XFlow: Internet-Scale Extensible Stream Processing

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

June 30, 2008

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 are optimized for a specific 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 can express and optimize a variety of global optimization metrics and constraints. It uses decentralized algorithms that work on localized, aggregated views of the system, while avoiding local optima. The system progressively refines the query deployment, the structure of the underlying overlay network, as well as the statistics collection process, based on the desired objectives. To facilitate light-weight dynamic changes, XFlow uses a publish-subscribe model to decouple sources and clients, as well as processing operators. The result is a loosely-coupled, flexible architecture consisting of multiple overlay trees that can gracefully scale and adapt to churn in system membership and workload.

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

1 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 "macroscopes" [3, 11], 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:

1. *Scalability*: ISM systems need to provide network and workload scalability: they should gracefully deal with 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.

2. *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.

3. *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.

Existing stream processing systems [5, 27] partially address some of the above requirements. These systems support complex continuous queries, while distributed versions [6, 8, 22, 30] allow for queries to be transparently distributed across multiple nodes. However, most current approaches implicitly assume a small number of sources and destinations, and, thus, do not provide mechanisms for "massively" distributing and parallelizing processing to take advantage of the large number of computing elements available in the network (e.g., via operator replication and partitioning). Furthermore, they focus on one performance measure, 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.

This paper describes XFlow, an extensible, highly-scalable and adaptive framework for distributing and optimizing stream processing queries. XFlow creates, maintains and optimizes an overlay network, given dynamic stream sources, clients with stream-oriented complex queries and application-specific performance expectations. The network consists of multiple, potentially overlapping overlay stream dissemination trees, created dynamically depending on the degree of stream sharing. XFlow can express a variety of optimization metrics and constraints. It progressively refines the placement and execution of the operators, the structure of the underlying overlay network, as well as the statistics collection process, to meet the desired objectives.

A key feature of XFlow's optimization framework is that it relies on *localized* state and interactions to reduce the *global* system cost. The optimizations are guided by a set of operator distribution operations (migration, replication, partition) and use localized, aggregated, network and workload statistics. Nodes apply these operations on their neighborhood as well as on specific promising network regions which they discover through dissemination of selective statistics. One of our key results is that even simple aggregations of statistics are sufficient to achieve efficient operation with low overhead (network traffic per node is below 300bytes in a network of 200 nodes), as well as allow XFlow to avoid local optima and converge to near-optimal configurations. Moreover, we employ probabilistic statistics dissemination techniques that manage to keep the network traffic within constant bounds independent of the number of queries and nodes with low performance degradation. This characteristic allows for high scalability and efficiency.

XFlow relies on the pub-sub paradigm [12, 17] as the underlying communication model. This model allows XFlow to effectively decouple 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 stream processing queries [5, 27] 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 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 and simplifies the overall system model while at the same time facilitating (i) the sharing of intermediate processing results and (ii) light-weight dynamic query modifications, such as adding, removing, migrating, and replicating operators.

The contributions of this paper can be summarized as follows:

1. We introduce a novel architecture that consists of multiple, dynamic overlay trees, for data collection, processing, and result dissemination.

2. We present a *generic* cost model that can express a range of optimization goals that evaluate the efficiency of *user queries* as well as various *resource utilization* metrics.

3. We describe a generic distributed *query optimization* framework that relies on 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.

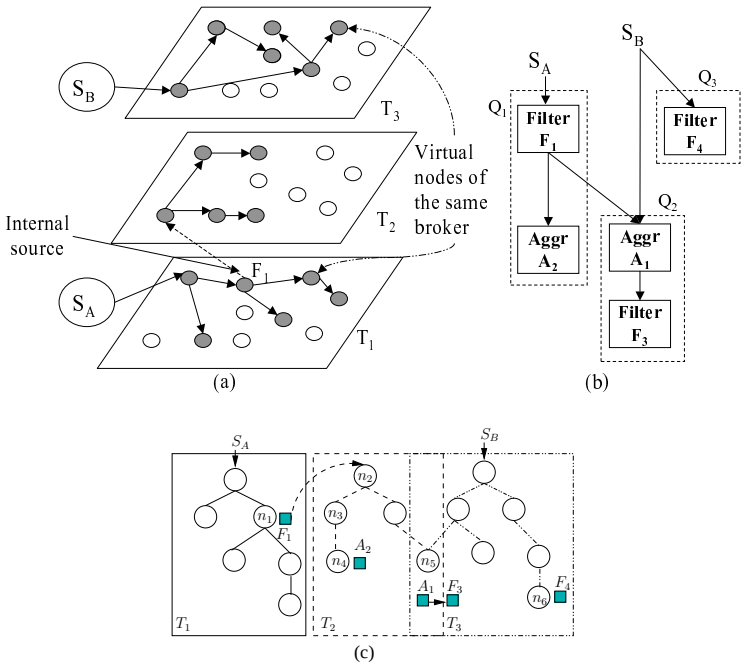


Figure 1: XFlow's system model.

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 that may result from localized operations.

5. We demonstrate the effectiveness, flexibility, practicality of XFlow through a prototype implementation. The results are obtained through experiments processing real-world feeds on 150 PlanetLab sites. We show that XFlow's localized statistics allow the system to converge to near-optimal configurations for a variety of metrics.

The rest of the paper is structured as follows. We describe the architecture of XFlow in Section 2. We introduce our cost model in Section 3 and describe our generic optimization framework in Section 4. We present our experimental results in Section 5, the related work in Section 6, and conclude the paper with final remarks and plans for future work in Section 7.

2 System Model

XFlow consists of an overlay network of cooperating *brokers* (or nodes) providing stream routing and stream-based query processing services (e.g., [5, 27]). *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-based continuous queries on the global schema. Each external source and client has a proxy running on a node acting on behalf of its corresponding entity.

Pub-Sub model. XFlow relies on the pub-sub paradigm as the underlying uniform 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 an overlay tree which distributes it to the subscribed consumers. Nodes hosting interested clients or operators must join this tree. Thus, XFlow consists of multiple overlay dissemination

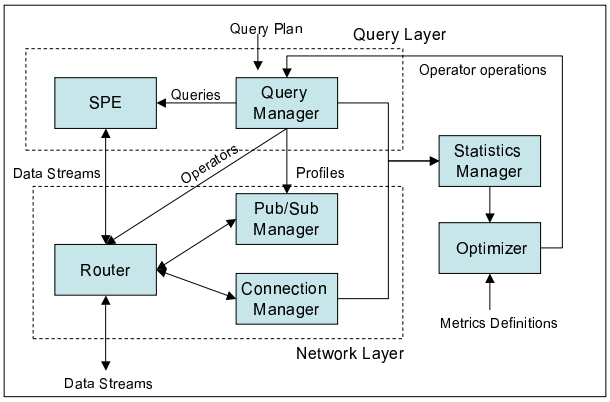


Figure 2: XFlow node architecture.

trees, potentially one for each internal or external source. Figure 1(a) shows a network of three trees: T_1 and T_3 publish the external streams S_A and S_B respectively, while T_2 publishes the output of the operator 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 functionality of the bootstrap can be easily distributed across multiple nodes to improve the availability and scalability. 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 published stream.

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. 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 [24]. 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.

The bootstrap also informs the client's host about the trees publishing its query's inputs. The host node joins these trees, by connecting to a node in the tree. The choice of the node depends on the applications expectations and constraints, e.g., for latency-sensitive applications it will be the closest node, or for applications with fanout restrictions we pick the node with the smallest number of children. Once the node joins the tree, it requests the query's input streams published by that tree. We refer to this request as the *profile* of that 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 node. Using the routing tree created, a node can route streams published through this tree only to its children interested in receiving them.

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 registers a query, the operators of the query subscribe 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 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 as part of the optimization process.

Figure 1(b-c) shows three queries and possible deployment. Q_1's operators are 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.

Node architecture. The architecture of a XFlow node is shown in Figure 2. 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 joins the proper trees. Tree connections are handled by the *connection manager*, while the *router* delivers input streams to the SPE and publishes output streams. It also routes maintenance data required by the connection

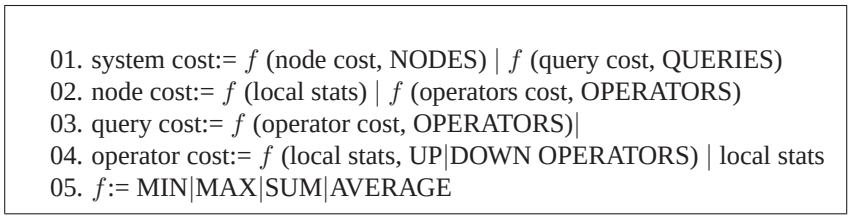


Figure 3: XFlow's cost model.

manager as well as any subscription/subscriptions requests 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*, XFlow identifies operations on the queries that improve the cost metrics and sends any query changes to the query manager.

Tree management *Conceptually*, XFlow creates one tree per each source (internal or external). In practice, 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* application-level 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.

We also employ techniques 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 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 are located on the same node.

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 or merging trees, adapting to membership changes. A simple heuristic could be merging trees that involve the same set of nodes. Such trees could be easily discovered by exchanging bloom filters among tree roots. The problem of effectively grouping data sources and mapping them to a given set of trees has been studied in [7, 28] 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.

2.1 Benefits of XFlow's model

A distinguishing feature of XFlow is its support for dynamic creation of multiple overlapping publish-subscribe trees. XFlow treats query operators as producers and consumers and thus, any optimizations (migrations, create/remove replicas) are handled as subscriptions/unsubscriptions or addition (or removal) of trees. In contrast to existing systems [6, 8, 30], XFlow 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 be scalable and robust in terms of the number of dynamic operator changes as these require no state updates on their neighbors in the query network. Moreover, it can facilitate dynamic stream sharing, since operators can seamlessly subscribe to any stream in the system. Note that high degree of sharing has been previously shown for both external [25] and internal streams [4].

Projects like SPC [20] and IFLOW [22, 23] also used the pub-sub semantics to facilitate reuse of streams. However, XFlow goes beyond centralized implementations of the pub-sub semantics and realizes this loosely-coupled model through *dynamic* creation of multiple overlay trees. 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 [21, 28]. One unique challenge we address is to perform optimizations across multiple trees.

3 Generic cost model

XFlow is generic stream processing system can be customized for the optimization metrics that meet an appli-

cation's performance expectations. In contrast to existing solutions [8, 30] 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 3. In this section, we describe XFlow's approach in detail.

3.1 Optimization metrics and constraints

The performance of distributed stream processing systems is often evaluated by query-related QoS targets [6] as well as resource utilization metrics [8, 22, 30]. The former refers to the efficiency and performance of queries (e.g., maximum output latency), whereas the latter addresses the overhead of in-network processing on the system nodes (e.g., bandwidth consumption, processing load distribution). Below we present our cost model, which is designed to express both metric types, and provide examples of metric definitions using our grammar.

To facilitate the expression of broad spectrum of metrics, each XFlow node evaluates a set of built-in *local statistics* for the overlay links it maintains, e.g., link latency/reliability, bandwidth capacity, as well as the local operators, e.g., input/output rates, selectivity, processing costs. Application designers can combine local statistics and define optimization metrics and constraints.

QoS metrics. XFlow expresses the performance of a query based on the performance of its operators. Specifically, we define the *query cost* as an aggregation of its *operator costs*, where the *system cost* is an aggregation of the query costs. Applications can use the aggregation functions shown in Figure 3 and customize XFlow for various metrics, e.g., average query output latency, query load, query throughput.

XFlow users can define the operator cost based on (i) local statistics or (ii) the aggregation of network statistics. The former case can capture metrics like the processing latency or load of an operator. The latter can express network-related metrics, like the total output latency, which includes the dissemination delay for fetching the input stream from its producer plus its processing latency. Nodes aggregate statistics on the network links connecting an operator to its neighbors in the query plan, that is the location of (i) its upstream operator or (ii) its immediate downstream operator. If there exist multiple upstream or downstream operators across multiple query plans, we define the cost of each one independently and average the costs.

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

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

Resource utilization metrics. XFlow allows also the definition of metrics that measure the overhead on the system nodes. We refer to a node's resource utilization as the *node cost* and is defined based on (i) the local statistics or (ii) the aggregation the local operators' cost. In this case the system cost is the aggregation of the node costs. An example is the processing load of a node, which is the total load of its locally executed operators. Based on this, we define below the maximum processing load across all nodes:

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

Another resource utilization metric is the outgoing bandwidth consumption of a node. We define this as the data its local operators publish (`op out rate`) plus the data it forwards in the network as part of the dissemination trees (`fwd rate`). Hence, the total bandwidth consumption can be expressed as:

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

Constraints. Applications can express constraints on the performance of queries as well as the resource utilization in the system. By using the same model they can define cost metrics for operators, queries or nodes and specify bounds on them, e.g., the maximum processing load, maximum query output latency, etc. For example, XFlow users can express constraints to guarantee sufficient bandwidth capacity on the overlay links connecting an operator with its upstream operator. This is crucial for applications with high input rates. The capacity between two operators is the minimum bandwidth capacity of the network links connecting them:

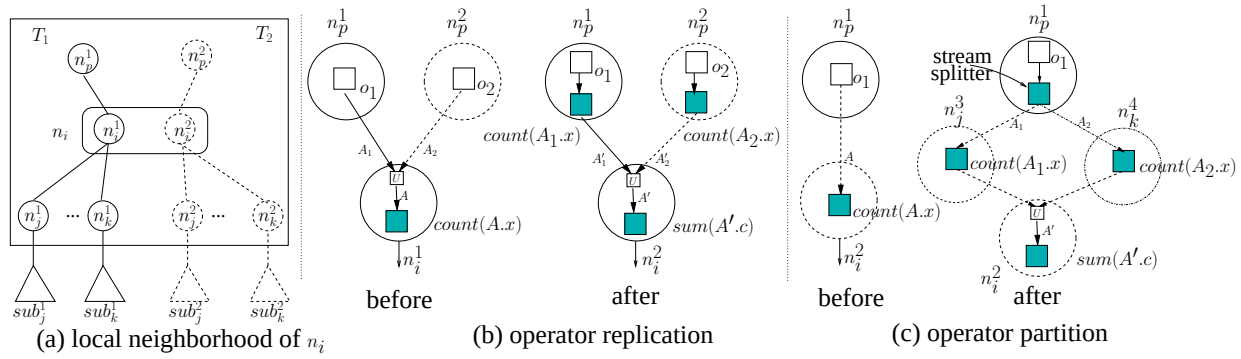


Figure 4: Neighborhood of n_i and operator replication and partition.

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

We note here that generic cost models where also proposed in [22, 29]. In [22] they use a less expressive model that simply sums network link costs. XPORT [29], 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 overhead of the query processing on the network nodes, whereas XPORT focuses only on the dissemination cost. Finally, we express and optimize performance targets and constraints across a general network of *multiple* overlay trees, whereas XPORT limits itself to a single tree.

3.2 Statistics collection

XFlow nodes are customized to collect and aggregate statistics required for the evaluation of the performance metrics and constraints. Each node collects only those statistics related to the cost of its local operators. Specifically, each node n_i maintains *only two* statistical values for every dissemination tree in which it participates: (i) the local statistic value, l_i, and, whenever required, (ii) the aggregation of network statistics, ϕ_i. XFlow exploits the aggregation-based model to reduce the metric collection overhead by computing the aggregating value ϕ_i in a decentralized fashion.

We will illustrate the above with an example. In order to measure the latency for operator A_2 in Figure 1, instead of collecting the latencies of every link between n_4 and n_1, every node measures the latency to its parent, collects the path latency between the parent and n_1 and adds them to get its own latency to n_1, namely the ϕ_i value. Generally, depending on the operator cost definition (upstream/downstream aggregation), each node on the overlay path connecting two operators evaluates the required local statistic on the link to its parent/children in the dissemination tree and combines this with the aggregated metric of its parent/children respectively to derive its final cost value. This approach reduces the statistics collection traffic as nodes evaluate partial results and collaborate in order to derive the final cost metrics.

4 Distributed optimization

XFlow distributes the query operators, aiming to minimize the global system cost. It continuously refines the deployment and execution of the query network, by dynamic combination of operator migration, replication, and partition. 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 nodes 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 cost metrics.

To address these challenges, we designed a general, decentralized framework that does not rely on global information and has low communication overhead. Each node maintains local, aggregated views of its own "neighborhood" and periodically attempts to distribute its local operators across the nodes within the scope of

this neighborhood. We refer to this operations as *localized optimizations*. An advantage of this approach is that non-overlapping neighborhoods can be optimized locally and concurrently.

To avoid local optima, we selectively distribute certain aggregated statistics of the network 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-definition model and the known semantics of the aggregation functions to derive generic properties and equations that quantify the benefits of any candidate optimization. In the rest of the section, we describe our optimizations and then focus on our cost model.

4.1 Operator distribution operations

XFlow nodes distribute their operators across their neighbors in the overlay trees they participate it. For a node n_i, these are its parents and the nodes in the k-level subtrees rooted at n_i, where k is a system parameter. In our current implementation, each such subtree has at most three levels, including also n_i's children across the existing trees. We denote as n_i^k an "logical" instance of node n_i in tree T_k, and Figure 4 shows (a) the neighborhood 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. 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 they are not already part of them). If the operator is an internal source, then the new host publishes the operator's output to 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 resources in the 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.

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. 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. We omit the implementation details of replication for our operators due to space limitations.

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 . Since each replica corresponds

to a different tree, a node can place the replica on its neighbors on this tree. Note that we do not require *all* replicas of an operator to be migrated, only the ones that can improve the 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 placed closer to its own data source, reducing the query latencies.

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.

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 can be applied 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 Local optimizations

XFlow attempts to apply the above optimization operations in the scope of its local neighborhood. An advantage of this approach is that for a given operator (or replica), we reduce the search space of candidate migrations by considering only nodes in the neighborhood of its current host. Furthermore, non overlapping neighborhoods can be optimized locally and concurrently. During the optimization process, nodes will consider migrating their operators to their parents, children or siblings across trees. Similarly for the cases of operator replication and partition, nodes attempt to migrate the new operator replicas within their neighborhood.

We employ a hill-climbing-based local search that applies the best of all possible localized optimizations. 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 localized search has very small communication overhead 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.

4.3 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 selectively disseminates network statistics, on the basis of the optimization metric, that allow nodes to efficiently identify promising non-local operations. These operations focus on specific low cost network areas and we refer to them as *directed* optimizations. We describe this approach in the next section.

4.3.1 Statistics collection

As a result of the statistics collection process, XFlow nodes maintain certain local and aggregated metrics. We *selectively* disseminate these metrics across nodes, allowing the discovery of alternative neighborhoods that could be utilized and improve the system's performance. XFlow relies on the definition of the cost metrics, specifically on the semantics of the aggregation functions, in order to determine: (i) which nodes may be interested in the statistics, and (ii) which statistics could be of potential interest. Moreover, it utilizes the dissemination trees to create filter-based routing paths that forward statistics from their producers only to the interested consumers. Nodes can exploit these statistics in order to identify promising optimization areas.

Statistics selection. As mentioned in Section 3.2, each node n_i maintains, for every dissemination tree in which it participates, the local value l_i and an aggregation of network statistics, ϕ_i. Examples of local values include the processing load of a node or link-based measures, e.g., latency, capacity, reliability, data rate, etc. Aggregated value examples include path latency, path bandwidth bottleneck, outgoing bandwidth consumption.

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

Table 1: Statistics dissemination & filtering. (c: system cost)

We note here that nodes maintain statistics required by the optimization *and* constrained metrics and thus our system can exploit both metrics.

Table 1 summarizes which statistics each node distributes in the network. Every node will forward its local value, which reveals the resource utilization of the node itself or its overlay links. Moreover, the aggregated value ϕ_i may provide some insight on the performance of the overlay *paths* leading to n_i, depending on its aggregation function. Intuitively, when additive functions are used, the cost of the overlay paths will be much higher than the cost of a single link and thus unlikely to be of interest to any nodes. However, for bottleneck-based functions, like MIN or MAX, overlay paths do not necessarily have higher cost than their individual links, making them equally reusable as each individual link. For example, the bandwidth capacity of a path depends on the connection order of its nodes. In these cases, XFlow informs the network about n_i's path properties by forwarding its aggregated value ϕ_i.

Selective dissemination. XFlow applies optimizations that can improve the global system cost. Therefore, statistics are selectively distributed only to nodes that can affect the system cost. Table 1 shows the conditions that should hold in order for a node to receive any statistics. These conditions are agnostic of the actual optimization metric and depend on the aggregation functions for the query and system cost. Intuitively, if the optimization metric is the average query latency, then every node can improve this metric by reducing the processing or network delay of all the queries it serves. On the other hand, if we are interested in the maximum query latency, then only the nodes that serve the query with the highest output latency, c, (or the ones with small difference $\pm\delta$ from the highest latency) will receive the statistics.

Furthermore, nodes are interested only in statistics of network components (links, paths, nodes) that perform better than their own local neighborhood. Table 1 shows the predicates used by each node n_i to filter 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, nodes want to receive statistics about link latencies lower than its local links. In the case of the MIN function, the value ϕ_i provides the lowest value the node is interested in. For example, nodes will receive the link and path bandwidth capacities that are higher that the capacity of its local paths.

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 only k nodes (we refer to them as top-k) which are picket based on the lottery scheduling algorithm [19]. Each nodes value is assigned a number of tickets proportional to its utility. For example, nodes with less load or latency will be given more tickets, thus having higher probability of winning the lottery and hence being selected. The algorithm guarantees no zero probability for selecting any nodes statistics. Our experiments revealed that this approach keeps the statistic traffic within constant bounds independent of the number of queries and nodes with low performance degradation.

Statistics Routing. XFlow uses the structure of its data-flow network to distribute network statistics across nodes. It avoids flooding the network by using each node's statistics filter to construct predicate-based routing paths that filter out unwanted statistics as early as possible. For example, there is no benefit in distributing the load statistics of the most loaded node in the system.

To construct these filtering paths, nodes propagate their statistics filters (Table 1) to their parents. Each node aggregates the filters of its children in each tree and propagates the aggregated filter upstream towards the root. Nodes are aware of the interests of their descendants and can efficiently filter statistics and selectively route them to the interested nodes. Since our overlay network consists of inter-connected trees, every participating node is guaranteed to receive the statistics.

4.3.2 Directed operations

Our optimization framework uses the collected statistics to discover specific low-cost neighborhoods that can be utilized to improve the system performance. We categorize these neighborhoods into two classes, *intra-tree* and

inter-tree neighborhoods.

Intra-tree neighborhoods. Dissemination trees connect nodes that receive and process the same data stream. Moreover, 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.

XFlow goes beyond simple operator migration and exploits the network statistics to replace existing links/paths with ones that have a better performance. This is implemented by migrating *both* operators connected through the old link/path to the two ends of the new link/path in a single optimization step. This operation could lead to faster improvement as it does not rely on single operator migrations but instead identifies beneficial overlay paths for connecting pairs of operators. Examples include replacing a node's path to the root by a path with better delay, capacity, or reliability.

4.4 Evaluating global cost changes

Our optimization framework includes a generic cost model that quantifies the expected benefit of an operation. We exploit the fact that our 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 these entities, we derive the metric-independent aggregated state required to quantify an optimization step with small communication overhead. For the purposes of illustration, we will describe our approach with respect to the SUM and MIN aggregation 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 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 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 n_i participates and every tree in which it publishes data.*
2. *If the cost of an operator 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 n_i participates.*

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: $G_i = \cap_{o_j \in D_i} E_j$.*

Let us consider the query plan and its distribution in Figure 5, 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 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 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 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.*

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

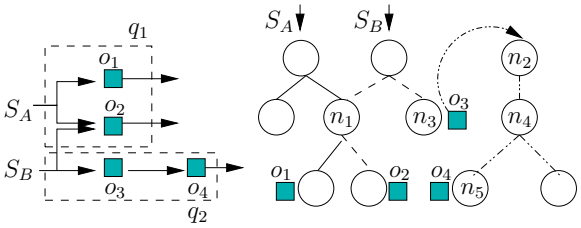


Figure 5: Example of operator distribution.

For example, if we aggregate operators' and queries' cost with the SUM function, then:

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

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.

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

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

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

$$\begin{aligned} b_\alpha \quad = \quad & \min_{n_i \in F_\alpha} \{ \min_{q_m \in Z_i} \{qc_m + \Delta qc_m\}, c(G_i) + \Delta c(G_i), \\ & \min_{q_m \notin Z_i, q_m \notin G_i} \{qc_m\}\} - c \end{aligned} \tag{3}$$

where c is the current system cost.

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

4.4.1 Impact on the query cost

We begin this section by evaluating the change on the dependent operators of a node and continue with the change on query costs and the aggregated dependent 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 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.*

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 by MIN).

To explain this, we'll use the example in Figure 5, 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 Δoc_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 5 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 λ. The λ 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 as shown in Table 3. 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.

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 Δqc_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 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 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 τ 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.

Impact evaluation steps. The above properties allow us to estimate the benefit of an optimization through a sequence of steps. First, we evaluate the change on the new cost of the migrated operator (or replica) by aggregating the local metrics for its potential new neighbors. Given a change on the operator cost, we can quantify the impact on the query costs. In the next step, we estimate any changes on the local values of any affected nodes. 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.

4.5 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. 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 operator's host node. This hill-climbing optimization search ensures improvement in every step, assuming there exists at least one beneficial operation.

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

Oper Aggr	SUM		MIN	
Query Aggr	$\|Z_i\|$	$O(\|Q\|)$	$h_i(o_j), \forall o_j \in O_i$	$O(\|O\|)$
SUM	$\|G_i\|$	$O(\|Q\|)$	$h_i(o_j), \forall o_j \in D_i$	$O(\|O\|)$
			$\|E_j\|, \forall o_j \in D_i$	$O(\|O\|)$
			$c(G_i)$	$O(1)$
MIN	system cost	$O(1)$	system cost	$O(1)$
	$c(G_i)$	$O(1)$	$c(G_i)$	$O(1)$
	$\beta_i(j), \forall q_j \in Z_i$	$O(\|Q\|)$	$h_i(o_j), \forall o_j \in O_i$	$O(\|O\|)$
	$B_j, \forall o_j \in D_i$	$O(\|O\|)$	$\beta_i(j), \forall q_j \in Z_i$	$O(\|Q\|)$
			$h_i(o_j), \forall o_j \in D_i$	$O(\|O\|)$
			$B_j, \forall o_j \in D_i$	$O(\|O\|)$

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

after the operation is applied the node resumes the data flow. To handle migration of stateful operators we adopt exist solutions [34].

Optimization state & traffic. Table 4 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 4.4. 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. 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.

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 changes on the local metrics of the affected nodes and on the operator costs. 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 between an operator's candidate location and the upstream operator. They collect the link latencies from the nodes connecting the candidate location to the location of the upstream operator. We omit the details due to space limitations.

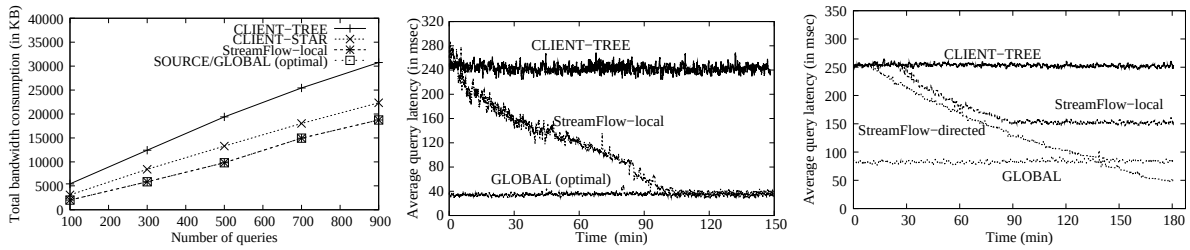
Batching optimization steps. XFlow allows multiple operations to be applied during a single period, e.g., migrating multiple operators. This could speed up convergence to better configurations. We use a standard best-first-search-like algorithm for identifying an effective set of operations. At each 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 and execute this process iteratively for tunable but fixed number of steps.

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 with the skew parameter set to 0.97. Each input stream is an RSS feed pulled from its source with a 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 and manipulator 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 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 and leads 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 picking as their parent a random node that respects the constraints from the



(a) Total bandwidth consumption. (b) Average query output latency. (c) Query latency & load constraints.

Figure 6: Convergence for different metrics ($|N|$=100, $|Q|$=500). (a) Total bandwidth consumption *converges* to optimal placement using only local optimizations. (b) Average query latency *converges* to optimal placement using local optimizations. (c) Load constraints prevent migration of all queries to the source locations. Directed optimizations allow XFlow to perform better than GLOBAL.

ones already in the tree.

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 3). Our experiments demonstrate XFlow's effectiveness, as it manages to improve these metrics significantly over a sequence of local and directed optimization operations.

5.1 Extensibility and Efficiency

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, 30]). 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 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 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 allow XFlow to achieve the optimal placement or a configuration with performance better than that of GLOBAL.

Bandwidth consumption. Figure 6(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 manualy 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 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 by using only it local optimizations (*XFlow-local*). It converges 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. We note here that previous approaches [10, 30] reported solutions that can achieve only 10-14% from the optimal placement.

Query latency. Figure 6(b) shows the average query output latency for 500 queries deployed on 100 PlanetLab sites during the date 10/11/07. We compare XFlows 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 the 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 node where the client is connected. 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 6(b) shows that XFlow manages to converge very close to the optimal configuration after a number of local 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. Moreover, nodes limit their fanout in the tree, as a large fanout would increase the overhead of processing and matching incoming tuples going to downstream nodes. Thus, trees with height higher than one will be constructed. We created such a network by imposing a upper bound of five (5) CPU cycles per time unit on the processing load of each node. Figure 6(c) shows the average output latency of 500 queries on 100 PlanetLab nodes. In this case, not all operators are be migrated to the root since this would violate its upper load limit.

We use this constrained metric to demonstrate the effectiveness of directed optimization. Figure 6(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 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 do not rely only on their neighbors for gradually improving the system's performance. Network statistics disseminated by their peers informs them about the performance of other nodes, i.e., their latency and load. Hence, XFlow directly migrates operators to nodes with less load and better location. These directed operations allow our system to converge to a configuration that outperforms GLOBAL.

Processing Load. We also studied the maximum processing load across all nodes for 500 queries (Figure 8). 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 which places every new query at the least loaded node. For this metric, we also computed the OPTIMAL placement using an exhaustive search of the solution space.

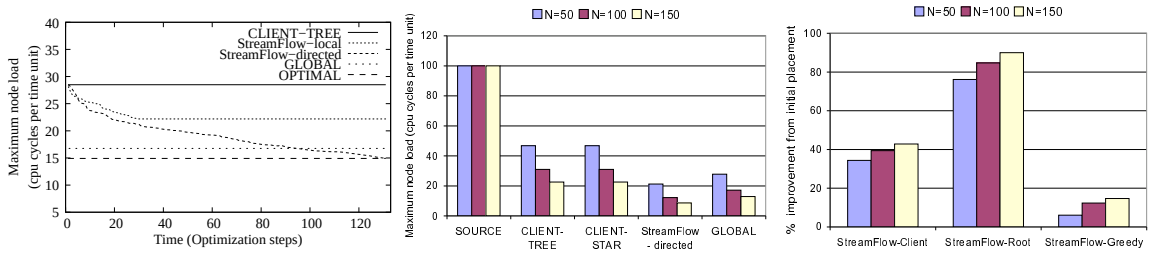
Figure 7(a) shows that XFlow converges to the OPTIMAL when directed optimizations are used. If only localized optimizations are applied, XFlow can improve its optimization metric but converges to a local optimum. However, by using the network statistics, the most loaded node can easily discover less loaded nodes and migrate part of its local processing, reaching eventually the optimal resource allocation. Moreover, as network size increases (Figure 7(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.

Effect of initial placement. We use the maximum load metric to study the effect of the initial placement of operators. Figure 7(c) 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 8). 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.

Probabilistic statstics. We used the maximum load metric to evaluate the effect of our probabilistic statistics dissemination. Figure 8 shows the progress of our optimization when we propagate all statistics (k=100) and the cases where only the top-40 and top-80 values are sent. These values are picked as described in Section 4.3.1. 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.

5.2 Scalability

In Section 4.5 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 (*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. Finally, to support directed optimizations, nodes selectively disseminate certain statistics (Section 4.3.1).



(a) Local & directed optimizations ($|N|$=100).

(b) Different network sizes.

(c) Different initial placements ($|N|$=100).

Figure 7: Maximum node load for 500 queries. (a-b) Non localized optimizations allow XFlow to outperform GLOBAL and *converge* to OPTIMAL. (c) Improvement for different initial placements. XFlow improves even the best initial placement (Greedy) by up to 28%.

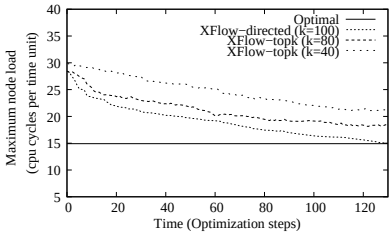


Figure 8: Optimization with probabilistic statistics for maximum node load for 500 queries ($|N|$=100). Directed optimizations using only top-k statistics for various k values. XFlow improves the maximum processing load and converges close to OPTIMAL even with limited statistical information.

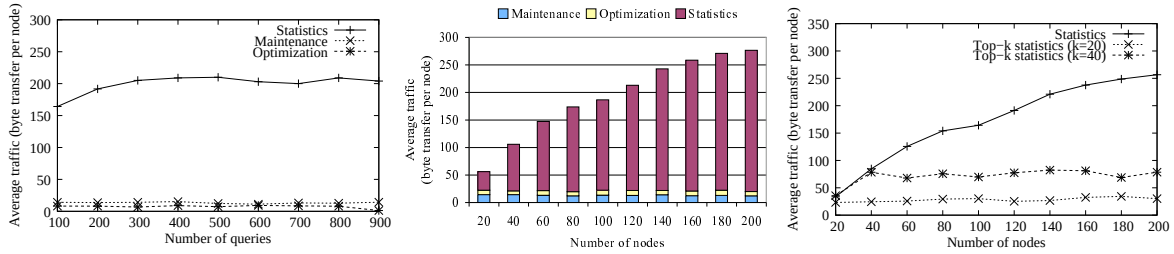
XFlow aggregation model allows nodes to maintain and exchange state of small size. This allows our system to be highly scalable with the number queries and nodes in our network. We demonstrate XFlow's scalability in Figure 9 which shows the traffic incurred for the maximum processing load metric. This traffic includes *optimization traffic*, the *maintenance traffic* (described in Section 4.5), and the statistic disseminated to support directed optimizations ((Section 4.3.1).

Figure 9(a) shows that XFlow scales with respect to the query set since the network traffic remains constant with the number of queries. Figure 9(b) shows the total network traffic for increasing number of nodes. Larger networks require statistics to be disseminated to more nodes, increasing the traffic with the number of nodes. However, even in this case the maintenance and optimization traffic do not increase and the overhead incurred by statistics dissemination. To reduce this overhead we propagate statistics using the probabilistic technique described in Section 4.3.1. Figure 9(c) compares the cases where all statistics all disseminated and when only the top-20 and top-40 are selected. The experiments reveal that the probabilistic dissemination keeps the statistics traffic independent of the number of nodes. Hence, XFlow's overhead is independent of the number of queries and nodes, demonstrating the scalability and efficiency of our approach.

Figure 10(a) shows the average optimization, maintenance and statistics data sent per node for the metrics we implemented. The results reveal that XFlow has very small overhead: it requires less than $240Bytes$ in the worst case. The optimization traffic goes up up $90Bytes$. More specifically, for the query latency, each node periodically sends its latency from the root of each tree to its children, in order for them to evaluate their own latency. Similarly for bandwidth, nodes send their input rates to their parent in each tree, while they calculate their processing cost locally. However, data is exchanged in order for nodes to evaluate their optimization state (Table 4), e.g., the maximum load of their subtree and in the network apart from their subtree.

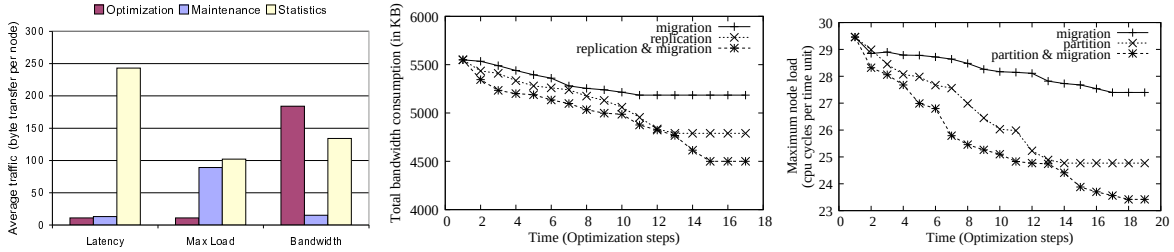
During optimization, for the query latency metric, nodes collect only the latency of all candidate locations for their operators or replicas, while for the maximum load metric, nodes need to know the current load of the candidate locations. Hence, the traffic for these cases is low. However, for the 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. In any case this traffic does not go beyond $180Bytes$.

Finally, regarding the statistics related traffic, there is only an overhead of $130-240Bytes$. For the latency and bandwidth consumption, nodes send their link's latencies and outgoing rate to every node with higher cost than



(a) Overhead for different number of queries. (b) Overhead for different network sizes. (c) Overhead reduction from probabilistic statistics.

Figure 9: 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 ($< 240bytes$ per node). (b) Statistics overhead increase with the network size but remains $< 300bytes$. Maintenance and optimization traffic remains *constant* with the number of nodes. (c) Probabilistic statistics dissemination keeps statistics traffic *constant* for different network sizes.



(a) Optimization overhead ($|N|$=100, $|Q|$=500). (b) Operator replication ($|N|$=100, $|Q|$=500). (c) Operator partition ($|N|$=100, $|Q|$=500).

Figure 10: (a) Network traffic for different metrics. Nodes exchange at most $190Bytes$ during the optimization and the statistics dissemination requires less than $240Bytes$. (b)-(c) Bandwidth consumption with replication and partitioning. Combining migration and replication/partitioning can lead to better resource utilization (b) and load distribution (c).

their local cost, while for the max load they forward their processing load, but only to the node with the highest load in the system. XFlow aggregation model allows nodes to maintain and exchange state of small size. This allows our system to be highly scalable with the number queries and nodes in our network. We demonstrate XFlow's scalability in Figure 9 which shows the traffic incurred for the maximum processing load metric. This traffic includes *optimization traffic*, the *maintenance traffic* (described in Section 4.5), and the statistic disseminated to support directed optimizations ((Section 4.3.1). This traffic is a measure of the optimization overhead.

5.3 Operator replication and partition

We now study our operator execution rules. In this section we examine first replication and then focus on operator partitioning.

Replication. Figure 10(b) shows the improvement on bandwidth consumption when replication is used on 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 with migration is very limited, since moving an operator in one tree might not yield any benefit, as this node may not be closer to the root of 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 combined. In this case, XFlow migrates each replica independently and closer to each stream's source, if this reduces the data forwarding.

Partition. Figure 10(c) shows results for the operator partition case when we try to improve 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, these replicas can move independently in their respective trees, utilizing more processing resources and improving the performance metric.

6 Related work

Data Dissemination. Distributed publish-subscribe systems have been proposed for dissemination of XML messages [16, 17, 19, 31] or relational data [13, 28]. These systems focus on either reducing the bandwidth us-

age [17, 28], improving the resource allocation [31], or providing efficient subscription processing mechanisms [13, 16, 18]. Moreover, they commonly support subscriptions with only predicate-based filters, so they cannot express complex stream processing queries. Application-level multicast systems [14, 15, 21, 35] 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 directly relevant to distributed stream processing solutions like Borealis [6] and Medusa [9]. Neither Borealis nor Medusa 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 SAND [8] and SBON [30] which focus on minimizing the bandwidth usage. Very recently, COSMOS [33] combined the pub-sub model with stream processing in order to improve load balancing among nodes. Finally, in [10] 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 constraints that express the desired query performance and resource utilization tradeoffs. It also provides a self-tuning stream flow overlay that can deal 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 a variety of operations, like operator migration, replication and partition, and overlay path replacement. Note that all the above systems can support run-time operator migration but not replication or partitioning. Our experimental results reveal that these operations can lead to efficient, near-optimal configurations when combined with exchange of selective aggregated statistics.

Extensible systems. The IFLOW messaging framework [22, 23] allows applications to express their own performance requirements. However, their can only express metrics as the sum of link costs. Moreover, they employ a hierarchical solution for assigning stream flows to network edges, where each level of the hierarchy is configured by a centralized, exhaustive search of the solution space. Our cost model is more expressive and can support 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.

Recent network-oriented efforts, such as MACEDON [32] and P2 [26], proposed systems promoting the advantages of generalization. MACEDON and P2 construct overlay networks by abstracting over commonalities present in most overlay construction algorithms. Finally, XPORT [29] 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 3.1. 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.

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. XFlow can express a variety of query optimization and resource utilization metrics, which it improves through a low-overhead metric-independent optimization framework. 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.

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.We believe we can optimize them them as well as we do our built-in operators 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] Ahmad et al. 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] Bonfils et al. Adaptive and decentralized operator placement for in-network query processing. In *IPSN*, 2003.

[11] Campbell et al. IrisNet: an internet-scale architecture for multimedia sensors. In *MM*, 2005.

[12] Carzaniga et al. Design and Evaluation of a Wide-Area Event Notification Service. *ACM TOCS*, 19(3), 2001.

[13] Carzaniga et al. Forwarding in a Content-Based Network. In *SIGCOMM*, 2003.

[14] Castro et al. Scribe: A large-scale and decentralized application-level multicast infrastructure. *JSAC*, 20(8), 2002.

[15] Castro et all. Splitstream: High-bandwidth multicast in cooperative environments. In *SOSP*, 2003.

[16] Chan et al. Efficient filtering of XML documents with xpath expressions. In *ICDE*, 2002.

[17] Chand et al. Scalable Protocol for Content-Based Routing in Overlay Networks. In *NCA*, 2003.

[18] Diao et al. Query Processing for High-Volume XML Message Brokering. In *VLDB*, 2003.

[19] Diao et al. Towards an Internet-Scale XML Dissemination Service. In *VLDB*, 2004.

[20] Jain et al. Design, implementation and evaluation of the linear road benchmark of the stream processing core. In *SIGMOD*, 2006.

[21] Kostic et al. Bullet: High bandwidth data dissemination using an overlay mesh. In *SOSP*, 2003.

[22] Kumar et al. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, 2005.

[23] Kumar et al. IFLOW: Resource-aware overlays for composing and managing distributed information flows. In *EuroSys*, 2006.

[24] Kuntschke et al. StreamGlobe: Processing and sharing data streams in grib-based P2P infrastructures. In *VLDB*, 2005.

[25] Liu et al. Client Behavior and Feed Characteristics of RSS, A Publish-Subscribe System for Web Micronews. In *IMC*, 2005.

[26] Loo et al. Implementing declarative overlays. In *SOSP*, 2005.

[27] Motwani et al. Query processing, approximation, and resource management in a stream management system. In *CIDR*, 2003.

[28] Papaemmanouil et al. Semcast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.

[29] Papaemmanouil et al. Extensible Optimization in Overlay Dissemination Trees. In *SIGMOD*, 2006.

[30] Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[31] Ramasubramanian et al. Corona: A high performance publish-subscribe system for the world wide web. In *NSDI*, 2006.

[32] Rodriguez et al. MACEDON: Methodology for automatically creating, evaluating, and designing overlay networks. In *NSDI*, 2004.

[33] Zhou et al. Leveraging distributed pub/sub systems for scalable stream query processing. In *BIRTE*, 2006.

[34] Zhu et al. Dynamic plan migration for continuous queries over data streams. In *SIGMOD*, 2004.

[35] Zhuang et al. Bayeux: An Architecture for Scalable and Fault-tolerant Wide-area Data Dissemination. In *NOSSDAV*, 2001.