Optimizing Multiple Distributed Stream Queries Using Hierarchical Network Partitions

Sangeetha Seshadri, Vibhore Kumar, Brian F. Cooper and Ling Liu
College of Computing, Georgia Institute of Technology
{sangeeta,vibhore,cooperb,lingliu}@cc.gatech.edu

Abstract

We consider the problem of query optimization in distributed data stream systems where multiple continuous queries may be executing simultaneously. In order to achieve the best performance, query planning (such as join ordering) must be considered in conjunction with deployment planning (e.g., assigning operators to physical nodes with optimal ordering). However, such a combination involves not only a large number of network nodes but also many query operators, resulting in an extremely large search space for optimal solutions. Our paper aims at addressing this problem by utilizing hierarchical network partitions. We propose two algorithms - Top-Down *and* Bottom-Up *which utilize hierarchical network partitions to provide scalable query optimization. Formal analysis is presented to establish the bounds on the search-space and to show the sub-optimality of our algorithms. Through simulations and experiments using a prototype deployed on Emulab [1] we demonstrate the effectiveness of our algorithms.*

1. Introduction

In many data stream systems, data is produced at multiple, geographically distributed sources. Examples include enterprise supply chain applications, scientific collaborations, and distributed network monitoring. 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 processing load at the central server. Instead, performing distributed processing of stream queries using techniques such as in-network processing [23, 15, 4] and filtering at the source [18] minimizes the communication overhead on the system and helps spread processing load, significantly improving performance. Then, we can think of a continual query as being "deployed" in the network, with data streams flowing between operators assigned to distributed physical nodes.

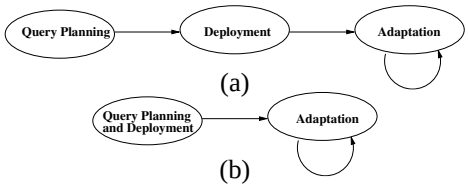

Figure 1. Approaches: (a) Plan, then deploy, and (b) Our approach.

The conventional approach used in distributed data stream systems [3, 19] is to construct a query plan (e.g., the stream query processing should follow a specified join ordering) at compile time, and deploy this plan at runtime to improve performance. This approach is shown in Figure 1(a). One 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 there exists an alternate join order that would be more efficient. In addition, the pre-defined join order may prevent us from reusing the results of an already deployed join from another query at runtime.

In this paper we argue that one effective way to address this problem is to consider the query plan and the deployment simultaneously (Figure 1(b)) and propose techniques for performing query planning in conjunction with deployment planning. One of the key ideas is to use hierarchical network partitions to scalably exploit various opportunities for operator level reuse in the processing of multiple stream queries.

Figure 2 compares this approach with two "Plan, then deploy" approaches with operator reuse enabled - an optimal deployment through exhaustive search and the Relaxation algorithm [19]. The figure shows that significant ($> 50\%$) cost savings can be achieved by combining the planning and deployment phases.

Acknowledgment: This work is partially supported by grants from NSF CSR, NSF IIS, NSF CyberTrust, a grant from AFOSR, an IBM Faculty Award, an IBM SUR grant and a HP equipment grant.

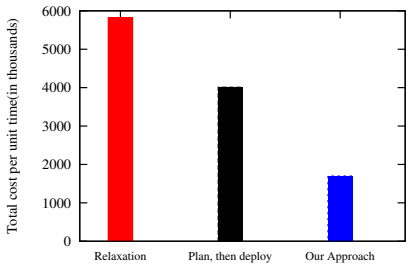

Figure 2. Comparison with typical approaches: The graph shows the total communication cost incurred by 100 queries over 5 stream sources each, on a 64-node network. The *cost of a deployment* is the total data transferred along each link times the link cost. The network topology was generated using the standard topology generator GT-ITM. Our approach that considers query plans and deployments simultaneously reduces the cost by more than 50% as it was able to exploit optimization opportunities such as operator reuse even during planning. The Relaxation algorithm, a "plan, then deploy" heuristic to determine an efficient operator placement was implemented using a 3-dimensional cost space.

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 Figure 2. With a network of 64 nodes, combining query plans and plan deployments simultaneously required us to examine 2.88×10^9 plans for a single query over 5 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. We propose to use hierarchical network partitions as a heuristic, aiming at trading some optimality for a much smaller search space. In particular, 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 two 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. Our algorithms are implemented over IFLOW [13], a distributed data stream system. The IFLOW system facilitates adaptivity by monitoring network and load conditions and re-triggering our optimization algorithms when conditions dictate redeployment of the query.

We present analysis and experiments that show the suboptimality of our algorithms is bounded. At the same time, our algorithms can reduce the search space by orders of magnitude compared to an exhaustive search, even using dynamic programming. For example, experimentally, the Top-Down algorithm was able to achieve, on average, solutions that were sub-optimal by only 10% while considering less than 1% of the search space.

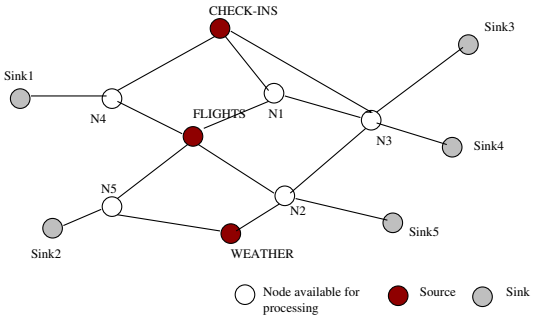

Figure 3. An example network N

The remainder of this paper is organized as follows. In Section 2 we present our algorithms and rigorously analyze their effectiveness. An experimental evaluation of the proposed solutions is presented in Section 3. We discuss related work in Section 4 and finally conclude in Section 5 with a discussion of possible future directions. We next present an application scenario that motivates our research.

1.1 Motivating Application Scenario

Our research is primarily motivated by enterprise-level data streaming systems such as the Operational Information System (OIS) [17] employed by our collaborators, Delta Air Lines. 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 Air Lines 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 a small example network and sample queries to illustrate the optimizations opportunities that may be available in such a setup.

Let us assume Delta's OIS to be operating over the small network N shown in Figure 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 physi-

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

1. **Network-aware join ordering:** Based purely on the size of intermediate results, we may normally choose the join order (FLIGHTS⋈WEATHER)⋈CHECK-INS. Then we would deploy the join FLIGHTS⋈WEATHER at node $N2$, and the join with stream CHECK-INS at node $N3$. However, node $N2$ may be overloaded, or the link FLIGHTS$\rightarrow N2$ may be congested. In this case, the network conditions dictate that a more efficient join ordering is (FLIGHTS⋈CHECK-INS)⋈WEATHER, with FLIGHTS⋈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 FLIGHTS.STATUS, CHECK-INS.STATUS
    FROM FLIGHTS, CHECK-INS
    WHERE FLIGHTS.DEPARTING='ATLANTA'
    AND FLIGHTS.NUM = CHECK-INS.FLNUM
    AND FLIGHTS.DP-TIME - CURRENT_TIME < 12:00:00
```

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

2. **Operator Reuse:** Although the optimal operator ordering in terms of the size of intermediate results for query Q1 may be (FLIGHTS⋈WEATHER)⋈CHECK-INS, in order to reuse the already deployed operator FLIGHTS⋈CHECK-INS, we must pick the alternate join ordering (FLIGHTS⋈CHECK-INS)⋈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⋈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. Besides enterprise systems, techniques for optimization of stream queries are critical to a variety of applications ranging from network monitoring [14] to scientific collaborations [2].

2. Query Optimization Algorithms

Our algorithms are implemented within the IFLOW system [13], a toolkit that supports distributed deployment of continuous queries over data streams. In this paper we focus on the optimization algorithms implemented specifically for SQL-like queries in the IFLOW system. Self-adaptivity is incorporated into the system through the Middleware Layer [13] which re-triggers the query optimization algorithm when the changes in network, load or data conditions demand recomputing of query plans and deployments.

Our focus is on optimizing 'select-project-join' queries. We leave queries involving aggregations and unions to future work. We assumed stream joins are performed using standard techniques (e.g. doubly-pipelined operators and windows if necessary). Our goal is to find a combination of a query plan (e.g. join order) and deployment (e.g. assignment of query operators to physical nodes) in order to optimize some application-provided performance function. This function might be a low level function, like response time or communication cost, or some more complex utility function. Our calculation of the performance metric takes into account the estimated selectivities of the query operators, measured online or using gathered statistics over the stream sources.

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 pre-computed statistics. Lemma 1 shows the size of the exhaustive search space for the query optimization problem in distributed data stream systems. Note that, a solution refers to any feasible query plan and deployment combination.

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:*

$$\mathcal{O}_{exhaustive} = \left(\frac{K(K-1)(K+1)}{6}\right) \times (N)^{(K-1)}$$

Sketch. The equation is arrived at by enumerating all possible join orders (including bushy joins) multiplied with the number of possible placements. □

Due to space restrictions the full proof of this and other theorems in this paper are presented in [20].

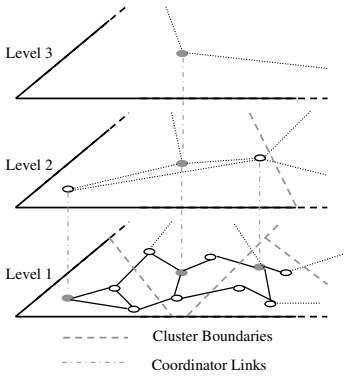

Figure 4. Hierarchical network clusters

As shown in the 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 (DP) 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, DP 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 [12].

2.1 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 sub-optimality by adjusting the max_{cs} parameter, which is the maximum number of nodes allowed per network partition. This tradeoff is complex, and is analyzed in detail in our discussion of the Top-Down (Section 2.2) and Bottom-Up (Section 2.3) algorithms.

2.1.1 Hierarchical Network Clusters

We organize physical network nodes into a virtual clustering hierarchy, by clustering nodes based on our optimization criteria. For example, if the metric is response-time, we cluster based on inter-node delays. If the metric is communication costs, we cluster 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 ***inter-node/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 the inter-node 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*. Nodes in *Level 2* are again clustered according to average inter-node 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 Figure 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 intra-cluster 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: $c_{act}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + \sum_{i=1}^{i<l} 2d_i$*

Sketch. At a particular level l the cost of traversal between nodes v_{nj} and v_{nk} is given by the inter-node 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 $l-1$. Inter-node traversal costs at this level are bounded by the value d_{l-1}. Thus the inter-node traversal costs between nodes v_{nj} and v_{nk} at level $l-1$ is given by $c^{l-1}_{est}(v_{nj}, v_{nk}) \leq c^l_{est}(v_{nj}, v_{nk}) + 2d_{l-1}$. This process continues down the hierarchy. At level 1, the estimated cost is the same as the actual traversal cost. Therefore the estimated traversal cost at level l is at most $\sum_{i=1}^{i<l} 2d_i$ less than the actual cost. □

The hierarchical organization is created and maintained using the following algorithm. 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. This 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 [16], for data aggregation in sensor networks [7] and other related applications. 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 and operator nodes can be handled by maintaining active back-ups of those nodes within

each cluster. However, the issue of fault tolerance is beyond the scope of this paper. Note that, given a network, multiple virtual clustering hierarchies can be created simultaneously with different values of the max_{cs} parameter.

2.1.2 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 sub-query. 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.

2.2 The Top-Down Algorithm

The *Top-Down* algorithm bounds sub-optimality 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.

The Top-Down algorithm can be easily extended to perform multi-query optimization by constructing a consolidated query at the top-most level of the hierarchy and then applying the algorithm to this consolidated query.

2.2.1 Bounding Search Space

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:

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

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 4 streams on a network with 1000 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*

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

Sketch. Assuming that α_i sources are considered at each level i of the hierarchy, we enumerate the total number of possible join orders, including bushy joins. 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. The proof then follows from Lemma 1. □

2.2.2 Sub-Optimality in the Top-Down Algorithm

The Top-Down algorithm works by propagating a query down the network hierarchy, described in Section 2.1.1. Given a query $\mathcal{Q}$, at each level a coordinator chooses a query plan and a deployment with the least cost for the sub-query 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 top most level of the hierarchy. Therefore the Top-Down algorithm is most sub-optimal when all the edges of the query plan are deployed at the top-most level. The following theorem establishes the relationship between sub-optimality of computed deployments and the hierarchical structure's properties - the number of levels and the cluster density.

Theorem 3. *A query $\mathcal{Q}$ deployed using the Top-Down algorithm over a network N is no more than $\sum_{e_k \in E^Q}(\sum_{i=1}^{i<h} 2d_i) \times s_k$ sub-optimal 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 intra-cluster traversal cost at level i and s_k is the stream rate for the k^{th} edge e_k.*

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

2.3 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 sub-optimality 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 $\mathcal{Q}$ over base stream sources arrives at a sink at *Level 1*, the sink informs its coordinator at *Level 2*. The coordinator rewrites the query $\mathcal{Q}$ as $\mathcal{Q}'$ with respect to two views - $V^{\mathcal{Q}}_{local}$ and $V^{\mathcal{Q}}_{remote}$ where $V^{\mathcal{Q}}_{local}$ is composed of base and derived sources available locally within the cluster and $V^{\mathcal{Q}}_{remote}$ is composed of base sources not available locally. The coordinator deploys $V^{\mathcal{Q}}_{local}$ within the current cluster, and then advertises $V^{\mathcal{Q}}_{local}$ 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 $\mathcal{Q}'$ from its next level coordinator.

This process continues up the hierarchy, with the query $\mathcal{Q}'$ progressively decomposed into locally available views and remote views and the re-written 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 $V^{\mathcal{Q}}_{local}$. The search space is limited to a single network partition and the local sub-query.

Operator reuse is taken into consideration by coordinators by taking into account all possible constructions of $V^{\mathcal{Q}}_{local}$ 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.

The Bottom-Up algorithm can also be extended to perform multi-query optimization. When a coordinator receives multiple requests from different sinks in its underlying cluster, it composes consolidated queries. The coordinator, thereafter applies the Bottom-Up algorithm to these consolidated queries.

2.3.1 Bounding Search Space

Recall our definition of β in Section 2.2.1. We now show in Theorem 4 that β also represents the 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 3 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,*

$$\mathcal{O}_{bottom-up} \leq \beta \mathcal{O}_{exhaustive}$$

Sketch. As in the case of Theorem 2 we compute this search space by considering all possible query trees and all possible placements of operators within a single cluster at each level. The proof then follows from Lemma 1. □

2.3.2 Sub-Optimality 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 sub-optimal. 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 sub-optimality 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. Therefore, it is possible to identify these scenarios a priori through static analysis of stream rates and selectivities and use the Top-Down algorithm in those cases.

We show in [20], that the sub-optimality of the plan chosen by Bottom-Up is bounded with respect to the most optimal deployment of the same join-ordering. This proves that Bottom-Up can offer better bounds than a random placement of the same query tree. Thus, Bottom-Up is ideal in situations where the network placement of operators is a more dominant factor than join-ordering and when quick deployments are needed, for possibly short-lived queries.

3. Experiments

We present both simulation based experiments and experiments conducted on Emulab [1] using IFLOW [13]. We show that in the average case Top-Down is only 10% sub-optimal, while Bottom-Up is 34% sub-optimal. However, the deployment time of the Bottom-Up is 70% less than that of the Top-Down.

Our simulation experiments were conducted over transit-stub topology networks generated using the standard tool, the GT-ITM internetwork topology generator [24]. Most experiments were conducted using a 128 node network, with a standard Internet-style topology: 1 transit (e.g. "backbone") domain of 4 nodes, and 4 "stub" domains (each of 8 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. The *cost of a deployment* is the total data transferred along each link times the link cost.

We used a synthetic workload so that we could experiment with a large variety of stream rates, query complexities, and operator selectivities. Our 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. In our experiments we use a cost formulation that tries to minimize the communication cost incurred per unit time by the deployed query plan. Therefore, as described in Section 2.1.1 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 [11] algorithm in order to create the clustering hierarchy.

3.1 Tuning Cluster Size: Sub-Optimality - Search Space Tradeoff

In this section we demonstrate how the max_{cs} parameter can be used to tune the tradeoff between the sub-optimality of the heuristic and minimizing the search space. The experiments were conducted on the 128 node topology described in Section 3, with 10 source streams. We averaged our results over 10 workloads generated using our random workload generator, each with a different set of placements for sources and sinks. Each workload consisted of 200 queries with 2-5 joins per query. Due to space constraints, additional experiments that study the variation of search space with max_{cs} are presented in [20].

Figure 5 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% 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 sub-optimality, 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 (using Theorem 4) that is acceptable.

Next, Figure 6 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 top-most 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 and each level adds more inaccuracy to the approximation. Therefore, the algorithm makes poorer planning decisions. To limit sub-optimality, we need a reasonably large max_{cs}. To bound search space, we need a small max_{cs}. Hence, a useful guideline for the Top-Down

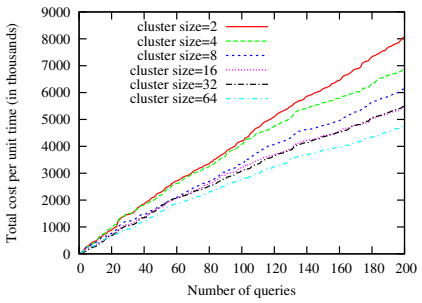

Figure 5. Bottom-Up: Cost

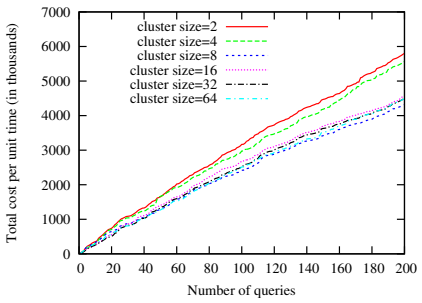

Figure 6. Top-Down: Cost

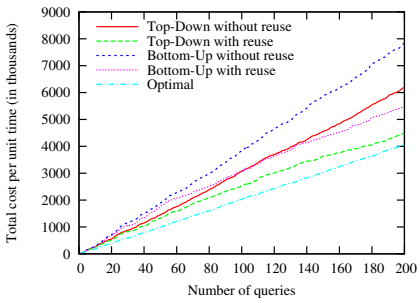

Figure 7. Sub-optimality

algorithm is:

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

3.2 Sub-optimality and Effect of Reuse

We now examine the effect of operator reuse in our algorithms. Figure 7 shows the cumulative cost of the optimal deployment computed using dynamic programming (DP) and the two algorithms both with and without operator reuse. The max_{cs} parameter was set to 32. 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 to provide an apples-to-apples comparison. Operator reuse was implemented through stream-advertisements. The communication cost of advertisements was negligible compared to the data streams themselves. The figure shows that Bottom-Up benefits by nearly a 30% decrease in cost per unit time through operator reuse, while Top-Down achieves cost saving of 27% per unit time through operator reuse.

Figure 7 also allows us to compare the deployed costs of the two algorithms with the optimal solution computed using DP. As can be seen, Top-Down with reuse performs nearly 19% better than Bottom-Up with reuse. This is because Bottom-Up may choose a sub-optimal plan since it does not consider an ordering of all operators at any level, unlike Top-Down. When compared to the optimal, Bottom-Up with reuse, performs sub-optimally by 34% and Top-Down by only 10%. This shows that the sub-optimality of Top-Down is minimal. The performance of Bottom-Up may be "good-enough" for short-lived queries which primarily require fast time-to-deployments.

3.3 Comparison with existing approaches

In this experiment we compare our algorithms with existing approaches - the Relaxation algorithm [19] and *In-network* [4], a network-aware query processing algorithm. Both [4, 19] are phased deployment approaches that first plan and then deploy (see Figure 1(a)).

Figure 8 shows the cumulative cost of deployments computed using the Top-Down, Bottom-Up algorithms as compared with the Relaxation and In-network algorithms. The graph also shows the costs of optimal deployments computed using an exhaustive search. Operator reuse was taken into consideration for all algorithms. We used a 3-dimensional cost space [19] for the Relaxation algorithm and considered a virtual hierarchy with max_{cs} 32 for the Top-Down, Bottom-Up algorithms. In order to correspond with this max_{cs} value, we divided the network into 5 *zones* for the In-network algorithm.

The graph shows that, when compared to the In-network algorithm, the Top-Down algorithm can provide nearly 40% additional cost savings per unit time, and the Bottom-Up algorithm, savings of 27%. Also, note that, the search space of this algorithm was nearly 70% that of the Top-Down algorithm and 200% that of the Bottom-Up algorithm.

When compared to Relaxation, Top-Down reduces cost by nearly 59% and Bottom-Up by nearly 49%. 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 algorithm. In our experiment, the 3-dimensional cost space was calculated using 4000 iterations and we used as many iterations for the Relaxation algorithm. The running time of the Relaxation algorithm was comparable to that of the Bottom-Up algorithm.

3.4 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 4 streams. We measured the average number of deployments considered over 4 different transit-stub topologies of different sizes generated using GT-ITM. Again, sinks were placed at random nodes in the network. Figure 9 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

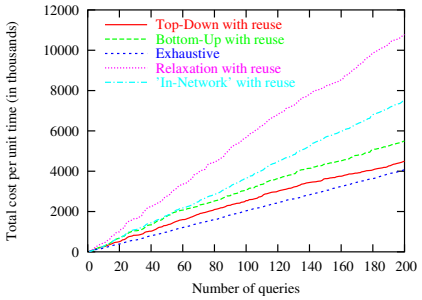

Figure 8. Comparison with other approaches

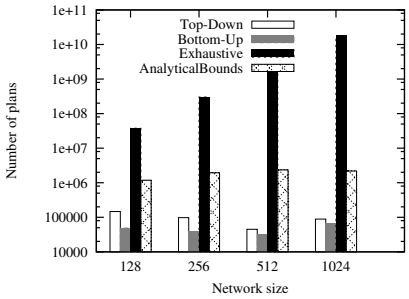

Figure 9. Scalability with Network Size

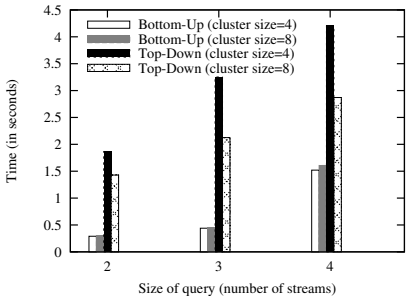

Figure 10. Query deployment time

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%. We also see that the search space per query with Bottom-Up is nearly 45% 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 sample networks. For example, clustering using max_{cs} 32 resulted in an average *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.

3.5 Prototype Experiments

The next set of experiments was conducted on Emulab using IFLOW [13]. IFLOW supports hierarchies and advertisements as described earlier. The testbed on Emulab consisted of 32 nodes (Intel XEON, 2.8 GHz, 512MB RAM, RedHat Linux 9), organized into a topology that was again generated with GT-ITM. Links were 100Mbps and the inter-node delays were set between 1msec and 6msec. The workload for the following experiments consisted of 25 queries over 8 stream sources and sinks distributed across the system. The number of joins per query varied from 1 to 3.

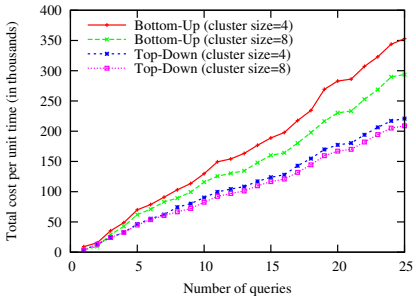

Figure 11. Cumulative deployed cost

3.5.1 Deployment Time and Cost

The first experiment conducted on Emulab validates our claim about the stricter search space bounds offered by the Bottom-Up algorithm. Figure 10 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% 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.

In this experiment, we studied the cost of deployments with the Bottom-Up and Top-Down algorithms for different values of max_{cs}. Figure 11 shows the cumulative cost incurred per unit time over 25 queries. We observe that the Top-Down algorithm offers a lower deployed cost than the Bottom-Up algorithm. This is in alignment with our simulation results. Since the Top-Down algorithm considers all operator orderings at the top-most level this algorithm leads to the selection of a better execution plan.

4. Related Work

Distributed query optimization has received a great deal of attention from researchers since the 1980s [12]. However, 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 like these systems.

A number of stream processing systems, both centralized and distributed, such as STREAM [6], Borealis [3], TelegraphCQ [8, 5] and NiagaraCQ [9, 22], have been developed to process queries over continuous streams of data. Our work is in the context of distributed stream processing systems. The use of in-network query processing [15, 23] in such systems to decide operator placement when the query tree is already known is described in [4, 19]. The network-aware algorithms in [4] firstly perform phased deployments which we have shown to be sub-optimal. Secondly, 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 [19] 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 sub-optimality, both due to lost reuse opportunities and the subsequent approximate placement decisions. Optimal placement of filter operators is discussed in [21]. However, the selection or placement of joins is not addressed. Note that although our problem bears some resemblance to the task scheduling problem [10], our algorithms are designed to deal with distribution at a much larger scale.

5. Conclusion & Future Work

We described the query-optimization problem in distributed data-stream systems and demonstrated that selection of an optimal execution plan in such systems must consider operator ordering, network placement and operator reuse. We presented a query optimization infrastructure that has two key components: a virtual hierarchical network structure and stream advertisements that enable operator reuse. We described algorithms *Top-Down* and *Bottom-Up* that find efficient execution plans while examining a very small search space. Experimental and analytical results showed that both algorithms offer costs that are comparable to optimal while exploring much fewer plans. In on-going work we are exploring run-time query plan migrations, and other optimization opportunities achievable through query containment.

References

[1] Emulab network testbed. *http://www.emulab.net/*.

[2] Terascale supernova initiative. `http://www.phy.ornl.gov/tsi/,2005.`

[3] D. J. Abadi et al. The Design of the Borealis Stream Processing Engine. In *CIDR*, 2005.

[4] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *VLDB*, 2004.

[5] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *SIGMOD*, 2000.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom. Models and issues in data stream systems. In *PODS*, 2002.

[7] J. Beaver and M. A. Sharaf. Location-aware routing for data aggregation for sensor networks. In *Geo Sensor Networks Workshop*, 2003.

[8] S. Chandrasekaran et al. TELEGRAPHCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[9] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang. NIAGARACQ: A scalable continuous query system for internet databases. In *SIGMOD*, 2000.

[10] R. Chow and T. Johnson. *Distributed Operating Systems and Algorithms*. Addison-Wesley Longman, 1997.

[11] A. K. Jain and R. C. Dubes. *Algorithms for clustering data*. Prentice-Hall, Inc., Upper Saddle River, NJ, USA, 1988.

[12] D. Kossmann. The state of the art in distributed query processing. *ACM Comput. Surv.*, 2000.

[13] V. Kumar et al. Implementing diverse messaging models with self-managing properties using IFLOW. In *IEEE ICAC*, 2006.

[14] X. Li et al. Mind: A distributed multi-dimensional indexing system for network diagnosis. In *IEEE Infocom, 2006.*

[15] S. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong. TAG: a tiny aggregation service for ad-hoc sensor networks. In *OSDI*, 2002.

[16] J. Moy. OSPF version 2, request for comments 2328. 1998.

[17] V. Oleson et al. Operational information systems - an example from the airline industry. In *WIESS, 2000.*

[18] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *SIGMOD*, 2003.

[19] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.

[20] S. Seshadri et al. Optimizing distributed stream queries using hierarchical network partitions(extended version). *http://www.cc.gatech.edu/~sangeeta/SMQ.pdf*.

[21] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *PODS*, 2005.

[22] S. D. Viglas and J. F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD*, 2002.

[23] Y. Yao and J. Gehrke. The cougar approach to in-network query processing in sensor networks. *SIGMOD Rec.*, 2002.

[24] E. W. Zegura, K. L. Calvert, and S. Bhattacharjee. How to model an internetwork. In *Infocom*, 1996.