

# Supporting Generic Cost Models for Wide-Area Stream Processing

Olga Papaemmanouil <sup>#1</sup>, Uğur Çetintemel <sup>\*2</sup>, John Jannotti <sup>\*2</sup>

<sup>#</sup>*Department of Computer Science, Brandeis University, Waltham, MA, USA*

<sup>1</sup>*olga@cs.brandeis.edu*

<sup>\*</sup>*Department of Computer Science, Brown University, Providence, RI, USA*

<sup>2</sup>*{ugur, jj}@cs.brown.edu*

**Abstract**—Existing stream processing systems are optimized for a specific metric, which may limit their applicability to diverse applications and environments. This paper presents XFlow, a generic data stream collection, processing, and dissemination system that addresses this limitation efficiently. XFlow can express and optimize a variety of optimization metrics and constraints by distributing stream processing queries across a wide-area network. It uses metric-independent decentralized algorithms that work on localized, aggregated statistics, while avoiding local optima. To facilitate light-weight dynamic changes on the query deployment, XFlow relies on a loosely-coupled, flexible architecture consisting of multiple publish-subscribe overlay trees that can gracefully scale and adapt to changes to network and workload conditions. Based on the desired performance goals, the system progressively refines the query deployment, the structure of the overlay trees, as well as the statistics collection process.

We provide an overview of XFlow’s architecture and discuss its decentralized optimization model. We demonstrate its flexibility and the effectiveness using real-world streams and experimental results obtained from XFlow’s deployment on PlanetLab. The experiments reveal that XFlow can effectively optimize various performance metrics in the presence of varying network and workload conditions.

## I. INTRODUCTION

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 “macrosopes” [1], [2], network performance and security monitoring [3], massively multi-player online games, and feed-based information mash-ups [4].

While many ISM applications share common characteristics (e.g., stream processing, overlay network construction, membership management), they often exhibit diverse application logic and performance requirements. For example, a camera-based surveillance application may need to perform feature extraction over MPEG streams, whereas a feed-oriented application may process XPath queries over RSS streams. Similarly, a network intrusion application may have strict response latency requirements, whereas an environmental monitoring

application running in a peer-to-peer setting may care more about fairness in bandwidth consumption.

Currently, ISM applications are implemented using custom, ad-hoc approaches that hinder their scalability and maintainability. For example, existing stream processing systems [5], [6] support complex continuous queries, while distributed versions [7], [8], [9], [10] allow for queries to be transparently distributed across multiple nodes. However, these systems focus on a single performance measure, which they optimize using hard-wired approaches, making it extremely difficult to effectively incorporate new optimization metrics. Implementing optimization mechanisms for new metrics is a challenge when the built-in metric is not the “right one” for a given application. Furthermore, most approaches are implicitly designed for clustered environments and assume a small number of sources and clients. Thus, they do not provide mechanisms for “massively” parallelizing processing over wide-area networks to take advantage of the large number of available computing elements (e.g., via operator replication and partitioning). Going forward, there is a need for general-purpose infrastructures that can effectively support a broad spectrum of ISM applications.

This paper presents XFlow, an ISM system for distributing stream processing queries. XFlow’s novelty lies on this extensibility, i.e., it can be easily customized to support application-specific processing logic, performance expectations and constraints. XFlow creates, maintains and optimizes an overlay network, given dynamic stream sources, clients with stream-oriented queries and application-specific performance expectations. The network consists of multiple, potentially overlapping, dissemination trees, created dynamically depending on the degree of stream sharing across queries. XFlow employs a unique combination of operator migration, replication and partitioning, and progressively refines the processing of the queries, the structure of the overlay trees, as well as the statistics collection process, to meet the desired objectives.

A key feature of XFlow is that it relies on *localized* state and interactions to reduce the *global* system cost. Our framework utilizes an aggregation-based metric definition model that allows us to rely only on localized, aggregated, network and workload statistics. Nodes distribute their queries on their neighborhood as well as on specific promising network regions which they discover through dissemination of these statistics. One of our key results is that even simple aggregations of

statistics are sufficient to achieve efficient operation with low overhead, as well as allow XFlow to avoid local optima and converge to near-optimal configurations. Moreover, we employ probabilistic techniques for disseminating statistics, keeping the network traffic within constant bounds independently of the number of queries and nodes and allowing for high scalability and efficiency.

XFlow relies on the pub-sub paradigm [11], [12] as the underlying communication model. This model effectively decouples sources and destinations over geography and time: sources publish their data without knowing where and when the consumers will access them and consumers subscribe their queries without knowledge of specific sources. 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. This flexible, loosely-coupled architecture allows for scalable dissemination of high stream volumes to a large number of consumers and robustness in the presence of high query subscription/unsubscription rate.

One of the key features of XFlow is that it uses the pub-sub model to also decouple the query operators. It 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 the overall system model while at the same time facilitates (i) the sharing of intermediate processing results and (ii) light-weight dynamic modifications, such as adding, removing, migrating, and replicating operators.

Our contributions can be summarized as follows:

- 1) We introduce a novel architecture consisting of multiple, dynamic overlay trees, for stream collection, processing and dissemination.
- 2) We present a *generic* cost model that can express a range of metrics for evaluating the overhead and efficiency of *user queries* as well as various *resource utilization* metrics. The model relies on statistics aggregation in order to reduce the required state and network traffic.
- 3) We describe a generic distributed *query optimization* framework that uses local, aggregated state and dynamically modifies the structure of the overlay as well the placement and processing of the operators 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.
- 4) We introduce a metric-independent statistics selection and dissemination mechanism that utilizes the semantics of the cost functions to identify available network resources. Based on this approach, we can carefully target our optimization towards low-cost, promising network regions and avoid local optima.
- 5) We demonstrate the effectiveness, flexibility and practicality of XFlow through a prototype implementation and experiments processing real-world feeds on the PlanetLab testbed. We show that XFlow’s localized statistics allow the system to converge to near-optimal configurations for a variety of metrics.

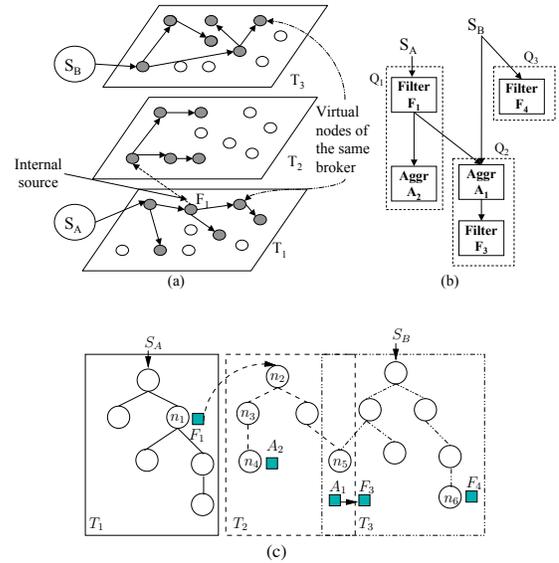


Fig. 1. XFlow’s system model.

The rest of the paper is structured as follows. We describe the architecture of XFlow in Section II. We introduce our cost model in Section III and describe the generic optimization in Section IV. We present our experimental results in Section V, the related work in Section VI, and conclude the paper with final remarks and plans for future work in Section VII.

## II. SYSTEM MODEL

XFlow consists of an overlay network of *brokers* (or nodes) providing stream routing and stream-based processing services (e.g., [5], [6]). *External sources* reside outside XFlow and publish data streams according to a well-defined global schema. Clients are also external and they subscribe stream-based continuous queries on the global schema.

**Pub-Sub model.** XFlow uses a uniform pub-sub mechanism for disseminating *all* data flows in the system. One implication is that each query operator publishes its output stream and subscribes 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.

Each source publishes its stream to a dissemination channel. XFlow implements each dissemination channel as an overlay distribution tree which connects its source to the subscribed consumers (i.e., operators or external clients). Figure 1(a) shows three trees:  $T_1$  and  $T_3$  distribute external streams ( $S_A, S_B$ ), while  $T_2$  disseminates the output of the operator  $F_1$ .

**Source registration.** Sources (internal or external) register by forwarding their output schema to the registration service. This service is distributed across multiple nodes to improve availability and scalability of the system. The registration service assigns the stream to a *root broker* 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 publishes the source’s output stream.

**Query registration.** To subscribe a new client, its host contacts the registration service and requests a list of *all* available internal and external streams along with their corresponding dissemination trees. Following the model of the Aurora/Borealis [5], [7], XFlow allows users to browse the query network and reuse existing processing operators. This way intermediate results can be shared by multiple queries. For example, in Figure 1(b) query  $Q_2$  uses the output of operator  $F_1$ . Automatic detection of sharable computations [13] could be easily incorporated and is outside the scope of this work.

Nodes subscribe their queries by connecting to the tree(s) publishing their input streams. To join a tree, they select as parent one of the tree members, based on the applications expectations and constraints. E.g., for latency-sensitive applications we chose the closest node, while for applications with fanout restrictions the parent’s new set of children should not violate the constraints. As, in practice, a single tree may disseminate multiple streams, nodes specify the input stream(s) they wish to receive through their *profile*, which is extracted from the selection predicates of the query. This profile is forwarded upstream to the root, creating a reverse routing path to the node. Using this routing tree, nodes can forward streams published through a tree only to the interested descendants.

**Stream processing model.** Queries 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. When a client registers a query, each operator subscribes to its 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 sources and clients are “pinned” to their brokers, 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.

Figure 1(b-c) shows three queries and their possible deployment.  $Q_1$  is distributed across two trees.  $F_1$ ’s host,  $n_1$ , subscribes to stream  $S_A$  through tree  $T_1$  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  $T_3$ .

**Tree management.** *Conceptually*, XFlow creates one tree per each internal and external source. In practice, any two pairs of brokers communicate through 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. 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.

Finally, 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.

01. system cost:= $f$ (node cost, NODES)   $f$ (query cost, QUERIES) 02. node cost:= $f$ (local stats)   $f$ (operator cost, OPERATORS) 03. query cost:= $f$ (operator cost, OPERATORS)  04. operator cost:= $f$ (local stats, UP DOWN OPERATORS)   local stats 05. $f$ := MIN MAX SUM AVERAGE
--

Fig. 2. XFlow’s cost model.

A simple heuristic could use bloom filters in order to discover trees that involve the same set of nodes and merge them. The problem of effectively grouping data sources and mapping them to a given set of trees has been studied in [14] 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.

### III. GENERIC COST MODEL

XFlow is a generic stream processing system that can be customized for different optimization metrics. In contrast to existing solutions [8], [10] that focus on specific metrics, our cost model can express a variety of performance measures through a sequence of statistics aggregation steps. A summary of our model is shown in Figure 2. In this section, we describe XFlow’s cost model in detail.

#### A. Optimization metrics and constraints

Stream processing systems are often evaluated by query-related QoS targets [7] as well as resource utilization metrics [8], [10]. The former refers to performance of queries (e.g., response latency), whereas the latter addresses the in-network processing overhead (e.g., bandwidth consumption). Our cost model is designed to express both metric types.

To facilitate the expression of performance metrics, nodes evaluate a set of built-in *local statistics* for the overlay links they maintain, e.g., link latency/bandwidth capacity, as well as their local operators, e.g., input/output rates, selectivity, processing costs. Application can combine these statistics and define their own optimization metrics and constraints.

**QoS metrics.** XFlow expresses the system performance as a function of the performance of its queries and operators. Specifically, the *system cost* is an aggregation of the cost of all queries, while the *query cost* is an aggregation of the cost of its operators. The cost of an operator could be defined based on its host’s local statistics, capturing metrics like processing latency or cpu load. Alternatively, it can be expressed as an aggregation of statistics on the network links connecting an operator to its neighbors in the query plan. These neighbors could be either the host of its upstream operator or the hosts of its immediate downstream operator. If there exist multiple upstream or downstream operators, we define the cost of each one independently and average the costs. This definition captures network-related metrics, such as query response latency, which includes the delay for forwarding streams from upstream to downstream operators.

The maximum query latency is an example of upstream aggregation: an operator’s latency is the sum of the link latencies on the path to its upstream operator plus its processing latency. The query latency is the sum of the latencies of its operators.

```

op latency= sum(latency, UP OPERATORS) + processing latency
query latency= sum(op latency, OPERATORS)
system cost= max(query latency, QUERIES)

```

**Resource utilization metrics.** We can also express metrics that measure the resource utilization of nodes. We refer to such metrics as the *node cost* and can be defined based on (i) the local statistics or (ii) the aggregation of the cost of local operators. In this case, the system cost is the aggregation of all nodes’ costs. An example is the processing load of a node, which is the total load of its locally executed operators. Hence, we define the maximum processing load as:

```

node load= sum(operator processing load, OPERATORS)
system cost= max(node load, NODES)

```

Another example is the outgoing bandwidth consumption of a node. This is the data the local operators publish (`op out rate`) plus the data the node forwards in the network (`fwd rate`). Hence, the total bandwidth consumption across all nodes is expressed as:

```

node out rate= sum(op out rate, OPERATORS) + fwd rate
system cost = sum(node out rate, NODES)

```

**Constraints.** Using our cost model, applications can express constraints for queries-related and resource utilization metrics. They can define cost metrics for operators, queries or nodes and specify bounds on them, e.g., maximum node load, maximum query response latency, etc. For example, they can express constraints to guarantee sufficient bandwidth capacity on the overlay links connecting an operator with its input producer. This is crucial for applications with high input rates. The capacity between two operators is the minimum available bandwidth of the network links connecting them:

```

operator capacity= min(capacity, UP OPERATORS)
operator capacity>= operator input rate

```

We note here that generic cost models were also proposed in [9], [15]. In [9] they use a less expressive model that simply sums network link costs. XPORT [15], a single-tree data dissemination system, proposes a similar aggregation-based model. XFlow differs from that model along two non-trivial dimensions. First, it incorporates high-level metrics that can express the efficiency of user queries as well the processing overhead on the network nodes, whereas XPORT focuses only on the dissemination cost. Finally, we express and optimize metrics and constraints across a general network of *multiple* trees, whereas XPORT limits itself to a single tree.

### B. Statistics collection.

XFlow nodes are customized to collect and aggregate statistics required for the evaluation of the performance metrics and constraints. To reduce the statistics traffic, nodes evaluate partial results and collaborate in order to derive the final cost metrics. Specifically, depending on the definition for operator cost, (i.e., upstream/downstream aggregation), each node on the path connecting two operators evaluates the required local statistic on the link to its parent/children in the dissemination tree and aggregates this with the aggregated metric of its parent/children respectively to derive its final cost value. For example, to measure the latency to the root of a tree, a node

measures the link latency to its parent and adds that to the latency of the parent to the root (which it receives periodically from its parent).

Based on this aggregation model, each node  $n_i$  maintains *only two* statistical values for every dissemination tree in which it participates: (i) the local statistic value,  $l_i$ , (e.g., latency to parent) and, (ii) the aggregation of network statistics,  $\phi_i$  (e.g., latency to root). Hence, our cost model requires state of *constant size* per node, which considerably improves the scalability of the system.

## IV. DISTRIBUTED OPTIMIZATION

XFlow includes a decentralized optimization framework that distributes queries, aiming to improve the global system cost. Its unique characteristic is that it employs a metric-independent model that dynamically combines operator migration, replication and partition and can uniformly handle a variety of cost metrics. Our framework does not rely on global information and has low communication overhead. Each node maintains aggregated information for its own “neighborhood” and periodically distributes its operators across its neighbors. We refer to these operations as *localized optimizations*.

To avoid local optima, we selectively distribute certain aggregated statistics to any potentially interested nodes and allow them to consider specific *directed* optimizations. These operations consider only network nodes that demonstrate a good performance and their resources could be used to improve the system performance. XFlow exploits its structured cost model and the known semantics of the aggregation functions to derive generic properties and equations that quantify the benefits of the candidate optimizations. In the rest of the section, we describe our optimization framework in detail.

### A. Operator distribution operations

Nodes distribute their operators across their neighbors, i.e., its parents and its descendants up to  $k$ -levels in each tree, where  $k$  is a system parameter. We denote as  $n_i^k$  a node  $n_i$  that is included in tree  $T_k$ , and Figure 3(a) shows the neighborhoods of  $n_i$  in two trees,  $T_1$  and  $T_2$ .

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

1) *Operator placement and migration:* Operator placement modifies where an operator is executed. As an example, to improve the query latency, operators may be placed closer to their upstream operators, as that would reduce the dissemination time for their input tuples. If an application aims to reduce bandwidth consumption, operators with selectivity less than one (e.g., filters) will migrate closer to their upstream operators, while if their selectivity is more than one (e.g., joins), they are pushed closer to the downstream operators. This reduces the amount of data emitted into the network.

XFlow relies on its pub-sub model to dynamically reroute streams to/from the new location of the migrated operator. Specifically, the new host of the operator subscribes to the

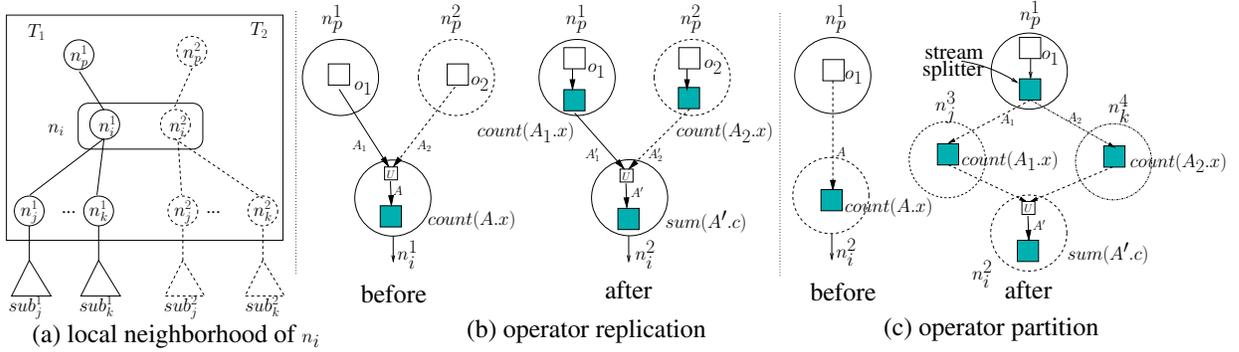


Fig. 3. Neighborhood of  $n_i$  and operator replication and partition.

operator’s input by joining its distribution tree and it publishes the operator’s output to the corresponding tree.

2) *Operator replication and partitioning*: The primary goal of replication and partitioning is to parallelize operator execution and utilize idle available resources in the network.

**Replication.** Replication is applied to operators that subscribe to multiple trees. Instead of collecting all inputs in a single node, replication exploits resources of multiple trees; it creates an operator replica for each tree and processes each input stream independently. To guarantee correctness, a *final operator* receives the output of all replicas and produces the final result. Replicas publish their outputs through a new tree, and the final operator’s host subscribes to these trees and publishes the final output stream.

The implementation of the final operator depends on the semantics of the replicated operator. For example, we union the outputs of filters, while we sum the outputs of count operators. In Figure 3(b) we use a union operator to initially merges streams  $A_1$  and  $A_2$  into stream  $A$  as well as merge the replicas outputs. The interested reader may find the implementation details in [16].

Replication can increase optimization opportunities. Replicas are handled as regular operators, which can be further migrated to strategic locations. For example, each replica can be placed closer to its own data source, reducing the query response latency. Since each replica subscribes to a different tree, it is migrated across nodes of its tree. Replica migration is more flexible than migration of the original operator; each replica processes a single input stream and, hence, it affects fewer nodes and trees than the original operators (which has multiple inputs).

**Operator partition.** Partition is applied on the input streams of operators. Each input is split into two sub-streams which are processed by a different replica of the operator, thereby exploiting data parallelism. The output of the replicas are processed by a final operator that produces the final result. Its semantic depends on the semantics of the original operator, similarly to the case of replication.

Partition is shown in Figure 3(c) for the count operator. Node  $n_p^1$  splits stream  $A$  and publishes two sub-streams  $A_1$  and  $A_2$ . In this example, we assume that node  $n_i$  participates in trees  $T_3$  and  $T_4$  and it places the replicas on its neighbors  $n_j$  and  $n_k$  on these trees. Finally, a union operator merges the

outputs and a sum operator produces the final result.

Partition may be used to reduce the processing load of the original host by migrating part of the processing to host of the replicas. Each replica has less load as it processes half of the initial input stream. Moreover, assuming a selectivity less than one for the replicas, the incoming rate of the final operator is reduced, decreasing the processing overhead of its host.

### B. Local optimizations

XFlow applies the above optimizations within the scope of its *local* neighborhood. Hence, nodes may migrate their operators (or replicas) to their parents, children or siblings across trees. Although at each step we consider a small network region, XFlow can gradually migrate operators to arbitrary network areas. Our experimental results reveal that for certain types of metrics this local search is scalable and effective, as it incurs very small network traffic and can converge to near-optimal configurations. For example, for non-constrained additive metrics, like average network latency, or total bandwidth consumption, our optimizations can migrate all queries close to their external sources.

### C. Directed optimizations

Although local optimizations can yield significant improvements for some metrics, we also discovered that certain measures are more prone to local optima. XFlow utilizes the available network statistics and allows nodes to identify promising non-local operations. These operations focus on specific low cost network areas and we refer to them as *directed* optimizations. In this section, we describe our approach.

1) *Statistics management*: XFlow nodes maintain certain local and aggregated statistics which they *selectively* disseminate to discover alternative neighborhoods that could improve the performance. XFlow relies on the definition of the system cost metrics, i.e., on the semantics of the aggregation functions (i.e., MAX, SUM, etc), in order to determine: (i) which nodes may be interested in the statistics, and (ii) which statistics could be of potential interest. Moreover, it creates filter-based routing paths on top of the existing overlay trees that forward the statistics from their producers only to the interested consumers. This statistics propagation process should run with a frequency that reflects the workload and network changes of the specific application.

Operator Aggr	Disseminated Statistics	Statistics Filters	Query Aggr	System Aggr	Condition
SUM	$(l_i, \phi_i)$	$< l_i$	SUM	SUM	$(\phi_i = c \pm \delta)$
			MIN	MIN	
MIN	$(l_i, \phi_i)$	$> \phi_i$	SUM	SUM	$(\phi_i = c \pm \delta)$
			MIN	MIN	
-	$l_i$	$< l_i$	SUM	-	$(l_i = c \pm \delta)$
			MAX	-	

TABLE I  
STATISTICS DISSEMINATION PER NODE. ( $c$ : SYSTEM COST)

**Statistics selection.** In Section III-B, we mentioned that each node  $n_i$  stores a local value,  $l_i$ , and an aggregated value,  $\phi_i$ , for every tree in which it participates. It periodically disseminates these statistics and informs its peers about the performance of its neighborhood. Specifically, the local value  $l_i$  reveals  $n_i$ 's resource utilization (e.g., processing load) or its neighbors' properties (e.g., latency to its parent), while the aggregated value  $\phi_i$  provides a performance measure of the *path* leading to  $n_i$  (or of the links to its children). Each node forwards statistics of constant size ( $O(1)$ ) for each tree it participates. The size of these statistics is independent of the number of queries and operators as well as the number of nodes and only depend on the number of dissemination trees.

**Selective dissemination.** XFlow distributes statistics only to nodes that may affect the global system cost. Table I shows the conditions that should hold in order for a node to receive any statistics from its peers. These conditions are *agnostic* of the actual optimization metric and depend on the aggregation functions that evaluate the query and system cost. For example, if the optimization metric is the average query latency, then every node can improve the system cost by reducing the processing or network delay of all the queries it hosts. On the other hand, in order to minimize the maximum query latency, we would forward statistics only to the nodes that process the query with the highest response time,  $c$ , (or with small difference  $\pm\delta$  from the worst response time).

Furthermore, nodes are interested only in information on network components (links, paths, nodes) that perform better than their own local neighborhood. Table I shows the predicates used by each node  $n_i$  to filter out non-informative statistics. XFlow uses the aggregation semantics to *automatically* customize these filters. Additive functions imply that the aggregated value will be higher than the local value, thus the lower-bound filter is their local value. For example, for the query latency case,  $n_i$  receives statistics about paths with less latency than its latency to its parent,  $l_i$ . In the case of the MIN function, the aggregated metric  $\phi_i$  provides the lowest value  $n_i$  is interested in;  $n_i$  will receive statistics about links and paths with higher bandwidth than the capacity of its own local path to the root,  $\phi_i$ .

**Probabilistic dissemination.** In order to reduce the statistics emitted in the network, we deploy a probabilistic-based technique for selecting only a subset of the available information. Specifically each node propagates the statistic of  $k$  nodes (we refer to them as *top-k*) which are picked based on the *lottery scheduling* algorithm [17]. Each node's value is assigned a number of tickets proportional to its "utility". This utility value depends on the semantic of the optimization metric. For

example, nodes with less load or latency will be given more tickets, thus having higher probability of "winning" the lottery and forwarding their statistics. The algorithm guarantees no zero probability for selecting any statistic. Our experiments revealed that this approach keeps the statistic traffic within constant bounds, independently of the number of queries and nodes, while it incurs low performance degradation.

**Statistics routing.** XFlow uses the structure of its network to distribute its statistics. It avoids flooding the network by constructing predicate-based routing paths that filter out unwanted statistics as early as possible. For example, we avoid forwarding load information of the most loaded node. To construct these filtering paths, nodes propagate their statistics filters (Table I) to their parents. Each node aggregates the filters of its children in each tree and propagates the aggregated filter upstream towards the root. This allows nodes to be aware of the interests of their descendants and selectively route filtered statistics to the interested nodes. Since our overlay network consists of inter-connected trees, every participating node is guaranteed to receive the statistics.

2) *Directed operations:* Our framework uses the collected statistics to discover specific low-cost neighborhoods. We categorize them into *intra-tree* and *inter-tree* neighborhoods.

**Intra-tree neighborhoods.** Dissemination trees connect nodes that receive and process the same data stream. Our statistics distribution process allows nodes to discover which of these nodes have better cost metrics, e.g., better path to the stream producer, less processing load, or less outgoing bandwidth consumption. Hence, each node considers migrating its operators (or its replicas) to a peer that receives its input stream through an alternative path with better performance. Another benefit of directed optimizations inside a tree is that overlay paths with good performance are re-utilized, improving the resource utilization metrics.

**Inter-tree neighborhoods.** XFlow nodes also receive statistics regarding nodes, links and paths that reside outside their dissemination trees. Thus, they can exploit any network component with low cost by incorporating them in their own trees. To achieve this, they consider migration of their operators to any node with lower cost and connect them to the tree through one of the existing nodes. To reduce the number of candidate locations each node has to consider, we check only the top- $k$  nodes with the least cost.

#### D. Evaluating global cost changes

Our framework includes a metric-independent cost model that quantifies the expected benefit of an operation. Our operations affect the performance of a subset of the nodes and operators, hence we focus on identifying the affected entities and evaluating the cost changes on them. Furthermore, we derive the state required to quantify an optimization operation. In most of the cases we maintain aggregated state in order to reduce the communication overhead. For the purposes of illustration, we will describe our approach with respect to the SUM and MIN functions. Similar results can be obtained for the AVG and MAX functions in a straightforward manner.

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 definitions identify 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.** The dependent operators of  $n_i$  is the set of operators,  $D_i$ , whose cost depends on  $n_i$ 's local metric,  $l_i$ :

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

**Definition 2.** 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:  $G_i = \bigcap_{o_j \in D_i} E_j$ .

Let us consider the query plan and its distribution in Figure 4, where the operator cost 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  $\alpha$  affects a set of nodes  $F_\alpha$ : 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 this node. 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 *aggregated dependent cost*.

**Definition 3.** The query dependent cost,  $c(G_i)$ , of a query 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 we aggregate operators' and queries' cost with the SUM function, then:

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

Our cost model tries to estimate the change on the aggregated dependent cost. In what follows, we illustrate the detail of this approach. The following property identifies the effect of an optimization on the system cost.

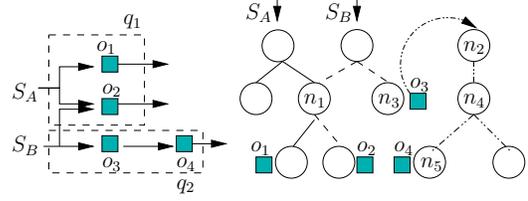


Fig. 4. Example of operator distribution.

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

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

If the system cost is defined by the MIN function, then:

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

where  $c$  is the current system cost.

Based on this definition, for each optimization operation, we need to identify the set of nodes affected and evaluate the impact on the cost of their local and aggregated cost of their dependent queries. In this section we discuss our approach.

1) *Impact on the query cost:* To identify the impact of an operation on a node's dependent queries we first evaluate the impact on the cost of its dependent operators. 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 operator cost is defined by 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'll use the example in Figure 4, and let us assume that the operator cost is its input bandwidth capacity, that is the minimum capacity of all the links connecting it 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 capacity of the link to its parent and  $h_2(o_4)$  is the minimum capacity 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  $\Delta oc_j$  on the cost of a dependent operator  $o_j \in D_i$ . If the operator cost is defined by the SUM function, then  $\Delta oc_j = \delta$ , while if it is defined by 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 4 that the operator 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 bandwidth capacity to its upstream operators, then the cost of  $o_4$  is the minimum capacity of the links between  $n_5$  and  $n_3$ . A change on the capacity between  $n_4$  and  $n_2$ , will change the cost of  $o_4$  by  $\lambda$ . The  $\lambda$  parameter identifies the difference between the new minimum capacity between nodes  $n_5$  and  $n_3$  and their current latency. This parameter can be further simplified under certain conditions. For instance, if the node with the minimum capacity is the same node for all the dependent operators, and we only decrease further its capacity, then every dependent operator will experience the same cost change. More details can be found in [16].

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

**Query cost-effect property.** Assume a change  $\Delta(o_i : oc_i \rightarrow oc_i + \delta)$  that triggers a change  $\Delta qc_j$  on the query cost  $q_j \in E_i$ . If the query cost is defined by the SUM function, then  $\Delta qc_j = \delta$  and if is defined by 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 operator latencies. Then, in Figure 4, 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 capacity of all its operators, then decreasing  $o_2$ 's capacity might create a new cost for  $q_2$ , depending on  $o_4$ 's capacity. The  $\tau$  parameter takes into account the minimum operator cost 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 can derive the new aggregated dependent cost based on the Definition 3 and the impact on the system cost based on Equations (2)-(3).

### E. Optimization protocol

Periodically, every node quantifies the benefit of all optimization operations on its local operators. For each operator, it considers all possible migrations, replications and partitions in its local and directed neighborhoods. We employ a hill-climbing-based local search that picks the most effective of all pairs (operator, operation) and sends it to the root of the tree. The roots of all trees collaboratively identify the best operation overall which is applied by the operator's host. This approach ensures improvement in each step, assuming there exists at least one beneficial operation. Moreover, to prevent needless optimizations, an operator is migrated (or replicated) only if the performance improvement is higher than a *minimum threshold*. This threshold depends on the cost of operator migrations/replications and ensures that optimizations are amortized over the lifetime of the query network.

XFlow speeds up its optimization by applying multiple operations as a single optimization step, e.g., migrating multiple operators. We use a standard best-first-search for identifying a beneficial set of operations. At each step, a node evaluates all possible operations for its local operators. 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 one more operator from the remaining

Oper Aggr	SUM	MIN
Query Aggr	$ Z_i $	$h_i(o_j), \forall o_j \in O_i$
SUM	$ G_i $	$h_i(o_j), \forall o_j \in D_i$ $ E_j , \forall o_j \in D_i$ $c(G_i)$
MIN	system cost $c(G_i)$ $\beta_i(j), \forall q_j \in Z_i$ $B_j, \forall o_j \in D_i$	system cost $c(G_i)$ $h_i(o_j), \forall o_j \in O_i$ $\beta_i(j), \forall q_j \in Z_i$ $h_i(o_j), \forall o_j \in D_i$ $B_j, \forall o_j \in D_i$

TABLE II

OPTIMIZATION STATE FOR  $n_i$  ( $B_j = \{(\beta_j(m), oc_j) | q_m \in E_j\}$ ).

ones. We continue by picking the best combination of the (now two) operations and execute this process iteratively for a tunable but *fixed* number of steps.

**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 it 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 exist solutions [18].

**Optimization state & traffic.** Table II shows the state nodes maintain in order to evaluate the impact on the operator and query costs. This state is derived from Equations 2 and 3 and the properties in Section IV-D. 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. Maintaining this state allows us to reduce the communication traffic 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.

Periodically, nodes exchange data in order to calculate their local state. We refer to this as the *maintenance traffic* which is calculated in a hierarchical fashion. Nodes 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.

During optimization, nodes exchange statistics with their neighbors in the trees. We refer to this as the *optimization traffic*, which is the information required from our protocol to evaluate the impact on the query and node costs (shown in Table II). This state depends on the definition of our cost metrics. For example, for the query latency metrics, nodes collect the latency of all candidate locations, while for the maximum load metric, nodes need to know the current load of these locations. Detailed description can be found in [16].

## V. 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 200 nodes participating in four dissemination trees. Thus, our networks consist of up to 800 node instances. The workload includes 900 queries with input streams chosen from a set of 700 real-world input streams

using a Zipf distribution with the skew parameter set to 0.97. Each input stream is an RSS feed pulled with frequency that creates an average stream rate of 3.2KB/sec.

Our queries are composed by a set of operators, similar to the operators of Yahoo!Pipes [4], a centralized feed aggregator that lets users mashup data sources. The operators may union, split, sort or filter the RSS feeds with a variety of conditions. Each query takes as input a random RSS feed (or two for the union operator) and applies a chain of five processing operators. An example query unions two 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 the 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. We assign clients to the remaining brokers randomly, when no constraints are defined. Otherwise, we use an assignment that respects the constraints. Depending on the inputs of their local queries, nodes subscribe to the proper tree and pick as their parent a random node in the tree that respects the constraints.

We used our prototype to implement three query distributions, 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 III). Our experiments demonstrate XFlow’s effectiveness, as it manages to improve these metrics significantly over a sequence of local and directed optimization operations.

#### A. Extensibility and effectiveness

We start our discussion by demonstrating that our operator placement framework can efficiently optimize different metrics. We examined four alternative placement approaches. CLIENT-STAR assigns operators to the location of their client and their host nodes connect directly to the root brokers, creating a star 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 workload and is not feasible 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 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. Furthermore, we show that certain metrics can converge to the optimal placement by using only localized operations, while for the rest of the metrics, directed optimizations achieve the optimal configuration or one that outperforms GLOBAL.

**Bandwidth consumption.** Figure 5(a) shows the total bandwidth consumption for varying number of queries. In these

experiments each query is a chain of five filter operators for which we manually set the selectivity to be 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 represent the *optimal* placement. CLIENT-STAR consumes more bandwidth since 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 by using only local optimizations (*XFlow-local*). It converges to a query deployment that requires low bandwidth consumption, i.e., over time, it moves almost all operators to the root brokers, performing the same as GLOBAL.

**Query latency.** Figure 5(b) shows the average query output latency for 500 queries deployed on 100 PlanetLab sites. We compare XFlow’s performance with the *optimal* in which each node has one overlay link to the root broker that publishes the input stream of its operators. This is provided by the GLOBAL as well the CLIENT-STAR placements. In these cases, assuming network latency dominates processing latency and no constraints on the node load, we execute each query on the root and forward the results to the client. This reduces the query latency as input tuples do not travel in the network. CLIENT-TREE uses multiple overlay links to connect an operator to its source, hence, its performance is worse. Figure 5(b) shows that XFlow converges very close to the optimal configuration after a number of migrations, which incrementally move our operators closer to the root of each tree.

**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. Furthermore, nodes will be forced to limit their fanout in the tree, as a large number of children would increase the overhead of forwarding incoming tuples to downstream nodes. Thus, trees with height higher than one will be constructed. We created such a network by imposing an upper bound of five (5) CPU cycles per time unit on the processing load of each node. Figure 5(c) shows the average output latency of 500 queries on 100 PlanetLab nodes.

Figure 5(c) shows that, when only local migrations are applied, XFlow can improve over the CLIENT-TREE placement but is outperformed by the GLOBAL. This is because our optimization converges to a local optimum: nodes cannot migrate their profiles to a better network region because their neighbors cannot accept any new operators without violating their load constraint. However, when directed optimizations are used (*XFlow-directed*) nodes become aware of peers with less latency and load. Hence, XFlow directly migrates operators to nodes outside their neighborhood and converges to a configuration that outperforms GLOBAL.

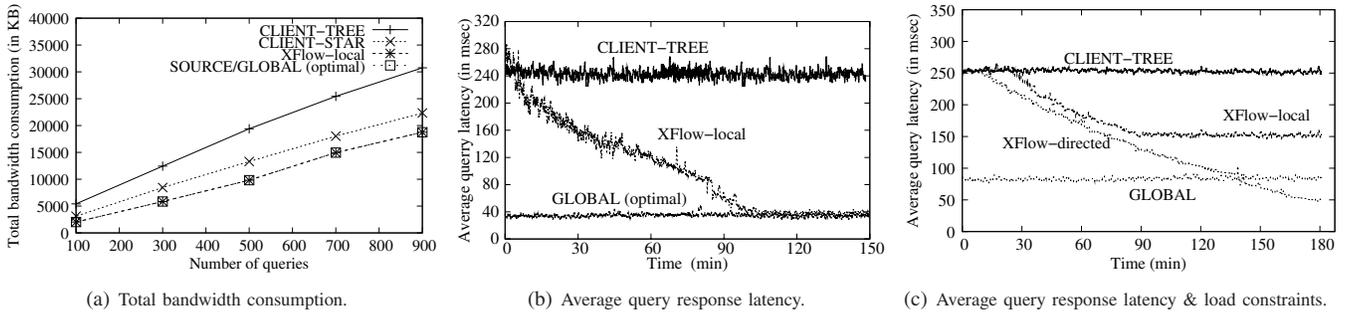


Fig. 5. Convergence for different metrics ( $|N|=100$ ,  $|Q|=500$ ). (a) Total bandwidth consumption *converges* to the optimal placement using only local optimizations. (b) Average query latency *converges* to the optimal placement using only local optimizations. (c) Load constraints prevent migration of all queries to the sources, while localized operations hinder the optimization. Directed optimizations allow XFlow to perform better than GLOBAL.

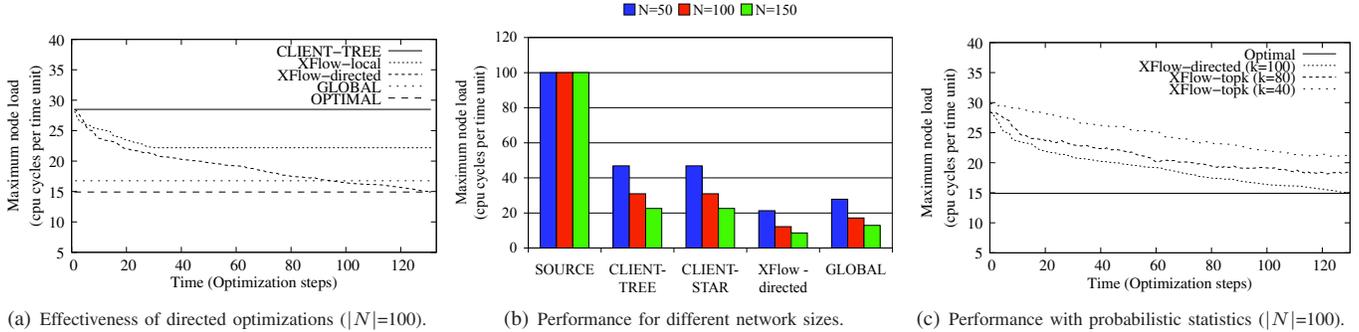


Fig. 6. Maximum node load for 500 queries. (a-b) Non localized optimizations allow XFlow to outperform GLOBAL and *converge* to OPTIMAL. (c) Directed optimizations using only top- $k$  statistics for various  $k$  values. XFlow improves the maximum processing load and converges close to OPTIMAL even with limited statistical information.

**Processing Load.** We also studied the maximum processing load across all nodes for 500 queries (Figure 6). For this metric, placing all operators on the root brokers (SOURCE) is the worst configuration 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 GLOBAL which assigns every new query to the least loaded node. For this metric, we also computed the OPTIMAL placement through exhaustive search.

Figure 6(a) shows that XFlow converges to the OPTIMAL when directed optimizations are used. Localized optimizations improve its optimization metric, but XFlow eventually converges to a local optimum (XFlow-local). In the case of XFlow-directed, statistics dissemination allow the most loaded node to discover less loaded peers and migrate part of its local processing, reaching eventually the optimal configuration. Moreover, as network size increases (Figure 6(b)), XFlow performs even better, since more nodes are available for distributing the processing. In all the cases, we converge to the OPTIMAL and outperform GLOBAL.

We also used the maximum load metric to evaluate the effect of our probabilistic statistics dissemination. Figure 6(c) shows XFlow’s performance when we propagate all statistics ( $k=100$ ) and the cases where only the top-40 and top-80 node metrics are forwarded, picked as described in Section IV-C.1. The results show that limited statistics do not incur significant performance degradation as XFlow is able to reach a configuration with performance close to that of OPTIMAL.

## B. Optimization overhead

XFlow relies on its aggregation model to reduce the size of statistics exchanged among nodes. This traffic includes the *optimization traffic*, the *maintenance traffic* (both described in Section IV-E), and the statistic required by the directed optimizations ((Section IV-C.1), and is a measure of our optimization’s overhead.

Figure 7(a)-(b) shows the traffic for increasing number of queries and nodes respectively when XFlow optimizes the maximum processing load metric. The results demonstrate that this overhead is independent of the number of queries while the localized traffic (i.e., optimization and maintenance state) is also independent of the network size. However, the traffic incurred by statistics dissemination increases with the number of nodes as these statistics are non-localized and thus, they are potentially forwarded to every node in the network. To reduce this traffic we propagate statistics based on the probabilistic technique we described in Section IV-C.1. Figure 7(c) shows the overhead when all statistics all disseminated and when only the top-20 and top-40 are selected. The results reveal that probabilistic dissemination maintains the statistics traffic constant as the network size increases.

Finally, Figure 8(a) shows the average optimization, maintenance and statistics data sent per node for all the metrics we implemented. The results reveal that XFlow has very small overhead: it requires less than  $240\text{Bytes}$  in the worst case, while the optimization traffic remains below  $90\text{Bytes}$ . Due to space limitations we omit the details for these metrics. The interested reader can refer to [16] for a detailed description.

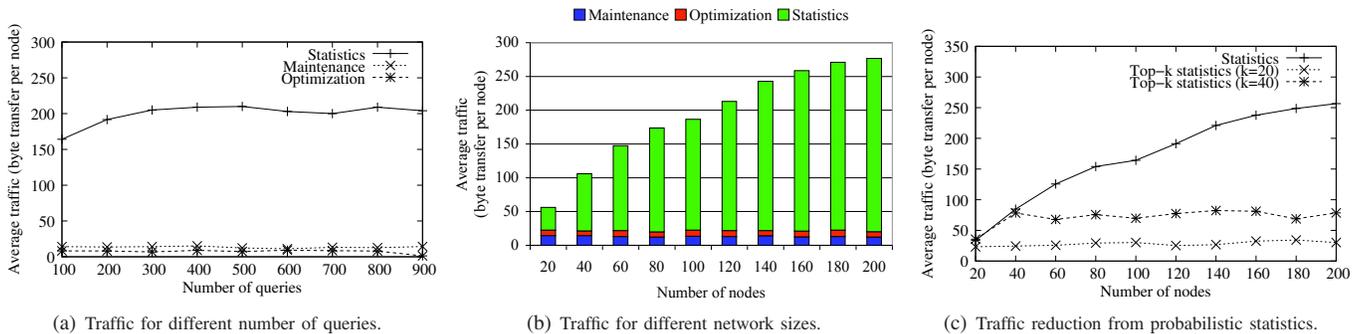


Fig. 7. Network traffic for the maximum processing load metric ( $|N|=100$ ). (a) XFlow’s overhead does not depend on the number of queries and has very small size ( $< 240$ bytes per node). (b) Statistics traffic increases with the network size but remains  $< 300$ bytes. Maintenance and optimization traffic remain *constant* with the number of nodes. (c) Probabilistic statistics dissemination keeps traffic *constant* for different network sizes.

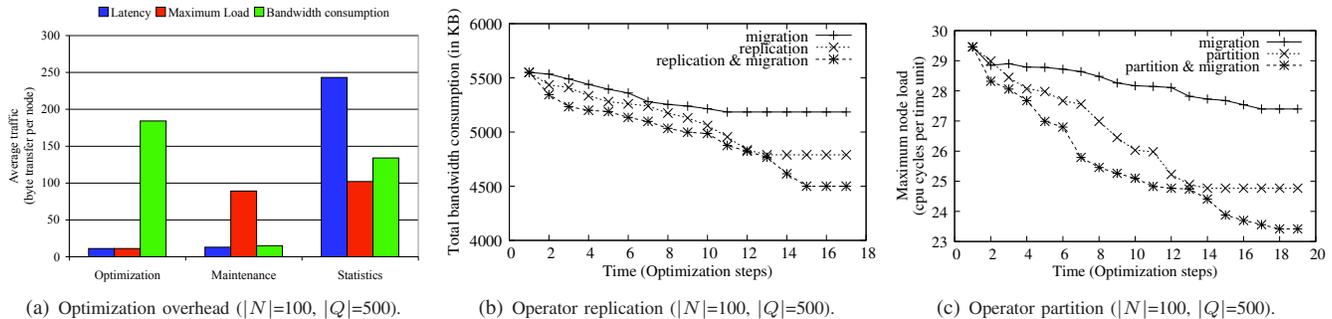


Fig. 8. (a) Network traffic for different metrics. Nodes exchange at most 190Bytes during the optimization and the statistics dissemination requires less than 240Bytes. (b)-(c) Optimization of the bandwidth consumption metric using operator replication and partitioning. Combining migration and replication/partitioning can lead to better resource utilization and load distribution.

### C. Operator replication and partition

Figure 8(b) shows the improvement on bandwidth consumption when we replicate operators with two input streams. We compare three cases in which we apply: (i) migration, (ii) replication, and (iii) migration and replication together. The improvement achieved when we only migrate operators is very limited. Replication is applied to operators that are subscribed to multiple trees. This prevents us from discovering beneficial migrations, as we need to discover a new host that could yield an improvement over all the trees. For example, a host could be closer to the root in one tree but required multiple hops to reach the root brokers of the remaining trees. When only replication is used, the performance is better. However, since these replicas are not reallocated, the bandwidth consumption can not improve further. Not surprisingly, best results are obtained when replication and migration are combined. In this case, XFlow migrates each replica independently and closer to each stream’s source, reducing the amount of data forwarding.

Figure 8(c) shows similar results for the operator partitioning case when we optimize the maximum load of the network. Migrating 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, replicas can move independently in their respective trees, utilizing more processing resources.

## VI. RELATED WORK

**Data Dissemination.** Distributed pub-sub systems have

been proposed for dissemination of XML messages [12], [19], [20] or relational data [21], [22]. These systems focus on either reducing the bandwidth usage [12], [22], improving the resource allocation [20], or providing efficient subscription processing mechanisms [19], [21]. Moreover, they commonly support subscriptions with only predicate-based filters, so they cannot express complex stream processing queries. Application-level multicast systems [23], [24], [25], [26] have also been proposed for scalable wide-area data dissemination using overlay networks. However, these systems do not support stream processing and address neither query deployment in wide-area networks nor metric extensibility.

**Distributed Stream Processing.** Our work is relevant to distributed stream processing solutions like Borealis [7] and Medusa [27]. Neither [7] nor [27] address overlay network management, which we believe is critical to ensure network scalability and adaptivity. Operator placement over wide-area networks has been studied in [8], [10], [28] which focus on minimizing the bandwidth usage. Finally, in [29] they deploy queries on sensor networks based on a localized heuristic.

In general, all the above systems lack extensibility in terms of their optimization metrics. XFlow can be customized for a variety of metrics and it provides a self-tuning overlay that deals with run-time resource and workload variations, a capability not present in many existing systems. Furthermore, a distinguishing feature of XFlow is its optimization framework, which can dynamically combine operator migration, replication and partitioning. Note that the above systems can support operator migration but not replication or partitioning.

Furthermore, XFlow supports dynamic creation of multiple overlapping dissemination trees and allows for optimizations across multiple trees. In [22], [25] it was shown that multiple trees enable better network utilization and reduce redundant transmissions by having clients become part of only those trees that publish relevant data. XFlow treats query operators as stream subscribers and publishers. In contrast to existing systems, i.e., Borealis [7], our model decouples operators, i.e., an operator does not require any state regarding the location of its upstream/downstream operators in the query network. This allows XFlow to handle a large the number of dynamic operator changes as changes do not require updating any state regarding their neighbors in the query network. Moreover, XFlow goes beyond centralized implementations of the pub-sub semantics, as in [9], [10], [30], and realizes this model through *dynamic* creation of multiple network trees.

**Extensible systems.** Recently, IFLOW [9] allowed for applications to express their own performance goals. However, they express metrics only as the sum of link costs and they assign stream flows to network edges based on a centralized, exhaustive search of the solution space. Our cost model is more expressive and supports a superset of the optimization metrics considered in IFLOW. Furthermore, we use a decentralized approach that incrementally improves the global cost based on simple, localized views of the network conditions.

Network-oriented efforts, such as MACEDON [31] and P2 [32], proposed also generic systems. They both construct overlay networks by abstracting over commonalities present in most overlay construction algorithms. Finally, XPORT [15] is an extensible dissemination system that adapts its structure to network conditions. It uses a generic cost model with which we provided a detailed comparison in Section III-A. Furthermore, XPORT uses a single dissemination tree, an approach that is neither efficient nor desirable, as it requires all external data to stream through the root, leading to the standard scalability and availability problems. Most importantly, the above systems do not support complex stream processing. Therefore, they can not express query-based performance metrics nor optimize them through operator-centric operations.

## VII. CONCLUSIONS AND FUTURE WORK

We proposed XFlow as a distributed infrastructure able to support a variety of ISM applications. XFlow can express a broad spectrum of optimization metrics, which it improves through metric-independent protocols. Nodes exploit the semantics of the optimization functions to derive aggregated statistics and discover promising network areas for applying query optimizations. Our experiments show that XFlow can converge to near-optimal configurations for a variety of metrics with low overhead.

XFlow presents an initial step towards an 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 and gather real user profiles and usage patterns. We also 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). Finally, we would like to provide better support for user defined functions. We believe we can optimize 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).

## ACKNOWLEDGMENT

This work has been supported by the National Science Foundation under grant No. IIS-0448284 and IIS-0325838.

## REFERENCES

- [1] "Earth scope." [Online]. Available: <http://www.earthscope.org>
- [2] Campbell et al., "IrisNet: An Internet-scale architecture for multimedia sensors," in *MM*, 2005.
- [3] "Distributed intrusion detection." [Online]. Available: <http://www.dshield.org>
- [4] "Yahoo pipes." [Online]. Available: <http://pipes.yahoo.com/pipes/>
- [5] Abadi et al., "Aurora: A new model and architecture for data stream management," in *VLDB Journal*, 2003.
- [6] Motwani et al., "Query processing, approximation, and resource management in a stream management system," in *CIDR*, 2003.
- [7] Abadi et al., "The Design of the Borealis Stream Processing Engine," in *CIDR*, 2005.
- [8] Ahmad et al., "Network-Aware Query Processing for Stream-based Applications," in *VLDB*, 2004.
- [9] Kumar et al., "Resource-aware distributed stream management using dynamic overlays," in *ICDCS*, 2005.
- [10] Pietzuch et al., "Network-Aware Operator Placement for Stream-Processing Systems," in *ICDE*, 2006.
- [11] Carzaniga et al., "Design and Evaluation of a Wide-Area Event Notification Service," *ACM TOCS*, vol. 19, no. 3, 2001.
- [12] Chand et al., "Scalable Protocol for Content-Based Routing in Overlay Networks," in *NCA*, 2003.
- [13] Kuntschke et al., "StreamGlobe: Processing and sharing data streams in grid-based P2P infrastructures," in *VLDB*, 2005.
- [14] Adler et al., "Channelization problem in large scale data dissemination," in *ICNP*, 2001.
- [15] Papaemmanouil et al., "Extensible Optimization in Overlay Dissemination Trees," in *SIGMOD*, 2006.
- [16] —, "XFlow: Internet-Scale Distributed Stream Processing." Brown University, CS-07-06, Tech. Rep.
- [17] Waldspurger et al., "Lottery Scheduling: Flexible Proportional-Share Resource Management," in *OSDI*, 1994.
- [18] Zhu et al., "Dynamic plan migration for continuous queries over data streams," in *SIGMOD*, 2004.
- [19] Diao et al., "Towards an Internet-Scale XML Dissemination Service," in *VLDB*, 2004.
- [20] Ramasubramanian et al., "Corona: A High Performance Publish-Subscribe System for the World Wide Web," in *NSDI*, 2006.
- [21] Carzaniga et al., "Forwarding in a Content-Based Network," in *SIGCOMM*, 2003.
- [22] Papaemmanouil et al., "SemCast: Semantic Multicast for Content-based Data Dissemination," in *ICDE*, 2005.
- [23] Castro et al., "Scribe: A large-scale and decentralized application-level multicast infrastructure," *JSAC*, vol. 20, no. 8, 2002.
- [24] —, "SplitStream: High-bandwidth Multicast In Cooperative Environments," in *SOSP*, 2003.
- [25] Kostic et al., "Bullet: High Bandwidth Data Dissemination Using An Overlay Mesh," in *SOSP*, 2003.
- [26] Zhuang et al., "Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination," in *NOSSDAV*, 2001.
- [27] Balazinska et al., "Contract-Based Load Management in Federated Distributed Systems," in *SIGMOD*, 2005.
- [28] Zhou et al., "Leveraging Distributed Pub/Sub Systems for Scalable Stream Query Processing," in *BIRTE*, 2006.
- [29] Bonfils et al., "Adaptive and decentralized operator placement for in-network query processing," in *IPSN*, 2003.
- [30] Jain et al., "Design, implementation and evaluation of the linear road benchmark of the stream processing core," in *SIGMOD*, 2006.
- [31] Rodriguez et al., "MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks." in *NSDI*, 2004.
- [32] Loo et al., "Implementing Declarative Overlays," in *SOSP*, 2005.