

# Leveraging Distributed Publish/Subscribe Systems for Scalable Stream Query Processing

Yongluan Zhou, Kian-Lee Tan, and Feng Yu

National University of Singapore

**Abstract.** Existing distributed publish/subscribe systems (DPSS) offer loosely coupled and easy to deploy content-based stream delivery services to a large number of users. However, the lack of query expressiveness limits their application scope. On the other hand, distributed stream processing engines (DSPE) provide efficient processing services for complex stream queries. Nevertheless, these systems are typically tightly coupled, platform dependent, difficult to deploy and maintain, and less scalable to the number of users. In this paper, we propose a new architectural design for a scalable distributed stream processing system, which provides services to evaluate continuous queries for a large number of clients. It is built by placing a query layer on top of a DPSS architecture. In particular, we focus on solving the query distribution problem in the query layer.

## 1 Introduction

Emerging monitoring applications, such as financial monitoring, sensor network monitoring, network management etc., have fueled much research interest in designing distributed publish/subscribe systems (DPSS) [1, 5, 10] and distributed stream processing engines (DSPE) [2, 7, 11, 13, 17]. DPSS, which is mostly developed by the networking community, aims at providing scalable content-based stream delivery service to a large number of users. It can be applied in applications such as stock/sports tickers, news feed etc. Such systems adopt a loosely coupled architecture and hence are easy to deploy. In such an architecture, data sources continuously publish data to the network without specifying the destinations. Hence the destinations are independent of the data sources. Instead, the distributed brokers will cooperatively route the data to the interested users based on the data content. In summary, DPSSs are efficient in content-based data delivery to a vast number of users.

On the other hand, existing DSPEs, which are mainly built by the data management community, are targeted at supporting complex continuous queries over data streams. Example applications include network management, sensor network monitoring etc. In these applications, the number of users is relatively smaller and the queries are much more complex than those of the DPSS applications (e.g., user queries are specified in SQL-like statements). Moreover, in DSPEs, the processors are coupled more tightly to enhance the processing efficiency. Query operators are allocated to the distributed processors to optimize

the system performance. Each processor evaluates the local operators and forward the results to the downstream processors or the end users. In short, DSPEs are efficient in complex stream query evaluations.

The above two types of systems target different application scenarios and hence adopt different architectures. However, there are emerging applications (e.g., financial market monitoring) that require the strengths of both systems. <http://www.traderbot.com> illustrates an example of such applications. It provides continuous query processing services over real time data such as stock trades, financial news etc. This kind of service has a potentially large number of users and hence the DPSS architecture is desirable to scale up the data delivery service. However, the user queries require a query interface with much more features than that can be provided by a DPSS. On the other hand, existing DSPEs are not scalable to the large user number of these applications.

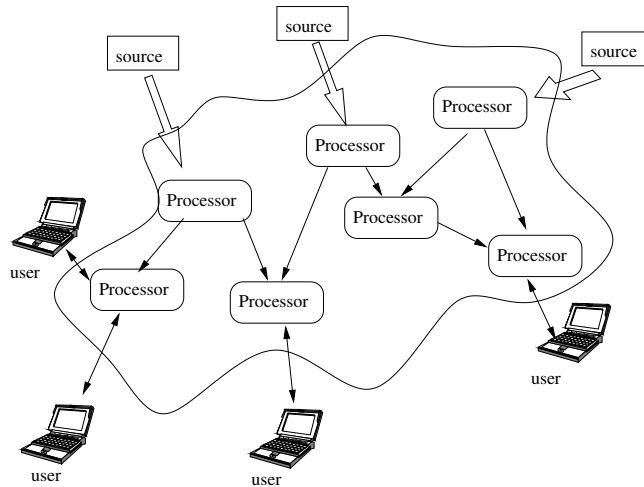
In this paper, we propose a new architectural design to bridge the gap between these two types of systems and to leverage their strengths. To handle both the streaming data and streaming queries, the system adopts a two-layered architecture. The data layer resembles the DPSS architecture and is responsible for disseminating source data streams among the processors as well as the result streams to the users. The query layer distributes the streaming queries to a number of distributed nodes for processing. The goal of query distribution is to achieve load balance and minimum communication cost. We model the problem as a graph partitioning problem and develop an efficient query distribution algorithm. Our performance study shows the efficiency of our techniques.

The rest of the paper is organized as follows. Section 2 presents the model and the challenges of the system. Sections 3 and 4 present the query distribution techniques. Representative experimental results are also presented in Section 4. Section 5 concludes the paper.

## 2 System Model and Challenges

Figure 1 shows the overview of our scalable distributed stream processing system, COSMOS (COoperative and Self-tuning Management Of Streaming data). The service is backed by a number of distributed processors interconnected with a widely distributed overlay network. These processors may be under independent administrations and may join or leave the system anytime. A number of data sources continuously publish their data to the network through the processors. User queries submitted to the system are specified in high level SQL-like language statements. For simplicity, we only consider continuous queries and assume they do not involve stored tables.

There are two major challenges in the system: data stream delivery and query processing. Two layers of services, the data layer and query layer, are provided to address these two problems respectively. Figure 2 shows the architecture of a processor in the system. The delivery of both the raw streams and the result streams, is handled by the data layer module. The raw streams received by the data layer module will be transformed and cleaned by the data cleaner and then



**Fig. 1.** Overview of COSMOS

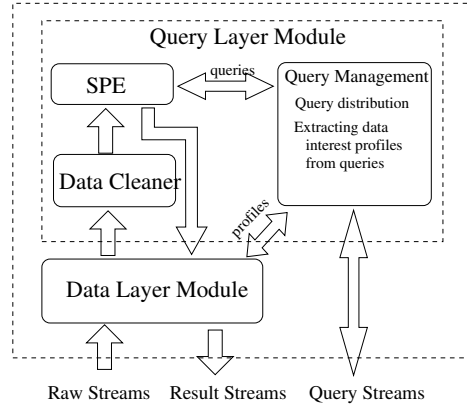
pushed to the stream processing engine (SPE) for processing. Existing single site SPE such as TelegraphCQ [6], STREAM [15] and Aurora [4], can be adopted in our system. The following subsections present more details and identify the challenges in both layers.

## 2.1 Data Layer

Given the user queries, the system should route the source data streams to the processors to feed the queries and deliver the result streams to the users. Existing DPSS architecture is employed to support this service. DPSS provides a scalable content-based stream delivery service. The service is backed by a number of brokers, which are organized into multiple dissemination trees. Data sources can just push their data into the network through their root brokers without the need to specify the destinations. Data destinations are identified by their data interest, which is specified by their profiles. A profile typically contains a set of predicates over the attributes of the data. Upon receiving a data item, a broker checks if its neighboring brokers or its local users are interested in the data item and only forwards it to those interested parties.

The data layer module of a processor in COSMOS plays the role of the brokers in the DPSS architecture. The data interest profiles used by this module is extracted from the local user queries by the query management module. More specifically, the selection predicates of each query are extracted and used to compose the profile. :w

As for the data delivery scheme, we adopt a similar scheme as SemCast [10]. In this scheme, the data space is partitioned into multiple subspaces by dividing each stream into multiple substreams. We denote the total set of substreams in



**Fig. 2.** Architecture of a processor

the system as  $SS = \{ss_1, ss_2, \dots, ss_{|SS|}\}$ . Logically, the data interest of a node can be represented as a bit vector  $q \in \{0, 1\}^{|SS|}$ , where  $|SS|$  is the total number of substreams.

$$q[i] = \begin{cases} 1 & \text{if substream } ss_i \text{ overlaps with the data interest of } q, \\ 0 & \text{otherwise.} \end{cases}$$

When a tuple arrives, it is matched to a substream and then sent to those destinations that are interested in the substream. Matching a tuple to a substream is actually searching the subspace (the substream) that covers a specific point (the tuple) in the data space. This can be solved by using existing techniques, such as R\*-tree [3].

To maximize the filtering efficiency, the data space partitioning strategy is critical. In SemCast [10], the authors proposed a semantic approach to partition the stream spaces based on the user profiles. As a first cut, we simply adopt this approach. As the focus of this paper is on the design of the query layer, we shall not discuss the data layer any further.

## 2.2 Query Layer

COSMOS is designed to support a large number of clients. Therefore, we have to distribute a large number of queries to the processors for processing. To enhance the processing efficiency, two objectives have to be considered. (1) To achieve maximum system utilization and minimum processing latencies, load balancing among the processors is desirable. (2) Due to the large volumes and continuity of streaming data, minimizing the communication cost in the system is also very important.

Unfortunately, typical DPSS does not consider load balancing and simply allocate user queries to the closest brokers. The DSPEs proposed in [13, 17] employed load balancing techniques for a cluster of locally distributed processors,

but they did not consider the communication cost. Thus, they are not suitable for a widely distributed network. On the other hand, in [2, 11], optimization algorithms were proposed to distribute the query operators across a set of widely distributed processors to minimize the communication cost. The processors are assumed to employ the same processing model and data model. However, these techniques failed to address the load balancing problem. Hence, they are suited for applications where the system is under a central administration and there are only a relatively smaller number of complex queries to process. [14] studied the static operator placement problem in a hierarchical stream acquisition architecture, which cannot be applied in our architecture. Load balancing is also ignored in this piece of work.

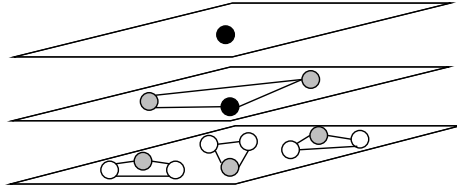
In this paper, we opt to distribute queries to processors instead of operators. In other words, we adopt a query level load distribution scheme (instead of an operator level scheme). Several reasons prompted this design decision. First, it is simpler (than an operator-based scheme) and practical. As nodes in a DPSS architecture may be under independent administrations, they may install different stream processing engines or different versions of the same engine. Hence distributing query load at the operator level may be infeasible. For instance, moving a window join operator from the STREAM system to a TelegrafCQ system is hard to implement, because it relies on a special data structure “synopsis” implemented in STREAM which is not only manipulated by the join operator itself but also other operators before or after the join operator. Furthermore, even if the processors use the same engine, one may upgrade its engine without informing the others. This might also give rise to problems unless forward and backward compatibility is implemented. Second, operator level load distribution may tighten the coupling of the processors. Besides adopting the same processing model and data model, the processors may also have to synchronize with each other during the processing of a query. Third, distributing at the operator level would be too complex to be scalable to a large number of query streams.

In addition, our query-level distribution scheme is essentially non-intrusive - existing single site processing engines can fit into our system without much extra software (re)development.

There are two problems to be addressed: (1) How to scale up the query distribution algorithm to a large number of user queries. (2) How to distribute queries to achieve the objectives mentioned above. These two problems are addressed in the following sections respectively.

### 3 Coordinator Network Construction

To enhance the scalability of the query distribution algorithm, we deploy a number of coordinators to perform this task. Among the processors in our system, we select a few of them as coordinators. Each such processor performs two separate logical roles: the stream processor and the coordinator, while others perform only the stream processor role. We assume that separate resources of these pro-



**Fig. 3.** Hierarchical Coordinator Structure

processors are reserved for these two roles. Hereafter, the words “processor” and “coordinator” refer to the logical roles.

To scale to fast streaming rate of queries, we organize the coordinators into a hierarchical structure. An example of this structure is illustrated in Figure 3. All the processors are clustered into multiple close-by (in terms of communication latency) clusters. Within each cluster, a processor, say  $x$ , is elected as the parent of the cluster, which is responsible for distributing queries to processors within this cluster. In this way,  $x$  only has to maintain the information of the processors in its cluster (e.g., statistics of the queries running in the processors etc.). For example, in the bottom plane of Figure 3, the processors are organized into three clusters, and one coordinator (drawn in gray) is selected within each cluster. The coordinators are also clustered level by level in a similar way. An interior coordinator manages a set of close-by coordinators (its children) and is responsible for distributing the queries to them for further distribution. It has to maintain a larger scope of information which is the total scope of its children. To enhance the scalability, the information in the parent is much coarser than those in the children (Section 4). Queries are first submitted to the root coordinator and then are distributed down one level at a time until a processor is reached.

## 4 Query Distribution

In this section, we present how queries are distributed to the processors. We try to achieve two goals:

- Balance the load among the processors. In this paper, we only focus on the CPU load. We assume the relative computational capability (the CPU speed) of each processor is known. For example, we can set the capability of one processor as the basic capability and associate it with a value 1. If a processor is  $l$  times more powerful than this basic processor, its capability is valued as  $l$ . Furthermore, the load of a query is estimated as the CPU time that the query will consume per unit time in the basic processor. Hence if the total query load is  $L$  and the total capability of the processors is  $C$ , the desirable load that should be allocated to a processor with capability value  $l$  is  $l \cdot \frac{L}{C}$ .
- 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. Following existing work [2, 11], we use the weighted unit-time communication cost, i.e. the per unit time message transfer rate of each link times the transfer latency of the link, to measure the communication efficiency. Here, we use the transfer latency to estimate the distance between two processors.

To minimize this cost, there are two issues to be addressed. First, the total message rate in the system should be minimized. Hence, for each tuple that has to be disseminated, it is desirable to disseminate it to as few processors as possible. That means we should minimize the overlap of the data interest of the processors. Second, we should avoid transferring data through links with long distances as far as possible. This suggests we should maintain data flow locality. For example, if a few queries have very large overlap in their data interest, distributing them to a few nearby processors can achieve better data flow locality than distributing them to a few faraway nodes as the nearby processors can cooperatively disseminate the data that are of interests to them.

To achieve the above two goals, we dynamically partition the queries into  $N$  partitions, where  $N$  is the total number of processors, and allocate them to the processors. The scheme balances the load among the processors while minimizes the overlap of the data interest between the partitions and maintains the data flow locality.

## 4.1 Problem Modeling

**4.1.1. Simple Approach.** To solve the problem, we model it as a graph partitioning problem. More specifically, we construct a *query graph* as follows. Each vertex in the graph represents a query and there is one edge between every two vertices that have overlap in their data interest. Each edge is weighted with the estimated arrival rate (bytes/second) of the data of interest to both end vertices (queries). The weight of a vertex is equal to the estimated load that would be incurred by the query at the processor with the basic capacity. These weights can be estimated based on previously collected statistics and may be re-estimated at runtime based on the new statistics. By doing so, we can model the query distribution problem as a graph partitioning problem:

*Given a graph  $G = (V, E)$  and the weights on the vertices and edges, partition  $V$  into  $N$  disjoint partitions such that each partition has a specified amount of vertex weights and the weighted edge cut, i.e. the total weight of the edges connecting vertices in different partitions, is minimized.*

Figure 4.1 illustrates an example query graph that comprises 5 queries. The weights of the vertices and edges are drawn around them. If, for example, we have to allocate the queries to two processors, we can consider two plans: (a) allocate  $Q_3$  and  $Q_4$  to one processor and the rest to another; (b) allocate  $Q_3$  and  $Q_5$  to one processor and the others to another. Both the two plans can achieve load balance. However, plan (b) has a better communication efficiency, where only 3 (bytes/second) of data are transferred twice, while in plan (a) 8 (bytes/second)

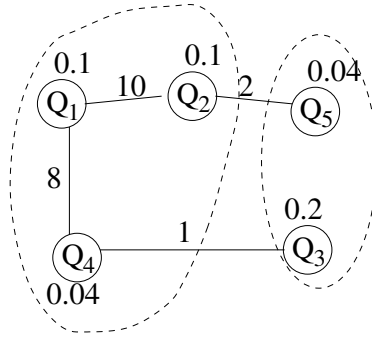


Fig. 4. A simple query graph model

of duplicate data are transferred. Note that only considering allocating similar queries together may not result in good performance. As can be seen from this example,  $Q_3$  and  $Q_5$  are not similar in their data interest but allocating them together results in a better scheme.

**4.1.2. Extended Approach.** The above simple model, unfortunately, does not capture the cost of transmitting the result streams to the users. To solve this problem, we extend the above model as follows.

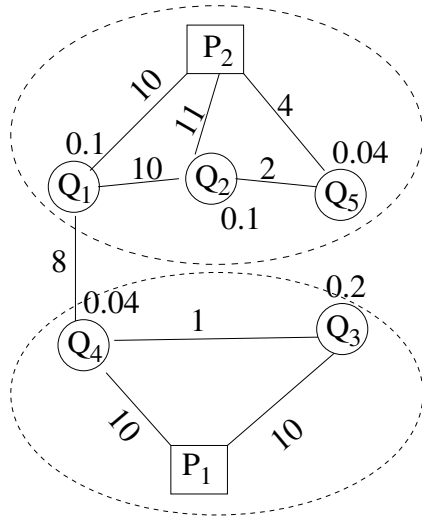
First, we adopt the same assumption as a DPSS that a user is allocated to his closest processor when he joins the system in the first place. The user and the processor are said to be local to each other. Since the result stream of a query is routed to the user by the DPSS architecture of the data layer, it will be first routed to the user’s local processors and then to the user. Therefore, the cost of transferring the query result from the processor to its local users are unavoidable.

Second, for each processor, we add one additional vertex into the query graph. We refer to such additional vertices as processor vertices. The weights of these vertices are set to zero. One edge is also added between each processor and each of its local queries (i.e., queries initiated from the processor). Each such edge is weighted by the estimated result stream rate (bytes/second) of the corresponding query.

Third, the graph partitioning problem statement is revised as follows: *Given a graph  $G = (V, E)$  and the weights on the vertices and edges, partition  $V$  into  $N$  disjoint partitions such that each partition has exactly one processor vertex and a specified amount of vertex weights and the weighted edge cut is minimized.* All queries within a partition are allocated to the processor in that partition.

Figure 4.1 illustrates an extended query graph. In comparison to Figure 4.1, there are two additional processor vertices, which are drawn in rectangles, and five additional edges between them and their local queries. Here, contrary to the prior conclusion, allocating  $Q_3$  and  $Q_4$  to  $P_1$  and the rest to  $P_2$  is a better plan.

**4.1.3. Challenges.** The graph partitioning problem is an NP-hard problem but has been extensively studied in a lot of application areas, such as parallel scien-



**Fig. 5.** An extended query graph model

tific computing, VLSI design etc. [12] provides a survey of the graph partitioning algorithms in the application of scientific computing. To enhance the scalability, parallel algorithms have also been proposed [8, 16]. Unfortunately, the problem in our context bears a few important differences from these previous studies:

(a) The semantics of the graph is different. In our problem, the edges between the query vertices represent the overlap relationship of the data interest among different queries, while, in prior work, they model the amount of communication between the vertices. Furthermore, there is no notion of processor vertices in prior work.

(b) Traditional graph partitioning algorithms do not consider how to assign the resulting partitions to the processors. That is because processors are all connected by fast local network and identical in terms of network locations. However, in a widely distributed network, maintaining data flow locality is critical to minimizing communication cost.

(c) To enhance the scalability of the partitioning algorithm, a distributed or parallel algorithm is needed. In prior work, processors are assumed to be connected by a fast local network [12]. Hence they employ parallel algorithms that require frequent communication between the processors. However, our system is a widely distributed large scale system and the communication cost among even the coordinators could be very high. Hence the existing parallel algorithm is unsuitable for our problem.

The above differences render the existing solutions inadequate for our problem.

---

**Algorithm 1:** Graph Coarsening

---

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

---

## 4.2 Proposed Approach

In this section, we present the proposed solution. Unlike the previously proposed parallel algorithms, which allow frequent communication between any pair of coordinators, we employ a *hierarchical graph partitioning* algorithm that is run on our hierarchical coordinator tree. Each leaf coordinator first collects the query specifications from its child processors. These queries are tagged with their original locations. A query graph is then generated over the queries in each leaf coordinator. The edge weight between a pair of vertices can be efficiently estimated by summing up the rates of the substreams of interest to both end vertices. The common interest of two queries can be determined easily based on their data interest vectors (Section 2.1).

Each coordinator (except the root) will perform Algorithm 1 to coarsen its local query graph before submitting to the parent. This algorithm repeatedly collapses two selected vertices until the number of vertices are smaller than or equal to  $v_{max}$ . The weights of the corresponding edges and the merged vertex are adjusted accordingly. We set  $v_{max}$  as  $|V|/f$ , where  $|V|$  is the total number of vertices and  $f$  is the fanout of the parent coordinator. For ease of presentation, we define the following functions:

- (1)  $adj(u)$  returns the set of adjacent nodes of  $u$ ;
- (2)  $pro(u)$  returns true if  $u$  is a processor vertex;
- (3)  $matched(A)$  returns all the matched vertices from a set of vertices  $A$ .

In the algorithm, a vertex  $u$  tends to collapse with a neighbor  $v$  which has an edge  $e(u, v)$  with a larger weight. Two processor vertices with different tags will not be merged together. That is because they belong to different child clusters and hence they should be put into different partitions and allocated to different clusters of processors.

The vertices in the coarsened graph are tagged with the current coordinator’s name and then the graph is submitted to the parent, who will perform the same procedure after receiving all the coarsened graphs from its children. Note that the procedure can run in parallel in different subtrees, which can accelerate the whole procedure.

Finally, when the root coordinator receives all the graphs from its children, it will generate a global query graph and then partition it into  $f$  partitions, one for each of its children. The partitioning is done based on the total computational capabilities of the processors within the scope of each child. Here we can apply any of the traditional graph partitioning algorithms [12] while ensuring the constraint that two processor vertices with different tags cannot be put into the same partition. In our experiments, we use the algorithm in [9]. Each child coordinator will then uncoarsen the subgraph assigned to it one level back. Based on the tags of the vertices, the information of the finer-grained vertices can be retrieved from the corresponding coordinator. Finally, the uncoarsened graph is partitioned as in the root coordinator. This procedure repeats at each level until all the queries are assigned to the processors.

We can see that, at each level of the above procedure, the queries are distributed to minimize the overlap of data interest between different regions of the network. Furthermore, the higher the result stream rate of a query, the more likely that the query is put close to its local processor. These help maintain data flow locality.

### 4.3 Experiments

In this section, we present a performance study of the proposed techniques. A network topology with 4096 nodes is generated using the GT-ITM topology generator. The Transit-Stub model, which resembles the internet structure, is used. 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. Our algorithms are implemented in C and the communication between the processors is simulated. The experiments are run on a Linux Server with an Intel 2.8GHz CPU.

The default cluster size parameter  $k$  used in the coordinator tree construction is set to 4. 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). As it is hard to collect large number of real query workload, we use synthetic query workload in our experiments. To simulate clustering effect of user behaviors,  $g = 20$  groups of user queries are generated and each group has different data hot spots. 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.

We compare our approach with three other approaches: (a) Naive: allocate the queries to the processors to balance the load without considering their data interest. (b) Greedy: an expected load limit is computed for each processor. If

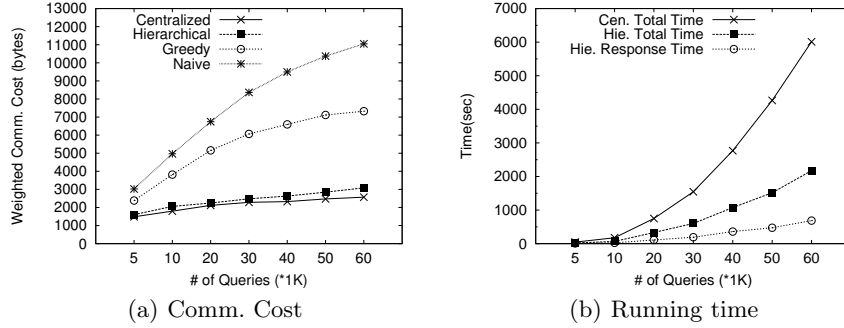


Fig. 6. Initial Distribution

the capability value of a processor is  $l$ , then the expected load limit to it is  $(1+a) \cdot l \cdot \frac{L}{C}$  (Recall Section 4), where  $a$  is a tunable parameter to ensure all the queries can be accommodated. Then queries are distributed one by one to the processors. Each query is distributed such that the current communication cost is minimized and the load limit of each processor is not violated. (c) Centralized: all queries are collected to a centralized node and distributed using a centralized graph partitioning algorithm. We use unit-time communication cost - the per unit time message transfer rate of each link times the transfer latency of this link, as the metric to evaluate the query distribution schemes. Figure 6(a) presents the unit-time communication cost of all the four approaches. It can be seen that Naive performs the worst because it cannot identify the data interest of the queries and optimize their locations. Greedy works a lot better by taking the data interest of queries into account. The two graph partitioning algorithms perform the best and their performances are similar. This also verifies that the graph coarsening procedure in our hierarchical partitioning algorithm does not incur much errors. We also report the response time and total time of the centralized and hierarchical graph partitioning algorithms in Figure 6(b). It is shown that both the response time and total time of the hierarchical approach are much less than the centralized one. The hierarchical approach has shorter response time because the coordinators in different branches can work in parallel. To understand why the hierarchical approach also performs better in total time, let us look at an example query graph with  $n$  nodes. The complexity of graph partitioning is  $k \cdot n^2$ , where  $k$  is a constant. Hence if, for example, we partition the graph into 2 sub-graphs with sizes of  $n_1$  and  $n_2$  and process them separately, then the complexity is  $k \cdot (n_1^2 + n_2^2) < k \cdot n^2$ . Since in the hierarchical algorithm, an interior coordinator partitions the query graph into subgraphs and passes them to the child nodes for further partitioning, the total time is also reduced in comparison to the centralized approach.

## 5 Conclusion and Future Work

In this paper, we propose a new architectural design to leverage the strength of distributed publish/subscribe systems to support scalable continuous query processing over data streams. This architecture retains the loose coupling and easy to deploy merits of a DPSS, while obtaining the processing capabilities of a DSPE. To handle both the query stream and data stream, two layers of services, query layer and data layer, are provided, respectively, by two functional modules. Solutions are proposed to solve the load distribution in the new architecture and performance study are performed to show their effectiveness. Based on the current results, there are a few directions that we will explore.

### **Adaptive Query Distribution.**

The proposed approach in this paper only considers static query distribution. However, in a continuous query context, the system parameters such as data rates, communication bandwidth, workload of the processors, the running user queries etc., are subject to changes in the midst of the query execution. Hence the initial distribution of queries may become suboptimal and a re-optimization is required. The simplest way to perform re-optimization is to re-run our query distribution algorithm from scratch. However, this may incur large overheads in both running the optimization algorithm and migrating the states of the queries. An adaptive algorithm that can minimize the number of query migration as well as optimizing the query distribution quality.

### **Heterogeneity of Query Engines.**

Since COSMOS allows different processors employ different query engines, it is possible that a processor can only process certain types of queries and different processors may have different query interfaces and different query semantics. Hence a query wrapper is required to be plugged into each processor to translate the query between the universal query language and the local query language. Furthermore, a user query may not be able to be executed at any processor. Therefore, query distribution algorithm that is aware of this kind of constraints is required.

### **Computation Sharing.**

By adopting the DPSS architecture and optimizing the query distribution, COSMOS minimizes the communication cost by exploiting the sharing of the communication among different queries. We can achieve better system performance by exploiting the opportunities to share the computations among the queries. A processor can publish the intermediate result streams to the system and other processors can subscribe to it if their running queries have similar operations.

## Fault Tolerance.

Fault tolerance is an important feature for such a service oriented system. It should be supported by both layers in the system with a different target. The fault tolerance module at the data layer is targeted at providing highly available data delivery service while the one at the query layer is aimed to provide highly available query processing service. Different techniques should be developed for these two layers respectively.

To date, we have implemented the proposed algorithms in this paper. We are now trying to incorporate various stream processing engines and then plan to deploy it onto real network environment.

## References

1. M. K. Aguilera, R. E. Strom, D. C. Sturman, M. Astley, and T. D. Chandra. Matching events in a content-based subscription system. In *PODC*, 1999.
2. Y. Ahmad and U. Çetintemel. Networked query processing for distributed stream-based applications. In *VLDB*, pages 456–467, 2004.
3. N. Beckmann, H.-P. Kriegel, R. Schneider, and B. Seeger. The R\*-tree: An efficient and robust access method for points and rectangles. In *SIGMOD Conference*, 1990.
4. D. Carney et al. Monitoring streams: A new class of data management applications. In *VLDB*, 2002.
5. A. Carzaniga, D. S. Rosenblum, and A. L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Transactions on Computer Systems*, 19(3):332–383, Aug. 2001.
6. S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.
7. M. Cherniack et al. Scalable distributed stream processing. In *CIDR*, 2003.
8. G. Karypis and V. Kumar. A coarse-grain parallel formulation of multilevel k-way graph partitioning algorithm. In *PPSC*, 1997.
9. G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.
10. O. Papaemmanouil and U. Çetintemel. Semcast: Semantic multicast for content-based data dissemination. In *ICDE*, 2005.
11. P. Pietzuch et al. Network-aware operator placement for stream-processing systems. In *ICDE*, 2006.
12. K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high-performance scientific simulations. pages 491–541, 2003.
13. M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, 2003.
14. U. Srivastava, et al. Operator Placement for In-Network Stream Query Processing In *PODS*, 2005.
15. The STREAM Group. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin*. 2003.
16. C. Walshaw, M. Cross, and M. G. Everett. Parallel dynamic graph partitioning for adaptive unstructured meshes. *J. Parallel Distrib. Comput.*, 47(2):102–108, 1997.
17. Y. Xing, S. B. Zdonik, and J.-H. Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.