A Distributed Stream Query Optimization Framework through Integrated Planning and Deployment

Sangeetha Seshadri, *Student Member, IEEE*, Vibhore Kumar, *Member, IEEE*,
Brian Cooper, and Ling Liu, *Senior Member, IEEE*

Abstract—This paper addresses the problem of optimizing multiple distributed stream queries that are executing simultaneously in distributed data stream systems. We argue that the static query optimization approach of “plan, then deployment” is inadequate for handling distributed queries involving multiple streams and node dynamics faced in distributed data stream systems and applications. Thus, the selection of an optimal execution plan in such dynamic and networked computing systems must consider operator ordering, reuse, network placement, and search space reduction. We propose to use hierarchical network partitions to exploit various opportunities for operator-level reuse while utilizing network characteristics to maintain a manageable search space during query planning and deployment. We develop top-down, bottom-up, and hybrid algorithms for exploiting operator-level reuse through hierarchical network partitions. Formal analysis is presented to establish the bounds on the search space and suboptimality of our algorithms. We have implemented our algorithms in the IFLOW [1] system, an adaptive distributed stream management system. Through simulations and experiments using a prototype deployed on Emulab [2], we demonstrate the effectiveness of our framework and our algorithms.

Index Terms—Computer-communication networks, distributed systems, distributed databases, distributed applications, database management, systems, query processing.

1 Introduction

Many data stream delivery and dissemination systems today produce stream data at multiple, geographically distributed locations. It is often too expensive to stream all of the data to a centralized query processor, both because of the high communication costs, and the high and yet continuously changing processing load at the central server. Therefore, in order to ensure efficiency and scalability, these naturally distributed applications adopt a distributed processing paradigm.

Distributed data streams systems are distinguished by a number of characteristics. First, a network of computing nodes with heterogeneous bandwidth and computing resources together serves as a distributed data stream delivery system. Second, data streams originate from multiple sources and are disseminated to multiple receivers. Third, multiple continuous stream queries are executing simultaneously on the stream delivery network with different input and output rates. Instead of shipping all data streams to a single node and processing all the stream queries in a centralized server, many have shown that performing distributed processing of stream queries using techniques such as in-network processing [3], [4], [5] and filtering at the source [6] minimizes the communication overhead on the system and helps spread processing load, significantly improving performance.

Given that data streams are typically produced from multiple disparate nodes, stream queries naturally consist of many operators (filters, joins, etc.) on multiple data streams of interest. We can think of a data stream query as a continual query being “deployed” in the network, with data streams flowing between operators associated with distributed physical streaming nodes, which may either be sensor nodes or the relay nodes in a data stream delivery network. The conventional approach to stream query processing used in many existing distributed data stream management systems [7], [8] consists of three consecutive phases: query planning, query deployment, and query adaptation. Concretely, the system constructs a query plan (e.g., the stream query processing should follow a specified join ordering) at compile time and deploys this plan at runtime to improve performance. Fig. 1a gives a sketch of this approach. A fundamental problem with this static optimization approach is its inability to respond to the unexpected data and resource changes occurring at runtime. For example, the join order chosen at compile time may require intermediate results to be transported to another network node over a long distance, even though

- *S. Seshadri is with the College of Computing, Georgia Institute of Technology, 801 Atlantic Drive, KACB 3201, Atlanta, GA 30332. E-mail: sangeeta@cc.gatech.edu.*
- *V. Kumar is with the IBM T.J. Watson Research Center, 19 Skyline Drive, Hawthorne, NY 10532. E-mail: vibhorek@us.ibm.com.*
- *B. Cooper is with Yahoo! Research, 2821 Mission College Blvd., Santa Clara, CA 95054. E-mail: cooperb@yahoo-inc.com.*
- *L. Liu is with the College of Computing, Georgia Institute of Technology, KACB, Room 3340, 266 Ferst Dr., Atlanta, GA 30332-0765. E-mail: lingliu@cc.gatech.edu.*

Manuscript received 12 July 2008; accepted 3 Oct. 2008; published online 10 Oct. 2008.
Recommended for acceptance by G. Agrawal.
For information on obtaining reprints of this article, please send e-mail to: tpds@computer.org, and reference IEEECS Log Number TPDS-2008-07-0262.
Digital Object Identifier no. 10.1109/TPDS.2008.232.

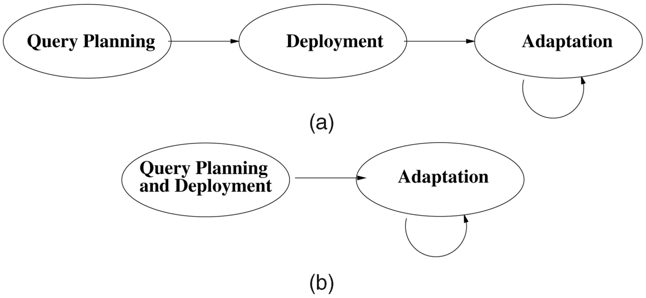

Fig. 1. Approaches. (a) Plan, then deploy. (b) Our approach.

there exists an alternate join order that is more efficient. Similarly, a predefined join order may involve a transfer or a processing of an intermediate result to a node that is currently unavailable, thus causing the query to halt even though an alternate join order exists and is available. Furthermore, given that each query plan is computed at compile time independently and once for all, the predefined join order from one query plan may prevent us from reusing the results of an already deployed join from another query at runtime. This limits the scope of the adaptation which aims at exploiting runtime environment properties to further optimize the efficiency of distributed stream query deliveries.

Bearing these issues in mind, we propose a distributed stream query optimization framework that considers the query plan and the deployment simultaneously (Fig. 1b). Our framework consists of the system architecture for integrating distributed stream query planning and query plan deployment and a suite of techniques for performing query planning in conjunction with deployment planning. One of the key ideas in our framework is to use hierarchical network partitions to scalably exploit various opportunities for operator-level reuse in the processing of multiple stream queries. Fig. 2 compares the approach of integrating planning and deployment through operator reuse with two existing "Plan, then deploy" approaches—the Relaxation algorithm [9] and an optimal deployment through exhaustive search. The graph shows the total communication cost (the total data transferred along each link times the link cost) incurred by 100 queries over five stream sources each, on a 64-node network. The figure shows that significant (> 50 percent) cost savings can be achieved by combining the planning and deployment phases.

It is well known that, as the size of the network grows, the number of possible plan and deployment combinations can grow exponentially. The cost of considering all possibilities exhaustively is prohibitive. Consider Fig. 2. With a network of 64 nodes, combining query plans and plan deployments simultaneously required us to examine nearly 3.02×10^9 plans for a single query over five streams. Clearly, a key technical challenge for effectively combining query planning and plan deployment is to reduce the search space in the presence of large networks and a large number of query operators.

One idea we explore in this paper is to address this challenge by using hierarchical network partitions as a heuristic, aiming at trading some optimality for a much smaller search space. Concretely, we organize the network of physical nodes into a virtual hierarchy and utilize this hierarchy along with "**stream advertisements**" to guide query planning and deployment. We develop three alternative algorithms to facilitate operator reuse through hierarchical network partitions. In the **Top-Down** algorithm, the query starts at the top of the hierarchy, and is recursively planned by progressively partitioning the query and assigning subqueries to progressively smaller portions of the network. In the **Bottom-Up** algorithm, the query starts at the bottom of the hierarchy, and is propagated up the hierarchy, such that portions of the query are progressively planned and deployed. While both algorithms choose efficient deployments by exploring only a small fraction of the search space, the Top-Down algorithm is more effective in limiting the suboptimality of the solutions while the Bottom-Up approach is more effective in reducing the search space, and thereby the time to deployment. We further develop a heuristic-based hybrid algorithm that combines the strengths of both the Top-Down and Bottom-Up algorithms—the **Net Present Cost (NPC)** algorithm. The NPC algorithm is a probabilistic algorithm that guides the planning process based on cost estimates of choosing a join order locally or delaying the decision to the next level. We have implemented our algorithms using IFLOW [1], a distributed data stream system. In this paper, we also present formal analysis and experiments to show that our algorithms can compute efficient deployments, and at the same time, reduce the search space by orders of magnitude compared to an exhaustive search, even using dynamic programming. For example, experimentally, the Top-Down algorithm on average was able to achieve solutions that were suboptimal by only 10 percent while considering less than 1 percent of the search space.

The remainder of this paper is organized as follows: We formally describe the distributed stream query optimization problem and give an overview of our distributed optimization framework in Section 2. We present the Top-Down, Bottom-Up, and NPC algorithms and a rigorous analysis of their effectiveness in Section 3. Our experimental evaluation of the proposed solutions is reported in Section 4. This paper ends with a discussion on the related work and a summary.

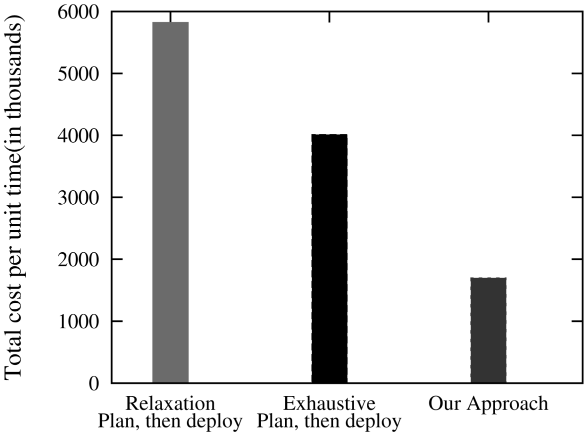

Fig. 2. Comparison with typical approaches.

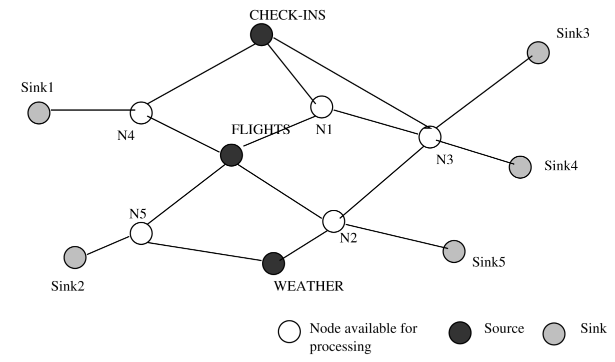

Fig. 3. An example network N.

2 System Overview

Many modern enterprise applications [10], [11], [12], scientific collaborations across wide-area networks [13], [14], and large-scale distributed sensor systems [15], [16] are placing growing demands on distributed streaming systems to provide capabilities beyond basic data transport such as wide-area data storage [17] and continuous and opportunistic processing [12]. An increasing number of streaming applications are applying "in-network" and "in-flight" data manipulation to data streaming systems designed for enterprise systems [18], financial management [19], scientific computing [20], [13], [21], and situation monitoring applications [20], [22], [23].

The specific motivating example that we present in this paper is based on enterprise-level data streaming systems such as the Operational Information System (OIS) [10] employed by our collaborators, Delta Airlines. An OIS is a large-scale distributed system that provides continuous support for a company or organization's daily operations. The OIS run by Delta Airlines provides the company with up-to-date information about all of their flight operations, including crews, passengers, weather, and baggage. Delta's OIS combines three different types of functionality: continuous data capture, for information like crew dispositions, passengers, and flight locations; continuous status updates, for systems ranging from low-end devices like overhead displays to PCs used by gate agents and even large enterprise databases; and responses to client requests which arrive in the form of queries.

In such a system, multiple continuous queries may be executing simultaneously and hundreds of nodes, distributed across multiple geographic locations are available for processing. In order to answer these queries, data streams from multiple sources need to be joined based on the flight or time attribute, perhaps using something like a symmetric hash join. We next use small example network and sample queries to illustrate the optimization opportunities that may be available in such a setup.

2.1 Motivating Application Scenario

Let us assume Delta's OIS to be operating over the small network N shown in Fig. 3. Let WEATHER, FLIGHTS, and CHECK-INS represent sources of data-streams of the same name and nodes $N1-N5$ be available for in-network processing. Each line in the diagram represents a physical network link.Also assume that we can estimate the expected data rates of the stream sources and the selectivities of their various attributes, perhaps gathered from historical observations of the stream-data or measured by special-purpose nodes deployed specifically to gather data statistics.

Assume that the following query Q1 is to be streamed to a terminal overhead display $Sink4$. Q1 displays flight, weather, and check-in information for flights departing in the next 12 hours.

```
Q1: SELECT FL.STATUS, WR.FORECAST, CI.STATUS
  FROM FLIGHTS FL, WEATHER WR, CHECK-INS CI
  WHERE FL.DEPARTING='ATLANTA'
  AND FL.DESTN = WR.CITY AND FL.NUM = CI.FLNUM
  AND FL.DP-TIME - CURRENT_TIME < 12:00:00
```

Network-aware join ordering. Based purely on the size of intermediate results, we may normally choose the join order (FLIGHTS $\bowtie$ WEATHER)$\bowtie$ CHECK-INS. Then, we would deploy the join FLIGHTS $\bowtie$ WEATHER at node $N2$, and the join with stream CHECK-INS at node $N3$. However, node $N2$ may be overloaded, or the link FLIGHTS → $N2$ may be congested. In this case, the network conditions dictate that a more efficient join ordering is (FLIGHTS $\bowtie$ CHECK-INS) $\bowtie$ WEATHER, with FLIGHTS $\bowtie$ CHECK-INS deployed at $N1$, and the join with WEATHER at $N3$.

Now, consider situations where we may be able to reuse an already deployed operator. This will reduce network usage (since the base data only needs to be streamed once) and processing (since the join only needs to be computed once). Imagine that query Q2 has already been deployed:

```
Q2: SELECT FL.STATUS, CI.STATUS FROM FLIGHTS FL,
  CHECK-INS CI
  WHERE FL.DEPARTING='ATLANTA' AND FL.NUM =
  CI.FLNUM AND FL.DP-TIME - CURRENT_TIME <
  12:00:00
```

with the join FLIGHTS $\bowtie$ CHECK-INS deployed at $N1$. Assume that the sink for the query Q2 is located at node $Sink3$.

Operator reuse. Although the optimal operator ordering in terms of the size of intermediate results for query Q1 may be (FLIGHTS $\bowtie$ WEATHER) $\bowtie$ CHECK-INS, in order to reuse the already deployed operator FLIGHTS $\bowtie$ CHECK-INS, we must pick the alternate join ordering (FLIGHTS $\bowtie$ CHECK-INS) $\bowtie$ WEATHER. Note that reuse may require additional columns to be projected. In contrast, if the sinks for the two queries are far apart (say, at opposite ends of the network), we may decide not to reuse Q2's join; instead, we would duplicate the FLIGHTS $\bowtie$ CHECK-INS operator at different network nodes, or use a different join-ordering. Thus, having knowledge of already deployed queries influences our query planning.

These examples show that the network conditions and already deployed operators must often be considered when choosing a query plan and deployment in order to achieve the highest performance.

2.2 System Definition

We now formally describe the components of our distributed data stream system. Let $N(V_n, E_n)$ represent a physical network of nodes, where vertices V_n represent the set of actual physical nodes and the network connections between the nodes are represented by the set of edges E_n. Let Q represent a single continuous query and let $P^Q = \{p_1^Q, \ldots, p_m^Q\}$ represent the set of all relational algebra query trees (e.g., operator orderings) for query Q. The *deployment* of a query tree p_j^Q over the network N is defined as a mapping $M(p_j^Q, N)$ that assigns each operator in p_j^Q to a network node $v_{nk} \in V_n$.

Since network costs are a primary concern in wide-area stream processing systems, to illustrate our techniques, we choose a formulation that tries to minimize the communication cost incurred per unit time by the deployed query plan. We use the metric of "network usage" [9] to compute costs of query deployments. The network usage metric computes the total amount of data in-transit in the network at a given instant. This metric captures the bandwidth-delay product of a query and trades off the overall application delay and network bandwidth consumption. We define a cost function $Cost(M(p_i^Q, N))$ that estimates the total network usage per unit time for the deployment $M(p_i^Q, N)$. Using network usage as the cost function, for a deployment $M(p_i^Q, N)$ the cost is given by $\sum_{l \in M(p_i^Q, N)} \lambda(l) \times latency(l)$, where $\lambda(l)$ represents the data rate over the physical link l. We present a further discussion on the choice of the cost metric in Section 5.

2.3 Optimization Problem

Our problem definition addresses the continual query equivalent of "select-project-join" queries that involve simple selection, projection, and join operations on one or more data streams. The focus of this paper is on join ordering and the initial placement of operators. Note that it may be possible to modify existing deployments to get a better solution. However, such modifications require us to consider the cost of reconfigurations and deal with translation of state as well. We leave such possibilities for the future. We assume stream joins are performed using standard techniques (e.g., doubly pipelined operators and windows if necessary). We assume that potentially, *any* operator can be deployed at *any* node in the system. Given a query, there could possibly be multiple execution plans that the system could follow to produce results. We assume that all such plans produce equivalent results.

Query-optimization problem. Given a query Q to be deployed over a network N, and a (possibly empty) set of existing query deployments $D = \{D_1, \ldots, D_n\}$, find a query tree $\{p_i^Q\}$ and a deployment $M(p_i^Q, N)$ for Q such that $Cost(M(p_i^Q, N))$ is minimum over all possible query trees and deployments.

3 Query Optimization Algorithms

In order to choose an optimal execution plan, traditional query optimizers typically perform an exhaustive search of the solution space using dynamic programming, estimating the cost of each plan using precomputed statistics. Lemma 1 shows the size of the exhaustive search space for the query optimization problem in distributed data stream systems.

Lemma 1. *Let $\mathcal{Q}$ be a query over K (> 1) sources to be deployed on a network with N nodes. Then, the size of the solution space of an exhaustive search is given by*

$$\mathcal{O}_{\text{exhaustive}} = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (N)^{(K-1)}.$$

Proof. We are given a network with N nodes, and a query $\mathcal{Q}$ over K streams $\langle \mathtt{S}_1, \mathtt{S}_2, \ldots \mathtt{S}_K \rangle$. The search space is given by all plans (permutations of join-orders) and all possible placements of each plan. The number of query rewritings, i.e., an enumeration of both linear and bushy joins of K streams is given by

$$\binom{K}{2} \times \binom{K-1}{2} \times \ldots \times \binom{2}{2} = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right).$$

The number of network placements of the joins in a query with K streams in a network of size N is given by $N^{(K-1)}$. Thus, the exhaustive search space $\mathcal{O}_{exhaustive}$ given by

$$\mathcal{O}_{\text{exhaustive}} = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (N)^{(K-1)}.$$

□

As shown in Lemma 1, the search space increases exponentially with an increase in the query size. Certainly, in a system with thousands of nodes, such an exhaustive search even with dynamic programming would be infeasible. We now present our optimization infrastructure and heuristics for finding good plans and deployments while avoiding the cost of exhaustive search. Note that in the case of distributed query optimization, dynamic programming does not result in any pruning of the search space without loss of optimality since the query optimization problem in distributed data stream systems does not exhibit the property of optimal substructure [24].

3.1 Optimization Infrastructure

In this section, we describe the key components of our optimization infrastructure—*hierarchical network partitions* that guide our planning heuristics and *stream advertisements* that facilitate operator reuse. We can tune the hierarchy to trade-off between search space size and suboptimality by adjusting the max_{cs} parameter, which is the maximum number of nodes allowed per network partition. This trade-off is complex, and is analyzed in detail in our discussion of the Top-Down (Section 3.2) and Bottom-Up (Section 3.3) algorithms.

3.1.1 Hierarchical Network Clusters

We organize physical network nodes into a virtual clustering hierarchy, by clustering nodes based on link costs which represents the cost of transmitting a unit amount of data across the link. We refer to this clustering parameter as ***internode/cluster traversal cost***. Nodes that are close to each other in the sense of this clustering parameter are allocated to the same cluster. We allow no more than max_{cs} nodes per cluster.

Clusters are formed into a hierarchy. At the lowest level, i.e., *Level 1*, the physical nodes are organized into clusters of max_{cs} or fewer nodes. Each node within a cluster is aware of

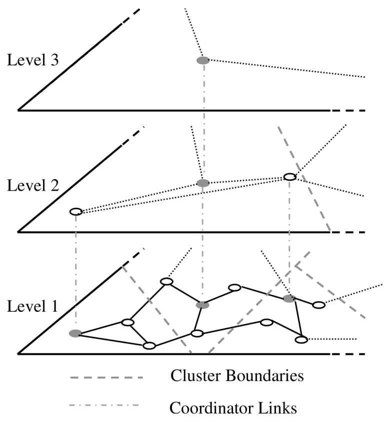

Fig. 4. Hierarchical network clusters.

the internode traversal cost between every pair of nodes in the cluster. A single node from each cluster is then selected as the *coordinator* node for that cluster and promoted to the next level, *Level 2*. There may be a set of nodes in a cluster, each of which qualifies to be a representative coordinator node as long as they do not modify the ordering of euclidean distances between the clusters. Nodes in *Level 2* are again clustered according to average internode traversal cost, with the cluster size again limited by max_{cs}. This process of clustering and coordinator selection continues until *Level N*, where we have just a single cluster. An example hierarchy is shown in Fig. 4.

As a result of our clustering approach, we can determine the upper bounds on the cost approximation at each level, which is described in the following theorem.

Theorem 1. *Let d_i be the maximum intracluster traversal cost at level i in the network hierarchy and $c_{act}(v_{nj}, v_{nk})$ be the actual traversal cost between the network nodes v_{nj} and v_{nk}. Then, the estimated cost between network nodes v_{nj} and v_{nk} at any level l, represented as $c^l_{est}(v_{nj}, v_{nk})$, is related to the actual cost as follows: $c_{act}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + \sum_{i=1}^{i<l} 2d_i$.*

Proof. At a particular level l, the cost of traversal between nodes v_{nj} and v_{nk} is given by the internode traversal cost between the nodes representing them at that level. However, each node will be resolved to some node in the underlying cluster at level $l-1$. Internode traversal costs at this level are bounded by the value d_{l-1}. Therefore, nodes at level $l-1$ will be at most d_{l-1} distance away from the node representing them at level l. Thus, the internode traversal costs between nodes v_{nj} and v_{nk} at level $l-1$ is given by $c^{l-1}_{est}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + 2d_{l-1}$. Similarly

$$c^{l-2}_{est}(v_{nj}, v_{nk}) \leq c^{l-1}_{est}(v_{nj}, v_{nk}) + 2d_{l-2}$$

$$\Rightarrow c^{l-2}_{est}(v_{nj}, v_{nk}) \leq c^{l}_{est}(v_{nj}, v_{nk}) + \sum_{i=l-2}^{i<l} 2d_i.$$

This process continues down the hierarchy. At level 1, the estimated cost is the same as the actual traversal cost and thus is at most $\sum_{i=1}^{i<l} 2d_i$ less than the actual cost. □

3.1.2 Discussion

The hierarchical organization is created and maintained as follows: When a node joins the infrastructure, it contacts an existing node that forwards the join request to its coordinator. The request is propagated up the hierarchy and the top-level coordinator assigns it to the top-level node that is closest to the new node. This top-level node passes the request down to its child that is closest to the new node. The child repeats the process, which continues until the node is assigned to a bottom-level cluster. Note that similar organization strategies appear in other domains such as hierarchies for Internet routing [25] and for data aggregation in sensor networks [26]. However, to the best of our knowledge, we are the first to use such hierarchical approximations and clustering techniques for distributed continual query optimization. The virtual hierarchy is robust enough to adapt as necessary. It can handle both node joins and departures at runtime. Failure of coordinator nodes can be handled by maintaining active backups of the coordinator node within each cluster. However, the issue of fault tolerance is beyond the scope of this paper.

In situations where nodes are distributed such that it is easy to find clusters meeting the clustering condition of *intercluster distances >> intracluster distances*, the planning decisions are likely to be less sensitive to the selection of coordinator nodes. However, situations where nodes in the entire system either are all widely distributed or are all close to one another in terms of network cost, may result in loosely defined clusters, which further impact the quality of coordinator nodes selected. Such situations are relatively rare. Also in the worst case, it is possible to choose appropriate values for max_{cs} in order to improve accuracy of the planning process. Also, note that since the hierarchy is only a virtual structure and since query deployment times (Section 4.6) are in the order of seconds, when it is known a priori that the node distribution in the network might possibly result in loosely defined clusters, it may be beneficial to compare planning decisions across multiple hierarchical structures with different values of max_{cs}.

3.1.3 Stream Advertisements

Stream advertisements are used by nodes in the network to advertise the stream sources available at that node. A node may advertise two kinds of stream sources—*base stream sources* and *derived stream sources*. We observe that each sink and deployed operator is a new stream source for the data computed by its underlying query or subquery. We refer to these stream sources as derived stream sources and the original stream sources as base stream sources. As a result of the advertisement of derived stream sources, nodes are now aware of operators that are readily available at multiple locations in the network and can be reused with no additional cost involved for transporting input data. The stream advertisements are aggregated by the coordinator nodes and propagated up the hierarchy. Thus, the coordinator node at each level is aware of all the stream sources available in its underlying cluster. Advertisements of derived stream sources are key to operator reuse in our algorithms. The advertisements are one-time messages exchanged only at the initial time of operator instantiation and deployment.

3.2 The Top-Down Algorithm

The *Top-Down* algorithm bounds suboptimality by making deployment decisions using bounded approximations of the underlying network; specifically, each coordinator's estimate of the distance between its cluster and other clusters. The algorithm works as follows: The query $\mathcal{Q}$ is submitted as input to the top-level (say level t) coordinator. The coordinator exhaustively constructs the possible query trees for the query, and then for each such tree constructs a set of all possible node assignments within its current cluster. The cost for each assignment is calculated and the assignment with least cost is chosen. An assignment of operators to nodes partitions the query into a number of views, each allocated to a single node at level t. Each node is then responsible for instantiating such a view using sources (base or derived) available within its underlying cluster. The allocated views act as the queries that are again deployed in a similar manner at level $t-1$, with all possible assignments within the cluster being evaluated exhaustively and the one with the least cost being chosen. This process continues until level 1, which is the level at which all the physical nodes reside, and operators are assigned to actual physical nodes. Since each level has fewer nodes and operators are progressively partitioned and assigned to different cluster coordinators, the search space is still much smaller compared to a global exhaustive search (even using DP). Whenever a coordinator is exhaustively mapping a portion of the query, it considers both base and derived streams available locally. Thus, operator reuse is automatically considered in the planning process. In particular, if the coordinator calculates that reuse would result in the best plan, derived streams are used; otherwise, operators are duplicated.

3.2.1 Bounding Search Space with the Top-Down Algorithm

In a network of N nodes that is organized into a clustering hierarchy, for a query $\mathcal{Q}$ over $K(>1)$ sources the search space depends on the clustering parameter max_{cs} and the resulting height $h(\approx log_{max_{cs}} N)$ of the hierarchy. We define the following:

$$\beta = h\Big(\frac{max_{cs}}{N}\Big)^{K-1}. \quad (1)$$

In Theorem 2, we prove that β represents the upper bound on the ratio of the search space of the Top-Down algorithm to that of the exhaustive search. Note that as the ratio $\frac{max_{cs}}{N}$ decreases linearly, β decreases exponentially. When $max_{cs} << N$, β is orders of magnitude less than 1 and thus, the Top-Down algorithm is orders of magnitude cheaper than exhaustive search. For example, for a query over four streams on a network with 1,000 nodes, with a max_{cs} value of 100, $\beta \approx 0.0015$.

Theorem 2. *Let $\mathcal{Q}$ be a query over K (>1) sources to be deployed on a network with N nodes. Let the clustering parameter used to organize the network into a hierarchical cluster be max_{cs}, and let the height of such a hierarchical cluster be h. If $\mathcal{O}_{top-down}$ represents the size of the solution space for the top-down algorithm, then $\mathcal{O}_{top-down} \leq \beta\mathcal{O}_{exhaustive}$.*

Proof. The worst case search space of the Top-Down algorithm results when all query tree nodes (sources, operators, and sink) appear in the same cluster. As in the case of Theorem 4, we compute this search space by considering all possible query trees and all possible placements of operators within a single cluster at each level.

We are given a network with N nodes, and a query $\mathcal{Q}$ over K streams $\langle \mathtt{S}_1, \mathtt{S}_2, \ldots \mathtt{S}_K\rangle$. At the top-level t, we have K streams. At any level, the search space is given by all plans (permutations of join orders) and all possible placements of each plan. Therefore, the search space $\mathcal{O}_t$ at level t is given by

$$\mathcal{O}_t = \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (max_{cs})^{(K-1)}.$$

In the worst case, the coordinator at each level may assign all streams to a single partition thereby causing the search space to be the same at all levels. Thus, $\mathcal{O}_{top-down}$ is given by

$$\mathcal{O}_{top-down} \leq h \times \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (max_{cs})^{(K-1)}. \quad (2)$$

Thus, from (2) and Lemma 1, we have

$$\mathcal{O}_{top-down} \leq \beta\mathcal{O}_{exhaustive}.$$

□

3.2.2 Suboptimality in the Top-Down Algorithm

The Top-Down algorithm works by propagating a query down the network hierarchy described in Section 3.1.1. Given a query $\mathcal{Q}$, at each level a coordinator chooses a query plan and a deployment with the least cost for the subquery assigned to it. As the network approximations increase at higher levels of the hierarchy (refer Theorem 1), it follows that the maximum approximation is incurred at the topmost level of the hierarchy. Therefore, the Top-Down algorithm is most suboptimal when all the edges of the query plan are deployed at the topmost level. The following theorem establishes the bounds on suboptimality of the top-down algorithm as compared to an optimal deployment.

Theorem 3. *A query $\mathcal{Q}$ deployed using the Top-Down algorithm over a network N is no more than $\sum_{e_k \in E^Q}(\sum_{i=1}^{i<h} 2d_i) \times s_k$ suboptimal compared to the optimal deployment of query $\mathcal{Q}$ over the same network N, where h is the number of levels in the network hierarchy of N, E^Q represents the set of edges of the tree chosen for query $\mathcal{Q}$, d_i is the maximum intracluster traversal cost at level i, and s_k is the stream rate for the kth edge e_k.*

Proof. The maximum suboptimality of the Top-Down algorithm occurs only when all the edges of the tree chosen for $\mathcal{Q}$ are mapped to the topmost level, i.e., no two nodes (operators or sources or sinks) lie in the same underlying cluster. The proof then follows directly from Theorem 1. □

The point of this proof is to establish a relationship between the hierarchical cluster structure and the suboptimality of the resulting solution. The intracluster traversal cost increases with the levels of the hierarchy since the hierarchical structure provides a more approximate representation of the network at higher levels. The height of the hierarchical structure in turn, is determined by the max_{cs} parameter and the density of node distributions in the network. The proof shows that the suboptimality can increase with increasing number of levels in the hierarchy and decreasing cluster density. This proof, along with Theorem 2, can help decide an optimization hierarchy that offers a desirable trade-off between search space and optimality for a given network. We present empirical results that corroborate these theorems in Section 4.

3.3 The Bottom-Up Algorithm

We now describe the *Bottom-Up* algorithm which propagates queries up the hierarchy, progressively constructing complete query execution plans. Unlike the Top-Down approach, the Bottom-Up algorithm does not provide a good bound on the suboptimality of the solution. However, in return, the Bottom-Up approach is usually able to further reduce the search space compared to the Top-Down algorithm. Thus, in situations where quick planning is needed, the Bottom-Up algorithm may be appropriate, perhaps to be replaced later with a Top-Down deployment.

Queries are registered at their sink. When a new query Q over base stream sources arrives at a sink at *Level 1*, the sink informs its coordinator at *Level 2*. The coordinator rewrites the query Q as Q' with respect to two views—V^Q_{local} and V^Q_{remote}, where V^Q_{local} is composed of base and derived sources available locally within the cluster and V^Q_{remote} is composed of base sources not available locally. The coordinator deploys V^Q_{local} within the current cluster, and then advertises V^Q_{local} as a derived stream at the next level. The above rewriting causes any joins between local streams to be deployed within the current cluster, leaving the joins of local streams with remote streams or joins between remote streams to be deployed further up in the hierarchy. The coordinator then requests Q' from its next level coordinator. This process continues up the hierarchy, with the query Q' progressively decomposed into locally available views and remote views and the rewritten query being requested from the current cluster's coordinator. The coordinator performs an exhaustive search only within its underlying cluster to determine an optimal execution plan for V^Q_{local}. The search space is limited to a single network partition and the local subquery.

Operator reuse is taken into consideration by coordinators by taking into account all possible constructions of V^Q_{local} that utilize derived sources within the cluster. When using a derived stream source, communication costs for transporting input data to the node that is the source of the derived stream, and processing costs for computing the result of the operator are incurred only once. Note that if it is cheaper to duplicate operators rather than reuse existing ones, the coordinator will do so.

For example, assume that query $Q1$ described in Section 2.1 arrives at a node which belongs to a cluster where sources FLIGHTS and CHECK-INS are available locally. The Bottom-Up algorithm proceeds as follows: $Q1$ is partitioned into local and remote views V^Q_{local} and V^Q_{remote}, respectively, where V^Q_{local} consists of sources FLIGHTS and CHECK-INS and V^Q_{remote} consists of source WEATHER as shown below.

V^Q_{local}: **SELECT** `FLIGHTS.STATUS, CHECK-INS.STATUS,`
`FLIGHTS.DESTN` **FROM** `FLIGHTS, CHECK-INS`
WHERE `FLIGHTS.DEPARTING='ATLANTA'` **AND**
`FLIGHTS.NUM = CHECK-INS.FLNUM`
AND `FLIGHTS.DP-TIME - CURRENT_TIME <`
`12:00:00`

V^Q_{remote}: **SELECT** `WEATHER.FORECAST, WEATHER.CITY` **FROM**
`WEATHER`

Query $Q1$ is then rewritten (with V^Q_{remote} expanded) as $Q1'$ given by

$Q1'$: **SELECT** `FLIGHTS.STATUS, WEATHER.FORECAST,`
`CHECK-INS.STATUS` **FROM** V^Q_{local}, `WEATHER`
WHERE `FLIGHTS.DESTN = WEATHER.CITY`

V^Q_{local} is deployed within the current cluster using any locally available derived streams, if required. Thus, the join between sources FLIGHTS and CHECK-INS is deployed locally within the cluster. V^Q_{local} is then advertised as a derived stream and query $Q1'$ is propagated to the next level for deployment at some higher level cluster. This process continues up the hierarchy until all sources for $Q1'$ are found locally in some cluster. At that point, all operators are placed at appropriate representative nodes and passed down the hierarchy for placement on an actual physical node.

3.3.1 Bounding Search Space with the Bottom-Up Algorithm

Recall our definition of β in Section 3.2.1. We now show in Theorem 4 that β also represents the upper bound on the ratio of the search space of the Bottom-Up algorithm to that of the exhaustive search. Although the worst case bounds are the same for the two algorithms, in Section 4.1 we show experimentally that the Bottom-Up algorithm examines a smaller search space in the average case. As before, when $max_{cs} << N$, β is orders of magnitude less than 1. Thus, the search space of the Bottom-Up algorithm is orders of magnitude less than the exhaustive search space.

Theorem 4. *Let $\mathcal{O}_{bottom-up}$ represent the size of the solution space for the bottom-up algorithm. Then, $\mathcal{O}_{bottom-up} \leq \beta \mathcal{O}_{exhaustive}$.*

Proof. We are given a network with N nodes, and a query Q over K streams $\langle S_1, S_2, \ldots S_K \rangle$. Let σ_i represent the number of streams, for query Q, requested by a node at level $i-1$ and available within the partition of a single coordinator at level i. Also, $\sigma_1 + \ldots + \sigma_h = K$. Let α_i represent the actual number of streams to be considered at level i. At the level where $V^{Q_i}_{remote} = \phi$, $\alpha_i = \sigma_i$. At all other levels, $\alpha_i = \sigma_i + 1$ to take into consideration the presence of the remote stream $V^{Q_i}_{remote}$. Thus, $\alpha_1 + \ldots + \alpha_h \leq K + h$. At any level, the search space is given by all plans (permutations of join orders) and all possible placements of each plan. Thus, the search space $\mathcal{O}_i$ at level i with α_i streams is given by

$$\mathcal{O}_i \leq \left(\frac{\alpha_i! \times (\alpha_i - 1)!}{2^{\alpha_i - 1}} \right) \times (max_{cs})^{(\alpha_i - 1)}.$$

Thus, the total search space in the Bottom-Up algorithm, $\mathcal{O}_{bottom-up}$ for a query Q is

$$\mathcal{O}_{bottom-up} \leq \sum_{i=1}^{i \leq h} \mathcal{O}_i. \quad (3)$$

Since ∀ i, $\alpha_i \leq K$, and not all $\alpha_i = K$ (since the query is totally composed of only K streams and streams found at each level are different), we have

$$\begin{aligned}\mathcal{O}_{bottom-up} &\leq \sum_{i=1}^{i\leq h}\left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (max_{cs})^{(K-1)} \\ &\leq \left(\frac{K! \times (K-1)!}{2^{K-1}}\right) \times (max_{cs})^{(K-1)} \times (h).\end{aligned}$$

Thus, from Lemma 1 and the above equation, we have $\mathcal{O}_{bottom-up} \leq \beta\mathcal{O}_{exhaustive}$. □

3.3.2 Suboptimality in the Bottom-Up Algorithm

The Bottom-Up algorithm partitions queries into locally and remotely available views as the result of which all local sources are now represented as a single source deployed at the coordinator. This results in a pruning of the plan search space since only join orderings between streams available within a single cluster are considered. While the Bottom-Up algorithm can find optimal join orderings among local sources, the resulting overall execution plan may be suboptimal. As an example, consider a high-volume stream S_r that is in a remote cluster, and which we want to join with two low volume, local streams S_1 and S_2. An overall optimal plan might be to perform a selective join between S_r and S_1 in the remote cluster, and then stream the resulting (low-volume) intermediate results to the local cluster for joining with S_2. The Bottom-Up algorithm will not consider this plan. However, note that the Bottom-Up algorithm may instead stream the results of $S_1 \bowtie S_2$ to the remote cluster for joining with S_r.

In the worst case, the resulting deployment may be arbitrarily bad making it impossible to bound the suboptimality of the algorithm. However, note that the situations under which this algorithm performs badly can be well characterized: it performs badly when streams available remotely have significantly higher data rates than those available close to the sink. In order to overcome this limitation, we next present heuristic-based hybrid algorithms that aim at improving the planning process of the Bottom-Up algorithm while retaining the advantage of a small search space.

3.4 The NPC Algorithm

In this section, we introduce a heuristic-based hybrid algorithm that combines the advantages of reduced search space from the Bottom-Up algorithm and improved query planning from the Top-Down algorithm. We present a heuristic-based hybrid algorithm—the *NPC* algorithm. The NPC algorithm is a probabilistic algorithm that uses cost estimates of local and delayed query-planning decisions to guide the planning process. A decision to choose a join order at the current level may result in a penalty if a poor join order is chosen. On the other hand, delaying the planning process to the next level will result in wasted planning time and also a possible increase in cost due to coarser approximations. In the NPC algorithm, the query planning process is delayed to the next level in the hierarchy only when the cost of making a decision at the current level exceeds the estimated cost of delaying the decision to the next level in the hierarchy. We next describe the computation of local and delayed cost estimates.

Let V^Q_{local} and V^Q_{remote} represent the subqueries composed of sources available locally and remotely, respectively. In the worst case, if a poor join order is chosen as the result of making a local decision, it may result in a high-volume remote query that needs to be streamed to the current cluster for joining with the local query. Let $\mathcal{C}_l$ denote the cost incurred immediately within a cluster at level l by making a join ordering decision locally at level l. We use the term "NPC," Γ_l, to indicate all present and future costs that may be incurred as the result of making a local join ordering decision at the level l. Then, the estimated NPC Γ_l is computed as follows:

$$\Gamma_l = \mathcal{C}_l + \sum_{i=l+1}^{h} p_i \times \lambda\left(V^Q_{remote}\right) \times d_i,$$

where p_i represents the probability of finding V^Q_{remote} at the level i, $\lambda(V^Q_{remote})$ represents the data rate of stream V^Q_{remote}, h represents the height of the hierarchy, and d_i represents the maximum intracluster traversal cost at level i. The probability of finding the remote query at a particular level is computed based on the fraction of network visible at that level, assuming sources are likely to appear anywhere within the network. In this expression, the first term represents the present cost and the second term represents all future costs likely to be incurred at higher levels in the hierarchy. Note that Γ_l is a conservative estimate of the future costs aimed mainly at penalizing query partitioning where the join order results in a high-volume remote stream that needs to be transported across longer distances.

The cost of delaying the decision to the next level, i.e., cost Ω_l at level l is computed as follows:

$$\Omega_l = \sum_{i=l+1}^{h} p_i \times \lambda(Q) \times d_i.$$

In order to compute Ω_l, we compute the expected future costs of delaying the query partitioning decision to the next level. The NPC algorithm then performs query partitioning at the current level l if $\Omega_l \geq \Gamma_l$. Unlike the other algorithms, the NPC algorithm requires knowledge of the hierarchical structure in terms of height, number of nodes in a cluster, and maximum intracluster traversal costs at each level. It also requires knowledge of join selectivities.

Since the NPC algorithm attempts to avoid poor join orders, it is expected to perform better than the Bottom-Up algorithm. However, since it continues to make query partitioning decisions based only on efficiency of join orders, oblivious to the availability of reuse opportunities, it is expected to produce less efficient deployments as compared to the Top-Down algorithm. The NPC algorithm performs query partitioning when it perceives that partitioning the query at some level is beneficial. As a result, it is more effective in reducing the search space compared to the Top-Down algorithm.

4 Experiments

We present both simulation-based experiments and prototype experiments conducted on Emulab [2] using IFLOW [1].

Our experiments focus on 1) the effect of the max_{cs} clustering parameter on the trade-off between suboptimality and search space, 2) the effectiveness of our algorithms as compared to existing approaches, and 3) and the efficiency of our algorithms compared to an optimal solution computed through an exhaustive search. Our experiments show that our algorithms result in acceptable suboptimality: the Top-Down algorithm is suboptimal by only 10 percent and the Bottom-Up algorithm by 34 percent while exploring less than 1 percent of the total search space. At the same time, our algorithms clearly outperform existing approaches. For example, the Bottom-Up algorithm reduces cost by nearly 25 percent when compared to the *In-network* [5] algorithm while exploring only a small fraction of the search space. Also, the NPC algorithm allows us to further fine tune the trade-off between search space and suboptimality and help us achieve plans that were close to the Top-Down algorithm in optimality and Bottom-Up algorithm in search space.

4.1 Experimental Setup

Our simulation experiments were conducted over transit-stub topology networks generated using the standard tool, the GT-ITM Internet-work topology generator [27]. Most experiments were conducted using a 128-node network, with a standard Internet-style topology: one transit (e.g., "backbone") domain of four nodes, and four "stub" domains (each of eight nodes) connected to each transit domain node. Link costs (per byte transferred) were assigned such that the links in the stub domains had lower costs than those in the transit domain, corresponding to transmission within an Intranet being far cheaper than long-haul links. As described in Section 2.2, we adopt the "network usage" metric [9] to compute costs of query deployments. Recall that, the network usage $u(q)$ of a query q represents the total amount of data that is in-transit for a query at any given instant.

As described in Section 3.1.1, our network is organized into a virtual clustering hierarchy based on link costs which represent the cost of transmitting a unit amount of data across the link. We used the K-Means [28] clustering in order to create the clustering hierarchy.

4.1.1 Workloads

We evaluate our approaches using two different workloads: a synthetic workload generated using a random workload generator and a real enterprise workload based on the enterprise OIS used by Delta Airlines [10], [1]. We used a synthetic workload so that we could experiment with a large variety of stream rates, query complexities, and operator selectivities. Our synthetic workload was generated using a uniformly random workload generator. The workload generator generated stream rates, selectivities, and source placements for a specified number of streams according to a uniform distribution. It also generated queries with the number of joins per query varying within a specified range (2-5 joins per query) with random sink placements.

The enterprise workload is a real-world workload consisting of gate-agent, terminal and monitoring queries posed as part of the day-to-day operations of Delta Airlines' enterprise OIS. The query workload is based on the five query sources whose characteristics are shown in Fig. 5. Each update record was assumed to be of the same size (100 bytes). Each query definition includes window (RANGE and SLIDE, i.e., the window size and frequency of computation of results, respectively) specifications for each of the input streams.

Source	Rate	Selectivity
Flights	1500/5 min	1/flight/5 min
Check-ins	240/min	2/flight/min
Baggage	500/min	4/flight/min
Weather	450/5 min	1/dest/5 min
Sales	70/min	1/dest/min

Fig. 5. Enterprise workload.

Gate agent queries, which originate at the gate of departure of a flight, constitute 80 percent of the workload, and the SLIDE for these queries is set to 1 minute. The queries use the following template.

```
 Q1: SELECT FL.GATE, BG.STATUS, CI.STATUS
  FROM FLIGHTS FL [RANGE 5 MINUTES], BAGGAGE BG
[RANGE 1 MINUTE], CHECK-INS CI [RANGE 1 MINUTE]
  WHERE FL.NUM = CI.FLIGHT AND FL.NUM =
  BG.FLIGHT AND FL.NUM = ?;
```

Terminal queries follow the template of query Q2 given below and represent 15 percent of the workload. The results of these queries, evaluated every minute, are streamed to overhead terminal displays.

```
 Q2: SELECT FL.NUM, FL.GATE, BG.AREA, CI.STATUS,
WR.FORECAST
  FROM FLIGHTS FL [RANGE 5 MINUTES], WEATHER WR
  [RANGE 5 MINUTES], CHECK-INS CI [RANGE 1 MINUTE],
BAGGAGE BG [RANGE 1 MINUTE] WHERE FL.DEST =
  WR.CITY AND FL.NUM = CI.FLIGHT
  AND FL.NUM = BG.FLIGHT AND FL.TERMINAL = ?
  AND FL.CARRIER_CODE = ''DL'';
```

Finally, the last 5 percent of the workload represent long-running ad-hoc monitoring queries over any combination of the five sources. For these queries, window ranges and slides are uniformly distributed between [1-5] minutes for all streams. Note that each gate agent, terminal and monitoring query may have unique selection predicates. In the current work, our focus is on join ordering and discovering join reuse opportunities. In our simulation experiments, sharing join operators between queries with different selection criteria over the input stream is implemented by modifying selection predicates at runtime. In our prototype implementation, the selections are instantiated as parameterized filters [29], and this reduces the task of selection predicate modification to the task of changing the parameter associated with the filter. While evaluating the benefit of reusing a join operator, the cost of projecting additional columns and relaxing the filter specifications upstream are also taken into consideration and operators are reused only when the resulting cost is less than that of deploying a new operator. The synthetic workload is used

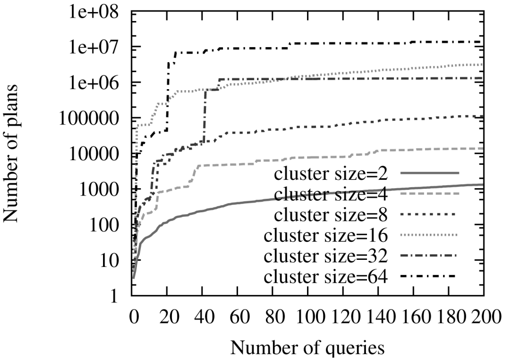

Fig. 6. Bottom-Up: plans.

to study the trade-off between suboptimality and search space in our algorithms.We use the enterprise workload consisting of 300 queries to compare the performance of our algorithms with existing techniques in a realistic setup.

4.2 Tuning Cluster Size: Trade-Off between Suboptimality and Search Space

An exhaustive search of all possible query plans and all possible placement of operators may not be feasible as network size increases. For example, an exhaustive search on a 128-node network for the deployment of a single query over five stream sources required enumeration of approximately 4.83×10^{10} plans that took nearly 3 hours to complete. In this section, we demonstrate how the max_{cs} parameter can be used to tune the trade-off between the suboptimality of the heuristic and minimizing the search space. The experiments were conducted using the synthetic workload described in Section 4.1.

4.2.1 Effect of Cluster Size on Search Space

In this experiment, we studied the effect of the cluster size parameter max_{cs} on the search space with the Bottom-Up and Top-Down algorithms. Figs. 6 and 7 depict the cumulative number of plans examined on a log scale, with varying max_{cs} for the Bottom-Up and Top-Down algorithms, respectively. As the figure shows, the number of plans increases as max_{cs} increases.

Interestingly, we notice that in both algorithms, a max_{cs} value of 32 results in a smaller search space than a value of 16. In both cases, the hierarchy had the same number of levels, but in the case of $max_{cs} = 32$, the upper level clusters were smaller (since the lower level clusters were larger). Since many sources are found remotely, most of the planning is done at the upper levels, and having small cluster sizes at those levels results in a smaller search space overall.

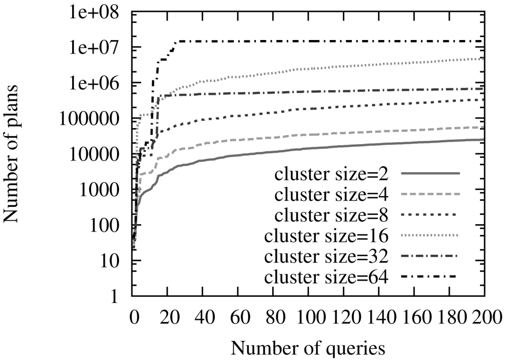

Fig. 7. Top-Down: plans.

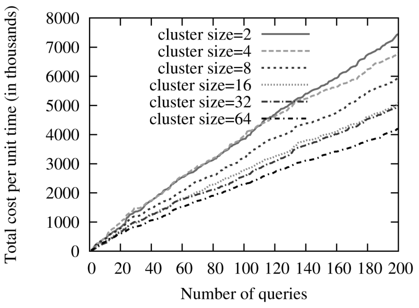

Fig. 8. Bottom-Up: cost.

In contrast, a max_{cs} value of 64 resulted in the maximum search space, nearly an order of magnitude larger than $max_{cs} = 16$. This is a straightforward effect of the increased probability of finding sources in a larger cluster. For example, a query over four streams, with all streams found within a 57-node *Level 1* cluster, considers as many as 3.3×10^6 deployments.

In general, the Bottom-Up algorithm considers on an average 67 percent fewer plans than the Top-Down algorithm. The exception to this rule occurs when the virtual hierarchy structure is an unbalanced structure with very few nodes at the top, as in the case of a $max_{cs} = 64$ (three top-level nodes) and $max_{cs} = 32$ (five top-level nodes). In these cases, due to the small cluster size at the level where the query is partitioned, the Top-Down algorithm has a smaller search space than the Bottom-Up algorithm.

4.2.2 Effect of Cluster Size on Cost

Fig. 8 shows the cumulative deployed cost per unit time of queries deployed incrementally using the Bottom-Up algorithm for different values of the max_{cs} parameter. It can be noticed that cost decreases as the max_{cs} value is increased. For example, a max_{cs} value of 64 results in a 21 percent decrease in cost compared to a max_{cs} value of 8. With smaller cluster sizes, the number of levels in the hierarchy increases. As a result, more deployments are computed at higher levels resulting in greater approximations. To summarize, in terms of suboptimality, fewer levels and more nodes per level is best. In terms of search space, fewer nodes per level is best. A useful guideline for choosing max_{cs} for the Bottom-Up algorithm is

- choose the largest value of max_{cs} that results in a search space (Theorem 4) that is acceptable.

Fig. 9 shows the effect of the cluster size parameter max_{cs} on the cost in the Top-Down algorithm. Note that large values of max_{cs} (> 4) result in deployed costs that are close to each other. The Top-Down algorithm considers all possible operator orderings at the topmost level (regardless of max_{cs}). This results in a good and mostly “similar” choice of operator ordering for a range of max_{cs} values. However, if max_{cs} is too small, there are many levels in the hierarchy

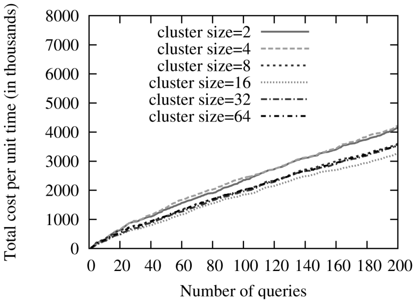

Fig. 9. Top-Down: cost.

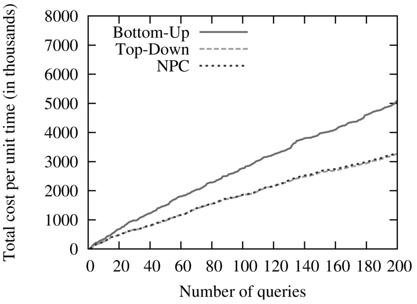

Fig. 10. NPC algorithm: cost.

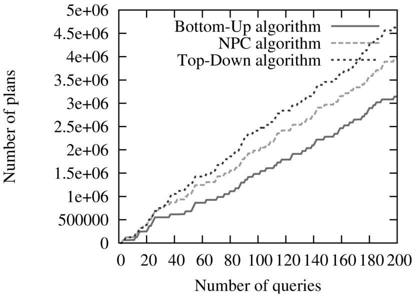

Fig. 11. NPC algorithm: plans.

and each level adds more inaccuracy to the approximation. Hence, a useful guideline for the Top-Down algorithm is

- choose the smallest value of max_{cs} that is large enough so that the height of the hierarchy results in reasonable suboptimality (based on Theorem 3).

4.3 Efficiency of NPC Algorithm

The NPC algorithm allows us to further fine tune the trade-off between search space and suboptimality. Fig. 10 shows the cost with the NPC algorithm with $max_{cs} = 16$ as compared with the Top-Down and Bottom-Up algorithms. We choose to present the graph for this value of max_{cs} since this is the largest value that results in a balanced virtual hierarchical structure with almost full clusters at each level representing the standard behaviors of the Top-Down and Bottom-Up algorithms. As the figure shows, the NPC algorithm results in plans that are suboptimal by only 1 percent compared to the Top-Down algorithm. At the same time, as Fig. 11 shows, the NPC algorithm explores 14 percent fewer plans than the Top-Down algorithm. Note that for each value of max_{cs} where the Top-Down algorithm explored fewer plans than the Bottom-Up algorithm, the NPC algorithm explored only as many plans at the Top-Down algorithm while still resulting in solutions that were close to the Top-Down algorithm in optimality. While the NPC algorithm avoids poor join orders, it performs less efficiently than the Top-Down algorithm since it is unable to take into account reuse opportunities that may appear at the upper levels in the hierarchy while deciding on join orders.

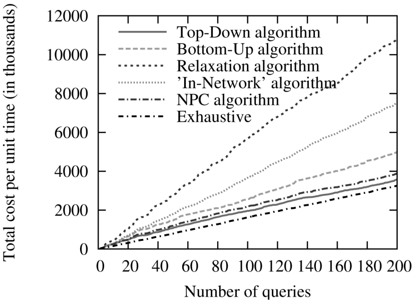

Fig. 12. Comparison with existing approaches.

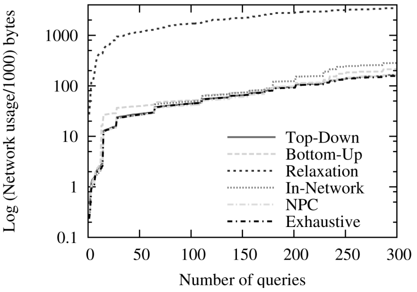

Fig. 13. Enterprise workload: cost.

4.4 Comparison with Existing Approaches

In this experiment, we compare our Top-Down and Bottom-Up approaches with existing approaches—the Relaxation algorithm [9] and *In-network* [5], a network-aware query processing algorithm. Both Relaxation and In-network are phased deployment approaches that first plan and then deploy (see Fig. 1a). Operator reuse was implemented through stream-advertisements. The communication cost of advertisements was negligible compared to the data streams themselves.

Figs. 12 and 13 show the cumulative cost of deployments computed using the Top-Down, Bottom-Up, and NPC algorithms as compared with the Relaxation and In-network algorithms, using the synthetic and enterprise workload, respectively. The graphs also show the costs of optimal deployments computed using an exhaustive search. Operator reuse was taken into consideration for all algorithms. We used a 3D cost space for the Relaxation algorithm and considered a virtual hierarchy with max_{cs} 32 for the Top-Down, Bottom-Up, and NPC algorithms. We chose this value of max_{cs} based on the above guideline for the Bottom-Up algorithm; and we used the same value for the Top-Down and NPC algorithms to provide an apples-to-apples comparison. In order to correspond with this max_{cs} value, we divided the network into five *zones* for the In-network algorithm.

Fig. 12 shows that, under the synthetic workload, when compared to the In-network algorithm, the Top-Down algorithm can provide nearly 40 percent additional cost

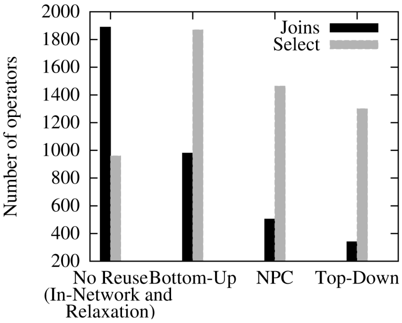

Fig. 14. Enterprise workload: # operators.

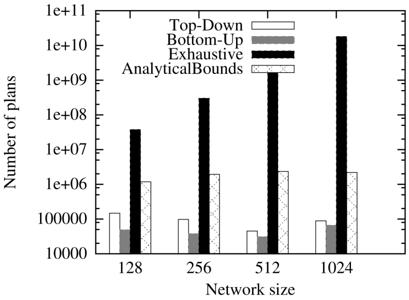

Fig. 15. Scalability with network size.

savings per unit time, and the Bottom-Up algorithm, savings of 27 percent. Also, note that, the search space of the In-network algorithm was nearly 70 percent that of the Top-Down algorithm and 200 percent that of the Bottom-Up algorithm. When compared to the Relaxation algorithm, the Top-Down algorithm reduces cost by nearly 59 percent and the Bottom-Up algorithm by nearly 49 percent. The search space of the Relaxation algorithm is not directly comparable with that of the Top-Down and Bottom-Up algorithms, due to the variable number of iterations that may be performed for each step of the Relaxation algorithm. In our experiment, the 3D cost space [30] was calculated using 4,000 iterations, and we used as many iterations for the Relaxation algorithm and the running time was comparable to that of the Bottom-Up algorithm.

Fig. 13 represents the network usage of the deployment algorithms under the enterprise workload, and Fig. 14 shows the cumulative number of operators deployed after 300 queries under the same workload. Note that the y-axis in Fig. 13 uses a log scale. The graph shows that compared to the In-network algorithm, Top-Down, Bottom-Up, and NPC algorithms result in approximately 42 percent, 25 percent, and 41 percent cost savings, respectively. However, note that, the In-network algorithm examined only 3 percent fewer plans compared to Top-Down and 170 percent more plans than Bottom-Up with this workload. When compared to the Relaxation algorithm, Top-Down, Bottom-Up, and NPC algorithms result in approximately 96 percent, 93 percent, and 95 percent cost savings, respectively.

Fig. 14 shows the total number of deployed operators. It is a well-known fact that join operators are expensive and reduce throughput. Fig. 14 shows that algorithms using our framework resulted in better utilization of system resources with the Top-Down, Bottom-Up, and NPC algorithms utilizing nearly 81 percent, 73 percent, and 48 percent fewer join operators compared to the phased-deployment algorithms (In-network and relaxation). The increase in the number of selection filters with our algorithms results from telescoping filter placements (i.e., placing a less restrictive filter first to allow a join operator to be reused, followed by more restrictive filters) to facilitate join reuse. Although the number of select operators has increased, these operators are stateless and require fewer processing resources. By reducing the number of join operators, we expect that the system throughput will increase significantly.

Figs. 12 and 13 also allow us to compare the deployed costs of our algorithms with the optimal solution computed using DP under the two workloads. Fig. 12 shows that the Top-Down algorithm performs better than the Bottom-Up algorithm by nearly 19 percent and when compared to the optimal, the Bottom-Up algorithm performs suboptimally by 34 percent and the Top-Down algorithm by only 10 percent. The NPC algorithm performs suboptimally by only 18 percent. Similarly, with the enterprise workload, as Fig. 13 shows, the suboptimality of the Top-Down algorithm is only 5 percent, while that of the Bottom-Up algorithm is 36 percent. On the other hand, the performance of the NPC algorithm is close to that of the Top-Down algorithm, with a suboptimality of only 6 percent.

4.5 Scalability with Network Size

In this experiment, we study the scalability of the algorithms with respect to the number of deployments considered as network size increases. We generated a workload of 100 queries using 10 stream sources with each query performing joins over four streams. We measured the average number of deployments considered over four different transit-stub topologies of different sizes generated using GT-ITM. Again, sinks were placed at random nodes in the network. Fig. 15 shows the deployments considered for a single query with Bottom-Up and Top-Down algorithms with max_{cs} 32 and exhaustive search. The figure also shows how the average case (experimental) compares with the worst case (theoretical) analytical bounds. Again, the value of max_{cs} was set to 32 to produce the largest feasible search space. (An exhaustive search on a 128-node network for the deployment of a single query took nearly 3 hours to complete on our system.) Note that the increase in $\mathcal{O}_{exhaustive}$ is offset by the decrease in β such that the worst case bounds are nearly identical across the different networks. Note that the y-axis has a log scale.

The values for exhaustive search were calculated using Lemma 1 and the analytical bounds using Theorems 2 and 4. Clearly, performing exhaustive searches in such systems is infeasible. Both the Top-Down and Bottom-Up algorithms decrease the search space by at least 99 percent. We also see that the search space per query with Bottom-Up is nearly 45 percent less than that of Top-Down. This can be attributed to the early splitting of queries between levels in the Bottom-Up algorithm resulting in fewer operators being considered for placement at each level. Meanwhile, the Top-Down algorithm must consider all operator deployments at all levels in the hierarchy.

Although the search space of Top-Down and Bottom-Up algorithms seems to first decrease with network size and then increase, note that this is only a particular characteristic of our

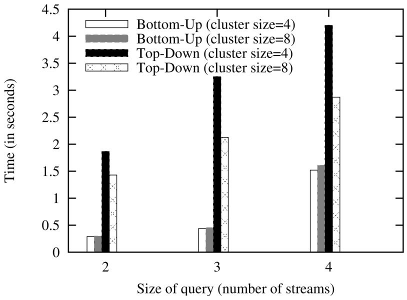

Fig. 16. Query deployment time.

sample networks. For example, clustering using max_{cs} 32 resulted in an average lowest level (i.e., *Level 1*) cluster size of 26 with a 128-node network, and 15 with a 510-node network. Thus, the search space for a 510-node network is less than that of the 128-node network. Note that the search space, while being limited by the max_{cs} parameter, is affected by the average cluster size too, which depends on the particular network topology.

4.6 Deployment Time

We conducted prototype experiments on Emulab using IFLOW [1], our implementation of the distributed data stream system which supports hierarchies and advertisements as described earlier. The testbed on Emulab consisted of 32 nodes (Intel XEON, 2.8 GHz, 512-Mbyte RAM, RedHat Linux 9), organized into a topology that was again generated with GT-ITM. Links were 100 Mbps and the internode delays were set between 1 and 6 ms. The workload for the following experiments consisted of 25 queries over eight stream sources and sinks distributed across the system. The number of joins per query varied from 1 to 3.

The experiment conducted on Emulab was aimed at measuring the time to deployment of a query over the system when using our algorithms. Fig. 16 shows the average deployment time in seconds for different query sizes. We observe that the deployment times of the Bottom-Up algorithm is almost 70 percent less than that of the Top-Down algorithm. This can be attributed to two factors: 1) the smaller search space in the Bottom-Up algorithm and 2) the fact that the Top-Down algorithm must always traverse the entire depth of the network hierarchy. We also observe that the deployment time of the Top-Down algorithm decreases with increasing max_{cs} value. With lower max_{cs}, there are more hierarchy levels to be traversed, resulting in higher deployment times. Our experiment allows us to conclude that our algorithms can greatly reduce the search space for the query deployment problem while offering efficient deployments with acceptable suboptimality.

5 Related Work and Discussion

Distributed query optimization has received a great deal of attention from researchers since the 1980s [24]. Classic efforts in this area include R^* [31] and Distributed INGRES [32]. Both the R^* algorithm and the Distributed INGRES algorithm execute at a *master* site. Since our system may consist of thousands of nodes, it is infeasible to maintain all network information at a single node or perform exhaustive searches for an optimal deployment.

A number of data-stream systems including SQL-based systems such as STREAM [33], dQUOB [13], TelegraphCQ [34], and NiagaraCQ [35], as well as general-purpose systems such as GATES [20], Borealis [7], and IFLOW [1] have been developed to process queries over continuous streams of data. Centralized stream processing systems have explored use of techniques like common subexpression elimination [13] and commutative ordering of operators [36], [35] to enable operator reuse. However, our problem is complicated by the need to consider an operator's network location while computing plans that can take advantage of reuse. At the other extreme, novel systems like Eddies [37] have also used a tuple-by-tuple routing approach to adaptively decide the execution plan of a query.

The paradigm of in-network query processing has been used earlier in sensor networks [4], [3] and also in scientific data flows [20] and large-scale visualizations [13] with data manipulations sometimes pushed to the source for efficiency. The use of this technique in stream-based systems to only decide operator placement when the query tree is already known is described in [5] and [9]. The network-aware algorithms in [5] first perform phased deployments which we have shown to be suboptimal. Second, they do not address the important question of how the query should be divided and assigned to different portions of the network. Clearly, as seen from our experiments on varying cluster sizes, this decision can impact the efficiency of the resulting deployments. Also, no analysis is provided on the impact of the number of zones and the placement heuristics on the computational complexity of the algorithms.

The Relaxation algorithm [9] is a novel heuristic for operator placements in distributed stream processing systems. However, the approach does not take into consideration planning and deployment simultaneously resulting in increased suboptimality, both due to lost reuse opportunities and the subsequent approximate placement decisions. Optimal placement for a single query on a sensor network is considered in [38]. However, we consider the more generic problem of determining both operator ordering and placements. Moreover, both our algorithms are able to bound the search space, and the Top-Down algorithm is also able to bound the suboptimality.

In our current design, we consider that communication overhead, especially the amount of data transmitted, is the dominating cost in a distributed data stream system for continuous streaming applications, in comparison to the amount of local processing at each node. Although our cluster hierarchy creation using the *network usage* metric does not explicitly take into account the differences in computing power of individual nodes, it does incorporate the effect of imbalance between computing capacity and communication capacity of a node in the process of creating our cluster hierarchy using the delay parameter in the *network usage* metric [9].

In a distributed data-stream system where communication and processing costs are not only high but also changing continuously, an optimal query execution plan should ideally try to achieve multiple objectives at the same time, such as minimum response-time, minimum communication, and processing cost per unit time. However, these objectives are often conflicting, since it is possible that lower delay paths have higher communication cost, or paths that incur low communication cost can cause a processing overload at some intervening network node. Epstein et al. [21] have developed an approach to optimize across such conflicting objectives, where the optimization criteria is some "application-dependent" cost function expressed in terms of objectives, such as communication cost and response time at each node; the latter is often dependent on the node's processing and buffering capacity and memory utilization. Note that, our framework and the query optimization algorithms presented in this paper are capable of incorporating existing "application-dependent" cost functions to find an efficient query execution plan, both in terms of cluster coordinator selection and distributed query planning.

An alternative approach to deal with a system of nodes with heterogeneous processing capacities would be to design a more complex task scheduler for the coordinator node, which allows the coordinator node that maps the query operators to the actual physical node to additionally take into consideration available processing capacities.

6 Conclusion

We have described a distributed stream query optimization framework that integrates query planning and deployment through hierarchical network partitions. Our framework consists of two key components: a hierarchical clustering of network nodes that allows network approximations and stream advertisements that enable operator reuse. We described three alternative algorithms—*Top-Down*, *Bottom-Up*, and *Hybrid*, which exploit different ways of using hierarchical network partitions for operator-level reuse and search space reduction. We show that although Top-Down and Bottom-Up algorithms can both choose efficient deployments while exploring only a small fraction of the search space, the Top-Down algorithm is more effective in limiting the suboptimality of the solutions, while the Bottom-Up approach is more effective in reducing the search space and the time-to-deployment. The hybrid algorithm *NPC* find efficient execution plans while examining a very small search space, allowing us to further tune the trade-off between search space and algorithm suboptimality. We show through both experimental and analytical results that our algorithms are efficient and scalable at costs comparable to optimal while exploring much fewer plans.

Acknowledgments

This work was partially funded by grants from the US National Science Foundation (NSF) Distributed Systems Research program and NSF CyberTrust program, an IBM PHD fellowship, and an IBM SUR grant.

References

[1] V. Kumar et al., "Implementing Diverse Messaging Models with Self-Managing Properties Using IFLOW," *Proc. Third IEEE Int'l Conf. Autonomic Computing (ICAC)*, 2006.

[2] *Emulab Network Testbed,* http://www.emulab.net/, 2008.

[3] Y. Yao and J. Gehrke, "The Cougar Approach to In-Network Query Processing in Sensor Networks," *SIGMOD Record*, 2002.

[4] S. Madden, M.J. Franklin, J.M. Hellerstein, and W. Hong, "TAG: A Tiny Aggregation Service for Ad-Hoc Sensor Networks," *Proc. Fifth Symp. Operating Systems Design and Implementation (OSDI)*, 2002.

[5] Y. Ahmad and U. Cetintemel, "Network-Aware Query Processing for Stream-Based Applications," *Proc. 30th Int'l Conf. Very Large Data Bases (VLDB)*, 2004.

[6] C. Olston, J. Jiang, and J. Widom, "Adaptive Filters for Continuous Queries over Distributed Data Streams," *Proc. ACM SIGMOD*, 2003.

[7] D.J. Abadi et al., "The Design of the Borealis Stream Processing Engine," *Proc. Second Biennial Conf. Innovative Data Systems Research (CIDR)*, 2005.

[8] M.A. Shah, J.M. Hellerstein, S. Chandrasekaran, and M.J. Franklin, "Flux: An Adaptive Partitioning Operator for Continuous Query Systems," *Proc. 19th Int'l Conf. Data Eng. (ICDE)*, 2003.

[9] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-Aware Operator Placement for Stream-Processing Systems," *Proc. 22nd Int'l Conf. Data Eng. (ICDE)*, 2006.

[10] V. Oleson, K. Schwan, G. Eisenhauer, B. Plale, C. Pu, and D. Amin, "Operational Information Systems—An Example from the Airline Industry," *Proc. First Workshop Industrial Experiences with Systems Software (WIESS)*, 2000.

[11] "Amazon ec2," *Amazon Elastic Computing Cloud,* aws.amazon.com/ec2, 2008.

[12] *IBM Websphere,* http://www-306.ibm.com/software/websphere/, 2008.

[13] B. Plale and K. Schwan, "Dynamic Querying of Streaming Data with the dQUOB System," *IEEE Trans. Parallel and Distributed Systems,* vol. 14, no. 4, 2003.

[14] *Terascale Supernova Initiative,* http://www.phy.ornl.gov/tsi/, 2005.

[15] S. Xiang, H.B. Lim, K.-L. Tan, and Y. Zhou, "Two-Tier Multiple Query Optimization for Sensor Networks," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS)*, 2007

[16] L. Luo, Q. Cao, C. Huang, T. Abdelzaher, J.A. Stankovic, and M. Ward, "Enviromic: Towards Cooperative Storage and Retrieval in Audio Sensor Networks," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS)*, 2007.

[17] *Akamai,* http://akamai.com/, 2008.

[18] *TIBCO,* http://www.tibco.com/, 2008.

[19] *IBM Unveils Enterprise Stream Processing System,* http://www.hpcwire.com/hpc/1623603.html, 2008.

[20] L. Chen, K. Reddy, and G. Agrawal, "GATES: A Grid-Based Middleware for Processing Distributed Data Streams," *Proc. 13th IEEE Int'l Symp. High Performance Distributed Computing (HPDC)*, 2004.

[21] Z. Cai, V. Kumar, and K. Schwan, "IQ-Paths: Self-Regulating Data Streams Across Network Overlays," *Proc. 15th IEEE Int'l Symp. High Performance Distributed Computing (HPDC '06),* citeseer.ist.psu.edu/745491.html, 2006.

[22] S. Ganguly, M. Garofalakis, R. Rastogi, and K. Sabnani, "Streaming Algorithms for Robust, Real-Time Detection of DDOs Attacks," *Proc. IEEE Int'l Conf. Distributed Computing Systems (ICDCS)*, 2007.

[23] X. Li et al., "Mind: A Distributed Multi-Dimensional Indexing System for Network Diagnosis," *Proc. IEEE INFOCOM*, 2006.

[24] D. Kossmann, "The State of the Art in Distributed Query Processing," *ACM Computing Surveys*, 2000.

[25] J. Moy, "OSPF Version 2," *request for comments 2328*, 1998.

[26] J. Beaver and M.A. Sharaf, "Location-Aware Routing for Data Aggregation for Sensor Networks," *Proc. Geo Sensor Networks Workshop*, 2003.

[27] E.W. Zegura, K.L. Calvert, and S. Bhattacharjee, "How to Model an Internetwork," *Proc. IEEE INFOCOM*, 1996.

[28] A.K. Jain and R.C. Dubes, *Algorithms for Clustering Data.* Prentice-Hall, Inc., 1988.

[29] G. Eisenhauer, F.E. Bustamante, and K. Schwan, "Event Services for High Performance Computing," *Proc. Ninth IEEE Int'l Symp. High Performance Distributed Computing (HPDC)*, 2000.

[30] R. Cox, F. Dabek, F. Kaashoek, J. Li, and R. Morris, "Practical, Distributed Network Coordinates," *Proc. Second Workshop Hot Topics in Networks (HotNets '03)*, ACM SIGCOMM, Nov. 2003.

[31] R. Williams et al., R^**: An Overview of the Architecture,* 2008.

[32] M. Stonebraker, "The Design and Implementation of Distributed INGRES," *The INGRES Papers: Anatomy of a Relational Database System,* 1986.

[33] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and Issues in Data Stream Systems," *Proc. ACM Symp. Principles of Database Systems (PODS),* 2002.

[34] S. Chandrasekaran et al., "TELEGRAPHCQ: Continuous Dataflow Processing for an Uncertain World," *Proc. First Biennial Conf. Innovative Data Systems Research (CIDR),* 2003.

[35] J. Chen, D.J. DeWitt, and J.F. Naughton, "Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries," *Proc. 18th Int'l Conf. Data Eng. (ICDE),* 2002.

[36] S. Babu, R. Motwani, K. Munagala, I. Nishizawa, and J. Widom, "Adaptive Ordering of Pipelined Stream Filters," *Proc. ACM SIGMOD,* 2004.

[37] R. Avnur and J.M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," *Proc. ACM SIGMOD,* 2000.

[38] U. Srivastava, K. Munagala, and J. Widom, "Operator Placement for In-Network Stream Query Processing," *Proc. ACM Symp. Principles of Database Systems (PODS),* 2005.

Sangeetha Seshadri received the BE degree in computer science and the MSc degree in mathematics from the Birla Institute of Technology and Science, Pilani, India, in 2002. She is currently working toward the PhD degree in the College of Computing, Georgia Institute of Technology, Atlanta. Previously, she worked as a senior applications engineer with Oracle India. Her research interests include storage systems and distributed middleware overlay systems, particularly, techniques and architectures for improving performance, scalability, and availability of such systems. She is a student member of the IEEE and the IEEE Computer Society.

Vibhore Kumar received the BTech degree in computer science from Banaras Hindu University, India, in 2002 and the MS degree in computer science and the PhD degree from the Georgia Institute of Technology in 2007 and 2008, respectively, where he worked on developing models and techniques for enabling scalable self-management of enterprise information systems. He is a research staff member at IBM T.J. Watson Research Center, Hawthorne, New York, where he works for the data-intensive systems and analytics group. His research interests include systems and techniques for scalable data analysis, enterprise information systems, and autonomic computing. He is a member of the IEEE.

Brian Cooper received the PhD degree from Stanford University. He is a research scientist at Yahoo! Research, Santa Clara, California. Before that, he was an assistant professor at Georgia Tech. His research interests include building distributed systems, and in particular, distributed systems that do database-style management and processing of data. At Yahoo!, he works on building very large distributed data storage and processing systems. Previously, he has worked on self-adaptive peer-to-peer systems, distributed streaming event processing, reliable distributed archival data storage, and XML indexing.

Ling Liu is a professor in the College of Computing, Georgia Institute of Technology, Atlanta. There, she directs the research programs in Distributed Data Intensive Systems Lab (DiSL), examining various aspects of data-intensive systems, ranging from network computing, mobile computing, event stream processing, to Internet data management, storage systems, and service-oriented architectures, with the focus on performance, security, privacy, and energy efficiency in building large-scale Internet systems and services. She has published more than 200 international journal and conference articles in the areas of Internet Computing systems, Internet data management, distributed systems, and information security. Her research group has produced a number of open source software systems, among which the most popular ones are WebCQ, XWRAPE-lite, and PeerCrawl. She is currently on the editorial board of several international journals, including the *IEEE Transactions on Service Computing* (TSC), the *IEEE Transactions on Knowledge and Data Engineering*, *International Journal of Peer-to-Peer Networking and Applications* (Springer), and *International Journal of Web Services Research, Wireless Network* (Springer). She is a recipient of the best paper award of ICDCS 2003, the best paper award of WWW 2004, the 2005 Pat Goldberg Memorial Best Paper Award, and the best data engineering paper award of the International Conference on Software Engineering and Data Engineering 2008. She is a recipient of the IBM faculty award in 2003, 2006, and 2008. Her research is primarily sponsored by the US National Science Foundation (NSF), AFOSR, and IBM. She is a senior member of the IEEE.