where all input data are processed, stream query processing may yield output from incomplete input data. Therefore, besides computation cost, output completeness, which is represented by output rate, is another important aspect for evaluating stream query plans.

In response to these challenges, we develop a multi-model based stream query optimization framework that considers both output rate and computation cost for stream query processing, in which a Rate Model and a Cost Model are built to measure output completeness (output rate) and computation cost respectively. We observe that these two models are related: computation cost is estimated based on input streams’ input rates; at the same time, output rate is estimated with the knowledge of query operators’ computation costs. However, as we will explain in Section 2.1, these two models are not exactly consistent: an optimal plan selected by the Rate Model may not the optimal one wanted by the Cost Model.

In most applications, output rate is more important, because the output generated from the partial input may be biased, especially for scientific applications. Therefore, we propose a new optimization goal: Among a set of plans with the maximum output rate, finding a plan with the minimum
cost. To accomplish such a goal, we propose a multi-model based optimization framework: a Rate Model is used to find all the plans with maximum output completeness and a Cost Model can select the one with minimum computation cost among the set of plans with maximum output rate. We realize both models by giving metric estimation for each query operator. Further, we implemented our optimization framework in Calder [18], a grid enabled stream processing system.

In summary, this work has the following contributions:

- We observe that stream query optimization should consider two aspects: output completeness and computation cost.
- We propose a new optimization goal: “among a set of plans with maximum output rate, finding a plan with the minimum cost”, and develop the multi-model based stream query optimization framework.
- We build a new cost model for stream query processing, which, to our knowledge, is the first systematical study for the cost model under stream query processing context. In addition, we also refine the rate model brought by Viglas and Naughton by providing more accurate and efficient rate estimations.
- We implement the multi-model based optimization framework on Calder, a grid enabled stream processing system.

The remainder of this paper is organized as follows: First, in Section 2, we discuss our motivating observations and propose the multi-model based optimization framework. Section 3 introduces preliminary definitions used in this paper. Then, in Section 4 and Section 5, we build a Cost Model and refine a Rate Model respectively, which are basis of our optimization framework. Section 6 reports experimental justification of the multi-model based stream query optimization. Last, Section 7 discusses related work and Section 8 future research plans.

2 Motivation and Framework

In this section, we first discuss three observations which motivate our work. Based on the analysis of the motivating observations, we present a new optimization goal, which covers both metrics of output completeness and computation cost. We further formalize such an optimization goal and propose the multi-model based optimization framework.

2.1 Motivating Observations

Query evaluation model defines an optimal query plan. In traditional DBMS systems, a cost model selects an optimal plan with minimum computation cost, because a plan with less computation cost can save computation resource for stream query processing system and make the system more scalable. The same argument still holds under the stream query processing context. Hence, minimum computation cost is still a necessary quality for an optimal query plan in stream query processing. However, as we discussed in Section 1, output completeness should also be counted to evaluate a stream query plan besides the computation cost. We call the output came from partial input data set as incomplete output. The output completeness can be measured by output rate. But output rate measurement is out of the scope of a Cost Model, although rate parameters can be used to estimate the cost.

Therefore, we need a Rate Model which is similar to a Cost Model in defining a series of formulas to estimate query plans’ metrics without execution. The difference is that Rate Model estimates a query plan’s output rate instead of its computation cost. Also, Rate Model has its own optimization goal: Maximum Output Rate, which is different from the Cost Model’s Minimum Computation Cost.

We notice that under the stream query processing context, Cost Model and Rate Model are related: computation cost is estimated based on input streams’ input rates in the Cost Model; at the same time, output rate is computed with the knowledge of query operators’ computation costs in the Rate Model. Then, are these two models consistent? Does Maximum Output Rate guarantee Minimum Computation Cost? The following observations give the answer. The symbols used here will be formally defined in Section 3. To explain it clearly, we use some formulas defined in Section 4 and Section 5.

Observation1: Plans with the same output rate but different costs

Figure 1. Observation1: Plans with same output rate but different costs
the operator. The arrow lines denote intermediate streams whose rates are marked above the lines. Although these two plans have the same output rate \( r_o \), planB’s computation cost 67480 is more than 28 times higher than planA’s cost 2400. Therefore, barely using maximum output rate as the optimization goal, we can not distinguish two plans and may choose planB which consumes more computation resources. Hence, purely using maximum output rate, e.g., the Viglas model [17], may result in choosing a plan with higher computation cost.

**Observation2:** The plan with minimum computation cost must have maximum output rate. We prove this by contradiction. Suppose two logically equivalent plans A and B: planA has minimum computation cost but its output rate is less than that of planB. All the equivalent plans are made of the same set of equivalent operators and they are only different in operators’ connection order and operators’ implementation. Hence, all operators of planA, which has minimum computation cost, must have less computation cost than those of planB. On the other side, planB has higher output rate means that at least one operator of planB can process data faster than that of planA. That is, planB has at least one operator with less computation cost than planA’s. This contradiction means that our assumption is wrong. That is, Plan B does not exist and Plan A with minimum computation cost must have maximum output rate. **Observation2** implies that minimum computation cost is a sufficient condition for maximum output rate. Therefore, it seems that we can use only minimum computation cost as a goal for stream query optimization. Is it true? We have the answer in the following.

**Observation3:** From observation2, we might hypothesize that less computation cost means higher output rate. However, it is not true. As shown in Figure 2, both \( S \) and \( P \) have different implementations in planA and planB, which cause their different costs. In planA, \( S \) is fast enough to let all input data pass smoothly. But planB has congestion in \( S \). This causes planA has higher output rate. But planB has lower cost than planA. Hence, a plan with less computation cost does not necessarily has higher output rate.

With pure minimum computation cost as optimization, a sub-optimal plan may be chosen because most searching algorithms are approximated. However, this sub-optimal plan selected based on the computation cost may have very low output rate. Hence, combined optimization goal is needed.

### 2.2 Multi-Model Based Optimization Framework

Based on above observations, we realize that although Cost Model and Rate Model are not totally independent, their goals are not always consistent with each other. Therefore, stream query optimization should leverage both models and have a combined optimization goal. In most applications, output rate is more important as the output generated from the partial input may be biased, especially for scientific applications. Therefore, we design our optimization goal as: **Among a set of plans with maximum output rate, finding a plan with the minimum cost.** Formally, we assume that \( \text{cost}(\cdot) \) is the function to compute a query plan’s computation cost using the Cost Model. Similarly, \( \text{rate}(\cdot) \) represents the function for a query plan’s output rate using the Rate Model. Then, an optimal query plan \( P_o \) can be defined as:

\[
P_o = \arg\min_{p \in \mathcal{P}}(\text{cost}(\mathcal{P}))
\]

where \( \mathcal{P} = \arg\max_{s \in \mathcal{S}}(\text{rate}(s)) \) and \( \mathcal{S} \) is the searching space of all the equivalent plans. Note that \( \mathcal{P} \) is a set of plans reaching the maximum output rate, instead of a single one.

Leveraging both models, we use a two-stage searching strategy to find an optimal plan satisfying the optimization goal. First, we find all the plans with maximum output completeness with Rate Model; then within those plans, we choose the one with the least computation cost based on Cost Model. In each stage, we can apply heuristics to reduce search space. As shown in [13], the following algebraic heuristics can be applied for stream query optimization:

- Pushing down selection operators in the query tree.
- Pushing up projection operators in the query tree.
- Reorder selection operators based on the statistical information about the incoming data.

In our future work(Section 8), we will discuss that we are working on more effective searching algorithm in the multi-model based stream query optimization framework.

### 3 Preliminary Definition

In Section 2, we proposed a multi-model based query optimization framework in response to our observations about stream query processing. To complete this framework, we
3.2 Cost Related Terminology

A stream $S$ is defined as a sequence of events, $S = \{e_i\}$ where $i$ is a monotonically increasing number and $0 < i < \infty$. An Event is an event input to the stream and the counterpart to a tuple of a traditional database table. Like the traditional query processing system, query processing engine, as a container of queries, is also the core part of the streaming query processing model. However, in the stream processing system, queries are continuous query [4] which stay in the engine continuously evaluating incoming events. Similar to [8], our Stream Query Processing Model shown in Figure 3, takes streams as input and output. A processing engine holds continuous queries and processes stream data event sequentially.

3.1 Stream Query Processing Model

A stream $S$ is defined as a sequence of events, $S = \{e_i\}$ where $i$ is a monotonically increasing number and $0 < i < \infty$. An Event is a unit data in the stream and the counterpart to a tuple of a traditional database table. Like the traditional query processing system, query processing engine, as a container of queries, is also the core part of the streaming query processing model. However, in the stream processing system, queries are continuous query [4] which will stay in the engine and continuously evaluate incoming events. Similarly to [8], our Stream Query Processing Model shown in Figure 3, takes streams as input and output. A processing engine holds continuous queries and processes stream data event sequentially.

3.2 Cost Related Terminology

In the traditional database cost model, disk I/Os are the principle cost. But in stream query processing, input and output events are passed in/out through networks and most processing occurs within main memory. That is, disk access is minimized in stream processing. Therefore, the CPU processing time becomes the major cost $c$ for a centralized system, which is the focus of this paper. In general, we use $c_s$, $c_p$ and $c_j$ denoting computation cost per input event for selection, projection and join operators respectively. As we will discuss in Section 4, these costs can be computed from the unit cost $c_{su}$ and $c_{pu}$, where $c_{su}$ is the unit cost to compare a condition and $c_{pu}$ is the unit cost to copy an attribute. Once we have each operator’s computation cost, we can estimate a query plan’s cost $C_q$ as the sum of all its operators’ computation cost per time unit.

$$C_q = \sum_{i=1}^{k} (c_i \cdot r_i)$$

where $r_i$ is the input stream’s rate for operator $i$. The rate definition can be found in Section 3.3.

3.3 Rate Related Terminology

Given two adjacent events $e_j$ and $e_{j+1}$ in a stream which come at time $t_j$ and $t_{j+1}$ separately, we define Stream Rate $r$ as $r = 1/(t_{j+1} - t_j)$. Accordingly, Input Rate $r_i$ and Output Rate $r_o$ of a query are the input or output stream’s Stream Rate for the query. Each query operator has Processing Rate $r_p$ which defines the number of events that can be processed by this operator per time unit.

Borrowing terms from the queuing theory, we define two modes for stream query plans: Overload Mode and Steady Mode. For an operator, if $r_p < r_i$, i.e., the processing rate for an operator is less than that of the input stream, then the congestion occurs and the operator falls behind. We call such a operator as an Overloaded operator. A query plan is defined in the Overload Mode if an overloaded operator exists in this plan. Otherwise, we call the plan works in Steady Mode.

Table 1 lists all the symbols defined in this section.

<table>
<thead>
<tr>
<th>Terminology</th>
<th>Symbol</th>
</tr>
</thead>
<tbody>
<tr>
<td>Input Rate</td>
<td>$r_i$</td>
</tr>
<tr>
<td>Output Rate</td>
<td>$r_o$</td>
</tr>
<tr>
<td>Processing Rate</td>
<td>$r_p$</td>
</tr>
<tr>
<td>Unit Cost for Selection</td>
<td>$c_{su}$</td>
</tr>
<tr>
<td>Unit Cost for Projection</td>
<td>$c_{pu}$</td>
</tr>
<tr>
<td>Selection Computation Cost</td>
<td>$c_s$</td>
</tr>
<tr>
<td>Projection Computation Cost</td>
<td>$c_p$</td>
</tr>
<tr>
<td>Join Computation Cost</td>
<td>$c_j$</td>
</tr>
<tr>
<td>Query Plan’s Computation Cost</td>
<td>$C_q$</td>
</tr>
</tbody>
</table>

Table 1. Terminology Definition

4 Cost Model

Many research efforts [16, 10, 5] have been spent on cost models in traditional DBMS system, although they measure different cost metrics from ours. However, there is little systematic research on a cost model for stream query processing till now, although systems [11, 6, 1] implicitly use a cost for stream query optimization.

We develop a new Cost Model for stream query processing. To our knowledge, this is the first cost model under the stream query processing context. As discussed in Section 1, a cost model includes a series of formulas to estimate query plans’ computation costs without execution. Based on such cost estimation, we can choose the plan with minimum computation cost. In the following we will explain how to estimate a query plan’s cost defined in Section 3.
4.1 Cost Estimation for Selection

According to Table 1, $c_s$, $c_p$ and $c_j$ denote the cost of selection, projection and join operators respectively. This section will show how to compute $c_j$ for different selection operators based on the Unit Selection Cost $c_{su}$, which is defined as the cost to evaluate one selecting condition. We know that there are mainly two ways to combine the basic select conditions: AND($\land$) or OR($\lor$). Therefore, we only study the following two cases: $(con1 \land con2)$ and $(con1 \lor con2)$. More complex selection cases can be derived based on these two basic cases.

The evaluation process for the case, $(con1 \land con2)$, is that $con1$ is evaluated first and $con2$ evaluated only when the $con1$ is true. So, if $p$ is the probability that $con1$ is true, the total cost to evaluate such condition $c_s1$ is:

$$c_{s1} = p \times (c_{su} + c_{su}) + (1 - p) \times c_{su}$$

Similarly, to evaluate the condition $(con1 \lor con2)$, we first evaluate $con1$. If $con1$ is true, we will stop. Only when $con1$ is false, will $con2$ be evaluated. So, the cost $c_{s2}$ for $(con1 \lor con2)$ is:

$$c_{s2} = p \times c_{su} + (1 - p) \times (c_{su} + c_{su})$$

From above two equations, we can see that the order of basic conditions does matter for a selection operator’s Local Computation Cost. This brings us the optimization opportunity. To lower the cost, having the basic condition with less True Value Probability, $p$, as the first condition in AND combination; but having the one with larger $p$ as the first condition in OR combination.

Now we look at how to estimate True Value Probability, $p$, for a basic condition. We adopt the heuristic rules discussed in [7]. There are three kinds of different basic conditions:

- **Constant Equation**, for example $(a < 23)$. $a$ is an attribute of a stream $S$. For such condition, we need $V(S, a)$, which is the number of distinct values the stream $S$ has in attribute $a$. Even though the stream $S$ has unlimited size, we will collect such statistic data from part of stream $S$. For example, we compute $V(S, a)$ on the data events coming in the recent 2 hours. Given $V(S, a)$, we define that condition $(a < 23)$ has the probability $p = 1/V(S, a)$ to be true.

- **Constant In-equality**, for example $(a \neq 15)$. We assume that all tuples satisfy the condition. That is, probability $p = 1$.

- **Inequality Comparison**, for example $(a < 10)$. 50% may be a reasonable value for $p$. But as mentioned in [7], queries involving an inequality tend to give a small fraction of the possible tuples. Thus, $p$ is assumed as $1/3$.

4.2 Cost Estimation for Projection

Comparing with the selection operator, the projection operator is much simpler. It has the relational algebra as $\pi(a_1, a_2...a_n)(S)$, in which $S$ is an input stream and $(a_1, a_2...a_n)$ is an attribute list. This attribute list is normally the subset of $S$’s attribute set. The output event is constructed by this attribute list $(a_1, a_2...a_n)$ for $n$ attributes.

As for the projection’s implementation, each event comes into the processing system and the system copies the selected attributes into the output event. We defines Unit Projection Cost, $c_{pu}$, as the cost to copy one attribute. This cost can be varied for the attributes with different data types. $c_{pu}$ is assumed as the average value. With this assumption, a projection operator with the attribute list $(a_1, a_2...a_n)$ has the cost $c_p$ as:

$$c_p = n \times c_{pu}$$

4.3 Cost Estimation for Join

Unlike the selection and projection operators, join operator is a binary full-relation operator. That is, the join operation needs the presence of the full relations. However, unlimited stream size makes traditional join algorithms not suitable for stream query processing. Consequently, many stream query processing engines implement join operation over a sliding window, instead of the full stream.

Consider a join operation $R \bowtie S$, where $R$ and $S$ are input streams. They have input rates $r_1$ and $r_2$ respectively. The selectivity of this join operation is $\sigma$. $R \bowtie S$ is actually equal to $\pi(\sigma(R \times S))$. That is, join operation is implemented based on selection and projection: we first do the comparison (the selection with just one condition) to see whether two events can be joined, if so, we project (combine) two input events to generate the output event. And the cost for each such projection is $n \times c_{pu}$, if the output event has $n$ attributes. Therefore, the join operator’s Local Computation Cost $c_j$ as defined in Section 3 should be:

$$c_j = c_{su} + \sigma \times n \times c_{pu}$$

To decide the selectivity $\sigma$, we need to know how the join attribute is related in two streams. We just mentioned that many Stream Query Processing systems implement the join operation on a sliding window. By assuming the join happens on data events falling into sliding windows, we
make streaming join operation similar to that in the traditional database. Therefore, we use a join’s approximation introduced in [7]. To simplify the problem, we have the following assumptions:

- If \( V(R, Y) \leq V(S, Y) \), then every Y-value in R will be a Y-value in S.
- \( V(R \bowtie S, A) = V(R, A) \) where A is non-join attribute in stream R.

With these assumptions, the join can be estimated as follows. Suppose \( V(R, Y) \leq V(S, Y) \), then every event \( e \) in R has \( 1/V(S, Y) \) chance of joining with a given event in S. Suppose there are \( N(S) \) events in stream S’s sliding window, the expected number of events that event \( t \) joins with is \( N(S)/V(S, Y) \). Considering there are \( N(R) \) events in stream R’s sliding window, the estimated size of \( R \bowtie S \) is \( N(R)N(S)/V(S, Y) \). Similarly, if \( V(R, Y) \geq V(S, Y) \), \( N(R \bowtie S) = N(R)N(S)/V(R, Y) \). Therefore, the estimated size for \( R \bowtie S \) is:

\[
N(R \bowtie S) = N(R)N(S)/\max(V(R, Y), V(S, Y))
\]

Accordingly, the selectivity

\[
\sigma = |N(R \bowtie S) - N(S)N(R)| = 1/\max(V(R, Y), V(S, Y)).
\]

where \( V(R, Y) \) and \( V(S, Y) \) are statistic data collected from all the events falling in the sliding window.

In a summary, this section defines a series of formulas to estimate computation cost and according selectivity for selection, projection and join operations. With these estimations, we can further estimate a query plan’s computation cost.

5 Rate Model

Rate Model is a new concept for stream query processing. Viglas and Naughton [17] propose to use Maximum Output Rate as a stream query optimization goal. To find a plan with maximum output rate, Viglas et al give a series of formulas to estimate the output rates for three basic query operators: selection, projection and join. However, their model requires computing the solution to an integral which is not efficient to evaluate a large number of query plans. Although they use rough heuristics to approximate integrals, this causes inaccurate estimations. Hence, we refine their models with a new estimation techniques on the join operation. In addition, we show that the output rate is fully determined by operator’s processing rate in Overload Mode and by streams’ input rate in Steady Mode. In this section, we introduce our Rate Model and all the symbols used in this section are defined in Section 3.

5.1 Rate Estimation for Projection

For unary operator: selection and projection, we have similar rate estimation as Viglas model. An operator’s processing rate \( r_p \) can be decided by its computation cost \( c \). For the projection operator, \( r_p = 1/c_p \). In the Steady Mode, there is no congestion and the input events are passed as incoming rate. Hence, output rate \( r_o = r_i \). However, under the Overload Mode, where processing rate \( r_p \) is less than input rate \( r_i \) and output event comes out at the operator’s processing rate \( r_p \), the output rate is \( r_o = r_p \). Therefore, we can say that output rate of projection operator is the minimum of \( r_i \) and \( 1/c_p \), that is, \( r_o = \min(r_i, 1/c_p) \).

5.2 Rate Estimation for Selection

To estimate the output rate of a selection operator, selectivity must be considered. Besides, selection’s rate estimation is similar to that of projection’s. Therefore, given a selection operator with selectivity \( \sigma \), input rate \( c_i \) and the input stream with input rate \( r_i \), the output rate \( r_o \) for this selection operator is \( r_o = \sigma \cdot \min(r_i, 1/c_i) \).

The selectivity \( \sigma \) varies for specific situations. The estimation for selectivity, \( \sigma \), is derived from [7] based on the detailed discussion in Section 4: Constant Equation, \( (a = 23) \), has the selectivity \( \sigma = 1/V(S, a) \); Constant Inequality, \( (a \neq 15) \), has the selectivity \( \sigma = 1 \) and Inequality Comparison, \( (a < 10) \), has the selectivity \( \sigma = 1/3 \).

5.3 Rate Estimation for Join

The join operator is a Binary Full-Relation operation. Traditional join algorithm does not work under streaming query processing scenario, where different Sliding Window Based join algorithms are applied. The Output Rate varies with different algorithms. In this paper, we study the join operator that executes natural join over a time-based sliding window [3], where the window size \( T_w \) is defined by the time interval. Other join implementations have similar analysis. We plan to do more complete study for various operator implementations in the future, as we will discuss in Section 8.

As shown in [15, 12], a time-based sliding window can be implemented at low cost, and when tied to input stream rate, can be more intuitive for users than a count-based sliding window, where the window size is defined as the number of interested events.

Suppose two streams of events \( R \) and \( S \). \( R \) is composed of events \( e_i: R = \{e_i\} \) and \( 0 < i < \infty \). \( i \) is a monotonically increasing number. Similarly, \( S = \{e_j\} \) and \( 0 < j < \infty \). Further, suppose stream \( R \) has an input rate of \( r_1 \) and \( S \) an input rate of \( r_2 \). We show that in the Steady Mode for the
system as described, Output Rate \( r_o \) only depends on the input streams’ Input Rate.

In addition, \( \sigma \) is selectivity of a join operator, which can be viewed as the ratio of the number of output events to the number of input events for the operator.

There are \( r_1 \cdot T_w \) events at any time residing in the sliding window of stream \( R \) and \( r_2 \cdot T_w \) events residing in the sliding window of stream \( S \). Without loss of generality, we take as the observation period the time interval \([t_0, t_0 + t]\). During this period, \( r_1 \cdot t \) events will arrive on stream \( R \), each of which will be joined by Cartesian product to all events in stream \( S \)'s sliding window. Those joins generate \( r_1 \cdot t \cdot r_2 \cdot T_w \cdot \sigma \) output events during the observation interval. An event output from a join is a tuple \( < e_i, e_j > \) where \( e_i \in S, e_j \in R \). Examining this from the side of stream \( S \) instead, with similar analysis we obtain \( r_2 \cdot t \cdot r_1 \cdot T_w \cdot \sigma \) events output during the observation interval \( t \).

Putting together, there are

\[
\begin{align*}
    r_1 \cdot t \cdot r_2 \cdot T_w \cdot \sigma + r_2 \cdot t \cdot r_1 \cdot T_w \cdot \sigma = 2 \cdot (r_1 \cdot r_2 \cdot t \cdot T_w \cdot \sigma)
\end{align*}
\]

(1)
events generated during such an observation period.

Even though we compute the joined events from two sides, there is no overlapping in the computation. As shown in Figure 4, join happens on two asynchronous input streams \( R \) and \( S \). In step1, an event \( R_j \) comes to the system and joins with all events in the sliding window on the \( S \) side. Then step2, an event \( S_k \) comes and joins with all events in \( R \)'s sliding window. The join on event \( R_j \) and event \( S_k \) is only counted once. Similarly all other joins are counted only once. As for the synchronous case where two input events come at the same time, the most processing system use one process to deal with join operation to save the cost from interprocess communication. Hence, two input events are still processed sequentially. Overall, we can say there is no overlapped join counted in Equation 1.

Therefore, the output rate is the number of events divided by the observation time \( t \), that is:

\[
\begin{align*}
    r_o &= \frac{2 \cdot (r_1 \cdot r_2 \cdot t \cdot T_w \cdot \sigma)}{t} \\
    &= 2 \cdot r_1 \cdot r_2 \cdot T_w \cdot \sigma
\end{align*}
\]

(2)

In addition, \( \tau \) is selectivity of a join operator, which can be viewed as the ratio of the number of output events to the number of input events for the operator.

Table 2 lists output rate estimations for three kinds of basic operators. In summary, we estimate join operators’ output rates with simple arithmetic formulas. Although query optimizers are expect to evaluate large number of plans, computing such simple formula, instead of integrals, is still practical to give satisfactory response. In addition, without using further approximation, which is introduced by integrating, can make more accurate estimations.

6 Experiments

In this section, we first experimentally demonstrate how the Rate Model evaluates query plans and why we need a multi-model based optimization framework. In the second experiment, we show how our multi-model optimization framework differentiate two logically equivalent plans.

6.1 Experiment setup

To validate our cost model, we test continuous streaming queries on dQUOB [14], a publicly accessible streaming query processing engine system. The dQUOB system, implemented in C++, consists of query parser, query optimizer and query processor. Hence, for a given input query, dQUOB always chooses an optimal or sub-optimal query plan and executes it. To conduct different query plans from a single query, we manually generate various relational algebra trees, directly put them into query processor for testing. The input streams are automatically generated, in which each event has unique Timestamp and EventId. The hardware setup includes a dual processor 2.8GHz workstation with 2GB memory running RedHat Enterprise Linux(RHEL).
6.2 Insufficiency of Rate Model: An Empirical Justification

In Section 5, we have analytically pointed out the insufficiency of the Rate Model: it can only distinguish equivalent plans in Overload Mode. In this experiment, we will experimentally justify such inefficiency and demonstrate that output rate can characterize the output completeness. We study two logically equivalent plans shown in Figure 5 and measure the output completeness as well as output rate of these two plans in both Overload Mode and Steady Mode. To simulate the Overload Mode, we can either increase input streams’ rates or slow down operators’ processing rates. In this experiment, we use the latter because the same input streams’ input rate on the same query should yield the same expected number on output events, which makes our comparison more clear. Table 3 lists the computation cost description about each operator in the plans. Input streams, streamL and streamR, are automatically generated. By simulating scientific streams, streamL has 13 attributes and streamR 8 attributes. They both come at rate of 1 event/ms with the size of 0.5 MB in both Overload Mode and Steady Mode.

The first part of this experiment is to compare two query plans’ output rates and measure their output completeness in the Steady Mode. To measure the output rates, we record the cumulative number of output data events at some specific time points. The result is shown in Figure 6, where the slope represents the output rate. Note that we did experiments on real streams, instead of using big files as the prefix of streams, and started measuring after the system enters the stable stage. Therefore, there is no warm up stage in our result. As shown in Figure 6, two query plans have the same output rate in Steady Mode. Further, in Figure 7, we compare the output numbers of each plan with the expected output numbers based on 50000 sequential input events. This experiment shows that both plans reach 100% output completeness and have the same output rate in Steady Mode.

In the second part of experiment 1, we did the same tests on two query plans in Overload Mode. Figure 8 shows that plan A has higher output rate than plan B. Further, from Figure 9, we can see both plans did not reach full completeness due to the congestion in the Overload Mode. However, plan A has more output events than plan B, which is also the reason why plan A has higher output rate than plan B.

From the both parts of experiment 1, we conclude that, first, output rate is a valid metric to measure a query plan’s output completeness; second, Rate Model can only distinguish equivalent plans in Overload Mode and all the equivalent plans should have the same output rate in Steady Mode, as long as they have same input rates. Hence, we can not only use Rate Model to evaluate query plans for stream query processing. Therefore, a multi-model optimization framework is needed.

6.3 Validation of Multi-Model Framework

As discussed in Section 2.2, our multi-model based optimization framework work in two stages: first, it finds all the equivalent plans reaching the maximum output rate. Then, among the set of plans generated from the first stage, we choose the plan with minimum computation cost by using Cost Model. In Experiment 6.2, we already demonstrate how the Rate Model works. This part mainly focuses on the second stage: how the Cost Model selects an optimal plan from a set of equivalent plans which have the maximum output rate.

In this experiment, we study two equivalent plans, which, shown in Figure 10, include selection, projection
and multi-join. We argue that such a query is a typical query under our study scope. Three input streams, stream1, stream2 and stream3, are involved in this query. They have 13, 8 and 12 attributes respectively. All three input streams are generated as the rate of 1 event/ms and each input event has the size of 0.5 MB.

As we discussed in Section 3.2, a query plan’s cost is the sum of all its operators’ computation cost per time unit. Hence, we estimate the computation cost of two plans with our cost model introduced in Section 4. Meanwhile, we measure two plans’ computation costs by recording all of their operators’ processing time during an observation period of $T = 60$ sec. In this experiment, we use a sliding window size of $T_w = 10$ sec for both join operators. All three input streams come at the rate of event/sec. Both estimated costs and experimental costs are also shown in Table 4. From the table, we can see that our cost model not only correctly orders two plans, but also accurately shows how much difference between two plans. At the same time, this experiment also shows that although these two plans, working in the Steady Mode, have the same output rate, they have very different cost: planB’s cost is almost 10 times more than that of planA.

### 7 Related Work

Significant effort [16, 10, 5] has been spent on the query optimization problem in traditional database by using well-recognized Cardinality-Based Cost Model. However, we cannot simply adopt this traditional optimization model for stream query processing because of new challenges brought by stream processing, as we discussed in Section 1.

In recent years, several groups have studied query optimization for stream processing systems. STREAM [2] applies Synopsis Sharing within a single query or among multi-queries. NiagaraCQ [6] exploits incremental group optimization with expression signature. These techniques are brought out as a performance boosting approach and both systems are under centralized environment. Borealis [1] has designed the QoS based multi-level optimization framework for a distributed stream query processing system. However, similar to [2, 1], it only considers the computation cost while evaluating a query plan. In a word, all above three systems evaluate a query plan only based on the computation cost. On the contrary, Viglas and Naughton [17] just consider the output rate in their rate-based query optimization model. In addition, to our knowledge, there is no systematical study on stream query optimization from cost model perspective yet. Therefore, we propose a new optimization goal, considering both computation cost and output rate, and realize this goal with our multi-model based stream query optimization framework.

### 8 Conclusion and Future Work

In this paper, we present our multi-model based optimization framework for stream query processing. Our contribution has four folds. First, we observe that stream query
optimization should consider two aspects: output completeness and computation cost. Second, we propose a new optimization goal: “among a set of plans with maximum output rate, finding a plan with the minimum cost,” and develop the multi-model based stream query optimization framework. Third, we build a new cost model for stream query processing, to our knowledge, which is the first systematical study for the cost model under stream query processing context. In addition, we also refine the rate model based on Viglas’s work to give more accurate and efficient rate estimations. Last, we implement the multi-model based optimization framework on Calder, a grid enabled stream processing system.

In our study for this multi-model based optimization framework, we also observed some open issues that warrant further research.

First, all above work is on a centralized system. However, because streams are distributed, a fact that follows from wide-spread distribution of sensors, a centralized stream data management system is likely to result in limited performance. Therefore, extending such a multi-model query optimization framework to a distributed environment is on our future research agenda. Specifically, we plan to extend a centralized cost model to include distributed cost metrics, e.g., network bandwidth cost, a distributed site’s computation workload and etc.

Second, as mentioned in Section 5.3, this paper focuses on the join operator implemented with time-based sliding window when we estimate the output rate of a join operator. However, there are many other implementations and algorithms [3, 8, 19], e.g. count-based sliding window, pipelined hash join and etc, for different operators in stream query processing. They share the similar selectivity estimation and counting methods with the operators discussed in this paper, but have their own unique characters. We plan to conduct more complete study on the rate/cost estimation of the operators with various implementations and algorithms.

Third, we currently use heuristic based searching algorithm with a two-phase strategy, as we discussed in Section 2. Such an algorithm is efficient but not very effective, i.e., it might miss some optimal plans. We are working on a more effective searching algorithm for multi-model based optimization framework.

References