# TOWARDS AN INTERNET-SCALE STREAM PROCESSING SERVICE WITH LOOSELY-COUPLED ENTITIES

**YU FENG**
*(B.Sci.(Hons.), NUS)*

**A THESIS SUBMITTED**
**FOR THE DEGREE OF MASTER OF SCIENCE**
**DEPARTMENT OF COMPUTER SCIENCE**
**SCHOOL OF COMPUTING**
**NATIONAL UNIVERSITY OF SINGAPORE**
**2006**

# Acknowledgements

I am truly fortunate to have had the opportunity to work with my supervisor, Tan Kian-Lee, who has been my supervisor since my honor year project and I am overwhelmed by gratitude for his patience and continuous guidance in research throughout all these years. He is the one who brought me to the world of research and taught me many things necessary for being a successful researcher. In particular, he showed me how to do critical thinking and writing and how to conduct convincing experimental studies. The meetings he had with me to discuss my research problems and to guide me through them with his profound knowledge are unforgettable experience to me. The research assistantship under him enabled me to pursue my research without financial concerns.

I am also indebted to Zhou Yongluan for many research interactions, and for significantly helping to improve the work in this thesis. It is with him that I have the first rank one publication in my life. Another critical support group during my master life is the database group of SOC, which I can always find the feeling of belonging.

I would particularly like to thank: Zhang Zhenjie, Yang Xiaoyan and Sheng Chang, whom I have worked with in several projects for the graduate courses; Lu Hua and Xu Linghao, who taught me many lessons on research, as well as on life.

Finally, I am eternally thankful to my family. Mom, Dad: you always believe in me, and it is largely because of your constant support and encouragement that I am completing my Master Degree. To my dear, Wenyue: you have provided me with valuable relief and encouragement when I needed. It is you that gives me the strength to overcome the innumerable obstacles that I encountered during my master study.

# Contents

## Summary

Stream processing engines are application-independent, specially designed query engines to process high-volume, real-time data streams. In recent years, many stream processing engines have been developed and employed by business entities to provide stream processing service over the Internet. However, due to the inherent limitations of those stream processing engines, these entities suffer from scalability, over-investment and availability problems. A system incorporating those entities, promoting joint cooperation can achieve better system resource utilization, economical efficiency and scalability. In this thesis, we present the architecture of a scalable distributed stream processing system made up of loosely coupled entities. It provides two layers of services: the query layer service and the data layer service.

The query layer service is to dynamically distribute queries to the most appropriate entity for processing to achieve load balance and minimize communication cost. This service is backed by a number of coordinators, which are special entities organized into a hierarchical structure. The query distribution problem is modelled as a graph partitioning problem and we leverage existing graph partitioning algorithms and derive a hierarchical graph partitioning algorithm to achieve load balance among the entities as well as minimum communication cost in transferring the data streams. We address the problem of fast incoming new queries (streaming queries) by employing an effective query routing scheme to route new coming queries to a suitable entity.

A runtime adaptive query redistribution mechanism is devised to adapt to the change of the environment like stream rates, user surging requests, etc. to enhance system performance during runtime.

Data dissemination is often neglected by existing stream processing systems. In many situations, especially in the wide-area, the network is the stream bottleneck. In our system, we identify this problem and address the problem of how to efficiently transfer data streams to various geographically dispersed stream processing entities. This is one aim of the data layer service besides providing query evaluation services to clients. In our system, stream processing entities are urged to collaborate in data dissemination besides evaluating assigned queries, rather than relying on the source nodes sorely in data dissemination. Cooperation trees for data dissemination are built and specially designed routing queries are employed to represent data interest of entities, which facilitate data dissemination from one entity to another selectively.

We design experiments to test the effectiveness of our proposed techniques and our simulation results show the efficiency and superiority of our proposed techniques with respect to those traditional ones.

# List of Tables

# List of Figures

# Chapter 1

# Introduction

## 1.1 Background and Motivation

In recent years, a new class of applications which operates on high-volume, real-time continuous streaming data has emerged. These applications have presences in various domains, including financial monitoring, large-scaled environment monitoring, network management, sensor networks, traffic control, etc. Different with the traditional database management systems(DBMSs) which are based on the "store-then-process" model, stream processing systems require results to be computed continuously over a long period of time and new results to be returned incrementally once they are available. That is why the queries in these systems are usually referred to as "Continuous Queries". It has been widely accepted that traditional DBMSs are inadequate for stream processing [Aba03, BBDW02, Cha03]. Thus a lot of work has been devoted to

developing stream processing engines [Car02, MSHR02, Cha03, The03, Che03], which are application-independent, specially-designed query engines to evaluate continuous queries. All these stream processing engines support complex continuous queries over push-based data streams, though they may adopt different data model and processing model during evaluation.

Due to the popularity of these stream-oriented applications, we foresee that there will be a lot of business entities that provide stream processing service to clients over the Internet. They charge clients for the service provided based on the computational power consumed. Since each entity is under a single adminstration, it installs and runs its own stream processing engine based on its business choice. The engine can be a centralized one like TelegraphCQ [Cha03], which is easy to deploy and maintain, or a distributed one like Aurora* [Che03], which offers significant computational power compared to the centralized one. Note an entity employs a cluster of processors to deploy a distributed stream processing engine. Regardless of what stream processing engine an entity installs, the system appears to be a black box to clients. Clients submit their queries to those entities through some graphic user interface and each entity processes client queries independently using its own stream processing engine.

There are several problems faced by each individual entity under this model:

- The scalability problem. For each entity, the capacity for the stream processing engine is limited, which is supposed to be sufficient to handle the expected

number of queries submitted by clients to the system. However, accidental request surges from clients may cause the system overloaded, which results in long user perceived delays and leads to client dissatisfaction. Moreover, even if the number of queries remains constant, data streams can be bursty, with unpredictable peaks during which the load may exceed available system resources.

- The over-investment problem. To cope with accidental surging client requests and bursty data streams, one entity may opt to invest a lot in hardware to upgrade the system. However, for most of the time the system resource is excessive and is not economical from business perspective.

- The availability problem. Though many stream processing engines are fault tolerant, special mechanisms to deal with system failure are needed, which is complex and costy. Moreover, some events like system reboots, software upgrade etc. are inevitable. But for a service provider, this kind of service interruptions may lead to loss of clients. The over-investment problem may easily arise when an entity tries to cope with this availability problem.

Moreover, to those authorities concerning system resource utilization like network bandwidth consumption, the system utilization factor for this model is low. A lot of data streams are flooding in the network, many of which are redundant. To enhance overall system resource utilization, achieve better economic efficiency as well

as to provide better service to end users, a more ambitious service is to incorporate numerous heterogeneous entities to form a *federation*. The federation exploits the processing power and capabilities of each individual entity as they cooperate in query evaluation as well as stream data dissemination and can achieve optimal performance under various situations. For example, for those with not enough capacity to handle surging client numbers themselves, they can be assured that their service will not be compromised as they have a strong backup - the whole federation. Also there is no service interruption once they are in the federation; they can rely on other entities to continue providing stream processing service when they temporarily go offline. Business entities need not worry about over-investment as their investments get remedied even if they do not have enough clients themselves. Furthermore, as the distribution of queries is optimized from the system level, bandwidth consumption will be low and the overall system performance improves. Therefore, either from the perspective of each individual service provider or the perspective of system resource utilization, this model provides a win-win solution.

However, this problem is challenging and poses several issues to be solved:

- How heterogeneous entities are organized in the federation. Currently distributed stream processing technologies are not applicable to this problem. In a distributed stream processing engine, processors are interconnected by a fast local network and are highly coupled, i.e. they use the same data model and

processing model in order to cooperate in evaluating client queries. For our context, entities are heterogenous and it is impractical for each of them to surrender its administration and install a uniform stream processing engine. Furthermore, extensive communication among processors may occur during the query evaluation process for a distributed stream processing system. Notably this is not feasible in a WAN context due to long latencies.

- How client queries are distributed among the entities. The distribution of queries should achieve maximum system utilization and minimum processing latencies. Load balancing among the entities is another important consideration. To the best of our knowledge, there is no prior work done that addresses both of these two aspects.

- How data streams are transferred to various entities. Much work on stream processing has focused on how to efficiently process the continuous queries but very little has been done with how to efficiently transfer newly generated data to the entities for processing.

## 1.2 Contributions

In this thesis, we present a new architectural design of a distributed stream processing system. Our contributions are:

- Our system leverages the power of each individual stream processing entity and incorporates them into an Internet-scale distributed stream processing system. The entities are loosely coupled, choosing stream processing engines according to their business choice independently. The system is easy to deploy as no modification on single site stream processing engines is needed.

- We identify two important issues to be addressed for the design of such a system and model them as two layers of services, namely, the query layer service and the data layer service. These two layers are orthogonal in terms of target(query v.s. data) and member(coordinator v.s. entities) and we employ a modular approach to design the services on these two layers.

- The service on the query layer is to dynamically distribute queries to the most appropriate entity for processing, with the aim to optimize the system performance. This service is carried out by a number of coordinators, which are organized into a hierarchical structure. The query distribution problem is modelled as a hierarchical graph partitioning problem and we leverage the existing algorithms to achieve load balance among the entities as well as minimize communication cost in transferring the data streams.

- We address the problem of fast incoming new queries (streaming queries) by employing an effective query routing scheme to route the new coming queries to a suitable entity. A runtime adaptive query redistribution mechanism is devised

to adapt to the change of the environmental conditions like stream rates, user surging requests, etc. to enhance system performance during runtime.

- In our system, we identify and address the problem of how to efficiently transfer data streams to various stream processing entities which are geographically dispersed. In our system, stream processing entities are urged to collaborate in data dissemination, rather than relying on the source nodes sorely. Data dissemination trees are constructed and specially designed routing queries are employed to selectively disseminate data from one node to another.

- We design experiments to test the effectiveness of our proposed techniques. A simulation system is implemented and our simulation results demonstrate significant performance gains with respect to traditional techniques.

## 1.3 Thesis Roadmap

The rest of the thesis is organized as follows. Chapter 2 reviews some related work and provides more background on this problem. Chapter 3 presents the detailed problem analysis and an overview of the system architecture. Core techniques addressing the various challenges at the query layer are presented in Chapter 4 , followed by the detailed design of the data layer in Chapter 5. Chapter 6 gives an extensive performance study of the various techniques proposed in this thesis. Chapter 7 concludes

the thesis and discusses the future work.

# Chapter 2

# Related Work

Our work is related to several research areas, namely, publish/subscibe system, stream processing systems, data dissemination and graph partitioning. In this chapter, let us review some of the related work to have a better understanding of our problem.

## 2.1 Publish/Subscribe System

In a publish-subscribe system, senders label each message with the name of a topic ("publish"), rather than addressing it to specific recipients. The messaging system then sends the message to all eligible systems that have asked to receive messages on that topic ("subscribe"). To some extent, the publisher resembles the source nodes and the subscriber resembles the entities in our context. However, in a publish/subscribe system, a node can be both a publisher and a subscriber, which is not

possible for our case. SIFT [YGM99] is a selective document dissemination system which allows users to subscribe to text documents by specifying a set of weighted keywords. It was one of the earliest projects to suggest the reversal of roles of queries and data in filtering systems through the use of an inverted index on the queries. Some other systems like [FJL+01], model messages as attribute-value pairs, and allow user profiles to contain a set of predicates over the values of those attributes. In [OJW03, SDR03, ARS04], clients subscribe to some data with precision requirements and the system exploits the precision requirements to do filtering on data streams thus reduce bandwidth consumption. Recently there is an increasing interest in XML filtering for publish/subscribe systems as XML provides more expressiveness in specifying data interests, resulting in more accurate filtering of messages. XFilter [AF00] and YFilter [Dia03] are two XML-document filtering engines that efficiently group and apply XPath queries over incoming documents. In [DRF04], the authors presented a distributed system providing large-scale XML dissemination service leveraging YFilter. This system is composed of many brokers which accept client queries specified in XPath statements and route new messages to clients when they match the XPath statements using YFilter. However, this problem has several differences with ours. Firstly, the focuses of these two systems are totally different. The prior one is to provide data dissemination service to clients, thus it focuses on how to efficiently distribute data to distributed brokers and then to a large number of clients.

On the contrary, our system aims to provide stream processing services with a number of single site stream processing entities over the internet. How to distribute client queries to those entities for processing is of paramount importance. Data dissemination from source nodes to various entities is a pre-requisite for stream processing. Secondly, the data dissemination system does not support complex queries, therefore the problem of load balance among the brokers is overlooked. However, for our system, to achieve optimal system resource utilization, load balance is a critical factor to take into account. Last but not least, many techniques proposed in [DRF04] are only applicable in the XML context like routing query construction etc., we need more generic algorithms in our problem.

## 2.2 Stream Processing System

Stream processing systems aim to process client queries on stream data. Different with ubiquitous query processing systems, stream processing systems allow client to submit queries which are executed for a potentially long period and notify clients when new results from the incoming streams are available. To distinguish those queries in stream processing systems from their counterparts in ubiquitous query processing systems, they are usually referred to as *Continuous Queries*(or *CQs*) [LPT99, BW01]. The core issue of such systems is how to devise novel algorithms to efficiently and effectively evaluate the continuous queries submitted by clients. Earlier work on

stream processing system focuses on centralized ones while later many distributed systems are proposed to address the scalability problem.

### 2.2.1 Centralized Stream Processing

The earliest work is Tapestry [TGNO92], which supports continuous queries on append-only relational databases. OpenCQ [LPT99] is a system integrating distributed heterogeneous information sources and supports continuous queries. It uses a processing algorithm based on incremental view maintenance. NiagaraCQ [CDTW00] is another system supporting continuous queries for monitoring persistent data sets spread over a wide-area network, e.g. web sites over the internet. It addresses scalability issue of the system in terms of number of queries that can be supported by the system by proposing techniques to group similar queries together for evaluation. CACQ [MSHR02] is a system designed to process a large number of continuous queries. Based on Eddy [AH00], it realized adaptive processing, dynamically reordering operators to cope with changes of arriving data properties and selectivity. This approach is followed by many distributed stream processing systems. TelegraphCQ [Cha03] is a new implementation based on the prototype of CACQ, with the focus on support for shared, continuous query processing over query and data stream. Other centralized stream processing systems include STREAM [The03], Aurora [Aba03], etc. All these stream processing engines can be installed inside an entity to provide stream

processing services to clients.

### 2.2.2 Distributed Stream Processing

Clearly there is a limit to the number of queries that can be handled by a single entity, no matter how efficient the algorithm the system utilized is. Recently, there have been several attempts to extend the single-site model to multi-set, distributed models and environments. PeerCQ [GL03] and CQ-Buddy [NST03] are two decentralized continuous query processing systems in peer-to-peer network. PeerCQ is an information monitoring system which uses continuous queries to express monitoring requests. The system performs service partitioning and load balancing using P2P framework. In CQ-Buddy, there is a simple model to measure the similarity between queries and similar queries are executed together within one server. Also it takes the differences in capabilities of peers into account and balances their loads. Both of these two systems are in the peer-to-peer network context and utilize the standard P2P framework for search, communication etc. For example, queries are routed in the network to find a suitable node for processing. This is not viable in our context for query distribution. The reasons are threefolds. Firstly, query routing in the WAN context is too costy. The scale of our system is so large that millions of queries are running in it at any moment. Network conditions deteriorate dramatically if such a number of queries are routing inside. Secondly, queries are joining/leaving the system

frequently due to the large base number and query routing tends to take a while to finish. It is not scalable to the fast changing query stream. Last but not least, query routing is prone to achieve local optimal rather than global optimal, while overall system efficiency is one of our design goals.

In [Che03], the authors introduced *Aurora*, a centralized stream processor. A large-scaled distributed stream processing system is then built using *Aurora**, which is a distributed version of *Aurora*, and *Medusa*, which is an infrastructure supporting federated operations across *Aurora**s. Client queries are decomposed into operators, which are dynamically distributed among nodes of *Aurora** or even among *Aurora** for processing. This system provides three key features, namely "a scalable communication infrastructure, adaptive load management and high availability". Our system has similar goals as *Medusa* and is motivated by it. However, there are several different challenges we face in our system. For example, besides considering load management among entities, we should also consider how to distribute queries to achieve system efficiency like minimal bandwidth consumption, etc. Moreover, queries are distributed in the granularity of operators among multiple processors inside an *Aurora** and between *Aurora**s(as federated operations) for processing. The advantage of this approach is it exploits the commonality among queries and common operators can be processed in a shared manner thus the performance is improved. Nevertheless, this approach requires all the processors in the system adopt the same stream processing model and synchronize between each other during the processing of a query. In other

words, the processors are tightly coupled. Though it is appropriate to deploy such a stream processing system inside an entity, it is not applicable to the WAN context. The reason is threefolds. Firstly, entities are heterogenous in terms of stream processing engines. Different engines use different data model and processing model and are highly incompatible. Secondly, even for entities adopting the same stream processing engine, each entity is an administrative autonomy. Though they are willing to join the system to achieve better performance and economical efficiency, it is unlikely they will surrender their autonomy and allow their processors to be tightly coupled. This also brings extra effort to re-engineer current single site stream processing engines to solve the synchronization in a WAN context. Last but not least, due to the large scale of our system, queries are submitted to/withdrawn from the system in a high rate, i.e. queries are streaming. Distributing queries in the granularity of operators at the system level may incur too much overhead and is too complex to scale well.

Another recent paper [AC04] describes a distributed query processing system on stream data. The system exploits the knowledge of network characteristics (e.g., topology, bandwidth etc.) when deciding on the network locations where the query operators are executed. This network-aware operator placement, however, is done separately for each query and does not make use of the relationships among queries to share their computation. Neither the load of each processor is considered. Recently researchers working on sensor network tossed a term called *in-network query processing*

which denotes data processing that takes place inside the sensor network. It is essentially a kind of distributed stream processing system. Madden et al. [MSHR02] were the first to study in-network process. They focused on simple aggregation queries, whose execution can be distributed over an arbitrarily large set of sensor nodes. They introduced a routing strategy that imposes a spanning tree onto the network: data is aggregated at every internal node in the routing tree. The work [BB03] introduced an adaptive and decentralized algorithm that progressively refines the placement of operators of a single query by walking through neighbor nodes thus reduces data traffic. In [SMW05] the authors address in-network query processing for queries involving possibly expensive conjunctive filters and joins and consider the problem of placing operators along the nodes of a sensor network hierarchy so as to minimize the computation and data transmission. The techniques proposed in these distributed systems are not applicable to our system due to different context. Nevertheless, they provide some insights on factors to take into accounts when we design the query distribution scheme for our system, like network location of the entity, query characteristics, etc.

To conclude, stream processing receives a lot of attention from the database community and many algorithms to address the efficient processing and the scalability problem have been proposed. They provide some guidelines and insights when we are designing algorithms for our system. For example, these existing systems usually subsume a new incoming query into one existing query group to exploit the similarity

among the queries. This motivates us to use a graph partition algorithm to cluster similar queries to one entity for processing. However, all the studies on stream processing have overlooked two important issues: how to efficiently route fast data streams to the wide distributed stream processing entities and how to route incoming queries to the most suitable entity at runtime.

## 2.3 Data Dissemination

The topic of data dissemination was first introduced by the network community. The earliest approach is to use multicast [AD93] to disseminate data. Multicast allows data from one source to be sent to multiple receivers and is bandwidth-efficient. However, due to the fact that it relies on the network layer paradigm, it is not flexible. This has led to a lot of work on application-level multicast(or content-based multicast) e.g. [CRZ00, ZZJ+01, CDKR02, Ban03, BS04]. In application-level multicast, members of a multicast group typically self-organized into an overlay topology, over which dissemination trees are created. The earlier work typically assumes a traditional multicast model, where all members of one multicast group have exactly the same interest and there is no filtering of data during the dissemination to reduce bandwidth consumption. Some following work focuses on how to reduce bandwidth consumption during dissemination by exploring the overlay structure [Cas03, KRAV03]. Most recently, researcher becomes aware of the relationship among queries. Semcast [PC05]

utilizes the relationship between clients' profiles like containment and partial overlap, and places similar ones in the same channel, for which a dissemination tree is built. In this way a number of dissemination trees are avoided. There is some work concentrating on how the dissemination tree is constructed also [SDR03, SRS02]. In these two papers, the authors introduced the notion *fidelity*, which is defined as the percentage of time that the data value at one node conforms to its coherency requirement. The nodes with more stringent coherency requirements are placed at the top of the dissemination tree while the less stringent ones are placed at the bottom. Note that filtering is done during data dissemination at the intermediate servers and these two approaches are more scalable and efficient. Nevertheless, they deal with a subset of queries with less expressiveness and generally are not applicable to systems with more expressive queries like continuous queries.

To conclude, work related to multicast dissemination systems focuses on how data should be disseminated, with great interests in constructing the overlay structure which facilitates the dissemination. In our problem, we incorporate the power of data dissemination into a distributed stream processing system to improve system utilization.

## 2.4 Graph Partitioning

Graph partitioning is a fundamental problem and has been studied extensively in a lot of literatures. The aim of graph partition is to partition the vertices in a graph into several subsets such that some objectives are met. The objectives can be:

- The load of each partition is equal or conforms to some pre-defined ratios.
- The weight sum of edges spanning subsets is minimum.
- Or both.

We can see there are some similarities between the graph partitioning problem and our query distribution problem, which motivates us to model the latter as a graph partitioning problem and use existing well-developed algorithms to solve it.

It is known that the graph partitioning problem with both load balance and minimal edge cut requirements is NP-Complete. Nevertheless a lot of heuristics are proposed which work well in practice. Earlier work focuses on static graph partition and dynamic graph repartitioning [KL70, KK98a, SKK97]. To scale to larger graphs, parallel graph partition algorithms have been proposed [KK98b, WCE97, HL95]. K. Schloegel et al. surveyed various graph partitioning algorithms in the application of scientific computing in [SKK03]. To the best of our knowledge, no prior work has used graph partitioning algorithms to solve the query distribution problem. We leverage these well-studied algorithms and adapt them to fit in our problem context. A

hierarchical query partitioning algorithm is proposed in this thesis, which is derived from previous work. It utilizes the hierarchical structure of the coordinator tree(see chapter 4.1 for more details) and can cope with the large scale of the system, in terms of query number as well as number of entities(number of partitions).

# Chapter 3

# Problem Formulation and System Overview

In this chapter, we present a detailed analysis of our problem and an overview of our system to solve the problem.

## 3.1 System Model

The objective of our system is to provide a service to evaluate complex continuous queries for a large number of clients over the Internet. This service relies on the numerous stream processing entities which are geographically dispersed. Each entity is a single administration domain and it independently makes the choice to install any sort of stream processing engine. Different stream processing engines may use different

data model and query evaluation model and are not compatible. Some entities may employ a cluster of processors to deploy a distributed stream processing engine. These processors are highly coupled and from our system's perspective they are indivisible. To ease the presentation, the terms "node" and "entity" will be used interchangeable hereafter, referring to one single administrative site using its own stream processing engine. As we can see, entities are heterogeneous in terms of both processing model and processing power. Different with the processors in one entity which are connected by LAN, entities are widely distributed and interconnected by the Internet to form an overlay network. We assume this overlay network resembles the structure of the Internet, i.e., In the overlay network, besides the stream processing entities, there are a number of stream sources that continuously generate data streams. Stream sources are widely distributed and may reside at any location over the Internet.

Client requests on the stream data are specified in a high level SQL-like queries like the ones in [The03]. Clients dynamically submit/withdraw their queries to/from the system through a graphic user interface and the system appears like a black box to them. Our system ensures client queries will be processed at an appropriate node that results in best system performance and the desired data will be delivered to the respective client in time. Clients are notified by some means like popup windows or emails for new query results. Figure 3.1 provides a overview of such a system.

Different with the nodes in traditional Peer-to-Peer network, entities in our system are expected to be much more robust and stable. This assumption is based on the

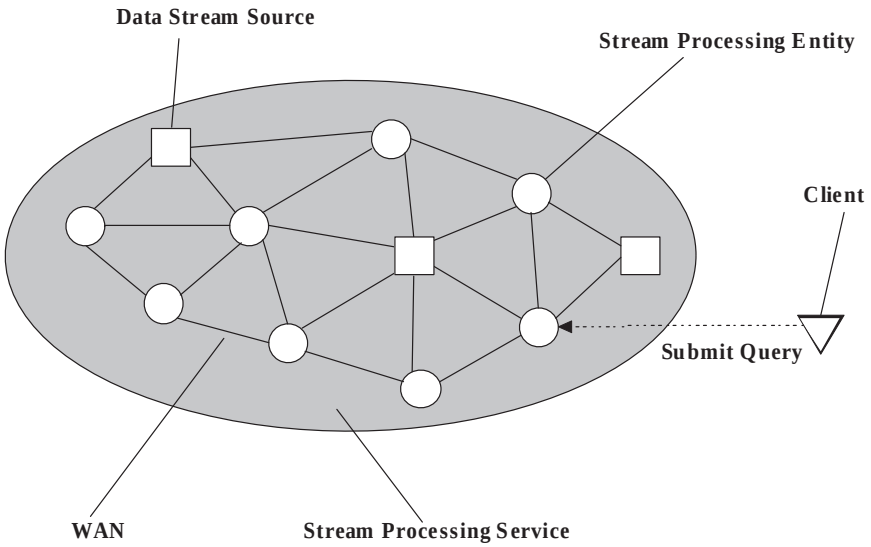



Figure 3.1: System model

fact that each entity itself is a stream processing service provider thus is robust to failures as a basic business requirement. They can utilize various mechanisms like backup servers, fault tolerant stream processing engines [SHB04], etc. to achieve system robustness, which are out of the scope of this thesis. The direct consequence of robust entities is that entities are unlikely to leave the system due to failure; they actively request departure before they leave. Moreover, entities join our system to form a federation to achieve better economical efficiency and provide better service to attract more clients. They are paid for their contribution to the system using their otherwise excessive computational power and are charged for consuming system resources which themselves lack to cope with surging client requests. It is not likely that some entities will purposely leave the system except some unusual situations like

server restart, system upgrading, etc. Also for every entity in the federation, they are prohibited from frequent join/leave activities, which is an obligation they agreed on before they can join the federation. Therefore, we assume the frequency of active entity departures is low.

To realize such a system, there are mainly two issues that arise to be addressed: How client queries are handled by the system and how stream data is disseminated from sources to various entities.

### 3.1.1 Query Distribution

When new queries arrive, our system has to assign them to an appropriate node for processing. One goal of our system is to achieve better system resource utilization, therefore load balance among the nodes are important. Either overloading or underloading will result in sub-optimal system performance, which leads to long processing latencies to clients. This kind of processing latencies should be avoided especially for mission critical, real-time monitoring applications. Moreover, query distribution can affect the communication cost incurred for transferring data streams from source nodes to entities. For example, assigning queries requesting one specific data stream to one entity avoids transferring that stream to many entities, which is the case if these queries are distributed randomly among entities. Figure 3.2 and Figure 3.3 depict two scenarios to illustrate how query distribution affects communication cost.

The source node $s$ are generating two types of data streams, namely stream $S1$ and $S2$. $Q1$ and $Q3$ have request data stream $S1$ while $Q2$ and $Q4$ have interest in stream $S2$. The first distribution strategy assigns $Q1$ and $Q2$ to node $P1$ and $Q3$ and $Q4$ to $P2$, which results in transmission of $S1$, $S2$ to both $P1$ and $P2$. On the contrary, the second distribution strategy assigns $Q1$ and $Q3$ to entity $P1$ and $Q2$ and $Q4$ to $P2$. In this case only $S1(S2)$ is transferred to $P1(P2)$, respectively. Due to the huge volume and continuous nature of data streams, how to minimize the communication cost of the system by optimizing query distribution is of paramount importance. Reduction in the communication cost not only improves the quality of the network, but also reduces the burden of each entity to receive useless information. Interestingly prior work either focus on ensuring load balancing [Bab04, XZH05] or minimizing communication cost [AC04] but not both.

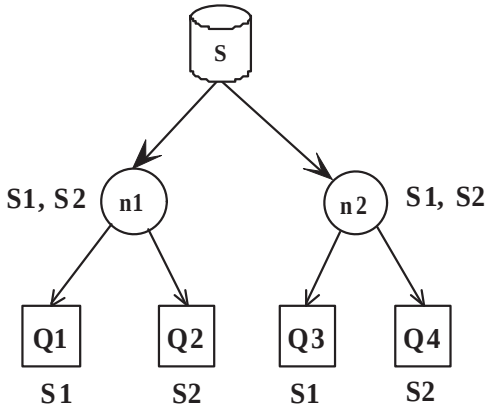



Figure 3.2: Query distribution with overlap of data interest

As discussed in Section 2.2.2, assigning queries in terms of operators among entities

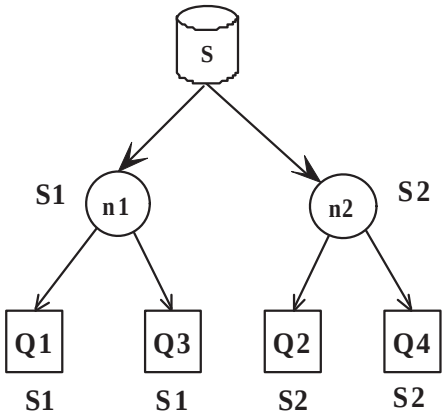



Figure 3.3: Query distribution without overlap of data interest

is not appropriate for our context. Therefore, our system opts to assign queries as a whole to entities and relies on the robustness and strength of each single site stream processing engine inside each entity to further optimize performance. Later we will see that our query assignment algorithm tends to assign similar queries to one entity, which implies high potential for further optimization inside each entity.

This problem we are facing is different from the "query routing" in peer-to-peer or distributed database systems. In these systems, query routing refers to routing queries to a location where the required data of the queries resides. However, in our context the relationship between data and queries is the opposite. We fix the location to run a query, which then in turn determines whether newly generated data should be routed to that location to avoid flooding the network. Therefore, the desired location to process one query from the system's perspective is a place resulting minimal additional communication cost and better system resource utilization. To

achieve system level/global optimality, we adopt a coordinator-based strategy, which exploits network locality and is easier (than say DHT) to support load balancing.

### 3.1.2 Data Stream Transport

As data sources and entities are inherently dispersed, data streams need to be transferred from data sources to entities for processing. Data stream transport refers to this process. However, this aspect of stream processing has been overlooked. Most of existing distributed stream processing systems [AC04, XZH05] adopted a flat system structure as illustrated in Figure 3.4. Functionally each entity plays a uniform role in the system thus data streams are pushed directly from various sources to each of them indifferently. As a result, each data source has to transmit every update of a single stream to all entities concurrently. In order to scale up to a large number of clients like millions of clients like our system does, the number of entities employed by the system must be large, say thousands of them. If such a flat structure is adopted, for each update of a single data stream, the source has to send a copy to thousands of entities. For a source node generating thousands of data streams, which is not uncommon like in [BW01, AC04, SDR03], it becomes the bottleneck of the whole system.

As we have seen, previous work has neglected this problem and relied solely on the sources to transfer the streaming data to all the entities. This approach not

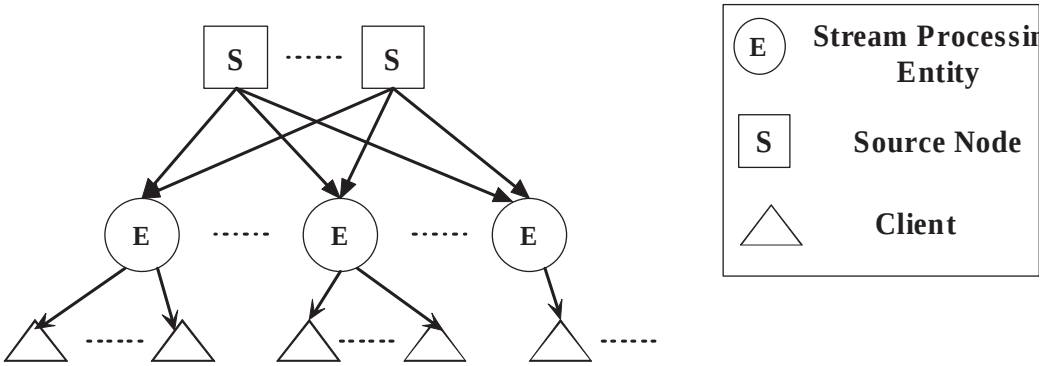



Figure 3.4: Flat Structure

only increases the workload of the sources which consequently makes them the potential bottleneck of the whole system, but also stresses the whole network by sending messages redundantly. To solve this problem, we propose a cooperation approach which employs the stream processing entities in data dissemination rather than relying solely on the source nodes themselves. In this scheme, the entities are organized in a hierarchical structure, which is widely adopted in large scale data dissemination system. Figure 3.5 illustrates the hierarchical structure. Essential it is a composition of trees, whose number depends on the number of stream sources. In each tree, a stream data source node is at the top of the tree, which keeps generating new data to be disseminated. Below the source node are the various stream processing entities which cooperates to disseminate data besides evaluating client queries. Each parent entity is responsible to transfer the upstream data to its child entities. In this way, the number of nodes the source nodes need to transfer the data is limited thus the problem of overloaded source nodes is resolved. Moreover, stream data is transferred

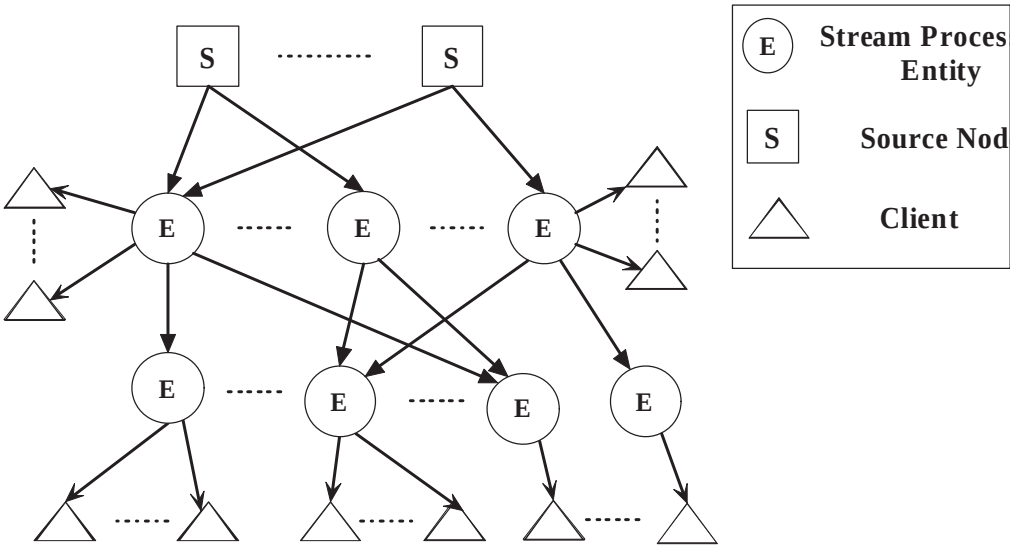



Figure 3.5: Hierarchical Structure

according to the data interest of the child nodes: only those data streams matching the data interest of the child nodes will be transferred. Redundant transmissions are eliminated to reduce the communication cost.

## 3.2 System Overview

From above discussion, we can see that our system needs to two services: the query management service and the data management service:

- The query management service deals with queries submitted to the system by clients, including distributing queries to entities for processing, adaptively re-distribute queries during runtime for better performance, managing new joining

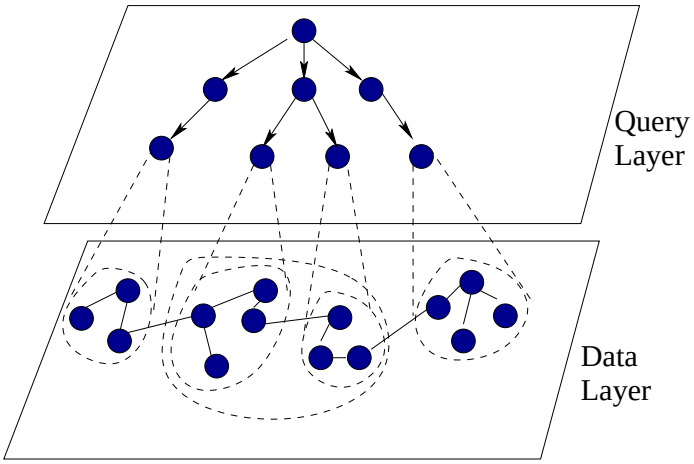



Figure 3.6: The two-layer services

queries as well as query updates. This service is provided by various coordinators in the system.

- The data management service focuses on data, including client query processing and data stream dissemination. This service is backed by those entities in our system.

Note that these two services are orthogonal in terms of target(query v.s. data) and member(coordinator v.s. entities). Thus we refer to them as "the two-layer services", which are illustrated in Figure 3.6. The techniques used in these two layers are presented in detail in the following chapters.

Query distribution is done by a number of coordinators, which are special stream processing entities in our system. Note the terms "Entity" and "Coordinator" are used to differentiate the logical roles of an entity in the system in this thesis. Nodes

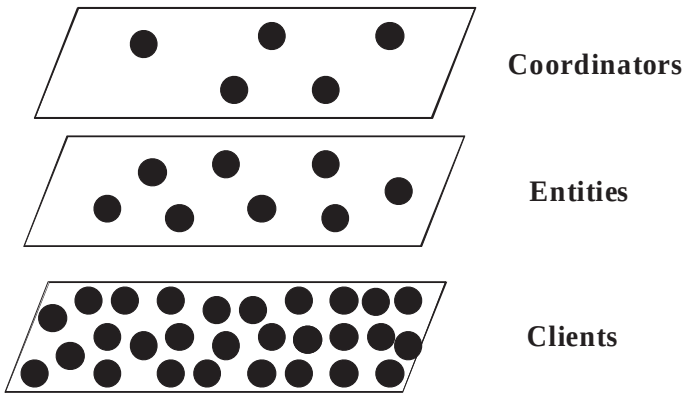



Figure 3.7: The Three-layer Network

are clustered into several groups based on their network localities and one node in each group is selected as the coordinator for this group. Each coordinator is in charge of its group, responsible for query management in this group. From this perspective, our system is composed of three layers of network: the coordinator layer, the entity layer and the client layer. Figure 3.7 demonstrates the three-layer network architecture of the system. The number of nodes in each layer is increasing downwards.

Besides evaluating queries, entities in the system cooperates on stream data dissemination. Cooperation trees are built to facilitate data dissemination and entities participates in these trees based on their data interest. which is represented by routing queries. To avoid flooding the network, data interest of an entity is represented by a local routing query. The data interest of a subtree of the cooperation tree is represented by a tree routing query, which is the aggregation of the local routing queries of entities in the subtree. Only data that passes those routing queries are

disseminated.

Figure 3.8 presents the architectural design of each entity in the our system. It contains the following modules.

**Stream Processing Engine:** This module is constructed by using any single site stream processing engine that has been developed, such as TelegraphCQ [Cha03], STREAM [The03], Aurora [Aba03], etc. It facilitates continuous query evaluation in an entity.

**Data Manager:** This module is responsible to selectively route data to the descendants or the local stream processing engine. The routing is based on the data interest of the destinations. It is guaranteed that all the data of interest to the destination would be transferred.

**Query Manager:** All the controls of the queries is handled by this module. The coordinators in the query layer communicates with entities/coordinators via this module.

**Cooperation Manager:** This module is responsible to assist the construction of cooperation trees. The exact operations depend on the cooperation tree construction strategy.

**Catalog Manager:** This module maintains the catalog.

**Communication Module:** This module provides an interface for this entity to communicate with other entities.

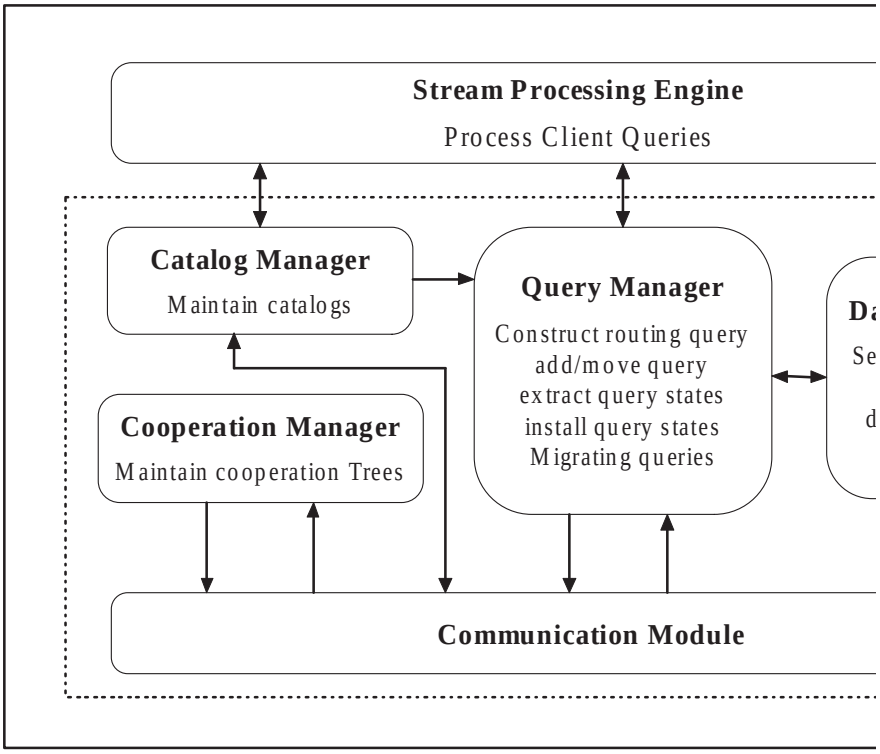



Figure 3.8: Architecture of an entity

The modules in the dotted box are the additional components needed for entities joining our system. They provide a kind of wrapper over the stream processing engine to avail an entity to be part of the "Federation" and enjoy the benefits of it.

## 3.3 Chapter Summary

In this chapter, we discussed the system model of our problem first. Two issues that arise in order to realize such a system are presented: the query distribution and the data stream transport. Query distribution should achieve load balance as

well as minimize communication cost of the system. Data stream transport is an issue that has been overlooked in related work, relying on the source nodes sorely for streaming data transfer. We identify and address this issue by introducing a hierarchical structure of entities, promoting collaboration among them for stream data transport. The overview of the system is then presented. It has two layers of services(query layer and data layer) and three lays of networks(coordinator, entity and client). The architecture of the entity in our system is illustrated. The following chapter will elaborate more on the two layers of services of our system.

# Chapter 4

# Query Layer Design

In this chapter, we present the design of the query layer. The query layer provides query management service in our system. It deals with the queries submitted to the system by clients, including distributing queries to entities for processing, adaptively redistribute queries during runtime for better performance, managing new joining queries as well as query updates. The challenge of this layer lies on the following issues.

- The load of the entities should be balanced during the initial query distribution as well as during runtime. Load balancing ensures the good system utilization and small processing delay of client queries.

- Queries distribution should be done in a manner that the commonality of queries running at different entities should be minimized, i.e. query distribution should

result in a clustering effect. Queries at one node should have great commonality while at different node should have as little commonality as possible. This clustering effect not only increases the opportunity query optimization can exploit during evaluation at each single site stream processing engine, but also reduces communication cost as the transfer of data required to evaluate similar queries at one node can be shared.

- The system has to handle the arrival of new queries and the removal of old queries. Continuous queries are potentially long running queries; that does not mean the update frequency of these queries is low. On the contrary, due to the large scale of the system, we expect the frequency of query joins and drop offs is high. The continuous update of queries is referred to as *query streaming*. Query distribution and information update should be done efficiently with minimal overhead to accommodate the fast changing nature of queries.

- One characteristic of the stream data is that the volume of a stream is highly fluctuating. Such fluctuations not only change the network condition but also deteriorate previous optimal stream processing performance. Thus a mechanism to adapt to network changes and data stream characteristics is vital to achieve runtime optimum performance. Thus queries are redistributed based on some guidelines to maintain load balancing and minimal communication cost.

To address the issues mentioned above, we decompose the query management service into a number of subtasks like initial query distribution, runtime adaptive query re-distribution, query streaming (query join/drop off) handling, etc. Essentially all these tasks are to assign the streaming queries to a desirable entity for processing. All these subtasks are carried out by coordinators of the system, which forms a network themselves. In the rest of this chapter, we will look at the design of the coordinator network in section 4.1 and then various techniques devised to cope with each subtask of this layer.

## 4.1 Coordinator Network

### 4.1.1 Coordinator Modeling

In our system, we adopt a coordinator-based approach for the query distribution. As mentioned above, queries are streaming. To accommodate the fast arriving rate of queries, multiple coordinators are required. These coordinators are special nodes selected from stream processing entities in the system, which play the coordinator role as well as the stream processor role. We assume separate resources of these entities are reserved for these two roles for simplicity and they are compensated for their additional contribution as Coordinators to the system. There are several alternatives to model those coordinators and their relationship.

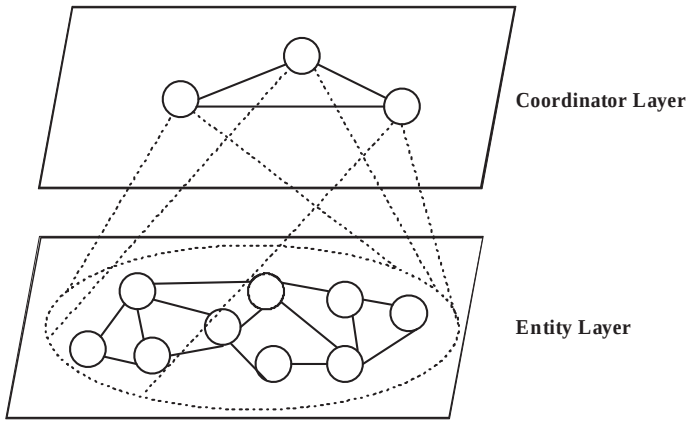



Figure 4.1: The naive replication approach

### Coordinators as Replicas

A naive approach is to model these coordinators as replicas as illustrated in Figure 4.1. In this model, every coordinator maintains information about every entity in the system, which is needed for query distribution. The information at each coordinator is identical so they are replicas of each other. In Figure 4.1, there are three replicated coordinators, which forms another layer named "Coordinator Layer" above the "Entity Layer". Utilizing the global information/statistics it keeps track of, each coordinator can independently decide the most suitable entity for any specific query. Also due to the uniformity of coordinators, incoming queries can be submitted through any of them for distribution. Load balancing among the entities is easy to achieve since each coordinator has complete information about the load of every entity in the system. The main drawback of this approach is the storage space required to keep track of the information of the entire system at each replicas. Moreover, the highly volatile nature

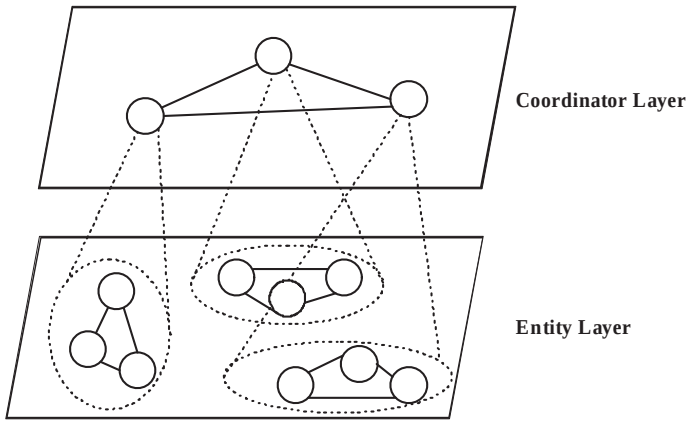



Figure 4.2: Flat structure of coordinators

of such information worsens the situation by introducing large amount of communication and maintenance overhead. When queries stream in and out of the system, every replicated coordinator needs to synchronize its information with other replicates to keep its information accurate and updated. Such synchronization is resource consumptive and prohibitive for a large number of coordinators.

### Coordinators as Peers

Another approach is to divide the system into several regions and each coordinator is in charge of one region. The division can take the geographical location of each entity into account and nearby entities are clustered to form one group. One entity in each group is selected as the coordinator. This approach is illustrated in Figure 4.2. Each coordinator has complete knowledge of the statistics/information of his own domain to make query distribution decisions inside its domain. When a new query is

submitted to an coordinator, the coordinator first checks if it can accommodate this query in its domain without incurring much additional cost. If not, it will ask for help from other coordinators. In this approach, the amount of information maintained by each coordinator is reduced dramatically and the intensive synchronization among coordinates is avoided. Note the extreme case for this modelling is each group contains only one node, i.e. the coordinator itself. This modelling resembles the super-peer model in Peer-to-Peer architecture and various techniques like random walk [GMS04] can be applied. Nevertheless, this approach has the same inherent disadvantage as the query routing in P2P network. Since each coordinator has no knowledge about other domains, the query distribution process may take a long time to settle the final location of a query. Also the distribution tends to achieve local rather than global optimization due to lack of global knowledge.

**Our Hierarchical Modeling**

We propose a hierarchical structure to model the coordinators. It is derived from the second alternative. Nodes are clustered into domains based on their network localities and within each domain one node is elected to act as the coordinator. These coordinators form another layer above the entity layer as in alternative 2. However, nodes on that layer are also clustered into groups and one node is selected as the coordinator for each group. This process continues level by level recursively until one single node remains. This node is referred to as the root node. Figure 4.3 is a simple

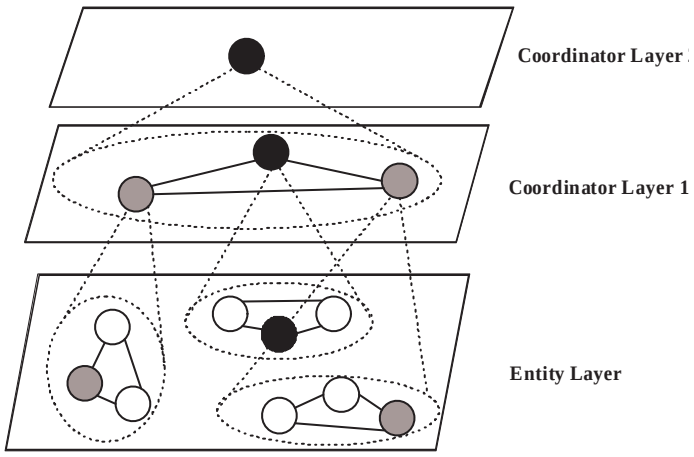



Figure 4.3: Hierarchical structure of coordinators

example to illustrate this process. On the bottom entity layer, there are nine entities which are clustered into three domains. One coordinator is selected for each cluster (denoted using dark circles) and these three nodes constitute the first coordinator layer. Within this coordinator layer, these nodes are clustered into one group and one node (denoted using a dark circle) is named as the coordinator for this group, which forms one higher coordinator layer. Since it is the only member of the top coordinator layer, this process ceases.

For those nodes on the coordinator layers, all of them are coordinators actually. We use the term "super coordinator" to denote the node in charge of each domain on these layers and "ordinary coordinators" to denote the rest. A super coordinator is responsible to distribute queries among the nodes in its domain for further distribution. For the coordinators on the bottom coordinator layer("coordinator layer 1" in Figure 4.3), they are responsible to distribute queries to the entities in their own

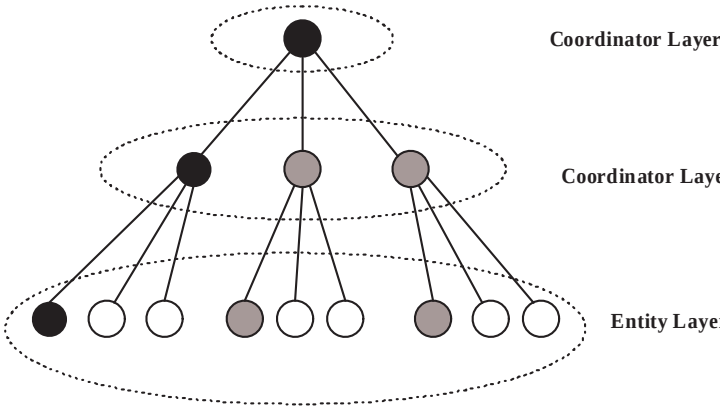



Figure 4.4: The tree representation

domain and it indicates the end of the query distribution process. As we can see, the scope of information needed in order to make query distribution decisions increases from bottom to top. For the coordinators on the coordinator layer 1, they only need to maintain information about the entities in its domain. For each super coordinator, the information maintained is the aggregation of that maintained by every node in its domain. Therefore, for the top level coordinator, it has to possess an overview of the whole system. This is necessary in order to achieve optimal at system level.

Figure 4.4 is another representation of this modelling using a tree structure. The leaf nodes of the tree are the stream processing entities and the rest are coordinators. We refer to those coordinator nodes whose child nodes are entities in the tree as *leaf coordinator*s. A parent node in the tree is the (super) coordinator of all of its children nodes, i.e. all the sibling nodes constitute a cluster and the parent node of them is the coordinator (super coordinator for non-leaf sibling nodes).

The root coordinator has complete knowledge of the queries running in the system. Clearly it may becomes the potential bottleneck of the system. To alleviate this scalability problem, we devise a novel technique which coarsens the information maintained by the super coordinators at each level. This technique is derived from approximation techniques [BS04] and tries to omit those less important details while capturing the main characteristic of the original information. The coarsened information provides a kind of summary of the information for each node in one domain, which is sufficient for the super coordinator to make an optimal query distribution decision. New queries are submitted to the root coordinator and then is routed down level by level until it finally reaches an entity, which will be responsible for its evaluation as well as result notification. The hierarchical tree structure determines the complexity of information updated by the arrival or removal of a query is $O(\log N)$, as only those coordinators on the path from the affected leaf coordinator to the root need to update their information.

Although in our design, all queries have to be routed through the root coordinator, it is still scalable to the fast query stream. There are two factors which determines the processing delay of the root coordinator for distributing a single query: the number of nodes the root coordinator can pass the query (potential candidate for data forwarding) and the time to check the suitability of each node. The number of nodes it can pass the query is a tunable parameter of our system as it is determined by the number of nodes in each cluster. The root can adapt to sudden surging of the query

stream at runtime by reducing the number of nodes in its cluster thus alleviate the burden of excessive checking. Moreover, checking is done based on the information which is specially coarsened to speed up the checking process. From both of these two aspects the root coordinator is unlikely to become the bottleneck of the system. Judging from the computation power of prevailing servers in reality, one root coordinator is sufficient to handle up to 10,000 new arriving queries per second(please refer to section 6.3 for more detail), which is an appropriate estimated figure for our system. Even if the rate of the query stream exceeds this estimate and is too fast to be handled by the root coordinator alone, a replicated root coordinator could be deployed. Note that the number of replicas is dependant on the rate of the queries stream, rather than the number of entities in the system as the naive replication scheme is. Without loss of generality, we restrict our discussion to only one root coordinator for simplicity in this thesis.

### 4.1.2 Coordinator Tree Construction

The coordinators are organized into a hierarchical tree structure as illustrated in Figure 4.4 and we refer to this structure as the *coordinator tree*. In this section, we will present the coordinator tree construction algorithm.

The coordinator tree construction algorithm should exploit the locality of the entities, keep the tree balance as well as accommodate the join/leave of the entities.

Note that the departure rate of entities is low and departures are explicitly requested by leaving entities as discussed in section 3.1. We adapted a distributed mechanism proposed in [SBK02] which is able to construct tree incrementally and dynamically. Moreover, this mechanism is capable of maintaining a tree with following properties, which exactly fits our requirements:

1. The number of child nodes for a coordinator at each level is between $k$ and $ck - 1$, with the only exception of the root node which can have less than $k$ children.

2. Parent is the center of its cluster, i.e. with the minimum average delay to all the other nodes in its cluster. Note this property ensures the locality of nodes are utilized.

The procedure of tree construction is as follows:

- When a new entity requests to join the system, its request will first be directed to the root coordinator. For each node in the coordinator tree when it receives the joining request, if it is a leaf coordinator, it will add the joining node to its cluster. Otherwise, it identifies one ordinary coordinator in his cluster i.e. a child node in the coordinator tree, that is closest to the joining node and forwards the joining request to that coordinator.

- When a node intends to leave the system, it will send a message to its parent node which is meant for job handover. The parent will decide how to redistribute

its jobs to the remaining child nodes. If this node is also a coordinator, a new coordinator which resides at the center of the remaining nodes in the cluster is selected to replace it.

- If a coordinator realizes the number of its children exceeds $ck-1$, it will partition its domain into two with equal sizes such that the radii of the two domains are minimized. The center of the two domains are selected as two new coordinators, replacing the original super coordinator in the upper coordinator layer. Note this process may propagate to the the root coordinator.

- If the number of children of a coordinator $x$ falls below $k$, $x$ will send a merge request to the closest sibling say $y$. The two domains are merged and $y$ continues to be the coordinator of the new domain while $x$ is downgraded from coordinator to a ordinary node. However, the scale of our system tends to expand therefore cluster mergers are delayed and are not propagated to upper coordinator layer to avoid redundant work.

- Periodically a new parent will be selected if the current coordinator is no longer the center among its domain.

## 4.2 Query Distribution

In this section, we present how queries are distributed among the entities using the coordinator network. First we discuss the goals for the query distribution and then present our model to solve the problem. Thereafter initial query distribution, online query routing and adaptive query redistribution schemes will be elaborated respectively.

### 4.2.1 Goals to Achieve

There are essentially two goals to achieve for query distribution: balance the load among the entities and minimize the communication cost.

- Balance the load among the entities. Load balancing ensures the good system utilization and results in small average processing delays of client queries. In this thesis, we focus on the CPU load and will study multi-object load balancing in future work. We assume the relative computational capability of each entity joined our system is known. This can be done by choosing one entity as the standard, assigning a value 1 to represent its capability and acting as a basis to measure the rest. If another entity is $k$ times more powerful than this basic entity, i.e. the evaluation time for one query at this entity is $1/k$ of that at the standard entity, the computation capability of this entity is represented as $k$. In this way, the total computational power of the system can be estimated as the

sum of all these capability values. Similarly, the load of a query is estimated as the CPU time that will be consumed to evaluate this query at the standard entity. Hence if the total query load is $L$ and the total computational power of the system is $C$, the expected load that should be allocated to a entity with capability value $k$ is $k \cdot \frac{L}{C}$. However, absolute load balancing is too stringent to implement in reality. Instead of achieving absolute load balancing, a certain degree of load imbalance among the entities is tolerable. In our system, the load allocated to each entity should not exceed $h\%$ of its expected value, where $h$ is a system parameter. Thus the load for each node is denoted by:

$$(1 + h\%) * k \cdot \frac{L}{C} \tag{4.1}$$

- Minimize the total communication cost. The communication cost has two components: the cost to transfer streams from the sources to every destination entity and the cost to transfer query results from each entity to clients. As claimed in [Bab04], most queries posed by clients in data stream applications contains filters and aggregations. Users are commonly interested in specific portions or some global overviews of the data, rather than consuming each update of the stream data. For example, most of the queries submitted by clients at http://www.traderbot.com return less than 10 result items, with the highest update rate of 1 update per minute. On the contrary, the data sources contain

more than 20,000 items, with the highest update rate of 1 update per second. Clearly the cost to transfer streams from the sources to entities dominates the overall cost. In our problem, we focus on minimize the first portion of the total cost. In our experiments, the ratio of output rate to input rate is chosen uniformly from 0.1% to 1%.

Unlike multicast which transfers data streams to each member in the multicast group, our system delivers the data streams selectively to avoid flooding the network with redundant or useless data. The data interest of an entity is the union of the data needed to evaluate all the queries allocated to it. Intuitively, for each tuple that is to be disseminated, it is desirable to disseminate it to as few entities as possible, for the sake of communication cost. This implies that we should minimize the commonality of the data interest among the entities to reduce communication cost, which has been illustrated by Figure 3.2 and Figure 3.3 in Section 3.1.1. Another way to explain the above phenomenon is that the query distribution strategy should result in specialization, i.e. each entity will be specialized in evaluating one sort of queries, requesting a particular portion of the data streams, and queries at different entities have little overlap of data interest. One side effect of this strategy is that the performance of the single site stream processing engines installed at each entity are improved significantly as most of them have optimization mechanisms to exploit the commonality of running queries. This is another motivation to distribute a group

of similar queries to a stream processing entity for processing. Therefore, the main saving in communication cost is gained from eliminating redundant data transmissions, which is achieved by query clustering, assigning one cluster to a single node. For example, for one specific stream, originally it is disseminated to 1000 nodes for processing. However, after query distribution, it may only be sent to 10 nodes, which is a great improvement in performance. Compared with this, the communication cost between sources to nodes are relatively less important. To simplify our problem and its modeling, we omit the source-node distance information in our model, which may be addressed in future work.

### 4.2.2 Problem Modelling

As we can see from above analysis, query distribution is essentially a partition problem: Queries are partitioned into $N$ groups with minimal data interest overlap among them, where $N$ is the total number of entities in the system. Also the load of each partition is conformed to the capability of the assigned entity. To solve this problem, we model it as a graph partitioning problem. We construct a *query graph* for the queries submitted to the system and then employ a graph partitioning algorithm to solve the query distribution problem.

In the *query graph*, each vertex represents a query and the edge between two vertices represents the overlap of data interest between these two queries. Each edge

is weighted with the estimated arrival rate (bytes/second) of the data which these two connected vertices (queries) have mutual interest. Vertices are weighted with the estimated computational load that the query would impose on the basic entity. These weights can be estimated based on previous collected statistics and may be re-estimated at runtime using new statistics. Figure 4.5 illustrates a simple query graph. This query graph comprises 5 queries and the weights of the vertices and edges are drawn around them. If, for example, we have to distribute the queries to two entities with equal processing capabilities for processing. We consider two distribution plans: (1) allocate $Q_3$ and $Q_4$ to one entity and the rest to another; (2) allocate $Q_3$ and $Q_5$ to one entity and the rest to another. Note both the two plans can achieve load balance, resulting in two partitions with load 4 each. However, plan (2) has a smaller communication cost, where only 3 (bytes/second) of data are duplicately transferred to both nodes, while 8 (bytes/second) of data are duplicately transferred in plan (1).

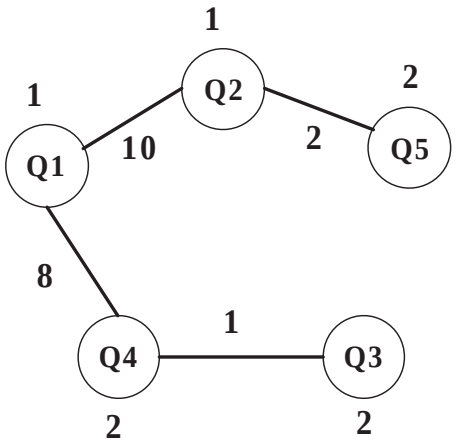



Figure 4.5: Example query graph

With the query graph, we can model the query distribution problem as a graph partitioning problem formally as follows:

> Given a graph $G = (V, E)$ and the weights on the vertices and edges, partition $V$ into $k$ 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.

Note the two goals of query distribution, namely load balance and minimizing communication cost are inherently embedded in this graph partitioning problem.

The graph partitioning problem is NP-hard but has been extensively studied in a wide context [SKK03], such as data mining, spatial databases, VLSI design etc. However, there are a number of notable differences of the problem in our context from previous studies:

- The semantic of the graph is different. In our problem, the edges of the graph represent the overlap of the data interest among different queries, while in prior work, they represent the amount of communication between the vertices.

- Traditional graph partitioning algorithms do not consider how to assign resulting partitions to processors. The reason is processors are all connected by fast local network and are uniform in terms of network locations. However, as

entities in our system would cooperate in stream dissemination and they are geographically dispersed, maintaining dataflow locality is critical to minimizing communication cost.For example, if a few partitions have very large overlap in their data interest, distributing them to a few nearby entities can achieve better dataflow locality than distributing them to a few faraway nodes.

- To enhance the scalability of the partitioning algorithm, parallel algorithms are used to do the graph partition, in the hope to reduce processing time. However these existing parallel algorithms are not applicable in our context. The reason is in previous work, processors are assumed to be connected by a fast local network [SKK03] thus the frequent communications between processors during the partitioning process are tolerable. Nevertheless, our system is built on WAN so the communication cost among the coordinators is too high to bear.

The differences listed above render the existing solutions inadequate to solve our problem. A customized graph partitioning algorithm which takes those difference into account is needed.

### 4.2.3 Initial Query Distribution

In this section, we present several initial query distribution algorithms. One naive query distribution algorithm is to distribute the query to where it is submitted to the

system until the load balance constraint is violated. If that happens, the query is re-distributed to a random node without reaching its expected workload. This algorithm is simple and fast, and can ensure no node is overloaded. The obvious problem of it is this algorithm does not utilize the data interest information of queries thus results in poor clustering quality of query distribution.

Let us consider another greedy algorithm. This algorithm assigns queries to entities one by one as they are submitted to the system and takes a greedy approach such that a new query is assigned to an entity with the least additional communication cost. Also for each node, a load limit 4.1 is imposed to ensure the load balance constraint is not compromised. The only problem with this approach is the quality of the solution given by this algorithm is not guaranteed and largely dependent on the sequence the queries are distributed.

With the query graph modeling, another initial query distribution algorithm is the centralized graph partitioning algorithm. In this algorithm, a central node collects all queries initially in the system and build a query graph for them. Then queries are distributed using a generic graph partition algorithm based on the partition it belongs. Since both load balance and data interest of queries are taken into account in the query graph modeling, this algorithm is believed to perform better than the previous two. However, the problem is efficiency: an centralized approach can hardly scale well and impose too much overhead in gathering all information to the central node.

Since both the queries and their distribution would be updated during runtime, the quality of initial query distribution becomes less critical. Therefore, we prefer a fast algorithm with reasonably good solution quality to slow algorithms even if they may lead to initial optimal solutions. Also a distributed algorithm with moderate communication among entities is more desirable than centralized one for efficiency.

With those considerations above, we devise a *hierarchical graph partitioning* algorithm that is specially designed to fit the hierarchical structure of the coordinator tree. There are two phases for this algorithm: the bottom-up phase and the top-down phase.

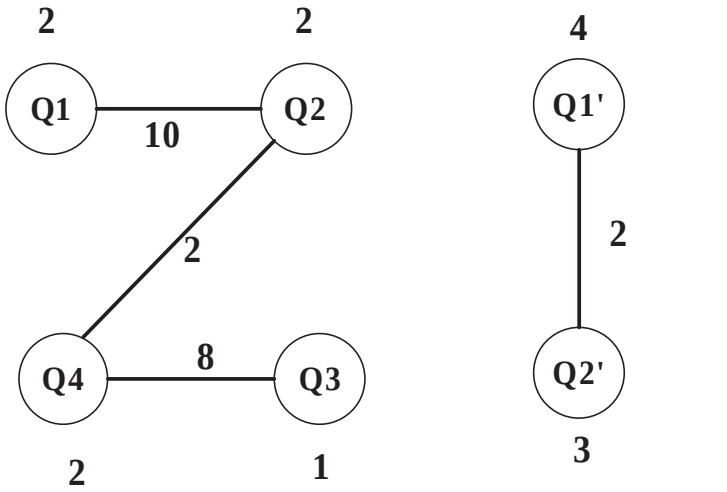
### The Bottom-up Phase

The bottom-up phase is essentially the preparation phase for query distribution: it gathers the necessary information and makes it ready for the top-down phase, which actually allocates queries to entities. Assume initially queries reside at the entities through with they are submitted to the system. Each leaf coordinator first collects the queries from the entities in its domain. The original locations of these queries are tagged for future reference. A query graph is then generated by each leaf coordinator for the queries collected. Recall that a vertex in the graph represents a query and the weight of it is the estimated work load to evaluate it in the basic entity. The weight of an edge between a pair of vertices is estimated based on the overlap of their data interest as discussed in 4.2.2. Later we will see that the data interest of a

query can be represented using a *routing query* (essentially a selection query), which will be presented in detail in Section 5.2. For now we assume the weight of an edge is the estimated arrival rate of those tuples that requested by both queries. With the query graph established, the coordinator passes this information to its parent in the coordinator tree, which integrates query graphs from children to construction its own query graph after receiving all query graphs from its children. Note this process can run in parallel in different subtrees thus accelerates the bottom-up phase. This process continues upwards until the root coordinator is reached, which will build a query graph containing all the queries submitted to the system.

One thing special about our algorithm is that instead of passing the entire query graph to its parent coordinator in the coordinator tree, each coordinator coarsens the query graph before submission. By the word "coarsen" we mean that some subsets of the vertices are collapsed, using a new vertex to represent that subset in the original graph. The weight of the new vertex is the weight sum of the collapsed vertices and the data interest of the new vertex is the union of that of the collapsed vertices. The routing query of this vertex is obtained by aggregating the routing queries of these vertices. Those edges internally connecting the collapsed vertices are omitted and the weights of those external edges are updated accordingly. The new query graph is more "coarser" compared to the original one and the number of vertices is reduced. Figure 4.6 is a simple example to illustrate query graph coarsening. The graph has four vertices, standing for four queries. The weights of vertices and edges are drawn

around them. Since $Q1$ and $Q2$ have great mutual data interest, reflected by the "10" edge weight, they collapse to form a new vertex $Q1'$ in the coarsened query graph. Similarly, $Q3$ and $Q4$ forms a new vertex $Q2'$ in the new graph. After coarsening, the number of vertices is reduced to 2 from originally 4 and the weight of a new vertex is the sum of the weights of vertices constituting it.




(a) Before coarsening (b) After coarsening

Figure 4.6: Query Graph Coarsening

There are two important issues regarding graph coarsening: the coarsening degree, i.e. how coarse is the new graph and how to choose subsets of vertices to be coarsened.

- The coarsening degree is a tunable parameter and we use a threshold $v_{max}$ to control it. Assume the number of vertices in the original graph is $|V|$. If $v_{max}$=1, all the vertices are collapsed into one and the new graph is the coarsest one, i.e. it provides little information about the original graph. On the contrary,

if $v_{max}$=$|V|$, then all vertices are reserved in the new graph and it retains all the information of the original one. Clearly, the value of $v_{max}$ should be in the range of 1 and $|V|$ to reduce the scale of the query graph while retaining useful information. In our scheme, we set $v_{max}$ as $|V|/f$, where $f$ is the fanout of the parent coordinator in the coordinator tree, i.e. the number of coordinators in a domain on the coordinator layers. The underlying heuristic of this choice is to make the average number of vertices in the query graph constructed by each coordinator equal.

- To choose the subsets of vertices to be collapsed, we partition the graph using traditional graph partitioning algorithm [SKK03] into $v_{max}$ partitions. This algorithm ensures the sum of edge weights of those edges connecting vertices from different partitions is minimum. That means the edges connecting vertices in the same partition have relatively larger weights. Since edge weights denote the commonality of data interest between vertices, this partition algorithm clusters similar queries into one partition. The intuition is that vertices with great commonalities tend to be assigned together to an entity or coordinator in the second top-down phase.

### The Top-down Phase

The top-down phase of the query distribution algorithm is to partition the query graph, assigning each partition to child coordinators or entities to implement query distribution. When the root coordinator has constructed the global query graph, the top-down phase of query distribution starts. The root coordinator partitions the graph into $f$ partitions, one for each of its children. The partitioning is done based on the total computational capabilities of the entities within the scope of each child coordinator to achieve load balancing. An algorithm similar to the one in [SKK97] is used. This algorithm claims to partition the graph into partitions with pre-determined vertices weight sum while maintaining the edge cut of different partitions small. Once the partitions are ready, they are distributed to the child coordinators based on the estimated workload of each partition. For each child coordinator receiving a partition of the query graph, what it does is to "uncoarsen" the subgraph assigned to it one level back. For each vertex in the coarsened query graph, it maintains the information like what it is composed of, the origins of the vertices constituting this vertex, etc. "Uncoarsening" refers to the process in which each coarsened vertex is converted and represented by the vertices composing it. It is the opposite process of "coarsening", resulting in a finer-grained query graph with more vertices in it. After the vertices are uncoarsened, the resulting graphs is partitioned as done in the root coordinator.

This procedure repeats at each level downwards until the entity layer is reached,

which indicates the complete of the query distribution algorithm. Again this procedure can run in parallel for different subtrees of the coordinator tree to reduce running time.

After running the query distribution algorithm, the actual migration of queries happens at the entity layer. At the end of this process, the load of queries allocated to each entity conforms to its processing capability as the query graphs are always partitioned with explicitly specified loads for each partition at each level of the hierarchical coordinator tree. Moreover, queries are distributed to different regions of the entity overlay network with minimal overlaps of data interest between different regions at each level. This clustering effect helps to maintain dataflow locality when data streams are disseminated from data sources. Hence our initial objectives of query distribution are perfectly achieved by the two-phase query distribution algorithm.

### 4.2.4 Online Query Routing

Unlike prior studies which assume queries are relatively stable or updated infrequently, our system addresses the problem of streaming queries. Queries are submitted to the system during runtime and due to the large scale of our system, they forms the query stream. To solve this problem, we employ a query routing algorithm using our hierarchical coordinator tree.

Note that after the initial query distribution phase, every coordinator in the coordinator tree has knowledge about the queries running in its domain, in the form of a coarsened query graph. For example, the root coordinator has knowledge about all the queries running since its domain is the whole system. This information at each coordinator is utilized for online query routing, which is another benefit of our query distribution algorithm. A newly-submitted query is first routed to the root coordinator. Based on the information of the queries running at each subtree, the root coordinator routes the query to one of its children, whose running queries have the greatest commonalities with the new query. The routing then continues level by level downwards until the query is assigned to an entity on the entity layer. Note that we relax the load balancing constraint on each entity during query routing tentatively in order to reduce the complexity of query routing thus to adapt to the fast query stream. The adaptive query redistribution can easily notice the unbalancing workload and redistribute workload among entities.

### 4.2.5 Adaptive Query Redistribution

During runtime, the characteristics of data streams are likely to change, like stream rate. Hence initial allocation of queries may become suboptimal. Moreover online query routing may introduce workload imbalance among the entities. Therefore, adaptive adjustment of query distribution during runtime is necessary. Again we

utilize the hierarchical coordinator tree and employ a hierarchical scheme. The adaptation operates in rounds and each round is initiated periodically by the root coordinator.

After making query redistribution decisions at own layer, the root coordinator transfers the change of each partition to each of its children. The child coordinator obtains the finer-grained information of the vertices newly allocated to it by uncoarsening them. Then the same procedure is carried out by the child coordinator to make query redistribution decisions. This process continues until the leaf coordinator finishes its query redistribution. Again, the actual migration of queries happen after all redistribution decisions are made and is done in the entity layer.

The adaptive redistribution algorithm in each coordinator is composed of two phases: load re-balancing followed by distribution refinement.

### Load Re-balancing Phase

In the load re-balancing phase, each coordinator tries to re-balance the load among its children. However, there are a few other considerations besides re-balancing the load:

- Minimize the overlap in data interest between different partitions. This is one of the objective of the query distribution algorithm and the load re-balancing phase should not compromise the quality of the partitions too much in order to balance the load among partitions.

- Minimize the query migration time. Since each query may contain stateful operations during runtime, like MAX, MIN, AGGREGATION, etc., the state of these operators have to be migrated together with the query. To avoid large overhead, we should minimize the number of queries that need to be reallocated.

To re-balance the load, there are a few possible approaches. One approach is to repartition the query graph from scratch. This approach can achieve good partitioning quality. However not only the decision making time is relative large, but also the query migration time is unacceptable due to the large number of query reallocations. Another alternative is to remove some vertices from the overloaded partitions and add them to some underloaded ones, without considering the commonalities of data interest between these partitions. This approach can achieve small query migration time and decision making time. However, communication efficiency might be unsatisfactory due to the deterioration of query clustering.

We employ a diffusion approach, which is a compromise of the two extreme approaches mentioned above. In the diffusion approach, the move of vertices are restricted to those between connected partitions, i.e. only those vertices that have edges connecting vertices in another partition are allowed to move to that partition. Under this guideline, queries are migrated to those partitions that have some commonalities with them and the effect of query clustering is preserved. Our query redistribution algorithm is presented in Algorithm 1. The "diffusion solution" specifies the load $m_{ij}$

that should be migrated from a partition $S_i$ to another partition $S_j$ for each $i, j$. We adopt the method proposed in [HB95] to derive a diffusion solution, such that the Euclidean norm of the transferred load is minimized which reflects small number of query movements.

**Algorithm 1**: 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 ← vertices in S_i whose benefits differ up to x% from the largest
          benefit;
6         V_d ← the dirty vertices in V;
7         if V_d = ∅ then V_d ← V;
8         Migrate the vertex v ∈ V_d from S_i to S_j such that it is of the largest
          load density and m_ij is larger than 90% of its weight ;
9 end
```

In this algorithm, several factors are taken into account when deciding which vertices are to be migrated.

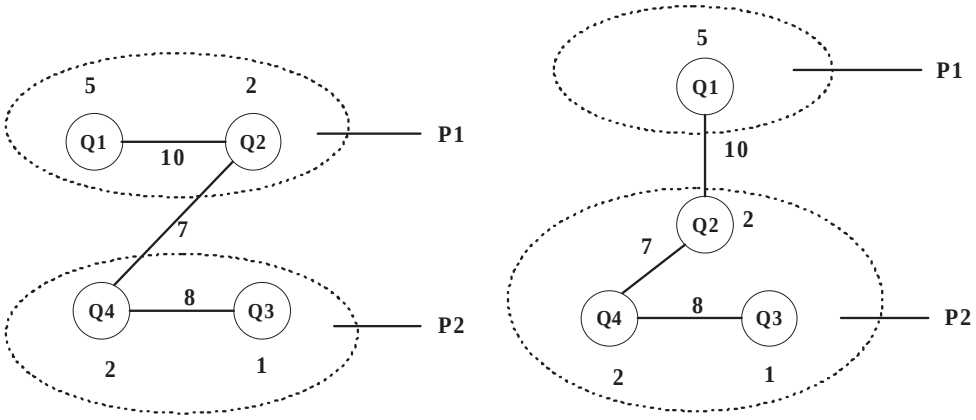
- Benefit of migration. The benefit of a vertex migrated from $S_i$ to $S_j$ is equal to the amount of weighted edge cuts that can be reduced by the migration. To achieve good partition quality, our algorithm tends to migrate those vertices with large benefits.

- Dirty vertices. As migration of vertices is carried at the end of each round, a vertex is called *dirty* if it has been decided to be migrated in the earlier iterations of the adaptation round. We give these dirty vertices higher priority

for migration in the following iterations as migrating them again would not increase additional query migration cost

- Load density. *Load density* of a vertex is equal to the weight divided by the size of its state. We favor migrating the denser ones because it may result in less state movement. The value of $x$ in line 5 can be used to trade partition quality for lower migration cost. With a larger $x$ value, we can consider more vertices with lower migration benefit.

Figure 4.7 is a simple example to illustrate load re-balance. There are four queries which are groups into two partitions, namely $P_1$ and $P_2$. However, during runtime the load of $Q1$ increases to 5 and load imbalance among these two partitions occurs. To resolve this problem, $Q2$ is migrated from $P_1$ to $P_2$. The benefit of this migration is 7.




(a) Before Load Re-balance (b) After Load Re-balance

Figure 4.7: Load Re-balance

**Distribution Refinement Phase**

The distribution refinement is carried out after the load re-balancing phase. This phase tries to see whether the quality of the current partitioning can be improved without violating the load balance constraint. It aims to reduce the weighted edge cut while maintains the load balance condition inherited from the load re-balancing phase. Again the border vertices are visited randomly and checked to see whether it has any of the properties:

1. Migrating the vertex back to its original partition can maintain load balance and the weighted edge cut remains the same.

2. Migrating the vertex to another partition can decrease the current weighted edge cut without violating load balance constraint.

3. Migrating the vertex to another partition can improve the load balance while maintaining the current weighted edge cut.

If a vertex has one of the properties above, it is migrated.

## 4.3 Chapter Summary

In this chapter, we present the detailed design of the query layer. The query layer provides query management service in the system. Its task can be decomposed into several subtasks, like initial query distribution, online query routing and adaptive

query redistribution. These tasks are carried out by various coordinators in the system, which are organized in a hierarchical structure. We model the query distribution problem as a graph partition problem and devise a novel hierarchical graph partition algorithm to distribute queries to entities, leveraging the hierarchical structure of the coordinators. This algorithm not only achieves the goals of query distribution, namely, load balance among the entities and minimizing communication cost, but also naturally distributes the necessary information needed for online query routing and adaptive query redistribution to those coordinators. The online query routing algorithm can efficiently handle streaming queries and assign them to the most appropriate entity for processing. The adaptive query redistribution algorithm addresses the problem of suboptimal performance due to changing conditions at runtime. It allows the system to adjust the query distribution to adapt to runtime environment so as to achieve optimal performance.

# Chapter 5

# Data Layer Design

The data layer, as the name suggests, focuses on how data in the system is managed. It provides two types of services essentially: client query processing and data stream dissemination. Client query processing is the basic function of our system and is supported by various stream processing engines installed at each entity. Despite of the heterogeneities of the stream processing engines, they are capable of evaluating queries submitted to the system with newly generated data and deliver query results to the respective clients. Data stream dissemination is a prerequisite for the query processing service, as it transfers the necessary data to individual entities for query evaluation. In this thesis, we focus on the data stream dissemination service provided by our system and rely on the stream processing engines to provide the client query processing service.

Once the locations for running the queries are fixed, we need to decide how data

streams are transferred from source nodes to the place where they are consumed. This is the data stream dissemination service provided by the data layer. A naive source-based approach for data stream dissemination is all data streams are disseminated from source nodes directly to stream processing entities. As mentioned in 3.1.2, entities in our system cooperate on data stream dissemination by participating in the cooperation trees rather than relying on the source nodes solely (Please refer to section 3.1.2 for a discussion of these two algorithms). To avoid flooding the network, each parent node in the cooperation trees only disseminates the data interesting to each of its descendants. In the rest of this chapter, we will see how cooperation trees are constructed and how data interest of entities is represented in our system to facilitate data filtering.

## 5.1 Cooperation Trees Construction

For each source node, one cooperation tree is constructed, with the source node as the root of the tree. Based on its data interest, an entity decides whether to participate in one particular cooperation tree.

Each tree is generated dynamically as follows. Firstly, a degree constraint is posed to each node based on its capability. This degree decides the maximum child nodes that entity can have for data dissemination. As data dissemination consumes system resources, the more powerful an entity is, the more number of child node it can

support. Each cooperation tree is constructed dynamically, with nodes joining and leaving meanwhile. Let us see two scenarios for node joining/leaving the cooperation tree respectively.

### 5.1.1 Joining a Cooperation Tree

Joining a cooperation tree occurs when a new query that requests a stream from a source $s$ starts running at a node $n_i$ while $n_i$ is not a member of that cooperation tree. We say $n_i$ is not covered by that particular cooperation tree rooted by $s$. $n_i$ initiates a request to join that tree to the source node $s$. If $s$ still has an available degree, $n_i$ would be directly placed under $s$. Otherwise, $s$ would search for a child node $n_j$ that has the least communication latency with $n_i$. If the latency between $s$ and $n_i$ is smaller than that between $s$ and $n_j$, $n_i$ will be placed as the child of $s$ and $n_j$ will be downgraded as a child of $n_i$. Otherwise, $n_i$'s request will be directed to $n_j$ who will then repeat the above procedure. The whole procedure stops when $n_i$ is added to the tree. This whole process is illustrated by Algorithm 2.

### 5.1.2 Leaving a Cooperation Tree

A node may leave a cooperation tree when the data disseminated from $s$ in this tree is no longer necessary for that node. Similar to joining the tree, a request to leave the tree is initiated by the particular node. The request together with the information

**Algorithm 2**: HandleJoiningRequest(rq)

```
1  begin
2      if there is a free output degree then
3          Add the requesting processor n_i as a child node in the cooperation tree;
4          Reduce the available degree by 1;
5      else
6          Find the child node n_j that is closest to n_i;
7          if n_i is closer to me than n_j is then
8              Add n_i as a child node;
9              Make n_j a child node of n_i;
10         else
11             Run HandleJoiningRequest(rq) at n_j;
12 end
```

of $n_i$'s children is sent to $n_i$'s parent, say $n_j$. $n_j$ will select one of $n_i$'s children $n_k$ to replace $n_i$ such that $n_k$ has the lowest communication latency with $n_j$. The other children of $n_i$ will automatically become child nodes of $n_k$ and will be notified about this change from $n_k$. To avoid frequent joining and leaving due to change of running queries, each entity will be locked in the trees for a period after it issues a request for leaving to $s$. Only after this period when it still does not need those data streams, would it be allowed to leave.

**Algorithm 3**: HandleLeavingRequest(rq)

```
1  begin
2      Find the child node n_k of the requesting node n_i that is closest to this node;
3      Delete n_i from child nodes;
4      Add n_k as a child node;
5      Notify n_k and other child node of n_i about the change to update
       information;
6  end
```

## 5.2 Routing Query Constructions

Each cooperation tree is composed of entities with mutual data interest in the data streams generated by the tree root. However, that does not mean those entities are interested with every update of those data streams. Therefore, transferring every update to all the entities may result in redundant communications and waste network recourses. In our system, we use *routing queries* to represent the data interest of each entity and only those tuples that can pass the routing query are delivered to the corresponding entities. Note that routing queries are not queries that submitted by clients. Instead, the routing query for an entity is derived from the queries running at it. In this section, we will see how routing queries are constructed from the queries running at one entity and how they facilitate data dissemination in the system.

For each entity, there are two types of routing queries: the local routing query and the tree routing query.

- A **Local routing query** is constructed using queries running in the stream processing engine of the entity. It essentially provides a filtering mechanism on top of the engine; only those data tuples needed to evaluate the running queries are routed to the engine for further processing.

- **Tree routing queries** are used to route data to the child nodes in the cooperation tree. There is one tree routing query for each child, which summarizes the

data interest of the entire subtree rooted at that child node. With this knowledge, a node can decide whether an incoming data tuple should be forwarded to one child node based on its tree routing query.

These two types of queries are closely related. Local routing queries are the bases for constructing tree routing queries. Under our scheme, each interior node of the cooperation tree submits a tree routing query to its parent which is an aggregation of the tree routing queries from its children and its local routing query.

Data interest of each query is specified by its selection predicates (normally in the "where" clause for SQL-like queries). One natural way to construct a local routing query is to extract the selection predicates from the local running queries and incorporate them using disjunctions. Since each tree routing query is the union/aggregation of all the local routing queries of the entities in the subtree(including its own), those nodes at the top of a cooperation tree thus have a large number of selection predicates in their tree routing queries. This imposes a high filtering and maintenance workload to those entities. Some work on data filtering [FJL+01, SDR03] proposed some optimization techniques like sharing evaluation of predicates etc. to alleviate this problem. However, they are not efficient in our context due to the large scale of our system. The large number of queries submitted by clients accounts for the large number of unique selection predicates which renders those optimizations inefficient.

Our system constructs routing queries using another approach, which is based

on data space partition. We partition the data space into multiple subspaces by dividing each data stream into multiple substreams. Those substreams are denoted by SS=$\{ss_1, ss_2, ..., ss_{|}ss|\}$. Local routing queries are represented as a bit vector $v \in \{0,1\}^{|SS|}$, where $|SS|$ is the total number of substreams. The value of an element of $v$ is:

$$v[i] = \begin{cases} 1 & \text{if this query has data interest in substream } ss_i, \\ 0 & \text{otherwise.} \end{cases} \tag{5.1}$$

Under this scheme, it is very easy to implement query aggregation. A simple "OR" operation on two bit vectors results in the union of them. Tree routing queries are constructed by using union of local routing queries. Thus a tree routing query for an entity is constructed by repeating the "OR" operation on all the tree routing queries(vectors) of its child nodes as well as its local routing query. This can be done efficiently as only bit operations are involved. Also with this presentation, all routing queries are of the same size, regardless how many descendants one node may have in the cooperation tree. Furthermore, compression techniques like [Teu78, BK91] can be easily applied to reduce the size of the bit vectors to save space.

When a tuple arrives at one node, it is matched against its tree routing queries of its children and then sent to those child nodes whose routing queries request that tuple. Note this checking is composed of two steps: 1) search for the subspace(the substreams) that covers a specific point(the tuple) in the data space; 2) check whether

any of the positions representing these substreams in the vector has a "1" value for each routing query. Step 1 can be solved by existing techniques, such as R-Tree [Bec90] and step 2 is straight forward, with some checking on the bit vectors.

We can see that there is a tradeoff between filtering power and filtering overhead. Filtering power refers to the ability to identify those irrelevant tuples of the data streams for forwarding while filtering overhead is the computation cycles(system resource) allocated to do the extra filtering at each node. To achieve the highest filtering power, we should partition each stream to as many substreams as possible. For example, we can partition a stream into many substreams by using different combinations of distinct attribute values of the stream. Let us see an example. Suppose the original stream has two attributes(say a and b), each can take two distinct values(say 1 and 2) which follows a uniform distribution. To obtain extreme filtering power, we can partition it to 4 substreams, as illustrated in Table 5.1. As they are the smallest substreams that can be derived, we call them the *finest* substreams. With this partition, only those tuples having exact matching values to the selection predicates of the child nodes are forwarded and the rate of each such substream is only one quarter of the original rate. Thus a lot irrelevant tuples are filtered out. On the contrary, let us consider another conservative partition strategy that only 1 substream is needed, i.e. no partition is required. In this case, no matter what values the attributes of a tuple from the stream may take, as long as a child node has running queries requesting data on either attribute $a$ or $b$ of this stream, this tuple is forwarded to it, with actually

only 1/4 the chance that this tuple is indeed relevant.

| Substream No. | Attribute values | Rate(Ratio of the original Rate) |
|---|---|---|
| 1 | a = 1 and b = 1 | 0.25 |
| 2 | a = 1 and b = 2 | 0.25 |
| 3 | a = 2 and b = 1 | 0.25 |
| 4 | a = 2 and b = 2 | 0.25 |

Table 5.1: Substreams

Though filtering power is a desirable property, too many substreams may impose too much overhead on the operations of routing queries. For example, based on the partition strategy like 5.1, the number of those finest substreams is exponential to the number of attributes, with the distinct values of each attribute as the base. Clearly this is not viable in reality for our system where thousands of data streams, each of which may take hundreds of values, are generated continuously. In [RLW+02], a few clustering algorithms are studied in the context of clustering cells(analogous to the finest substreams in our problem) of a regular grid(analogous to the original streams) in event space for a content-based publish/subscribe system. We adopt the K-Means cluster algorithm in our system to cluster these finest substreams into coarser grained ones, which is claimed to perform the best in [RLW+02]. Although we use static clustering in our experiments, it can be easily to be turned into an adaptive scheme by periodically invoking the K-Means algorithm, which is iterative inherently.

## 5.3 Chapter Summary

In this chapter, the design of the data layer of our system is presented. The data layer mainly provides two types of services: client query processing and data stream dissemination. Client query processing services is backed by various stream processing engines installed in each entity. Each entity notifies a client when new results of his query is available. The data dissemination service is our focus in this thesis, which is neglected in previous work. We observe that the source nodes may become the bottleneck of the whole system if we rely solely on them for data dissemination. Thus we employ a cooperative approach to solve this problem. Entities in the system participates in cooperation trees based on its data interest and cooperate in data dissemination. We present the cooperation tree construction algorithm in 5.1. To avoid flooding the network, data interest of an entity is represented by a local routing query. The data interest of a subtree of the cooperation tree is represented by a tree routing query, which is the aggregation of the local routing queries of entities in the subtree. Only data that passes those routing queries are forwarded.

# Chapter 6

# Experimental Study

To study the performance of various proposed techniques in our system, we implemented a simulator using C to simulate the communication between stream processing entities. In this chapter, we present the experimental results obtained by running simulations. First the experiment settings will be introduced. Then we demonstrate the results of query distribution, including initial query distribution, adaptive query redistribution and query routing in section 6.2. Section 6.3 shows the results for cooperative stream dissemination. A short summary concludes this chapter afterwards.

## 6.1 Experiment Settings

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. For

a detailed discussion of the Transit-Stub model, please refer to the user manual at http://www-static.cc.gatech.edu/projects/gtitm/. There are three types of nodes in the topology: data sources, stream processing entities and routers. We randomly choose 100 and 256 nodes for the first two types respectively and the rest acts as routers. We have repeated experiments on different topology with 4096 nodes and different data sources and entities. The results of these experiments are similar to the one presented in this section.

Table 6.1 summarized the various parameters of our system and their default values.

| Parameter Name | Description | Default Value |
|---|---|---|
| $N$ | Total number of nodes in the topology | 4096 |
| $S$ | The number of nodes generating streams | 100 |
| $E$ | The number of stream processing entities | 256 |
| $C$ | Coordinator Cluster size in the coordinator tree | 4 |
| $F$ | Fan-out for the cooperation tree | 8 |
| $\|SS\|$ | Total substream number | 20,000 |
| $r_i$ | Stream rate for substream $i$ | 1-10 bytes/second |
| $g$ | Client query groups | 20 |
| $qs_i$ | The number of substreams requested by a query $i$ | 100-200 |
| $q$ | The number of queries in our system | 5,000-60,000 |
| $a\_time$ | The interval between two adaptive rounds | 50 seconds |

Table 6.1: System Parameter

Note that all the streams in our system are pre-partitioned into 20,000 substreams and are randomly distributed among 100 source nodes. We introduce the concept of client query groups to simulate the clustering effect of user behaviors. Each group

has different data hot spots and is represented by a permutation of the substreams. The number of substreams requested by a query is uniformly chosen between 100 and 200 and the substreams requested by queries within a group follow a zipfian distribution with $\theta = 0.8$. Different $\theta$ and $g$ are tested and the results are similar. The workload of a query is simulated by the sum of input stream rates, which varies form 1 byte/second to 10 byte/second randomly. Number of queries in our system various from 5,000 to 60,000, to test performances under different system load. All simulations are run on a Linux server with an Intel 2.8GHz CPU.

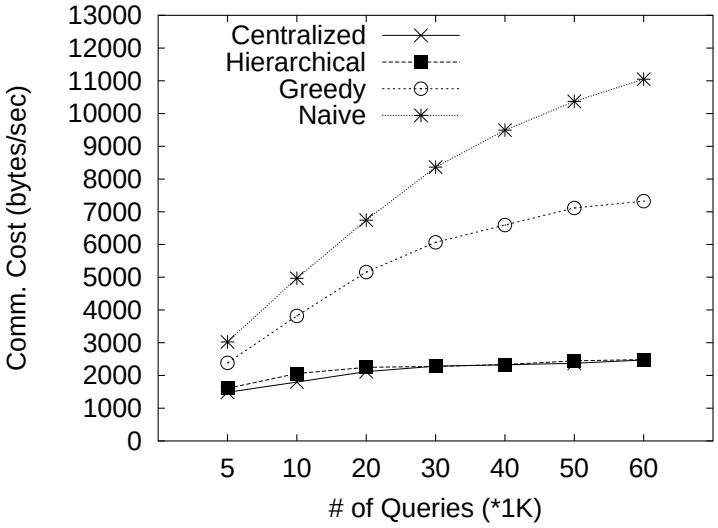
## 6.2 Query Distribution

### 6.2.1 Initial Query Distribution

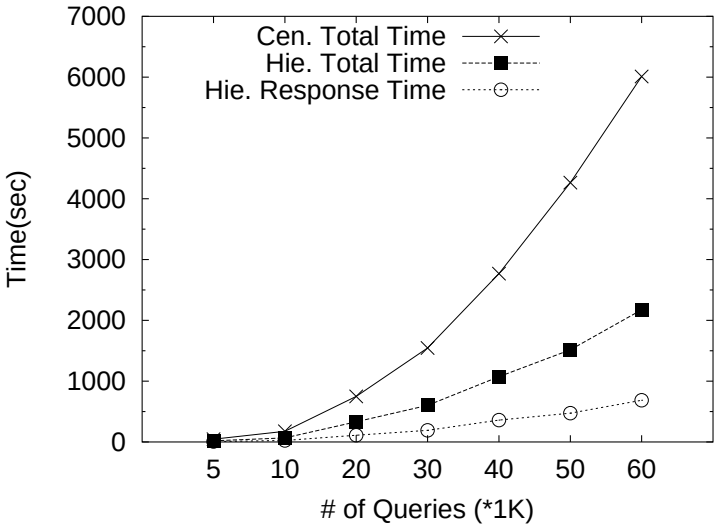
In this experiment, we look at the performance of the initial query distribution scheme. Recap that we introduce a novel hierarchical graph partitioning algorithm for initial query distribution. To illustrate its efficiency, it is compared with other three approaches:

1. Naive. This approach focuses on the load balance among nodes. It distributes queries to entities in a way that the workload of entities are balanced while it does not consider the data interest of queries during distribution.

2. Greedy. This approach assigns queries to entities one by one and takes a greedy




(a) Comm. Cost




(b) Running Time

Figure 6.1: Initial Distribution

approach such that a new query is assigned to an entity with the least additional communication cost. Also for each node, a load limit 4.1 is imposed to ensure the load balance constraint is not compromised.

3. Centralized. This approach designates a central node which collects all queries and distributes them using a centralized graph partition algorithm. Both load balance and data interest of queries are taken into account.

The metric for evaluating the query distribution schemes is unit-time communication cost. It is calculated by adding up the values of the per unit message transfer rate on each link times the latency of this link. This metric is widely used in network-related research. Figure 6.1(a) depicts the unit-time communication cost for all four approaches. We can see that the Naive approach (denoted by asterisk) performs the worst. This is because the Naive approach ignores the data interest of queries when allocating queries, which misses out the opportunity for further optimization. The Greedy approach (denoted by blank circles) works much better than Naive does as it considers the data interest of queries during query distribution. Nevertheless, the two graph partition approaches, namely centralized and hierarchical approach (denoted by X and solid box respectively in the figure), perform similarly and are far better than the rest. The reason is that query distribution in these two approaches is done with a comprehensive knowledge of all the queries in the system while the Greedy approach assigns one query at a time. Moreover, the performance of our hierarchical

approach is very close to the centralized approach. This indicates the error introduced by the query graph coarsening is negligible.

The centralized approach results in smallest communication cost and is the best approach from partition quality perspective. To further compare the centralized approach with our scheme, the response time and total time of these two approaches are recorded and presented in Figure 6.1(b). Note the response time and total time for the centralized approach is the same as one node is responsible for the whole process while the response time for our approach is smaller than the total time due to parallelism. From the graph, it can be seen that both the response time (denoted by blank circle) and total time (denoted by solid box) for our hierarchical query distribution algorithm is much less than that of the centralized approach (denoted by X). Its response time is smaller because query distribution under different subtrees of the coordinator tree can run in parallel. To understand why the total time for the hierarchical approach is smaller, let us see an example. Suppose the query graph contains $n$ vertices. The complexity for graph partitioning algorithm is in $O(n^2)$. Therefore the total time to partition this graph is $k * n^2$ where $k$ is a constant. Now if we divide the graph into two subgraphs with vertices numbers $n_1$ and $n_2$ respectively and then run the graph partition algorithm. In this case the total time to partition the graph is $k * (n_1^2 + n_2^2)$, which is smaller than $k * n^2$. In our hierarchical algorithm, each coordinator partitions a subgraph of the original graph in a similar way thus the total time of it is also smaller than the centralized approach. One point to note that since

the complexity for graph partitioning algorithm is in $O(n^2)$, the centralized approach may not be practical when the number of queries in the system is huge. However, the response time of the hierarchical query distribution algorithm remains viable.

### 6.2.2 Adaptive Query Redistribution

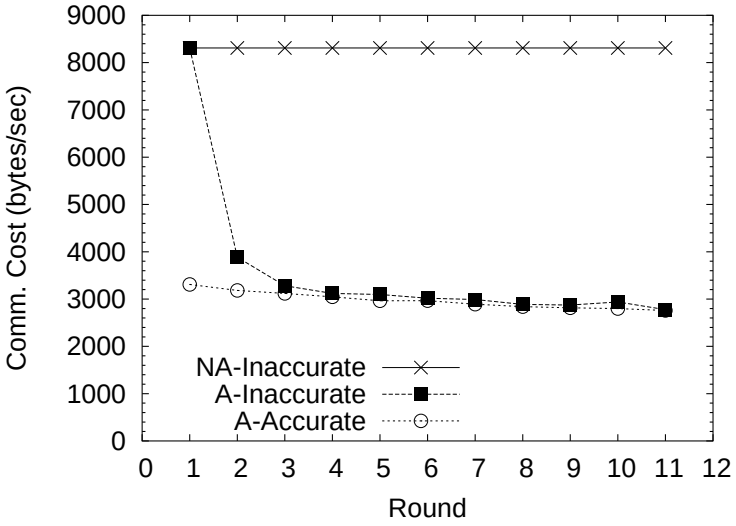
In this section, we present two experiments designed to study the effectiveness of the adaptive query redistribution scheme.

In the above experiment, we assume accurate statistics can be obtained to facilitate the query distribution. However, in reality apriori statistics are hard to collect and may be obsolete during runtime. We rely on our adaptive query redistribution algorithm to dynamically adjust the query distribution to cope with this situation. In this experiment, we use a random initial query allocation scheme to model the effect of inaccurate statistics. Three situations are examined in this experiment:

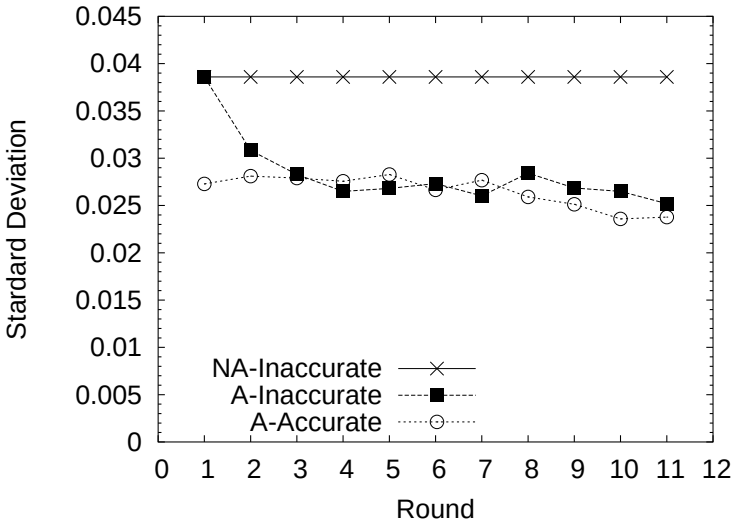
1. NA-Inaccurate. The apriori statistics are inaccurate and no adaptation is carried out.

2. A-Inaccurate. The apriori statistics are inaccurate and adaptive query distribution is done in rounds.

3. A-Accurate. The apriori statistics are accurate and adaptive query distribution is done in rounds.




(a) Comm. Cost




(b) Standard Deviation of Load

Figure 6.2: Adapting to inaccurate statistics

In this experiment, we assume the statistics are static. From the Figure 6.2(a) and Figure 6.2(b), we can see both communication cost and the standard deviation of workload are refined with the number of adaptation rounds: The communication cost keeps decreasing and the workload distribution is more balanced. This experiment shows the capability and effectiveness of our adaptive query redistribution algorithm to cope with inaccurate statistics.

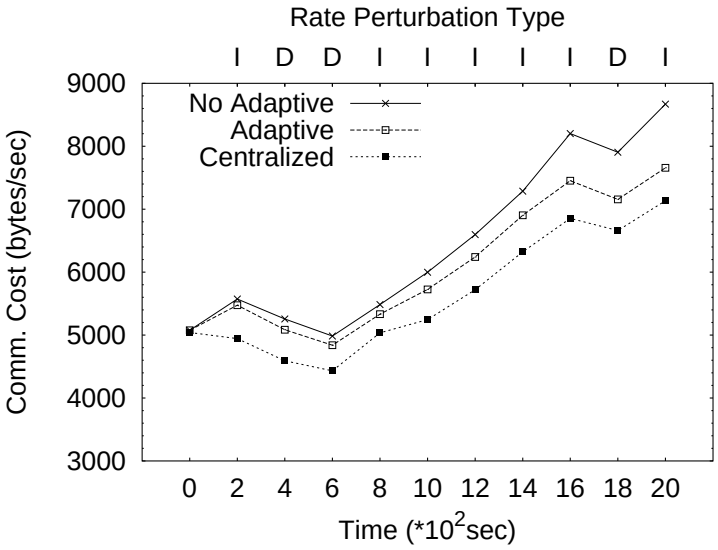
In the second experiment, we drop the assumption that the statistics are static to model the fluctuation of stream rates during runtime. In the simulation, we randomly choose 5% of the total substreams, i.e. 1000 substreams, and increase(denoted by "I") or decrease(denoted by "D") their rates by a factor of 10. Note load imbalance among the entities occurs when data stream rate changes as workload is estimated by the rate of incoming streams. Again three schemes are compared in this experiment:

1. No Adaptive. As the name suggests, no adaptation is done.

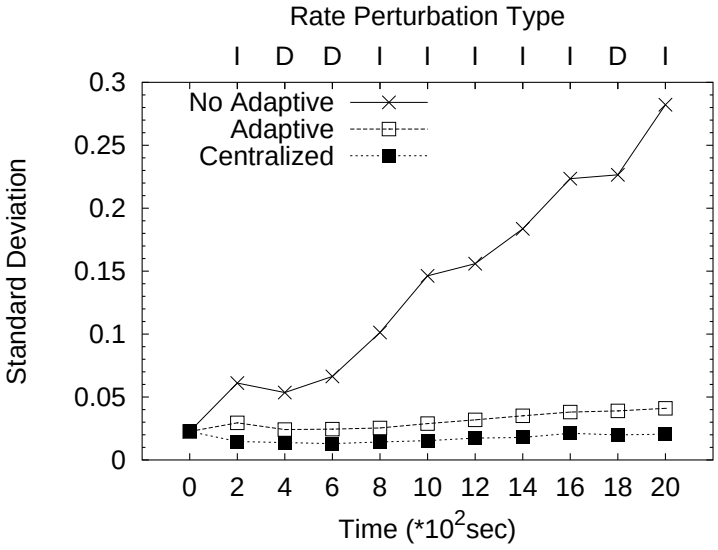
2. Adaptive. Use our adaptive query redistribution algorithm to adjust query distribution.

3. Repartitioning. Centralized graph partitioning algorithm is used to repartition the query graph from scratch. This approach represents the best result of query distribution for each particular time point.

Figure 6.3(a) and Figure 6.3(b) illustrate the communication cost and the standard deviation of the load in the system. The statistics are recorded every 200 seconds,




(a) Comm. Cost




(b) Standard Deviation of Load

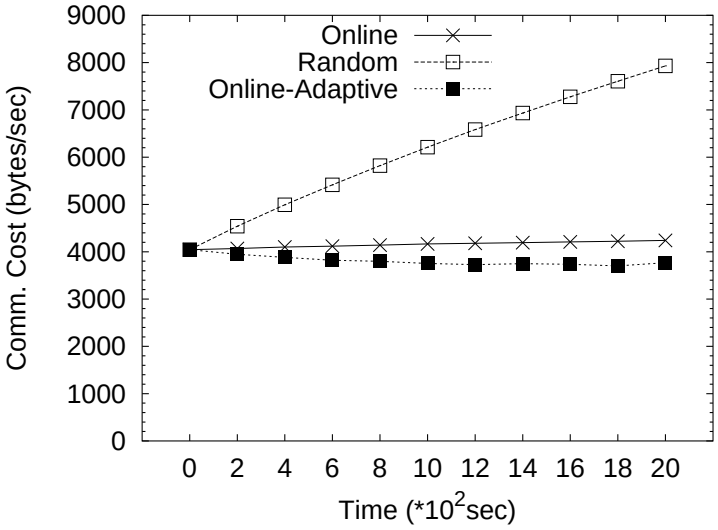
Figure 6.3: Perturbation of Stream Rates

right after one adaptation round(remember the adaptation interval is 50 seconds). As expected, the No Adaptive scheme (denoted by X in the figure) performs the worst in terms of both load balancing and communication cost. The adaptive query redistribution algorithm (denoted by blank box) performs close to centralized repartitioning (denoted by solid box). With adaptation, communication cost is slightly larger than the centralized one and the load of the system is always balanced, no matter how the load of the whole system changes. With the smaller overhead compared to the centralized repartitioning approach, our adaptive query redistribution algorithm can achieve reasonably good performance thus it is preferred.

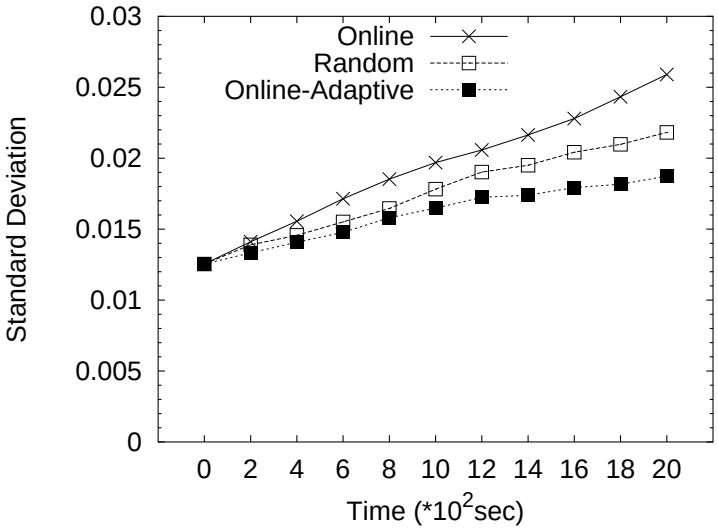
### 6.2.3 Query Routing

The experiments under this section are designed to study the performance of the query routing algorithm. In the first experiment, we consider the consequence of adding new queries to the system. Initially the system has 30,000 queries running. We add 1,500 queries into the system incrementally at a 200 seconds interval and record the statistics at the end of each interval. Two statistics are recorded, namely unit-time communication cost and the standard deviation of loads among entities. Three schemes are compared in this experiment:

1. Random. New queries are randomly assigned to entities without considering its data interest. However, the load limited imposed on each entity is not violated




(a) Comm. Cost




(b) Standard Deviation of Load

Figure 6.4: Add New Queries

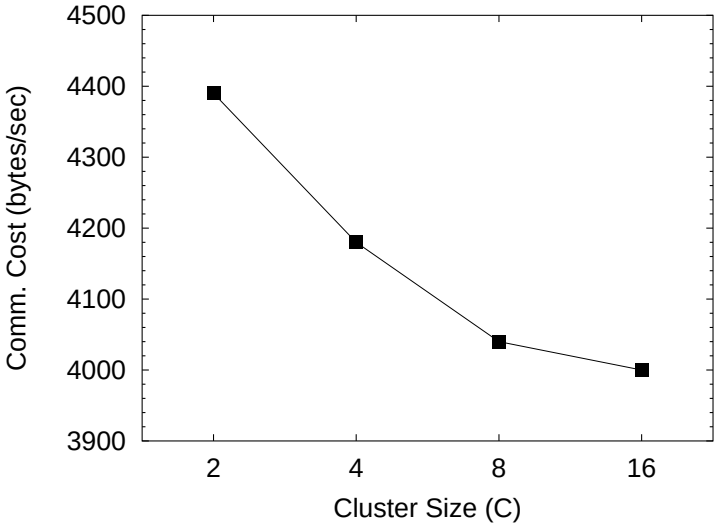
by the assignment.

2. Online. Use the online query routing algorithm described in Section 4.2.4.

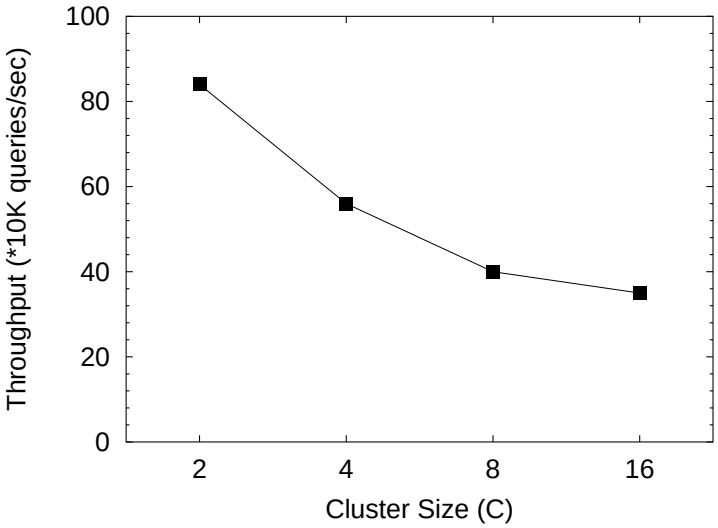
3. Online-Adaptive. Besides the online routing algorithm, adaptive query redistribution is running also.

The Random scheme (denoted by blank box) performs worst in terms of communication cost: its communication cost keeps increasing with more queries added in the system. The Online scheme (denoted by X) maintains the same communication cost while the communication cost of Online-Adaptive (denoted by solid box) drops slightly thanks to its ability to refine the query partitioning. However, in terms of load balance, Online performs worst as we allow load imbalance to exist in our online query routing algorithm to reduce routing overhead. Online-Adaptive again performs the best of three as it is able to re-balance the load distribution among nodes while sustaining the good partition quality.

In the second experiment, we want to see the scalability of the query routing algorithm to fast query streams. We vary $C$, the coordinator cluster size while keeping other settings similar to the first experiment. Two important statistics are collected: the communication cost of the system and the time needed for the root coordinator to distribute one query. The communication cost captures the quality of query distribution while the time for routing one query can be used to compute the throughput




(a) Comm. Cost



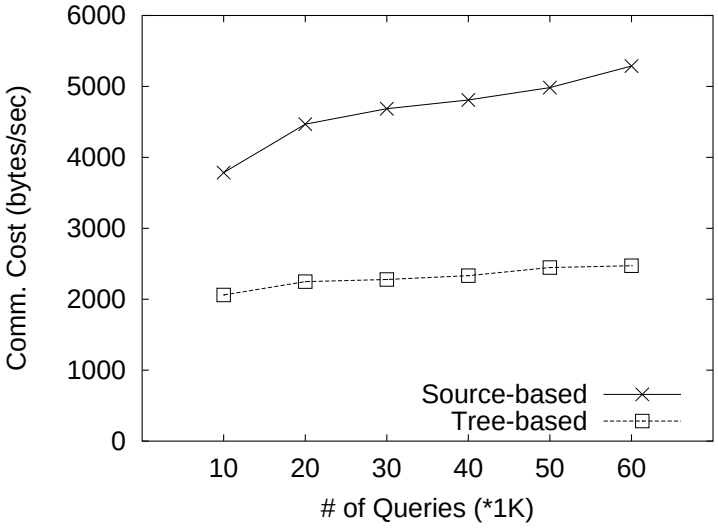

(b) Throughput of the root coordinator

Figure 6.5: Experiment on different cluster size

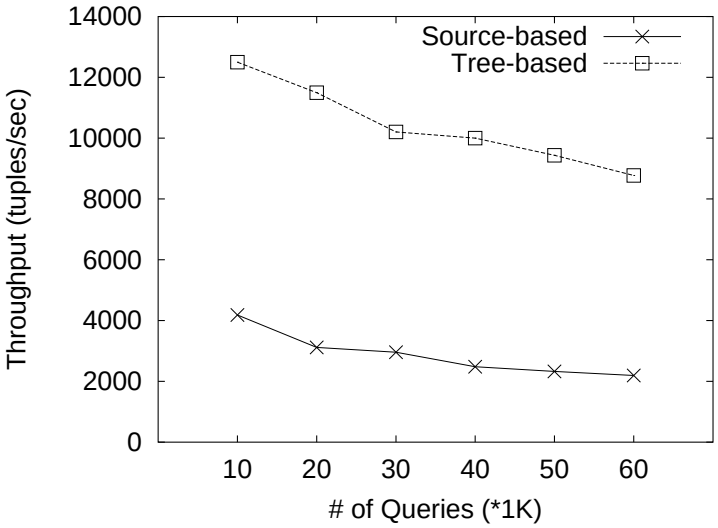
of the root coordinator, i.e. the maximum query rate the root coordinator can handle. The throughput of the root coordinator is an indication of the scalability of the query routing algorithm as the root coordinator is the only potential bottleneck of our system. Figure 6.5(a) shows the communication cost of the system with different cluster size $C$. It is shown that the smaller the cluster size, the larger the communication cost. With a smaller cluster size, the depth of the coordinator tree is larger. As query graph coarsening is done at each level, the graph at the root coordinator is coarser, which results in inferior partition quality. Figure 6.5(b) shows relationship between the throughput of the root coordinator and different $C$s. It is a inverse relationship: throughput decreases with increasing $C$. This is reasonable as with a larger cluster size, the root coordinator has more choices to route the query thus more time is needed to check the suitability of each candidate, which implies lower throughput. From the above two figures, we can see a tradeoff between query distribution quality and query stream throughput. A smaller $C$ favors high query stream throughput, with compromise in system performance. How to adaptively adjust the value of $C$ is an interesting topic for future work.

## 6.3 Cooperative Stream Dissemination

In this experiment, we examine the performance of our cooperative stream dissemination scheme. It is compared with a traditional source-based scheme, where all data




(a) Comm. Cost




(b) Throughput of source nodes

Figure 6.6: Stream Dissemination

streams are disseminated from source nodes directly to stream processing entities. Similar to our cooperation tree scheme, in this source-based scheme stream processing entities notifies the source nodes about their data interest using routing queries and only those tuples that can pass the routing queries are forwarded. Two metrics are used to compare the performance of these two schemes: the unit-time communication cost and the throughput of source nodes. The throughput of the source nodes is measured by the maximum stream rate that can be handled by the source node in the system. It is computed by:

$$Throughput = \frac{1}{max(time\ to\ disseminate\ one\ tuple)} \tag{6.1}$$

Note the time to disseminate one tuple consists two portions: the time to matching the tuple to the routing query of each entity and the time to pack and send out the tuple to those successful entities. In our simulation, we set the average time to pack and forward a tuple to be 100us, which is a relative value without diminishing the validity of our conclusion.

The results of these two metrics shown in Figure 6.6(a) and Figure 6.6(b) demonstrate the superiority of our cooperative stream dissemination scheme (it is called tree-based in the figure, denoted by the curve with blank boxes) over the source-based scheme (the curve with X). It imposes less communication cost while attains higher throughput. The reason for this is the source-based scheme relies solely on

the source nodes to disseminate stream data. For each newly generated tuple at one source node, the source node matches it against the routing queries from every entities in the system and forwards it. On the contrary for our scheme the source node has a limited number of child nodes. Clearly our scheme can achieve higher throughput. The communication cost is smaller is because the transfer of each tuple is shared in our scheme. Consider two entities which are far from a source node while are close two each other. In the source-based scheme, two packages from the source node are transferred separately. In our scheme, one package is sent out from the source node to one entity first and the receiving entity forwards the package to the other entity near it. Therefore the long route from source node to the entities is shared in some way, which results in smaller communication cost. To conclude, our cooperative stream dissemination performs better than the traditional source-based schemes, in terms of lower communication cost as well as higher source nodes throughput.

## 6.4 Chapter Summary

In this chapter, we study the performance of various algorithms proposed in this thesis. A simulation of the system is implemented and the simulation results show the effectiveness of these algorithms, which are hierarchical graph partitioning, adaptive query re-distribution, online query routing as well as cooperative stream processing. The effectiveness of these algorithms makes the future development and deployment

of our system in a real network environment viable.

# Chapter 7

# Conclusion and Future Work

## 7.1 Summary

Stream processing engines are application-independent, specially designed query engines to process high-volume, real-time data streams. In recent year, many stream processing engines have been developed and employed by business entities to provide stream processing service over the internet. However, due to the inherent limitations of those stream processing engines, these entities suffer from scalability, over-investment and availability problems. A system incorporating those entities, promoting joint cooperation can achieve better system resource utilization, economical efficiency and scalability. In this thesis, we present the architecture of a scalable distributed stream processing system made up of loosely coupled entities. The aim of this system is to provide stream processing service to clients in an Internet scale.

Our system incorporates the processing capability of each individual stream processing entity into an Internet-scale distributed stream processing system, which exploits the aggregated bandwidth and processing power. Entities in the system are loosely coupled and can be heterogeneous in terms of stream processing engines thus existing well-developed singe site stream processing engines can be utilized without much modification. Unlike previous work on distributed stream processing systems which overlooked the data dissemination from source nodes to various stream processing engines, our system provides two layers of services: the query layer and data layer.

The query layer service is to dynamically distribute queries to the most appropriate entity for processing to achieve load balance and minimize communication cost. This service is backed by a number of coordinators, which are special entities organized into a hierarchical structure. The query distribution problem is modelled as a graph partitioning problem and we leverage existing graph partitioning algorithms and derived a hierarchical graph partitioning algorithms to achieve load balance among the entities as well as minimum communication cost in transferring the data streams. The problem of fast incoming new queries (streaming queries) is addressed by employing an effective query routing scheme to route the new coming queries to a suitable entity. A runtime adaptive query redistribution mechanism is devised to adapt to the change of the environment like stream rates, user surging requests, etc. to enhance system performance during runtime.

Data dissemination is often neglected by existing stream processing systems. In

many situations, especially in the wide-area, the network is the stream bottleneck. In our system, we identify this problem and address the problem of how to efficiently transfer data streams to various geographically dispersed stream processing entities. This is one aim of the data layer service besides providing query evaluation to clients. In our system, stream processing entities are urged to collaborate in data dissemination besides evaluating assigned queries, rather than relying on the source nodes sorely in data dissemination. Cooperation trees for data dissemination are built and specially designed routing queries are employed to represent data interest of nodes, which facilitate data dissemination from one node to another selectively.

We design experiments to test the effectiveness of our proposed techniques. A simulation of the system is implemented and our simulation results demonstrate significant performance gains with respect to traditional techniques.

## 7.2 Future Work

The design of this system is complex and there are some interesting issues that we can explore in future research. Some directions are:

- We can further exploit the query distribution problem to incorporate the situation when the locations of some queries are restricted, i.e. only subsets of the entities they can be assigned to. Such cases can occur due to user QoS requirements, entity computational power as well as requirements on specially

designed stream processing engines for some queries.

- Our query distribution model does not take the communication cost between source nodes and processing entities into account. Our assumption is that the main saving in communication cost is gained from eliminating redundant data transmissions, which is achieved by query clustering, assigning one cluster to a single node. Nevertheless, we may further improve the performance by capturing source-node distances in our model.

- Currently the data space is pre-partitioned into several subspaces and this partition is static. A dynamic data space partitioning algorithm will be useful to adapt to environment changes during runtime.

- We have implemented simulations to test the effectiveness of various techniques. It would be interesting if we can develop the whole system and deploy it in a real network environment and verify the validity of the results presented here.

- More in-depth research on the business model of this system is needed, like how entities are compensated and charged, etc.

# Bibliography

[Aba03] D. Abadi et al. Aurora: A new model and architecture for data stream management. *VLDB Journal*, (2), 2003.

[AC04] Y. Ahmad and U. Cetintemel. Network-aware query processing for stream-based applications. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, 2004.

[AD93] S. Ahn and David H. C. Du. A load balancing multicast tree approach for group-based multimedia applications. In *Proc. of Intl. Conf. on Local Computer Networks*, 1993.

[AF00] M. Altinel and M. J. Franklin. Efficient filtering of xml documents for selective dissemination of information. In *Proc. of the 2000 Intl. Conf. on Very Large Data Bases*, 2000.

[AH00] R. Avnur and J. M. Hellerstein. Eddies: continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management*

*of Data*, 2000.

[ARS04] S. Agrawal, K. Ramamritham, and S. Shah. Construction of a temporal coherency preserving dynamic data dissemination network. In *The 25th IEEE Real-Time Systems Symposium*, 2004.

[Bab04] B. Babcock et al. Load shedding for aggregation queries over data streams. In *Proceedings of 20th International Conference on Data Engineering, (ICDE 2004)*, 2004.

[Ban03] S. Banerjee et al. Construction of an efficient overlay multicast infrastructure for real-time application. In *INFOCOM 2003, the 22nd Annual Joint Conf. of the IEEE Computer and Communications Societies*, 2003.

[BB03] B. Bonfils and P. Bonnet. Adaptive and decentralized operator placement for in-network query processing. *International Workshop on Information Processing in Sensor Networks(ISPN)*, 2003.

[BBDW02] B. Bakcock, S. Babu, M. Datar, and J. Widom. Models and issues in data stream systems. In *Proceeding of the 2002 ACM Symp. On Principles of Database Systems*, 2002.

[Bec90] N. Beckmann, et al. The R-tree: An efficient and robust access method for points and rectangles. In *Proceedings of the 1990 ACM SIGMOD Intl. Conf. on Management of Data*, 1990.

[BK91] A. Bookstein and S. T. Klein. Compression of correlated bit-vectors. *Information Systems*, 16(4):387–400, 1991.

[BS04] E. Brosh and Y. Shavitt. Approximation and heuristic algorithms for minimum delay application-layer multicast trees. In *INFOCOM 2004, the 23rd Annual Joint Conf. of the IEEE Computer and Communications Societies*, 2004.

[BW01] S. Banu and J. Widom. Continuous queries over data streams. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data Record*, 2001.

[Car02] D. Carney et al. Monitoring streams: A new class of data management applications. In *Proceedings of. Very Large Databases (VLDB)*, 2002.

[Cas03] M. Castro et al. SplitStream: High-bandwidth multicast in cooperative environments. In *19th ACM Symposium on Operating Systems Principles*, 2003.

[CDKR02] M. Castro, P. Druschel, A. Kermarrec, and A. Rowstron. SCRIBE: A large-scale and decentralized application-level multicast infrastructure. *IEEE Journal On Selected Areas in communications(JSAC)*, 2002.

[CDTW00] J. Chen, D. J. Dewitt, F. Tian, and Y. Wang. NiagaraCQ: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD Intl. Conf. on Management of Data*, 2000.

[Cha03] S. Chandrasekaran et al. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *CIDR*, 2003.

[Che03] M. Cherniack et al. Scalable distributed stream processing. In *Proc. of the First Biennial Conference on Innovative Data Systems Research (CIDR'03)*, 2003.

[CRZ00] Y. Chu, S. G. Rao, and H. Zhang. A case for end system multicast. In *ACM SIGMETRICS intel. conf. on Measurement and modeling of computer systems*, 2000.

[Dia03] Y. Diao et al. Path sharing and predicate evaluation for high-performance xml filtering. *ACM Trans. Database Syst.*, 28(4):467–516, 2003.

[DRF04] Y. Diao, S. Rizvi, and M. J. Franklin. Towards an internet-scale xml dissemination. In *Proc. of the 2004 Intl. Conf. on Very Large Data Bases*, 2004.

[FJL+01] F. Fabret, H. A. Jacobsen, F. Llirbat, J. Pereira, K. A. Ross, and

D. Shasha. Filtering algorithms and implementation for very fast publish/subscribe. In *Proc. of the 2001 ACM SIGMOD Intl. Conf. on Management of Data*, 2001.

[GL03] B. Gedik and L. Liu. PeerCQ: A decentralized and self-configuring peer-to-peer information monitoring system. In *Proceedings of ICDCS*, 2003.

[GMS04] C. Gkantsidis, M. Mihail, and A. Saberi. Random walks in peer-to-peer networks. In *Proceedings of IEEE INFOCOM*, 2004.

[HB95] Y. F. Hu and R. J. Blake. An optimal dynamic load balancing algorithm. Technical report, Daresbury laboratory, 1995.

[HL95] B. Hendrickson and R. Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM J. Sci. Comput.*, 16(2):452–469, 1995.

[KK98a] G. Karypis and V. Kumar. Multilevel k-way partitioning scheme for irregular graphs. *Journal of Parallel and Distributed Computing*, 48(1):96–129, 1998.

[KK98b] G. Karypis and V. Kumar. A parallel algorithm for multilevel graph partitioning and sparse matrix ordering. *Journal of Parallel and Distributed Computing*, 48(1):71–95, 1998.

[KL70] B. W. Kernighan and S. Lin. An efficient heuristic procedure for partitioning graphs. *The Bell System Technical Journal*, pages 291–307, 1970.

[KRAV03] D. Kostic, A. Rodriguez, J. Albrecht, and A. Vahdat. Bullet: High bandwidth data dissemination using an overlay mesh. In *Proc. of ACM SOSP*, 2003.

[LPT99] L. Liu, C. Pu, and W. Tang. Continual queries for internet scale event-driven information delivery. *IEEE Trans. on Knlwledge and Data Engineering*, 1999.

[MSHR02] S. Madden, M. Shah, J.h M. Hellerstein, and V. Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD Intl. Conf. on Management of Data*, 2002.

[NST03] W. S. Ng, Y. Shu, and W. H. Tok. Efficient distributed continuous query processing using peers. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, 2003.

[OJW03] C. Olston, J. Jiang, and J. Widom. Adaptive filters for continuous queries over distributed data streams. In *Proc. of the 2003 ACM SIGMOD Intl. Conf. on Management of Data*, 2003.

[PC05] O. Papaemmanouil and U. Cetintemel. SemCast: Semantic multicast for content-based data dissemination. In *Proceedings of the 21st International Conference on Data Engineering, (ICDE 2005)*, 2005.

[RLW+02] A. Riabov, Z. Liu, J. L. Wolf, P. S. Yu, and L. Zhang. Clustering algorithms for content-based publication-subscription systems. In *Proceedings of the 22 nd International Conference on Distributed Computing Systems (ICDCS'02)*, page 133, 2002.

[SBK02] B. Bhattacharjee S. Banerjee and C. Kommareddy. Scalable application layer multicast. In *INFOCOM 2002, the 21st Annual Joint Conf. of the IEEE Computer and Communications Societies*, 2002.

[SDR03] S. Shah, S. Dharmarajan, and K. Ramamritham. An efficient and resilient approach to filtering and disseminating streaming data. In *Proc. of the 2003 Intl. Conf. on Very Large Data Bases*, 2003.

[SHB04] M. A. Shah, J. M. Hellerstein, and E. A. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proceedings of the 2004 ACM SIGMOD Intl. Conf. on Management of Data*, 2004.

[SKK97] K. Schloegel, G. Karypis, and V. Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *J. Parallel Distrib. Comput.*, 47(2):109–124, 1997.

[SKK03] K. Schloegel, G. Karypis, and V. Kumar. Graph partitioning for high-performance scientific simulations. *Sourcebook of parallel computing*, pages 491–541, 2003.

[SMW05] U. Srivastava, K. Munagala, and J. Widom. Operator placement for in-network stream query processing. In *Proc. of the 2005 ACM SIGMOD Intl. Conf. on Management of Data*, 2005.

[SRS02] S. Shah, K. Ramamritham, and P. Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *Proc. of the 2002 Intl. Conf. on Very Large Data Bases*, 2002.

[Teu78] J. Teuhola. A compression method for clustered bit-vectors. *Information Processing Letters*, pages 308–311, 1978.

[TGNO92] D. Terry, D. Goldberg, D. Nichols, and B. Oki. Continuous queries over append-only databases. In *Proceedings of the 1992 ACM SIGMOD Intl. Conf. on Management of data*, 1992.

[The03] The STREAM Group. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin*, 26(1), September 2003.

[WCE97] 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.

[XZH05] Y. Xing, S. B. Zdonik, and J. Hwang. Dynamic load distribution in the borealis stream processor. In *Proceedings of 21st International Conference on Data Engineering, (ICDE 2005)*, pages 791–802, 2005.

[YGM99] T. E. Yan and H. Garcia-Molina. The SIFT information dissemination system. In *TODS*, 1999.

[ZZJ+01] S. Q. Zhuang, B. Y. Zhao, A. D. Joseph, R. H. Katz, and J. D. Kubiatowic. Bayeux: An architecture for scalable and fault-tolerant wide-area data dissemination. In *Proc. of the 11th International Workshop on Network and Operating System Support for Digital Audio and Video (NOSSDAV)*, 2001.