Distributed Operator Placement and Data Caching in Large-Scale Sensor Networks

Lei Ying*, **Zhen Liu**†, **Don Towsley**‡, **Cathy H. Xia**†
*Department of Electrical and Computer Engineering, UIUC
Email: lying@uiuc.edu
†IBM T.J. Watson Research Center, Hawthorne, NY 10532
Email: {zhenl,cathyx}@us.ibm.com
‡Department of Computer Science, UMASS-Amherst
Email: towsley@cs.umass.edu

***Abstract*—Recent advances in computer technology and wireless communications have enabled the emergence of stream-based sensor networks. In such sensor networks, real-time data are generated by a large number of distributed sources. Queries are made that may require sophisticated processing and filtering of the data. A query is represented by a query graph. In order to reduce the data transmission and to better utilize resources, it is desirable to place operators of the query graph inside the network, and thus to perform in-network processing. Moreover, given that various queries occur with different frequencies and that only a subset of sensor data may actually be queried, caching intermediate data objects inside the network can help improve query efficiency. In this paper, we consider the problem of placing both operators and intermediate data objects inside the network for a set of queries so as to minimize the total cost of storage, computation, and data transmission. We propose distributed algorithms that achieve optimal solutions for tree-structured query graph topologies and general network topologies. The algorithms converge in $L_{\max}(H_Q+1)$ iterations, where $L_{\max}$ is the order of the diameter of the sensor network, and H_Q represents the depth of the query graph, defined as the maximum number of operations needed for a raw data to become a final data. For a regular grid network and complete binary tree query graph, the complexity is $O(\sqrt{N}\log_2 M)$, where N is the number of nodes in the sensor network and M is the number of data objects in a query graph. The most attractive features of these algorithms are that they require only information exchanges between neighbors, can be executed asynchronously, are adaptive to cost change and topology change, and are resilient to node or link failures.**

I. Introduction

Recent advances in computer technology and wireless communications have enabled the emergence of stream-based sensor networks. Broad applications include network traffic monitoring, real-time financial data analysis, environmental sensing, large-scale reconnaissance, surveillance, etc. In these applications, real-time data are generated by a large number of distributed sources, such as sensors, and must be processed, filtered, interpreted or aggregated in order to provide useful services to users. The sensors, together with many other shared resources, such as Internet hosts and edge servers, are organized into a network that collectively provides a rich set of query processing services. Resource-efficient data management is a key challenge in such stream-based sensor networks.

Two different approaches are commonly used for managing and processing data in such networks. The first approach involves setting up a centralized data warehouse (e.g. fusion center) to which sensors push potentially useful sensor data for storage and processing [1]. Users then make a variety of sophisticated queries on the data stored at this central database. Clearly, this approach does not efficiently use resources because all data must be transmitted to the central warehouse whether or not they are of interest. Moreover, this may require a large investment in processing resources at the warehouse while at the same time processing resources within the network may go idle. The second approach involves pushing queries all the way to the remote sensors, see e.g. [2]. Such querying of remote sensor nodes is generally more bandwidth efficient. However, it is greatly limited by the low capability, low availability, and low reliability of the edge devices such as sensors.

A third approach that pushes query processing into the network is thus desirable in order to reduce data transmission and better utilize available shared resources in the network. Furthermore, given the fact that various queries are generated at different rates and only a subset of sensor data may be actually queried, caching some intermediate data objects inside the network may help increase query efficiency. In this paper we address the question of where to place specific query operators and when and where to store intermediate data objects. Intuitively, one would like to place the operators as close as possible to the edge devices (sensors) so as to reduce transmission costs. However, devices close to the edge are likely to have limited processing and storage capabilities. Thus they may not be capable of handling sophisticated queries. Efficient placement algorithms are therefore needed to achieve the minimum overall cost in computation, storage and communication.

In-network query processing has received increasing attention in past several years. Previous work on in-network query processing has mostly focused on the operator placement problem [3], [4], [5], [6], [7], [8], [9]. Most studies assume that queries are generated at the same rate as the data and are applied to all data. They do not exploit the fact that some queries may be generated at such low rates that they need only

be applied to a fraction of the data generated.

In this paper, besides considering the placement of various querying operators, we also account for the randomness of the querying process. We assume that different queries occur at different frequencies and that only a subset of sensor data may actually be queried. Each query is associated with a given query graph, consisting of operators and data objects, that describes how a response to the query is obtained logically. We explore the advantages of caching intermediate data objects inside the network relative to conventional push and pull mechanisms. We integrate the operator placement and caching problem together by considering node assignment for both operators and intermediate data objects that yields the minimum total communication, computation, and storage cost.

We consider large-scale sensor networks whose conditions can change dynamically. Thus centralized solutions become expensive as they require global knowledge of the network. We propose distributed algorithms and show that they achieve optimal solutions for various query graph topologies and network topologies. Our algorithms are similar in spirit to the Bellman-Ford algorithm [10]. We assume communication is local in the sense that a node communicates only with its neighbors. The algorithm proceeds iteratively where each node updates its states according to the information received from its neighbors, and then sends updated information to its neighbors.

A. Contributions

The contributions of this paper are as follow:

- A formal definition of the operator placement and intermediate data caching problem with the goal of minimizing a unified cost metric incorporating computation, communication and storage costs (Section III).
- Distributed algorithms that solve the above problem for tree-structured query graphs and general sensor network topologies. The algorithms converge in $L_{\max}(H_Q+1)$ iterations, where $L_{\max}$ is on the order of the diameter of the sensor network, and H_Q represents the depth of the query graph. For a regular grid network of N nodes and a complete binary tree query graph of M data objects, the convergence time is $O(\sqrt{N}\log_2 M)$. The algorithms can be executed in a synchronous or asynchronous way, where optimality is guaranteed in both cases. The algorithms are also adaptive to cost and topology changes and resilient to node or link failures. This is significant improvement over the existing methods. (Section IV).
- Evaluation of the proposed algorithms on grid networks and tree-structured query graphs that show a significant reduction in the cost of query processing over that of some other naive methods. (Section V).

II. The Model

In this section, we describe the system and corresponding model.

A. Distributed Environment

We consider an environment where data streams are generated at multiple data sources. The target applications require sophisticated processing (e.g. fusion, filtering, correlation, and other analysis) of these data streams. The system consists of a large number of nodes (sensors and edge servers), capable of executing multiple stream-oriented operators. These nodes, possibly geographically dispersed, are organized into a network and collectively provide a rich set of query processing services for multiple concurrent stream-based applications.

B. Sensor Network Model

A sensor network is represented by a directed graph $(\mathcal{V},\mathcal{E})$. The edges of the network correspond to communication links, which are either wired or wireless. We assume that if $(i,j)\in\mathcal{E}$, then $(j,i)\in\mathcal{E}$. That is, communication is feasible in both directions for every pair of connected nodes. Also assume $|\mathcal{V}|=N$, and there are three types of nodes in the network: sensor nodes, relay nodes, and a fusion center.

All nodes have varying computation and data transmission capabilities. They may also have storage space to cache data. We assume each node only has local knowledge about its neighboring nodes. Let A_i denote the set of (adjacent) neighbors of node i such that for each $j\in A_i$, either $(i,j)\in\mathcal{E}$ or $(j,i)\in\mathcal{E}$. Furthermore let $d_{\max}=\max_i|A_i|$, which is the maximum number of neighbors of any node (maximum node degree).

C. Discrete Stream-Based Data Generation

Sensors monitor and collect data about the environment by sampling it periodically. We assume that time is discretized, and that each sensor takes one sample during each time slot. Thus one snapshot of the whole environment is taken by the network during each slot. Note that strict synchronization is not required at the various nodes in the network. Each node simply needs to index the snapshots in sequence. For now, we focus on a single snapshot. We will propose a distributed algorithm that also works well asynchronously.

D. Query Graph

We assume that the network supports a set of queries. Each query Q is represented by a query graph $\mathcal{Q}$, consisting of operators and data objects. Similar to previous work [5], [4], [11], [12], [7], [6], we assume that the query graphs are given. Query graphs can be built either directly by users, or derived by a query planner. In general, query graphs are logical graphs depicting the interaction between various operators and data objects. They are typically decoupled from the physical network topology and focus on the semantics of the query and workflow relations. Such decoupling is also necessary given commonly varying network conditions. The construction of

efficient query graphs (query planning) is an important but difficult problem. It is out of the scope of this paper.

We assume that each query graph is a tree $\mathcal{Q} = (V_o, V_d, E_Q)$ that describes how a response to the query is obtained logically. Here, V_o is a set of operators, and V_d is a set of data objects.

For simplicity, we assume that there is exactly one data object generated by each operator. Thus the operator is assigned the same label as the generated data. Each operator may require multiple data objects as input, and each data object may be requested as input by one operator. Consider for example the query graph given in Figure 1. Operator 7 can generate data 7 and requires data 1 and 3 as inputs.

The data objects are divided into three classes:

Sensor data (raw data): Data generated by the sensor nodes, e.g., data $1, 2, 3, 4$ in Fig. 1. They are pinned to corresponding sensors.

Final data: Answers to the query, e.g., data 8 in Fig. 1. The final data need to be delivered to the fusion center.

Intermediate data: Data that are neither sensor data nor final data, for example data 6 or 7 in Fig. 1.

We assume $|V_d| = M$ and H_Q is the depth of the query graph, which is defined as the maximum number of operations needed for a raw data to become a final data, for example, $H_Q = 2$ as in Figure 1. For any $k \in V_d$, we denote by Δ_k the set of data objects in V_d that are needed to generate data object k, i.e. the children of data k. For example, $\Delta_7 = \{1, 3\}$ in Fig. 1. Denote by D_k the size (in number of bits) of data object k.

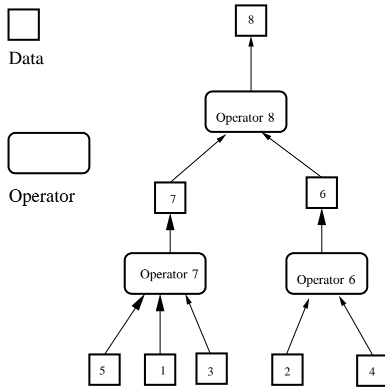

Fig. 1. Tree Structured Query Graph

Recall that one snapshot of the environment is created during each time slot. In order to exploit the fact that queries may be generated at a lower rate and that they are applied only to a fraction of the data generated, we assume that for a given query Q with query graph $\mathcal{Q}$, each snapshot will be evaluated by $\mathcal{Q}$ with probability q. We shall refer to q as the query frequency, which is a number in between zero and one. If $q = 1$, then the query is evaluated at every snapshot. If q is close to 0, each snapshot is evaluated with a small probability.

Throughout the paper, we use index k to denote any data object in the query graph $\mathcal{Q}$, and indices i, j to denote nodes in the sensor network $\mathcal{N}$.

III. Minimum Cost Query Processing

In this section, we formally define the operator placement and data caching problem.

A. Query Processing Scheme

A query processing scheme (Π, R, S) needs to solve the following three problems.

(1) **Operator Placement** (Π)**:** Where to place the operators? Let Π denote the operator placement matrix such that $\Pi(k, i) = 1$ if data k is generated at node i. That is, the operator generating data k is placed at node i. We denote by $i^{\Pi}(k)$ the node that generates data k under scheme Π. We assume each operator can be placed at only one node, thus $\Pi(k, j) = 0$ for all $j \neq i^{\Pi}(k)$. Although redundant placements may be beneficial, especially when data k is requested by multiple upstream operators, we do not consider this in the paper.

(2) **Routing** (R)**:** Given an operator placement scheme Π, how to route the data from the node generating the data to the node that requires the data as input? For example, all data in Δ_k (children of data k) need to be transmitted to node $i^{\Pi}(k)$ to generate data k.

(3) **Data Caching/Storage**(S)**:** Where to cache the data? Here, S is the caching matrix, $S(k, i) = 1$ if data k is cached at node i.

B. Cost Metrics

Suppose a query process scheme (Π, R, S) is given for query Q with query graph $\mathcal{Q}$ on sensor network $\mathcal{N}$. Let $f_{\text{cost}}(\Pi, R, S; \mathcal{N}, \mathcal{Q})$ denote the cost of this scheme; it accounts for three different costs.

(1) **Processing cost** $f_p(\Pi, R, S; \mathcal{N}, \mathcal{Q})$: The cost of processing and aggregation. Let $r_p(k, i)$ denote the cost to generate data k at node i. For sensor data k, $r_p(k, i) = 0$ if node i is the source for data k and $r_p(k, i) = \infty$ otherwise.

(2) **Transmission cost** $f_t(\Pi, R, S; \mathcal{N}, \mathcal{Q})$: The cost to transmit the data (under routing scheme R). Let $r_t(i, j)$ denote the cost to transmit one unit of data over link (i, j). In general, $r_t(i, j) \neq r_t(j, i)$.

(3) **Storage cost** $f_s(\Pi, R, S; \mathcal{N}, \mathcal{Q})$: The cost to store/cache the data. Let $r_s(i)$ denote the cost of storing one unit data at node i. If there is no storage space available at node i, $r_s(i) = \infty$.

Thus, the total cost of scheme (Π, R, S) is

$$f_{\text{cost}}(\Pi, R, S; \mathcal{N}, \mathcal{Q}) = (f_p + f_t + f_s)(\Pi, R, S; \mathcal{N}, \mathcal{Q}).$$

C. Minimum Cost Optimization

We are interested in determining the query process scheme (Π^*, R^*, S^*) that solves

$$\arg \min_{(\Pi, R, S)} f_{\text{cost}}(\Pi, R, S; \mathcal{N}, \mathcal{Q}). \tag{1}$$

Routing and operator placement clearly impact the efficiency of the query processing. Data caching also needs to be taken into account due to the randomness of query processing. If the query frequency $q = 1$, we should transmit and process all raw data for all queries and deliver the results to the fusion center. If q is zero or close to zero, we should cache the raw data where they are generated, and transmit and process relevant data only when a query is acquired. When q lies between zero and one, it may be more beneficial to cache data at nodes in the middle of the network. Thus, choosing proper locations to cache the (intermediate or final) data should be considered when designing efficient query processing schemes.

Consider a special case where the storage cost is zero and $q = 1$. In this case, we should always transmit and process all raw data for all queries and deliver the results to the fusion center. Then, the optimization problem can be simplified to

$$(\Pi^*, R^*) = \arg\min_{(\Pi,R)} f_{\text{cost}}(\Pi, R; \mathcal{N}, \mathcal{Q}). \tag{2}$$

If query graphs are not tree-structured, even the simple optimization problem (2) is known to be NP-complete as shown in [13]. Thus we focus on tree-structured query graphs as defined in Subsection II-D, and propose distributed algorithms which solve the general optimization problem (1) with polynomial complexity.

IV. Distributed Solution

In this section, we study optimization problem (1) for the case that the query graph has a tree structure, where the tree is of depth H_Q. We assume the sensor network is of general topology. We first explore a special case where the query frequency $q = 1$ and the storage cost is zero. We then extend the algorithm to the case where $q \in (0, 1)$ and the storage cost is not zero.

A. Minimum Cost Operator Placement and Routing

In the case where the query frequency $q = 1$ and the storage cost is zero, we only need to consider the placement and routing problems.

1) Centralized Solution: In the context of parallel programming, [13] considers the problem of assigning the modules (tasks) of a program over the processors of an inhomogeneous distributed processor system. They have shown that when the task graph has tree structure, the problem can be solved using a centralized algorithm with complexity $O(MN^2)$. However the centralized algorithm has some limitations. For example, it requires the fusion center to know the entire topology of the network. When the network topology changes, say due to a link or node failure, the whole scheme needs to be re-calculated. So the centralized algorithm is hard to implement in large-scale sensor networks, and this motivates us to propose distributed algorithms in this paper.

2) Distributed Operator Placement and Routing: Our proposed distributed algorithms require that each node i maintains for each data item k the following state information: $C^p(k,i), J^p(k,i), \Pi^p(k,i)$ and $I(k,i)$:

- $C^p(k,i)$: the cost of the *current* best query process scheme to obtain data k at node i;
- $J^p(k,i)$: the neighbor that transmits data k to node i in the current scheme;
- $\Pi^p(k,i)$: indication of whether data k is generated at node i in the current scheme. $\Pi^p(k,i) = 1$ if so and 0 otherwise;
- $I(k,i)$: indication that node i has updated the states of data k, but not yet distributed them to node i's neighbors. $I(k,i) = 1$ if so, 0 otherwise.

We introduce a distributed algorithm that yields the optimal solution to problem (2). When the states $C^p(k,i)$, $J^p(k,i)$, $\Pi^p(k,i)$ for all nodes i and data objects k converge to the optimal scheme (R^*, Π^*), the network executes the following trace back algorithm to resolve the query with the minimum cost.

(i) Generate message (data k^*) at the fusion center, where data k^* is the final data.

(ii) If node i receives message (data k) from node j, it transmits data k to node j whenever data k is available. Furthermore,
   (a) If $\Pi^p(k,i) = 1$, node i sends message (data m) to node $J^p(m,i)$ for each $m \in \Delta_k$, and generates data k whenever data k's children are available.
   (b) If $\Pi^p(k,i) = 0$, node i sends message (data k) to node $J^p(k,i)$.

We next describe the distributed algorithm that yields the optimal states $C^p_*(k,i), J^p_*(k,i), \Pi^p_*(k,i)$. Note that if data k is generated at node i, then the total cost of obtaining data k at node i is:

$$C^p(k,i) = r_p(k,i) + \sum_{m \in \Delta_k} C^p(m,i); \tag{3}$$

if instead data k is transmitted to node i from a neighboring node $j \in A_i$, then the total cost is:

$$C^p(k,i) = C^p(k,j) + D_k r_t(j,i). \tag{4}$$

The basic idea of our distributed algorithm is that each node iteratively (i) updates its states according to (3) and (4) using the information received from its neighbors, (ii) updates its current scheme if it can be improved, and then (iii) sends updates to its neighbors.

The distributed algorithm to solve problem (2) is given in Algorithm 1. After initializing the state information according to lines 1-4, the following two steps are then executed repeatedly:

(i) Lines 6-10: If $I(k,i) = 1$, which means data k's states at node i have been updated but not distributed to node i's neighbors, then node i sends data k's states to all of its neighbors and $I(k,i)$ is set to zero.

Algorithm 1 Query frequency $q = 1$

```
1: for each node i do
2:   for each data object k do
3:     C^p(k,i) = 0 if data k is sensor data created at node
       i, and C^p(k,i) = ∞ otherwise;
4:     I(k,i) = 1;
5: while true do
6:   for each node i do
7:     for each data k do
8:       if I(k,i) = 1 then
9:         Send data k's states to its neighbors;
10:        I(k,i) = 0;
11:  for each node i do
12:    for each data object k (bottom-up along the query
       tree) do
13:      for each j ∈ A_i do
14:        if (node i receives data k's states from node j)
           AND (C^p(k,i) > C^p(k,j) + D_k r_t(j,i)) then
15:          C^p(k,i) = C^p(k,j) + D_k r_t(j,i);
16:          J^p(k,i) = j, Π^p(k,i) = 0, and I(k,i) = 1;
17:      if C^p(k,i) > r_p(k,i) + Σ_{m∈Δ_k} C^p(m,i) then
18:        C^p(k,i) = r_p(k,i) + Σ_{m∈Δ_k} C^p(m,i);
19:        J^p(k,i) = i, Π^p(k,i) = 1, and I(k,i) = 1.
```

(ii) Lines 11-19: According to the information received from its neighbors, node i checks whether it should update the states of its data. For each data k, node i first checks whether the cost could be reduced by obtaining data k from its neighbors (lines 13-16). Then, node i checks whether it is better to generate data k at node i (lines 17-19). Let $I(k,i) = 1$ if data k's states are updated.

The algorithm terminates when no additional information is exchanged between nodes. Note that the algorithm is synchronous: nodes have to wait for the completion of step (i) in order to execute step (ii), and wait for the completion of step (ii) in order to execute step (i). Later we extend the algorithm to be asynchronous, and the asynchronous algorithm terminates faster than the synchronous algorithm in some cases.

The idea of this algorithm is similar to the Bellman-Ford algorithm [10], where each node updates its states according to the information received from its neighbors, and then sends the updated information to its neighbors. In fact when the query graph contains a single data object with no processing requirement, the problem becomes the standard routing problem and our algorithm reduces to the Bellman-Ford algorithm, which finds the min-cost routing path to deliver the data from source to sink.

We next show that Algorithm 1 solves the optimization problem within $L_{\max}(H_Q+1)$ iterations. Recall that H_Q is the depth of the query tree, and $L_{\max}$ is the number of hops of the longest minimum cost path. Specifically, $L_{\max} = \max_{ij} L_{ij}$, where L_{ij} denotes the number of hops of the minimum cost path from node i to node j. Before showing this, we introduce the following notations:

(1) Routing path $\alpha(j,i;k)$: A path over which data k is transmitted from node j to node i.
(2) Data collection tree $\beta(k,i)$ for data k rooted at node i: A tree over which data k is obtained at node i. If data k is generated at node j and forwarded to node i, then

$$\beta(k,i) = \alpha(j,i;k) \cup (\cup_{m\in\Delta_k}\beta(m,j)). \tag{5}$$

(3) Denote by $H(\beta(k,i))$ the depth of tree $\beta(k,i)$: the maximum path length from the root to a leaf in $\beta(k,i)$. If data k is generated at node j and forwarded to node i, from (5) we have

$$H(\beta(k,i)) = H(\alpha(j,i;k)) + \max_{m\in\Delta_k} H(\beta(m,j)).$$

Theorem 1: Algorithm 1 converges to the optimal solution of Problem (2) within $L_{\max}(H_Q+1)$ iterations, and the number of messages sent out per node is $O(Md_{\max}L_{\max}H_Q)$. Furthermore, if the sensor network is formed by wireless nodes, the number of messages sent out per node is $O(ML_{\max}H_Q)$.

Proof: The proof is omitted here due to page limitation. Please refer to [14] for the detailed proof. ■

From the theorem above, we observe that the convergence rate depends on H_Q and $L_{\max}$, where H_Q is the depth of the query tree, and $L_{\max}$ is the number of hops of the longest minimum cost path. If the query graph is a complete binary tree, $H_Q = (\log_2(M+1)) - 1$. Also $L_{\max}$ is usually of the same order as the diameter of the sensor network. If the network is a regular grid, then the diameter is $\sqrt{2N}$. Thus if the query graph is a complete binary tree and the senor network is a regular grid, we expect the number of iterations to be $O(\sqrt{N}\log_2 M)$, and the number of messages each node sent out to be $O(\sqrt{N}M\log_2 M)$.

B. Distributed Caching

In the previous subsection, we focused on the case where the query frequency $q = 1$, corresponding to the situation where every data snapshot is queried. In this subsection, we consider a more realistic case where the query frequency $q < 1$; the user does not query every snapshot, but only those data in which it is interested. As we discussed in Section III-A, PUSH is the optimal scheme for $q = 1$. But for $q < 1$, caching can further reduce the cost. Now, any caching scheme must account for the fact that different nodes have different amounts of storage space. For example, a sensor node may only have limited storage space; on the other hand, a relay node and the fusion center may have substantial storage. Thus, we prefer to cache data at nodes with substantial storage, and we associate different holding costs to different nodes to reflect this fact. Also, when we cache data in the middle of the network, the nodes caching the data need to be notified each time a query is generated. We assume the size of the signaling message needed to do this is D_q.

In this subsection, we present a distributed algorithm that determines the minimum cost scheme. First, each node i needs to maintain the following information for each data

k: $C^p(k,i)$, $J^p(k,i)$, $\Pi^p(k,i)$, $C^s(k,i)$, $J^s(k,j)$, $\Pi^s(k,i)$, $S(k,i)$ and $I(k,i)$.

In particular, $C^p(k,i)$, $J^p(k,i)$ and $\Pi^p(k,i)$ are associated with the *current* best processing scheme to obtain data k at node i; while $C^s(k,i)$, $J^s(k,i)$, $\Pi^s(k,i)$, and $S(k,i)$ are associated with the *current* minimum cost scheme involving *caching*, where $S(k,i)$ represents the caching decision with $S(k,i) = 1$ if data k is cached at node i, and 0 otherwise.

Note that in the algorithm, $C^p(k,i)$, $J^p(k,i)$ and $\Pi^p(k,i)$ are used to check whether caching data k at node i is better than pulling data k from other nodes when the query is injected.

If the final data is not pushed to the fusion center, we need to notify the sensors or the nodes caching the data each time a query is injected. Thus, to obtain the minimum cost query processing scheme, we need to find the minimum cost path to transmit the required signaling messages. The distributed Bellman-Ford algorithm [10] is used to find minimum cost paths, and is included in our algorithm—Algorithm 2. Thus, node i needs to maintain for each node j (not just for $j \in A_i$) the following information: $R(i,j)$, $G(i,j)$ and $I^b(i,j)$. In particular $R(i,j)$ is the *current* cost to transmit one unit of data from node i to node j; $G(i,j)$ is the next hop node i should take to reach node j; and $I^b(i,j) = 1$ if the information $(R(i,j), G(i,j))$ has been updated, but not distributed to node i's neighbors.

The distributed algorithm to obtain the minimum cost scheme is given as Algorithm 2. Similar to Algorithm 1, it first initializes network state. It then repeats following two steps:

(i) Lines 3-8: If the states of node i have been updated, node i sends the updates to its neighbors.

(ii) Lines 9-31: Node i updates its state according to the information received from its neighbors. It first uses the distributed Bellman-Ford algorithm to update $R(i,j)$, $G(i,j)$ and $I^b(i,j)$ (lines 10-13). Then, node i updates the cost related to caching (lines 17-19 and lines 23-25). Finally, for each data k, node i compares the current scheme with three different schemes according to the updated information received: (1) Obtain data k from node j ($j \in A_i$) when the query is injected (lines 20-22); (2) Compute data k at node j when the query is injected (lines 26-28); (3) Cache data k at node j (lines 29-31). If any of the three schemes is better, node i chooses the best and updates the states of data k.

In the following theorem, we show that the algorithm converges to the optimal scheme within $(H_Q+2)L_{\max}$ iterations.

Theorem 2: Algorithm 2 converges to the optimal solution of Problem 1 after at most $(H_Q+2)L_{\max}$ iterations, and the number of messages exchanged is $O(Md_{\max}L_{\max}H_Q)$.

Furthermore, if the sensor network is formed by wireless nodes, the number of messages sent out per node is $O(ML_{\max}H_Q)$.

Proof: The proof is omitted here due to page limitation. Please refer to [14] for the detailed proof. ■

Algorithm 2 Query frequency $q < 1$.

1: Initialize the states at each node as line 1-4 in Algorithm 1;
2: **while** true **do**
3: **for** each node i **do**
4: **if** $I^b(i,j) = 1$ **then**
5: Send $R(i,j)$ to its neighbors, and let $I^b(i,j) = 0$;
6: **for** each data k **do**
7: **if** $I(k,i) = 1$ **then**
8: Send data k's states to its neighbors, and let $I(k,i) = 0$;
9: **for** each node i **do**
10: **for** each $h \neq i$ **do**
11: **for** each $j \in A_i$ **do**
12: **if** node i receives $R(j,h)$ and $R(j,h) + r_t(i,j) < R(i,h)$ **then**
13: Let $R(i,h) = r_t(i,j) + R(j,h)$, $G(i,h) = j$, and $I^b(i,h) = 1$;
14: **for** each data object k (bottom-up along the query tree) **do**
15: **for** each $j \in A_i$ **do**
16: **if** node i receives data k's states from node j **then**
17: **if** $C^p(k,i) > C^p(k,j) + D_k r_t(j,i)$ **then**
18: Let $C^p(k,i) = C^p(k,j) + D_k r_t(j,i)$;
19: Le $J^p(k,i) = j$, $\Pi^p(k,i) = 0$, and $I(k,i) = 1$;
20: **if** $C^s(k,i) > C^s(k,j) + q(D_k r_t(j,i) + D_q R(i,j))$ **then**
21: Let $C^s(k,i) = C^s(k,j) + q(D_k r_t(j,i) + D_q R(i,j))$.
22: Let $J^s(k,i) = j$, $\Pi^s(k,i) = 0$, $S(k,i) = 0$, and $I(k,i) = 1$;
23: **if** $C^p(k,i) > r_p(k,i) + \sum_{m\in\Delta_k} C^p(m,i)$ **then**
24: Let $C^p(k,i) = r_p(k,i) + \sum_{m\in\Delta_k} C^p(m,i)$;
25: Let $J^p(k,i) = i$, $\Pi^p(k,i) = 1$, and $I(k,i) = 1$;
26: **if** $C^s(k,i) > qr_p(k,i) + \sum_{m\in\Delta_k} C^s(k,i)$ **then**
27: Let $C^s(k,i) = qr_p(k,i) + \sum_{m\in\Delta_k} C^s(m,i)$.
28: Let $J^s(k,i) = i$, $\Pi^s(k,i) = 1$, $S(k,i) = 0$, and $I(k,i) = 1$;
29: **if** $C^s(k,i) > C^p(k,i) + D_k r_s(i)$ **then**
30: Let $C^s(k,i) = C^p(k,i) + D_k r_s(i)$.
31: Let $\Pi^s(k,i) = \Pi^p(k,i)$, $J^s(k,i) = J^p(k,i)$, $S(k,i) = 1$, and $I(k,i) = 1$;

Similar to Algorithm 1, when states $C^s(k,i), J^s(k,i)$, $\Pi^s(k,i)$, and $S(k,i)$ for all nodes i and data objects k converge to the optimal ones, the network executes the following trace back algorithm to resolve the query with the minimum computation, communication and storage cost.

(i) Generate message (Data k^*, Pull) at the fusion center, where data k^* is the final data.

(ii) If node i receives message (Data k, Pull) from node j, then

(a) If $\Pi^s(k,i) = 1$ and $S(k,i) = 1$, node i sends message (data m, Push) to node $J^s(m,i)$ for each $m \in \Delta_k$, and generates data k once data k's children are available.
(b) If $\Pi^s(k,i) = 1$ and $S(k,i) = 0$, node i sends message (data m, Pull) to node $J^s(m,i)$ for each $m \in \Delta_k$, and requests data m from node $J^s(m,i)$ for each $m \in \Delta_k$ when there is a request for data k at node i.
(c) If $\Pi^s(k,i) = 0$ and $S(k,i) = 1$, node i sends message (data k, Push) to node $J^s(k,i)$.
(d) If $\Pi^s(k,i) = 0$ and $S(k,i) = 0$, node i sends message (data k, Pull) to node $J^s(k,i)$, and requests data k from node $J^s(k,i)$ when there is a request for data k at node i.

If node i receives message (Data k, PUSH) from node j, it transmits data k to node j whenever data k is available. Furthermore,

(a) If $\Pi^s(k,i) = 1$, node i sends message (data m, Push) to node $J^s(m,i)$ for each $m \in \Delta_k$, and generates data k whenever data k's children are available.
(b) If $\Pi^s(k,i) = 0$, node i sends message (data k, Push) to node $J^s(k,i)$.

C. Asynchronous Algorithms

Algorithms 1 and 2 execute in a synchronous way. For example, consider Algorithm 1, nodes update their states (lines 11-19) only after all nodes finish distributing the updated information to their neighbors (lines 6-10); similarly, nodes send out their updated information only after all nodes finish updating their states. This requires that all nodes be fully synchronous and know when other nodes finish updating and finish sending out updated information. However, the algorithms can be easily modified to run in an asynchronous way, where each node updates its states right after it receives information from its neighbors; and sends updated messages to its neighbors right after its states are updated. It is easy to see that if nodes process update messages one by one, then the asynchronous versions converge after a finite number of message exchanges. Furthermore, if the computation time is negligible compared to the transmission time, all messages have the same transmission time over the same link, and nodes processes update messages one by one, then the asynchronous versions terminate faster than the synchronous algorithms. A detailed description and analysis of these asynchronous algorithms can be found in [14], and is omitted here due to page limitation.

D. Cost and Topology Change

Communication, computation, and storage costs can change over time. Furthermore, links can fail. In this section we show how to modify our algorithm so that it can handle such changes. Consider first the case where one of the costs decreases, e.g., node i's storage cost decreases due to the addition of storage space to node i. [15] showed that it suffices to re-initiate execution of the algorithms (node i first updates its states according to the decrease in cost and then distributes the updated information to its neighbors, and then all nodes begin to exchange information). The algorithm then converges to the new optimal scheme in a finite number of steps.

Now consider the case of either a cost increase or a link failure (the cost is infinity). Simply executing the algorithm will result in a "count to infinity" problem [10]. It may take an infinite number of iterations for the algorithm to converge to the new optimal solution when a link fails, or B iterations to converge if the cost increase is B. We can use the algorithm proposed in [15] to handle this case. The idea is to "freeze" the query-processing scheme when some cost increases or a link fails. All nodes then update the cost of the current scheme according to the changes. After that, nodes are "unfreezed" and start to exchange cost information to find the new optimal query-processing scheme.

V. Simulation-Based Evaluation

In this section, we investigate our algorithms through simulations. We shall focus on the synchronized algorithms although we expect similar results for the asynchronous algorithms.

We study the proposed algorithms in the following simulation environment: the network is an $N \times N$ grid and query graphs are tree structured. Grid based network infrastructures have been promoted in streaming applications in a number of studies, e.g. [12], [16]. Tree query graphs are related to the most common query services such as filtering or aggregation services [3], [17], [18].

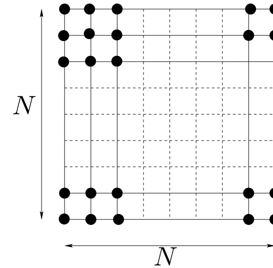

Fig. 2. An $N \times N$ Grid Network.

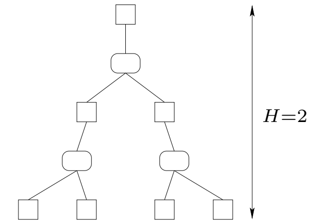

Fig. 3. A Tree-Structured Query Graph.

Figure 2 shows an $N \times N$ grid network used in our simulation. We measure the size of networks in terms of width of the grid (in nodes). Thus, a width 50 network contains 2500 nodes. To make the problem interesting, we assume storage and computation costs are comparable to the communication costs. Such an assumption typically holds for a query that involves expensive operations, such as text, image or video filtering, thus it may be expensive to execute at low-capability nodes or sensors.

We consider a query graph which is a complete binary tree with depth H as shown in Figure 3. For simplicity, we assume the size of all data objects to be one, the size of signaling message to be 0.1, and the probability of query injection to be q.

Note that we do not claim that the parameter settings in our experiments are the most representative choices in practice, rather we use them for illustration purposes only, so as to demonstrate the significant cost reduction of our algorithm over other naive methods, and to provide some insights on how the placement and caching strategies should vary under different scenarios. In the following experiments, for each parameter setting, we execute the simulation 50 times and report the averages. At each time, a new grid network and a new complete binary tree query are randomly generated.

A. *Case Studies on Cost Efficiency*

We first investigate the cost-efficiency of Algorithm 2. We compare Algorithm 2 to the following two algorithms under different sizes of networks, different sizes of query trees, and different probabilities of query injection:

1) PUSH: The sensors transmit the data to the relay nodes after the data are generated. The relay node will process the data if possible and then transmit to the fusion center. The routing and operator placement schemes are obtained in Algorithm 1.
2) PULL: The fusion center signals the sensor nodes when the query is injected. Then the data are transmitted and processed as the scheme obtained in Algorithm 1.

We assume 10% of the nodes (which are randomly chosen) in the network are high performance nodes. The storage cost of high-performance nodes is zero, and the computation cost is one per unit data. For the rest of the nodes in the network, the unit storage cost is 50 and unit computation cost is 5. The fusion center is assumed to be the left-top node whose storage and computation costs are both zero. For each link, the unit communication cost is an integer randomly chosen from 0 to 20. The sources for the leaf (raw) data are uniformly randomly pinned to the nodes of the grid network.

Experiment 1: In this experiment, we fix $H = 8$, $q = 0.2$ and vary the grid width by setting $N = 10, 15, \ldots, 50$. We compare the total query processing cost using Algorithm 2 with that under PUSH and PULL, the number of iterations taken for Algorithm 2 to converge is also measured. The average costs of the three algorithms and the number of iterations taken for Algorithm 2 to converge are shown in Figure 4. Observe that when the size of the network is small

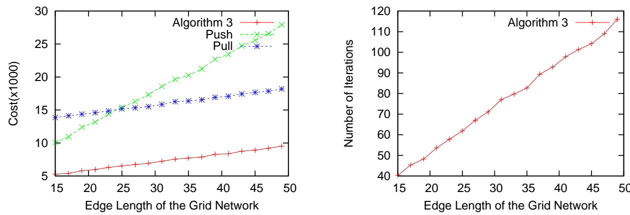

Fig. 4. Experiment 1

(10×10), the storage cost dominates the communication cost, so PUSH is better than PULL. When the size of the network is large (50×50), the communication cost dominates the storage cost, and PULL is better than PUSH. However for all N, Algorithm 2 significantly reduces the total cost by using an optimal caching scheme. Furthermore, from Figure 4, we can see that Algorithm 2 converges to the optimal query processing scheme very fast. For example it only takes 118 iterations in a grid network with 2500 nodes.

Experiment 2: In this experiment, we fix $N = 50$, $q = 0.2$ and vary the depth of the query graphs by setting $H = 1, 2, \ldots, 8$. The total query processing costs under the three different schemes and the number of iterations taken for Algorithm 2 to converge are shown in Figure 5.

Again we see that the total cost is significantly reduced by using the query processing scheme obtained by Algorithm 2, and Algorithm 2 converges to the optimal scheme very fast.

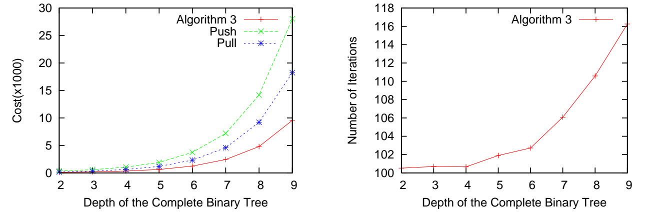

Fig. 5. Experiment 2

Experiment 3: In this experiment, we fix $N = 40$, $H = 7$ and vary the query probability by setting $q = 0.1, 0.2, \ldots, 1$. The results are shown in Figure 6.

Observe that Algorithm 3 outperforms both PUSH and PULL for all values of q. Also PULL outperforms PUSH For small q, and becomes worse than PUSH when q is around 0.4; while PUSH becomes optimal for large q (when q is larger than 0.6).

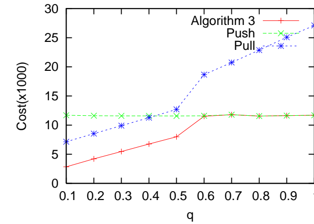

Fig. 6. Experiment 3

B. *Case Studies on Caching Strategy*

Next, we investigate the caching strategy and show how Algorithm 2 adapts the caching locations under different settings. We consider a 40×40 grid network and a complete binary tree query with depth 7. To identify the caching strategy, we assume the sensor data are pinned to the nodes far from the fusion center, i.e., the raw data are randomly positioned in the region

$$([N-4, N-1] \times [0, N-1]) \cup ([0, N-1] \times [N-4, N-1]).$$

We also assume that the costs are hierarchical, i.e., the closer to the fusion center, the lower the computation and storage

costs. For a node positioned at (i, j), it is assumed that the computation cost is $(i+j)^\gamma$ and storage cost is $10(i+j)^\gamma$, where γ is some positive constant. For each link, the unit communication cost is an integer randomly chosen from 0 to 20. To evaluate the caching strategy, we calculate the total number of hops required to transmit/process the data objects from the nodes caching them to the fusion center. The smaller the number of hops, the closer to the fusion center the data is cached.

Experiment 4: In this experiment, we fix $q = 0.2$ and vary γ from 0 to 1, where $\gamma = 0$ implies that all nodes have a uniform cost, and $\gamma > 0$ implies that the nodes far from the fusion center have less computation and storage resources, so the computation cost and storage costs are larger. From Figure 7, we see that Algorithm 2 pushes the data to be near the fusion center as γ increases. Thus, if the network consists of resource-poor sensors, a good caching strategy is to cache the data in the middle of the network.

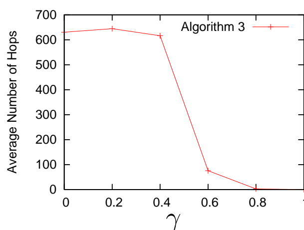

Fig. 7. Experiment 4

Experiment 5: In this experiment, we fix $\gamma = 0.4$ and vary q from 0 to 1. From Figure 8, we can see that Algorithm 2 pushes the data to be near the fusion center as q increases. When $q \geq 0.8$, the number of hops is equal to zero, which means that PUSH is optimal.

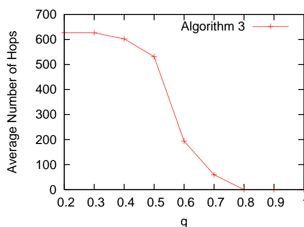

Fig. 8. Experiment 5

VI. Conclusions

In distributed stream-based sensor networks, it is important to utilize the limited bandwidth, computation and storage for efficient in-network processing in anticipation of a mix of queries that is only known stochastically. In this paper, we have considered the operator placement and intermediate data caching problem as a whole, and formally defined the problem as the optimal node assignment for the operators and intermediate data objects that requires the minimum total cost in communication, computation and storage. We have proposed distributed algorithms to solve the problem and have shown that they yield the minimum cost query process schemes when the queries are tree-structured. This is a significant theoretical improvement over the heuristics previously proposed. Our algorithms are practically attractive in that they require only information exchanges between neighbors, can be executed asynchronously, and are adaptive to cost and topology changes and resilient to node or link failures.

We believe our algorithms can be extended to handle general query graphs and to take into account various capacity constraints. In either case, the problem becomes NP-hard and one needs to rely on heuristics or approximation methods in order to produce affordable efficient solutions. This is part of our on-going research. Our preliminary investigation shows that by adapting our previous algorithms using simple heuristics, the performance gain over other naive methods can also be significant as for the tree query graphs.

References

[1] S. Chandrasekaran and et.al., "Telegraphcq: Continuous dataflow processing for an uncertain world," in *Proc. of CIDR*, 2003.
[2] P. Bonnet, J. Gehrke, and P. Seshadri, "Towards sensor database systems," in *Proc. of MDM*, 2001, pp. 3–14.
[3] S. Madden and Others, "Tag: A tiny aggregation service for ad hoc sensor networks," *ACM SIGOPS Operating Systems Rev.*, pp. 131–146, 2002.
[4] B. Bonfils and P. Bonnet, "Adaptive and decentralized operator placement for in-network query processing," in *Proc. of IPSN*, 2003.
[5] Y. Ahmad and U. Cetintemel, "Network-aware query processing for stream-based applications," in *Proc. of 30th VLDB Conf.*, 2004.
[6] U. Srivastava, K. Munagala, and J. Widom, "Operator placement for in-network stream query processing," in *Proc. of PODS*, 2005, pp. 250–258.
[7] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, M. Welsh, and M. Seltzer, "Network-aware operator placement for stream-processing systems," in *Proc. of ICDE*, 2006.
[8] Z. Abrams, H.-L. Chen, L. Guibas, J. Liu, and F. Zhao, "Kinetically stable task assignment for networks of microservers," in *Proc. of IPSN*, 2006, pp. 93–101.
[9] Z. Abrams and J. Liu, "Greedy is good: On service tree placement for in-network stream processing," in *Proc. of ICDCS*, 2006.
[10] D. Bertsekas and J. Tsitsiklis, *Parallel and Distributed Computation: Numerical Methods*. Prentice-Hall, Englewood Cliffs, N.J., 1989.
[11] D. J. Abadi and et.al., "The design of the borealis stream processing engine," in *Proc. of CIDR*, 2005, pp. 277–289.
[12] L. Chen, K. Reddy, and G. Agrawal, "Gates: A grid-based middleware for processing distributed data streams," in *Proc. of HPDC*, 2004.
[13] S. Bokhari, "A shortest tree algorithm for optimal assignments across space and time in a distributed processor system," *IEEE Trans. on Soft. Eng.*, vol. 7, no. 6, 1981.
[14] L. Ying, Z. Liu, D. Towsley, and C. Xia, "Distributed operator placement and data caching in large-scale sensor networks," *IBM Research Report, RC 24297*, 2007.
[15] J. M. Jaffe and F. M. A. Moss, "A responsive distributed routing aglorithm for computer networks," *IEEE Trans. on Comm*, pp. 1758–1762, 1981.
[16] R. Kuntschke, B. Stegmaier, A. Kemper, and A. Reiser, "Streamglobe: Processing and sharing data streams in grid-based p2p infrastructures," in *Proc. of VLDB*, 2005, pp. 1259–1262.
[17] S. R. Madden, M. J. Franklin, J. M. Hellerstein, and W. Hong, "Tinydb: an acquisitional query processing system for sensor networks," *ACM Transactions on Database Systems (TODS)*, vol. 30, no. 1, pp. 122–173, March 2005.
[18] A. Dobra, M. Garofalakis, J. Gehrke, and R. Rastogi, "Processing complex aggregate queries over data streams," in *Proc. of 2002 SIGMOD*, pp. 61–72.