Integrating Pub-Sub and Stream Processing for Internet-Scale Monitoring

Olga Papaemmanouil, Uğur Çetintemel, John Jannotti
Department of Computer Science, Brown University
{olga, ugur, jj}@cs.brown.edu

Abstract

Existing stream processing systems are designed for clustered deployments, and cannot adequately meet the scalability and adaptivity requirements of Internet-scale monitoring applications. Furthermore, these systems commonly optimize for a specific QoS metric, which may limit their applicability to diverse applications and environments.

This paper presents XFlow, a generic distributed data collection, processing, and dissemination system that addresses these limitations. XFlow integrates a pub-sub model with data flows for stream processing. The underlying pub-sub model decouples sources and clients, as well as the processing operators, leading to a loosely-coupled architecture that can gracefully scale, adapt to churn in system membership and workload, and facilitate sophisticated optimizations.

We first provide an overview of XFlow's architecture. We then describe XFlow's optimization model that changes the placement and implementation of operators to meet application-specific performance goals and constraints. Finally, we demonstrate the flexibility and the effectiveness using real-world streams and experimental results obtained from our PlanetLab deployment.

1. Introduction

Motivation and goals: The confluence of ubiquitous, high-performance networking and increased availability of receptors that report physical or software events has led to the emergence of a new class of distributed, large-scale applications, which we collectively refer to as Internet-Scale Monitoring (ISM). An ISM application is a networked system that consists of large numbers of geographically dispersed entities: sources that generate large volumes of data streams and consumers that register large numbers of queries over these data streams, which are acquired, processed and then distributed in real-time to consumers. Example applications include planetary-scale sensor networks or "macroscopes" [3, 10], network performance and security monitoring [1, 2], massively multi-player online games, and feed-based information mash-ups [4]. ISM applications are currently implemented using custom, ad-hoc approaches that hinder their scalability and maintainability. Going forward, there is a need for general-purpose infrastructures that can effectively support a broad spectrum of ISM applications. We introduce such an infrastructure, named *XFlow*, which is an extensible, data stream acquisition, processing and distribution system.

Several goals and capabilities guided the design of XFlow:

- *Scalability*: ISM systems need to provide high network and workload scalability: they should gracefully deal with increasing geographically distributed system components and large numbers of simultaneous queries. To achieve this, the system should scale out and distribute its processing across multiple nodes.

- *Adaptivity*: ISM systems are expected to operate over the public Internet, with large numbers of unreliable receptors, on commodity machines, some of which may contribute their resources only on a transient basis (*e.g.*, in peer-to-peer settings). In general, they should adapt to churn, time-varying workload, and resource availability.

- *Extensibility*: While many ISM applications share common characteristics (*e.g.*, stream-based processing, overlay networks, membership management), they often exhibit diverse application-specific logic and performance requirements and constraints. For example, a camera-based surveillance application may need to perform feature extraction over MPEG streams, whereas a feed-oriented application may involve XPATH queries over RSS streams. Similarly, a network intrusion

application may have strict result latency requirements, whereas an environmental monitoring application running in a peer-to-peer setting may care more about fairness in bandwidth consumption. An ISM system should be easily customized to support application-specific data types, processing logic, performance expectations, and constraints.

Integrating pub-sub and stream processing: Distributed pub-sub systems [12, 14] address some of the above requirements. These systems effectively decouple sources and destinations over geography and time: publishers send data without knowing where and when the consumers will access them and consumers declaratively express their profiles and receive matching data without knowledge of specific sources. This loosely-coupled architecture allows for high scalability. and robustness in the presence of churn. Moreover, extensible pub-sub infrastructures have been proposed [25], allowing applications to define their own performance expectations. However, pub-sub systems have traditionally only supported *stateless* queries (*e.g.*, predicate-based filters over events) and simple data transformations.

Existing stream processing systems [5, 23], on the other hand, support complex *stateful* continuous queries, while distributed versions [6, 8, 26] allow for queries to be transparently distributed across multiple nodes to improve system scalability and availability. However, most current approaches implicitly assume a small number of sources and destinations, and, thus, do not provide mechanisms for "massively" distributing processing to take advantage of the large number of computing elements available in the network (e.g., via operator replication and partitioning). Furthermore, these systems focus on one or a few performance measures, which they optimize using hard-wired approaches, making it extremely difficult to effectively incorporate new optimization metrics. Implementing mechanisms for new metrics is a challenge when the built-in metric is not the "right one" for a given application. For example, in Borealis [6], each new metric addition took us almost as much as the last one to realize and test.

This paper describes XFlow, an extensible, highly-scalable and adaptive framework for distributing and optimizing stream-oriented continuous queries. XFlow relies on the pub-sub paradigm as the underlying communication model, a design decision we argue has the potential to meet the aforementioned design goals. This unique combination leads to a loosely-coupled, flexible stream-processing architecture that can scale, adapt and be easily customized for specific target applications. In XFlow, sources publish their data and clients subscribe their processing needs through queries consisting of stream-oriented operators [5, 23]. It is the responsibility of the system to collect and process the data and distribute the results to the clients, while meeting application-specific performance expectations.

One of the key features of XFlow is that it uses the underlying pub-sub model to also decouple the query operators. XFlow treats operators as regular stream sources and consumers—each operator subscribes to the stream generated by its upstream operator in the data flow and also publishes the stream it produces. This approach unifies and simplifies the overall system model while at the same time facilitating (i) the sharing of intermediate processing results and (ii) dynamic query modifications, such as adding, removing, migrating, and replicating operators.

From an operational perspective, XFlow creates, maintains and optimizes a pub-sub overlay network, given dynamic stream sources, clients with stream-oriented complex profiles, application-specific performance expectations, and available broker machines. The broker network consists of multiple, potentially overlapping overlay trees, in principle one for each source. XFlow uses a generic cost model, capable of expressing a variety of useful cost functions. At run-time, it iteratively modifies the placement and execution of the query operators to reduce the system cost and meet constraints. The optimizations are guided by a set of operator distribution operations, which include migration, replication and partition. XFlow employs low-overhead mechanisms to identify the operation with the highest benefit.

While XFlow leverages our previous work on extensible pub-sub systems [25] and stream processing [6], it also constitutes a whole new research direction. First, it uses a novel architecture that tightly integrates stream processing and pub-sub and consists of multiple,potentially overlapping overlay trees, for data collection, processing, and result dissemination. Second, it provides a *generic* cost model that can express a range of cost metrics and constraints which evaluate the efficiency of *user queries*. Third, in contrast with our previous work on pub-sub [25] that focused on network optimizations, XFlow employes a metric-independent distributed *query* optimization protocol that modifies where and how the query operators are executed through a set of migration, partition and replication operations. To the best of our knowledge, XFlow is the first system to allow this combination of query optimizations.

The rest of the paper is structured as follows. We describe the architecture of XFlow in Section 2. We

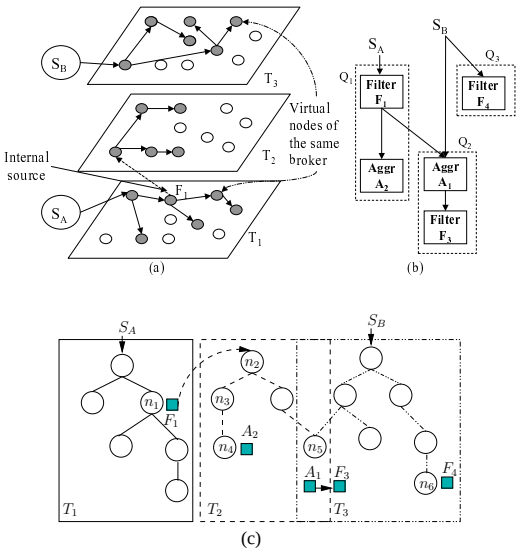


Figure 1. XFlow's system model.

introduce our cost model in Section 3 and describe our generic optimization framework in Section 4. We present our experimental results in Section 5 obtained by using real-world web feeds and deployment on PlanetLab and describe related work in Section 6.

2. System Model

XFlow consists of an overlay network of a large number of cooperating *brokers* (or nodes), providing stream routing and stream-based query processing services (*e.g.*, [5, 23]). *External data sources* reside outside system boundaries and publish data streams according to a well-defined global schema. Clients are also external and are eventual consumers of query results: they subscribe their interests expressed in terms of stream-oriented continuous queries on the global schema. Each external source and client has a proxy running on a broker, acting on behalf of its corresponding entity.

In the rest of the discussion, we use node and broker, interchangeably.

Pub-Sub model. XFlow relies on the pub-sub model as the underlying uniform mechanism for disseminating *all* data flows in the system. One implication is that each query operator publishes its output stream, while subscribing to its input stream(s). As operators also publish data, we refer to them as *internal sources*. Both external and internal sources are assigned system-wide unique identifiers and have well-defined schemas. The global schema is the union of external and internal schemas.

XFlow creates multiple overlay dissemination trees, potentially one for each internal or external source. Each source publishes its stream to an overlay tree which distributes it to the subscribed consumers. Hence, nodes hosting interested clients or operators must join the tree. The main advantage of using multiple trees is to enable better network utilization and reduce redundant transmissions by having clients become part of only those trees that publish relevant data [18, 24]. Figure 1(a) shows a network of three trees: T_1 and T_3 publish the external sources' streams S_A and S_B respectively, while T_2 publishes the output of the internal source F_1.

Source registration. When a source (internal or external) is first registered, it forwards its stream's schema to the registration service (referred to as *bootstrap node*). For an external source, the bootstrap is contacted through its proxy. The bootstrap picks a broker as the *root broker* for that stream based on its topological distance to the source, the available bandwidth of the broker and the expected data volume. This broker will be the root of the tree that disseminates the source's stream. Upon receipt of the first subscription for the data stream, the root broker will activate the data source, which in turn starts publishing its data streams. The functionality of the bootstrap can be easily distributed across multiple nodes to improve the availability and scalability.

Query registration. Clients also pick one XFlow node to host their proxy. To subscribe, the client's host contacts the bootstrap and requests a list of the streams published by both external and internal sources. Users can inspect the streams' schemas and define their queries on any collection of external or internal sources. This allows for intermediate results to be shared by multiple queries. For example, in Figure 1(b) query Q_2 uses the output of operator F_1 of query Q_1. XFlow currently requires the user to browse the existing query network to manually identify common sub-queries, similar to [4, 6], although it can also incorporate techniques for automatic detection of sharable computations [20].

The bootstrap also informs the client's host about the trees publishing its query's inputs. The host node joins these trees, creating a new *instance* for each tree, referred to as a *virtual node*. Each virtual node connects randomly to one node in a tree and requests the query's input streams published by that tree. We refer to this request as the *profile* of that virtual node and is extracted from the selection predicates of the query. This profile is forwarded upstream to the root, creating a reverse routing path to the virtual node. Using the routing tree created, a virtual node can route

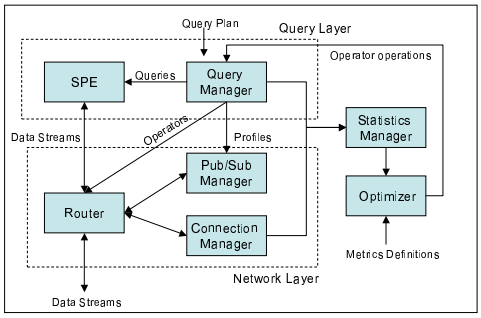


Figure 2. XFlow node architecture.

streams published through this tree only to its children interested in receiving them. Note that topology-based optimizations of pub-sub routing trees were addressed in XPORT [25].

Stream processing model. User profiles are expressed as directed, acyclic data-flow graphs of stream-oriented operators (*e.g.*, [5]), operating over the global schema. XFlow has a built-in set of standard windowed operators (filters, unions, aggregates, joins) and also allows for arbitrary user-defined functions to be linked as operators. An example of three simple queries is shown in Figure 1(b). When a client subscribes by forwarding its query to a node, the operators of the query get subscribed to their input streams. This is done along the upstream to downstream direction, in an order that is consistent with a topological sort of the operators in the query plan.

While the external source and client proxies are "pinned" to the brokers with which they first registered, the query operators are free to roam. All operators of a given query are initially assigned to the same broker; however, as we describe below, operators may be relocated, partitioned or replicated over time as part of the optimization process.

Figure 1(b) and (c) shows three queries and a possible distribution of their operators across the network. Q_1's operators are distributed across two trees. F_1's host, n_1, joins tree T_1, to receive stream S_A, and publishes its output to tree T_2. Hence, n_4, the host of A_2, joins T_2. Node n_5, the host of Q_2, joins also T_2. Moreover, it joins T_3 and subscribes to stream S_B. Finally n_6, hosting F_4, joins tree T_3.

2.1 Node architecture.

The architecture of an XFlow node is shown in Figure 2 and reveals how we integrated the pub-sub model with a stream processing system (SPE). Query plans are parsed by the *query manager* and sent to the SPE for execution. Profiles are extracted and pushed to the *pub/sub manager* which contacts the bootstrap and requests the trees publishing the streams specified by the profile. Nodes join trees through the *connection manager* that maintains the connections with each node's parents and children. The *router* delivers input streams to the SPE and output streams to clients or downstream nodes in the trees. It also routes any maintenance data required by the connection manager as well as any profile updates sent by the pub/sub manager. The *optimizer* is the extensible component in our architecture. Application designers can specify the own performance criterion (see details in Section 3), and XFlow customizes its functionality based on these metrics. Using data on the query and the overlay network, obtained from the *statistics manager*, it identifies operations on the queries that improve the cost metrics and sends any query changes to the query manager.

2.2 Tree management.

In principle, XFlow creates one tree per each source (internal or external). This basic approach may seem unwieldy, as each source and operator will have its own tree. In practice, we employ simple techniques to control the number of trees. To control the number of trees, first, we allow the definition of super-operators that combine multiple connected operators in a single unit. A single tree is created for each super-operator, as the constituent operators will not publish data. Moreover, we expect that many overlay trees will be entirely local to a single node at the network level; for example, in the case when an operator and its upstream parent operator are located on the same node.

Second, sources can be assigned to already existing trees, reducing the number of trees. For example, streams requested by highly overlapping sets of subscribers can be published through the same tree, while streams can be periodically reassigned across root brokers, adapting to membership changes. The problem of effectively grouping data sources and mapping them to a given set of trees has been studied in [7, 24] and is beyond the scope of this paper. For simplicity of exposition, we assume that this grouping is already done and use source to mean collections of sources.

Finally, one network-level optimization we perform is to have any two pairs of brokers communicate through only a single TCP connection, independently of the number trees in which they are neighbors. These connections create an overlay mesh on top of which "logical" trees are built, that share the overlay links of this mesh. Hence, the cost of creating a tree is small, since nodes will set up connections with their peers only the first time they need to connect on some tree.

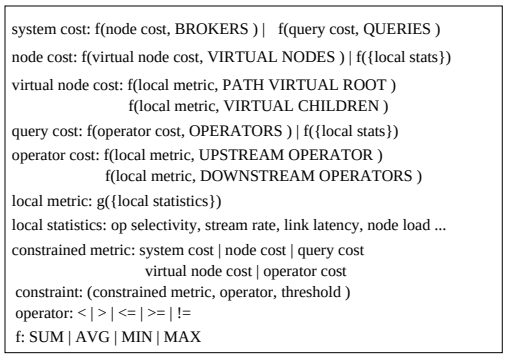


Figure 3. Cost metric grammar.

We also studied the maintenance and optimization overhead of multiple trees in Section 5. The results reveal that the traffic required for optimizing XFlow is quite low, even when multiple trees are created. Moreover, our study shows that multiple trees can achieve better distribution of the processing load, as they can better utilize the available brokers.

Finally, as part of our future work we plan on exploring tree-based optimization rules that consider merging multiples trees in order to improve the system performance. A simple heuristic could be merging dissemination trees that involve the same set of nodes. Same tree memberships could be easily identified by the roots with the exchange of bloom filters.

3. Extensible cost-model

The cost-model of XFlow leverages that of XPORT [25], an extensible single-tree dissemination system that allows application designers to specify the optimization metrics. XPORT focuses on *node-oriented* metrics that evaluate the overhead the dissemination process incurs on each node.

XFlow differs from XPORT's basic cost model along three non-trivial dimensions. First, it incorporates high-level *query-oriented* metrics that can express the overhead and efficiency of user queries. Second it allows node-oriented metrics to evaluate also the overhead the query processing imposes on each node. Finally, it allows both query-oriented and node-oriented metrics to be defined and evaluated across a broker network of *multiple* overlay trees, whereas XPORT limits itself to a single tree. In the rest of the section, we describe XFlow's grammar in detail (Figure 3) and provide examples.

3.1. Cost metric model

XFlow nodes maintain a set of built-in *local metrics* that measure various statistics of the network (*e.g.*, link latency, bandwidth) and the local operators (*e.g.*, input/output rates, selectivity, processing costs). Application designers can also combine these statistics and measure their own local metrics for the nodes, *e.g.*, the processing load of a node. Based on these local metrics, optimization metrics can be defined.

Query-oriented metrics. These metrics express performance criteria for the operators and queries in our system. Initially, we define the cost of an operator by aggregating local metrics collected from neighbors of the current operator's location. These neighbors are the nodes connecting the operator to the location of (i) its upstream operator in a query plan (referred to as *upstream aggregation*) or (ii) its immediate downstream operator (*downstream aggregation*). If the operator has multiple upstream or downstream operators across multiple query plans, we define the cost of each one independently and average the costs to compute the final cost of the operator. Note that nodes hosting operators are responsible for collecting the local metrics from their neighbors. We aggregate the cost of all the operators of a query to define the query cost, while the system cost is computed by aggregating the costs of all queries.

An example metric for upstream aggregation is the operator latency. In this case, an operator's latency, *op_l* can be computed as the latency to its upstream operator, l, plus its processing latency, *proc_l*. The query latency, *q_l*, is then the sum of the costs of all its constituent operators. Thus, for the average query latency we have:

```
op_l= sum(l, UPSTREAM OPERATOR) + proc_l
q_l= sum(op_l, OPERATORS)
system cost= avg(q_l, QUERIES)
```

In this case, the processing latency of an operator as a local metric and each virtual node is customized to measure the latency to its parent and collect the link latency of the nodes on the path to its upstream operator in the query plan. For example, for the query distribution in Figure 1, n_4, the host of A_2, measures its latency to n_3 and collects the latency of links (n_3, n_2) and (n_2, n_1) from n_3 and n_2 respectively.

Node-oriented metrics. For node-oriented metrics, we first define the cost of a *virtual* node, *i.e.*, its cost in a single tree by aggregating local metrics from the virtual node's children or path to the root of the tree. We aggregate the cost of all its virtual nodes to define the cost of a node, *i.e.*, its cost across all trees. Finally, the global system performance is an aggregation of the cost of all nodes.

An example metric is the average outgoing bandwidth across nodes. The cost of a virtual node, *v_out*,

is the data it forwards to its children in the tree. The node cost, *n_out*, is then the sum of the data each virtual node sends, plus the data sent to its clients or published to another tree, *i.e.*, the output rate, *op_out*, of its local operators. *op_out* as a local metric, given by the product $r \times s \times k$, where r is the incoming stream rate (tuples in time unit), s the selectivity of the operator and k the average tuple size.

```
v_out= sum(in_rate, VIRTUAL CHILDREN)
n_out= sum(v_out, VIRTUAL NODES)
           +sum(op_out, OPERATORS)
system cost = avg(n_out, BROKERS)
```

Our cost model allows for some aggregation steps to be omitted and node and query oriented metrics to be combined. An single aggregation metric example is the maximum processing load across nodes. The local metric of each node is its processing load, *i.e.*, the sum of its local operators' load. We define the load of an operator (in % of CPU cycles per time unit) as the product $r \times s \times c$, where r is its input rate, s its selectivity and c its cost (% of CPU cycles per tuple). Hence, the maximum load is defined as:

```
node load= sum(load, OPERATORS)
system cost= max(node load, BROKERS)
```

Constraints. Applications can express constraints on performance metrics, using the same cost model: they can define cost metrics for operators, queries or nodes and specify bounds on them, *e.g.*, on the number of children of each virtual node, the maximum processing load across brokers, the number of replicas per operator or the maximum query latency. XFlow customizes its optimization to respect these constraints.

We inherit from XPORT two types of aggregation functions for defining the operator, query and system cost (or virtual, node and system cost): (i) *bottleneck* (MIN, MAX) and (ii) *additive* (SUM, AVG). This categorization is based on how our optimization framework handles the different combinations of these categories (see Section 4.2). These functions permit applications to define a variety of commonly used system cost measures, like maximum load, bandwidth consumption, maximum query latency, average query load, etc.

4. Distributed optimization

XFlow distributes the query operators, aiming to minimize the global system cost. It continuously refines the placement of the operators and their execution, through migration, replication and partition operations. We collectively refer to these as *operator distribution operations*.

Our optimization framework strives to meet three requirements. First, it must be *scalable* in terms of the number of operators and the number of brokers in our network. Second, our operations should be *efficient* and *adaptive* to time-varying network or workload conditions. Finally, we require a *metric-independent* model that uniformly applies to and handles a variety of application-defined cost metrics.

To address these challenges, we designed a general, decentralized framework that does not rely on global information and has low communication overhead. Our protocol continuously considers *localized distribution operations* on the running operators. Each node maintains information about its own "neighborhood" and periodically attempts to distribute its local operators across the nodes within the scope of this neighborhood. An advantage of this approach is that non-overlapping neighborhoods can be optimized locally and concurrently. XFlow exploits its structured cost-definition model and the known semantics of the aggregation functions to derive generic properties and equations to quantify the benefits of any candidate distribution operation. In the rest of the section, we describe our operator operations and then focus on our cost-based model.

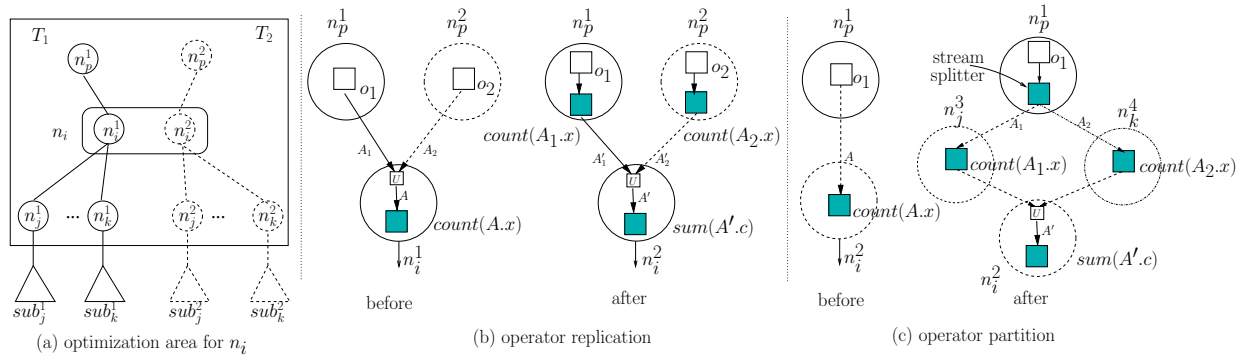
4.1. Operator distribution

XFlow nodes distribute their operators across their neighbors in the overlay trees they participate it, *i.e.*, their *optimization area*. For a node n_i, this area includes the set of nodes in a k-level subtree rooted at each virtual node of n_i, where k is a system parameter. In our current implementation, each such subtree has at most three levels, including the virtual node of n_i, its parent and its children in the tree. We denote as n_i^k a virtual node of node n_i in tree T_k, and Figure 4 shows (a) the optimization area of n_i which includes the nodes of the subtrees in two trees, T_1 and T_2.

We distribute the operators with two types of operations: (i) *operator placement operations*, which migrate operators to alternative locations and (ii) *operator execution operations*, which change the implementation of operators by replicating or partitioning them across multiple nodes.

4.1.1. Operator Placement and Migration

Operator placement modifies where an operator is executed. For a given operator, we reduce the space of the candidate network locations by considering only nodes in the optimization area of its current host. Although at each step we consider a small number of alternative locations, we can gradually migrate operators



(a) optimization area for n_i (b) operator replication (c) operator partition

Figure 4. Optimization area of n_i and operator replication and partition.

to a different network area than that of their original hosts. As an example, if improving the query latency is our performance goal, operators are continuously placed one level higher in the tree, *i.e.*, closer to their upstream operators, as that would reduce the network latency of their input tuples. Moreover, if an application aims to reduce bandwidth consumption, operators with selectivity less than one (*e.g.*, filters) migrate closer to their upstream operators, while if the selectivity is more than one (*e.g.*, joins), they are pushed closer to the downstream operators. This process reduces the overhead of forwarding tuples to the network.

We exploit our underlying pub-sub model to dynamically reroute data flows to the new locations of the operators. More specifically, the new host of the operator subscribes to the inputs of the operator by joining the trees that publish these streams (if not already part of the them). If the operator is also an internal source, then the new host publishes the operator's output to the root of the corresponding tree.

4.1.2. Operator Replication and Partitioning

The primary goal of replication and partitioning is to parallelize operator execution to utilize idle available resource in the broker network.

Replication. Replication is applied to operators that subscribe to multiple trees. Instead of collecting all streams and executing the operator on a single node, replication exploits processing resources of multiple trees. The goal is to process each input stream independently on each tree and combining the results to construct the final output. To implement this, a replica is created for each of the trees the operator receives input from. Each replica will receive one of the inputs of the original operator. To guarantee correctness, a *final operator*, receives the outputs of all replicas and produces the final result. The pub-sub network easily routes stream flows from the replicas: they publish their outputs through a tree, and the final operator's host subscribes to them by joining these trees.

Operator	Replica Operator	Final Operator
count	count	sum
filter	filter	union
rename	rename	union
copy	copy	union
split	split	union
truncate	truncate	union
union	union	union
unique	unique	unique

Table 1. Operator replication/partition implementation.

Depending on the semantics of the replicated operator, we need to use a different implementation for the final operator. For example, for filters we merge the already filtered outputs of our replicas using a union operator, while for count operators, the final operator sums the replicas' results.

In our current implementation we used feed processing operators similar to the operator set in Yahoo!Pipes [4]. The set of operators along with the operators used for their replicas and final operators are shown in Table 1.

Replication for the count operator is shown in Figure 4(b). A union operator precedes the count merging streams A_1 and A_2 into stream A. We count the attribute x, thus, after replication, we apply the count independently on A_1 and A_2 and carry the results in the c attribute of their replicas' outputs A_1' and A_2'. We sum the c value from both streams (after we merge them) in order to get the final result.

The benefits of replication primarily depend on the location of the replicas. For example, if improving the query latency is our objective, then placing the replicas in Figure 4(b) on the same node as the original operator

will not decrease latency. To increase the benefit, nodes can migrate the replicas inside their optimization area. Since each replica corresponds to a different tree, a node can place the replica on its neighbors on this tree, *i.e.*, its parent, one of its children or siblings. Note that we do not require *all* replicas of an operator to be migrated, only the ones that can further improve performance.

Replication can also increase the opportunities for optimization. Replicas are single input operators that can be handled independently by our framework. Thus, they are more flexible to migrate as they affect fewer nodes and trees than operators with multiple inputs and multiple subscriptions. For example, each replica can be migrated closer to its own data source, reducing the network latency of queries.

Operator Partition. Partition is applied on the input stream of an operator. It splits the stream into two flows and each sub-flow is processed by a different replica of the operator, thereby exploiting data parallelism. The output of the replicas are processed by a final operator which produces the final results. Its type depends on the original operator, similarly to the case of replication. Similarly, replicas can be migrated inside the optimization area. The hosts of the replicas join the trees providing their inputs and the pub-sub network routes the output streams published by the replicas to the location of the final operator.

Partition is shown in Figure 4(c) for the count operator. Node n_p^1 splits stream A and publishes two sub-streams A_1 and A_2. These streams are routed to the two replicas and their outputs are merged and processed by the final operator. For this example, we assume that node n_i participates in trees T_3 and T_4 and it places the replicas on some instances of its neighbors n_j and n_k on these trees.

Partition is used to reduce the processing requirements of the original host and move a portion of the processing to another node. Each replica is an independent operator that processes half of the initial stream. Moreover, assuming a selectivity less than one for the replicas, the incoming rate of the final operator is also lower than that of the initial stream, thus the processing overhead of its host node is decreased.

4.2. Evaluating cost improvement

Our optimization framework includes a generic cost model that quantifies the expected benefit of an operation with low overhead. We exploit the fact that our localized operations affect only a subset of the nodes and operators, and we evaluate the impact on the cost metrics of only the affected entities. For the purposes of illustration, we will describe our approach with respect to the query-oriented metrics and the SUM and MIN aggregation functions. Similar results can be obtained for the node-oriented metrics and the AVG and MAX functions in a straightforward manner.

Symbol	Definition
l_i	local metric of n_i
oc_i	cost of operator o_i
qc_i	cost of query q_i
O_i	operators executed on n_i
Z_i	the set of queries including operators in O_i
E_i	the set of queries including operator o_i
D_i	dependent operators of node n_i
G_i	dependent queries of node n_i
F_α	affected nodes from operation α
b_α	benefit of operation α

Table 2. Model Terminology

We start by providing the dependencies among nodes, operators and queries. We assume a set of queries Q. Each query $q_i \in Q$ consists of a set of operators and let O be the total set of operators. Let also E_i be the set of queries that include operator $o_i \in O$. We denote the local metric of node n_i as l_i. Any operation that involves this node could affect its local metric (*e.g.*, adding an operator increases its processing load). We denote such changes as $\Delta(n_i : l_i \rightarrow l_i + \delta)$. The cost of an operator $o_i \in O$, oc_i, is defined by aggregating the local metrics of some nodes (*e.g.*, latency of the nodes to the upstream operator). This cost could change due to changes on the local metrics (*e.g.*, increase on the link latency) or changes on the set of nodes (*e.g.*, placing the operator on another node.) The following definition identifies which non-local operators and queries (that do not reside on this node) may be affected by a change on a node's local metric.

Definition 1 (DEPENDENT OPERATORS). *The* dependent operators *of node n_i is the set of operators, D_i, whose cost metrics depend on n_i's local metric, l_i:*

1. *If the cost of an operator is defined as an upstream aggregation, then D_i is the set of the first operators reached by n_i's paths to the leaves of every tree n_i participates and every tree in which it publishes data.*
2. *If the cost of an operator is defined as a downstream aggregation, then D_i is the set of first operators reached by n_i's path to the root of every tree n_i participates.*

Definition 2 (DEPENDENT QUERIES). *The* dependent queries *of n_i is the set of queries, G_i, that include at least one dependent operator of n_i. In particular: $G_i = \cap_{o_j \in D_i} E_j$.*

Let us consider the query plan and its distribution in Figure 5, where the cost of an operator is its latency (*i.e.*, upstream aggregation). Then $D_2 = \{o_4\}$,

as changes on the latency between n_2 and n_3, affects the output latency of o_4.

Each operation α affects a set of nodes F_α: migration has an impact on the origin and destination nodes, while replication and partition affect the origin node and the nodes where the replicas are placed. Changes on a node $n_i \in F_\alpha$ could change the cost of all dependent queries as well as the cost of queries that include any operator executed on these nodes. Let O_i be the set of operators executed on n_i and Z_i the set of queries including an operator in O_i. XFlow does not evaluate the new cost of every dependent query, but instead quantifies the impact of their cost changes on the global system cost, expressed by the *query dependence cost*.

Definition 3 (QUERY DEPENDENCE COST). *The* query dependence cost, $c(G_i)$, *of a query dependent set G_i, is the aggregation of the cost of every query $q_j \in G_i$, using the aggregation function that defines the system cost.*

For example, if the query cost and the system cost are defined based on the SUM function (*i.e.*, we aggregate operators' and queries' cost with the SUM function), then:

$$c(G_i) = \sum_{q_m \in G_i} qc_m = \sum_{o_j \in D_i} \sum_{m \in E_j} qc_m. \qquad (1)$$

Our cost model tries to estimate the change on the query dependence cost. In what follows, we illustrate the detail of this approach. The following property identifies the effect of an operation on the system cost, based on the query dependence cost.

System cost-effect property. *Assume the system cost is defined based on the* SUM *function. Given an operator distribution operation α, the expected change of the system's performance is the following*:

$$b_\alpha = \sum_{n_i \in F_\alpha} \Big(\sum_{q_m \in Z_i} \Delta qc_m + \Delta c(G_i) \Big). \qquad (2)$$

If the system cost is defined based on the MIN *function, then:*

$$b_\alpha = \min_{n_i \in F_\alpha} \Big\{ \min_{q_m \in Z_i} \{qc_m + \Delta qc_m\}, c(G_i) + \Delta c(G_i), \min_{q_m \notin Z_i, q_m \notin G_i} \{qc_m\} \Big\} - c \qquad (3)$$

where c is the current system cost.

In the following section we discuss how we can evaluate the above equations.

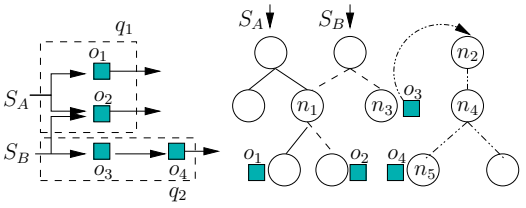


Figure 5. Example of operator distribution.

4.2.1. Query cost changes

We begin this section by evaluating the change on the dependent operators of a node and continue with the derivation of the change on query costs and the query dependence cost. We first focus on the case where the operator cost is defined based on the MIN function (*i.e.*, we aggregate local metrics of neighbors using the MIN function). In this case, whether a node can affect an operator's cost depends on the rest of the nodes on the path to the upstream (or downstream operator). We capture this in the following definition.

Definition 4 (CRITICAL VALUE). *Let n_i be a node. If the cost of operators is defined based on the* MIN *function, then the* critical value *of a dependent operator $o_j \in D_i$, w.r.t. n_i, $h_i(o_j)$, is the minimum local metric of all nodes on the path between n_i and the current location of o_j.*

To explain this, we use the example in Figure 5, and let us assume that the cost of an operator is the minimum latency to its upstream operator. This is an upstream aggregation, thus n_2 has o_4 as its dependent. The local metric of each node is the latency to its parent and $h_2(o_4)$ is the minimum latency of all nodes connecting n_2 and n_5.

Operator cost-effect property. *Assume a change $\Delta(n_i : l_i \rightarrow l_i + \delta)$ that triggers a change Δoc_j on the cost of a dependent operator $o_j \in D_i$. If the operator cost is defined based on the* SUM *function, then $\Delta oc_j = \delta$, while if it is defined based on the* MIN *function, then $\Delta oc_j = \lambda$, where $\lambda = \min\{l_i + \delta, h_i(j)\} - \min\{l_i, h_i(j)\}$.*

To explain the above, we assume in Figure 5 that the operator's cost is its output latency. Hence, we aggregate the link latencies between nodes n_5 and n_3 to get the latency of operator o_4. If the latency between n_4 and n_2 increases, then the latency of o_4 will have the same increase. If the operator cost is the minimum latency to its upstream operators, then the cost of o_4 is the minimum latency of the links between n_5 and n_3. A change on the latency between n_4 and n_2, will change the cost of o_4 by λ. The λ parameter identifies the difference between the new minimum latency be-

l_i change	Conditions	λ
$\delta < 0$	$\min_{o_j \in D_i}\{h_i(j)\} \geq l_i$	δ
$\delta > 0$	$\min_{o_j \in D_i}\{h_i(j)\} \geq l_i$	δ
$\delta > 0$	$h_i(j) \geq (l_i + \delta)$	δ

Table 3. Change on operator cost oc_j upon change on local metric l_i (if oc_j is defined based on MIN function).

tween nodes n_5 and n_3 and their current latency. This parameter can be further simplified under certain conditions as shown in Table 3. For instance, if the node with the minimum latency is the same node for all the dependent operators, and we only decrease further its latency, then every dependent operator will experience the same cost change.

Changes on the cost of an operator $o_i \in O$ will affect the cost, qc_j, of any query $q_j \in E_i$. The following property describes how changes on the cost of operators can be translated to changes on the queries' costs.

Query cost-effect property. *Assume a change* $\Delta(o_i : oc_i \rightarrow oc_i + \delta)$ *that triggers a change* Δqc_j *on the cost of a query* $q_j \in E_i$. *If the cost of a query is defined based on the* SUM *function, then* $\Delta qc_j = \delta$ *and if is defined based on the* MIN *function, then* $\Delta qc_j = \tau$, *where* $\tau = \min\{\beta_i(j), (oc_i + \delta)\} - qc_j$ *and* $\beta_i(j) = \min_{o_m \in P_j, o_m \neq o_i}\{oc_m\}$.

Assume the query cost is the sum of its operators' latencies. Then, in Figure 5, decreasing the latency of o_2 will decrease the cost of q_1 and q_2. If the query cost is defined as the minimum latency of all its operators, then decreasing o_2's latency might create a new cost for q_2, depending on o_4's latency. The τ parameter takes into account the minimum cost of all query operators except o_2 (that is value $\beta_i(j)$) and identifies the change on the query cost. Once we have evaluated the cost changes on dependent queries, we evaluate the new value for the query dependence cost, based on Definition 3.

Benefit evaluation steps. The above properties allow us to estimate the benefit of a given operation. First, we evaluate the change on the cost of the migrated operator (or replica) by aggregating the local metrics for its potential new neighbors. The aggregation function and the set of neighbors depend on the definition of the operator cost (upstream or downstream aggregation). Given a change on the operator cost, we quantify the impact on their query costs based on the query effect property.

In the next step, we estimate any changes on the local values of the affected nodes, based on the definition of the local metric. Since the local metric is a function on local statistics, we evaluate the new local metric based on the local statistics of the affected nodes. For example, if the local metric is the node's processing load, and we migrated an operator to the node, then we add on the destination node's load the expected processing load of that operator, as estimated by its current host's statistics. Depending on these changes, our protocol uses the operator and query cost-effect properties to evaluate the impact on the dependent queries of these nodes and the new query dependence cost. Based on this impact, we use Equation 2 or 3 to evaluate the benefit of the operation.

4.3. Operator distribution protocol

Periodically, every node evaluates the benefit of all operations on its local operators. For each operator, it considers all possible migrations, replications and partitions (the last two operations are combined with the possible replica migrations). The most effective of all pairs (operator, operation) is sent to the root of the tree. The roots of all trees collaboratively identify the best operation which is applied by the host of the operator. This approach ensures improvement in every step, assuming that at least one beneficial operation has been identified.

Dynamic operator modifications. Our system adopts the "pause-drain-resume" approach to migrate or change the execution of stateless operators. When a node decides to modify an operator, its pauses the data flow to that operator, and starts buffering any incoming tuples. The operator executes any remaining tuples and after the operation is applied the node resumes the data flow. To handle migration of stateful operators we adopt solutions proposed in the literature [31].

Optimization state. Table 4 shows the state nodes need to maintain for the different functions for aggregating the operator and query costs. This state is derived based on Equations 2 and 3 and the operator and query cost-effect properties. For example, to use the operator effect property for the MIN function, nodes need to know the critical value of their local and dependent operators. By maintaining this state we reduce the communication overhead during the optimization process. Note that this state is common for any operation of our framework and independent of the actual performance metric. Moreover, it depends in most cases on the dependent entities of a node and not on the global set of nodes, queries or operators in the system.

The query manager of a node n_i maintains also information about the set of local operators, *e.g.*, their cost and the root brokers publishing their inputs. This

Operator Aggr	SUM	MIN
Query Aggr	size of Z_i	critical value $\forall o_j \in O_i$
SUM	size of G_i	critical value $\forall o_j \in D_i$
		size of E_j $\forall o_j \in D_i$
		dependence cost $c(G_i)$
MIN	system cost,	system cost
	dependence cost $c(G_i)$	dependence cost $c(G_i)$
	value $\beta_i(j)$ $\forall q_j \in Z_i$	critical value $\forall o_j \in O_i$
	B_j $\forall o_j \in D_i$	value $\beta_i(j)$ $\forall q_j \in Z_i$
		critical value $\forall o_j \in D_i$
		B_j $\forall o_j \in D_i$

Table 4. Optimization state for n_i and $B_j = \{(\beta_j(m), oc_j) | q_m \in E_j\}$.

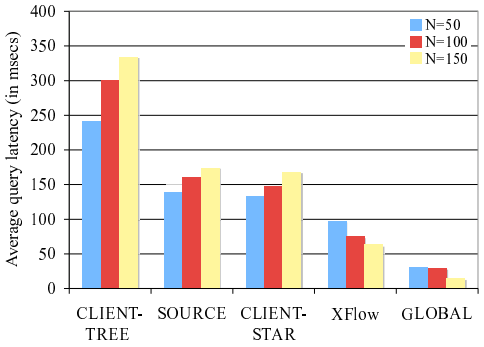


Figure 6. Average query latency on PlanetLab for 500 queries. XFlow performs better than all practical approaches and approximates GLOBAL. (N is the network size.)

operator-related state has size $O(|O_i|)$. Nodes also need some information about the trees they participate, *e.g.* the cost of each virtual node they own, its profile in each tree and the profiles of each of its children per tree. This state is of size $O(|T_i|)$, where T_i is the set of trees in which n_i participates.

Optimization traffic. During optimization, nodes exchange statistics with their neighbors in the optimization unit. This is the information required from our protocol to evaluate the changes on the local metrics of the affected nodes and on the operators to be migrated, replicated or partitioned. This state depends on the definition of our cost metrics. For example, for the query latency metrics defined in Section 3, nodes need to know the latency of the candidate locations from the upstream operators. Thus, they collect the local metrics from the nodes connecting the candidate location to the location of the upstream operators.

Periodically, nodes exchange data in order to calculate their local state. To minimize this maintenance traffic, nodes gather and calculate this information in a hierarchical fashion. They aggregate the state from their children and push the result to their parents across the trees. For example, to maintain the size of the downstream dependent operators, nodes aggregate the number of operators in their subtrees and forward the sum to their parents.

Batch-executing multiple optimization steps. Although applying a single operation per optimization period will improve performance over time, considering multiple operations could speed up convergence to better configurations. To this end, we allow multiple operations to be applied during a single period, *e.g.*, migrating multiple operators.

We use a standard best-first-search-like algorithm for identifying an effective set of operations. At each step, we consider all possible operations for each operator on the node and evaluate their benefits. The best operation that does not violate any constraint is selected, and, given the new configuration defined by this operation, we reevaluate the benefit of distributing another operator from the remaining ones. We continue by picking the best combination of the (now two) operations, continue this best-first search iteratively, and stop after a tunable but fixed number of steps.

Benefits of the pub-sub model. Using a pub-sub dissemination model has a number of advantages for our optimization. First, the loosely-coupled architecture of pub-sub networks allows them to deal efficiently with churn. XFlow treats our operators as consumers and subscribers and thus, our optimization operations are handled as subscriptions/unsubscriptions from the trees. This allows XFlow to be scalable and robust in terms of the number of dynamic operator changes. Second, it allows for dynamic creation of multiple trees, depending on the degree of sharing of external and internal sources. Clients can join trees that carry only relevant data, reducing redundant transmissions, while multiple large trees increase the neighborhoods over which we can distribute operators. High degree of sharing has been shown in practice for external [21] and internal sources [4]. In the case of low inputs sharing, grouping input streams will allows XFlow to get the similar benefits.

5. Performance Evaluation

We have implemented an initial prototype of XFlow in Java and studied its performance on the PlanetLab testbed. We used a network of up to 150 PlanetLab sites and up to 900 queries. The input streams for each query are chosen from a set of 700 real-world input streams using a Zipf distribution. Each input stream is a feed published from an RSS source, pulled withs a frequency that creates an average stream rate of 3.2KB/sec.

Operator	Description
count	counts the number of items in a feed
filter	blocks items that match a filtering rule
rename	renames item attributes
copy	copies item attributes
sort	sorts items based on an ordering rule
split	splits a feed into two identical copies
truncate	limits the number of items that pass through it
union	merges two feeds together
unique	combines items containing identical strings

Table 5. RSS feed processing operators.

Our queries are composed by a set of operators, similar to the operator set of Yahoo!Pipes [4], a centralized feed aggregator and manipulator that lets users mashup data sources. The operators may union, split, sort or filter the RSS feeds with a variety of conditions. The operators used are described in 5. Unless otherwise stated, each query takes as input a number of randomly chosen RSS feeds and applies a chain of at least five processing operators. An example query unions the input feeds, filters them based on a string in the title, sorts the results by date, truncates them and returns the top-k items.

The external RSS sources are assigned randomly to a set of four root brokers, while the clients are hosted by the remaining brokers. Grouping sources into four trees, allows more nodes to connect to the same tree, leading to larger neighborhoods over which operators can be distributed. We assign clients to the remaining brokers randomly, when no constraints are defined. Otherwise, we use an assignment that respects the constraints. and depending on the inputs of their local queries, nodes connect to the proper tree, by picking a random node from the ones already in the tree as their parent.

We used our prototype to implement three distributions of operators, each one optimizing a different metric which are: (i) average query latency, (ii) maximum processing load across all nodes and (iii) total bandwidth consumption (defined in Section 3). Our experiments demonstrate XFlow's effectiveness, as it manages to improve these metrics significantly over a sequence of optimization steps.

5.1. Operator placement

We start our discussion by demonstrating the efficiency of our operator placement approach. We examined four alternative placement approaches (note that similar approaches were used for comparison in [8, 26] for smaller networks). CLIENT-STAR assigns operators to the location of their client and their host nodes connect directly to the root brokers, creating a star

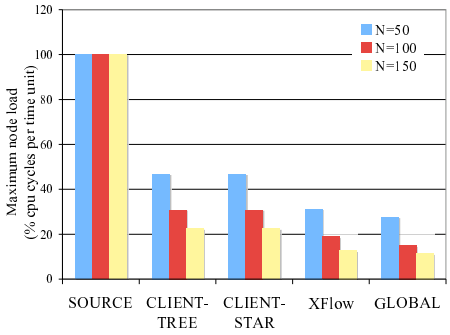


Figure 7. Maximum processing load for 500 queries. XFlow performs close to the GLOBAL approach.

topology for every tree. CLIENT-TREE also assigns the operator to the host of the client, however, nodes connect to the trees through a random member of the tree. SOURCE places operators to the root brokers publishing their input stream. The root brokers process the queries and forward the results to the clients. GLOBAL applies a greedy strategy that considers all existing queries in the order they were registered to the system and places each after an exhaustive search over all possible placements. This requires global knowledge of the query set and workload, and is not infeasible in practice, but it gives a target upper bound for the performance of our algorithms. For each metric, we used a different implementation of GLOBAL.

We compare these approaches with XFlow's protocol and show that, although the best placement depends on the optimization metric, XFlow consistently performed very close to the best placement *regardless* of the performance metric.

Latency. Figure 6 shows the average query output latency for 500 queries and different network sizes. This latency includes the processing as well as the network latency. The more operators a node must process and the higher its fanout in the tree, the greater the processing latency that feeds observe when traveling through that node. For the GLOBAL case, we assign operators to the node with the lowest latency from the root broker publishing their input. To reduce further the network latency, we assign clients to the same node as their queries, so that output results are not forwarded in the network. GLOBAL performs best for 150 nodes because the node with the least latency to the root brokers was not included in the set of 50 and 100 nodes.

CLIENT-TREE has the highest latency as queries receive their inputs through random paths to the roots, possibly with multiple network hops. The latency increases with the network size, as more nodes partici-

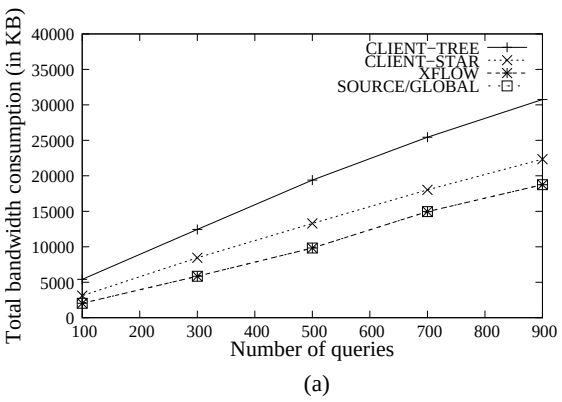


Figure 8. Total bandwidth consumption for 100 Nodes. XFlow converges to the best placement.

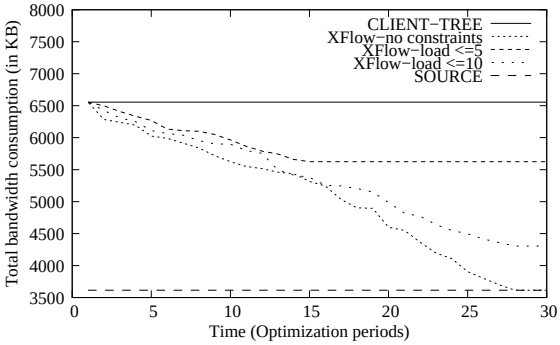


Figure 9. Total bandwidth consumption under constraints on the processing load of each node. The performance improves, but it is limited by the constraints.

pate and the random paths tend to have more hops. SOURCE and CLIENT-STAR create a star topology. Hence, the processing latency at the roots of the trees is high, since they have to filter the incoming tuples for each of their children. As the network size increases, the fanout of the roots, and thus the processing latency, increases. SOURCE has worse performance, since the roots perform all the processing.

Our performance is comparable to GLOBAL, and significantly outperforms all the other practical approaches. It incrementally migrates operators and improves the latency by up to 75% for the case of 150 nodes, by placing operators on brokers with smaller network and processing latency. Our protocol applies around 160 migrations, replacing an average of 17 operators at each optimization step.

Processing Load. We also studied the maximum processing load across all nodes for 500 queries (Figure 7). For this metric, placing all operators on the root brokers (SOURCE) leads to worse performance, as it utilizes all the resources from the root brokers. CLIENT-TREE and CLIENT-STAR distribute the load across all nodes that host a client. However, they perform worse than XFlow and GLOBAL. GLOBAL places every new query at the least loaded node. XFlow applies around 42 migrations and performs near as well as GLOBAL. As network size increases, it performs even better, since more nodes are available for distributing the processing.

Bandwidth. Figure 8 shows the total bandwidth consumption for varying number of queries. In this experiments each query is a chain of five filter operators for which the selectivity is uniformly distributed in [0,1]. The best performance in this case is achieved by placing operators with selectivity less than one close to the sources. This approach eliminates input tuples close to their sources, reducing the amount of data forwarded in the network. Thus, both SOURCE and GLOBAL place the operators on the root brokers and achieve the best performance. CLIENT-STAR consumes more bandwidth as all input tuples are forwarded to the clients for processing. CLIENT-TREE performs worse, because input tuples are forwarded to their clients through multiple hops. However, XFlow manages to continuously refine the operator placement and converge to a distribution that requires low bandwidth consumption, *i.e.*, over time, it moves almost all operators to the root brokers, performing the same as GLOBAL.

Constrained metrics. One way of achieving the benefits of pushing all operators closer to the sources, without saturating the processing resources of the root brokers and the nodes close to them, is by adding constraints on the maximum load of the nodes. These constraints will prevent operators from migrating to the root brokers. We expressed such constraints and studied XFlow's bandwidth consumption on 100 PlanetLab sites using 200 queries.

Figure 9 shows the total bandwidth consumption when we set the load constraints to 5% and 10% of the node's CPU cycles and when no constraints exist. Our system incrementally improves its performance in all cases. In the absence of constraints, all operators end up being executed at the root brokers. However, when constraints exist, many operators cannot move to higher levels, as this would lead to a violation of the load constraint.

Figure 10 also shows the average query latency on 500 queries on 100 PlanetLab nodes when constraints

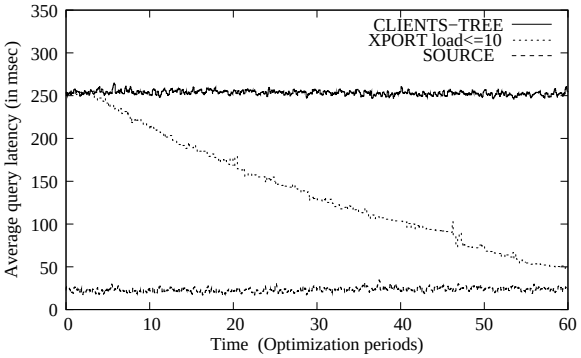


Figure 10. Average query latency on 100 PlanetLab sites with constraints on the nodes' processing load.

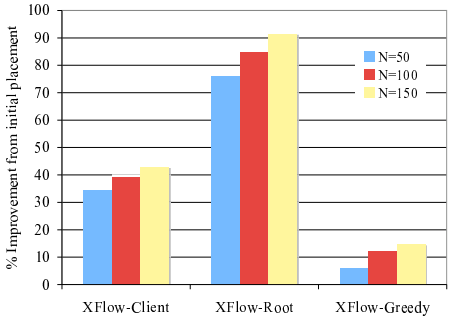


Figure 11. Improvement of maximum load for different initial placements. Performance is significantly improved for all cases.

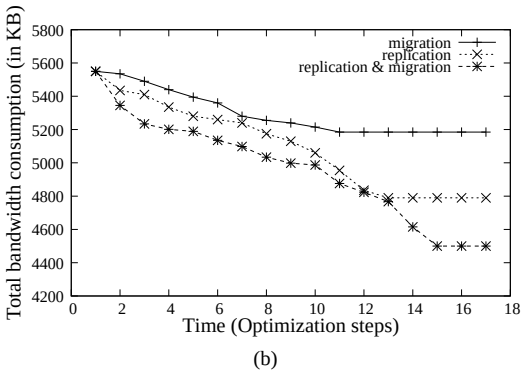


Figure 12. Total bandwidth consumption for 150 Nodes. Combining replication and migration reduces the bandwidth usage.

on the processing load are imposed. Again not all operators are be migrated to the root since this would violate its upper load limit. So our placement does not converge to the SOURCE approach. However, XFlow considerably improves the average query latency and converges to a configuration that performs 80% better than the CLIENT-TREE.

Effect of initial placement. We use the maximum load metric to study the effect of the initial placement of operators. Figure 11 shows the improvements we can achieve compared with the system's performance when we initially place the operators (i) on the clients' hosts (XFlow-Client), (ii) on the root brokers (XFlow-Root) and (iii) on the least loaded node (XFlow-Greedy). XFlow significantly improves the maximum load for all cases. The improvement is higher for the XFlow-Root case since maximum load for the initial performance is already high (see Figure 7). Our protocol migrates as many operators as possible from the root brokers and distributes the processing overhead across the network. XFlow-Client and XFlow-Greedy demonstrate smaller improvements as we already start with a good initial placement. However, even in the case of XFlow-Greedy, we manage to improve the performance by 28% when the network size increases to 150 nodes, by utilizing better the available nodes in the system.

5.2. Operator replication and partition

We now study our operator execution rules. In this section we examine first replication and then focus on operator partitioning. We allow our operators to be replicated or partitioned only once.

Replication. Figure 12 shows the improvement on bandwidth consumption when replication is used for 500 queries. We compare three cases in which we apply: (i) migration, (ii) replication, and (iii) migration and replication together. The improvement achieved with migration is very limited, since moving an operator to another node in one tree might not yield any benefit, as this node may not be subscribed to the second tree. When only replication is used, the performance is better. However, since no migration rules are allowed, these replicas are not reallocated and the bandwidth consumption can not improve further. Not surprisingly, best results are obtained when replication and migration are used together. In this case, XFlow will try to migrate our replicas closer to the sources, if this reduces the data forwarding. Since these replicas take input from a single tree, it is easier to identify a beneficial migration.

Operator partition. Figure 13 shows results for the operator partition case when we try to improve the maximum load of the network for 500 queries. Migrat-

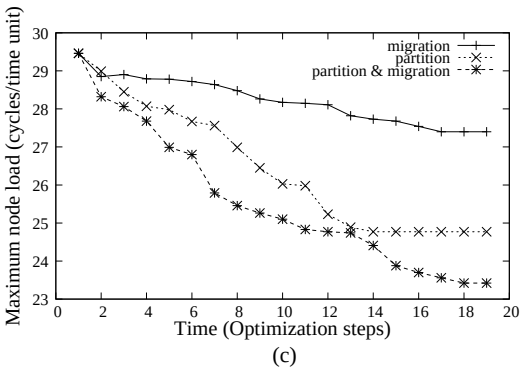


Figure 13. Maximum load across nodes. Combining migration and partition can lead to better performance.

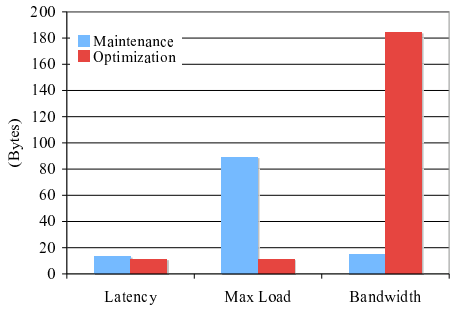


Figure 14. Network traffic for different metrics.

ing an operator incurs some benefits, however, since the load of a query depends on its input rate, reducing this rate will reduce its load as well. Partitioning an operator can achieve this, as now half of the input rate is processed by each replica. Moreover, when migration is allowed, these replicas can move independently in their respective trees, utilizing more processing resources and improving performance.

5.3. Optimization Overhead

In Section 4.3 we discussed the optimization traffic, *i.e.*, the information exchanged due to the optimization process. Intuitively, this traffic is a measure of the optimization overhead. Moreover, in order for nodes to maintain their optimization state, they periodically exchange data, leading to *maintenance traffic*. In our current implementation, nodes are set to exchange this data every second but this rate can be much coarser in a real deployment. We run experiments on 150 PlanetLab nodes and 500 profiles for the three metrics we implemented. Figure 14 shows the average optimization and maintenance data for each node. The results reveal that our protocol had very small overhead, as it does not exceed 200Bytes in the worst case.

More specifically, for the query latency, each virtual node periodically sends its latency from the root of the tree to its children, in order for them to evaluate their own latency. Similarly for bandwidth, virtual nodes send their input rates to their parent in the tree. In the case of maximum load, nodes calculate their processing cost locally. However, data is exchanged in order for nodes to evaluate their local state, *e.g.*, the maximum load of their subtree and the maximum load in the network apart from their subtree.

During optimization, for the query latency metric, nodes collect only the latency of all candidate locations for their operators or replicas; while for the maximum load metric, nodes need to know the current load of the candidate locations. Hence, the traffic for these cases is low. However, for the bandwidth consumption metric, nodes exchange the profiles of their neighbors, the profiles of their children, and statistics on the input rates of their streams. This information is required in order for a node to evaluate how the output rate of the candidate location will be affected.

6. Related work

Much work has focused on pub-sub systems including XRoute [14] and ONYX [16] for XML messages and SIENA [11] and SemCast [24] for relational data. These systems focus on either reducing the bandwidth usage [14, 24] or improving the efficiency of profile matching [11, 13, 15]. Moreover, they commonly support subscriptions with only predicate-based filters over individual events, so they cannot express complex stream-processing queries across multiple messages and streams. The SASE [29] event processing system extends the pub-sub model to allow the execution of stateful event processing queries. Among other differences, SASE is a centralized system and addresses neither operator placement in wide-area networks nor extensibility.

Our work is directly relevant to distributed stream processing. In general, these systems are designed and evaluated in small-scale cluster environments, thus, they do not address scalable and adaptive data collection and distribution. They also lack extensibility in terms of optimization metrics they support. As a case in point, Borealis [6] supports run-time operator migration but not operator replication or partitioning. Another example is Medusa [9], which is a similar system that performs dynamic load distribution in a federated setting based on agoric principles. Furthermore,

neither Borealis nor Medusa addresses overlay network management, which we believe is critical to ensure network scalability and adaptivity.

Network-based querying and dissemination systems are also related to our work. PIER [17] aims to build an Internet-scale querying system on top of a routing network given by a DHT, where operators are placed randomly. The operator placement problem has been studied in [8, 26, 28, 19]. SANE [8] and SBON [26] focus on minimizing the bandwidth usage, while [28] assumes a hierarchical stream collection which is not applied in our architecture. Very recently, COSMOS [30] combined the pub-sub model with stream processing in order to improve load balancing among nodes and minimize the communication cost. None of these approaches addresses adaptive query distribution and allow for operators to be replicated or partitioned. Moreover, they lack the generality and extensibility of XFlow, as they focus on specific optimization metrics.

Recent network-oriented efforts, such as MACEDON [27], P2 [22] , proposed systems promoting the advantages of generalization and extensibility. MACEDON and P2 construct overlay networks by abstracting over commonalities present in most overlay construction algorithms.

7. Conclusions and Future Work

Sophisticated Internet-scale Monitoring (ISM) applications are fast emerging. We proposed XFlow as a distributed infrastructure able to support a variety of ISM applications. Our key contributions include (i) a generic cost model that can express a variety of query optimization metrics, and (iii) a low-overhead metric-independent optimization framework that employs operator migration, replication and partition to improve system cost and (iii) a novel architecture that tightly integrates stream processing and pub-sub. Our experiments show that the XFlow is viable and effective.

XFlow presents an initial step towards a robust ISM system. There are several areas for immediate exploration and extension. First, we would like to implement a real ISM application and deploy XFlow as a public service on PlanetLab. This experience will allow us to better debug our system gather real user profiles and usage patterns. First, we plan to extend our optimization with tree-centric operations (tree merges, splits, etc.) and optimizations that affect the quality of query results (*e.g.*, allowing load shedding operators). We would also like to provide better support for user defined functions, which will likely occur frequently in ISM applications that involve rich data types. We believe we can optimize them them as well as we do our built-in operators, whose semantics are well-known, by providing a narrow "hinting" interface which the user can use to specify relevant properties of the operator (*e.g.*, whether/how the operator can be parallelized).

References

[1] Distributed intrusion detection, http://www.dshield.org.
[2] Distributed monitoring framework, http://dsd.lbl.gov/dmf.
[3] Earth scope, http://www.earthscope.org.
[4] Yahoo pipes, http://pipes.yahoo.com/pipes/.
[5] Abadi et al. Aurora: A new model and architecture for data stream management. In *VLDB journal*, 2003.
[6] Abadi et al. The design of the Borealis stream processing engine. In *CIDR*, 2005.
[7] Adler et al. Channelization problem in large scale data dissemination. In *ICNP*, 2001.
[8] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.
[9] Balazinska et al. Contract-based load management in federated distributed systems. In *SIGMOD*, 2005.
[10] Campbell et al. IrisNet: an internet-scale architecture for multimedia sensors. In *MM*, 2005.
[11] A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, 2003.
[12] Carzaniga et al. Design and evaluation of a wide-area event notification service. *ACM TOCS*, 19(3), 2001.
[13] Chan et al. Efficient filtering of XML documents with xpath expressions. In *ICDE*, 2002.
[14] R. Chand and P. Felber. Scalable protocol for content-based routing in overlay networks. In *NCA*, 2003.
[15] Y. Diao and M. J. Franklin. Query processing for high-volume XML message brokering. In *VLDB*, 2003.
[16] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale XML dissemination service. In *VLDB*, 2004.
[17] Huebsch et al. Querying the internet with PIER. In *VLDB*, 2003.
[18] Kostic et al. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.
[19] Kumar et al. IFLOW: Resource-aware overlays for composing and managing distributed information flows. In *EuroSys*, 2006.
[20] Kuntschke et al. StreamGlobe: Processing and sharing data streams in grib-based P2P infrastructures. In *VLDB*, 2005.
[21] Liu et al. Client behavior and feed characteristics of rss, a publish-subscribe system for web micronews. In *IMC*, 2005.
[22] Loo et al. Implementing declarative overlays. In *SOSP*, 2005.
[23] Motwani et al. Query processing, approximation, and resource management in a stream management system. In *CIDR*, 2003.
[24] Papaemmanouil et al. Semcast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.
[25] Papaemmanouil et al. Extensible optimization in overlay dissemination trees. In *SIGMOD*, 2006.
[26] Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[27] Rodriguez et al. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI*, 2004.
[28] Srivastava et al. Operator placement for in-network stream query processing. In *PODS*, 2005.
[29] E. Wu, Y. Diao, and S. Rizvi. High-performance complex event processing over stream. In *VLDB*, 2006.
[30] Zhou et al. Leveraging distributed pub/sub systems for scalable stream query processing. In *BIRTE*, 2006.
[31] Zhu et al. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, 2004.