

# Toward Massive Query Optimization in Large-Scale Distributed Stream Systems

Yongluan Zhou<sup>1</sup>, Karl Aberer<sup>2</sup>, and Kian-Lee Tan<sup>3\*</sup>

<sup>1</sup> University of Southern Denmark

<sup>2</sup> EPFL, Switzerland

<sup>3</sup> National University of Singapore

**Abstract.** Existing distributed stream systems adopt a tightly-coupled communication paradigm and focus on fine-tuning of operator placements to achieve communication efficiency. This kind of approach is hard to scale (both to the nodes in the network and the users). In this paper, we propose a fundamentally different approach and present the design of a middleware for optimizing massive queries. Our approach takes the advantages of existing Publish/Subscribe systems (Pub/Sub) to achieve loosely-coupled communication and to “intelligently” exploit the sharing of communication among different queries. To fully exploit the capability of a Pub/Sub, we present a new query distribution algorithm, which can adaptively and rapidly (re)distribute the streaming queries at runtime to achieve both load balancing and low communication cost. Both the simulation studies and the prototype experiments executed on PlanetLab show the effectiveness of our techniques.

**Keywords:** Distributed Stream Systems, Publish/Subscribe Systems, Query Optimization, Load Balance, Overlay Network.

## 1 Introduction

There is a recently emerging demand for large-scale and widely distributed stream processing systems. Below is an example scenario, which is also the application context of this paper.

With the rapid development of sensor network technologies, more and more sensor networks are being deployed by many different organizations, such as research institutes and governments etc., to monitor and study our surrounding environment. The SensorScope project at EPFL (<http://sensorscope.epfl.ch>) is one such example. One can imagine that a stream processing system would be installed locally at each deployment to perform real-time data collection and analysis. It is desirable to pose queries involving multiple deployments across the border of countries and even continents. This demands a large-scale and

---

\* Kian-Lee Tan is partially supported by research grant R-252-000-237-112 from the National University of Singapore.

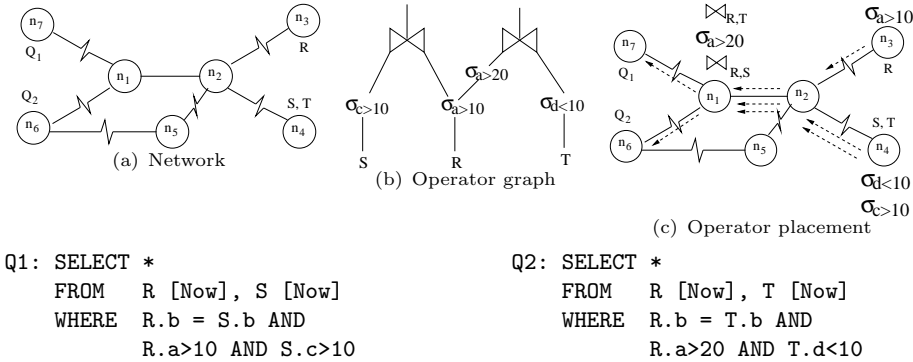


Fig. 1: Operator placement

loosely-coupled architecture to exploit the autonomous and distributed stream systems to provide a global stream processing service.

While such a system is desirable, two problems should be carefully considered. First, the communication cost could be very high as it may involve inter-country and even inter-continental communication. Moreover, streams are typically of a very high rate and have to be transferred continuously. In comparing to the abundant processing power provided by the large number of servers, network bandwidth is the bottleneck in such a context. Hence, it is critical to perform query optimization to achieve high communication efficiency.

Second, for such an autonomous system, it is desirable to adopt a loosely-coupled communication architecture, where data sources can just push their data to the network without keeping track of the destinations, and a data consumer can retrieve data of its interest without knowing the location of the sources.

### 1.1 Existing Distributed Stream Systems

In existing distributed stream systems [1, 3, 13, 17, 18], the communication between data sources and consumers adopts the tightly-coupled client-server paradigm. A node directly connects to the sources to get the streams it wants. As mentioned, this is not desirable in our context.

Furthermore, operator placement algorithms [1, 3, 13, 17] are often employed to optimize communication efficiency. Such schemes typically adopt a two-phase optimization algorithm, which resembles the earlier work on query optimization for distributed database systems, such as [22]. In the first phase, all the user queries are collected to a central place and then a global operator graph is generated. In the second phase, optimization algorithms are run to distribute the operators to minimize the communication cost [3, 17]. Let us look at an illustrating example. Figure 1(a) is an example network composed by seven nodes. Nodes  $n_7$  and  $n_6$  issue two queries  $Q_1$  and  $Q_2$  (written in CQL [23]) respectively. The

first phase optimization generates a global operator graph for these two queries as shown in Figure 1(b). Then the second phase may place the operators as depicted in Figure 1(c).

While this approach is effective, it assumes that an optimized global operator graph is available. So far, it lacks scalable algorithm to generate a good global operator graph. It is even harder to maintain the global operator graph if new queries were admitted and old queries were terminated frequently. In addition, the operator placement algorithm assumes the knowledge of the underlying overlay network. This tightens the coupling between the network layer and the application layer which may not be desirable for a large-scale system.

## 1.2 Pub/Sub Systems

Looking from a different angle, we observe that the above optimization can be divided into three sub-tasks. (1) Avoid duplicate data transfer. In the above example, both  $Q_1$  and  $Q_2$  are interested in stream  $R$ . It is desirable to send the data of  $R$  only once over each link along the path to the destinations. This can be done by sharing the data access of stream  $R$  of the two queries. (2) Perform early data filtering. In the example, this is done by allocating the selection operators close to the source nodes of the streams. (3) Place the query operators in proper places. The placement should consider the data rate of the operators' input and output as well as the common data interest among the different operators. For example, in Figure 1(c), the two join operators are allocated to the same place. Hence they can share the bandwidth consumption of transferring data from stream  $R$ . Otherwise, if, for example, we place one of the join operators to node  $n_5$  instead, then extra bandwidth will be consumed. Furthermore, if their output rates are much higher than their input rates, then it might be more beneficial to place them at their respective destinations (i.e.  $n_6$  and  $n_7$ ).

It is interesting to note that the first two sub-tasks have been solved nicely in the literatures of Distributed Publish/Subscribe systems [2, 7, 16] or Content-Based Networking (CBN) [10]. In these systems, messages are routed based on their contents as well as the interest profiles of nodes rather than the IP addresses of the destinations. Each message is represented as a set of attribute/value pairs. The interest profile (or subscription) of a node is specified as constraints on the attribute values. Only those messages whose attribute values satisfy the constraints will be sent to that node.

We illustrate more details by a scenario in an example Pub/Sub: Siena [7]. First, the data source  $n_3$  in Figure 2(a) advertises the data that it provides through a multicast tree. The advertisement has similar data structure as a subscription and specifies the constraints that the messages produced by the source will satisfy. Then every node knows what kind of messages will be sent from its neighbors. Now, the data receivers ( $n_6$  and  $n_7$  in Figure 2(b)) can multicast their subscriptions under the guidance of the advertisements of the data sources. At  $n_1$  the two subscriptions are merged before being propagated to  $n_2$ .  $n_2$  only propagates the subscription to  $n_3$  based on the advertisement

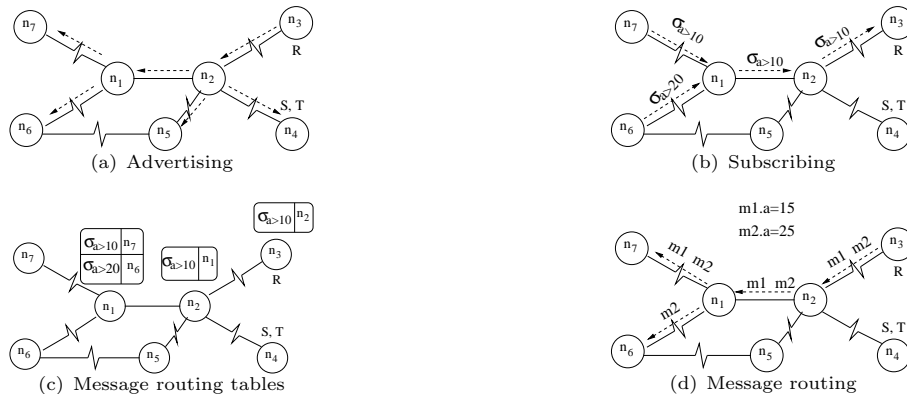


Fig. 2: Distributed Pub/Sub Systems

information. After the subscription propagation, a routing table is built at each node as shown in Figure 2(c). When a message is produced, the source just send it to the neighbor who is interested in the message. Figure 2(d) shows how two example messages are routed in the network.

It can be seen that a Pub/Sub inherits the advantage of the multicast communication paradigm where a message is sent over each link at most once. Furthermore, the messages are filtered as soon as possible on the way to the interested parties. More importantly, this is done without global planning. Finally, data sources and destinations in a Pub/Sub are loosely-coupled as they need not keep track of each other. As mentioned, this is desirable in our context.

### 1.3 COSMOS

Based on the above observation, we intend to build our distributed stream processing system by using a Pub/Sub as the communication substrate to achieve loosely-coupled communication. We design a middleware, called COSMOS (CO-operated and Self-tuning Management Of Streaming data), which perform query optimization by leveraging the underlying Pub/Sub to accomplish the searching of common data interest of the queries and the global tuning of the placement of filters along the overlay paths (from the sources to the destinations).

Now what is left behind is the third sub-task of the optimization: placing the query operators in the proper locations. To do so, a new query distribution scheme is proposed.

In this paper, we distribute the query loads in the unit of queries (instead of operators) to reduce the complexity of the problem and make the adaptation algorithms run faster at runtime. Furthermore, allocating operators of a query to multiple nodes would require synchronizations among the nodes, including the synchronizations during query processing as well as those during query insertions

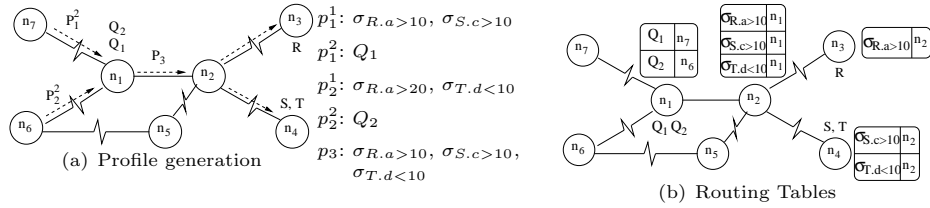


Fig. 3: COSMOS

and removals. This not only impairs the scalability of the system but also hard to be implemented in a loosely-coupled and autonomous system.

Our query distribution scheme distinguishes itself in several aspects: (1) It is more scalable. It does not require global planning to generate a global operator graph. Moreover, new hierarchical techniques are also employed to enhance the algorithm’s scalability. (2) It takes the communication characteristics of a Pub/Sub into consideration, which has not been explored before. (3) It targets both load-balancing and minimizing communication cost, while existing approaches [3, 20, 17, 15, 24, 21] only focus on either one of them. (4) It stresses the problem of fast arrival and removal of queries, which is also often overlooked in most existing work.

Figure 3(a) shows an example distribution with both  $Q_1$  and  $Q_2$  distributed to  $n_1$ . Then,  $n_1$  will generate two subscriptions for each query. For example, for  $Q_1$ , one subscription  $p_1^1$  is generated to retrieve the source data to feed  $Q_1$ . Furthermore, another subscription  $p_1^2$  is generated and sent to the query originator  $n_7$ .  $p_1^2$  is inserted into the Pub/Sub by  $n_7$  to receive the result stream of  $Q_1$ . Similarly  $p_2^1$  and  $p_2^2$  are generated for  $Q_2$ . At  $n_1$ , the two subscriptions  $p_1^1$  and  $p_2^1$  will be merged into a subscription  $p_3$ , which will then be inserted into the Pub/Sub. Figure 3(b) shows the routing tables at each node after all the subscriptions are inserted. Finally, queries are evaluated at  $n_1$  when the data are received from the underlying Pub/Sub. The results will then be transferred by the Pub/Sub to  $n_6$  and  $n_7$  respectively.

In short, we propose a fundamentally different query optimization approach to achieve communication efficiency, which does not require the maintenance of a global operator graph. It leverages Pub/Sub to eliminate duplicate data transfer, to perform early data filtering and to achieve loose-coupling of data sources from data consumers.

#### 1.4 Paper Layout

The rest of the paper is organized as follows. We first present an overview of the system in Section 2. The load distribution scheme is presented in Sections 3. Section 4 presents the results of a performance study of the load distribution scheme. Section 5 concludes the paper.

## 2 System Overview

The whole system consists of a number of, say  $N$ , distributed processors interconnected with a widely distributed overlay network. In addition, a number of data sources continuously publish their data to the network through the processors. A user first connects to a processor, which works as the proxy for him. In this case, the user and the proxy are said to be local to each other. User queries, specified in an SQL-like language similar to CQL [23], are submitted through their proxies. The proxy is also responsible for retrieving the result stream and sending it back to the user. For simplicity, only continuous queries are considered and no stored tables are involved. The query is first passed to the COSMOS middleware, which places the query at an appropriate processor to optimize the system performance. This paper focuses on solving this optimization problem.

The delivery of data streams are handled by the Pub/Sub middleware, which is assumed to support subscriptions similar to those in Siena [7]. Below, we use an example to illustrate how COSMOS leverage the Pub/Sub component.

### 2.1 An Illustrating Example

Table 1 lists a few queries specified using CQL [23]. These queries are extracted and simplified from the typical snow drift monitoring tasks performed by the environmental scientists.

Let us first look at  $Q_3$ . Assume it is distributed to a processor, say  $n_1$ , by COSMOS. Then the COSMOS component at  $n_1$  generates two subscriptions. The first one,  $p_1^3$ , will be used by  $n_1$  to fetch the source data requested by  $Q_3$  via the Pub/Sub component. The content of  $p_1^3$  contains the following:

- A list of streams that are requested by  $Q_3$ :  $\mathcal{S} = \{S1, S2\}$ . This is used by the Pub/Sub to select the data based on their source stream.
- A list of requested data attributes:  $\mathcal{P} = \{S2.*\}$ . The Pub/Sub can perform projection of the unnecessary attributes as soon as possible to reduce the network traffic.
- A list of filters:  $\mathcal{F} = \{S1.snowHeight > 10\}$ . This will be used to perform early data filtering in the Pub/Sub.

The second subscription,  $p_2^3$ , is generated for the user to fetch the query result stream. To do so, a unique stream name is created for the result stream ( by using the unique identifier of the processor  $n_1$ , such as the IP address). Then  $p_2^3$  contains this stream name.

Assume another user submits another query  $Q_4$  to the system (the second query in Table 2), which is also allocated to processor  $n_1$ . We can generate the first subscription,  $p_1^4$ , in a similar way. Pub/Sub can automatically perform data communication sharing.

The tricky part is the second subscription,  $p_2^4$ . A naive way is to generate separate result stream for  $Q_4$ . Hence we can use the unique name of  $Q_4$ 's result stream to compose  $p_2^4$ . This situation is shown in Figure 4(a). The result streams

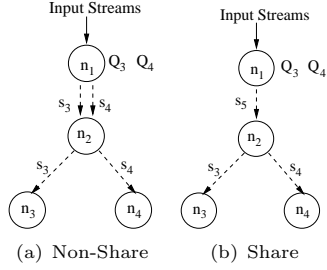


Fig. 4: Result stream delivery

$Q_3$ :	SELECT S2.* FROM Station1 [Range 30 Minutes] S1, Station2 [Now] S2 WHERE S1.snowHeight > S2.snowHeight S1.snowHeight ≥ 10
$Q_4$ :	SELECT S1.snowHeight, S1.timestamp, S2.snowHeight, S2.timestamp FROM Station1 [Range 1 Hour] S1, Station2 [Now] S2 WHERE S1.snowHeight > S2.snowHeight
$Q_5$ :	SELECT S2.*, S1.snowHeight, S1.timestamp FROM Station1 [Range 1 Hour] S1, Station2 [Now] S2 WHERE S1.snowHeight > S2.snowHeight

Table 1: Example queries

of  $Q_3$  and  $Q_4$ , i.e.  $s_3$  and  $s_4$  are transferred separately to the two users' proxies,  $n_3$  and  $n_4$ , respectively.

However, it can be easily seen that the result streams of  $Q_3$  and  $Q_4$  could have significant overlapping contents. Therefore these common contents have been transferred twice over the link between  $n_1$  and  $n_2$  in the above scheme. To further reduce the cost, we should exploit the sharing of result stream delivery.

At each site, if there are multiple queries with overlapping results, the COSMOS component will compose a new query  $Q$  whose result is the superset of the overlapping queries and only inserts this  $Q$  into the processing engine. In our example,  $Q_5$  (the third query in Table 1) would be created and inserted into the processing engine at node  $n_1$  instead of the two individual queries  $Q_3$  and  $Q_4$ . As shown in Figure 4(b),  $Q_5$  is run in  $n_1$  and its result stream  $s_5$  is sent to  $n_2$ , where it is “split” into two streams. The “splitting” is fulfilled by composing appropriate subscriptions for the users to retrieve their results. In our example they can be composed as follows:

- $p_2^3$ :  $\mathcal{S} = \{s_5\}$ ,  $\mathcal{P} = \{S2.*\}$ ,  $\mathcal{F} = \{-30(\text{minute}) \leq S1.timestamp - S2.timestamp \leq 0 \text{ AND } S1.snowHeight \geq 10\}$ .
- $p_2^4$ :  $\mathcal{S} = \{s_5\}$ ,  $\mathcal{P} = \{S1.snowHeight, S1.timestamp, S2.snowHeight, S2.timestamp\}$ ,  $\mathcal{F} = \{-1(\text{hour}) \leq S1.timestamp - S2.timestamp \leq 0\}$

To implement this approach, we extend traditional query containment and equivalence theorems to continuous window-based queries. Readers can refer to [25] for more details of this approach.

### 3 Query Distribution

In this section, the details of the query distribution algorithms in the COSMOS middleware is presented. For ease of exposition, it is assumed that a data source is also a processor in this paper. Hence, we refer to all the nodes in the network as processors. The word “data source” refers to those processors which are the origins of one or more source streams.

In the following subsections, we first present the theoretical model of the problem and then present the proposed solution.

### 3.1 Problem Modeling

In the problem model, we assume we do not have the knowledge of the overlay network topology of the Pub/Sub component. This is to achieve loose coupling between the components.

**3.1.1. Objectives.** Two objectives are considered in our algorithms:

- Balance the load among the processors. We assume the relative computational capability (the CPU speed) of a processor  $n_i$  is known and we quantify it as  $c_i$ . Furthermore, the load of a query is estimated as the CPU time that it will consume for every unit time in a processor with  $c_i = 1$ . Hence if the total query load is  $L$  and the total capability of the processors is  $C$ , the maximum load that can be allocated to a processor  $n_i$  is  $(1 + \alpha) \cdot c_i \cdot \frac{L}{C}$ . Parameter  $\alpha$  is added to allow slight load imbalance to trade for better communication efficiency. It is set to 0.1 in our experiments.

- Minimize the total communication cost. The communication cost can be divided into two parts: (1) transferring source streams from the sources to the processors; (2) transferring query results from the processors to the users. Similar to existing work [3, 17, 15], to measure the communication efficiency, we use the weighted unit-time communication cost  $\sum_{i,j} r(n_i, n_j) \cdot d(n_i, n_j)$ , where  $r(n_i, n_j)$  is the per-unit time traffic (bit/s) on the link between  $n_i$  and  $n_j$ , and  $d(n_i, n_j)$  is the transfer latency of the link.

To achieve both of the above two goals, the queries should be allocated onto the  $N$  processors such that the communication cost is minimized without violating the load constraints. To develop the algorithm, we model the problem as a graph mapping problem in the following subsection.

**3.1.2. Graph Mapping Model.** We first construct a network graph  $NG = \{V_n, E_n, W_n\}$ , where each vertex  $v_i \in V_n$  represents a processor in the network and there is one edge  $e_{ij} \in E_n$  between each pair of vertices  $v_i$  and  $v_j$ . The weight of each vertex  $v_i$  is given by  $W_n(v_i)$ .  $W_n(v_i)$  is equal to  $c_i$ , the processor's capability value. Furthermore, the weight of an edge  $e_{ij}$  is also given by  $W_n(e_{ij})$  and is equal to the communication latency between  $v_i$  and  $v_j$ . Figure 5(a) shows an example network graph. Here, there are two data sources,  $s_1$  and  $s_2$ , which have no computational capability (in terms of complex query processing) and two processors,  $n_1$  and  $n_2$ , have the same  $c_i$ .

Second, a *query graph*,  $QG = \{V_q, E_q, W_q\}$ , is constructed. There are two types of vertices in  $V_q$ : query vertex (q-vertex) representing a query and network vertex (n-vertex) representing a node in the network. An edge between a q-vertex and a n-vertex represents either a query requests source data from the data source or a query's result should be sent back to the proxy. In addition, if a query's data source and its proxy happen to be the same node, only one edge connects the query and that node. Figure 5(b) shows the query graph when four queries are submitted to the network of Figure 5(a). In the figure, there are four q-vertices, which are drawn in rectangles, and four n-vertices, which are drawn in circles.  $Q_1$  and  $Q_2$  request source data from  $s_1$  and  $s_2$  respectively and their results should be sent back to  $n_1$ .

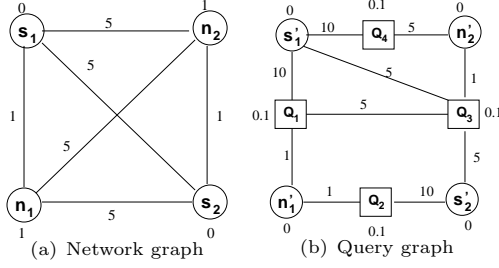


Fig. 5: Graphs

	Scheme	Load	WEC
Scheme 1	$Q_1, Q_2 \rightarrow n_1$ $Q_3, Q_4 \rightarrow n_2$ $s_i \rightarrow s'_i, n_i \rightarrow n'_i$	$n_1: 0.2$ $n_2: 0.2$	165
Scheme 2	$Q_1, Q_4 \rightarrow n_1$ $Q_2, Q_3 \rightarrow n_2$ $s_i \rightarrow s'_i, n_i \rightarrow n'_i$	$n_1: 0.2$ $n_2: 0.2$	115
Scheme 3	$Q_1, Q_3 \rightarrow n_1$ $Q_2, Q_4 \rightarrow n_2$ $s_i \rightarrow s'_i, n_i \rightarrow n'_i$	$n_1: 0.2$ $n_2: 0.2$	110

Table 2: Mapping Schemes

In a query graph, each q-vertex is weighted with the estimated query load, while n-vertices are assigned with zero weights. In addition, each edge is weighted with the estimated data rate (bit/s) of the corresponding streams. For example, in Figure 5(b),  $Q_1$ 's load is of value 0.1. In addition, it requests 10 bit/s data from source  $s_1$  and generates 1 bit/s result streams to  $n_1$ .

However, the above model is still not enough for our problem. It ignores the sharing of data communication among queries in a Pub/Sub. To accurately model the communication cost, we add one edge between each pair of queries that have overlap in their data interest. The edge weight is equal to the rate of the data that are of interest to both of its end vertices (queries). The intuition is to penalize allocation schemes that distribute the two queries to two nodes that are very far away from each other. In Figure 5(b), the data requested by  $Q_1$  from  $s_1$  happens to contain those of  $Q_3$ . So the weight of the edge between  $Q_1$  and  $Q_3$  is equal to the one between  $s_1$  and  $Q_3$ .

Now, we can model the query distribution problem as a graph mapping problem which maps the vertex set of one graph to the vertex set of another graph. A mapping from a vertex set  $V_1$  to another vertex set  $V_2$  is defined as a boolean function  $M(v_i, v_j)$ , where  $v_i \in V_1$  and  $v_j \in V_2$ , under the constraint that for each  $v_i \in V_1$  there is exactly one  $v_j \in V_2$  such that  $M(v_i, v_j) = true$ . The formal problem statement is as follows:

Given a query graph  $QG = (V_q, E_q, W_q)$  and a network graph  $NG = (V_n, E_n, W_n)$ , find a mapping  $M$  from  $V_q$  to  $V_n$ , such that the mapping

1. **obeys network constraint:** an n-vertex  $v_i$  in  $V_q$  is mapped to a vertex  $v_j$  in  $V_n$  which represents the same network node as  $v_i$ ;
2. **and obeys load-balancing constraint:**

$$\forall v_j \in V_n, \sum_{\substack{v_i \in V_q \\ M(v_i, v_j)}} W_q(v_i) \leq (1 + \alpha) \cdot W_n(v_j) \cdot \frac{W_q^v}{W_n^v}, \quad (3.1)$$

where  $W_q^v = \sum_{v_i \in V_q} W_q(v_i)$  and  $W_n^v = \sum_{v_j \in V_n} W_n(v_j)$ ;

3. *and minimizes the Weighted Edge Cut (WEC): which is given by*

$$WEC = \sum_{\substack{v_k \in V_n \\ v_l \in V_n}} \sum_{\substack{v_i \in V_q \\ v_j \in V_q \\ M(v_i, v_k) \\ M(v_j, v_l)}} W_q(e_{ij}) \cdot W_n(e_{kl}). \quad (3.2)$$

In Table 2, we present three mapping schemes from the query graphs to the network graphs in Figure 5, which obey both the network constraint and the load-balancing constraint. In scheme 1, we map all the queries to their own local processors, while scheme 2 is the optimal mapping if we ignore the potential sharing of communication of  $Q_1$  and  $Q_3$ . We can see that scheme 3 is a better mapping, which has a smaller *WEC* value.

### 3.2 Challenges and Approach Overview

There are a few practical difficulties to solve this problem. First, it is hard to construct the global network graph and query graph when the size of the network and the number of queries scales up. A scalable algorithm is required. Second, even if we have the global graphs, finding the optimal mapping is an NP-Hard problem [19]. Hence, an efficient heuristic-based approach is needed. Third, the queries and stream statistics could change over runtime. A runtime algorithm is required to redistribute the queries.

To address the problems, distributed coordinators are employed to perform the heuristic graph mapping and remapping algorithms. They are organized into a hierarchical tree. Each leaf coordinator constructs a network (sub)graph which consists of an exclusive set of processors while a parent coordinator constructs a network (sub)graph composed by its child coordinators. This provides a hierarchical view of the network graph. On the other hand, each coordinator also holds a query (sub)graph which is a coarsened overview of its descendants' and this constructs a query graph hierarchy. Each coordinator only performs the mapping and runtime remapping of its query (sub)graph to its network (sub)graph. The rest of this section presents the detail of our scheme.

Finally, it is required to frequently estimate the overlaps between a pair of queries in the following algorithms, which could be very expensive if it is done by semantical reasoning. Therefore, we partition each stream into a number of substreams, and represent each query's data interest as a bit vector indicating whether a substream overlaps with its interest. In this way, efficient bit operations could be used to quickly perform the estimation.

### 3.3 Network Graph Hierarchy

The coordinators are a subset of processors chosen from all the processors in the system. Each such processor performs two separate logical roles: the stream processor and the coordinator. We assume that separate resources of these processors are reserved for these two roles. For non-coordinators, they perform only

---

**Algorithm 1:** Query graph coarsening algorithm

---

```
1 while  $|V| > v_{max}$  do
2   Set all the vertices as unmatched;
3   while  $\exists$  unmatched vertices  $\wedge |V| > v_{max}$  do
4     Randomly select an unmatched vertex  $u$ ;
5      $A \leftarrow adj(u) - mat(adj(u))$ ;
6     if  $is\_n(u)$  then
7        $A \leftarrow A - \{v | v \in adj(u) \wedge is\_n(v) \wedge (u.clu \neq v.clu \vee v.clu = unknown)\}$ ;
8       Select a vertex  $v$  from  $A$  such that the edge  $e(u, v)$  is of the maximum
9       weight;
10      Collapse  $u$  and  $v$  into a new vertex  $w$ ;
11      Set  $w$  as matched;
12       $w.weight \leftarrow u.weight + v.weight$ ;
13      Re-estimate the weights of the edges connected to  $w$ ;
14      if  $is\_n(u)$  OR  $is\_n(v)$  then
15         $is\_n(w) \leftarrow true$ ;
16         $w.clu = is\_n(u)?u.clu : v.clu$ ;
```

---

the stream processor role. Hereafter, the words “processor” and “coordinator” refer to the logical roles.

The coordinators are organized into a hierarchical tree. At the bottom level, each processor forms a separate cluster and the processor is also called the parent of this cluster. At the second level, the processors are clustered into multiple close-by (in terms of transfer latency) clusters. Within each cluster, the median is selected as the coordinator of the cluster which is also called the cluster’s parent. The median of a set of processors  $\{n_1, n_2, \dots, n_l\}$  is defined as the processor  $n_i$  with minimum total transfer latency to all processors in the cluster, i.e.  $\sum_{1 \leq j \leq l} d(n_i, n_j) \leq \sum_{1 \leq j \leq l} d(n_k, n_j)$  for any  $n_k$ . These coordinators are also clustered level by level in a similar way. We say a processor belongs to a cluster of an internal coordinator (at any level) if it is the descendant of this coordinator.

Each coordinator constructs a network subgraph containing only its child coordinators (or child processors for the leaf coordinators). Here, the weight of a vertex is equal to the total capability values of all its descendant processors.

We adapt schemes proposed by the networking community to construct a hierarchical tree of coordinators, such as [5]. The mechanism in [5] tries to maintain a tree with the following properties: (1) the size of the cluster in each level is between  $k$  and  $3k - 1$  (except the cluster of the root whose size could be less than  $k$ ); (2) the parent is the median of its cluster. The tree is constructed incrementally and dynamically. Interested readers can refer to [5].

### 3.4 Query Graph Hierarchy Construction

In this subsection, we look at how to construct the query graph hierarchy. To begin, each leaf coordinator collects the query specifications from its child nodes

and generates a query graph over them. If the number of vertices of the query graph is larger than  $v_{max}$ , then it runs Algorithm 1 to coarsen the query graph. The graph mapping algorithm at each coordinator, which will be presented in the following sections, is performed on this coarsened query graph. The coarsening algorithm repeatedly collapses two selected vertices until the number of vertices is smaller than or equal to  $v_{max}$ . In the algorithm, a vertex  $u$  tends to collapse with a neighbor  $v$  which has an edge  $e_{u,v}$  with a larger weight, because these two vertices are more likely to be mapped to the same vertex in the network graph. For ease of exposition, we define the following functions: (1)  $adj(u)$  returns the set of adjacent vertices of  $u$ ; (2)  $is\_n(u)$  returns true if  $u$  is an n-vertex; (3)  $matched(A)$  is all the matched vertices in a vertex set  $A$ . In addition, for each n-vertex  $u$ , a field  $clu$  indicates which child cluster of the current coordinator covers  $u$ . Two n-vertices belong to two different child clusters shall not be merged together because they have to be mapped to different child clusters in the graph mapping algorithm. Note that if  $u$  is not covered by any child cluster of this coordinator, then their  $clu$  field is set as unknown.

The q-vertices in the (coarsened) graph are tagged with the current coordinator's name and then submitted to the parent coordinator who will perform the same procedure after receiving all the (coarsened) graphs from its children. Note that the procedure is run in parallel in different subtrees to accelerate the whole procedure. The procedure stops when the root gets the (coarsened) query graph. Now every coordinator holds its query graph. Finally, each coordinator periodically propagates the update of its query graph to its parents at runtime.

### 3.5 Initial Query Distribution

Once the initial query graph hierarchy is constructed, the root coordinator starts mapping its (coarsened) query graph to its network (sub)graph. The query subgraph mapped to each child is uncoarsened one level back and sent to the child. This procedure repeats at each level until all the queries are assigned to the processors. Note that, to uncoarsen a vertex, information of the finer-grained vertices, if necessary, is retrieved from the corresponding coordinator based on the tags of the vertex.

The algorithm is illustrated in Algorithm 2. It starts by using a greedy algorithm to get an initial mapping:

- (a) Map each n-vertex to a child that manages the node that n-vertex represents.
- (b) Map the q-vertices one by one in descending order of their weights. For each q-vertex, among the children that can accommodate it (i.e. their load-balancing constraints will not be violated after mapping the q-vertex to anyone of them), map it to the one that minimizes the current WEC. If no children can accommodate it, then map it to the one with the minimum violation of the load-balancing constraint.

Note that finding a mapping that satisfies the load-balancing constraint is an NP-Complete problem. Our algorithm does not guarantee finding such a mapping.

---

**Algorithm 2:** Graph mapping algorithm

---

**Input:**  $NG = (V_n, E_n, W_n), QG = (V_q, V_q, W_q)$

- 1 use a greedy algorithm to get the initial mapping;
- 2 compute the gain  $gain(v_i, v_j)$  for each q-vertex  $v_i \in V_q$  and each  $v_j \in V_n$  ;
- 3  $minWEC \leftarrow$  current WEC;  $minMapping \leftarrow$  current mapping;
- 4 **repeat**
- 5     current mapping  $\leftarrow minMapping$ ;
- 6     **repeat**
- 7          $maxGain \leftarrow -\infty$ ;  $vertexToRemap \leftarrow \emptyset$ ;  $vertexToRemapTo \leftarrow \emptyset$ ;
- 8         **for** each  $v_j \in V_n$  **do**
- 9             Find an unmatched q-vertex  $v_i \in V_q$  currently mapped to  $v_j$  and a vertex  $v_k \in V_n$ ,  $gain(v_i, v_k)$  is maximized and remapping  $v_i$  to  $v_k$  does not violate load-balancing or improves a violation (if any);
- 10             **if**  $gain(v_i, v_k) > maxGain$  **then**
- 11                  $maxGain \leftarrow gain(v_i, v_k)$ ;  $vertexToRemap \leftarrow v_i$ ;
- 12                  $vertexToRemapTo \leftarrow v_k$ ;
- 13             **if**  $vertexToRemap \neq \emptyset$  **then**
- 14                 set  $vertexToRemap$  as matched;
- 15                 remap  $vertexToRemap$  to  $vertexToRemapTo$ ;
- 16                 update  $gain(v_i, v_k)$  for any  $v_i$  directly connected to  $vertexToRemap$ ;
- 17                 **if**  $current\ WEC < minWEC$  **then**
- 18                      $minWEC \leftarrow$  current WEC;
- 19                      $minMapping \leftarrow$  current mapping
- 20     **until**  $vertexToRemap = \emptyset$ ;
- 21 **until**  $minWEC$  is the same as the last iteration;

---

Lines 4-20 iteratively improve the mapping by trying to remap the q-vertices to other vertices in  $NG$ . Here, we use the value of  $gain(v_i, v_k)$  to heuristically guide our remapping, which is equal to the reduction of the WEC value by remapping  $v_i \in V_q$  to  $v_k \in V_n$ . To achieve some capability of climbing out of local minima, a q-vertex  $v_i$  with a negative  $gain(v_i, v_k)$  value would be considered for remapping as long as its gain value is the highest and its remapping will not violate the load-balancing constraint of  $v_k$ . The mapping with minimum WEC value will be restored at the beginning of each outer iteration.

### 3.6 Online New Query Insertion

Unlike prior work which assumes queries are relatively stable, our system stresses the problem of fast query streaming. The new queries have to be quickly distributed to the desirable processors. A good distribution can avoid runtime query migration at a later time (see the next subsection).

While there are many possible new query distribution schemes, in this paper, we only study the use of the hierarchical coordinator tree and show the significance of new query insertion for the system performance. In this scheme,

a new query is first routed to the root coordinator which then routes it to one of its children. The routing is done level by level until the query is assigned to a processor. At each coordinator, the query is added to the query graph and the weights of the new edges are estimated. Then the new vertex is mapped to a vertex in the network graph such that the resulting WEC is minimized.

Although all queries have to be routed through the root coordinator, this scheme is scalable to very fast query streams. This is because it only needs to route the queries to a few children based on some coarse-grained information. As shown in Section 4, it can handle more than 800,000 queries per second in our experimental PC. For higher query stream rates, we can perform online routing only on some queries while simply keeping the other queries at their proxies. Further trade-offs between routing quality and routing efficiency is an interesting piece of future work.

### 3.7 Adaptive Query Redistribution

During runtime, the queries, the workload of processors and the characteristics of data streams may change. Hence the initial allocation of queries may become suboptimal. Thus adaptive adjustment of the query distribution has to be performed. Again we employ a hierarchical scheme. The adaptation works in rounds and each round is initiated by the root coordinator periodically. After making the redistribution decisions, the root coordinator would transfer the change of the distribution to each of its children. Each child coordinator retrieves the finer-grained information of the vertices newly allocated to it from their original coordinators. Then the child coordinators would perform the same procedure to make redistribution decisions. This process continues until the leaf coordinators are done with the redistribution. Note that the actual migration of queries happens after all decisions are made and is done among the processors.

The adaptive redistribution algorithm in each coordinator is composed of two phases: load re-balancing followed by distribution refinement. In the load re-balancing phase, the coordinator tries to re-balance the load among its children. Besides that, there are a few other goals to be achieved:

1. Minimize the WEC of the mapping.
2. Minimize the query migration time. Since migrating queries may incur the migration of stateful operators (e.g. join), we should minimize the size of the states to be moved.

In the load balancing phase, to avoid re-mapping from scratch, which may incur too many query migrations, we adopt a load diffusion approach [14]. A diffusion solution specifies the load  $m_{ij}$  that should be migrated from a coordinator  $n_i$  to another coordinator  $n_j$  for each  $(i, j)$  pair. Authors in [14] proposed a method to derive a diffusion solution such that the Euclidean norm of the transferred load is minimized which results in a small number of query migrations. Our redistribution algorithm is presented in Algorithm 3. As  $n$ -vertices are not considered for redistribution, the vertices in the algorithm only refer to the  $q$ -vertices. The benefit of remapping a vertex from  $n_i$  to  $n_j$  is defined as the

---

**Algorithm 3:** Adaptive load re-balance

---

```
1 begin
2   Compute the diffusion solution  $m_{ij}$  for every  $i, j$  pair;
3   while there exists an  $m_{ij} > 0$  do
4     Randomly select a pair  $i, j$  such that  $m_{ij} > 0$ ;
5      $V \leftarrow$  query vertices in  $n_i$  whose benefits differ up to  $x\%$  from the largest
      benefit;
6      $V_d \leftarrow$  the dirty query vertices in  $V$ ;
7     if  $V_d = \emptyset$  then  $V_d \leftarrow V$ ;
8     Remapping the vertex  $v \in V_d$  from  $n_i$  to  $n_j$  such that it is of the largest
      load density and  $m_{ij}$  is larger than 90% of its weight;
9 end
```

---

reduction of the WEC given by Eqn (3.2). To achieve good mapping quality, our algorithm tends to remap those vertices with large benefits.

Furthermore, a vertex is called *dirty* if it had been picked for remapping in the earlier iterations in the same adaptation round. We give these vertices higher remapping priority because moving them again would not increase the amount of query migration (Note that queries are actually moved after all the decisions are made in one round.). In addition, the *load density* of a vertex is equal to the weight divided by the size of its state. We favor remapping the denser ones because it may result in less state movement. The value of  $x$  in line 5 can be used to trade mapping quality for lower migration cost. With a larger  $x$  value, we can consider more vertices with lower migration benefit. In our experiments, we set  $x = 10$ .

The distribution refinement phase attempts to reduce the WEC while maintaining the load balancing condition. Again the query vertices are visited randomly and checked to see whether they belong to one of the following categories:

- (1) Mapping the vertex back to its original location can maintain load balance and the current WEC.
- (2) Mapping the vertex to another node can decrease the current WEC without violating load balance.

The checks are performed in the order given above. Whenever such a vertex is found, the remapping is performed.

### 3.8 Statistics Collection

Stream statistics are periodically multicast to the coordinators from the sources. As stated before, we partition the data streams into multiple substreams and the data interest of a user query is represented as a bit vector. Hence the stream statistics we need is the data rate of each substream. In addition, each processor periodically collects the average CPU time that each of its running queries consumes per unit time. If any value is changed, then it will be (re)submitted to the parent coordinator to (re)estimate the workload that the query may incur.

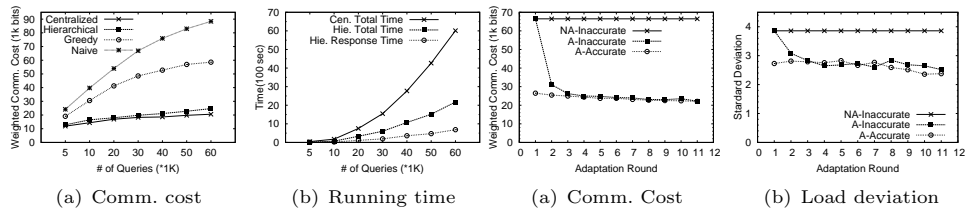


Fig. 6: Varied #queries

Fig. 7: Adapting to inaccurate statistics

## 4 Experiments

This section presents a performance study of the proposed techniques. Two sets of experiments are conducted. In the first one, simulations of a large scale distributed system and a huge query set are conducted to test the various performance aspects of COSMOS. In the second one, we deploy our system prototype over PlanetLab with a real data set to compare the performance of COSMOS with the state-of-the-art operator placement algorithms. All software are implemented in C/C++.

### 4.1 Simulation Study

A network topology with 4096 nodes is generated using the Transit-Stub model in the GT-ITM topology generator. Among these nodes, 100 nodes are chosen as the data stream sources, and 256 nodes are selected as the stream processors, and the remaining nodes act as the routers.

The default cluster size parameter  $k$  used in the coordinator tree construction is set to 4, which will be varied in the experiments. All the streams are partitioned into 20,000 substreams and they are randomly distributed to the sources. The arrival rate of each substream is randomly chosen from 1 to 10 (bytes/seconds). To simulate clustering effect of user behaviors,  $g = 20$  groups of user queries are generated and each group has different data hot spots. The group that a query belongs to is chosen randomly and the number of substreams that a query requests is uniformly chosen from 100 to 200. For the queries within every group, the probability that a substream is selected conforms to a zipfian distribution with  $\theta = 0.8$ . To model different groups having different hot spots, we generate  $g$  number of random permutations of the substreams. The number of queries are varied from 5,000 to 60,000 and we set their workload to be proportional to their input stream rates. The adaptive interval of the adaptive query redistribution algorithm is set to 200 seconds. Because the cost of transmitting the result streams from the processors to their local users are identical for any query distribution scheme, we subtract such cost from the reported figures to ease the comparison.

**4.1.1. Initial Query Distribution..** In the first experiment, we study the performance of the initial query distribution scheme with different number of

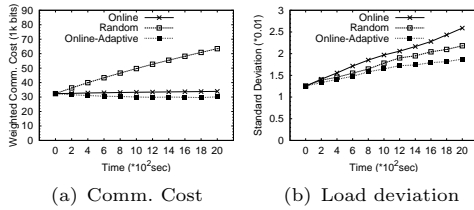


Fig. 8: New query arrival

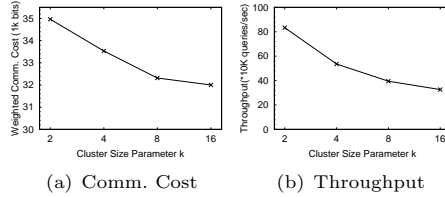


Fig. 9: Varied Cluster Size

queries. It is compared with three approaches: (a) Naive: allocate the queries to their local processors. (b) Greedy: only run the greedy algorithm in Algorithm 2. (c) Centralized: a centralized node constructs a global query graph and a global network graph, and runs Algorithm 2 to perform a global mapping. While this approach is limited in its scalability, it serves as a benchmark to examine the optimality of other approaches. Figure 6(a) presents the results of all the four approaches. Naive performs the worst because it cannot identify the data interest of the queries and optimize their locations. Greedy works a lot better. The two graph mapping algorithms perform the best and their performances are similar. This also verifies that the graph coarsening procedure in our hierarchical mapping algorithm does not incur much errors.

We also report the response time (i.e. the time interval from the begin to the end of the mapping) and the total time (i.e. the total CPU time consumed in all the coordinators) of the centralized and hierarchical graph mapping algorithms in Figure 6(b). Note that the response time and total time are equivalent for the centralized approach. It is shown that both the response time and total time of the hierarchical approach are much lower than the centralized one.

**4.1.2. Adaptive Query Distribution.** In the second set of experiments, we study the performance of the adaptation scheme. In the above experiments, the graph mapping algorithms perform well if accurate apriori statistics exist. However, apriori statistics are hard to collect in a large scale system. Hence, in the first experiment, we study the situation that the apriori statistics are inaccurate. We model this situation by using a random initial query allocation scheme. Three algorithms are compared: (1) NA-Inaccurate: non-adaptive algorithm with inaccurate statistics; (2) A-Inaccurate: adaptive algorithm with inaccurate statistics; (3) A-Accurate: Adaptive algorithm with accurate statistics. Figures 7(a) and 7(b) present the communication cost and the standard deviation of the system load over the observation period. It can be seen that the adaptive algorithm can gradually refine the initial query distribution scheme to minimize the communication cost and balance the system load.

In another experiment, we study the situation that new queries arrive in the system. Initially, there are 30,000 queries and new queries are added into the system incrementally at a 200 seconds interval. At the start of each interval, there are 1,500 new queries coming in. We reported the average communication

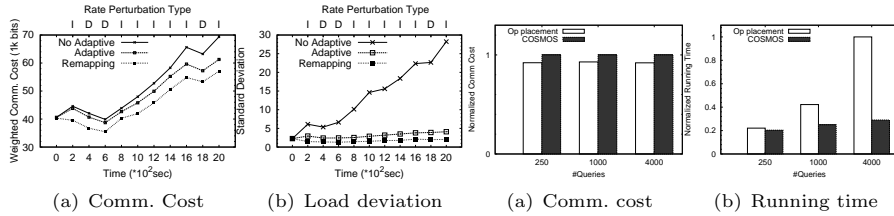


Fig. 10: Perturbation of stream rates

Fig. 11: Prototype study

cost during each interval and the standard deviation of the processor loads. Three schemes are compared: (1) Random: randomly allocate the new queries without considering their interest; (2) Online: use our online new query insertion algorithm; (3) Online-Adaptive: use both the online new query insertion and the adaptive query redistribution. The results are shown in Figure 8(a) and 8(b). The performance of Random gets worse with more queries added, while Online can maintain low communication cost but with increasing load imbalance. Online-Adaptive performs the best in both metrics because of its ability to re-balance the load distribution and to refine the query distribution.

In the fourth experiment, we examine the scalability of our system to fast query streams. The settings are similar to that of the above experiment. We collect the time for the root coordinator to distribute a query and then compute the maximum query rate that it can accommodate. We study the root coordinator because it is the potential bottleneck of the system. We vary the cluster size parameter  $k$ . The results are shown in Figure 9. We can see that, with a smaller value of  $k$ , the query distribution quality is worse. That is because there are more levels in the coordinator tree and more graph coarsening is performed. On the other hand, the throughput of query streams gets better with a smaller value of  $k$ . The reason is the root coordinator needs to route queries to fewer number of children. Hence, adaptively setting the parameter  $k$  is an interesting piece of future work.

Finally, we examine the situation when the rates of streams change. At run-time, we increase (denoted by “I”) or decrease (denoted by “D”) the rates of 800 random streams several times so that load imbalance exists within the system. Here, we compare the adaptive scheme with two schemes: (1) Re-mapping: use the centralized mapping algorithm to remap the global query graph to the global network graph; (2) Non-Adaptive: no adaptation is done. Figures 10(a) and 10(b) depict the communication cost as well as the standard deviation of the load in the system after each change. It is clear that adaptive query redistribution performs close to centralized remapping and can re-balance the system load to adapt to the new data characteristics without increasing the communication cost. While the remapping algorithm can achieve better results, it incurred about 7 times more query migrations than the adaptive algorithm did.

## 4.2 Prototype Study

In this experiment, our prototype system is deployed on 30 nodes over PlanetLab from different countries and continents. We use our stream processing system, GSN (<http://gsn.sourceforge.net>), which is tailored for processing data from heterogeneous sensor networks. Real readings from 100 sensors deployed in our SensorScope project (<http://sensorscope.epfl.ch>) are used as the dataset. 5 nodes act as the data sources, each with equal number of sensors. A number (250 ~ 4000) of random queries are generated. Each query contains one to three random selection predicates on the sensor readings and sensor types together with one to three join predicates on the timestamp. A random node is chosen as the proxy for each query.

In the operator placement approach, an algorithm similar to [12] is implemented to generate an optimized global operator graph. In addition, the algorithm proposed in [3] is also implemented to optimize the operator placement. In COSMOS, the coordinator tree is constructed such that each cluster has 2 ~ 3 members. Since [3] did not study adaptive query optimization, a static query set is used to compare the two approaches.

Figure 11(a) shows the communication cost of the query plans generated by the two approaches. To ease the comparison, we normalize the values over those of COSMOS. One can see that COSMOS can achieve similar communication efficiency as the existing operator placement algorithms with varied number of queries. The slight difference can be partially attributed to the fact that the operator placement algorithms in [3] do not consider load balancing and hence it can obtain a plan with lower communication cost. In Figure 11(b), we depict the response time of the two algorithms. In this figure, we normalize the values over the largest one (i.e. the response time of the operator placement algorithm with 4,000 queries) to see the trend with increasing number of queries. The result suggests that COSMOS is much more scalable than the existing operator placement algorithms with larger number of queries. This confirms the efficiency of the new system architecture and the hierarchical query placement algorithm.

## 5 Conclusion

This paper proposes a massive query optimization approach for distributed stream systems. A Pub/Sub is adopted as the communication substrate. Techniques are proposed to leverage the Pub/Sub to “intelligently” eliminate duplicate data transmission and perform early data filtering in a scalable way. Furthermore, a scalable load distribution scheme further improves the system’s performance. The load distribution problem is modelled as a graph mapping problem, which considers both load balancing and communication cost minimization and also takes account of the communication characteristics of a Pub/Sub. Both static and adaptive query distribution algorithms are proposed. A new hierarchical scheme is utilized to enhance the algorithms’ scalability. An extensive simulation study verifies the efficacy and efficiency of all the proposed techniques.

## References

1. D. J. Abadi et al. The design of the borealis stream processing engine. In *CIDR*, 2005.
2. M. K. Aguilera et al. Matching events in a content-based subscription system. In *PODC*, 1999.
3. Y. Ahmad et al. Networked query processing for distributed stream-based applications. In *VLDB*, 2004.
4. L. Amini et al. Adaptive control of extreme-scale stream processing systems. In *ICDCS*, 2006.
5. S. Banerjee et al. Scalable application layer multicast. In *SIGCOMM*, 2002.
6. D. Carney et al. Monitoring streams: A new class of data management applications. In *VLDB*, 2002.
7. A. Carzaniga et al. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.
8. A. Carzaniga et al. A routing scheme for content-based networking. In *INFOCOM*, 2004.
9. A. Carzaniga and A. L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, 2003.
10. A. Carzaniga and A. L. Wolf. Content-based networking: A new communication infrastructure. In *Infrastructure for Mobile and Wireless Systems*, 2001.
11. S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
12. J. Chen et al. NiagaraCQ: A Scalable Continuous Query System for Internet Databases. In *SIGMOD*, 2000.
13. M. Cherniack et al. Scalable distributed stream processing. In *CIDR*, 2003.
14. Y. F. Hu and R. J. Blake. An optimal dynamic load balancing algorithm. Technical report, Daresbury laboratory, 1995.
15. V. Kumar et al. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, 2005.
16. O. Papaemmanouil et al. Semcast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.
17. P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.
18. T. Repantis et al. Synergy: sharing-aware component composition for distributed stream processing systems. In *Middleware*, 2006.
19. K. Schloegel et al. Graph partitioning for high-performance scientific simulations. pages 491–541, 2003.
20. M. A. Shah et al. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.
21. U. Srivastava, et al. Operator placement for in-Network stream query processing. In *PODS*, 2005.
22. M. Stonebraker, et al. Mariposa: A New Architecture for Distributed Data. In *ICDE*, 1994.
23. The STREAM Group. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 2003.
24. Y. Xing et al. Dynamic load distribution in the borealis stream processor. In *ICDE*, 2005.
25. Y. Zhou et al. Rethinking the design of distributed stream processing systems. In *NetDB*. 2008.