Network-Aware Query Processing for Stream-based Applications

Yanif Ahmad Uğur Çetintemel
Computer Science Department, Brown University
{yna, ugur}@cs.brown.edu

Abstract

This paper investigates the benefits of *network awareness* when processing queries in widely-distributed environments such as the Internet. We present algorithms that leverage knowledge of network characteristics (e.g., topology, bandwidth, etc.) when deciding on the network locations where the query operators are executed. Using a detailed emulation study based on realistic network models, we analyse and experimentally evaluate the proposed approaches for distributed stream processing. Our results quantify the significant benefits of the network-aware approaches and reveal the fundamental trade-off between bandwidth efficiency and result latency that arises in networked query processing.

1 Introduction

The need for widely-distributed query processing is becoming increasingly apparent with the proliferation of applications that require sophisticated processing of data generated or stored by large numbers of distributed sources (such as data streams generated by sensor networks or Internet-based data collections). Existing query processing approaches commonly address relatively small-scale systems and fail to exhibit good network scalability, a design goal that has only recently started to receive attention within the database community [8, 12] and that we believe will be central to next-generation data processing systems.

In this paper, we study the benefits of *network awareness* when processing queries in widely-distributed environments such as the Internet. We argue that exploiting knowledge of the underlying network characteristics (such as topology and link bandwidths) can significantly improve the efficiency of network-bound query processing. We present *network-aware* operator placement algorithms that utilize such characteristics to identify the network locations where the operators of a given query plan should be executed. The algorithms differ in which nodes they consider as candidates for operator placement and how they take network knowledge into account.

Specifically, we present two novel network-aware approaches for push-based continuous queries and distributed stream processing. The first approach uses heuristics that exploit pair-wise server communication latencies. The second approach extends the first one by identifying and involving in processing "well-located" servers that would otherwise not participate in the process, thereby implementing "in-network" query processing.

We describe the basic design of a distributed query processing system, built on top of a Distributed Hash Table (DHT) [22, 26], that implements the proposed placement algorithms. We have fully implemented the system and the algorithms, and use the code base to conduct a detailed emulation study under realistic network models. Our results show that, compared to representatives of traditional network-unaware approaches, the proposed approaches can significantly reduce the overall system bandwidth consumption, a key efficiency metric for large-scale networked systems. Furthermore, the algorithms can be tuned to satisfy target query-result latency bounds, typically at the expense of extra bandwidth consumption. Even though our work assumes push-based continuous queries, the results are more general and apply to pull-based pipelined queries as well.

Our work is done in the context of the *SAND* (*S*calable *A*daptive *N*etwork *D*atabases) project that strives to develop a highly-scalable and adaptive network-oriented database system and the *Borealis* project that strives to extend core data-stream processing functionality to heterogeneous distributed environments.

The rest of the paper is organized as follows: Section 2 describes the basic system and network model that we assume throughout the paper. Section 3 presents the centralized versions of the network-aware operator placement algorithms. Section 4 describes how these algorithms can be effectively implemented in a distributed manner, leveraging basic DHT primitives. Section 5 analyzes the processing and message complexity of both the centralized

This work has been supported in part by the National Science Foundation under the ITR grant IIS-0325838.

**Proceedings of the 30th VLDB Conference,
Toronto, Canada, 2004**

and distributed approaches. Section 6 characterizes the efficiency and effectiveness of the approaches using an emulation. Section 7 summarizes prior research relevant to our work and highlights the main differences. Finally, Section 8 provides concluding remarks and directions for future research.

2 Basic System Model

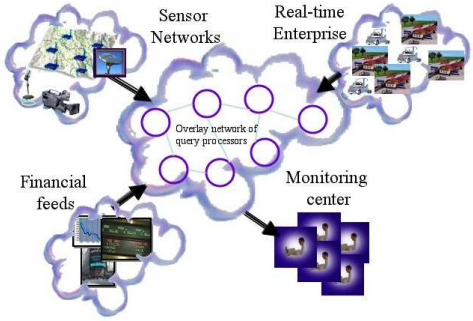

Figure 1: A widely-distributed stream processing environment.

We consider a widely-distributed query processing environment with geographically dispersed data sources that produce high-volume, fast data streams. Our target applications require sophisticated processing (e.g., fusion, aggregation, correlation) of the source data streams. Our system consists of a large number of cooperating servers, capable of executing stream-oriented query operators (e.g., [4, 5, 18]). The servers are organized into an overlay network and collectively provide processing services for multiple concurrent stream-based applications.

Figure 1 illustrates the basic application and environmental model. In the figure, the small clouds represent data sources (such as sensor networks and financial feeds) and the big cloud represents the networked stream processing system. The system transparently partitions the queries that describe the processing requirements of applications across its nodes, multiplexing its distributed resources in order to improve the overall performance, scalability, and availability.

2.1 Data and Processing Model

For our purposes, a data stream is a continuous sequence of tuples generated by a stream source (or simply *source*). Data streams are processed according to a processing network, which is a collection of (potentially overlapping) processing trees. A *processing tree* is a directed dataflow-style tree of stream-oriented operators that collectively represent a query plan. Users build such query networks either directly using a GUI, through a scripting language (e.g., [4, 12]) or indirectly through the compilation of high-level SQL-like language statements (e.g., [18]). In either case, queries are built using a standard set of operators [1, 18], which can also include user-defined functions. We assume that the specified processing tree is final and do not consider further semantic opportunities. We plan to explore this in future work. The output tuples that result from processing are delivered to an *application proxy* (or simply proxy) that is responsible for forwarding these tuples to the relevant application(s).

2.2 Server and Network Model

Our system consists of an application-level overlay network (e.g., [19, 20, 22, 26]) of geographically dispersed cooperating servers, interconnected physically by IP networks, and logically, through a DHT infrastructure. We assume that the overlay servers communicate via IP unicast.

The DHT infrastructure acts as a networking substrate, providing localized knowledge of the server space and flexibility in our routing and search operations. This flexibility is core to supporting a variety of placement mechanisms when investigating algorithmic tradeoffs in optimizing bandwidth usage. While the algorithms presented here are not reliant upon a specific DHT, we describe our system generating placement overlays on top of the Tapestry system [26].

In Tapestry, overlay servers are assigned an identifier obtained from securely hashing the servers' IP addresses. Overlay servers co-ordinate themselves into a connected network, and maintain local routing tables referring to servers whose addresses prefix match the local address, at varying lengths. Using this routing table construction, Tapestry offers a lookup mechanism designed to reach its destination in $\mathcal{O}(\log n)$ overlay hops (where n denotes the number of overlay participants), while ensuring a bound of $\mathcal{O}(n \log n)$ on the system space requirements. Furthermore, Tapestry servers' routing tables are created utilizing network locality information, making it well-suited for our purposes. Incorporating locality cues reduces the routing stretch factor, and as such, will transitively affect stretch factors in our placement overlays, and the response time for its construction.

2.3 Control Model

For improved scalability and parallelism, we use a distributed control model. For each processing tree, we create a corresponding *control tree* of coordinator nodes as follows. When a processing tree is up for execution, it is logically partitioned into a number of subtrees, called *zones*. Each zone is assigned a *coordinator* node that is responsible for the placement (and periodic dynamic re-placement) of the operators in the zone. Coordinators are also responsible for ensuring correct and highly-available execution of their zones. The application proxy is always assigned as the root coordinator and decides how many zones to create. Each zone is then assigned a *zone id*, which is used to identify the node (through the DHT) that will serve as the coordinator for the zone. Coordinators communicate among themselves and the nodes that execute the operators in their zones in order to dynamically optimize processing.

Figure 2 provides a high-level view of this basic model, illustrating a processing tree and the corresponding control amd processing networks overlayed on top of the physical IP network. The processing tree is partitioned into three zones, each assigned to a coordinator node. Each coordinator decides on the placement of the nodes in its zone using

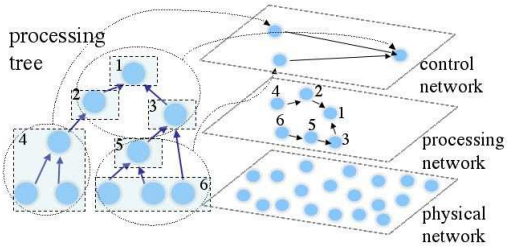

Figure 2: A processing tree and the corresponding control and processing overlay networks.

appropriate algorithms. An advantage of this approach is that zones can be optimized locally, concurrently and asynchronously.

3 Operator Placement

In this section, we first formally define the operator placement problem for processing widely-distributed data streams. We then describe three algorithms to construct processing overlay networks. The first algorithm, *Edge*, considers only source locations and the proxy for placement. Edge is an adaptation of the standard pull-based site-selection approaches to push-based streaming data. The second algorithm, called *Edge+*, is a network-aware version of Edge. Edge+ takes into account the pair-wise "network distances" (i.e., transmission latencies) between the servers. The third approach, called *In-Network*, considers a carefully selected subset of all network locations, in addition to the sources and the proxy, when making placement decisions. Finally, we describe an extension that imposes bounds on the processing delays.

In order to focus on networking-related costs, we ignore operator processing costs and other related overheads that arise during query execution. We also assume that the placement algorithms are applied periodically to dynamically adapt to changes.

3.1 Problem Statement

Consider a processing tree $T = \{O, A\}$, defined by a set of stream-oriented operators O, and their connected inputs and outputs A. Our model also consists of a network topology $G = \{V, E\}$, of peer nodes V, and their links E. A function, β, on processing tree edges yields the data-flow bandwidth between operators. In our model, β is defined as a product of the input bandwidths and selectivity (or more generally productivity) of the input operators. Our goal is to place each operator at a peer, while minimizing the bandwidth utilized in the network.

We assume that the leaves in our processing tree represent stream sources, and that a function DHT yields the sources' locations on the topology. We attempt to find a mapping function λ defined on the operators, O, yielding network locations in V. Our objective on the network edges between the resulting placements follows:

$$\min_{\lambda} \sum_{a \in A} c(a) \tag{1}$$

$$\text{s.t.} \qquad \lambda(l) = DHT(l) \qquad \forall l \in leaves(T) \tag{2}$$

where

$$c(a) = \begin{cases} 0 \text{ if for } a = (m, n) : \lambda(m) = \lambda(n) \\ \beta(a) \text{ otherwise} \end{cases} \tag{3}$$

In our cost function $c(a)$ above, we state that processing tree edges, $a \in A$, incur no cost if both endpoints of an edge are placed at the same location, and a cost of $\beta(a)$ otherwise. We use the terms *tree cost* and *overlay cost* to refer to the functions β and c, respectively. We regard λ as defining a query overlay, whose members are given by the range of λ, and edges are the network edges corresponding to mapped tree edges. Figure 3 provides an overview of the symbols used in this model, along with their definitions.

Symbol	Definition
o	processing tree operator.
c_i	arbitrary child of operator o.
$\beta(a)$	cost of processing tree edge a (*tree cost*)
$c(a)$	cost of mapped tree edge a (*overlay cost*)
$d(u, v)$	network distance between locations u and v
$\gamma(o, v)$	cost of operator o at location v
$\lambda(o)$	mapped location of operator o
$\phi(o)$	candidate location set of operator o

Figure 3: Model terminology.

3.2 Edge Placement

The *Edge* algorithm strives to find a good placement when the candidate locations are constrained to the sources and the proxy. Because the optimal solution of a simpler version of the problem is known to be NP-complete [16], we propose a greedy algorithm that traverses the processing tree operators in post-order, optimizing progressively larger subtrees as it proceeds.

As our algorithm encounters operators in its traversal, it places operators at the minimal cost location identified using one of three cases: placement at (1) one of its children's locations, (2) a *common* location, or (3) the proxy's location. We now describe how to compute the partial overlay's cost at each of these cases in greater detail. Note that the cost of an operator depends upon the placement of its children. We use the term *configuration* throughout the text to denote the placement and cost of an operator, and the placement of its children (and subsequently the entire subtree).

Let us consider an operator o with children $\{c_1, \ldots, c_n\}$. In case (1) above, we place the operator at a location that maximizes the total tree cost between the operator and all of its children. This configuration eliminates the maximal tree cost from our overlay cost. Formally, this is:

$$\lambda'(o) = \arg \max_{v \in \bigcup_{i=1}^{n} \lambda(c_i)} \sum_{\{c_i : \lambda(c_i) = v\}} \beta(o, c_i) \tag{4}$$

This local minimization clearly does not yield a globally optimal processing overlay. In case (2), we check for locations where we may potentially place all children of an operator. Placing an operator and its children at this common location ensures that all edges between the operator and its children incur zero overlay cost. We define cl, the

set of common locations as an intersection of each child's dl (the set of descendant leaf locations) of an operator o:

$$dl(o) = \bigcup_{l \in leaves(o)} \lambda(l) \ , \ cl(o) = \bigcap_{i=1}^{n} dl(c_i)$$

To facilitate this heuristic, we need to compute the cost of placing an operator at a specific location:

$$\begin{aligned} \gamma(v,o) &= \infty \quad \text{if } o \in leaves(T) \wedge v \neq DHT(o) \\ \gamma(v,o) &= \sum_{\{c_i : v = \lambda(c_i)\}} \gamma(v, c_i) + \qquad \text{otherwise} \\ &\sum_{\{c_i : v \neq \lambda(c_i)\}} \min\{\gamma(\lambda(c_i), c_i) + \beta(o, c_i), \gamma(v, c_i)\} \end{aligned} \tag{5}$$

This is γ (Equation 5), yielding the overlay cost of a subtree rooted by operator o, with o placed at v. We consider two configurations to achieve this, the minimized sum (over all children) of either (i) the existing child configuration cost and the cost of any additional edge required to place the operator at v, or (ii) the cost of placing the child at v.

In the case (3), we consider the cost of placing an operator at the proxy's location. Our motivation here is that, in placing an operator, we do not account for its outgoing tree cost. Considering this configuration helps when tree costs are higher near the root of the processing tree. In this scenario, all operator-child tree edges add to the overlay cost (assuming that the proxy is not a stream source).With a proxy location r, we now have our final mapping function, λ, for an operator o:

$$\lambda(o) = \begin{cases} DHT(o) & \text{if } o \in leaves(T) \\ \arg \min_{v \in cl \cup \{\lambda'(o), r\}} \gamma(v, o) & \text{otherwise} \end{cases} \tag{6}$$

3.3 Network-Aware Edge Placement: Edge+

The Edge algorithm does not utilize any network knowledge in making placement decisions. To better model costs on a large-scale network, we extend the original approach to include a symmetric distance function, d, that represent the network latencies between locations. In the extended algorithm, *Edge+*, this change is reflected in the cost function. Our modified overlay cost is a product of tree cost and the distance between overlay edge endpoints. We replace Equation 3 with:

$$c(m,n) = \begin{cases} 0 & \text{if } \lambda(m) = \lambda(n) \\ \beta(m,n) \cdot d(\lambda(m), \lambda(n)) & \text{otherwise} \end{cases}$$

The control flow of Edge+ is similar to that of Edge. In Edge+, we consider configurations from the three cases in Edge, and additionally examine a distance-oriented case. In this fourth case, given a child's descendant locations, we selectively enumerate location permutations that meet a distance criterion. We select permutations whose total separation is less than than the total separation of the operator configuration selected by the Edge algorithm. Formally, we choose configurations, $\{v_i \in dl(c_i)\}^n$, (of cardinality n), such that:

$$\sum_{i=1}^{n} d(v_1, v_i) \leq \sum_{i=1}^{n} d(\lambda(o), \lambda(c_i))$$

Above, we see a configuration's total separation is a sum of distances from one location to all other locations. Note we leverage our symmetry assumption here. Our intuition in selecting these configurations is to optimize for cost by reducing distances between our operator-child mappings. Providing the magnitude of this reduction is greater than any increase in the overlay cost of placing the child operators with the desired permutation, we are left with an overlay of lower total cost.

3.4 In-Network Placement

Using the techniques described above, we now describe the In-Network algorithm that considers placing operators at arbitrary network locations. In-Network extends on the previous algorithms by considering *select* configurations from a set of candidate locations other than just the sources and the proxy. A greedy, global search strategy would consider configurations from all locations. However this is computationally intractable for large topologies. We here describe a heuristic to effectively prune the configurations considered.

In-Network pursues a similar line to Edge+ in selecting configurations of small total distance. Given that configurations may include arbitrary locations, we reduce the candidate set size with the following heuristic. A location is removed from an operator's candidate set, unless its distance to all current child placements is less than all pairwise distances between child placements. Formally, operator o, with children $\{c_1, \ldots, c_n\}$ has a candidate set:

$$\begin{aligned} \phi(o) = \{v_i \in V : \forall c_i, c_j \in C. \\ d(v_i, \lambda(c_i)) < d(\lambda(c_i), \lambda(c_j)) \wedge \\ d(v_i, \lambda(c_j)) < d(\lambda(c_i), \lambda(c_j))\} \end{aligned}$$

For intuition, the current child placements define a *convex* set of locations, and our selected configurations lie in this convex set. Hence, these configurations have a smaller total separation, in an appropriate part of the topology. However, this may still result in a large number of potential configurations, especially in scenarios where distances between stream sources are relatively large. We further rank configurations by their separations and select a number, k, of these configurations in increasing order of our ranking. Selecting a minimal cost configuration is the same as in Edge+. Once we have chosen our configuration, we place each child at a location corresponding to our configuration, and add the cost of any edges needed to connect the mapping of the operator to its children.

3.5 Latency-Constrained Placement

A desirable property of our mapping function λ would be the ability to place a path-based constraint on the sequence of locations a path in the processing tree is mapped to on the overlay network. A straightforward example is a delay

constraint indicating a desired response time on the processing tree. We now abstract this into our model. Let us consider a path constraint, of value l. For a set of leaf-to-root paths P, we model the constraint as:

$$\sum_{(a,b)\in p} d(\lambda(a), \lambda(b)) \leq l \qquad \forall p \in P$$

The above inequality states that the total distance of a mapped leaf-to-root path must be bounded by the constraint value. We add this to our constraint in Equation 2, in this version of the problem. Our updated definition of λ, to meet this constraint, follows. First we define our set of valid locations, for an operator o with children $\{c_1, \ldots, c_n\}$:

$$L = \quad \phi(c_i) - \{v_i \in \phi(c_i) : \delta(c_i) + d(v_i, \lambda(c_i)) > l\}$$

$$\text{where} \quad \delta(o) = \max_{p \in paths(subtree(o))} \sum_{(a,b)\in p} d(\lambda(a), \lambda(b))$$

We now redefine our mapping function:

$$\lambda(o) = \begin{cases} DHT(o) & \text{if } o \in leaves(T) \\ \arg\min_{v \in L} \gamma(v, o) & \text{otherwise} \end{cases}$$

Here, we ensure that each member of the set L meets the delay constraint. Thus, when we come to mapping the root location, we only consider configurations meeting the constraint. Clearly, this constraint reduces the size of our search space. We assume the processing tree will meet the delay constraint when all operators, except leaves, are placed at the application proxy. If this is not the case, then we have an "infeasible" mapping, given the application proxy is unable to even access the desired sources within the given constraint. This leads to a source placement problem, which may be potentially be solved with replication techniques. This issue lies outside the scope of this paper.

4 Distributed Query Placement

It is evident that placing tree operators at arbitrary locations requires substantial network state to be fed into the mapping algorithm. This requirement significantly restricts the scalability and effectiveness of centralized approaches in the presence of a large number of highly distributed stream sources. In this section, we describe the distributed versions of Edge, Edge+, and In-Network. In the rest of the paper, unless otherwise noted, we will use these names to refer to the distributed versions of the protocols. All the distributed protocols use basic DHT primitives for improved scalability, look-up efficiency, and fault tolerance. As described before, we use the Tapestry as the underlying lookup substrate.

4.1 Overview

In the distributed version of our algorithms, we construct the processing overlay in a bottom-up manner, concurrently determining placements for siblings. We assume that each operator is assigned a globally unique identifier. As described earlier, we subdivide our processing tree into zones, and assign mapping responsibilities for each zone to a coordinator. Coordinators are chosen as the Tapestry peer whose address matches a zone's (or subtree's) identifier. For now, we assume that a subtree's identifier is that of the subtree's root. Thus, in our distributed algorithm we subdivide the optimization search space, and the collection of metadata to drive our search. Once a coordinator has placed its subtree, it communicates with its upstream coordinator (i.e., the coordinator responsible for the subtree rooted at an ancestor operator). We ensure that an operator is aware of its ancestors by appending leaf-to-root (*LR*) paths to specific messages. These paths, along with lists of operators' common locations, are precomputed at the proxy. This distributed mapping process repeats until the root of the processing tree is placed.

Deciding on the number and selection of zones is an open research issue, which should account for factors including the relevant network state, the load on the potential coordinators, and how much parallelism is feasible. Our intuition also indicates that the number of sources plays a significant role in terms of determining divisions of the workload that optimize control overhead. We here investigate one extreme of subtree assignment, a finely-grained scenario where a subtree is a single operator, leaving a more general investigation of this issue to future work.

4.1.1 Local State

Overlay peers maintain two tables, indexed by operator identifiers, to participate in the mapping protocol. The first table, known as the *boundary children* table maintains a list of children for every subtree the peer is responsible for mapping. Our definition of a subtree is one where the subtree does not necessarily extend to the leaves. Boundary children are thus defined as the children connected to subtrees created during workload assignment. The second table, the *operators mapped* table, maintains feasible operator configurations and their associated costs. We now describe how these data structures are used in our protocol. For convenience, Figure 4.1.1 summarizes the messages used in the protocols .

4.2 Edge Placement

Distributed placement is a two-phase process: (1) the application proxy distributes the mapping workload to the coordinators, and (2) the coordinators communicate to perform the mapping, instantiating an overlay.

Coordinator initialization. Workload distribution is performed in two steps. In the first step, we populate the boundary children tables of all coordinators. We traverse the processing tree at the proxy, sending an `ADD_SUBTREE` message to every coordinator. This message consists of a coordinator's assigned subtree, and its boundary children. In the second step, we initiate our decentralized tree mapping at the sources. Here the application proxy sends both an `OPERATOR_MAPPED` and a `MAP` message to each

Message type	Message contents
ADD_SUBTREE	subtree workload, boundary children list
OPERATOR_MAPPED	operator, min-cost configuration, alternate configurations, constraint metadata
MAP	operator, child configuration, common locations, LR path, network view, candidates, constraint metadata
INVALIDATE_MAP	operator
OPERATOR_REMAPPED	operator, configuration

Figure 4: Protocol messages and their contents.

source. The OPERATOR_MAPPED message, used to populate the operators mapped table, includes a configuration of placing the operator at the recipient, as well as a list of alternative configurations. This list is empty for sources, as they reside at fixed locations.

Iterative mapping step. The MAP message contains an operator, its present configuration, and the application proxy's address. The MAP message also contains the LR path used by coordinators for algorithm control. The MAP message triggers our placement mechanism. Mapping a subtree requires all boundary children to have already been placed in the network. Thus, an operator's last mapped boundary child completes the placement of the operator itself. Once a coordinator receives a MAP message, it computes a minimal cost placement for the operators within the subtree assigned to it, using the placement metadata. The MAP messages are sent only for subtree roots based on our workload assignment. We implicitly assume that sources are thus coordinators for leaves, and simply forward the map message based on the LR path they receive.

In the Edge algorithm, computing the least cost mapping of an operator occurs as described in Section 3. In the decentralized scenario, we may potentially have to reconfigure the mapping of the tree below this point, to ensure that our overlay is correctly built. We turn to how we perform this reconfiguration step shortly. Recall that the common location metadata was included in LR paths. Following placement, we compute the parent's cost at all common locations for each ancestor. This is a precomputation step that we utilize when placing the parent's ancestors.

Backtracked placement. We take the following steps should our subtree placement require a reconfiguration of its boundary children (e.g., if it is placed at a common location shared by its children). An OPERATOR_REMAPPED message, containing a configuration, is sent to each reconfigured child's coordinator. The coordinator verifies that the placement is valid by checking for its existence in the possible configurations for the child. The placement is then invalidated at its previous location with an INVALIDATE_MAP message, and sent to the newly mapped location via a OPERATOR_MAPPED message. We check if further reconfiguration is necessary, to enable the desired placement and cost. If so, this remapping process continues down the tree.

Final placement. The final step in subtree mapping simply involves sending OPERATOR_MAPPED messages to the locations of each operator in the newly mapped subtree. We batch our reconfigurations and placement notifications until an entire subtree is mapped by a coordinator, to improve the control efficiency. Following this, the mapping process repeats itself when we send a MAP request from the subtree root's location, to the location of the next ancestor in the LR paths received from the boundary children.

4.3 Edge+

The distributed version of Edge+ follows the same strategy as the distributed Edge, but is augmented to incorporate topology information into the protocol. The protocol employs a similar placement decision to its corresponding centralized version. The protocol is thus responsible for providing the necessary network state for every operator to our placement mechanism.

In Edge, a coordinator mapping a subtree collected an optimal configuration for each of the subtree's boundary children, in addition to the configurations for common locations. In this version of the algorithm, we extend this to include the configurations at each boundary child's descendant locations. We also aggregate each child's localised network view. The network view contains metadata on the network state, such as latencies between particular nodes. Furthermore, we collect metadata as necessary, namely pairwise distances between descendant locations. This network view is initially empty, and transported by our OPERATOR_MAPPED and MAP messages. This configuration cost and topology metadata for each descendant location is used in the Edge+ placement mechanism described in section 3.3.

Note however that this approach is not entirely equivalent to the centralized version. In the centralized version, we were able to compute an operator's cost at any of its siblings' descendant locations. Since we precompute placement costs in the decentralized algorithm to produce a two-phase protocol, we cannot precompute the cost of an operator at its siblings' descendant locations unless we know these locations a priori. This implies both a larger search space, and, more importantly, more number of rounds for our algorithm to operate, given that we would have an extra round of all pairs of siblings exchanging descendant location metadata.

4.4 In-Network Placement

We now describe the distributed version of the In-Network approach, which enables operators to be mapped and executed at arbitrary peers. Earlier, we observed that a coordinator performing a mapping requires optimization metadata containing configurations and distances between potential locations. With this information, the existing placement mechanism is sufficiently general to place an operator at any of the given locations.

The question we address involves selecting *interesting peers*: a set of locations from the topology, that are potentially good candidates upon which to place the operator, in addition to the stream sources locations. This requires performing a "walk" on the network discovering the existence of peers and tracking their distances. Our selection of such "interesting" locations is based on a shortest path tree between the application proxy and the source servers. Specifically, this is a shortest path tree on the Tapestry overlay, and is simply obtained using the routes to each source server. We collect this location information while initially distributing the workload to each source. This set of candidate locations is included in every MAP message, and is used during the placement of each operator.

4.5 Delay-Constrained Placement

We now describe protocol modifications to handle the delay-constrained placement problem. Under our modular approach to distributed optimization, we simply adapt the content of messages in our protocol, to feed additional metadata into our local heuristic to find a solution. To provide this input to our optimization mechanisms, we first modify our local state. We now require that each peer maintains constraint metadata, for each configuration in its operators mapped table. For delay bounds, this is a list of running totals of path delays, of operator-source paths in partially constructed overlays, as well as the constraint itself. In our protocol, this list of path delays is added to the OPERATOR_MAPPED and MAP messages.

In the OPERATOR_MAPPED and MAP messages sent to the sources, we start with a path delay list containing the delay between the source and the proxy. During placement, path delays are aggregated at each operator, and undergo a triangulation transformation. Specifically, we consider each delay list element in turn, and remove the path delay between the corresponding child's placement and the proxy. We then add the delay between the operator's placement and the proxy to every list element. This triangulation is performed whenever an operator is mapped to a different location than any of its children. This approach ensures that we provide correct path delays for partially built overlays of subtrees, whenever we search for valid configurations.

5 Algorithm Analysis

In this section, we briefly analyse our algorithms in order to provide approximate bounds on their (1) bandwidth efficiency (i.e., the ability to reduce the bandwidth necessary to execute a processing tree), (2) computational complexity, and (3) message complexity (in the distributed scenario).

Bandwidth efficiency. Let us consider a processing tree with maximum fanout d, and height $h+1$. This tree has d^h leaves, and let us consider that these leaves are placed at k distinct locations in our network topology, where $0 < k \leq d^h$. We index operators in the processing tree as o_i^j, representing an operator at height j, and tree layer index i. Operators within a tree layer j are numbered left to right from $\{1, \ldots, d^j\}$. In the experimental evaluation that follows, we compare the cost of our overlays to a baseline cost. This baseline cost is viewed as the cost of evaluating the processing tree in a centralized manner, namely at the proxy itself. Our first analytical result is to quantify the probability, P_S, that our mapping algorithms are able to achieve any reduction in bandwidth over this baseline cost.

We now introduce more terminology for this analysis. We define the baseline cost as the bandwidth incurred during transfer of each source to the proxy. These costs are denoted $\{c_1^h, c_2^h, c_3^h, \ldots c_{d^h}^h\}$, and correspond to costs for operators $\{o_1^h, o_2^h, o_3^h, \ldots, o_{d^h}^h\}$. Furthermore, in the algorithms incorporating topology information, we need to distinguish between costs of operators at different locations. This is represented by a prefix subscript, ${}_l o_i^j$, giving the cost of operator o_i^j at location l. The baseline cost is thus, $C_B = \sum_{i=0}^{d_h} c_i^h$. We assume that the costs of sources correspond to the rates of the data streams they represent, and as such are chosen uniformly at random between two size bounds. Our mapping algorithm allows us to state that operators are mapped to a child location only if the cost at that location is less than the cost at the proxy location. Therefore we are interested in the probability that each operator at height $h+1$ has a smaller cost at the proxy location, than at the locations of operators at height h, for the baseline cost to apply. For an arbitrary operator at height h this holds if its cost is larger at the location we are considering mapping to, than the minimum cost of any its children at this specific location, i.e ${}_l c_i^h > \min\{{}_l c_{d^i}^{h+1}, \ldots, {}_l c_{d^{i+1}-1}^{h+1}\}$. Furthermore, this must hold over all locations, implying

$$\begin{aligned} c_i^h &> \max_l \min\{{}_l c_{d^i}^{h+1}, \ldots, {}_l c_{d^{i+1}-1}^{h+1}\} \\ &> \max\{c_{d^i}^{h+1}, \ldots, c_{d^{i+1}-1}^{h+1}\} \quad \text{(fixed leaf locations)} \end{aligned}$$

Above, we note that for leaves, this minimum is simply the cost of the leaf actually at the location (since we assume leaves are at fixed locations) and we drop the location prefix. For the baseline overlay to be output by our algorithm, this must additionally hold across all operators at level $h-1$. Choosing all of these costs from a uniform random distribution, we may state the probability of the above condition as:

$$Pr(c_i^h > \max\{c_{d^i}^{h+1}, \ldots, c_{d^{i+1}-1}^{h+1}\}) = \frac{1}{d+1}$$

This yields a probability that our algorithm improves over the baseline cost of:

$$P_S = 1 - \left(\frac{1}{d+1}\right)^{d^h}$$

Note that the probability that we choose the baseline cost decreases exponentially as our processing tree size increases (in terms of width and depth). For our algorithm that is capable of mapping operators to arbitrary locations, this probability acts as a lower bound, since we may still optimize the cost of our placement even if the costs of operators at level $h-1$ are greater than at level h, but we omit the analysis of this scenario for brevity.

Computational complexity. In the centralized algorithm, a simple greedy algorithm considering placing each operator at each leaf location has complexity $\mathcal{O}(k \times \frac{d^h-1}{d-1})$. Recall that k represents the number of unique sources our leaves are assumed to be placed at. The term $\frac{d^h-1}{d-1}$ represents the number of internal operators in our processing tree whom we must place. Our algorithm performing placements on the sources alone (without topology information) reduces this complexity as follows. When we attempt to map an operator o_i^j, we consider only the minimum cost placement for the children $\{o_{d^i}^{j+1}, \ldots, o_{d^{i+1}-1}^{j+1}\}$, and the cost at any common locations of these children if they exist. Considering minimum cost placements has complexity $\mathcal{O}(d)$ per operator. We may also bound the number of common locations an operator's children may share based on the operator's depth. Additionally, we state that the children of an operator at depth j share at most $\min(\lfloor d^{h-j} \rfloor, \frac{d^h-k}{d-1})$ common locations. The first term in this value for the number of common locations captures the number of descendant leaves, while the second captures the greatest number of common locations that may arise given k unique leaf locations. Combining these two, the computational complexity of our algorithm placing elements at sources only without topology information is:

$$\mathcal{O}((d + \min(\lfloor d^{h-j} \rfloor, \frac{d^h-k}{d-1})) \times \frac{d^h-1}{d-1})$$

Intuitively, from the analysis above, we notice that as the number of unique locations increases, we perform fewer computations of placements for common locations. In the algorithms utilising topology information, placing each internal operator also requires computing placement costs for permutations of descendant locations. For placement at arbitrary locations this is exponential in terms of the topology size, motivating the need for a heuristic in selecting our set of candidate locations. However, we omit the analysis of our heuristic for the sake of brevity.

Message complexity. We finally briefly discuss the message complexity of our distributed tree mapping algorithms. We start with the first phase, namely the workload distribution to the coordinators. We assume a coordinators are responsible for mapping subtrees of size d^{t+1}, implying mapping requires d^{h-t} coordinators. Populating the boundary children tables requires $\mathcal{O}(d^{h-t})$ `ADD_SUBTREE` messages. Subsequently we send `MAP` and `OPERATOR_MAPPED` messages to the sources, of whom, in the worst case, there are $\mathcal{O}(d^h)$. The two stages yield our initialization overhead of $\mathcal{O}(d^h)$ messages. We now consider a map request sent to a coordinator whose subtree depth is $j \in \{1, \ldots, \frac{h+1}{t+1}\}$. First we collect the necessary topology information, requiring $\mathcal{O}((d-1) \times d^{2(h+1-j(t+1))})$ messages. The first component represents the partitions between whom we collect pairwise distances, and the second represents the number of locations in each partition. We also incur messages to notify operators of their placements, specifically $\mathcal{O}(d^{t+1})$ `OPERATOR_MAPPED` messages. Finally we forward the map message, yielding a total of approximately $\mathcal{O}(d^t + (d-1)d^{2(h-jt))})$ messages per map request (relabelling our height and subtree sizes). This sums over d^{h-t} coordinators, of varying subtree depth, yielding a total complexity of mapping all internal operators of:

$$\mathcal{O}(d^h + (d-1)\sum_{j=1}^{h/t} d^{2(h-jt)}) = \mathcal{O}(d^h + d^{h(h-1)/t})$$

Intuitively, this reflects a smaller control overhead as we increase the subtree size. Note that in many cases we may utilise batching to deliver multiple messages to a single source, reducing control overhead. Analytically this is reflected above with bounds $\mathcal{O}(\min(k, d^i))$ replacing $\mathcal{O}(d^i)$ whenever we consider the number of operators at depth i. However a tighter bound analysis lies outside the scope of this paper.

6 Experimental Evaluation

6.1 Experimental Setup

We built an initial prototype system running the algorithms of Section 4 on top of Tapestry, using the OCaml language. In the experiments, we simulated the underlying network: the network topologies used were obtained from the GT-ITM [25] topology generator. We generated ten independent transit-stub topologies (14 transit domains, with 500ms of longest pair-wise path delay between stub nodes).

We generated our workload of processing trees with specific characteristics: unless stated otherwise, we considered binary trees with depths ranging from three to five. Operator selectivities were also selected uniformly at random from [0,1]. Unless otherwise specified, all results shown are averaged over ten independent runs, each mapping 100 processing trees.

In the experiments, we compare the distributed versions of Edge, Edge+, and In-Network, to a naive approach, called *baseline*. Baseline simulates an on-line warehousing model where all the streams are forwarded to the proxy (using shortest paths) for processing.

We control the placement of the data sources, to better understand the consequences of varying placements, using two metrics: *average proxy distance (APD)* and *average server distance (ASD)*. Average proxy distance represents the average distance between the application proxy and each source. Average server distance represents the average distance between a source and every other source, for all sources. Using these two metrics, we define three interesting configurations: *uniform*, *star*, and *cluster*. In uniform, as the name implies, all servers and proxies are uniformly spread (APD and ASD values to be equal to 0.4 of the length of the network diameter). In the star placement, the proxy lies near the "centroid" of the sources. In the experiments, we fix the APD to be approximately half of the ASD. In the cluster topology, the sources each has a considerably larger distance to the proxy, than between themselves. We achieve this configuration with an APD approximately twice that of the ASD. Figure 5 illustrates these source-proxy location configurations.

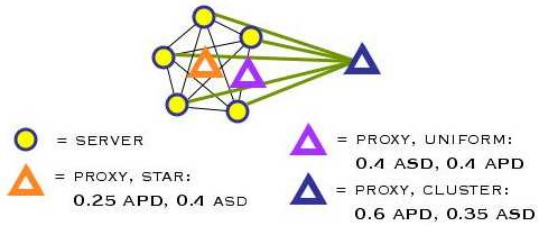

Figure 5: Source-proxy location scenarios: *star, uniform, cluster.*

As the key efficiency metric, we use *bandwidth consumption ratio*, which quantifies the ratio of the overall bandwidth consumed by a given approach to that of the baseline. To quantify the effectiveness of the system, we present results for overlay *stretch*, which is the ratio of the longest path length on the constructed overlay to the longest path length from the sources to the proxy (path lengths are specified in terms of delay). For simplicity, we ignore operator processing costs as well as operator queueing delays. As a result, stretch is an estimate of the extra latency that the system incurs (when producing result tuples) when yielding the bandwidth savings estimated by the bandwidth consumption ratio. We also investigate our algorithms' behaviours by tracking the percentages of operators placed at the proxy, at sources and inside the network. These two metrics are commonly used when evaluating large-scale networked systems.

6.2 Basic Algorithm Comparison

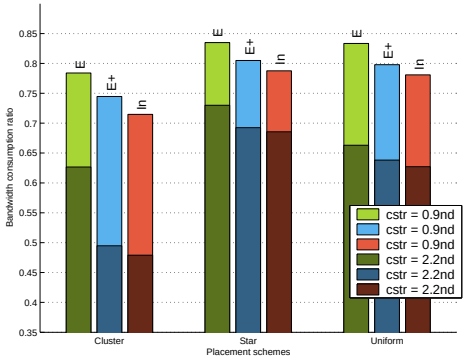

Figure 6: Bandwidth consumption ratio for various source and proxy locations.

Our first set of experiments compare the three placement algorithms described in Section 3. Figure 6 shows the bandwidth consumption ratio achieved by each mapping algorithm, for the three network configurations described above, for tight and loose delay constraints (120 and 300 ms, defined as a ratio of the average network diameter, nd). We see that the algorithms perform similarly on both the star and uniform placement schemes. In turn, the cluster scenario seems to offer greater scope for optimization. Furthermore, relaxing the delay constraint has little effect on the bandwidth consumption ratio. Both Edge+ and In-network consistently offer advantages in the bandwidth consumption ratio, over Edge, across all placement schemes. Utilizing topology information is clearly beneficial, especially in the loosely constrained cluster scenario. It is difficult to differentiate Edge+ and In-network for the star and uniform scenarios. This result arises because the proxy acts as the ideal intermediate location where the streams can be pushed and executed.

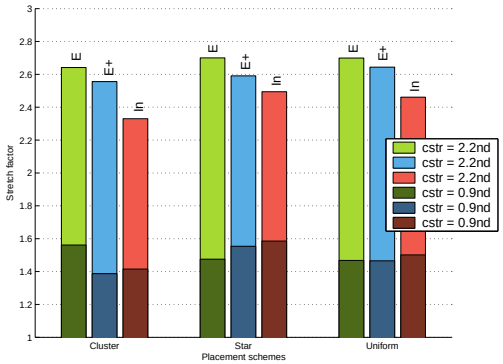

Figure 7: Latency stretch for various source and proxy locations.

In Figure 7 we investigate the effects of the three algorithms on the stretch factor. The Edge algorithm exhibits a worse stretch factor under all loosely constrained scenarios. However, Edge is comparable to Edge+ and In-network under the tightly constrained scenarios. This is a direct effect of the tightness of the constraint requiring placements similar to the baseline mapping. In the loosely constrained placement, In-network consistently outperforms Edge+. Here, In-network yields a lower stretch factor, due to the placement of operators between the cluster of sources and the proxy. Operator placement generally occurs in the "direction" of the proxy, creating a mapping of tree paths tending towards shortest network paths. Meanwhile, in Edge+, there is no such consideration of direction, implying streams may be temporarily pushed away from the direction of the proxy, lengthening the end-to-end delay. In the tightly constrained scenario, In-network tends to perform worse than Edge+, because it achieves a better optimization for bandwidth consumption. Here Edge+ is less capable of performing optimization, and has a greater tendency to simply push streams directly to the proxy, yielding a lower end-to-end delay.

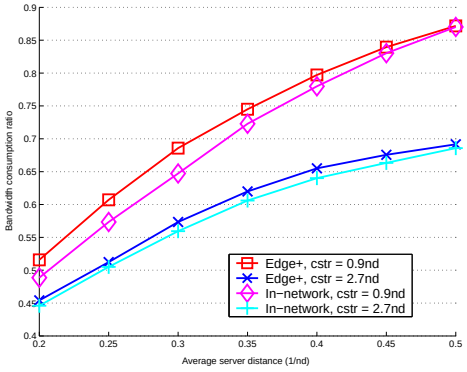

Figure 8: Source distribution effects on bandwidth.

Figure 8 shows the effects of varying the ASD (as a ratio of the network diameter) upon the bandwidth consumption

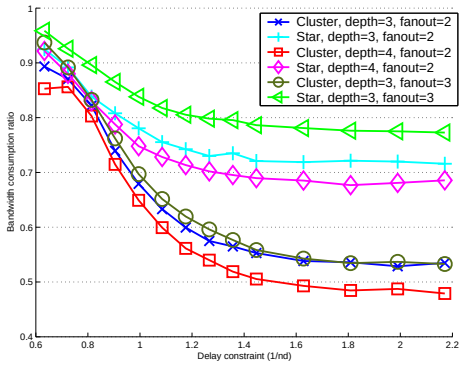

Figure 9: Impact of processing tree structure on bandwidth.

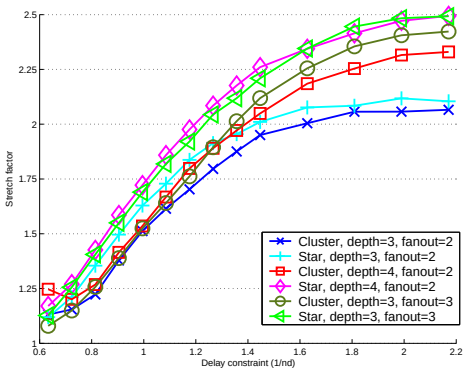

Figure 10: Impact of processing tree structure on latency.

ratio, for the cluster configuration. We see that, in general, as sources are placed further and further apart, the bandwidth consumption ratio increases. Since our cost function representing bandwidth is proportional to distance, this result is to be expected. We also observe that for tight constraints, the rate of this increase is far greater. Under tight constraints, we have little possibility of improving the cost, and so as sources are separated, we tend to the baseline approach. Under loose constraints, we may actually still perform optimizations at larger source separations, resulting in a shallower gradient for the increase in the bandwidth consumption ratio.

6.3 Varying the Processing Tree Structure

We now study the effects of the processing tree structure upon the bandwidth consumption ratio and stretch factor, when mapped by the In-network algorithm (we omit the results for Edge and Edge+ to simplifiy the presentation, as In-network dominates these algorithms in terms of bandwidth efficiency).

Figure 9 shows the bandwidth consumption ratio as a function of the delay constraint, for three forms of processing trees. First, as the delay constraint gets looser, the bandwidth consumption initially decreases, prior to tailing off. This occurs because the constraint becomes less and less restrictive on our feasible configurations, and stops interfering with the optimization. We witness that deeper trees result in lower bandwidth consumption, while wider trees result in a larger bandwidth consumption ratio. Deeper trees offer scope for optimization since there are a larger number of internal operators whom we may place into more elaborate configurations. Operators with higher degrees (i.e., fanout) prove more problematic to place, we directly affect data flows to a larger number of operators. Hence there is less scope for optimization, when the sources reside at a large number of unique locations.

Figure 10 plots the stretch factor as a function of delay constraint, for varying processing tree characteristics. Stretch factor behaves as a dual to the bandwidth consumption ratio. As the delay constraint is loosened, the stretch factor rises, but tails off as the constraint has less and less effect. Both deeper and wider trees result in a larger stretch factor, and a later tail-off point. In both cases the larger stretch factor is a direct effect of the increased number of operators that must be mapped.

6.4 Selectivity Impact

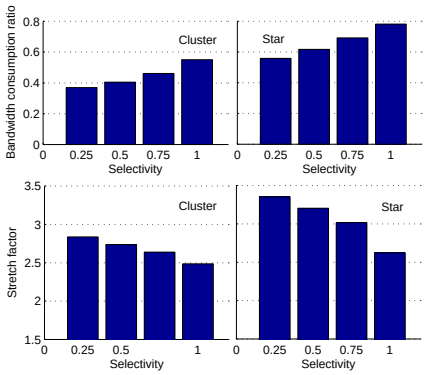

Figure 11: Impact of operator selectivities on bandwidth and latency.

We now study the impact of operator selectivities on the bandwidth consumption ratio and stretch factors. Figure 11 shows these effects for four selectivity values. Note that these selectivities are an upper bound on the actual operator selectivities, which are chosen uniformly at random from 0 to the bound. We see that as the selectivity approaches a value of 1, the bandwidth consumption ratio for both the cluster and star placement schemes increase. This is because selectivity is a direct indicator for the scope of potentially optimizing a tree; operators with selectivities close to 1 incur similar bandwidth consumption regardless of where they are placed. We also see that star placements generally have higher bandwidth consumption ratios, consistent with the results shown earlier. In terms of stretch factors, we observe that increasing selectivity results in a decreasing stretch factor. Operators with unit selectivities cannot be placed using as elaborate a configuration as operators high low selectivities. Hence the paths in our tree mappings tend to more strongly resemble the baseline mapping, since such operators are generally tightly grouped.

To provide a deeper intuition for these results, we show in Figure 12 percentages of operators that are placed at the proxy, or inside the network, for varying selectivities. Operators that are placed at neither the proxy nor inside the network are obviously placed at the sources (not ex-

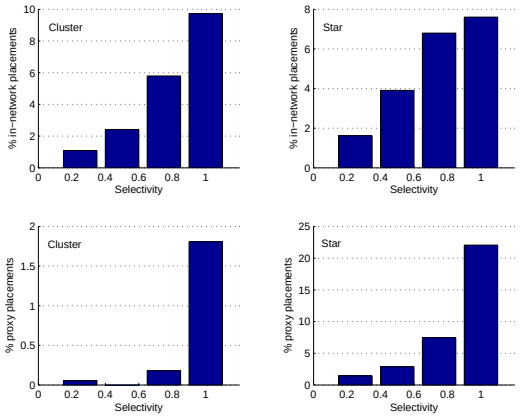

Figure 12: Operator location distributions.

plicitly shown). The algorithm generally tends to utilize in-network placements when sources are clustered moreso than when sources lie in the star formation. Furthermore, in a star placement scheme, operators are generally more likely to be placed at the proxy than at sources, when compared to a cluster placement. Examining these values per selectivity value, we see that less selective operators result in larger numbers of operators placed inside the network. This trend is also true for the operators that are placed at the proxy. As expected, highly selective operators are being pushed all the way to the sources.

7 Related Work

The work presented in this paper is related to several topics in distributed and parallel databases and systems, and networking.

Distributed query processing. Distributed query optimization and in particular the site selection problem are closely related to our work and have been explored extensively in the context of distributed and federated databases [9, 10, 15, 16, 23]. To the best of our knowledge, none of these approaches address widely-distributed processing and network awareness, and are thus well represented by the Edge approach.

More recent work addressed Internet-scale query processing and distribution scalability. IrisNet [8] focuses on querying wide-area sensor databases using XPath queries. IrisNet relies on the DNS to identify the remote databases relevant to a given query, which is then processed using XML and XPath specific optimizations. Similar to our work, PIER [12] addresses DHT-based highly-distributed query processing, although in a pull-based setting. PIER discusses how CAN [19] can be used as a hashing function on the indexes of relations, distributing tuples across a very large number of sites. While PIER and our work share many common goals, there are also some significant differences. Our algorithms attempt a finer-grained control of the placement decisions, whereas in PIER, the operations themselves are randomly distributed across peers by CAN. The semantic details of our operators are abstracted away from the placement mechanism. Instead we focus on optimizing the network positioning of operators, and as such, operator specific optimizations such as those presented by PIER may still apply.

In-network query processing has been studied in the context of sensor databases. Recent work by Madden *et al.* [17] demonstrated the advantages of in-network data aggregation in a wireless multi-hop sensor network. In such a resource-constrained environment, potential optimizations are severely restricted and network scalability is typically not a key design goal.

Stream processing and continuous queries. Recently, there has been much work on data-stream processing (e.g., [1, 4, 5, 6, 18, 21]). Most of these efforts have commonly assumed an on-line warehousing model where all source streams are routed to a central site where they are processed. There have also been some preliminary proposals that extend the single-site model to multi-site, distributed models and environments [2, 7, 21]. Our work is also a step in this general direction.

NiagaraCQ [6] is a continuous query processing system designed for Internet-scale query processing. Babcock and Olston [2] investigated the use of adaptive filters that are executed at the stream sources based on per-query precision requirements registered at stream sources. Neither of these work investigated network-aware operator placement issues that we discuss here.

Overlay networks. Overlay networks [14, 22, 26] strive to address scalability and fault tolerance issues that arise in large-scale content distribution, using the same principles of in-network processing during message routing. Intermediate routers are envisaged as having the capability to perform certain functionality on the messages they forward. Our work can be regarded as addressing how to distribute "active networking" functionality across the overlay network servers, an issue that has not yet been addressed.

Distributed task partitioning. The parallel computing community has long studied the distribution and allocation of tasks in homogeneous distributed environments. Grid computing researchers have also developed architectures tailored to large-scale scientific applications, that perform resource allocation to servers much in the same way we map operators to locations. One common approach to allocating tasks is a multilevel graph partitioning algorithm [3, 11]. This approach partitions the computational structure of an application in an attempt to minimize the resulting data flow across partitions. The models proposed do not leverage application specific characteristics, such as selectivity, rather they are purely concerned with application structure. Furthermore, these approaches have not yet addressed wide distribution and network scalability, our main concerns in this study.

8 Conclusions and Future Work

With the proliferation of applications that involve sophisticated processing of large numbers of distributed data sources, there is a growing need for generic widely-distributed query processing services. Such systems will be fundamentally network-oriented. We believe that net-

work scalability, which has been largely ignored by previous efforts, will be a key design goal for next-generation data processing systems. This paper addresses one of the key challenges towars achieving this goal.

We argued that widely-distributed query processing can greatly benefit from *network awareness* in terms of improved bandwidth efficiency and result latency. Previous approaches largely ignored the impact of the characteristics of the interconnecting network and the relative locations of the servers on processing. We presented network-aware operator placement algorithms and described in detail their distributed implementation, on top of a DHT infrastructure, for distributed stream processing. We analyzed the algorithms and experimentally evaluated them using a prototype implementation and realistic network models. Comparison with representative network-unaware approaches verified the benefits that can be attained through the use of network information and in-network processing.

There are several important directions for future research. One immediate direction involves exploiting opportunities for sharing among multiple queries during the mapping process. Another direction involves exploiting semantic, operator-specific optimization opportunities. Finally, we would like to integrate our prototype with Borealis (follow-on to Aurora [1]) and verify the validity of the results presented here with a real application and deployment.

This work has been done in the context of the SAND project. SAND strives to extend core data management and processing functionality to highly-distributed environments and applications. This work is an initial step in this general direction.

Acknowledgements

We are grateful to John Jannotti, Eli Upfal, Stan Zdonik, and the anonymous reviewers for valuable discussions and feedback.

References

[1] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal: The International Journal on Very Large Data Bases*, 2003.

[2] B. Babcock and C. Olston. Distributed top-k monitoring. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[3] R. Biswas, B. Hendrickson, and G. Karypis. Graph partitioning and parallel computing. *Parallel Computing*, 26(12):1515–1517, Nov. 2000.

[4] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of the 28th International Conference on Very Large Data Bases (VLDB'02)*, Aug. 2002.

[5] S. Chandrasekaran, A. Deshpande, M. Franklin, and J. Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for Internet databases. In *Proc. of the 2000 ACM SIGMOD International Conference on Management of Data*, May 2000.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[8] A. Deshpande, S. Nath, P. B. Gibbons, and S. Seshan. Cache-and-query for wide area sensor databases. In *Proc. of the 2003 ACM SIGMOD International Conference on Management of Data*, June 2003.

[9] M. J. Franklin, B. T. Jónsson, and D. Kossmann. Performance tradeoffs for client-server query processing. *SIGMOD Record (ACM Special Interest Group on Management of Data)*, 25(2):149–160, June 1996.

[10] L. M. Haas, D. Kossmann, E. L. Wimmers, and J. Yang. Optimizing queries across diverse data sources. In *Proceedings of the 23rd International Conference on Very Large Databases*, Aug. 1997.

[11] B. Hendrickson and R. Leland. A multilevel algorithm for partitioning graphs. In *Supercomputing'95*, 1995.

[12] R. Huebsch, J. M. Hellerstein, N. L. Boon, T. Loo, S. Shenker, and I. Stoica. Querying the Internet with PIER. In *Proc. of the 29th International Conference on Very Large Data Bases (VLDB'03)*, Sept. 2003.

[13] Y. E. Ioannidis and Y. Kang. Randomized algorithms for optimizing large join queries. In *Proc. of the 1990 ACM SIGMOD international conference on Management of data*, June 1990.

[14] J. Jannotti, D. K. Gifford, K. L. Johnson, M. F. Kaashoek, and J. W. O'Toole, Jr. Overcast: Reliable multicasting with an overlay network. In *Proceedings of the 2000 Fourth Symposium on Operating Systems Design and Implementation (OSDI 2000)*, San Diego, California, Oct. 2000.

[15] D. Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys*, 32(4):422–469, 2000.

[16] L. F. Mackert and G. M. Lohman. R* optimizer validation and performance evaluation for local queries. In *Proc. of the 1986 ACM SIGMOD International Conference on Management of Data*, pages 84–95. ACM Press, 1986.

[17] S. Madden, M. J. Franklin, J. Hellerstein, and W. Hong. TAG: A tiny aggregation service for ad-hoc sensor networks. In *Proceedings of the 5th USENIX Symposium on Operating Systems Design and Implementation (OSDI '02)*, Boston, Massachusetts, Dec. 2002.

[18] R. Motwani, J. Widom, A. Arasu, B. Babcock, S. Babu, M. Datar, G. Manku, C. Olston, J. Rosenstein, and R. Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, Jan. 2003.

[19] S. Ratnasamy, P. Francis, M. Handley, R. Karp, and S. Schenker. A scalable content-addressable network. In *Proceedings of the ACM SIGCOMM '01 Conference*. ACM Press, 2001.

[20] A. I. T. Rowstron and P. Druschel. Pastry: Scalable, decentralized object location, and routing for large-scale peer-to-peer systems. In *Proceedings of the IFIP/ACM International Conference on Distributed Systems Platforms Heidelberg*. Springer-Verlag, 2001.

[21] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of the 19th International Conference on Data Engineering (ICDE 2003)*, Mar. 2003.

[22] I. Stoica, R. Morris, D. Karger, M. F. Kaashoek, and H. Balakrishnan. Chord: A scalable peer-to-peer lookup service for internet applications. In *Proceedings of the ACM SIGCOMM '01 Conference*. ACM Press, 2001.

[23] A. Tomasic, L. Raschid, and P. Valduriez. Scaling heterogeneous databases and the design of disco. In *Proceedings of the 16th International Conference on Distributed Computing Systems (ICDCS '96)*. IEEE Computer Society, 1996.

[24] S. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *Proc. of the 2002 ACM SIGMOD International Conference on Management of Data*, June 2002.

[25] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *IEEE Infocom*, volume 2, pages 594–602, San Francisco, CA, March 1996.

[26] B. Y. Zhao, L. Huang, S. C. Rhea, J. Stribling, A. D. Joseph, and J. D. Kubiatowicz. Tapestry: A global-scale overlay for rapid service deployment. *IEEE J-SAC*, 22(1):41–53, January 2004.