Query Plan Execution in a Heterogeneous Stream Management System for Situational Awareness

Indrakshi Ray
Computer Science Department
Colorado State University
Fort Collins, Colorado
Email: iray@cs.colostate.edu

Sanjay K. Madria
Computer Science Department
Missouri University of Science and Technology
Rolla, Missouri
Email: madrias@mst.edu

Mark Linderman
Information Directorate
Air Force Research Laboratory
Rome, New York
Email: mark.linderman@rl.af.mil

***Abstract*—Battlefield monitoring involves collecting streaming data from different sources, transmitting the data over a heterogeneous network, and processing queries in real-time in order to respond to events in a timely manner. Nodes in these networks differ with respect to their processing, storage and communication capabilities. Links in the network differ with respect to their communication bandwidth. The topology of the network itself is subject to change, as the nodes and links may become unavailable. Continuous queries executed in such environments must also meet some quality of service (QoS) requirements, such as, response time, throughput, and memory usage. We propose that the processing of the queries be shared to improve resource utilization, such as storage and bandwidth, which, in turn, will impact the QoS. We show how multiple queries can be represented in the form of an operator tree, such that their commonalities can be easily exploited for multi query plan generation. Query plans may have to be updated in this dynamic environment (network topology changes, arrival of new queries, arrival pattern of streams altered); this, in turn, necessitates migrating operators from one set of nodes to another. We sketch some ideas about how operator migration can be done efficiently in such environments.**

***Index Terms*—Data stream management systems, situational awareness**

I. Introduction

Situation monitoring applications, such as, battlefield monitoring, emergency and threat management, are on the rise [1], [2], [3], [4]. Such applications need to collect high-speed data from multiple domains, process it, compute results on-the-fly, and take actions in real-time. Data Stream Management Systems (DSMSs) [2], [5], [6], [4], [7], [8] have been proposed to address the data processing needs of such applications.

Most of the works in DSMS assume a centralized architecture where the data collected by sensors is sent to a DSMS that is responsible for real-time processing of the data. The centralized architecture is suitable when the nodes collecting sensor data have a good network connection to the DSMS. Researchers have also worked on stream architectures where the processing of queries is distributed over various nodes [7], [9]. Researchers have started investigating several issues pertaining to stream processing over heterogeneous networks [10], [11], [12], [13], [14]. However, query processing over a heterogeneous, dynamic network is still in its infancy and a lot of issues need to be addressed to support the type of applications described below.

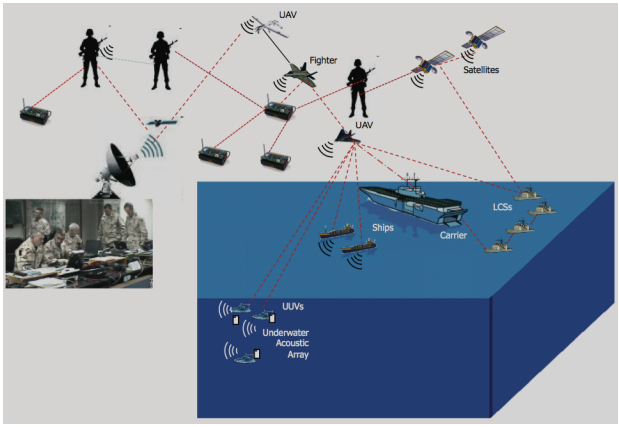

Fig. 1. Battlefield Monitoring

Consider, the cooperative combat air patrol application [15], shown in Fig. 1. We have Unmanned Aerial Vehicles (UAVs), helos, fighters, Airborne Warning and Control Systems (AWACS), ships, carrier, and Littoral Combat Ships (LCSs) cooperatively defending a strategic body of water. In addition, we have various types of ground platforms that may be stationary or mobile. Mobile platforms may travel in designated or random tracks at varying speeds. We need information about contact positions, lines of bearing, tracks, and other signals. Some example queries submitted to such a system are: (i) Send alert about Surface-to-Air Missile (SAM) locations within 12 NM of the area bounded by latitude1, longitude1, and latitude2, longitude2. (ii) Stream all forward blue force positions within 2 NM of the area bounded by latitude 1, longitude 1 and latitude 2, longitude 2. DSMS continuously executes such queries until there is no more streaming data or when the user explicitly stops it.

The battlefield example operates over a heterogeneous network, which consists of different types of nodes interconnected by various types of links. The nodes differ with respect to their computation and storage capacity, whereas the links vary with respect to their communication capabilities. Moreover, the network topography may vary at different instances, as some types of nodes and links may not be operational at any given point of time; such cases typically arise in sensor networks

where nodes have limited power supply and may die. Distributing the processing of information over this heterogeneous network may reduce computation and communication costs and provide a better response time to queries. Consequently, a distributed DSMS that operates over a heterogeneous and dynamic network is needed.

A number of research issues must be addressed before realizing a heterogeneous, distributed and dynamic DSMS. We focus our attention on the problem of query plan execution in such a dynamic environment. Continuous queries must be executed in an efficient manner in this environment, parts of which are resource constrained. Continuous queries typically have Quality of Service (QoS) requirements, such as tuple latency, throughput, accuracy, and memory usage. We also have various types of resource constraints: (i) link bandwidth, (ii) processing, storage, and communication capabilities of various nodes, and (iii) link and node availability. The application and the operational environment will together determine the QoS metrics and the constraints that are relevant.

In the context of our motivating example, the timeliness of the query execution is important and so we define a QoS metric based on tuple latency. Bandwidth is the major bottleneck in our heterogeneous network. Consequently, the objective of the query plan generation is to minimize the bandwidth utilization, subject to the constraints at the individual nodes and links of the network, such that the QoS requirements are satisfied.

In a battlefield scenario, when an interesting event occurs, many queries may be posed w.r.t. the specific event. Thus, multiple queries may be issued over the same or related data streams. We argue that multiple queries on same or related data streams may share their processing to alleviate the processing, storage, and communication costs. We demonstrate how multiple queries can be represented as operator trees and how they can be mapped on to the network to minimize the bandwidth utilization, such that the constraints of the links and nodes and QoS requirements are satisfied.

A change of the network topology, arrival of new queries, and change in the arrival rate of stream elements may make the current query plan inefficient. Consequently, the query plan must be changed and operators executing on the current nodes may have to be migrated on other nodes. Operator migration necessitates changing the data flow; for blocking operators (those which process a set of tuples) state information must be transferred as well. Towards this end, we give some heuristics about how to perform efficient operator migration.

The rest of the paper is organized as follows. Section II shows how to represent multiple queries whose processing can be shared. Section III formulates our goal and formalizes the constraints of our environment. Section IV provides some heuristics on the initial query plan generation and task migration. Section V concludes the paper with pointers to future directions.

II. Multi Query Plan Representation

In DSMSs, queries are executed continuously in a resource constrained environment. Consequently, it is important for the queries to share their processing to the extent possible such that resource utilization is minimized. We discuss how plans corresponding to multiple queries can be represented. Note that, for a given query, generating all the plans is infeasible, so we demonstrate our ideas using a single query plan and later show how the plans from different queries can be merged.

We use some example queries to illustrate our ideas. We have a stream `OBJECT` that streams the position of detected objects in a given zone whose schema is given below.

```
OBJECT (object id (oid), object type (type),
        latitude (lat), longitude (lon),
           detection time, detector id (did))
```

Consider the queries:

- $Q1$: SELECT * FROM $object$
  WHERE $lat1 \leq lat$
  AND $lat \leq lat2$

- $Q2$: SELECT COUNT(oid) FROM $object$
  [RANGE 1 minute]
  WHERE $lat1 \leq lat$
  AND $lat \leq lat2$

- $Q3$: SELECT * FROM $object$
  WHERE $lat1 \leq lat$
  AND $lat \leq lat2$
  AND $type =$ "SAM"

We change *select* operators having more than one condition into multiple *select* operators, such that each operator is associated with a single condition. We change the *project* operators that eliminate multiple attributes into a set of *project* operators each of which eliminates a single attribute. We rewrite this query plan in the form of an operator tree which captures all the information needed for processing the query.

Definition 1: **[Operator Tree]** An *operator tree* for a query Q_x, represented in the form of $OPT(Q_x)$, consists of a set of vertices V_{Q_x} and a set of edges E_{Q_x}. Each internal vertex v_i corresponds to some operator in the query Q_x. Each edge (v_i, v_j) in this tree connecting vertex v_i to vertex v_j signifies that the output of vertex v_i is the input to vertex v_j. Each vertex v_i has a label that provides details about processing the corresponding operator. The name component of the label, denoted by $v_i.op$, specifies the type of the operator, such as, *select*, *project*, *join* and *average*. The parameter component of the label, represented by $v_i.parm$, denotes the simple filter condition for the *select* operator or the eliminated attribute for the *project* operator. The synopsis component, denoted by $v_i.syn$, is needed for operators specified with windows that require a set of tuples to accumulate before processing can start, such as, *join*, *count*, and *sum*. The synopsis component has two attributes, namely, type and size, that specify the type of window (tuple-based, time-based) and its size. The input queue component of the label, denoted by $v_i.inputQueue$, lists the input streams needed by the operator. The output queue component, denoted by $v_i.outputQueue$, indicates the output streams produced by the operator that can be used by

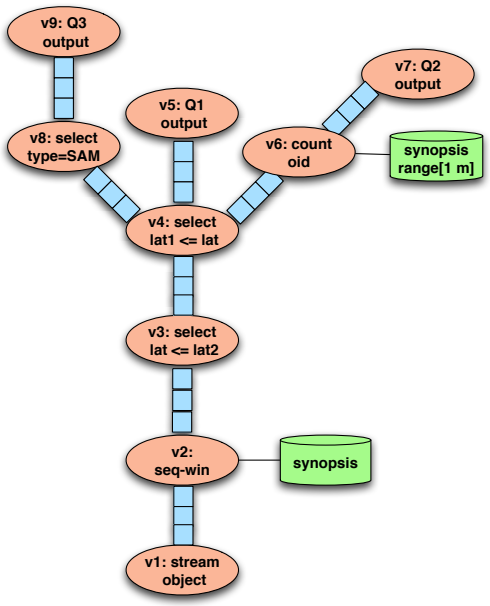

Fig. 2. Merged Operator Tree for Q_1, Q_2 and Q_3

the vertices or sent as response to the user. The leaf nodes of the operator tree represent the source nodes for the data streams and the root nodes are the sink nodes receiving output.

Definition 2: **[Operator Tree Path]** An *operator tree path* is a path in the operator tree from the source node to the sink node.

Operator tree for $Q1$ is given by a subtree of the tree shown in Fig. 2. $OPT(Q1) = (V_{Q1}, E_{Q1})$ where $V_{Q1} = \{v1, v2, v3, v4, v5\}$ and $E_{Q1} = \{(v1, v2), (v2, v3), (v3, v4), (v4, v5)\}$. The operator tree shown in Fig. 2 has three paths: $\langle v1, v2, v3, v4, v5\rangle$, $\langle v1, v2, v3, v4, v6, v7\rangle$, and $\langle v1, v2, v3, v4, v8, v9\rangle$.

Typically, in a DSMS there can be several queries that are being executed concurrently. Query sharing will minimize the resource consumption of these queries. Query sharing obviates the need for evaluating the same operator(s) multiple times if different queries need it. In such a case, the operator trees of different queries can be merged. In the Fig. 2, we show how the operator trees of Q_1, Q_2, and Q_3 can be merged. They use the same *seq-win* operator, as there is one seq-win operator per stream. The three queries also share some *select* operators.

If vertices belonging to different operator trees are equivalent, then only one vertex needs to be computed for evaluating the queries corresponding to these different operator trees. In the following we describe our notion of equivalence.

Definition 3: **[Equivalence of Vertices]** Vertex $v_i \in V_{Q_x}$ is said to be *equivalent* to vertex $v_j \in V_{Q_y}$, denoted by $v_i \equiv v_j$, where v_i, v_j are in the operator trees $OPT(Q_x)$, $OPT(Q_y)$ respectively, if the following condition holds: $v_i.op = v_j.op \wedge v_i.parm = v_j.parm \wedge v_i.syn = v_j.syn \wedge v_i.inputQueue = v_j.inputQueue$.

Definition 4: **[Query Sharing]** Query Q_x can be *shared* with an ongoing query Q_y submitted only if there exists $v_i \in V_{Q_x}$ and $v_j \in V_{Q_y}$, such that $v_i \equiv v_j$.

When operator trees corresponding to queries have equivalent vertices, we can generate the merged operator tree using Algorithm 1. Fig. 2 illustrates the sharing of $OPT(Q1)$, $OPT(Q2)$ and $OPT(Q3)$. The merged operator tree signifies the processing of the partially shared queries.

Thus, the given continuous queries may generate multiple operator trees; some of which represent multiple queries with sharing and others denote single queries.

INPUT: $OPT(Q_x)$ and $OPT(Q_y)$
OUTPUT: $OPT(Q_{xy})$ representing the merged operator tree
Initialize $V_{Q_{xy}} = \{\}$
Initialize $E_{Q_{xy}} = \{\}$
foreach *vertex* $v_i \in V_{Q_x}$ **do**
  $V_{Q_{xy}} = V_{Q_{xy}} \cup v_i$
end
foreach $(v_i, v_j) \in E_{Q_x}$ **do**
  $E_{Q_{xy}} = E_{Q_{xy}} \cup (v_i, v_j)$
end
foreach *vertex* $v_i \in V_{Q_y}$ **do**
  if $\nexists v_j \in V_{Q_x}$ *such that* $v_i \equiv v_j$ **then**
    $V_{Q_{xy}} = V_{Q_{xy}} \cup v_i$
  end
end
foreach $(v_i, v_j) \in E_{Q_y}$ **do**
  if $(v_i, v_j) \notin E_{Q_{xy}}$ **then**
    $E_{Q_{xy}} = E_{Q_{xy}} \cup (v_i, v_j)$
  end
end

Algorithm 1: Merge Operator Trees

III. Constraints in Multi Query Plan Generation

Let $\mathbf{Q}$ be the set of queries submitted to the DSMS. Let $\{OPT(Q_a), OPT(Q_b), \ldots, OPT(Q_p)\}$ be the set of operator trees for the queries in $\mathbf{Q}$. Let $\mathbf{V}$, $\mathbf{E}$ be the set of vertices, edges respectively, of the operator trees for the queries in $\mathbf{Q}$. We use the notation v_i to denote a vertex of the operator tree, where $v_i \in \mathbf{V}$. We use (v_i, v_j) to denote the edge joining vertex v_i to vertex v_j, where $(v_i, v_j) \in \mathbf{E}$. Consider the three queries $Q1$, $Q2$, and $Q3$ given in Section II and whose operator tree appears in Fig. 2. In this case,

$\mathbf{V} = \{v1, v2, v3, v4, v5, v6, v7, v8, v9\}$
$\mathbf{E} = \{(v1, v2), (v2, v3), (v3, v4), (v4, v5), (v4, v6), (v6, v7), (v4, v8), (v8, v9)\}$.

Each operator tree must be executed in our heterogeneous network $(\mathbf{N}, \mathbf{L})$ which consists of the set of nodes $\mathbf{N}$ and links $\mathbf{L}$. Each node i is represented as a 5-element vector, $\langle location(i), memory(i), cpu(i), power(i), oper(i)\rangle$ where $location(i)$, $memory(i)$, $cpu(i)$, $power(i)$, $oper(i)$ denote the current location, available memory, available processing power, available battery power, and operators supported by node i at any given point of time. Each link (i, j) is represented as a 3-element vector, $\langle band(i, j), avail(i, j), ls(i, j)\rangle$, where

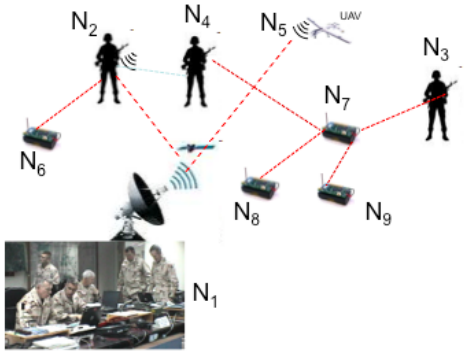

Fig. 3. Dynamic and Heterogeneous Network

$band(i,j)$, $avail(i,j)$, $ls(i,j)$ denote available bandwidth, link availability, link stability respectively. Example of a heterogeneous network is given in Fig. 3.

Note that, we use the term *vertices* and *edges* when we refer to an operator tree, whereas we use the term *nodes* and *links* when we refer to the distributed network. In order to execute the query (queries) represented by the operator tree, the various vertices of the operator tree must be mapped on to one or more nodes of the network. For example, the vertex v_i of the operator tree can be mapped on to node p and vertex v_j can be mapped on to node q. Thus, if v_i needs to send its results to v_j, we must use the link (p,q) of the network to transmit the result. Let $result_{pi,qj}$ be the set of output stream tuples obtained by computing vertex v_i of the operator tree on node p that must be sent to vertex v_j at node q. Let $size(x)$ denote the cardinality of stream x. Let $c_{p,q}$ be the bandwidth cost involved to transmit unit data from node p to node q. The bandwidth cost to transmit this data, denoted by *bandCost*, is given by,

$$bandCost(p,q) = (\sum_{i,j \in \mathbf{V}} size(result_{pi,qj})) * c_{p,q}$$

Our goal is to minimize this total cost:

$$\min(\sum_{(p,q) \in \mathbf{L}} bandCost(p,q))$$

Let us consider the operator tree given in Fig. 2. Clearly, if all the vertices, namely, $v1, v2, \ldots, v9$ are mapped on to the same network node, say, node N_1 in Fig. 3, then the bandwidth utilization is minimum. In practice this may not be feasible, as the source of the stream and sink that receives the output may be located on different nodes.

In order to map an operator tree to our network, the source and sink of the operator tree must be executed at designated nodes of the network; source vertex of the operator tree is mapped to the node producing the data stream whereas the sink vertex is mapped to the node which receives the results of the query. The cost of the query plan will therefore depend on how the other vertices are mapped on to the nodes. Here again, there exists numerous possibilities of mapping operator tree vertices to network nodes. Each such mapping produces a query plan.

Consider the network shown in Fig. 3 which consists of nine nodes, namely, N_1, N_2, ..., N_9. N_1 is a base station. N_5 is mobile but has adequate storage and processing capabilities, but limited bandwidth. N_2, N_3, and N_4 are mobile nodes with medium processing and storage capabilities and limited bandwidth. N_2, N_3, N_4, and N_5 act as cluster heads. N_6, N_7, N_8, and N_9 are nodes with limited storage, processing, and bandwidth. Consider mapping the operator tree shown in Fig. 2 to the network shown in Fig. 3. Suppose N_8 is detecting the objects in some zone and N_3 and N_4 need results of queries $Q1$ and $Q2$ respectively. Even with this simple configuration there are various possibilities. For example, N_8 may execute the entire query $Q2$ and send its results to N_4 via N_7. This incurs the minimum communication cost. Due to the constraints described below, this may not always be feasible.

A. Constraint 1: QoS Requirements

Continuous queries have to satisfy certain QoS requirements with regards to performance, memory usage, and accuracy. In this work we only consider the QoS performance metrics which is tuple latency. Tuple latency is the amount of time it takes for a tuple to be processed by the DSMS including any wait time incurred. Thus, it is the duration from the time a tuple arrives at the leaf node of the operator tree to reach the output buffer of the root node. Note that, tuple latencies are applicable only for tuples that are used in the output computation. The tuple latencies in our work are computed as follows.

Let $p_i = \langle v_{i1}, v_{i2}, \ldots v_{in} \rangle$ be one such path in an operator tree where v_{ij}, $1 \leq j \leq n$, denote a vertex in the operator tree. Latency of v_{ij}, denoted by $latency(v_{ij})$, depends on the operator type, specific algorithm for computing the operator, the waiting time encountered in the queues, and the window size for the blocking operator. Latency of $(v_{ik}, v_{i(k+1)})$, denoted by $latency(v_{ik}, v_{i(k+1)})$, depends on the size of the results sent from vertex v_{ik} to vertex $v_{i(k+1)}$ and also on the bandwidth of the channel connecting v_{ik} to $v_{i(k+1)}$. The tuple latency along this path p_i, denoted by $latency(p_i)$, depends on the processing latency at each vertex and the communication latency at each link in this path.

$$latency(p_i) = \sum_{j=1}^{n} latency(v_{ij}) + \sum_{k=1}^{k=n-1} latency((v_{ik}, v_{i(k+1)}))$$

Let m be the number of operator paths in the set of operator trees. The total tuple latency of the system, denoted by $system\_latency$, adapted from [2], is computed as follows:

$$system\_latency = \sum_{i=1}^{m} w_i * latency(p_i)$$

where w_i is the ratio of the number of output tuples along path p_i to the number of output tuples from the set of operator trees. QoS requirements may be that the system latency be less than some given threshold value, denoted by *threshold*. Thus,

$$system\_latency \leq threshold$$

B. Constraint 2: Bandwidth of Links

We must also consider the communication cost that each node incurs while forwarding the results to the downstream nodes. Recall that $band(p, q)$ indicates the available directed bandwidth of link (p, q). Let $comm(p, q)$ be the communication bandwidth incurred by node p to forward its result to the operator at node q. $comm(p, q) \leq band(p, q)$. Suppose N_7 and N_4, shown in Fig. 3, are connected by a low bandwidth channel. In order to reduce the communication cost for transferring results from N_7 to N_4, the *select* operators must be applied at or before node N_7.

C. Constraint 3: Processing/Storage Capacity of Nodes

Since nodes are resource constrained, it is important to calculate the resource utilization of the nodes during query plan generation. We use the notation n_{ij} to signify that the operator tree vertex v_j is executing at the network node i. Recall that $cpu(i)$, $memory(i)$ indicate the available CPU and memory for node i. Let $proc(i)$, $mem(i)$ denote the processing cost, storage cost respectively incurred at node i due to the execution of the various operators. $proc(i) = \sum_j proc(n_{ij})$ and $mem(i) = \sum_j mem(n_{ij})$. Note that, $proc(i) \leq cpu(i)$ and $mem(i) \leq memory(i)$ at any given point of time. In the context of our example, node N_8 may not have the capacity to perform multiple *select* operations.

D. Constraint 4: Link and Node Availability

Link can be used if it is available and if the link stability exceeds a given threshold. Thus, link (i, j) can be used only if $avail(i, j) \neq 0$ and $ls(i, j) \geq t$, where t is some threshold for link stability.

A node cannot be used if it does not have enough power available to perform the operation. Recall that $power(i)$ represents the available power of node i. Let $pow(i)$ denote the power needed at node i to execute the various operators. $pow(i) = \sum_j pow(n_{ij})$ where $pow(n_{ij})$ is the power needed to compute operator $v_j.op$ at node i using the available implementation. Clearly, $pow(i) \leq power(i)$.

Node may also get disconnected due to link failures. Thus, node i can be used only if $\exists j \in \mathbf{N}, avail(i, j) \neq 0 \wedge ls(i, j) \geq t$ for some link stability threshold t.

For example, node N_4 and N_5 may not have direct connectivity at the given point of time. Consequently, for processing query $Q1$ we cannot use node N_5, although it is a powerful node. In such a case, an alternate route must be used.

E. Constraint 5: Operator Placement Restrictions

Continuous queries may be complex, involving various operators, such as, *select*, *project*, *join*, and various forms of aggregation operations. Not all these operators are supported at every node. Consider the blocking operators, like *join* or *count*. These operators require a set of stream tuples to be stored before the operations can be applied. Some nodes may not have enough storage capability for processing these operations. Similarly, *join* processing often are computation intensive and cannot be supported by many nodes due to resource constraints. For vertex v_j of the operator tree to be executed at node i, $v_j.op \in oper(i)$. In the context of our example, node N_8 may not support aggregate operation *count* which may force a part of the tree to be executed at the other nodes.

IV. Operator Distribution and Migration

In a heterogeneous dynamic environment, the query plan manager is responsible for distributing the operators to the various nodes and migrating the operators when the topology of the network changes. The query plan manager is typically allocated in static nodes having good processing, storage, and communication capabilities. We can have various architectures of the query plan manager. The simplest architecture is where the query plan manager is located on a single node which may be the base station. In this case, the query plan manager has complete knowledge of the network topology at any given time. It is aware of the various nodes of the network, their processing and storage capabilities, and types of operations they can support. It also knows about the links of the network and their bandwidth. The query plan manager also knows the source nodes which receive the input streams and the sink nodes that produce the outputs. The query plan manager is aware of the arrival pattern of the streams and has other statistical information that are needed to calculate the cost of the query.

Query plan manager can also be implemented in a distributed manner using a set of nodes with sufficient computation, communication, and storage resources. Thus, the role of the query plan manager can be performed by multiple cluster heads of the network each of which controls a network partition. The cluster heads periodically exchange information to learn about the topology of the network and the input and derived streams associated with each partition.

In the distributed setting, an operator tree corresponding to one or more queries can be submitted to any partition. The operator tree is first forwarded to the partition that contains the maximum number of source vertices for the query. The local query manager is responsible for executing part of the operator tree; it generates the local query plan involving nodes and links in its own partition. The rest of the operator tree and the partial results are streamed to the query manager of the partition that contains the streams needed to do the next step of the computation. When a single partition does not have enough resources, it forwards the unfinished operator tree and the input streams to its neighboring cluster head having the available resources. If the local query plan manager has all the data streams and available resources to compute the query, it will execute it locally without communicating with the other cluster heads.

The query plan manager receives the set of operator trees corresponding to the submitted queries. It uses simple heuristics to transform them to reduce resource usage. One heuristic is to reorder the *select* operations whenever possible such that the operator with the lowest selectivity is applied first. A second heuristic is to merge the *project* operators operating on the same stream, each of which eliminates a single attribute, into one projection eliminating multiple attributes. Other heuristics can be applied as well, such as reordering of *select* and *project*, such that the size of the output is minimized. Application of such heuristics will transform the operator trees into alternative forms with lower communication and processing overheads.

The initial task distribution involves mapping these operator trees on to the network such that the constraints outlined in the above section are satisfied and the cost of the bandwidth utilization is minimized. The first step is to map each operator tree on to the network, such that the leaf vertices map to the nodes generating the streams and the root vertices map to nodes requiring the output. The challenge is to map the remaining vertices on the nodes of paths from the sources to the sinks.

We start the mapping for the operator tree which involves the maximum number of queries. Typically, there are many paths from a source to a sink. We eliminate those nodes where the available power, storage and bandwidth capacity are below a certain threshold. We next eliminate those links where the bandwidth or the link stability is below a given threshold. We place the select and project operators on the nodes as close to the source as possible, since these reduce the number and size of tuples respectively. We choose nodes with the greatest processing and storage capabilities and map the vertices containing join operators to these nodes. Whenever possible, we try to map multiple closely connected vertices on the same node, so as to reduce the communication costs. We iteratively allocate the remaining operator trees in a similar manner.

Once the initial allocation of tasks to nodes have been performed, we may have to migrate the tasks to other nodes in order to accommodate various types of changes. A change in the network topology may necessitate a change of the query plan. The arrival rate of elements of streams or arrival of new queries may also trigger a change in the query plans. When the query plans are changed, we may have to migrate operators from existing nodes to other ones. Note that, the size of outputs from *select*, *project*, *count*, *sum* is typically smaller than the size of the input. Consequently, we try to migrate these operators as close as possible to their upstream nodes. We try to move *join* operators as close as possible to their downstream nodes if its output size is greater than its input size. Migrating non-blocking operators, such as *select* and *project*, is relatively easy; we need to redirect the input and output streams only to the appropriate node. Migrating blocking operators, such as *join*, *sum* and *count*, is more complex; the state information (how many elements of the stream in the given window are yet to be processed) must also be transferred to the new node. Consequently, we try to avoid migrating these operators.

V. Conclusion

Some critical situation monitoring applications are executed over a dynamic and heterogeneous network where the nodes and links have differing capacities and the topology of the network is subject to change. We have discussed some challenges and proposed some ideas with respect to generating and adapting query plans in this environment. A lot of work remains to be done. We plan to develop and implement the algorithms on query sharing, multi query plan generation, and operator migration. We plan to develop a test bed where we can study the effects of various multi query plans and the different types of operator migration. We plan to see the effectiveness of the proposed heuristics and also need to evaluate how sensitive the multi query plans are with respect to the various constraints in the network.

References

[1] S. B. Zdonik, "Monitoring and Mining Data Streams," U.S. Army Medical Research and Materiel Command, Maryland, Annual Report DAMD17-02-2-0048, 2004.

[2] S. Chakravarthy and Q. Jiang, *Stream Data Processing: A Quality of Service Perspective Modeling, Scheduling, Load Shedding, and Complex Event Processing*, ser. Advances in Database Systems. Springer, 2009, vol. 36.

[3] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "NFM^i: An Inter-domain Network Fault Management System," in *Proc. of the ICDE*, Tokyo, Japan, Apr. 2005, pp. 1036–1047.

[4] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik, "Retrospective on Aurora," *VLDB Journal: Special Issue on Data Stream Processing*, vol. 13, no. 4, pp. 370–383, 2004.

[5] Q. Jiang, R. Adaikkalavan, and S. Chakravarthy, "MavEStream: Synergistic Integration of Stream and Event Processing." in *IEEE International Workshop on Data Stream Processing, ICDT*, Jul. 2007.

[6] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems." in *Proc. of the PODS*, June 2002, pp. 1–16.

[7] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik, "Scalable Distributed Stream Processing," in *Proc. of the CIDR*, 2003.

[8] Q. Jiang and S. Chakravarthy, "Anatomy of a Data Stream Management System," in *ADBIS Research Communications*, 2006.

[9] D. J. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J.-H. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, N. Tatbul, Y. Xing, and S. B. Zdonik, "The Design of the Borealis Stream Processing Engine," in *Proc. of the CIDR*, 2005, pp. 277–289.

[10] D. J. Abadi, S. Madden, and W. Lindner, "REED: Robust, Efficient Filtering and Event Detection in Sensor Networks," in *Proc. of the VLDB*, 2005, pp. 769–780.

[11] M. Daum, "Deployment of Global Queries in Distributed and Heterogeneous Stream Processing Systems," in *Proc. of the DEBS Doctoral and PhD Workshop*, 2009.

[12] N. Pollner, M. Daum, F. Dressler, and K. Meyer-Wegener, "An Overlay Network for Integration of WSNs in Federated Stream Processing Environments," in *Proc. of the Med-Hoc-Net*, 2011, pp. 157–164.

[13] P. Pietzuch, J. Ledlie, J. Shneidman, M. Roussopoulos, and M. W. amd M. I. Seltzer, "Network-Aware Operator Placement for Stream-Processing Systems," in *Proc. of the ICDE*, Atlanta, GA, Apr. 2006.

[14] M. Strübe, M. Daum, R. Kapitza, F. Villanueva, and F. Dressler, "Dynamic Operator Replacement in Sensor Networks," in *Proc. of the MASS*, 2010.

[15] S. Chakravarthy, M. Kumar, S. Madria, and W. Naqvi, "Middleware Architectures and Services," Air Force Research Laboratory, Rome, NY, Tech. Rep. FA8750-09-2-0199, Feb. 2012.