RI08002, 4 November 2004 Computer Science

IBM Research Report

Performance Modeling and Placement of Transforms for Stateful Mediations

Vinayaka Pandit
IBM Research Division
IBM India Research Lab
Block I, I.I.T. Campus, Hauz Khas
New Delhi - 110016, India.

Rob Strom
IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne
NY 10532-1596

Gerry Buttner
IBM T. J. Watson Research Center
19 Skyline Drive, Hawthorne
NY 10532-1596

Roman Ginis
BAE Systems Advanced Information Technologies
Autonomic Distributed Systems

IBM Research Division
Almaden - Austin - Beijing - Delhi - Haifa - T.J. Watson - Tokyo - Zurich

Performance Modeling and Placement of Transforms for Stateful Mediations

Vinayaka Pandit[1] and Rob Strom[2] and Gerry Buttner[2] and Roman Ginis[3]

[1] IBM India Research Laboratory, email:pvinayak@in.ibm.com
[2] IBM TJ Watson Research Center, email:{strom, gbuttner}@watson.ibm.com
[3] BAE Systems Advanced Information Technologies, Autonomic Distributed Systems Division, email:roman.ginis@baesystems.com

Abstract. In this paper we propose a new technique for placing large delivey plans for streaming systems on a network of machines to optimize efficiency measures such as latency. In the model we consider, there is a large network of machines and the different fixed end-points of the network act as publishers and subscribers of information. Information demanded by subscribers is a transformed view of the information published by the publishers. The transformed view is the outcome of an acyclic network of simple transformations operating on the publishers' information or some intermediate transformed view of it. We propose algorithms for the optimal placement of the acyclic transform network on the network of machines. As an example scenario to evaluate the efficacy of our algorithms we consider SQL queries on streaming relational tables. The transform network in this case is the SQL operator tree for the query. We first show how to model the performance of individual operators acting on distributed streams and then develop the optimal placement strategy for different optimization measures. We present our work on a distributed message-oriented middleware and a programming platform for large-scale publish-subscribe applications called SMILE. In our system, we use incremental implementation each of the relational operator for streaming data. We demonstrate that our technique performs significantly better than straightforward approaches like greedy, and random placement.
Keywords: Stream Processing, Publish-Subscribe, Local Search, Operator Placement, Distributed Computing, and Performance Modeling.

1 Introduction

In this paper, we consider performance modeling and efficient evaluation of streaming queries on a distributed infrastructure. Our platform is called SMILE (Smart MIddleware, Light Ends) [15, 23] which is a distributed middleware for publish-subscribe systems. The middleware facilitates anonymous exchange of information between sets of publishers and subscribers on a distributed network. The middleware not only provides anonymity between publishers and subscribers, but also allows transforms to published streams, allowing subscribers to subscribe to states derived from complex operations, such as aggregation, join,

and top-K over these streams (or other derived states). For example, the published streams can be events describing problem reports on managed devices and events describing resolutions, and a subscription might be to the set of devices having the top 5 number of unresolved problem reports. The SMILE platform allows the publishers to publish information in the form of tuple updates to a relation and the subscribers to specify information by specifying SQL queries over the relations. Two prominent issues in such a setting are the implementation of various operators (also called "transforms") for a streaming model and efficient execution of the operators on a distributed network of servers (also called "message brokers"). While there has been a flurry of recent work [1, 8, 5, 19, 7, 21, 6] focusing on computation in the streaming model, the problem of efficient execution over a network has not received much attention. Notable exceptions to this trend are [25, 2]. In this work, we consider the problem of placing the transforms of a query plan to optimize efficiency measures such as end-to-end latency.

The contributions of the SMILE system are twofold: First, SMILE exposes a simple and easy-to-use programming model to deriving state from distributed streams, allowing the users to focus on specifying *what* state to derive rather than specifying *how* to derive it. Second, SMILE hides from developers and administrators the details of how to efficiently deploy multiple queries between multiple publishers and subscribers on a distributed system, and how to recover from lost or re-ordered messages, or failed message brokers. The scalability of SMILE's middleware crucially depends on how efficiently it executes its queries and routes results to the consumers. In support of efficient execution, SMILE includes steps to model the performance of a flow of events through a tree of operators, to use this model to compute an optimal placement of transforms on message brokers that can be used to drive deployment. It is this modeling and placement that is the subject of this paper.

In many deployments of messaging-oriented middleware under the publish-subscribe paradigm, the transforms simply route and filter messages. However, there is a growing interest in more sophisticated middleware, where subscriptions can define not just a filtered stream, but a continuously derived view that can depend on an event history or multiple histories. Subscriptions are specified by means of declarative queries in a language such as SQL or XQuery, and compiled into a *transform network* or *operator tree* consisting of a flow graph of transforms that take in published messages, and incrementally compute the changes to the derived state. Such systems are variously called streaming systems or continuous query systems [21, 1, 29, 8]. SMILE provides a simple programming interface which allows the users to specify complex, large queries in simple, easy to understand, English-like declarative fashion viewing streams as tuple-updates to a base relation. The framework constructs the equivalent SQL query and constructs the operator tree along with the java objects corresponding to the transforms and with necessary functionality to resolve names and communicate on a wide-area distributed network. This is expected to make application development for such middleware very easy and popular.

2 Problem Formulation

Consider a financial system where clients are investment houses distributed around the globe, who subscribe to a service (such as from Reuters or Bloomberg) to receive stock trade streams in real time from all the major exchanges in the world. For example, an event stream from NYSE carries information about trades or offers (to buy or sell), including the stock issue, price, exchange name, etc. Typically a particular trader is only interested in very specific information concerning their investment, such as "Portfolio Value," and wants to continuously observe the value of a portfolio based on the latest ticker prices. SMILE allows a trader to compute this *derived* stream from original stream published by the exchange by writing a 'subscription' (e.g. in SQL) as follows:

```
CREATE Subscription "Portfolio Value" =
SELECT SUM (stock.price * portfolio.volume)
FROM (stock JOIN portfolio ON stock.name)
WHERE (stock.timestamp is LATEST)
```

The SMILE system compiles it into a DAG of relational operators like JOIN, SELECT, and PROJECT acting on different streams. Every event occurring in all the stock exchanges passes through this transform graph so that the query can be evaluated. In this paper, we consider the problem of placing such a transform graph efficiently on a topology graph of brokers.

2.1 Transform Graph

We now elaborate on the intermediate representation of the user query which is one of the two inputs to the placement problem. As in databases, user's SQL-like query is compiled into an operator tree which we call as *transform graph*. The transform graph is a DAG of relational operators connected by their inputs. The inputs could be either original streams which are called as *producers* or derived streams called *derived views*. Derived views which are subscribed to by fixed locations in the distributed infrastructure are called as *consumers*. The producers update their relations in a streaming manner. In general, we use Q to denote a transform graph. We use O to denote the set of operators in Q.

Transforms can be specified as operations on histories (e.g., an operator could compute the moving average of the last minute's trades), but they are typically *implemented* incrementally: that is, the transform keeps the current state and enough information so that in response to a message describing a change to its input, can produce another response describing a change to its output. For example, a transform for computing the moving average over the last minute takes as input insertion and deletion events and generates the resulting updates to the moving average. We assume window sizes such that we can compute the transformations without worrying about the limitations imposed by the "traditional" streaming model which assumes that the transforms work on "massive" datasets. Note that, depending on the operation being performed, a transform could have one or more operation histories as its inputs.

A *pipelined transform flow* is a path in the transform graph from a producer P to a consumer C. It consists of an ordered sequence of transforms $T = \{t_1, .., t_n\}$. When P generates an event e, this event traverses the transform graph which effectively computes a composite function $g = t_n(...t_2(t_1(e)))$. This computation is carried in a pipeline, such that once t_k has computed the result $t_k(e)$ and sent it to t_{k+1}, it can immediately start on the next event e'.

2.2 Topology Graph and Computation

The distributed infrastructure is a key component of the SMILE system. It consists of a set of processing units each of which is called a *broker*. The network of communication channels which connects the brokers in the system is called as *topology graph*.

Individual transform of the transform graph are assigned to the different brokers in the topology graph which carry out the computations. The flow of events between different transforms on different brokers is assigned to the communication links connecting the two brokers. All the transforms assigned to a brokers are coalesced into execution modules $\mathcal{M}$. An $m \in \mathcal{M}$ contains a connected subgraph Q' of the transform graph such that the operators in Q' are all assigned to the same broker. The sources of m are either input stream messages, or are messages travelling between a transform which is not on the same broker and a transform which is in m. The sinks of m are either output stream messages, or are messages travelling an edge in Q to a transform which is assigned to a different broker. Figure 1 shows a broker with two distinct flows.

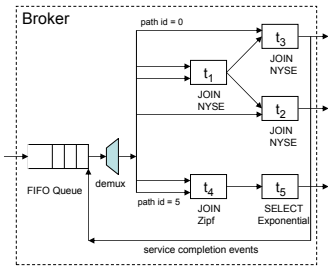

Fig. 1. Broker with two flows

The messages arrive on the input queue[1] of a broker. The broker is modeled as a server that dequeues a message from the input queue, determines which transform it is destined to, and then executes that transform. This may cause one or more output messages to be produced. Messages destined outside of the broker will be delivered either to the client or to the channel to the next broker.

[1] We are assuming single-threaded brokers here, but the model can be extended for multi-threaded brokers.

Messages destined for another transform in the same execution module will cause that transform to execute, possibly generating more messages. This cascade will continue until all dependent transforms have executed, and then the next message will be dequeued. We observe that dequeing of messages and scheduling of the transforms within a broker so as to address issues like starvation of operators and messages is itself a research problem and not addressed here.

2.3 Problem Statement

The input consists of a transform graph Q and a topology graph $G = (V, E)$ where V is the set of brokers and E is the set of network links. Every broker has certain processing power, and for each link, we have the available bandwidth (depending on current network status). We also have a set of producers with specific inut rates. A subset of the operators in Q are fixed to specific locations in G as they are tied to the producers or consumers. Let FX denote the set of fixed operators. We, (i) model the performance of a transformation graph deployed on a topology graph and (ii) place the operators in $O \setminus FX$ of the transform graph on G to optimize efficiency measures such as latency.

2.4 Related Work

Pipelined processing of relational operators has been studied as efficient methods for evaluating traditional database queries in parallel environments [14][18] and for continuous queries [27], long-standing requests to a database system that need to produce outputs as soon as their inputs change. Much of the work related to performance of pipelined operators for stream processing has focused on the development of efficient algorithms for evaluating various transforms, and, in the more recent research projects[7][8][1][21], on the performance optimizations of memory and load management[6]. The problem of placing the operators of a query to optimize efficiency measures for distributed query processing has been considered in [25, 2]. Srivastava et. al [25], consider the problem of operator placement to minimize end-to-end delay when the topology of brokers is a single chain. They give an approximation algorithm for the problem. But, they do not report any experimental evaluation of their algorithm. There is no available implementation of their algorithm against which to compare the results of our experiments. Ahmad and Cetintemel [2] consider the problem of operator placement to minimize the bandwidth utilized in the network. They assume that the processing time of the transforms is zero whereas we consider the processing time of the operators to be nonzero. Thus, the combinatorial structures of our problem and theirs are quite different. Both [25, 2] consider only static version in which all the source-destination pairs are known prior to placement. They do not consider the dynamic version in which only the sources are known apriori and the clients can subscribe to states after the placement has been carried out. We give algorithms for both the static as well as dynamic version of the placement problem. Thus, we contribute significantly to the state of the art in operator placement for distributed query processing.

2.5 Contributions

We propose an analytical framework based on queueing theory which models the performance of important relational transforms like select, projects, joins, sliding window in terms of parameters like input rate, selectivity etc. We extend the modeling for the case of composition of individual transforms on a network. Based on this modeling, we develop two new algorithms for efficient deployment. The first algorithm computes placement to minimize latency of a query using a novel local search heuristic. The second algorithm computes efficient placement for dynamic subscription. This problem is modeled as a balanced-k-partition of the transformation graph and its embedding on the topology graph. We employ a novel local search heuristic to obtain near-optimal assignments. We experimentally validate our performance modeling as well as demonstrate the efficacy of our placement algorithms.

3 Performance Models

3.1 Modeling Approach

We use queueing models to describe the behavior of brokers on the topology graph. We start by modeling the simple relational operators which constitute building blocks of flows in a transform graph. For each transform, it is necessary to determine the distribution of its service time per input message, and the distribution of the "batch sizes" of output messages. The batch sizes are significant, since some transforms (such as SELECT) might absorb some messages without generating new messages (a batch size of zero), and other transforms (such as JOIN) might absorb a message and transmit multiple messages.

3.2 Relational SELECT

The select transform in a stream processing system acts as a filter. It tests every arriving event for a logical predicate and passes it through if the predicate is satisfied. From the performance perspective we need to analyze the service time of the transform and its departure process. Although a select predicate represented in a conjuctive normal form, could take varying amount of processing time, in most applications, the time consumed is assumed to be a constant dominated by the access time for different fields of the record. In our modeling, we assume that the service time is approximately constant, and the batch size is distributed as a random variable, either 0, or 1, whose mean, a measured parameter ϱ, corresponds to the *selectivity* of the transform.

3.3 Sliding Window

We model every stream as a mapping of discrete ticks to "silences", "events", or "unknown" values. The sliding window transform operates on a stream $\mathcal{I}$ and considers only its W most recent events. The output of a sliding window

transform is another stream $\mathcal{O}$ except that in this case, a silence can mean either no-event or a tick that is "old" (that is there are more than W later ticks known).

We exploit the fact that, every stream can be maintained by a *time horizon* h, such that for $i \leq h$, tick i maps to either an event or a silence and for every $i > h$, it maps to unknown, representing the fact that the events are received in order [2]. Messages updating $\mathcal{I}$ correspond to new events, and contain the tick number m of the new event, and the range of preceding silent ticks (so that lost or out-of-order messages can be detected; ideally range should be $[h+1, m-1]$). When an event is received, it is recorded in the list, and the horizon is advanced from h to m. Thus, it generates one update event in the output stream and upto $m-h$ "anti-events" to eject the events from the earliest $m-h$ ticks in the current window.

We consider the case where the events arrive by a stochastic process. Clearly, the output distribution matches the input distribution. What we need to analyse more carefully is the batch size. Suppose the probability distribution function(pdf) $W_1(t)$ models the probability waiting for t ticks for the first event. Similarly, suppose $p(k,t)$ gives the probability of exactly k events occurring in t ticks. When an event occurs after t ticks, we generate one output update and anti-events for the events recorded in the t earliest ticks. We can derive the probability of generating k anti-events and hence, a batch size of $k+1$ as follows:

$$P_k = \int_0^\infty W_1(t)p(k,t)dt$$

Given the set of P_k, one can derive the moments M_1 and M_2 as $M_1 = \sum_{k=0}^{\infty} kP_k$ and $M_2 = \sum_{k=0}^{\infty} k^2 P_k$, where the mean M_1 and the variance $v = M_2 - {M_1}^2$. The expected batch size is then $1 + M_1$.

In particular, for a Poisson process with an arrival rate of λ, we know that $p(k,t) = (\lambda t)^k e^{-\lambda t}/k!$ and $W_1(t) = \lambda e^{-\lambda t}$. Simple substitution steps show that $P_k = 2^{-1-k}$. One can also verify that $M_1 = 1$ and $M_2 = 3$, thus yielding an expected batch size of 2 with a variance of 2.

3.4 Relational JOIN

One of the key transforms in a stream processing systems is the relational JOIN. Stream joins are widely used for specifying correlations and operating on more than one stream at a time, for example matching buy offers and sell offers that match in issue and price.

We consider the streaming version of the double-hash join. To compute $J(R_1, R_2)$ it maintains two hash tables H_1 and H_2 corresponding to R_1 and R_2 respectively. As the tuples of the two relations continue to arrive indefinitely, the two phases of the join algorithm are executed for every tuple that arrives. For example, a new tuple r arriving to R_1 is first hashed in H_1 and then immediately probed against H_2. As a result, the streaming join continuously produces

[2] The above description is a simplification of the actual implementation, which allows for the possibility that event messages can either be lost or can arrive out of order.

rows in response to changes to its input relations R_1 and R_2 to complete the join result $R_1 \bowtie R_2$. We assume that over long time periods, the state needed for each transform is kept in main memory and will not grow without bounds. In practice, this assumption is either valid or can be enforced via mechanisms like short windows, expiration time etc.

Transform Parameters: Let $J(R_i, R_j)$ be a join transform and let $p(v \in R_i)$ be the probability mass function (pmf) of the distribution of values in relation R_i with respect to the join predicate of J. Let $z_i, z_j \in \mathcal{N}^+$ be the number of tuples available for joining for the relations R_i and R_j respectively. While $p(v \in R_i)$ and $p(v \in R_j)$ and z_i, z_j can be different, for simplicity of exposition and without loss of generality, we will assume that the pmf's and the window sizes are respectively the same for both relations and refer to them simply as p and z respectively.

Now for a given tuple $v \in R_i$ the number of tuples matched in R_j is then:

$$m(v) : R_i \rightarrow \mathbb{R}^+ = z * p(v)$$

From this we can define *selectivity* of $J(R_i, R_j)$, as the probability density function of the number of tuples matched N by a new event as:

$$\varrho(n) = \sum_{n=m(v)} p(v)$$

where n is a value of N. The random variable N and its probability distribution $\varrho(N)$ are key to characterizing the service time and the output batch size.

Mapping to Performance Parameters: Suppose that the amount of processing work required by the stream join is c (cycles/tuple) for a tuple that matches exactly one tuple in the target table (which is a special case when the join is performed on the key column of the target table). Then, it would require $k * c$ cycles if the tuple matched k tuples on the target table. This includes the hashing of the tuple, the lookup cost, the retrieval and return of the k matched tuples.

Using our definition of selectivity one evaluation (processing one incoming tuple event) of stream join J requires $C = N \cdot c$ cycles. Furthermore, when this join is deployed on a single processor machine β with speed $\widehat{\beta}$ cycles/(unit time), its *service time* would be described by:

$$D = \frac{C}{\widehat{\beta}} = \frac{N * c}{\widehat{\beta}} \quad \text{(time)} \tag{1}$$

Now D is a random variable describing the service time of join J. As expected, it is a function of only the original distribution of values in the joined relations p, the window size z, cost c and processor speed $\widehat{\beta}$. Its mean and variance are as follows:

$$\begin{aligned} E[D] &= \tfrac{c}{\widehat{\beta}}\, E[N] \\ \sigma^2[D] &= \tfrac{c^2}{\widehat{\beta}^2}\, \sigma^2[N] \end{aligned} \tag{2}$$

Using this terminology, the *service rate* of join J is just $\frac{1}{E[D]}$ joins/sec and the batch size is essentially N. As mentioned earlier, when the pdf's, window sizes or single-tuple costs are different for the relations being joined, then the calculations above can be carried out separately for each join relation.

4 Flow Performance Model

Using the performance models developed in 3, we can characterize the performance of execution modules inside a broker and eventually, the performance of a query deployment on the topology graph.

As described earlier each broker hosts a subgraph of the complete transform graph. As shown in Figure 1 the subgraph can consist of a number of logically unrelated segments of various flows. In order to characterize the performance of the broker using a queueing model we need to estimate: 1) the Mean service time ($\frac{1}{\mu}$) 2) Service time Variance (σ_s^2) and the 3) interarrival and interdeparture time Variances (σ_a^2, σ_d^2). Of course, we will have a mean of the arrival rate (λ).

4.1 Service Time

Let F be the set of flows in a broker. Let $f \in F$ be a flow that consists of a tree (as the transform graph network is feed-forward) of transforms $T = \{t_1, .., t_n\}$. The transforms t_i will be executed in a scheduler-dependent order. The scheduling strategy can also change dynamically and non-uniformly across the network. We assume that there is a serialization of the transforms on a single broker.

The service time of the flow is, of course, the sum of the service times of the transforms, however, some transforms will need to be executed more than once if their ancestors in the execution sequence produced more than one event. We call this the *intra-batching effect*, which has a very important ramification: many events can be leaving a flow in response to a single incoming event and their number and distribution depends on the batch sizes produced within the flow. Considering this, we define service time as follows.

Definition 1. *The Service time D_f of a flow $f \in F$ is the total amount of time the server is occupied due to an incoming event arriving at f.*

In the special case, when the output batch sizes of each transform in the flow is exactly 1, the difference in the departure and arrival times for a given event would be the traditional service time for that event.

Formalizing this definition, let $\theta_i \subseteq Q$ be the set of transforms in the path from transform t_i to the root transform t_1 (the entry transform for events in this flow), with $\theta_1 = \{\}$. Then, if B_j is the batch size of the transform t_j,

$$D_f = \sum_i D_i \prod_{j|t_j \in \theta_i} B_j \tag{3}$$

The exact execution order for a flow will depend on the broker implementation,

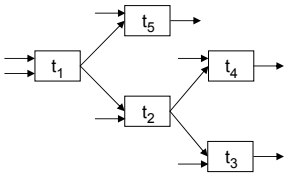

Fig. 2. A flow in a broker

the hardware and operating system scheduler where the flow would be deployed. However, all of these factors will only change the departure process distribution, but not the total number of computing cycles given by D_f.

Computing Service Time Means and Moments: Using equation 3 we can immediately estimate D_f by adding and multiplying the expected values of the individual transforms' service times (according to the flow structure using θ_i). If the service times and batch sizes are independently distributed, the variance can be computed using the Goodman's formula [12] as follows:

$$\sigma(X * Y) = E(x)^2\sigma(Y) + E(Y)^2\sigma(X) + \sigma(X)\sigma(Y)$$

The $E(D_f)$ and its variance $\sigma(D_f)$ are immediately useful in the $M/G/1$ and $G/G/1$ models as will be shown ahead.

Service Time Distribution: If the information about the transform service time distribution allows it, we can also derive the complete pdf for D_f. This can be used to determine a queueing model which gives a good approximation.

If the service times of the transforms in the flow can be considered independent, then one can compute $\phi_f(d_f)$, the probability density of flow service time D_f, by convolving the summands, using Laplace or Fourier transforms on the pdfs of the summands to convert to the frequency domain and multiplying the resulting functions. If the service times are dependent for some transforms, then the pdf of their sum can be found using the joint pdf of their service times.

For the product terms one can use the methods in [22] [11], namely the pdf of $V = XY$, where X and Y are two random variables, is

$$f_v(\nu) = \int_{-\infty}^{\infty} f_{X,Y}(x, \frac{\nu}{x})\frac{1}{|x|}dx$$

where $f_{X,Y}$ is the joint pdf. If X and Y are independent, the one can use a shortcut of finding the pdfs of $\ln X$ and $\ln Y$, and converting the product into a convolvable sum using the following:

$$\prod_{i=1}^{n} x_i = \exp \sum_{i=1}^{n} \ln x_i \tag{4}$$

where x_i are random variables for $1 \leq i \leq n$, and then performing a transformation of variables to get the resulting pdf of the product.

4.2 Departure Distribution

Note that, for the messages leaving broker, the delay across the communication link is significantly larger than the offsets caused by departure distribution. Thus, when messages leaving a broker arrive at another broker,what matters most is their arrival rate rather than the arrival distribution. So, we focus on capturing the departure rate at a broker.

If we have the pdf of the service time distribution of a flow as derived in Section 4.1, we can construct a corresponding density function for the departure process. Let $a_f(t)$ be the pdf of inter-arrival times T_a for some flow f in broker β. Let $S_f(t)$ be the cdf of the service time of the flow. The cdf $CT_f(t)$ of the inter-departure time random variable T_f can be estimated [13] as:

$$CT_f(t) = \rho S_f(t) + (1-\rho)\int_0^t S_f(t-u)a(u)du$$

where $\rho = \sum_f(\frac{\lambda_f}{\mu_f})$, the broker utilization due to all flows. For each transform t_i whose outputs leave the broker, that transform will emit a batch of events of size X_i, each time an event arrives at the root of its flow, where:

$$X(i) = \prod_{j|t_j \in \theta_i} B_j \tag{5}$$

Therefore, the outgoing event rate N_i (in events per unit time) at transform t_i is $N_i = \frac{X_i}{T_f}$, whose distribution ϕ_N we can be readily computed using the same log expansion approach as in equation (4).

If we do not have either the interarrival or the service time distributions, we can approximate the departures using:

$$c_d^2 = 1 + (1-\rho^2)(c_a^2 - 1) + \rho^2(\max(c_s^2, 0.2) - 1)$$

where c_d^2 is the coefficient of variance for the departure distribution, while the departure rate is the arrival rate $\lambda * X_i$. However, note that the approximation for c_d^2 is likely to be sensitive to the transform evaluation schedule on the broker.

5 Complete Broker Model

To model the complete broker with multiple incoming streams and multiple flows we propose using the aggregation / disaggregation approach due to Whitt [28]. The basic idea of the approach is to first aggregate the input streams into a single stream and pretend that all the transform flows behave as one server. Then, compute the marginal metrics for the individual flows from the combined result. The formula for aggregation applicable in our case is:

$$\hat{\mu} = \frac{\hat{\lambda}}{\sum_f(\frac{\lambda_f}{\mu_f})}; \quad \hat{c_s^2} = \frac{\hat{\mu^2}}{\hat{\lambda}}\left(\sum_f \frac{\lambda_f}{\mu_f^2}(c_{s_f}^2 + 1)\right) - 1 \tag{6}$$

where $\mu_f = 1/D_f$, the service rate for flow f and λ_f is its input rate. $c_{s_f}^2 \equiv \sigma(D_f)/E[D_f]^2$ is the squared coefficient of variance for flow service time.

If the combined input stream distribution is known to be Poisson, then one can directly use the Pollaczek-Khintchine(PK) formula [13] for M/G/1 using the service time and variance derived in the previous section. In this case, the aggregate $\hat{\lambda}$ is the sum of the expected values of the individual flow input rates. For other cases we have to assume a general distribution for arrivals, for which we use second Whitt's [28] formula:

$$\hat{c_a^2} = (1 - w) + w \left(\sum_f c_{a_f}^2 \frac{\lambda_f}{\hat{\lambda}} \right) \tag{7}$$

$$w = [1 + 4(1 - \rho)^2 (v - 1)]^{-1} \tag{8}$$

$$v = \left[\sum_f (\frac{\lambda_f}{\hat{\lambda}})^2 \right]^{-1} \tag{9}$$

where $c_{a_f}^2$ is the coefficient of variance for the flow f and $\rho = \hat{\lambda}/\hat{\mu}$.

We can now use these to compute the expected queue wait via a G/G/1 approximation due to Marchal[20]:

$$W_q = \left(\frac{\rho}{1 - \rho} \right) \left(\frac{\hat{c_a^2} + \hat{c_s^2}}{2} \right) \left(\frac{1}{\hat{\mu}} \right) \tag{10}$$

This can be used to compute the expected latency W_f of a flow f through a broker by adding its expected service time to its queueing delay:

$$W_f = W_q + 1/\mu_f \tag{11}$$

In summary, if the service time for each flow is predicted correctly we can use existing machinery for estimating system performance. For the delay across network links, we model it in the standard way in terms of the batch sizes of all the operators communicating over that link and the available bandwidth on the link. This means that the first few flows placed on a link have to account for additional flows that may come later.

6 Placement for end-to-end delay minimization

In this section, we consider the placement for end-to-end delay minimization. We consider a scenario in which a transform graph has to be deployed on a topology graph and the locations of both the producers and the consumers is known. The goal is to minimize the average time required for an event originating at a producer to reach its destined consumer. Both the service time for the transforms on the brokers they are placed in, and the latency across communication links have to be taken into account in the end-to-end delay. Let st_{ob} denote the actual

service time of operator o on a broker b. Let G denote the topology graph and Q denote the transform graph. Suppose π denotes the assignment of the trasforms to the brokers, the end-to-end delay is given by

$$\sum_{b \in V} \sum_{o \in O: \pi(o)=b} s_{ob} + \sum_{(o,o') \in Q} bs(o_i) \cdot PL(\pi(o), \pi(o'))$$

where $bs(o_i)$ denotes the actual batch size of o_i and $PL(b, b')$ denotes the actual delay on the shortest path between brokers b and b'. We would like to compute an assignment π with minimum end-to-end delay. As it is not possible to obtain actual statistics even before placement, we optimize with respect to the performance models developed in the previous sections.

Our design of the placement algorithm is based on the following simple observation. Consider a transform graph of depath two, i.e, the producers and the consumers are separated by a relational transform. In such a case, the producers and the consumers are the fixed points of the placement. The middle level of relational transforms have to be placed in the network to minimize the total latency, i.e, the sum of processing times, and network delays. We consider a formulation of this special case to motivate our algorithm design.

Let the transform graph be Q. As always, let P denote the set of producers, and C denote the set of consumers. Let M denote the set of operators in the middle layer. It is easy to see that the placement problem is essentially that of deciding the set of operators to be installed at every broker in the network. Let $s_{o,b}$ denote the service time of operator o on a broker b. Let π denote the assignment function of operators to the brokers. Let $d_{c,b}$ denote the delay involved in a fixed point $c \in P \cup C$ either sending an event update to or receiving an event update from, an operator installed at broker b. Let $\phi(c, b)$ denote a boolean function which is true if there is an operator installed at b which the fixed point c needs to contact for update. The placement problem is to compute an assignment π which minimizes:

$$\sum_{b \in V} \sum_{o: \pi(o)=b} s_{o,b} + \sum_{c \in P \cup C} \sum_{b \in V} \phi(c, b) d_{c,b}$$

The first component of the above objective function can be thought of as the cost of setting up operator "facilities" in the network and the second component can be thought of as the cost of fixed points reaching the facilities. In fact, this formulation (overlooking minor details), is that of the facility location problem discussed in [24]. The case of general transform graph can similarly be modeled as a generalization of the above problem called Facility location with Hierarchical costs [26]. It is well known that local search is perhaps the best approximation algorithm for facility location problem involving capacities [17, 4, 30, 10, 26]. This is the motivation for us to choose local search heuristic for the placement problem. Instead of the complex, exponential sized neighborhoods considered in classical approximation algorithms, we consider simple local operations for fast evaluations. Our experiments show that on practical instances, we still obtain high-quality placements.

```
1. For every operator o ∈ Q \ FX do,
     let b be a randomly chosen broker in V. π(o) = b.
2. improveCond = true;
3. While improveCond do,
     contCond = true;
     for all b ∈ V and while (contCond = true) do,
       Lst = list of all operators assigned to b
         for all o in Lst and while (contCond = true) do,
           π' = π; π'(o) = a randomly chosen neighbor of b;
           If cost(π') < cost(π)
             π = π'; contCond = fasle;
           endIf
         endfor
     endfor
   endWhile
4. return π.
```

Fig. 3. Placement Algorithm

We consider the simple local search heuristic for placement described in Figure 3. The function *cost* in this algorithm includes the total service time of the operators and the communication delays. It conducts a simple local search on the search space of all possible assignments. We include elements of randomness in the starting point as well as the improvement neighbor for better exploration of the search space. It is a standard technique in local search and can be modified to include restarts and jump-outs of local minima. The approximation guarantees of [30, 10, 26] give plausible explanations for the high quality assignments computed by our algorithm.

7 Placement for Dynamic Subscriptions

The SMILE platform supports the following scenario. Consider a popular query Q_{pop} which a service provider would like to install in the system to which the subscriptions can be attached at any location of the network on a dynamic basis. In general, the service provider would like to support many useful views represented in the form of a large transform graph and might prefer to place it on a small subgraph of his network. Thus, when the assignment is being computed, only the list of producers is available. In such a scenario, no meaningful estimate of end-to-end delay can be formulated. So, we consider placing the transform graph such that, the load on the brokers is roughly balanced and the communication across the network to evaluate the query ignoring the future subscriptions is minimized. In this case, we make an assumption that the number of operators in the transform graph is more than the number of brokers in the topology graph denoted by k. Dynamic subscriptions can be used to support applications like portfolio tracking of a large number of portfolios.

We relate the placement for dynamic subscription to that of embedding of a *minimum cost k-balanced graph partition* [3] of the transform graph on to the topology graph. The load balancing requirement implies that we need to compute a k-balanced partitioning of the trasnform graph. Consider two operators o_i and o_j which are connected by an edge in the transform graph and which are placed in two different brokers $\pi(o_i)$ and $\pi(o_j)$. The communication load on the network due to embedding of this edge on the topology graph is given by $\mathbf{exp\_bs}[o_i] \cdot dist_G(\pi(o_i), \pi(o_j)$ where $dist_G$ denotes the distance on the topology graph. The optimal load balanced placement is that embedding of a k-balanced partitioning whose load on the topology graph is minimum. We adapt the popular local search based heuristic for the balanced graph partitioning algorithm considered in [16, 9] to compute locally minimum k-balanced partition embeddings. It is easy to see that every iteration can be implemented very efficiently when the local operation is that of exchanging the partitions of the end-points of every cut edge (this is necessary to maintain k-balancedness). Thus, we start with a random k-balanced partition. At each step, we consider local moves of exchanging the partitions of end-points of every cut-edge. If there is an exchange which improves the quality, we make that move. We terminate no such exchange improves the quality of the embedded partition.

8 Experimental Results

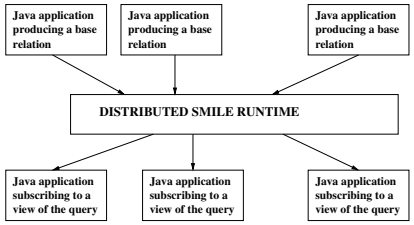

Fig. 4. Experimental Environment

In this section, we present the experimental evaluation of our modeling and algorithms on the SMILE system. Our experimental set-up is as shown in Figure 4. A set of SQL queries required for the experiments are compiled and deployed on the distributed runtime. Java client applications are used to generate streams corresponding to different base relations in the deployed system. Similarly, Java client applications are used to subscribe to the different views computed by the query deployment. These client applications also contain the instrumentation to record time stamps for different events in the system. As the client applications are external to the SMILE system, careful calibration is required to ensure reasonable accuracy of our experimental evaluation. We begin

by presenting a series of calibration steps to account for the interaction of the client applications with the SMILE runtime.

The client application needs to establish connection with the runtime and maitain the connection throughout its execution. We calibrate this interaction by creating a query with empty transform, i.e, the output view is same as the input stream. We install the empty transform on a single broker and execute a client program which generates a stream and subscribes to the output of the empty transform. Our experiments showed that, per message, this interaction costs an average 0.241 milliseconds with a standard deviation of 0.006.

When multiple producers are deployed in the system and several views subscribed to, a significant cost is incurred in the execution of the corresponding client applications. To calibrate the delay thus incurred, we create a query which has inputs from multiple streams and the output is a set of views which are replicas of the input streams. We observe that the overhead depends on the number of streams and views (i.e, the number of client applications). We measure the overhead for n base relations for different values of n. Later, when we conduct experiment with a real query with x streams and y subscribed views, we account for the overhead using the measurements for this dummy experiment with $x + y/2$ base relations. The table indicating the overhead is as follows.

	5 streams	10 streams	15 streams	20 streams
Delay (in ms per msg)	0.37	0.54	0.63	0.76

Fig. 5. Delay observed because different number of client applications

Finally, we have to account for the overheads incurred due to the fact that the Java application is distributed. Streams can be published from different brokers and the views can be subscribed to at different brokers. We calibrate this delay by again considering the empty transform. This time, we deploy the empty transform query as follows. The stream is produced at the first broker and the view is subscribed to, at the second broker. Again, over multiple long streams with different arrival rates, we observed an average delay of 5.346 milliseconds with a standard deviation of 0.178.

We now present a set of experiments to validate our modeling of the performance. The goal of the experiments is to show that the predicted performance of out models is close to the experimentally observed performance. We present all the experimental measurements after applying the corrections as indicated by the calibration carried out above. We first measure the performance of simple, one transform networks on a single broker. Specifically, we measure the performance of the SELECT, WINDOW, and JOIN transforms. For each of these, we run experiments with uniform arrival rates, random arrival rates, and poisson arrival rates. The numbers we present are the average over these distributions as they did not deviate significantly in our setup. Figure 6 shows the performance measures for these transforms. We used a many-to-many join in which

roughly half the messages of the two streams joined with each other, thus keeping transform extremely busy.

	SELECT	WINDOW	1:1 JOIN	Many to Many JOIN
Time	0.035	0.063	0.054	0.285

Fig. 6. Performance Numbers of different operators

We now consider some queries, whose transformation network we know apriori. Specifically, we consider two queries, one with depth three and other with depth five (shown in Fig. 9). We measure the end-to-end latency for these queries on one broker. We then compute the predicted end-to-end delay by the analytical model. Our experiments show that the predicted delay is close to the observed delay. Observe that the long delay in case of Q2 is due to the presence of many-to-many joins.

	Q1	Q2
Predicted Delay (per event)	0.485	0.96
Observed Delay (per event)	0.569	1.13

Fig. 7. Experimental measurement of composed queries

We now present the performance of our placement algorithm and compare it with a greedy heuristic. We have experimented with queries which have 15 transforms and topologies with 3 and 4 brokers. Here, we present the experimental evaluation of the algorithms for the dynamic subscription with the restriction that the load should be evenly distributed. We compare our algorithm with the following intuitive greedy algorithm. It considers every producer to consumer (subscribed view) pair and places it greedily in a load balancing manner (i.e, at every step, load is roughly balanced). We present experimental numbers when the messages are generated at different rates (Figure 8). We observed that, sometimes, the high message rates resulted in slightly higher delays than expected. The performance of an algorithm suffers a lot when the output of a transform which generates a large number of messages is input to a transform placed on a different broker. This happened more frequently with the greedy when the number of brokers was 3. However, a consistent trend is that, our algorithm always outperforms the greedy.

To qualitatively illutstrate why our placement performs better than greedy, we present the placement computed for the transform network Q shown in Figure 9. In this figure Pr indicates a producer, S indicates a select operator, P indicates a project operator, W indicates a window operator, A indicates an aggregate operator, and J indicates a join operator. In particular, J1 and J2 are

	100 msgs/sec	10 msgs/sec	1 msg/sec
Greedy on 3 Brokers	36.329 ms	28.927 ms	25.41ms
Our Algorithm (sec. 7) on 3 brokers	11.42ms	9.81ms	6.832ms
Greedy on 4 Brokers	34.29ms	33.137ms	32.528ms
Our Algorithm (Sec. 7) on 4 brokers	28.821ms	23.204ms	21.376ms

Fig. 8. Experimental evaluation of greedy and our algorithm

many-to-many joins. On a four broker network, Figure 10 shows the placement computed by the greedy and our algorithm. In both the experiments, the producers were available where the corresponding transforms S1. . .S5 were placed. Suppose, we consider the number of edges that cross brokers as an indication of expected delay, Greedy has 11 crossing edges, our algorithm has only 5 crossing edges. As explained in Section 7, in the dynamic subscription, we would like to embed a balanced partitioning of the transform graph on the topology graph such that the size of cut implied by the embedding is minimum. This pictorial illustation shows that our algorithm seems to achieve such an objective.

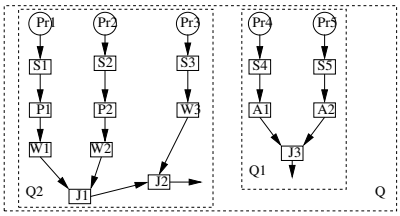

Fig. 9. Query Network

Placement	S1	S2	S3	S4	S5	P1	P2	W1	W2	W3	J1	J2	A1	A2	J3
Greedy	4	3	4	1	2	4	3	3	2	1	2	3	1	4	1
Our Algorithm	2	4	3	3	1	2	4	2	4	3	1	1	2	2	4

Fig. 10. Placement computed by Greedy and our algorithm

One of the limitations of our experimentation is the small size of the broker networks (and hence, moderate sized queries). In simulations with large task graphs and topology graphs, we have observed that our algorithms do well for both the static and dynamic subscriptions. In certain cases we output solutions which are order of magnitudes better than those produced by the greedy approach. In our future work, we aim to validate this by carrying out experiments

on large broker networks and compare the performance of our algorithm with those presented in [25, 2].

9 Conclusion and Future Work

In this paper, we considered issues related to the performance modeling and placing an operator tree on a distributed network to optimize efficiency measures such as latency. Experiments show that our performance modeling is reasonably accurate and local search based optimizations work well in practice. An important problem in distributed query processing is, how different scheduling policies inside a broker affect end-to-end latency and if they necessitate new algorithms. Prior work and our work address the problem of placement to optimize the efficiency of a single query. Although multiquery optimization is well studied in the traditional databses, it has not been addressed in the context of distributed stream processing. In our future work, we plan to extend the SMILE framework for multiquery optimization as well.

References

1. D. Abadi, D. Carney, U. Cetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *VLDB Journal*, 2003.
2. Y. Ahmad and U. Cetintemel. Network aware query processing for stream based applications. In *Proceedings of Very Large Data Bases (VLDB)*, 2004.
3. K. Andreev and H. Räcke. Balanced graph partitioning. In *ACM Symposium on Parallel Algorithms and Architectures(SPAA)*, pages 120–124, 2004.
4. V. Arya, N. Garg, R. Khandekar, V. Pandit, A. Meyerson, and K. Munagala. Local search heuristics for k-median, and facility location problems. *Siam Journal of Computing*, 33:544–562, 2004.
5. R. Avnur and J. Hellerstein. Eddies: continuously adaptive query processing. pages 261–272, 2000.
6. B. Babcock, M. Datar, and R. Motwani. Load shedding for aggregation queries over data streams. *20th International Conference on Data Engineering*, 2004.
7. S. Chandrasekaran and M. Franklin. Streaming queries over streaming data, 2002.
8. J. Chen, D. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: a scalable continuous query system for Internet databases. pages 379–390, 2000.
9. C. Fiduccia and R. Mattheyses. A linear time heuristic for improving network partitions. In *Proceedings of the 19th IEEE Design Automation Conference*, 1982.
10. N. Garg, R. Khandekar, and V. Pandit. Improved approximation for the universal facility location. In *Proceedings of the 16th Annual ACM-SIAM Symposium on Discrete Algorithms*, January 2005.
11. A. Glen, L. Leemis, and J. Drew. Computing the distribution of the product of two continuous random variables. *Computational Statistics and Data Analysis*, 2002.
12. L. Goodman. On the exact variance of products. *Journal of the American Statistical Association*, (55):708–713, 1960.
13. D. Gross and C. Harris. *Fundamentals of queueing theory*. Wiley and Sons, 3rd edition, 1998.

14. H. Hsiao, M. Chen, and P. Yu. On parallel execution of multiple pipelined hash joins. pages 185–196, 1994.
15. Y. Jin and R. Strom. Relational subscription middleware for internet-scale publish-subscribe. In *DEBS*, 2003.
16. B. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, 49(2):291–307, 1970.
17. M. Korupolu, C. Plaxton, and R. Rajaraman. Analysis of a local search heuristic for facility location problems. In *Proceedings of the 9th Annual ACM-SIAM Symposium on Discrete Algorithms*, 1998.
18. M. Lo, M. Chen, C. Ravishankar, and P. Yu. On optimal processor allocation to support pipelined hash joins. pages 69–78, 1993.
19. S. Madden, M. Shah, J. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, 2002.
20. W. Marchal. Some simpler bounds on the mean queueing time. *Operations Research*, 22:1083–1088, 1978.
21. R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, resource management, and approximation in a data stream management system. *Proc. of the 1st Biennial Conference on Innovative Data Systems Research*, 2003.
22. V. Rohtagi. *An introduction to probability theory and mathematical statistics*. Wiley Series in Probability and Statistics, 1976.
23. R.Strom. Extending a content based publish-subscribe system with relational subscriptions. Technical report, IBM Research Report, 2003.
24. D. Shmoys, C. Swamy, and R. Levi. Facility location with service installation cost. In *Proceedings of the 15th ACM-SIAM Symposium on Discrete Algorithms*, 2004.
25. U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proceedings of ACM Symposium on Principles of Database Systems (PODS)*, 2005.
26. Z. Svitkina and E. Tardos. Facility location with hierarchical facility costs. In *Proccedings of the 17th ACM-SIAM Symposium on Discrete Algorithms*, 2006.
27. D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD international conference on Management of data*, pages 321–330. ACM Press, 1992.
28. W. Whitt. The queuing network analyzer. *Bell Systems Technical Journal*, 66:2779–2813, 1983.
29. Y. Xing, S. Zdonik, and J. Hwang. Dynamic load distribution in the borealis stream processor. In *In Proceedings of Internation Conference on Data Engineering (ICDE)*, 2005.
30. J. Zhang, B. Chen, and Y. Ye. Multi-exchange local search algorithm for capacitated facility location problem. In *Proceedings of Integer Programming and Combinatorial Optimization (IPCO)*, 2004.