SCALABLE AND ADAPTABLE DISTRIBUTED STREAM PROCESSING

ZHOU YONGLUAN
(B.Eng, Zhejiang University)

A THESIS SUBMITTED

FOR THE DEGREE OF DOCTOR OF PHILOSOPY

DEPARTMENT OF COMPUTER SCIENCE

NATIONAL UNIVERSITY OF SINGAPORE

2006

Acknowledgement

This thesis is the result of several years of intensive academic learning. I would like to thank my friends, colleagues and family alike, without whose support, I would not be able to get through such a long and tough journey. To begin, I thank my supervisor, Prof. Tan Kian-Lee, for his many hours of discussions, detailed and constructive advice and guidance, as well as his confidence in my own ability. His enthusiasm in my research work gives me the impetus to complete it. I would also like to thank Dr. Chan Chee Yong and Dr. Panos Kalnis for their critical and helpful comments on my work. Thanks are owed to Dr. Anthony Tung, who gave me much advice and guidance in the beginning of this journey. I am grateful to Yu Feng, without whose help, I would not complete many experiments in this thesis. My gratitude also goes to my colleagues in NUS database group for their great friendship: Lu Hua, Lu Jiaheng, Ji Liping, Sun Chong, Wu Ji, Xiang shili, Xu Linhao, Yan Ying, Yu Tian, Zhang Zhenjie and many many more. The chief supporter is of course, my beloved wife Jianer, whose unfailing love and unrelenting support accompanies me throughout the PhD journey. I thank God for the extraordinary chance of having her as my wife, friend and companion. Last but not the least, I am forever indebted to my parents and parents in law for their love and dedication.

Contents

Summary

Data stream processing has a wide applicability, ranging from computer network management to financial monitoring to environment monitoring through sensor network. Consequently, they have received much research attention in recent years. Early research focused on developing centralized stream processing engines. These systems are limited in their scalability to the number of users or the volumes of streams. This thesis examines the design of a large scale distributed stream processing system, COSMOS (Cooperative and Self-tuning Management Of Streaming data), with the emphasis on its scalability and adaptability issues.

COSMOS is composed of a number of widely distributed stream processing Service Providers (SP). It adopts a two-layer architecture, namely the inter-provider layer and the intra-provider layer. The inter-provider layer manages the cooperation among the widely distributed SPs while the intra-provider layer harnesses a cluster of locally distributed processors inside an SP. By identifying the challenges in the two layers, we propose different architectures and techniques for them respectively.

At the inter-provider layer, two overlays, the query overlay and the data overlay, are designed to handle the query stream and data stream respectively. The query overlay is responsible to distribute the queries to the SPs for processing to achieve

both load balancing and minimum communication cost. We model the query distribution problem as a graph partition problem and proposed a hierarchical query distribution approach, which is scalable to a large number of queries and adaptable to the changes of data and system characteristics. Furthermore a query management scheme is proposed to leverage the power of the data overlay to minimize the communication cost.

The data overlay is responsible to disseminate the source data from the sources to the widely distributed SPs and the result data from the SPs to the users. To achieve high dissemination efficiency, the SPs are organized into multiple overlay dissemination trees. We propose an adaptive and cost-based overlay tree construction scheme, which can self-tune the tree structure at runtime in according to the changes of system parameters, such as processing delays, transmission delays, and data rates etc.

At the intra-provider layer, operators of the queries allocated to an SP are distributed to the locally distributed processors for processing. In terms of query optimization, there are two challenges within this layer: operator ordering and operator placement. We propose an adaptive scheme to optimize the operator ordering in the midst of the processing of a query. It employs multiple distributed Eddies [8] at different processors to adapt the order of operators distributed to multiple processors. It can quickly detect the change of operator selectivities, transmission speed and processors' workload at runtime and continuously re-optimize the operator orders accordingly.

We also propose a dynamic operator placement scheme which dynamically allocate the query operators to the processors. It is aimed to minimize the delay of result tuples. By analyzing the problem using a cost model, we identify a few heuristics to achieve this objective. A scalable scheme is proposed to implement

these heuristics dynamically.

Extensive experiment results show that the proposed techniques are effective and efficient.

List of Tables

List of Figures

Chapter 1

Introduction

In many emerging applications, such as stock tickers, sports tickers, network management, sensor network, financial monitoring etc., data occurs naturally in the form of active continuous data streams. These applications typically require the processing of complex queries over large volumes of data in a responsive manner. They have fueled much research interest in designing stream processing engines [22, 1, 31, 81]. Such engines support complex continuous queries over push-based data streams, specified by SQL-like languages [31, 81] or operator networks built through a GUI [22, 1].

Early research efforts have focused on centralized processing engines [22, 31, 81]. These systems are not scalable to large volumes of streams and queries. Furthermore, the sources of data streams are naturally distributed. Using a centralized engine would require the transmission of all the data to a central node. This may incur a large amount of transmission over the network and aggressively consume the precious network resources. To solve the above problems, a distributed stream processing engine is inevitable.

In this thesis, we look at the design of a large scale distributed stream processing engine, namely COSMOS (COoperative and Self-tuning Management of Streaming data), based on our vision of future applications. The major challenge

in the design of such a complex system is the unmanageability of its performance tuning. To tackle this problem, we adopt the notion of *autonomic computing* and design self-tuning techniques to manage the system. The system can adaptively refine its configuration without the intervention of human activities.

1.1 Problem Statement and Motivation

In applications which have a potentially large number of clients, such as financial market monitoring, there are some emerging service providers (SP) that provide stream processing services for a large number of clients. One example of such kind of SP is TRADERBOT (`http://www.traderbot.com`). Instead of only providing stock quotes, such an SP should be able to evaluate user specified complex queries over a large number of fast updating data objects (the temporal values and statistics of individual stocks, indices ect.) and deliver the results in a real time manner back to the clients. These source data are streamed into the SP in real time from a large number of sources (such as exchanges) which are widely distributed over the whole world. The following are some example queries:

1. Send me the quotes of Google when its price drops below $300. This is a simple continuous selection query over a data object: Google stock. We can write this query into a SQL-like language as follows:

```
SELECT *
FROM   Quote
WHERE  name = GOOGLE AND price < 300
```

2. Send me the quotes of both Google and Microsoft if the price of Google drops below $300 and the price of Microsoft also drops below $20. This is a

continuous selection and join query over two data objects. This query can be written with SQL:1999-like WITH syntax as follows.

```
WITH
  Google as
  {
    SELECT *
    FROM   Quote
    WHERE  name = GOOGLE
  }
  Microsoft as
  {
    SELECT *
    FROM   Quote
    WHERE  name = Microsoft
  }
(SELECT *
 FROM   Google, Microsoft
 WHERE  Google.price < 300 AND Microsoft.price < 20 AND
        Google.time = Microsoft.time);
```

3. Continuously inform me all NASDAQ stocks between $20 and $200 that have moved down more than 2% in the last 20 minutes. This query includes selections, join and window-based aggregations over a set of objects.

```
WITH
  Nasdaq as
```

```
 {
   SELECT *
   FROM   Quote
   WHERE  market = "NasdaqGS"
 }
(SELECT   *
 FROM     Nasdaq [Range 20 minute] as N1, Nasdaq [Now] as N2
 WHERE    N1.name = N2.name
 GROUP BY N1.name, N2.price
 HAVING   N2.price =  min(N1.price) AND
          max(N1.price)=1.02* min(N1.price));
```

In this query, the statements "Nasdaq [Range 20 minutes]" defines a window over the stream "Nasdaq" which contains the tuples arrived in the last 20 minutes, while "Nasdaq [Now]" defines a window that contains only the last arrived tuple from "Nasdaq". More details and semantics of window will be introduced in Chapter 2.

To support the evaluation of such complex queries, stream processing engines can be employed by an SP. In order to scale up the volumes of streams and queries that can be processed, an SP could employ an architecture of a cluster of processors interconnected by a fast local network. Hence an efficient architecture that can harness the power of such a processor cluster is inevitable.

Furthermore, we envisage that more and more such SPs would emerge in different cities, states, countries etc. Our system, COSMOS, is targeted at a more ambitious service which integrates the processing power and capabilities of the

various SPs to provide a central access portal to all the clients. A client can submit his queries to the system through any SP, which serves as his proxy and is responsible to deliver the result stream back to him. The participating SPs are expected to cooperate based on business agreements and they are encouraged to process queries assigned to them by these agreements. For example, an SP can be paid based on the length of time when it executes the queries. We also assume there is a known global schema of the data. Each participating SP only needs to install a wrapper which is responsible to cooperate with other SPs. Due to different business considerations, these SPs may employ different processing engines. Hence different SPs may have different data models, processing models as well as user interfaces. The SPs are not expected to surrender their administrations. Furthermore, they are typically interconnected by a widely distributed network, which brings different requirements to the architectural design.

Based on the above observations, we can see that COSMOS is naturally partitioned into two layers: the inter-provider layer and the intra-provider layer. This structure is illustrated in Figure 1.1. At the inter-provider layer, the widely distributed SPs cooperatively distribute the user queries to the SPs for processing and disseminate the data to feed the queries. At the intra-provider layer, the closely coupled processors of an SP efficiently process the queries allocated to that SP. In the following subsections, we shall discuss the challenges of this system. We first discus the general challenges of the whole system in Section 1.1.1 and then concentrate on those specific to the two layers in Sections 1.1.2 and 1.1.3 respectively.

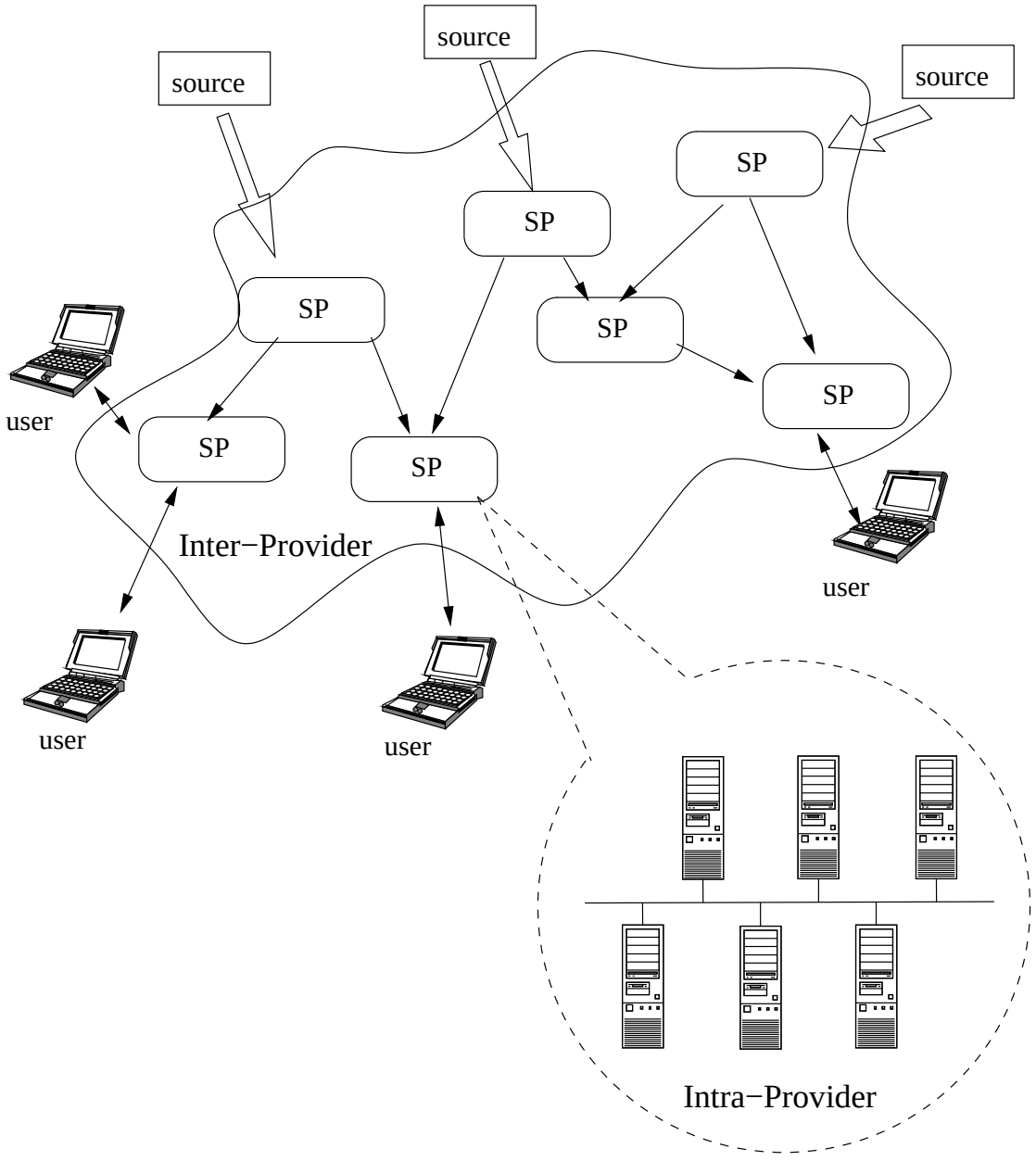


Figure 1.1: System Overview

1.1.1 Scalability and Adaptability

To design such a distributed processing system, the first problem we have to address is its scalability. It should be scalable to the number of clients, the number of service providers and processors as well as the volumes of data streams. To achieve this goal, minimizing the query execution cost, balancing the load distribution and minimizing communication cost is critical. Furthermore, the degree of coupling of the distributed nodes in the system should be carefully considered.

In addition, the large scale and the unpredictability of such a system bring a lot of difficulties in the tuning of system performance. First of all, it is unrealistic to collect accurate statistics of system parameters from a large number of distributed nodes. These parameters include the properties of the data streams (such as arrival rates, value distributions, etc.), the processing servers' load and the network transfer bandwidth and delay etc. Second, these system parameters are hard to predict and may evolve over time. Consequently, operator selectivities, operator cost, operator processing delay as well as data transfer rates will fluctuate at run time. Due to these unpredictable factors, the initial query plans may result in unsatisfactory system performance. The problem is exacerbated by a large number of continuous queries that run long enough to experience the changes of the system parameters. As such, any suboptimal performance will persist for a long time. To be robust to such inaccurate and fluctuant parameters, the system should have the ability to continuously self-tune/adapt its behavior without the intervention of human activities.

1.1.2 Challenges at the Inter-Provider Layer

There are a few challenges for the inter-provider layer. First, the SPs are autonomous. This brings two problems to the system design. (1) The providers

may join or leave at any time which is out of the system's control even without failure. The problem of how to restore from any of these events should be considered. (2) The hardware and software configuration in the whole system is heterogeneous. For instance, different stream processing engines may be installed in different providers due to different business decisions. These providers may consequently have different data models and processing models.

Second, both the number of SPs and users could be very high. Hence, algorithms requiring to keep track of the status of the whole network at a single node is non-scalable and even impossible. For example, in existing distributed stream processing systems, a stream source has to keep track of all the queries requesting its data. A more "intelligent" communication mechanism is required to decouple the sources and users.

Third, in a WAN environment, the communication cost could be very high as it may involve inter-country and even intercontinental communication. Furthermore, streams are typically of a very high rate and are transferred persistently. Hence, achieving communication efficiency should be an important objective in the system design. More specifically, we should exploit the opportunities of the sharing of the communication among different queries. For example, two users in Singapore and Malaysia, respectively, may be interested in the stock market of the New York Exchange and submit their queries to the two servers in their own countries respectively. One approach is to plan the two queries separately. The requested source streams as well as the intermediate result streams of these two queries would be transferred separately even though they may share a large amount of common contents. This incurs unnecessary overheads because these streams may have similar transfer path due to the proximity of their destinations. With a large number of user queries, such overhead would be overwhelming. While performing

multiple query optimization may alleviate this problem, it impairs the system's scalability. For example, Borealis [1] proposed to generate a giant operator graph for all the queries submitted to the system. However, no scalable algorithm was proposed to achieve this goal so far.

To address the above problems, the providers should cooperate in an "intelligent" and loosely coupling way. In this thesis, we focus on the cooperations in the two major services provided by the system: the stream dissemination service and the query processing service. We build two service overlays to address them respectively.

Data overlay. This overlay is designed to support the stream dissemination service, which is responsible to deliver the source streams from the stream sources to the SPs and the result streams from the SPs to the end users. As mentioned earlier, the mechanism to provide this service should be able to exploit the sharing of the common communication of different queries and decouple the data sources and the destinations. This requirement brings our attention to the multicast paradigm. With a multicast network, we can associate each data stream with a (virtual) multicast address. When a server receives a query, it joins the multicast groups of all the relevant streams of the query. The sources do not need to keep track of the receivers. Instead the sources simply pass the data to the multicast network together with the corresponding multicast address and then the multicast network routes it to all the multicast group members. The common communication among different receivers are naturally shared.

However a multicast network is limited in its expressiveness of data interest and hence is not adequate to minimize the communication cost in our context. Fortunately, a new multicast-like networking method: content-based network (CBN) is emerging in recent years [27]. In a CBN, each datagram consists of several

attribute-value pairs. A node in the network can express its data interest as a few selection predicates on the attributes of the datagram. The data interest can be viewed as the virtual network address. Again the sources and the destinations are not known to each other. Datagrams are routed by the network based on the data interest of the receivers. We can see that a CBN retains the merits of a multicast network (i.e., communication for common items is shared, and the data sources and receivers is loosely coupled) and achieves better communication efficiency by providing a more powerful interface to express data interest.

While multicast and content-based network can be implemented at different network levels, recent implementation effort has focused at the application level due to the unsuccessful deployment of IP-level multicast network. Hence we will employ application level content-based network in our system.

Query overlay. This overlay is to provide the query processing service. To maximize the system resource utilization and the system throughput, the SPs should share the workload with each other. Again the cooperation of SPs in this service should also be loosely coupled. Thus, many tightly-coupled cooperation techniques developed by existing systems cannot be directly tapped upon. For instance, dynamically distributing the operators of a query to multiple providers violates the loosely-coupling property. Moreover, this may also not be feasible, e.g., moving a window join operator from the STREAM system to a TelegrahCQ system is hard to implement, because it relies on a special data structure "synopsis" implemented in STREAM which is not only manipulated by the join operator itself but also other operators before or after the join operator. Furthermore, even though the providers use the same engine, one may upgrade its engine without informing the others. This would also bring problems unless forward and backward compatibility is implemented.

Therefore, SPs are designed to share load in the unit of queries. In other words, we adopt a query level load distribution scheme (instead of an operator level scheme). The load distribution algorithm is expected to run much faster than operator level algorithms as the number of queries is much smaller than that of the operators. This also facilitates scaling to fast query arrivals and departures.

Under the above architecture, three challenges are addressed in this thesis. (1) How can the query overlay leverage the power of the data overlay to enhance the communication efficiency? (2) How to allocate the queries to the SPs to balance the workload and minimize the communication cost? (3) How to construct the dissemination trees to efficiently disseminate the streaming data to the widely distributed SPs? Again, all these decisions should be made in an adaptive and scalable way.

1.1.3 Challenges at the Intra-Provider Layer

For a SP, the first problem is how to receive the data feed by the upstream SP and forward them to the downstream SP. Relying on a single processor to receive all the streams is not scalable. Furthermore, the transfer cost at the inter-provider layer is much higher than the one at the intra-provider layer. Hence, within each SP, we assign a processor as the delegation of each data stream that is sent to the SP. The delegation processor is responsible to route the streams to other processors in the same SP as well as to transfer the streams to the child SPs.

The processors within each SP are assumed to be under a central administration and interconnected by a fast local network. This eases the employment of tightly-coupled techniques to enhance the system efficiency. Particularly, we can distribute the operators of each query to multiple processors to achieve desirable system performance. There are two important questions to be answered: 1) How

to decide the ordering of operators which has significant effect on the running cost of a query? 2) How to distribute the operators to the processors to balance system workload and minimize the communication cost? All these decisions should be made and refined at runtime. The adaptation of the query plan should be done efficiently. Furthermore, to enhance the scalability, the decisions should only be made by distributed nodes based on local information instead of the global system state.

1.2 Contributions

This thesis makes the following contributions:

- We propose a new loosely-coupled and large-scale architecture for the inter-provider layer. To handle the streaming data and streaming queries respectively, the system is composed by two overlays. The data overlay resembles the CBN architecture and is responsible for disseminating source data streams among the distributed SPs as well as the result streams to the users. The query overlay dynamically distributes the streaming queries to a number of distributed nodes for processing.

- In Chapter 3, we propose optimization techniques at the query overlay. More specifically, we present how to utilize the power of the data overlay to enhance the communication efficiency. We also present how to distribute the query workload to achieve both load balance and minimum communication cost. We model the query distribution problem as a graph partitioning problem and develop techniques to adaptively and rapidly (re)distribute the streaming queries. Our extensive performance study shows the efficiency of our techniques.

- In Chapter 4, we focus on some specific applications, such as disseminating

stock quotes, where users can tolerate a certain degree of inaccuracy. For these applications, we can further optimize the data overlay network by utilizing the knowledge of such user tolerances. An adaptive mechanism is designed to construct the overlay dissemination trees to efficiently transfer the streams from the sources to the widely distributed SPs. We focus on constructing dissemination trees to minimize the average *loss of fidelity* of the system. Based on a novel and thorough cost model, we propose both adaptive and static overlay tree construction mechanisms. The extensive performance study shows that the adaptive mechanisms are effective in a dynamic context and the proposed static tree construction algorithms perform close to optimal in a static environment.

- In Chapter 5, we design a new highly adaptive distributed query processing architecture, which can quickly detect fluctuations in selectivities of operations, as well as transmission speeds and workloads of servers, and accordingly change the operation order of a distributed query plan during execution. We have implemented a prototype based on the Telegraph system [64]. Our experimental study shows that our mechanism can adapt itself to the changes in the environment and hence approach to an optimal plan during execution.

- In Chapter 6, a dynamic operator placement scheme is proposed to adaptively distribute the operators of multiple queries within the processor cluster of each SP. We formalize and analyze the operator placement problem in the context of a locally distributed continuous query system. We also propose a solution, that is asynchronous and local, to dynamically manage the load across the system nodes. Essentially, during runtime, we migrate query operators/fragments from overloaded nodes to lightly loaded ones to achieve better performance. Heuristics are also proposed to maintain good data flow locality. Results of a performance study shows the effectiveness of our technique.

1.3 Layout

The rest of this thesis is organized as follows:

- Chapter 2 provides a general introduction and related work about this field.
- Chapter 3 presents the optimization techniques for the query overlay.
- Chapter 4 introduces the adaptive dissemination infrastructure construction for the data overlay.
- Chapter 5 presents an adaptive operator ordering scheme at the intra-provider layer.
- Chapter 6 proposes an efficient dynamic operator placement scheme for the intra-provider architecture.
- We conclude our work in Chapter 7 with a summary of our contributions. We also discuss some limitations and provide directions for future work.

The whole architecture of the system has been presented in [88]. The results of Chapter 3 appear in [95, 93, 94] which have been submitted for publication. Partial contents of Chapter 4 and Chapter 6 are published in [92] and [89, 91] respectively. Finally, The contents in Chapter 5 has been published in [87, 90] .

Chapter 2

Background

Our work is related to several areas: stream query processing systems, stream delivery systems, distributed and parallel database systems, as well as general distributed and parallel systems. In this chapter, we review the existing research results that are generally related to our system as a whole. For those related work that are only relevant to specific contributions of our work, we will present them in the respective chapters.

2.1 Stream Processing Systems

A data stream is an unbounded bag of tuples that conform to a fixed schema. Typically, the schema includes a special attribute, *timestamp*, that indicates a tuple's arrival time in a logical application time domain. Applications involving data streams such as network monitoring, financial analysis and sensor network require continuously evaluating user queries over the continuous data streams.

2.1.1 Centralized Techniques

Early research focused on developing centralized stream processing systems. STREAM [81], Aurora [22] and TelegraphCQ [31] are some representative research systems in this

area.

In STREAM and TelegraphCQ, user queries are specified in query languages derived mainly by extending SQL with window semantics. Aurora takes a different approach by providing a graphical interface that allows users to compose queries by linking boxes and arrows, where boxes are the various query operators and the arrows indicate the flow of the streams. The dataflow graph is then translated into an internal XML-based query definition language.

No matter which approach is adopted, the streaming query languages in these systems are all distinguished from traditional queries by introducing the window semantics. Since data streams are unbounded, a window predicate should be specified in a query on each involved stream to avoid unbounded resource consumption in the processing of the query. There are two types of window that are widely adopted, namely tuple-based window and time-based window. A tuple-based window on a stream S with an integer window size N produces a temporal relation $R(\tau)$, which contains the N tuples from S with the largest timestamps larger or equal to τ. On the other hand, a time-based window on S with a window size T produces a temporal relation $R(\tau)$, which contains all the tuples from S with timestamps within the range $[\tau - T, \tau]$. For example, in CQL [5], a stream identifier, say S, in the FROM clause will be associated with a window predicate in the form as "S [Range 10 minute]" , which specifies a time-based window on stream S with the window size of 10 minutes. The default window size in CQL is infinite (specified as "[Range Unbounded]"), while the minimum window size is 0 (specified as "[Now]").

An important problem to be solved in such systems is how to optimize the query processing. Existing efforts have focused on the following directions.

Adaptive Query Optimization

One difficulty in stream query optimization is that the data characteristics are hard to collect and is subject to change at runtime. As queries run for a very long time, they are expected to undergo such changes. Hence traditional query optimization techniques based on static statistics are not optimal.

CACQ [58] uses the techniques of Eddies [8] and SteMs [66] to address the problem of query plan adaptation. Instead of constructing a query plan as in traditional query optimization techniques, TelegraphCQ adaptively decides the operator ordering for each individual tuple. Our operator ordering mechanism at the intra-provider layer presented in Chapter 5 extends the centralized Eddies [8] scheme to a distributed environment. [17] improves the tuple routing strategies of Eddies by generating different routing orders for tuples with different values.

The STREAM group proposed the use of adaptive filter reordering [10] as well as adaptive caching of intermediate results [11] are proposed to minimize the query evaluation cost in both CPU and memory consumption.

On the other hand, [97] studied the mechanisms to minimize the cost and delay of adapting the plan at the midst of processing.

Resource Sharing

In a stream processing system, there would be a number of continuous queries being executed simultaneously. As these queries would have common operations, exploiting the sharing of resource consumption among them would bring drastic benefit.

CACQ [58] proposed the use of "macro" operators, such as group filters and SteMs, to exploit the share of computation and memory consumption among selection and join operations. Each "macro" operator will be utilized by multiple

queries to share computation. Furthermore, in each "macro" operator, an index would be built for the related queries, which is used to efficiently find out the set of queries that are satisfied by an input tuple.

STREAM [6], on the other hand, studies the opportunities of resource sharing among sliding-window aggregate operations. Techniques are proposed for different classes of aggregation functions (algebraic, distributive and holistic), different window types (time-based, tuple-based, suffix and historical), and different input models (single stream and multiple substreams).

More recently, authors in [54] also proposed techniques to share resources among aggregate queries. Unlike prior work, the proposed approach does not require static analysis of fixed query workloads and hence facilitates frequent query arrival and departure.

2.1.2 Techniques for Locally Distributed Systems

One direction of recent efforts on enhancing the scalability of stream processing systems is deploying it onto locally distributed systems.

The research group of TelegraphCQ proposed Flux [74, 73], which employs a cluster of tightly-coupled processors to enhance the scalability of TelegraphCQ. In Flux, a dynamic load balancing strategy for horizontal (or intra-operator) parallel processing of operators are employed. In their system, the network connections are assumed to be very fast and hence the communication cost is ignored. A centralized synchronous controller is used to collect workload information and to make load balancing decisions.

The Borealis system, the descendant of the Aurora project, employs a dynamic load balancing [86] algorithm which considers the load correlation of operators. This scheme, similar to Flux, also assumes the communication cost could

be ignored and employs a centralized controller to make load balancing decisions. Therefore, in this scheme, it is possible that the intermediate result streams of a query would be transferred many times over the network.

We, on the other hand, argue that the communication cost cannot be ignored even in a fast local network. First, the volume of the data streams could be very high. Transferring them many times over the network as in the Borealis system would incur network congestions. Second, even though the network is not congested, the delay incurred by sending the tuple over the network many times cannot be ignored. Furthermore, the above researches assume the operator ordering is pre-determined and only focus on the load balancing problem.

An independent piece of work [82] also performed studies on distributing the eddies [8] mechanism. The authors focused on the study of several practical tuple routing policies. They assume there exists an efficient distributed processing architecture based on eddies. Our work, on the other hand, focuses on a complementary problem: developing a new and practical distributed processing architecture based on eddies. The tuple routing strategies proposed in [82] can be incorporated into our architecture. Furthermore, the simple distributed eddy mechanism proposed in [82] can be viewed as a special case of our system.

2.1.3 Techniques for Widely Distributed Systems

The Medusa [32] system also adopts an architecture to integrate multiple administratively independent participant. In their load distribution algorithm, it is assumed that the participants employ the same type of processing engine so that operators of a query can be distributed to multiple participants for processing. Furthermore, their architecture does not address the problem of transferring the data steams to large number of nodes and rely solely on the sources to disseminate

the streams.

In project of SAND (Scalable Adaptive Network Databases) [2], the authors studied the allocation of operators of a query in a widely distributed network for a given operator ordering. The proposed algorithm allocates the operators along the path from the input to the output to achieve minimum communication cost incurred by the query. inTransit [55] has a similar problem setting as SAND. It first partitions the network into hierarchical clusters and then exhaustively search the optimal allocation scheme of operators at each level. SBON [63] is yet another effort in this direction. The authors proposed the use of spring relaxation to allocate the query operators.

In PMJoin [96], on the other hand, the authors studied not only operator placement but also the operator ordering problem to minimize the communication cost. PMJoin identifies that the optimal operator ordering and placement for tuples with different values would be different. Hence it partitions each stream into multiple substreams and perform optimizations on them respectively.

We can see that all the above approaches adopt a tightly-coupled architecture. Operators of a query are allocated to multiple nodes and hence all the nodes are required to employ the same processing model and data model. Furthermore, synchronizations in runtime operations might also be needed. Therefore, they cannot fit into our loosely-coupled architecture. Moreover, the above approaches ignore the problem of load balancing and only focus on minimizing communication cost. We believe that a more optimal approach should take both into consideration.

PeerCQ [40] also took a loosely-coupled architecture, a DHT-like P2P system, to process continuous queries over data object updates. Queries with similar trigger conditions tend to be hashed to a similar identifier and consequently tend to be allocated to the same peer. This saves communication cost as the updates

interested to the multiple queries running at the same node only need to be transferred once. The authors also proposed some heuristics to take into account of load balancing, distance between the peers and the data sources, the availability of the cache. However, besides this system was not explicitly proposed for stream processing, it fails to exploit the sharing of common communication among queries running at different nodes which is one of the main optimization objectives of COSMOS' inter-provider layer.

2.2 Stream Delivery Systems

The popularity of multimedia streams and event steams attracted much attention of the networking community. Consequently, a lot of efficient algorithms are proposed to address the problem of delivering fast streams to a large number of users.

2.2.1 Stream Multicast Systems

Stream multicast has a number of applications such as video conference, video on demand etc., where data are sent from the sources to a group of interested receivers.

Early efforts have been focused on deploying multicast at the IP layer [35]. IP multicast can achieve high communication efficiency as each data packet would be transferred at most once over a physical link and copies of the data packet would only be generated when the links to the receivers splits. Unfortunately, IP multicast requires the introduction of extra complexities (such as group management functions) into routers and the changes at the infrastructure level. Hence, its deployment is not very successful.

Recently, a new application level multicast paradigm is proposed [33, 14]. This kind of paradigm builds a multicast overlay on top of a unicast IP network. While it can ensure that a data packet is only transferred once over an overlay link, it is still possible that the data packet is transferred multiple times over a physical link. However, by placing the complexities onto the application level, it trades communication efficiency for the ease of deployment. Furthermore, it also facilitates the implementation of higher level features such as error, flow and congestion control. Following this direction, a lot of recent efforts have been focused on optimizing the overlay multicast trees [15, 20] to enhance the efficiency of this paradigm.

While stream multicast systems are efficient in sending data to a large number of receivers, they send each data packet to all the receivers and hence they do not readily fit into our system.

2.2.2 Distributed Publish/Subscribe Systems

Applications, such as stock/sports tickers, news feed etc., prompted a large number of efforts on developing publish/subscribe systems [39]. Such systems are typically implemented by installing the middlewares in a set of distributed servers. Users issue subscriptions that specify their data interest to these servers, while the data sources also publishes their data to these servers. The servers cooperatively route the data to the interested users.

In topic-based publish/subscribe systems [3], data interests are classified into multiple predefined topics. Users' subscriptions only need to specify the topics that they are interested. We can see that this type of system can be efficiently implemented directly using multicast systems mentioned above.

A more complicated type of publish/subscribe systems are called content-based publish/subscribe systems [67]. Users' subscriptions are specified on the

contents of the data instead of predefined topics. For example, in Gryphon [13] and Siena [24, 25], a data item is composed by multiple attributes. Subscriptions are expressed as filters on the values of these attributes. It is obvious that content-based publish/subscribe systems are more flexible in expressing data interest and hence can deliver more relevant data to the users and avoid unnecessary communication.

In [27], the authors proposed extending the mechanism of content-based publish/subscribe systems to construct a a new networking infrastructure, content-based network, to support content-based communication. Content-based networking aims to implement the communication style of a content-based publish/subscribe system on a true distributed network environment by leveraging established networking techniques. Recently, the authors proposed the routing and forwarding schemes of a content-based network in [26, 28] respectively. As we can see, such a networking approach fits well with the requirements of our system in data stream delivery. It not only can inherently exploit the sharing of communication among different receivers, but also can achieve high communication efficiency by sending only the interested data to the receivers.

Moreover, there is a recent trend to enhance publish/subscribe systems for more complicated event notification services. Some efforts focus on developing more expressive subscription languages, e.g. [36]. However, these languages are far less expressive than SQL languages. A more recent effort [30] studied how to leverage database systems to process stateful subscriptions on the updates of database tables. However, the concept of such table updates is more limited than the data stream concept addressed in our system. Furthermore, it only focused on several ad hoc subscription types without performing a systematic study.

2.3 Parallel and Distributed Query Processing

In this section, we review some related work on traditional parallel and distributed database query processing.

2.3.1 Parallel Query Processing

A survey on traditional centralized database query processing as well as parallel query processing can be found in [42]. Many efforts have focused on how to deploy an operator on to a parallel system, such as join operators [70], sort operators [61, 7] etc. The operator Exchange [41] was proposed to encapsulate the parallelization details of these operators when pipelined query plan is composed by multiple parallelized operators.

Work in [65, 59, 19] proposed some "semi-dynamic" load balancing strategies to determine the degree of parallelism and placement of operators in a parallel database system. The decisions are made just before the execution and hence cannot adapt to the run time changes in a data stream system. In [57], the authors explored some dynamic load balancing strategies on passive dataset.

2.3.2 Distributed Query Processing

Distributed query processing over traditional passive database also has been extensively explored. Reference [52] is a recent survey that provides a thorough introduction of basic and the state of the art techniques in this area.

Distributed INGRES [38] extended the recursive optimization algorithm of the centralized INGRES [85] by adding a strategy for selecting the fragments to transfer and the sites for processing. System R* [71, 56] also extended the query optimization of System R [72], which is based on dynamic programming,

by considering two more decisions: the sites to evaluate the operation and the method of transferring data between sites.

SDD-1 query optimization algorithm [16] addressed the problem of transferring large size relations and made extensive use of semi-joins. The optimization objective function is expressed in terms of total communication time (local time and response time are not considered). The algorithm first selects an initial feasible solution that is iteratively refined in a hill-climbing way.

Mariposa [79] is a distributed DBMS developed in UC Berkeley. This system uses a three-phase processing scheme. The first phase, compilation, is to construct a locally optimal plan assuming all data are local to the home site. This step fixes items such as join order and the application of join and restriction clauses. The second phase, parallelization, determines the degree of intra-operator parallelism required for various subtrees of the plan tree and inserts collector nodes throughout the plan. The last phase distributes the tree nodes among the various Mariposa sites and executes it. By introducing an economic bidding process, the site selection decision can adapt to the changing costs of the operations from query to query. However, this system can only provide inter-query adaptivity and only affects the choice of processing sites. The operation order is fixed before query processing. In addition, in terms of degree of parallelism, our scheme is more flexible due to the use of symmetric join algorithms, while Mariposa fixes it to be the number of fragments of the outer join class of the leftmost bottom join node.

Furthermore, dynamic load balancing or load sharing are also extensively explored in other parallel or distributed systems [84, 77, 37]. Most of them do not consider the situation that a task is partitioned into several pieces and distributed to multiple nodes. Hence the data communication induced by the task partition is overlooked in the strategies of these systems. Task partitioning has been addressed

in the work of parallel scientific computations [45]. These focus particularly on the dynamic re-partitioning of computational meshes. Moreover, all of the above dynamic strategies only focused on non-continuous tasks and hence their aim is to minimize a task's response time, i.e. the time that the whole task could be finished. Our objective is much different from this.

Chapter 3

Query Distribution

In this and the next chapter, we shall study the query overlay and the data overlay in the inter-provider layer of COSMOS. In this chapter, we first present the system model of the inter-provider layer which consists of two overlays: the data overlay and the query overlay. We assume there is a data overlay that can disseminate the data streams efficiently based on the data interest profiles of the distributed nodes and focus on the design of the query overlay in this chapter. The query overlay is responsible for distributing the user queries for processing. We target to solve two problems in this chapter.

- How to leverage the power of the data overlay to enhance the communication efficiency. Existing systems only adopt a simple unicast communication paradigm and hence have not studied this problem.

- How to distribute the queries to the SPs for processing. Unlike existing approaches which either focus on load balancing [74, 86] or minimizing communication cost [2, 63], we propose an algorithm to distribute queries across the SPs to balance the processing load as well as to minimize the communication cost in transferring the streaming data.

The rest of the chapter is organized as follows. Section 3.1 presents more de-

tails of the system model, the assumptions and the challenges of the inter-provider layer. Section 3.2 presents the query management techniques to enhance the communication efficiency of the data overlay. The query distribution techniques are presented in Section 3.3. We report the results of a performance study in Section 3.4.

3.1 System Model of the Inter-Provider Layer

The system is backed by a number of distributed service providers interconnected with a widely distributed overlay network (see Figure 1.1 for the overview of the whole system). These SPs are autonomous and may join or leave the system anytime. A number of data sources continuously publish their data to the network through the SPs. While each SP is composed by a cluster of SPs, at the inter-provider layer, we treat each SP as a single node for brevity.

There are two major challenges here: data stream delivery and query processing. Two overlay, the data overlay and query overlay, are constructed to address these two problems respectively. Figure 3.1 shows the architecture of a node. There are two levels of modules in the architecture for the two overlay respectively. Note that we do not require that every node is equipped with modules for both overlays. Some nodes could have only the data overlay modules. They only participate in the data overlay and plays the role similar to that of the brokers in a distributed publish/subscribe system. Contrary to existing systems, each node can be under different administrations and run by different entities. Hence, COSMOS allows different stream processing engines (SPE) or different versions of the same SPE to be installed in different SPs. Existing SPEs such as TelegraphCQ [31], STREAM [81] and Aurora [22] can be employed in COSMOS. For each type of

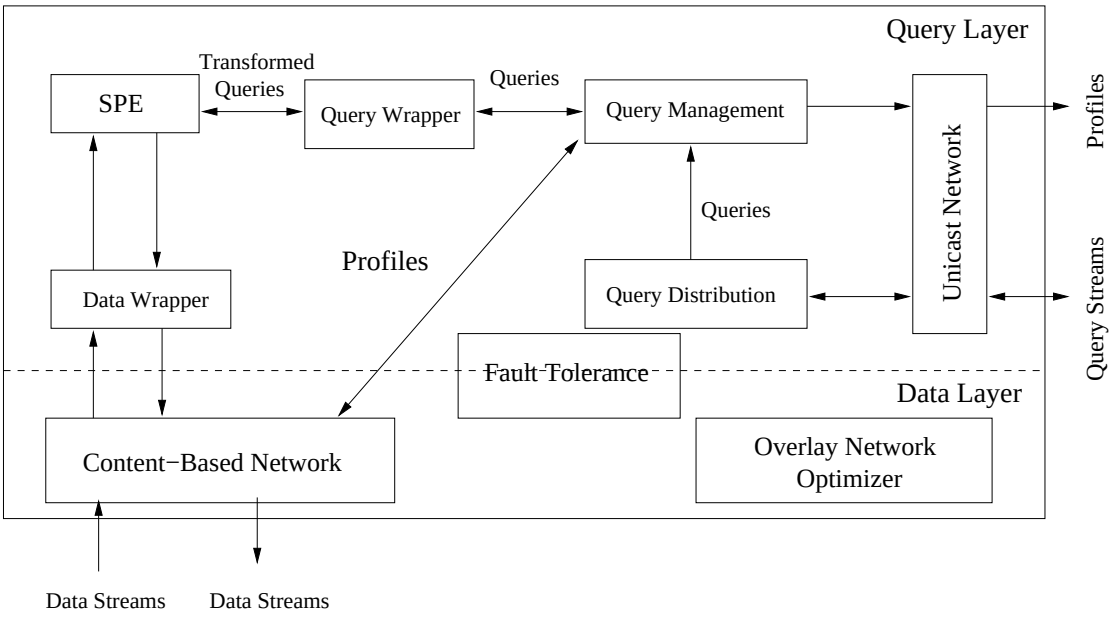


Figure 3.1: Software architecture of a node at the inter-provider layer

SPE, a data wrapper and a query wrapper can be plugged into the system to translate the data and the queries between COSMOS and the SPE.

3.1.1 Data Overlay

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

item and only forward it to those interested parties.

The data overlay modules of an SP in COSMOS plays the role of the brokers in a CBN. Following traditional CBN, we can compose a data interest profile as follows. Each profile is a disjunction of a few filters. Each filter is defined on one stream and is only applicable to this stream. Furthermore, a filter is a conjunction of constraints on the values of a set of attributes from the stream that the filter is defined on. A tuple is said to be covered by a filter, if the tuple is from the data stream of the filter and satisfies all the constraints in the filter. Furthermore, a tuple is covered by a profile if it is covered by any filters in the profile.

To exploit more opportunities to reduce communication cost, we extend the data overlay to perform projections. Early projection can save the cost of transmitting unnecessary attributes. Hence, in addition to the filters mentioned above, each profile also contains one set of attribute names for each of its requesting streams. When a node receives a tuple, it first finds out which stream the tuple is from and then evaluates the corresponding filters on the tuple. For each profile that has a filter being satisfied by the tuple, the projecting attribute set of the corresponding stream is retrieved and the projection operation is done on the tuple.

In summary, a data interest profile p_i is a triple $\langle \mathcal{S}, \mathcal{P}, \mathcal{F} \rangle$, where $\mathcal{S}$ is a set of stream names, $\mathcal{P}$ specifies the set of attributes of streams in $\mathcal{S}$ that are of interest, and $\mathcal{F}$ is a set of filters applied to streams in $\mathcal{S}$ in a similar form as traditional subscription profiles.

In this chapter, we assume there exists an efficient data overlay and adopt a similar scheme as SemCast [62]. In this scheme, the data space is partitioned into multiple subspaces by dividing each stream into multiple substreams. We denote the total set of substreams in the system as $SS = \{ss_1, ss_2, \ldots, ss_{|SS|}\}$. Logically,

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

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

When a tuple arrives, it is matched to a substream and then sent to those destinations that are interested in the substream. As the focus of this chapter is on the design of the query overlay, we shall not discuss the data overlay any further in this chapter.

3.1.2 Query Overlay

In COSMOS, a user first connects to a SP which works as the proxy for the user and is responsible for retrieving the result stream from the network and sending it back to the user. User queries are specified in an SQL-like language similar to CQL. They are handled by the query overlay. For simplicity, we only consider continuous queries and assume they do not involve stored tables. Figure 3.2 illustrates the handling procedure of a new user query. It is first distributed to an SP, say sp_i, by the load management service (provided by the Query Distribution module in Figure 3.1) for processing. The query management module of sp_i will analyze the query, and a new query or a modification of an existing query is sent to the SPE. A subscription profile, say p_1, is composed for sp_i to retrieve the source data from the sources. It will be submitted to the data overlay and handled by the content-based network. In addition, another subscription profile p_2 is created for the user to retrieve his result stream.

In the following sections of this chapter, we will study the the two critical

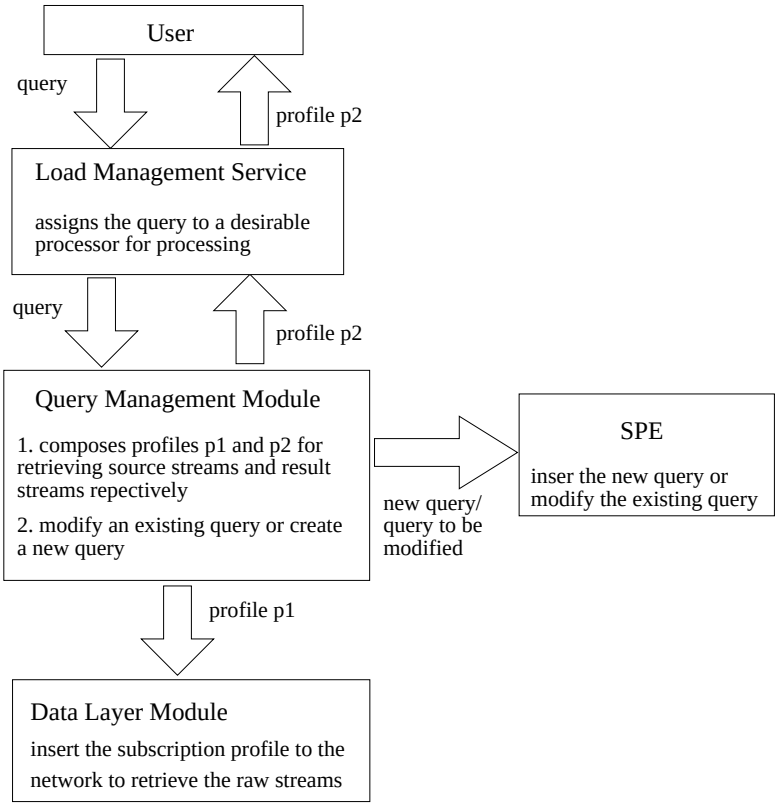


Figure 3.2: The handling procedure of a query

component of the above procedure, namely the query management module and the query distribution module.

3.2 Query Management

As mentioned before, when a query is allocated to an SP, the query management module is responsible for composing the data interest profiles for the SP to retrieve the data for processing and the profiles for the users to retrieve the results.

For each query, a profile is composed for retrieving the source data. The selection predicates applied to each individual source stream are extracted to compose the filters of the profile. Then a projection predicate is composed by using all the

attributes that appear in the query. Consider the following two queries (specified in CQL [81]) as an example:

```
Q1:                                    Q2:
    SELECT R.A, S.C                        SELECT R.A
    FROM   R, S                            FROM   R
    WHERE  R.B=S.B AND R.A>10              WHERE  R.A>20
```

Then the profiles $p_i = \langle \mathcal{S}_i, \mathcal{P}_i, \mathcal{F}_i \rangle$ to retrieve the source data can be composed as follows respectively: $\mathcal{S}_1 = \{R, S\}$, $\mathcal{P}_1 = \{R.A, R.B, S.B, S.C\}$, $\mathcal{F}_1 = \{R.A > 10\}$, $\mathcal{S}_2 = \{R\}$, $\mathcal{P}_2 = \{R.A\}$, $\mathcal{F}_2 = \{R.A > 20\}$.

In existing stream processing engines, different result streams are generated for different queries and transferred to the users independently. This is because users are assumed to be directly connected to the server in traditional systems. Following this approach, we can also compose one profile for each user to retrieve the result stream. First, a unique stream name is assigned to the result stream. Then a profile can be composed by using this unique stream name without filter and projection predicates.

However, this approach does not exploit the sharing of result stream delivery among different queries and may result in large communication overhead in our system as illustrated by the following example. Table 3.1 lists a few queries drawn from an auction stream monitoring application specified using CQL [81]. The schema of the two streams are:

- OpenAuction (itemID, sellerID, start_price, timestamp)
- ClosedAuction(itemID, buyerID, timestamp)

Consider the join queries, Q_3 and Q_4, presented in Table 3.1. We can see that the result tuples of Q_3 and Q_4 have overlaps in their result streams (since the auctions closed within five hours contains those closed within three hours). Furthermore, their projection attributes also overlap. Consider an overlay structure

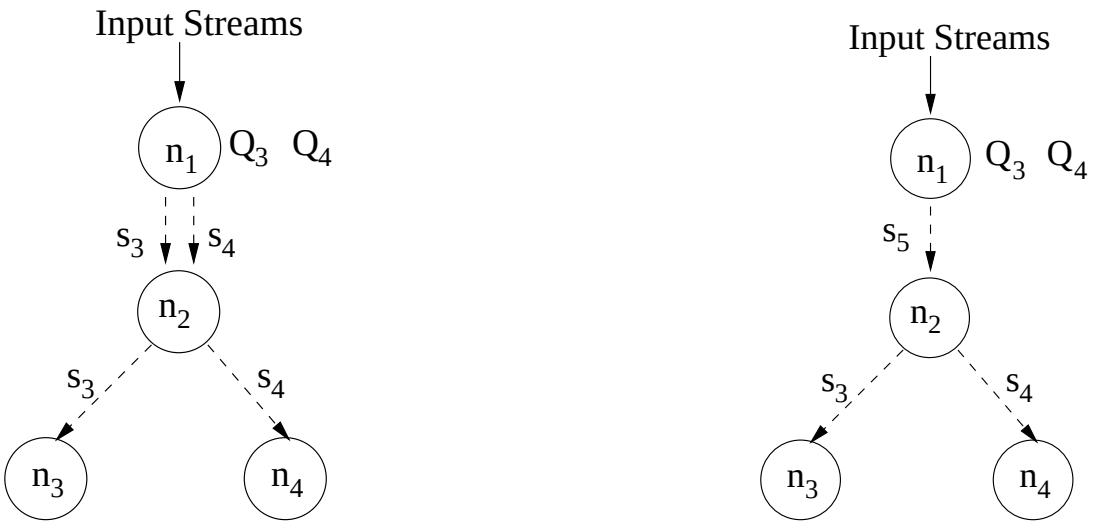


Figure 3.3: Result stream delivery

depicted in Figure 3.3(a). Nodes n_3 and n_4 issue two queries Q_3 and Q_4 respectively. These two queries are allocated to node n_1 for processing. Using traditional techniques, their result streams, s_3 and s_4, are transmitted separately as shown in Figure 3.3(a). Hence the overlapping contents of s_3 and s_4 are transmitted twice over the link between n_1 and n_2.

Note that existing multi-query optimization techniques also suffer from this problem. For example, one shared join operator can be created for the above two queries. However this join operator would still generate two separate result streams for the two queries respectively.

To solve the problem, we should send one result stream s_5 to n_2, which is the superset of both s_3 and s_4, and "split" it into two streams s_3 and s_4 at node n_2. This approach is illustrated in Figure 3.3(b). To implement this scheme, one approach is to re-engineer a "specialized" SPE which can generate one result stream for multiple queries. However, such an intrusive approach is not desirable as it requires complex "low-level" software development and tightly coupled interaction

Q_3: Report all auctions that closed within three hours of their opening	
SELECT	O.*
FROM	OpenAuction [Range 3 Hour] O,
	ClosedAuction [Now] C
WHERE	O.itemID = C.itemID
Q_4: Report the items and buyers of auctions closed within four hours of their opening	
SELECT	O.itemID, O.timetamp, C.buyerID, C.timestamp
FROM	OpenAuction [Range 5 Hour] O,
	ClosedAuction [Now] C
WHERE	O.itemID = C.itemID
Q_5: Report all auctions that closed within five hours of their opening and their buyers	
SELECT	O.*, C.buyerID, C.timestamp
FROM	OpenAuction [Range 5 Hour] O,
	ClosedAuction [Now] C
WHERE	O.itemID = C.itemID

Table 3.1: Example queries

between the SPEs and the distributed system.

Instead, we propose a query reformulation approach. For a group of queries that have overlapping results, our method composes a new query Q that contains all the queries in this group, i.e. the result of Q is a superset of the result of each query in this group. For example, we can create a new query Q_5 listed in Table 3.1, which contains Q_3 and Q_4, and issue Q_5 to the SPE at n_1 instead of Q_3 and Q_4. The result stream s_5 is "split" at n_2 using the filtering mechanism of the data overlay. More specifically, the following two profiles are sent to n_2 by n_3 and n_4 respectively:

- p_1: $\mathcal{S} = \{s_5\}, \mathcal{P} = \{O.*\}, \mathcal{F} = \{-3(hour) \leq O.timestamp - C.timestamp \leq 0\}$.
- p_2: $\mathcal{S} = \{s_5\}, \mathcal{P} = \{O.itemID, O.timetamp, C.buyerID, C.timestamp\}, \mathcal{F} = \{-5(hour) \leq O.timestamp - C.timestamp \leq 0\}\rangle$

Tuples that pass p_1 are sent to n_3 and those that pass p_2 are sent to n_4.

In our approach, each SP maintains a number of query groups such that queries inside each group have overlapping results and it is beneficial to rewrite these queries into one query Q which contains all the member queries Q_i. Such a query Q is called the representative query of the query group. The benefit of the rewriting can be estimated as $\sum_i C(Q_i) - C(Q)$, where $C(Q)$ is the estimated rate of the result stream of Q.

Query containment and equivalence is a fundamental problem which has been extensively studied in the literature. For example, [29, 68] studied the conjunctive select-project-join queries and union thereof; [34, 60] discussed the aggregate queries; [51] studied queries with arithmatic comparison predicates; [21] investigated problems of recursive queries. We, however, need to extend these techniques to the continuous stream query context. Some related literature studies the use of views to answer user queries [44]. This direction studied how to rewrite a query such that the given views of the underlying relations can be utilized to answer the original query. However, our work is kind of the other way round. We have to compose a "view" of the streams that can be utilized to answer multiple queries using the simple filtering mechanism in a CBN.

First of all, we have to extend the query containment and equivalence definition of traditional queries to continuous stream queries. Traditionally, query containment and equivalence is defined as follows.

Definition 3.1 *A query Q_1 is contained by another continuous query Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if for all database instances D, $Q_1(D)$ is a subset of $Q_2(D)$, i.e. $Q_1(D) \subseteq Q_2(D)$, where $Q_i(D)$ is the result of evaluating Q_i over D. Q_1 and Q_2 are equivalent if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.*

The above definition is based on set semantics. It can be extended to bag semantics

in a straightforward way.

However, in the continuous query context, the result data are continuously generated and hence this traditional definition is no longer applicable. To address this problem, we assume there is an application discrete time domain $\mathcal{T}$ where the timestamps of the input stream data are drawn from. We denote the temporal result data set of a query Q evaluated on a stream instance S at the time instance $\tau \in \mathcal{T}$ be $Q(S, \tau)$, which is the result of evaluating Q over all the data from S with timestamps smaller or equal to τ. Furthermore, let S be the whole set of streams. We have the following definition.

Definition 3.2 *A continuous query Q_1 is contained by another continuous query Q_2, denoted by $Q_1 \sqsubseteq Q_2$, if for all stream instances S, $Q_1(S, \tau) \subseteq Q_2(S, \tau)$ at any application time instance τ. Q_1 and Q_2 are equivalent if $Q_1 \sqsubseteq Q_2$ and $Q_2 \sqsubseteq Q_1$.*

The second problem is how to determine the containment relationship between two continuous queries. We assume that there is an approach to determine such relationship between two traditional non-continuous queries. The major difference between continuous stream query and traditional query is the introduction of window semantics. In a typical continuous query over data streams, each source stream is associated with a window predicate. In this chapter, we only consider the time-based sliding window predicate introduced in CQL [81]. Recall that a time-based window takes a positive time-interval T as a parameter and defines a temporal relation composed by tuples arrived within the last T time units, where T ranges from zero to infinity. Note that if all window predicates have a parameter $T = \infty$, we can use the traditional approach to determine the containment relationship by simply ignoring the window predicates.

For queries with window predicates, we have the following lemma and theorems.

Lemma 3.1 *For a query with only a window-based join operation of two streams S_1 and S_2 with window sizes of T^1 and T^2 respectively, two tuples t_1 from S_1 and t_2 from S_2 generate a join result tuple t if and only if both the following conditions are true:*

(1) they satisfy the join predicates;

(2) $-1 \cdot T^1 \leq t_1.timestamp - t_2.timestamp \leq T^2$.

Proof: Let us look at the "if" part first. If condition (2) is satisfied, then there exists a time instance τ such that t_1 appears in $S_1(\tau, T^1)$ and t_2 appears in $S_2(\tau, T^2)$, where $S_i(\tau, T^i)$ is the set of tuples in the window of S_i with size T^i at time τ. Based on the semantics of window-based continuous query [5], tuple t should be included in the result set at time τ.

Now we proof the "only if" part. It is trivial that if condition (1) is not satisfied, t_1 and t_2 can not be joined. Furthermore, if t_1 and t_2 appear in the result set, then, from the semantics of window-based continuous query [5], there should be a time instance τ such that $t_1 \in S_1(\tau, T^1)$ and $t_2 \in S_2(\tau, T^2)$. This also implies that $-1 \cdot T^1 \leq t_1.timestamp - t_2.timestamp \leq T^2$.□

Theorem 3.1 *A select-project-join continuous query Q_1 is contained by another select-project-join continuous query Q_2 if both the following conditions are true:*

(1) $Q_1^\infty \sqsubseteq Q_2^\infty$, where Q_i^∞ is a query resulted from setting all the window sizes of Q_i as ∞;

(2) $T_1^i \leq T_2^i$, where T_j^i is the window size of the ith stream of query Q_j.

Proof: At any time instance τ, let the set of arrived tuples of the ith stream be $S_i(\tau)$ and $S(\tau) = \{S_1(\tau), S_2(\tau), \ldots, S_n(\tau)\}$, where n is the total number of

streams. Let $T_i = \{T_i^1, T_i^2, \ldots, T_i^n\}$, $Q_i^{T_j}$ be the query resulted by setting the window sizes of Q_i as T_j, and finally $S^{T_i}(\tau)$ be the set of the tuples from $S(\tau)$ whose timestamps are within the range of $[\tau - T_i^j, \tau]$ for a tuple from $S_j(\tau)$.

First, from the definition of window, it is obvious that $Q_i^{\infty}(S^{T_2}(\tau)) = Q_i^{T_2}(S(\tau))$. Furthermore, since $Q_1^{\infty} \sqsubseteq Q_2^{\infty}$, we have $Q_1^{\infty}(S^{T_2}(\tau)) \subseteq Q_2^{\infty}(S^{T_2}(\tau))$ and hence

$$Q_1^{T_2}(S(\tau)) \subseteq Q_2^{T_2}(S(\tau)) \tag{3.1}$$

Let a result tuple t from $Q_1^{T_1}(S(\tau))$ is generated by joining a set of source tuples $\{t_1, t_2, \ldots, t_n\}$, where t_i is a tuple from S_i. From Lemma 3.1, we know that these source tuples satisfy the join predicates and $-1 \cdot T_1^i \leq t_i.timestamp - t_j.timestamp \leq T_1^j$. Since $T_1^i \leq T_2^i$, we also have $-1 \cdot T_2^i \leq t_i.timestamp - t_j.timestamp \leq T_2^j$. Therefore, based on Lemma 3.1, t is also a result tuple from $Q_1^{T_2}(S(\tau))$ and then we have

$$Q_1^{T_1}(S(\tau)) \subseteq Q_1^{T_2}(S(\tau)) \tag{3.2}$$

From both Equations 3.1 and 3.2, we can derive that $Q_1^{T_1}(S(\tau)) \subseteq Q_2^{T_2}(S(\tau))$ and hence $Q_1 \sqsubseteq Q_2$. $\square$

Theorem 3.2 *An continuous aggregate query Q_1 is contained by another continuous aggregate query Q_2 if both the following conditions are true:*

(1) $Q_1^{\infty} \sqsubseteq Q_2^{\infty}$, where Q_i^{∞} is a query resulted from setting all the window sizes of Q_i as ∞;

(2) $T_i^1 = T_i^2$, where T_i^j is the window size of the ith stream in query Q_j.

The proof of this theorem is similar to Theorem 3.1 and hence, for brevity, we shall not reiterate here.

With the above lemma and theorems, we can generate the representative query for each group of queries. The profile for a user to retrieve his results from the result stream of the representative query is actually to re-tighten the constraints that have been "loosened" in the representative query.

3.3 Query Distribution

In this section, we present the details of our query distribution module. Two goals are considered:

• Balance the load among the SPs. In this thesis, we only focus on the CPU load. We assume the relative computational capability (the CPU speed) of each SP is known. For example, we can set the capability of one SP as the basic capability and associate it with a value 1. If a SP is l times more powerful than this basic SP, its capability is valued as l. Furthermore, the load of a query is estimated as the CPU time that the query will consume per unit time in the basic SP. Hence if the total query load is L and the total capability of the SPs is C, the desirable load that should be allocated to a SP with capability value l is $l \cdot \frac{L}{C}$. However, instead of achieving absolute load balancing, we allow a certain degree of load imbalance among the SPs. The load allocated to a SP should not exceed $(1 + \alpha) \cdot l \cdot \frac{L}{C}$. In our study, we set α to 10%.

• Minimize the total communication cost. The communication cost can be divided into two parts: (1) transferring source streams from the sources to the SPs; (2) transferring query results from the SPs to the users. Following existing work [2, 63, 55], to measure the communication efficiency, we use the weighted unit-time communication cost $\sum_{\forall i,j} r(n_i, n_j) \cdot d(n_i, n_j)$, where $r(n_i, n_j)$ is the per-unit time traffic (bit/s) on the link between n_i and n_j, and $d(n_i, n_j)$ is the transfer

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

To achieve both of the above two goals, we should allocate the queries onto the N SPs such that the communication cost is minimized under the condition that the load is balanced. Furthermore, the query distribution module is at the query overlay while the data dissemination topology is decided by the data overlay. Therefore, to achieve loose coupling between the two overlays, our algorithm should not assume any knowledge of the topology of the content-based network at the data overlay. Otherwise, any change of data overlay may require the re-optimization of the query overlay and vice versa. In the following subsections, we first present the theoretical model of the problem and then present the proposed solution.

Unfortunately, typical DPSS does not consider load balancing and simply allocates user queries to the closest brokers. The DSPEs proposed in [74, 86] employed load balancing techniques for a cluster of locally distributed SPs, but they did not consider the communication cost. Thus, they are not suitable for a widely distributed network. The load management scheme proposed in [91] is also only for

locally distributed systems. On the other hand, in [2, 63, 55], optimization algorithms were proposed to fine tune the distribution of the query operators of each query across a set of widely distributed SPs to minimize the communication cost. However, these techniques failed to address the load balancing problem. They are suited for applications that only need to process a small number of complex queries. Authors in [4] proposed scheduling algorithms to maximize the weighted output rate under the situation with very limited buffer size and bursty stream rates, which is not considered in this chapter. The authors of [78] studied the static operator placement problem in a hierarchical stream acquisition architecture, which cannot be applied in our architecture. Load balancing is also ignored in this work. A common weakness of the above techniques, which require the construction of a global query/operator graph, is their scalability to the number of users.

Our new load management scheme distinguishes itself by achieving both load-balancing and minimum communication cost, taking the communication characteristics of the DPSS architecture into consideration as well as its scalability. Recall that COSMOS is designed to support a large number of clients. Therefore, we expect the queries to be *streaming*, i.e. the frequency of query arrival and query departure/completion is high. As such, in COSMOS, we opt to distribute queries to SPs instead of operators. In other words, we adopt a query level load distribution scheme (instead of an operator level scheme). Several reasons prompted this design decision. First, it is simpler (than an operator-based scheme) and practical. Consider the SPs in COSMOS are autonomous. Thus, they may install different stream processing engines or different versions of the same engine. Hence distributing query load at the operator level may be infeasible. For instance, moving a window join operator from the STREAM system to a TelegrahCQ system is hard

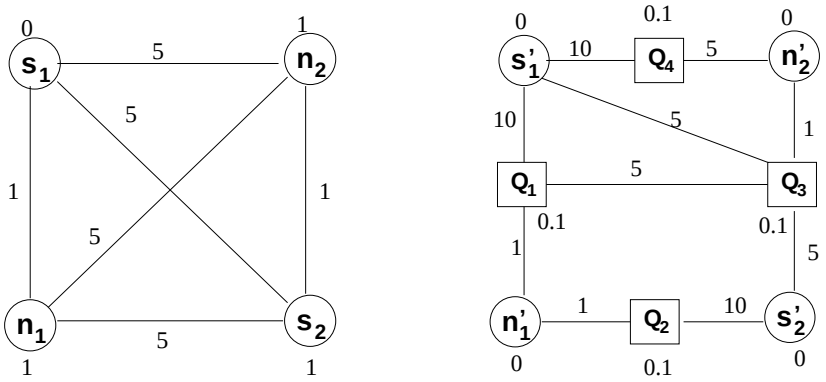
to implement, because it relies on a special data structure "synopsis" implemented in STREAM which is not only manipulated by the join operator itself but also other operators before or after the join operator. Furthermore, even if the SPs use the same engine, one may upgrade its engine without informing the others. This might also give rise to problems unless forward and backward compatibility is implemented. Second, operator level load distribution may tighten the coupling of the SPs. Besides adopting the same processing model and data model, the SPs may also have to synchronize with each other during the processing of a query. Third, the number of queries in the system would be large. Moreover, at each instance of time, there would be a large number of queries streaming in and out of the system. Distributing at the operator level would be too complex to be scalable to the fast query streams.

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

3.3.1 Problem Modeling

In this subsection, we model the problem as a graph mapping problem. This model differs from those of the existing work by taking the communication characteristics of a DPSS into consideration. For ease of exposition, we assume a data source is also a SP here. Hence, we refer to all the nodes in the network as SPs. The word "data source" refers to those SPs which are also the origins of one or more source streams. Actually, for any node that cannot process queries, we can treat it as a SP with zero capability value.

We first construct a network graph $NG = \{V_n, E_n, W_n\}$, where each vertex $v_i \in V_n$ represents a SP in the network and there is one edge $e_{ij} \in E_n$ between



(a) Network graph (b) Query graph

Figure 3.4: Graph

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

Table 3.2: Mapping Schemes

each pair of vertices v_i and v_j. The weight of each vertex v_i is given by $W_n(v_i)$. $W_n(v_i)$ is equal to the SP's capability value. Furthermore, the weight of an edge e_{ij} is also given by $W_n(e_{ij})$ and is equal to the communication latency between v_i and v_j. Figure 3.4(a) shows an example network graph composed by four network nodes. The weights of the vertices and edges are drawn around them. In this example, there are two data sources, s_1 and s_2, which have no computational capability (in terms of complex query processing) and two SPs, n_1 and n_2, have the same computational capability value.

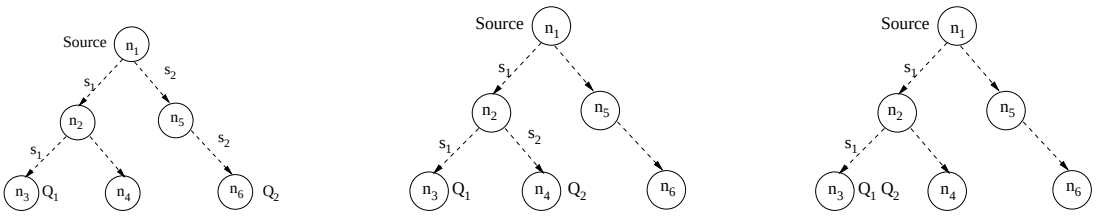
Second, we also construct a *query graph*, $QG = \{V_q, E_q, W_q\}$. There are two types of vertices in V_q: query vertex (q-vertex) and network vertex (n-vertex).

A q-vertex represents a query while an n-vertex represents a SP in the network. Figure 3.4(b) shows the query graph when four queries are submitted to the network of Figure 3.4(a). In the figure, there are four q-vertices, which are drawn in rectangles, and four n-vertices, which are drawn in circles.

If a query requests data from a data source, there is an edge that connects the query and the data source. Furthermore, we adopt the same assumption as a DPSS that a user is allocated to his closest SP when he joins the system in the first place. The user and the SP are said to be local to each other. Since the result stream of a query is routed to the user by the DPSS architecture of the data overlay, it is first routed to the user's local SP and then to the user. Therefore, the cost of transferring the query result from a SP to its local users are unavoidable. Hence we do not need to model the communication between the SP and its local users, instead we use one edge to connect each query to its local SP to model the communication of the query's result stream. For example, in Figure 3.4(b), Q_1 and Q_2 request source data from s_1 and s_2 respectively and both are local to n_1. In addition, if a query's data source and its local SP happen to be the same node, only one edge connects the query and that node.

In a query graph, the weights of the vertices and edges model the query workload and the communication traffic of the stream delivery. Each q-vertex is weighted with the estimated load that would be incurred by the query at the SP with the basic capability, while n-vertices are assigned with zero weights. In addition, each edge is weighted with the estimated data rate (bit/s) of the source stream(s) (for those edges that represents source stream deliveries) or the result stream (for those edges that represents result stream deliveries). For example, in Figure 3.4(b), Q_1 would consume 0.1 CPU cycles in a SP with basic capability. In addition, it requests 10 bit/s data from source s_1 and generates 1 bit/s result

streams whose user is local to n_1. Moreover, if a query's data source and its local SP happen to be the same node, then the weight of the edge connecting the query and that node is equal to the sum of the data rate of the source stream(s) and the result stream.



(a) Scheme 1 (b) Scheme 2 (c) Scheme 3

Figure 3.5: Different allocation schemes of Q_1 and Q_2

Until now, our model is similar to those appear in existing work. However, it is not enough for our problem. Note that query communication can be shared among queries in a DPSS architecture. For example, consider queries Q_1 and Q_2 in Section 3.2. In Figure 3.5, we illustrate three different allocation schemes of Q_1 and Q_2 in a network configuration. In the figure, s_i represents the streaming data that are requested by the query Q_i. We can see, from the data retrieving profiles of Q_1 and Q_2 that s_1 contains s_2. In Figure 3.5(a), Q_1 and Q_2 are allocated to two nodes n_3 and n_6 respectively. n_3 and n_6 are far away from each other and hence they have different data routing paths from the same source. As depicted in Figure 3.5(b), by changing the allocation of Q_2 to n_4 which is closer to n_3 and hence shares part of the data routing path from the source with n_3, we can save some communication cost. That is because the data requested by the two queries have large overlap and the transfer of these data can be naturally shared by the content-based network in the data overlay if they have similar routing path. Furthermore, allocating them to the same node can further reduce the communication cost as illustrated in Figure 3.5(c). Therefore, to accurately model the communication

cost, we should take this effect into consideration.

Specifically, we add one edge between each pair of queries that have overlap in their data interests. The weight of such an edge is equal to the arrival rate of the data that are of interest to both of its end vertices (queries). The intuition is to penalize allocation schemes that distribute the two queries to two nodes that are very far away from each other. For instance, the weight of the edge connecting Q_1 and Q_3 in Figure 3.4(b) is equal to the rate of the data that are of interest to both of them. In this case, the data requested by Q_1 from s_1 happens to contain those of Q_3.

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

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

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

$$\forall v_j \in V_n, \sum_{\substack{v_i \in V_q \\ M(v_i, v_j)}} W_q(v_i) \leq (1+\alpha) \cdot W_n(v_j) \cdot \frac{W_q^v}{W_n^v}, \tag{3.3}$$

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

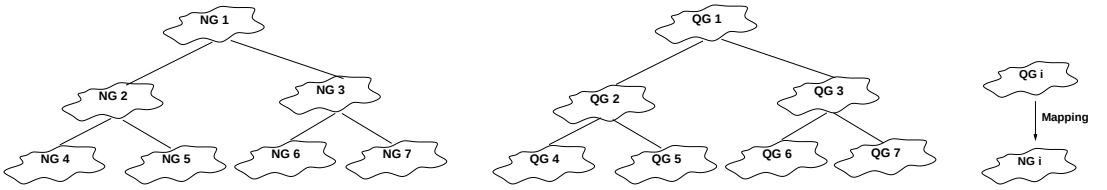
3. ***minimizes the Weighted Edge Cut (WEC):*** *which is given by*

$$WEC = \sum_{\substack{v_k \in V_n \\ v_l \in V_n}} \sum_{\substack{v_i \in V_q \\ v_j \in V_q \\ M(v_i, v_k) \\ M(v_j, v_l)}} W_q(e_{ij}) \cdot W_n(e_{kl}). \tag{3.4}$$

In Table 3.2, we present three mapping schemes from the query graphs to the network graphs in Figure 3.4, which obey both the network constraint and the load-balancing constraint. In scheme 1, we map all the queries to their own local SPs, while scheme 2 is the optimal mapping if we ignore the potential sharing of communication of Q_1 and Q_3. We can see that scheme 3 is more optimal, which has a smaller WEC value.

3.3.2 Challenges and Approach Overview

There are a few practical difficulties to solve this problem. First, it is hard to construct the global network graph and query graph when the size of the network and the number of queries scales up. A scalable algorithm is required. Second, even if we have the global graphs, it is an NP-Hard problem [69]. Hence, an efficient heuristic-based approach is needed. Third, the queries and stream statistics could change over runtime. A runtime algorithm is required to redistribute the queries.



(a) Network graph hierarchy (b) Query graph hierarchy (c) Mapping

Figure 3.6: Approach overview

To address the problems, distributed coordinators are employed to perform

the heuristic graph mapping and remapping algorithms. They are organized into a hierarchical tree. Each leaf coordinator constructs a network (sub)graph which consists of an exclusive set of SPs while a parent coordinator constructs a network (sub)graph composed by its child coordinators. This provides a hierarchical view of the network graph (Figure 3.6(a)). On the other hand, each coordinator also holds a query (sub)graph which is a coarsened overview of its descendants' and this constructs a query graph hierarchy (Figure 3.6(b)). Each coordinator only performs the mapping and runtime remapping of its query (sub)graph to its network (sub)graph (Figure 3.6(c)). The rest of this section presents the detail of our scheme, which is implemented in the query distribution module.

3.3.3 Coordinator Hierarchy Construction

The coordinators are a subset of SPs chosen from all the SPs in the system. Each such SP performs two separate logical roles: the stream processor and the coordinator while the non-coordinators perform only the stream SP role. We assume that separate resources of these SPs are reserved for these two roles. Hereafter, the words "processor" and "coordinator" refer to the logical roles.

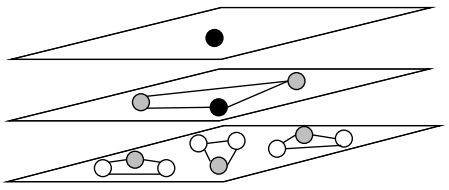


Figure 3.7: Hierarchical Coordinator Structure

The coordinators are organized into a hierarchical tree. An example of this structure is illustrated in Figure 3.7. At the bottom level, each SP forms a separate cluster and the SP is also called the parent of this cluster. At the second level, the SPs are clustered into multiple close-by (in terms of transfer latency) clusters.

Within each cluster, the median is selected as the coordinator of the cluster which is also called the cluster's parent. The median of a set of SPs $\{n_1, n_2, \ldots, n_l\}$ is defined as the SP n_i with minimum total transfer latency to all SPs in the cluster, i.e. $\sum_{1\leq j\leq l} d(n_i, n_j) \leq \sum_{1\leq j\leq l} d(n_k, n_j)$ for any n_k. These coordinators are also clustered level by level in a similar way. We say a SP belongs to a cluster of an internal coordinator (at any level) if it is the descendant of this coordinator.

Each coordinator constructs a network subgraph containing only its child coordinators (or child SPs for the leaf coordinators). Here, the weight of a vertex is equal to the total capability values of all its descendant SPs. For the example in Figure 3.7, the SPs are organized into three clusters at the bottom plane and the median of each cluster (drawn in gray) is selected as the coordinator. At the next plane, the coordinators only form one cluster and hence the clustering stops here.

In this chapter, we adapt the distributed mechanism proposed in [14] to dynamically construct a hierarchical tree of coordinators. Here we only briefly present the mechanism and refer the interested readers to reference [14] for more details. The mechanism tries to maintain a tree with the following properties: (1) the size of the cluster in each level is between k and $3k-1$ (except the cluster of the root whose size could be less than k); (2) the parent is the center of its cluster, i.e. with the minimum average delay to all the other nodes in the cluster. The tree is constructed incrementally and dynamically.

1. When a new SP requests to join the network, its request is first directed to the root node. For each node that receives a join request, if it is the leaf coordinator, it adds the new node as its child node. Otherwise, it identifies the child coordinator closest to the new node and direct the request to that child.

2. If a node leaves the network, a message is sent to its parent and children (if any). If it is a coordinator, a new parent is reselected among its remaining

Algorithm 3.1: Query graph coarsening algorithm

```
1  while |V| > v_max do
2      Set all the vertices as unmatched;
3      while ∃ unmatched vertices ∧ |V| > v_max do
4          Randomly select an unmatched vertex u;
5          A ← adj(u) − mat(adj(u)) ;
6          if is_n(u) then
           A ← A−{v|v ∈ adj(u)∧is_n(v)∧(u.clu ≠ v.clu∨v.clu = unknown)};
7          Select a vertex v from A such that the edge e(u, v) is of the
           maximum weight;
8          Collapse u and v into a new vertex w;
9          Set w as matched;
10         w.weight ← u.weight + v.weight;
11         Re-estimate the weights of the edges connected to w;
12         if is_n(u) OR is_n(v) then
13             is_n(w) ← true;
14             w.clu = is_n(u)?u.clu : v.clu;
```

children. Furthermore, heartbeat messages are sent periodically among the parent and children to detect any node failure.

3. If a coordinator finds out that the number of its children exceeds $3k - 1$, it partitions the cluster into two clusters, each of size at least $\lfloor 3k/2 \rfloor$, such that the radii among the two clusters are minimized. The centers of the two clusters are selected as the two new parents.

4. If the number of children of a coordinator x falls below k, it sends a merge request to the closest sibling y. y adds all the children of x to its cluster.

5. Periodically, a new parent is selected if the current parent is no longer the center among its cluster.

3.3.4 Query Graph Hierarchy Construction

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

and generate a query graph over them. If the number of vertices of the query graph is larger than v_{max}, then it runs the algorithm in Figure 3.1 to coarsen the query graph. The graph mapping algorithm at each coordinator, which will be presented in the following sections, is performed on this coarsened query graph. The coarsening algorithm repeatedly collapses two selected vertices until the number of vertices is smaller than or equal to v_{max}. In the algorithm, a vertex u tends to collapse with a neighbor v which has an edge $e_{u,v}$ with a larger weight, because these two vertices are more likely to be mapped to the same vertex in the network graph. For ease of exposition, we define the following functions: (1) $adj(u)$ returns the set of adjacent vertices of u; (2) $is\_n(u)$ returns true if u is an n-vertex; (3) $matched(A)$ is all the matched vertices in a vertex set A. In addition, for each n-vertex u, a field clu indicates which child cluster of the current coordinator covers u. Two n-vertices belong to two different child clusters shall not be merged together because they have to be mapped to different child clusters in the graph mapping algorithm. Note that if u is not covered by any child cluster of this coordinator, then their clu field is set as unknown.

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

3.3.5 Initial Query Distribution

Once the initial query graph hierarchy is constructed, the root coordinator starts mapping its (coarsened) query graph to its network (sub)graph. The query sub-

graph mapped to each child is uncoarsened one level back and sent to the child. This procedure repeats at each level until all the queries are assigned to the SPs. Note that, to uncoarsen a vertex, information of the finer-grained vertices, if necessary, is retrieved from the corresponding coordinator based on the tags of the vertex.

Algorithm 3.2: Graph mapping algorithm

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

The algorithm is illustrated in Figure 3.2. It starts by using a greedy algorithm to get an initial mapping:

(a) Map each n-vertex to a child that manages the node that n-vertex represents.

(b) Map the q-vertices one by one in descending order of their weights. For each q-vertex, among the children that can accommodate it (i.e. their load-balancing constraints will not be violated after mapping the q-vertex to anyone of them),

map it to the one that minimizes the current WEC. If no children can accommodate it, then map it to the one with the minimum violation of the load-balancing constraint.

We cannot guarantee to find a mapping efficiently satisfying the load-balancing constraint because it is an NP-Complete problem.

Lines 3.2-3.2 iteratively improve the mapping by trying to remap the q-vertices to other vertices in NG. Here, we use the value of $gain(v_i, v_k)$ to heuristically guide our remapping, which is equal to the reduction of the WEC value by remapping $v_i \in V_q$ to $v_k \in V_n$. To achieve some capability of climbing out of local minima, a q-vertex v_i with a negative $gain(v_i, v_k)$ value would be considered for remapping as long as its gain value is the largest and its remapping will not violate the load-balancing constraint of v_k. The mapping with minimum WEC value will be restored at the beginning of each outer iteration.

3.3.6 Online Query Routing

Unlike prior studies which assume queries are relatively stable or updates are infrequent, our system stresses the problem of fast query streaming. The new queries have to be quickly routed to the desirable SPs. While there are many possible query routing schemes, in this thesis, we only study the use of the hierarchical coordinator tree and show the significance of online query routing for the system performance. In this scheme, a new query is first routed to the root coordinator which then routes it to one of its children. The routing is done level by level until the query is assigned to a SP. At each coordinator, the query is added to the query graph and the weights of the new edges are estimated. Then the new vertex is mapped to a vertex in the network graph such that the WEC is minimized.

Although all queries have to be routed through the root coordinator, this

scheme is scalable to very fast query streams. This is because it only needs to route the queries to a few children based on some coarse-grained information. As shown in Section 3.4, it can handle more than 800,000 queries per second in our experimental PC. For higher query stream rates, we can perform online routing only on some queries while simply put the other queries at their local SPs. Further trade-offs between routing quality and routing efficiency is an interesting piece of future work.

3.3.7 Adaptive Query Redistribution

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

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

1. Minimize the WEC of the mapping.

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

Algorithm 3.3: Adaptive load re-balance

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

In the load balancing phase, to avoid re-mapping from scratch, which may incur too many query migrations, we adopt a load diffusion approach [46]. A diffusion solution specifies the load m_{ij} that should be migrated from the child coordinator c_i to another child coordinator c_j for each (i, j) pair. Authors in [46] proposed a method to derive a diffusion solution such that the Euclidean norm of the transferred load is minimized which may result in a small number of query migrations. Our redistribution algorithm is presented in Figure 3.3. The n-vertices are not considered for redistribution. Therefore, the vertices in the algorithm only refer to the q-vertices. The benefit of remapping a vertex from c_i to c_j is defined as the reduction of the weighted edge cut given by Eqn (3.4). To achieve good mapping quality, our algorithm tends to remap those vertices with large benefits.

Furthermore, a vertex is called *dirty* if it had been picked for remapping in the earlier iterations in the same adaptation round. We give these vertices higher remapping priority because moving them again would not increase the amount of query migration (Note that queries are actually moved after all the decisions are

made in one round.). In addition, the *load density* of a vertex is equal to the weight divided by the size of its state. We favor remapping the denser ones because it may result in less state movement. The value of x in line 5 can be used to trade mapping quality for lower migration cost. With a larger x value, we can consider more vertices with lower migration benefit. In our experiments, we set $x = 10$.

The distribution refinement phase attempts to reduce the weighted edge cut while maintaining the load balancing condition. Again the query vertices are visited randomly and checked to see whether it belongs to one of the following categories: (1) Mapping the vertex back to its original location can maintain load balance and the current WEC. (2) Mapping the vertex to another node can decrease the current WEC without violating load balance. The checks are performed in the order given above. Whenever such a vertex is found, the remapping is performed.

3.3.8 Statistics Collection

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

3.3.9 Coping with Network Changes

While we do not address the issue of fault tolerance in this thesis, our techniques allow the dynamic joining/leaving of nodes in the system. We cope with such changes of the network as follows:

- New SP joins. We do not need to explicitly address this kind of changes, because the adaptive query redistribution mechanism will detect an uneven load distribution and then redistribute the queries accordingly.
- Processor departs. If the departure is actively requested by the SP, then it will request the parent coordinator to redistribute its running queries. After being informed of the new locations of the queries, the SP extracts the runtime states of the queries and send them to the new locations. A node departure without such explicit actions is deemed as node failure and treated by the fault tolerance mechanism, which is not discussed in this thesis.
- Cluster splits/merges and parent changes. When a cluster is split, the parent of the cluster will also split its query graph accordingly and transfer them respectively to the two new parents. When two clusters are merged, the two query graphs are merged in the parent of the new cluster. Upon the change of parent, the query graph is simply transferred to the new parent.

3.4 Experiments

In this section, we present a performance study of the proposed techniques. A network topology with 4096 nodes is generated using the GT-ITM topology generator. The Transit-Stub model, which resembles the internet structure, is used. Among these nodes, 100 nodes are chosen as the data stream sources, and 256 nodes are selected as the SPs, and the remaining nodes act as the routers. Our algorithms

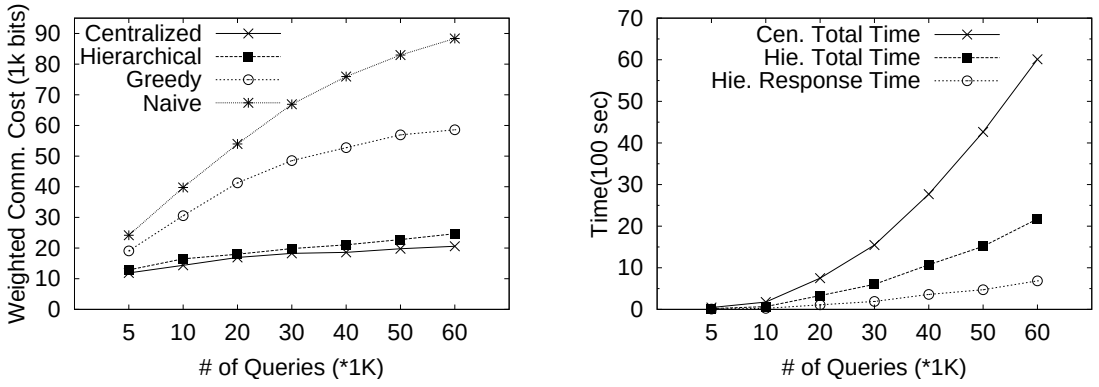
are implemented in C and the communication between the SPs is simulated. The experiments are run on a Linux Server with an Intel 2.8GHz CPU.

As it is hard to collect a large number of real query workload, following existing work [2, 63, 55], we use synthetic query workload in our experiments. To avoid biased parameter settings, we perform sensitivity studies on all the following parameters and only report those that have significant effects.

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

3.4.1 Initial Query Distribution

In the first experiment, we study the performance of the initial query distribution scheme with different number of queries. It is compared with three approaches:

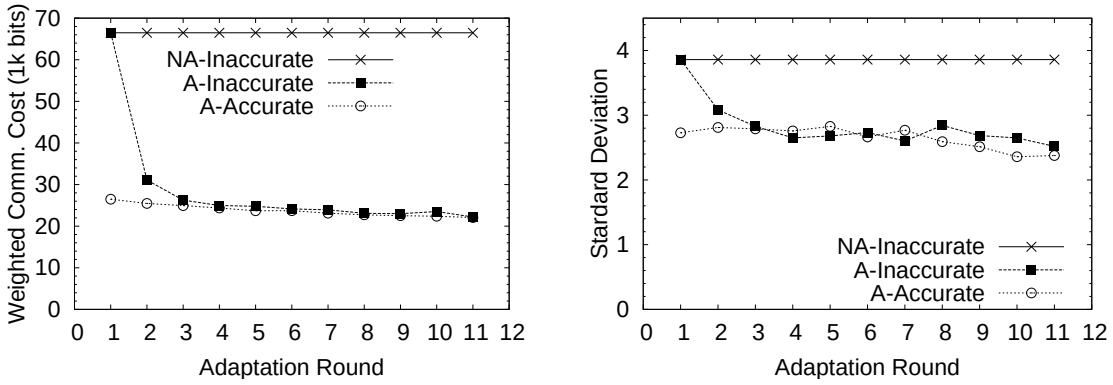


(a) Comm. cost (b) Running time

Figure 3.8: Varied #queries

(a) Naive: allocate the queries to their local SPs. (b) Greedy: only run the greedy algorithm in Figure 3.2. (c) Centralized: a centralized node constructs a global query graph and a global network graph. The algorithm in Figure 3.2 in Section 3.3.5 is run to perform a global mapping. Figure 3.8(a) presents the results of all the four approaches. Naive performs the worst because it cannot identify the data interest of the queries and optimize their locations. Greedy works a lot better. The two graph mapping algorithms perform the best and their performances are similar. This also verifies that the graph coarsening procedure in our hierarchical mapping algorithm does not incur much errors.

We also report the response time (i.e. the time interval from the begin to the end of the mapping) and the total time (i.e. the total CPU time consumed in all the coordinators) of the centralized and hierarchical graph mapping algorithms in Figure 3.8(b). It is shown that both the response time and total time of the hierarchical approach are much lower than the centralized one.



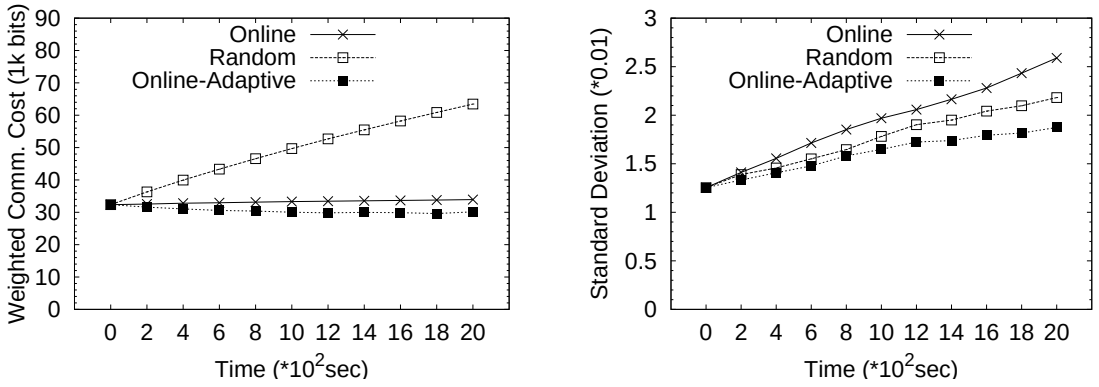
(a) Comm. Cost (b) Standard deviation of load

Figure 3.9: Adapting to inaccurate statistics

3.4.2 Adaptive Query Distribution

In the second set of experiments, we study the performance of the adaptation scheme. In the above experiments, the graph mapping algorithms perform well if accurate apriori statistics exist. However, apriori statistics are hard to collect in a large scale system. Hence, in the first experiment, we study the situation that the apriori statistics are inaccurate. We model this situation by using a random initial query allocation scheme. Three algorithms are compared: (1) NA-Inaccurate: non-adaptive algorithm with inaccurate statistics; (2) A-Inaccurate: adaptive algorithm with inaccurate statistics; (3) A-Accurate: Adaptive algorithm with accurate statistics. Figures 3.9(a) and 3.9(b) present the communication cost and the standard deviation of the system load over the observation period. It can be seen that the adaptive algorithm can gradually refine the initial query distribution scheme to minimize the communication cost and balance the system load.

In another experiment, we study how the system performs when new queries arrive in the system. Initially, there are $30,000$ queries in the system and new queries are added into the system incrementally at a 200 seconds interval. At the

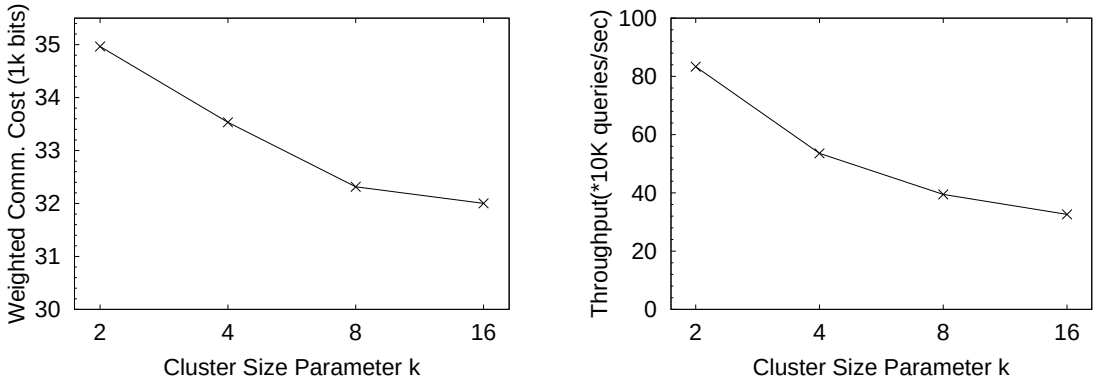


(a) Comm. Cost (b) Standard deviation of load

Figure 3.10: New query arrival

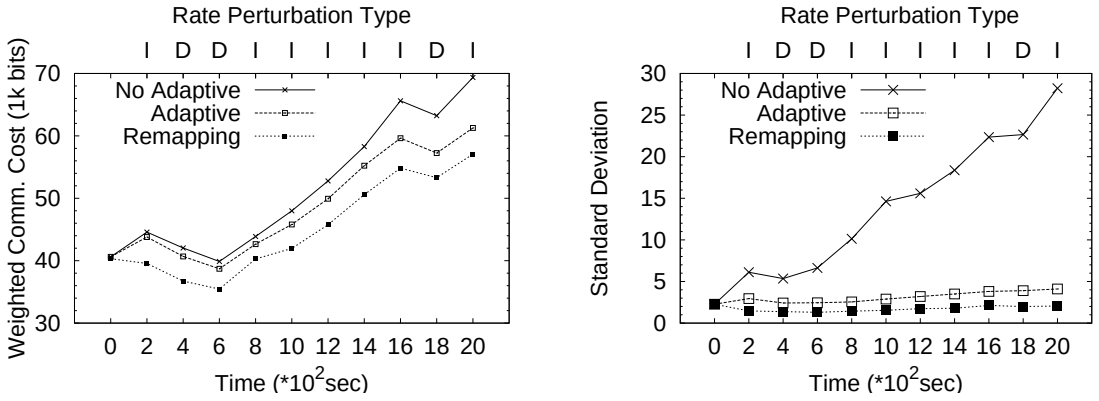
start of each interval, there are $1,500$ new queries coming in. We reported the average communication cost during each interval and the standard deviation of the SPs' workload. Three schemes are compared: (1) Random: randomly allocate the new queries without considering their interest; (2) Online: use our online query routing algorithm; (3) Online-Adaptive: use both the online query routing and the adaptive query redistribution. The results are shown in Figure 3.10(a) and 3.10(b). The performance of Random gets worse with more queries added, while Online can maintain low communication cost but with increasing load imbalance. Online-Adaptive performs the best in both metrics because of its ability to re-balance the load distribution and to refine the query distribution.

In the fourth experiment, we examine the scalability of our system to fast query streams. The settings are similar to that of the above experiment. We collect the time for the root coordinator to distribute a query and then compute the maximum query rate that it can accommodate. We study the root coordinator because it is the potential bottleneck of the system. We vary the cluster size parameter k. The results are shown in Figure 3.11. We can see that, with a smaller value of k, the query distribution quality is worse. That is because there are more levels in the



(a) Comm. Cost (b) Throughput

Figure 3.11: Varied Cluster Size



(a) Comm. Cost (b) Standard deviation of load

Figure 3.12: Perturbation of stream rates

coordinator tree and more graph coarsening is performed. On the other hand, the throughput of query streams gets better with a smaller value of k. The reason is the root coordinator needs to route queries to fewer number of children. Hence, adaptively setting the parameter k is an interesting piece of future work.

In the last experiment, we examine the performance of the system when the rates of streams change. At runtime, we increase (denoted by "I") or decrease (denoted by "D") the rates of 800 random streams several times so that load imbalance exists within the system. Here, we compare the adaptive scheme with

two schemes: (1) Re-mapping: use the centralized mapping algorithm to remap the global query graph to the global network graph; (2) Non-Adaptive: no adaptation is done. Figures 3.12(a) and 3.12(b) depict the communication cost as well as the standard deviation of the load in the system after each change. It is clear that adaptive query redistribution performs close to centralized remapping and can re-balance the system load to adapt to the new data characteristics without increasing the communication cost. While the remapping algorithm can achieve better results, it incurred about 7 times more query migrations than the adaptive algorithm did.

3.5 Summary

In this chapter, we proposed a new architectural design to leverage the strength of CBN to support scalable continuous query processing over data streams. This architecture retained the loose coupling and easy to deploy merits of a CBN, while it provided the complex query processing capabilities over data streams. To handle both the query stream and data stream, two overlays, the query overlay and the data overlay, were constructed, respectively, by two levels of functional modules. We presented the design detail of two modules at the query overlay, namely, the query management module and the query distribution module. A few issues in the new architecture were addressed: managing the queries to exploit the sharing of communication, constructing the coordinator tree, distributing the queries to balance load and minimize communication cost, online routing of fast query streams, adapting the query distribution. Solutions to these issues are presented and performance study showed their effectiveness.

Chapter 4

Data Stream Dissemination

In the previous chapter, we have seen how queries can be distributed in COSMOS. However, the data layer, which employs traditional content-based network, is not optimal for some applications where a data source continuously disseminates fast changing data objects (e.g., sensor data, stock prices and sport scores) to a number of SPs. Since the data are changing very frequently, disseminating the data upon any change would be very costly. To reduce the cost, we exploit the clients' tolerable inaccuracy of data. Clients submit queries to the SPs with their own preferences on data coherency requirements. Based on the requirements of the running queries, each SP would have its own coherency requirement of each interesting data object. Furthermore, as relying solely on the source to disseminate to all the SPs is not scalable, nodes are organized into one or more dissemination trees (with the data source being the root node) so that data/messages are transmitted to each SP through its ancestors in the dissemination tree. Each node of the tree would selectively disseminate only interesting data to its child nodes by filtering out the unnecessary ones.

The dissemination efficiency is evaluated using the metric *fidelity*, which has been used in previous work [76, 75]. It measures the portion of time that the values in the SPs conform to their coherency requirements. The loss of fidelity at

each SP is due to the dissemination delay of the update messages, which includes the communication delay as well as the processing delay in its ancestors in the trees. Therefore, minimizing the loss of fidelity can be viewed as minimizing the delay of the update messages. Interestingly, while it is important to design optimal dissemination trees in this context, there is very little study on this subject.

In this chapter, we present a cost-based approach to adapt dissemination trees in a dynamic changing environment. Our contributions include:

- We formalize the problem by formally defining the metric (fidelity) used to measure the effectiveness of the algorithms and the objective of the algorithms (i.e., minimize the average loss of fidelity over all the SPs).

- We propose a novel and thorough cost model which considers both the processing cost in the SPs as well as the communication cost in the network links. With the cost model, we can explore a larger solution space than existing methods do to achieve a more cost-effective scheme.

- Based on the cost model, we propose an adaptive runtime scheme that is robust to inaccurate statistics and runtime changes in the data characteristics (e.g., data arrival rates) and system parameters (e.g., workloads, bandwidths etc.). The proposed scheme enables nodes to independently make decisions based on localized statistics collected from neighbouring nodes to transform a dissemination tree from one form to a more cost-effective one. Furthermore, we extend the cost model to incorporate the adaptation overhead. Given apriori statistics of the system characteristics, we propose two static optimization algorithms to build a dissemination tree for relatively static systems. These static trees can also be used as initial trees in a dynamic environment.

- We conducted an extensive performance study which shows that the proposed tree construction scheme performs close to optimal, and the adaptive scheme

is also robust to changing conditions at runtime.

The rest of this chapter is organized as follows. Section 4.1 formulates the problem and presents motivations. In Sections 4.2 and 4.3, we present our solution to the single object dissemination problem, and its extension to the multi-object dissemination problem respectively. A performance study is presented in Section 4.4. Finally, we conclude in Section 4.5.

4.1 Problem Formulation and Motivations

This section formulates the problem and presents the motivation.

4.1.1 Problem Formulation

Table 4.1 lists a number of major notations that would be used frequently throughout the whole chapter. In the system, there is a data source s that stores a set of data objects $O = \{o_1, o_2, \cdots, o_{|O|}\}$, a set of nodes (SPs) $N = \{n_1, n_2, \cdots, n_{|N|}\}$, and a large number of clients. Each client submits queries involving a subset of data objects through a node (SP), and specifies a preference on the coherency on the data objects. In this chapter, a user's coherency requirement (cr) on a data object is specified as the maximum tolerable divergence of the data value from its exact value. Our approach does not restrict the way in which the divergence is measured. The possible metrics include the number of changes since last update, the deviation of the values (for numerical data), the edit distance (for string data) or the difference of the update time stamp. Instead of just picking anyone of them, our system allows a customizable divergence function. We denote the divergence function as $DIV(o_x(n_i, t), o_x(n_j, t))$, where $o_x(n_i, t)$ is the value of data object o_x perceived by node n_i at time t.

Table 4.1: Notations

s	the source node
n_i	the ith node
o_x	the xth data object
$LF(n_i)$	the lost of fidelity of n_i
C_i	the set of child nodes of n_i
GC_i	the set of grandchild nodes of n_i
O_i	the set of objects requested by n_i
O_i^m	the set of objects requested by the subtree rooted at n_i
$cr_{i,x}$	the coherency requirement of n_i on o_x
r_i	the rate of the update message meant for n_i
r_i^m	the rate of the update message meant for any node in the subtree rooted at n_i
r_i^c	the sum of the update rate over all nodes in the subtree rooted at n_i
$r_{i,x}$	the rate of the update message from o_x meant for n_i
$r_{i,x}^m$	the rate of the update message from o_x meant for any node in the subtree rooted at n_i
$d(n_i, n_j)$	the communication delay between n_i and n_j
$D(s, n_i)$	the total communication delay of the path from s to n_i in the tree
t_i^p	the time needed to perform filtering of a message at n_i
t_i^c	the time needed to perform transmission of a message at n_i
t_i^e	the time needed to collect information at n_i
t_i^d	the time needed to compute the adaptation benefit of a transformation at n_i
t_i^a	the amortized adaptation cost at n_i
t_i	the expected processing time at n_i
$g(n_i)$	the processing delay of message in node n_i
$p(n_i)$	the parent node of n_i
ρ_i	the workload of n_i

From the system's point of view, each node n_i can be viewed as a super-client that requests a subset of data objects O_i from the source, which should be the union of the objects that are requested by the queries running on n_i, and the coherency requirement $cr_{i,x}$ of n_i on object o_x is equal to the most stringent requirement of its queries that involve o_x. To determine whether an update tuple should be transferred to the child node or a client, our system also employs a customizable function $match(m, n_i)$, which returns either $true$ or $false$ for a given tuple m and a child node n_i. An application developer can design different functions for different divergence functions. [76] proposed such a function for numerical data dissemination. We will not go into detail of the design of this function and concentrate on the construction and adaptation of dissemination trees in this chapter.

To ensure scalability, we model a generic dissemination scheme as follows. The SPs N together with the source s compose an overlay network which can be modeled as a directed complete graph $G = (V, E)$, where $V = N \cup \{s\}$ and E consists of the directed arcs connecting each pair of nodes in V. To build an efficient dissemination scheme, the nodes in V are organized into one or more overlay dissemination tree T. Each T is composed by s, a set of nodes $V' \in N$ and arcs $E' \in E$. The root of all the trees is the source s. Once new values of the data objects at s arrive, s would initiate the messages and disseminate only the necessary ones to each of its child SPs in all the dissemination trees. Upon receiving a message, a SP would also selectively disseminate it to its child SPs. This process happens in each SP until the messages reach the leaf SPs.

Since it is possible for an SP's coherency requirement to be less stringent than that of its descendants, every SP n_i has an *effective* coherency requirement $cr^m_{i,x}$ on an object o_x which corresponds to the most stringent one among all the $cr_{i,x}$s of the subtree rooted at n_i. A parent performs the filtering of messages based

on the $cr^m_{i,x}$ values of its children. In addition to disseminating messages to the child SPs, an SP that receives a message also has to check whether any of its clients' coherency requirements are violated. If so it would update the results of the query submitted by those clients. In this chapter, we assume that clients are pre-allocated to certain SPs, and focus on the construction of dissemination trees composed only by the SPs and the source. Henceforth we would use "SP" and "node" interchangeably and would only consider the dissemination within the dissemination trees.

Following [75, 76], we adopt the notion of *fidelity* as a measure of the performance of a dissemination system. Informally, the fidelity on a data object at a node during an observation period is defined as the percentage of time that the data value at that node conforms to the coherency requirement. To build our cost model, we formulate this metric in a formal way as follows. Let the value of a data object o_x at time t at the source and a node n_i be $o_x(s,t)$ and $o_x(n_i,t)$ respectively, and the coherency requirement of n_i on o_x be $cr_{i,x}$. Then the fidelity of n_i on data object o_x at time t is defined as:

$$f(n_i, o_x, t) = \begin{cases} 1 & : \quad DIV(o_x(s,t), o_x(n_i,t)) < cr_{i,x} \\ 0 & : \quad DIV(o_x(s,t), o_x(n_i,t)) \geq cr_{i,x} \end{cases} \tag{4.1}$$

And the fidelity of n_i on o_x during the observation period $[t1, t2]$ can be computed as

$$F(n_i, o_x, t1, t2) = \frac{\int_{t1}^{t2} f(n_i, o_x, t)}{t2 - t1}.$$

If our observation period is the whole life of the system, it can be rewritten as

$F(n_i, o_x)$. Furthermore, the average fidelity at node n_i is computed as

$$F(n_i) = \frac{1}{|O_i|} \sum_{\forall o_x \in O_i} F(n_i, o_x).$$

The *loss of fidelity (LF)* is defined as the complement of fidelity, which is $LF(n_i) = 1 - F(n_i)$. Our objective is to minimize the average loss of fidelity over all nodes

$$AvgLF = \frac{1}{|N|} \sum_{i=1}^{|N|} LF(n_i).$$

Since the loss of fidelity is due to the delay of the messages, we adopt an eager approach: the source node continuously pushes update messages to child SPs as soon as the corresponding coherency requirements are violated, and each SP, upon receiving any update messages, also pushes the necessary ones to its children as soon as violations occur.

We define the *Min-AvgLF problem* formally as follows: *Given a source s, a set of data objects O, a set of SPs N, and the set of requesting data objects O_i of each SP n_i as well as the coherency requirement $cr_{i,x}$ of n_i on each $o_x \in O_i$, construct/adapt one or more dissemination trees T to minimize the average loss of fidelity (AvgLF) of the system.*

By the celebrated Cayley's theorem, the number of spanning tree of a complete graph is $|V|^{|V|-1}$, where $|V|$ is the number of nodes in the graph. This means that brute-force searching is prohibitive even for a moderate number of nodes (e.g. 16 nodes). Even worse, a more restrictive problem is already NP-Hard [18].

4.1.2 Motivations

In view of the complexity of the problem, existing approaches such as DiTA [75] adopt two heuristics: (a) the coherency requirement of a parent node is at least as stringent as its children; (b) Each node has an apriori constraint on the fanout, i.e., the maximum number of child nodes is predetermined. However, under these restrictions, the resulting dissemination tree would be far from optimal. This is because they only explore a limited solution space and ignore the differences of the nodes in their capabilities as well as their communication delays. For example, although a node has a slow CPU, a long distance from the source, a low bandwidth or a high workload, it would still be put at the upper level of the tree as long as its coherency requirement is relatively stringent. However, all its descendants would suffer from the long processing delay in the slow node or the long transmission delay. This would result in severe loss of fidelity. Furthermore, multiple runs of trial and error is required to obtain an optimal fanout constraint. This may impede the deployment of the system. To handle these limits and find out the trade-offs, we believe a cost-based approach that captures both communication and processing cost is likely to lead to a more cost-effective dissemination tree.

Yet another challenge is that the optimality of a dissemination scheme depends on the current system parameters (such as data arrival rates, system workloads etc.). However, in a large scale distributed system, this information is hard to estimate or collect beforehand. Moreover, these parameters would fluctuate over time. For example, users would change their coherency requirements; a SP's workload would change as the number of clients connected to it are increased or decreased; or the message rate of each SP would also change due to the fluctuation of the data values. Since the dissemination system runs continuously, it can experience these changes at runtime, which would make the previously optimal scheme sub-

optimal. The problem of adapting to inaccurate statistics and system changes has been extensively explored in other problems such as query processing [10, 58]. Unfortunately, few efforts have been devoted to adapting the structure of a dissemination tree at runtime. Moreover, a decentralized scheme is highly preferable due to scalability and reliability problems.

4.2 Single Object Dissemination

In this section, we look at the scheme to construct a tree T to disseminate a single data object. We note that T is a spanning tree of the overlay graph G. We first present the cost model to evaluate the LF of a tree T, then describe the runtime adaptation scheme and finally, present the two static tree construction schemes. All the algorithms proposed do not place any restriction on the maximum fanout allowed; neither do they require the internal nodes to be more stringent in the coherency requirements than its child nodes.

4.2.1 Cost Model

In a cost-based approach, a cost function is used to evaluate the goodness of a potential solution. In our case, we propose a novel cost model to measure the LF of a dissemination tree. In the cost model, we make the following assumptions and simplifications:

1. A message sent from n_i to n_j incurs a communication delay, whose expected value is denoted as $d(n_i, n_j)$.

2. The messages received by a node are processed in a FIFO manner. Upon receiving a message, n_i would check every child to see whether the message should be disseminated to it. The processing order of the children is assumed to be

random. Let the time to perform the filtering be t_i^p and the time to perform the transmission be t_i^c. t_i^c includes the time to package the message and the time to send out the packages. The latter part is inversely proportional to the available bandwidth of n_i.

3. Each node would assign a portion of its resources (e.g. CPU, bandwidth, etc.) to perform the task of disseminating data to its child nodes. This portion of resources might be adjusted periodically. However, within each period, we assume it is fixed. Furthermore, the workload of a node is defined as the fraction of time that the node is busy.

Given these assumptions, now let us see how to estimate the loss of fidelity of a node n_i. The LF of n_i arises because of the delay of an update message. If the number of messages per unit time (i.e., the average message arrival rate) for n_i is r_i and the average delay of each update message is D_i, then the average LF of n_i is $LF(n_i) = r_i \cdot D_i$. r_i is related to the data characteristics and the coherency requirement of n_i. Now we need to estimate D_i. At a closer look, D_i includes the communication delay in all the links and the processing delay in all the nodes along the path from the root to n_i. To compute the communication delay, we define $D(n_j, n_i)$ as the communication delay from n_j to n_i in the dissemination tree T. It is obvious that $D(n_j, n_i)$ is the sum of the communication delay of the overlay edges in the unique path from n_j to n_i. Hence the total communication delay of a message from s to n_i is $D(s, n_i)$. In the following paragraphs, we would present how to estimate the second part of the delay: the processing delay.

The processing delay of a message for n_i in each of its ancestor n_k can be divided into the queuing time and the processing time. Let us estimate them one by one.

1. Queueing time. In our model, each node is a queuing system. From basic

queuing theory [50], the expected queuing time of a message in a M/M/1 system is equal to $\frac{\rho}{1-\rho}t$ where ρ is the workload of the system and t is the expected processing time of a message. The workload of the system is equal to the message arrival rate times the expected per-message processing time t. Hence to estimate the queuing time, we have to estimate the expected per-message processing time. Note that our tree construction scheme does not require the coherency requirement of a parent node to be more stringent than that of its descendant nodes. Thus, every node has an effective coherency requirement cr_i^m, which should be the most stringent cr within the subtree rooted at n_i. Consequently, there is an effective message arrival rate r_i^m for n_i, which should be equal to the maximum message arrival rate within the subtree rooted at n_i. For each message arrived at a node n_k, the probability that it is sent to a child n_j is r_j^m/r_k^m. Hence the expected processing time of a message in n_k for each of its children n_j is

$$t_{kj} = t_k^p + t_k^c \frac{r_j^m}{r_k^m}. \tag{4.2}$$

Therefore, if we denote the set of child nodes of n_k as C_k, then the expected processing time of a message in n_k can be estimated as:

$$t_k = \sum_{n_j \in C_k} t_{kj}. \tag{4.3}$$

Given t_k, the average processing time of a message, we can derive that the workload of n_k is $\rho_k = r_k^m t_k$. Hence the queuing time of a message in node n_k is $\frac{\rho_k}{1-\rho_k}t_k$. Note that this covers both the queuing times for processing and transferring a message.

2. Processing time in n_k for a message received by n_j. Since the children are processed in random order, before checking a child node n_j, there are on average $(|C_k| - 1)/2$ other children that have been processed. The expected length of this

time is equal to $(1/2)(t_k - t_{kj})$. Then it takes t_k^p time to check for n_j and then takes another t_k^c time to transmit the message to n_j. This means that the expected processing time in n_k for a message received by n_j is $(1/2)(t_k - t_{kj}) + t_k^p + t_k^c$.

Summing up the queuing time and the processing time, we can derive the processing delay in n_k for a message received by n_j as

$$g(n_k, n_j) = \frac{1+\rho_k}{2(1-\rho_k)} t_k + t_k^p + t_k^c - \frac{1}{2} t_{kj}. \tag{4.4}$$

This function can accurately estimate the processing delay. However, it distinguishes the delays for different children, which will bring higher cost in our algorithm. Hence we propose an approximation, where we use the average processing delay over all the children, to approximate the delay for each of them. We can derive, with simple calculations, that this processing delay is

$$\begin{aligned} g(n_k) &= \frac{1}{|C_k|} \sum_{n_j \in C_k} g(n_k, n_j) \\ &= \frac{1+\rho_k}{2(1-\rho_k)} t_k + t_k^p + t_k^c - \frac{1}{2|C_k|} t_k \end{aligned} \tag{4.5}$$

Now, we would derive the cost function to estimate the loss of fidelity for a node n_i as

$$\begin{aligned} LF(n_i) &= r_i \times [D(s, n_i) + g(p(n_i)) + g(p(p(n_i))) \\ &\quad + \cdots + g(s)] \end{aligned} \tag{4.6}$$

where $p(n_i)$ denotes the parent of n_i.

4.2.2 Adaptive Reorganization of Dissemination Tree

In this subsection, we present our runtime scheme that adaptively reorganizes a given dissemination tree to a more cost-effective one. The algorithm is a distributed local search scheme. At each state, distributed nodes would search the neighbor states that can improve the current state. Neighbouring states are generated based on a set of transformation rules . In the following subsections, we first present the local transformation rules that specify how the states could be transformed and how to estimate the benefit of the transformations. Then we present how to efficiently make adaptation decisions. Finally we summarize the set of information that has to be collected at runtime to support the adaptive scheme and present how to extend the cost model to incorporate the adaptation cost.

Local Transformation Rules

In this section we define several local transformation rules that transform a scheme into its neighbor schemes. We have identified six rules.

1. **Node Promotion**: Promote a node n_i to its parent's sibling. All the nodes in the sub-tree rooted at n_i are also moved along with n_i. Figure 4.1(a) shows an example of this transformation. In the example, n_i is promoted to a sibling of its previous parent n_j. This transformation might be beneficial, for example, when the workload of n_k is reduced as a result of a decrease in the number of its clients and hence more of its resources are assigned to the dissemination task. Promoting n_i can reduce the communication delay of messages sent to n_i and all its descendants if $d(n_k, n_j) + d(n_j, n_i) > d(n_k, n_i)$. This would also be helpful if we underestimate the capacity of n_k when building the initial dissemination tree.

2. **Node Demotion**: Demote a node n_i to a child of one of its siblings. The

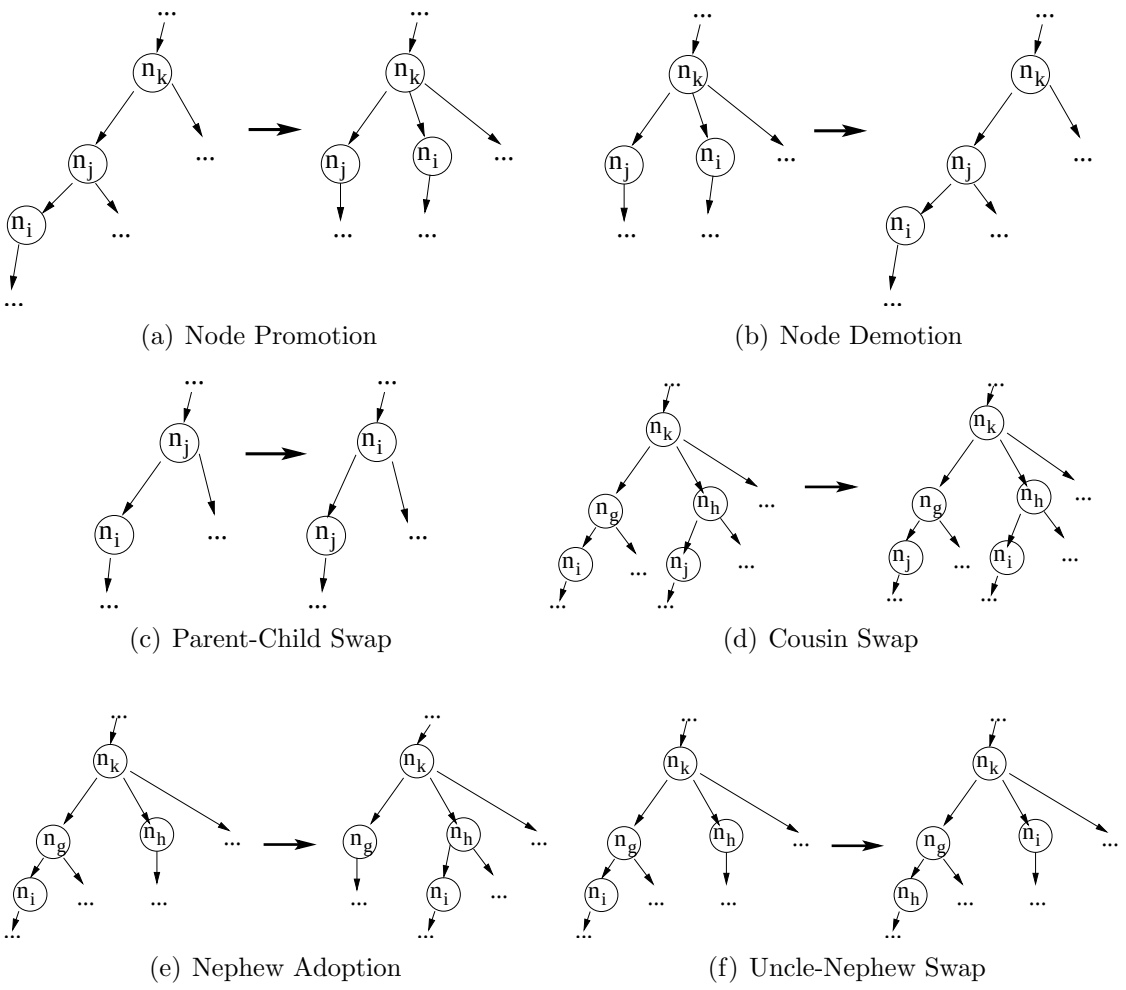


Figure 4.1: Local Transformation Rules

children of n_i would also be moved along with n_i. In the example shown in Figure 4.1(b), n_i is demoted to the child of its prior sibling n_j. This transformation may be beneficial, for example, when n_k's workload is increased and hence less resources are assigned to the dissemination task. Demoting n_i can reduce the dissemination load of n_k and hence reduce the processing delay of messages to be sent to the descendants of n_k. In addition, it also helps to handle any overestimation of the capacity of n_k in the initial tree building.

3. **Parent-Child Swap**: Swap the positions of n_i and its parent. Again all their other children would be brought along with them. In Figure 4.1(c), the

positions of n_i and its parent n_j are swapped.

4. **Cousin Swap**: Swap the position of two nodes n_i and n_j which have the same grandparent n_k. Their original children would still be connected with them. Figure 4.1(d) shows an example.

5. **Nephew Adoption:** A node n_h adopts its *nephew* n_i and adds it as its own child. As shown in Figure 4.1(e), n_i's grandparent is the parent of n_h. In this transformation, n_i is added as a child of n_h. The children of n_i are moved along with it.

6. **Uncle-Nephew Swap:** Swap the positions of n_h with its nephew n_i. Again, their children are moved along with them. Figure 4.1(f) depicts an example.

Actually the first two basic transformation rules are complete, i.e. any other transformations can be composed based on these two transformations. For example, Nephew Adoption can be composed by first promoting n_i and then demoting it to a child of n_h. However, using composite transformations directly may help avoid being stuck in a local optimum. The four composite transformations presented above are proposed based on this intuition. While the composite transformations can be extended to involve arbitrary nodes, we only consider these transformations to keep the runtime adaptation scheme relatively simple and less costly (the computation complexity is limited to $O(C^2)$, where C is the largest fanout in the tree). In addition, as shown in our experiments, it performs close to a centralized randomized algorithm that considers random tree transformations. So employing more complicated transformations would incur much more overhead without significant performance gain.

Based on our cost model, we can recompute the cost of the dissemination tree after the transformations, which will take $O(|N|)$ time. But since the transforma-

tions only affect part of the tree, rather than computing the cost from scratch, we can compute the change of the cost in *constant time*. Here we would use Node Promotion to illustrate.

As depicted in Figure 4.1(a), node n_i is to be promoted, and n_j and n_k are the parent and grandparent of n_i respectively prior to the transformation. After the transformation, the messages to be sent to n_i would no longer experience the transmission delays $d(n_k, n_j)$ and $d(n_j, n_i)$, and the processing delay in n_j. However it would experience the new transmission delay $d(n_k, n_i)$. This would also affect all the nodes below n_i. Hence this results in the change of $AvgLF$ which is

$$\begin{aligned} \Delta AvgLF_1 &= \frac{1}{|N|} r_i^c [d(n_k, n_i) - d(n_k, n_j) \\ &\quad - d(n_j, n_i) - g(n_j)], \end{aligned}$$

where r_i^c is the aggregated message rate over all nodes in the subtree T_i rooted at n_i, i.e. $r_i^c = \sum_{n_p \in T_i} r_p$. Furthermore, the load in n_k and n_j would be changed after the transformation. Hence all the nodes below them would experience the change of the cost due to the load changes. This results in the change of $AvgLF$ which is

$$\begin{aligned} \Delta AvgLF_2 &= \frac{1}{|N|} \left\{ (r_j^c - r_j)[g'(n_j) - g(n_j)] \right. \\ &\quad \left. + (r_k^c - r_k)[g'(n_k) - g(n_k)] \right\}, \end{aligned}$$

where $g'(n_j)$ and $g'(n_k)$ denote the estimated new processing delay in n_j and n_k respectively if the transformation is to have taken place. $\Delta AvgLF$ is equal to the sum of $\Delta AvgLF_1$ and $\Delta AvgLF_2$. Other transformations can be analyzed similarly.

Adaptation of Dissemination Tree

The adaptation scheme works as follows: periodically, compute the benefit (i.e., $(-1)\cdot\Delta AvgLF$) of each possible transformation, and then perform those that have positive benefits. To implement this procedure, there are several choices. In one extreme, we can select a SP to act as a centralized controller to make the adaptation decisions. However, as discussed, this approach suffers from problems of scalability and reliability. In another extreme, we can design a totally distributed approach. In this approach, each node makes the decisions independently and asynchronously. Each node would keep track of all its possible transformations, such as promoting/demoting itself, swapping with its child/parent/nephew/uncle, etc. However, this totally unstructured scheme would result in (a) Conflicting decisions being made by different nodes, e.g., n_i may determine to promote itself and meanwhile its parent may want to swap with it. Extra mechanisms have to be employed to resolve this problem, potentially increasing the complexity of such a scheme. (b) Wastage of computational resources as a result of multiple nodes arriving at the same decisions, e.g., n_i and its parent may determine to swap with each other at the same time.

To alleviate these problems, we propose a more structured mechanism. The adaptation operates in rounds. The root node initiates each round by creating a token. Only when a node holds a token, could it make an adaptation attempt. Algorithm 4.1 presents the operations to be executed in a node that receives a token. Each node receives a token can make its own decision independently without any synchronization with the other nodes. Instead of allowing every node attempts to try all kinds of transformations, we restrict each node to consider only the transformations involving its children and grandchildren. These include promoting a grandchild (node promotion), demoting a child (node demotion), swapping

a child and a grandchild (parent-child swap and uncle-nephew swap), swapping two grandchildren (cousin swapping), and moving a grandchild from one child to another child (nephew adoption). A node sends reorganization requests (if any) to the involved descendants, e.g. n_j in both Fig. 4.1(a) and (b), n_i and n_j in Fig. 4.1(c), n_g and n_h in both Fig. 4.1(d) and (e), n_g in Fig. 4.1(f). After the adaptation (if any) has been carried out, a copy of the token is sent to each of its non-leaf children. The next round of adaptation would be initiated by the root node if the adaptation interval is exceeded. If a node receives a token when it is still doing an adaptation, it would just ignore the token. Furthermore, if a node receives a reorganization request when it is already holding a token, then it would also ignore the reorganization request to avoid any contradictions.

Algorithm 4.1: AdaptationAttempt

```
1  begin
2      maxBenefit ← 0; t ← NULL;
3      for each possible transformations t1 involving the children and
       grandchildren do
4          if maxBenefit < Benefit(t1) then
5              maxBenefit ← Benefit(t1);
6              t ← t1;
7      if t ≠ NULL then Perform t;
8      for each child n_j do
9          if n_j is not a leave node then
10             Send one copy of the token to n_j;
11 end
```

In the midst of a tree transformation, data are disseminated through the old path. After the new connections are created, the old connections are dropped and the dissemination is transferred to the new connections.

Information Collection

Given the adaptation scheme described above, we now look at what information should be collected at runtime. Since each node would only consider transformations involving its children and grandchildren, it would collect state information from its children and grand-children. Hence a node contains at most the information of $O(C^2)$ nodes, where C is the maximum out-degree of all nodes. The information to be collected has to enable us to calculate the benefit of the transformations. Specifically, the information stored in a node n_i is as follows:

1. The overlay paths from n_i to its children and grand-children. This information is collected only once and need not be collected again at runtime. This is because any change in the structure in this part is determined by n_i itself and n_i updates the information itself.
2. The values of r_j^m, r_j^c, as well as t_j^c and t_j^p of each of its children and its grand-children.
3. The value of r_i^c. Actually, r_i^c can be computed based on the r_j^c value stored in each child node n_j, i.e. $r_i^c = (\sum_{n_j \in C_i} r_j^c) + r_i$.
4. The physical communication delay between n_i and each of its children or grand-children, and those between each of its children and each of its grand-children.

The information collection scheme is also a window-based scheme. Each node asynchronously maintains its own information collection window. At the end of each window, a node would measure the necessary information. If it detects that the new value is increased to $(1+\tau)$ times or decreased to $1/(1+\tau)$ times of its previous value, it would send the new value to its parent. In our experiments, we set τ to be 0.2.

Modeling the Adaptation Cost

The adaptation scheme incurs runtime overhead, which includes the cost of information collection and decision making and depends on the fanout of the nodes. To keep the adaptation cost low, there are two approaches: (1) extend the cost model to reflect the adaptation cost so that the tree construction would inherently restrict the fanout; (2) adopt a coarser-grained cost model when fanout increases. We study the first approach in this chapter and defer the second one as our future work. Let the set of grandchildren of n_k be GC_k. Assume the time spent by n_k to collect information for one node be t_k^e and the time to consider each possible decision be t_k^d. Furthermore, the length of the information collection and decision making period be T_e and T_d respectively. Then by computing the number of nodes to collect information and the number of possible decisions to be considered, we can compute the amortized adaptation cost t_k^a.

$$\begin{aligned} t_k^a &= \frac{t_k^e}{T_e} \cdot (|C_k| + |GC_k|) + \frac{t_k^d}{T_d} \cdot (|GC_k| \\ &\quad + |C_k|^2 + \sum_{n_j \in C_k} |C_j| \cdot (|GC_k| - |C_j|) \\ &\quad + 2 \sum_{n_j \in C_k} (|GC_k| - |C_j|)) \end{aligned} \tag{4.7}$$

This cost can also be computed in constant time by storing and performing incremental updates of some of the intermediate values. t_k^a is added to $g(n_k)$ to extend our cost model to factor in the adaptation overhead.

4.2.3 Static Tree Construction Algorithms

In this subsection, we present two static tree construction algorithms: a greedy algorithm and a randomized algorithm based on Simulated Annealing[49]. Given

apriori statistics on the system parameters, the two algorithms can generate a good dissemination tree. Such a tree can be used in environments that are static and not subject to runtime changes. For a highly dynamic environment, the algorithms provide a good initial scheme (as compared to a randomly generated dissemination tree) that can speed up the convergence to the optimal scheme as dissemination trees are refined adaptively based on the runtime characteristics.

Greedy Algorithm

The algorithm is presented in Algorithm 4.2. It adopts a greedy heuristic. The algorithm sorts the nodes in ascending order of $d(s, n_i) + t_i^p + t_i^c$. Then it adds the nodes into the dissemination tree one by one in the sorted order. The partially built dissemination tree T is represented as the set of nodes and edges in the tree. For each new node $N[i]$, it selects one node n_j within the partially built tree to act as the parent of $N[i]$ so that the average loss of fidelity $AvgLF$ of the new tree $T \cup \{N[i], e(n_j, n_i)\}$ is minimized. The estimation of $AvgLF$ is based on Equations (4.3), (4.5) and (4.6). To save the computational time, simple techniques can be employed to compute the new $AvgLF$ value incrementally based on the current $AvgLF$ of the partial tree. For brevity, we do not present the details here. Given each potential parent, it takes $\log |N|$ time to estimate the new $AvgLF$. Therefore, the computational complexity of Algorithm 4.2 is $O(|N|^2 \log |N|)$.

The dissemination tree built by using this algorithm has the following property:

Theorem 4.1 *If the height of the tree is h, and the delay between pairs of nodes satisfy the triangle inequality*[1]*, then the communication delay of a message received*

[1]If every non-leaf node has at least 2 children, then $h \leq \log |N|$. In addition, some studies [80] have shown that violations of triangle inequality is not very frequent, which is only about $1.4\% \sim 6.7\%$.

Algorithm 4.2: Greedy

```
1 begin
2     Add s to T;
3     N[0] ← s;
4     N[1 ··· |N| − 1] ← Sort the other nodes in ascending order of value
      d(s, n_i) + t_i^p + t_i^c;
5     for i = 1; i < |N|; i++ do
6         e ← arg min_{0≤j<i} AvgLF(T ∪ {N[i], e(n_j, n_i)});
7         Add N[i] and e to T;
8     return T;
9 end
```

by n_i is at most $2d_i \cdot h$ where $d_i = d(s, n_i) + t_i^p + t_i^c$. Further assume that the fanout of each node is at most C and the maximum message rate over all nodes is at most r, then the processing delay of a message received by n_i is at most

$$h \cdot (\frac{1 + r \cdot C \cdot d_i}{2(1 - r \cdot C \cdot d_i)} C \cdot d_i + d_i)$$

Proof: Let us first look at the worst case communication delay of the messages sent to a node n_i. Because of the triangle inequality, when n_i is added to T, the transfer delay $d(n_k, n_i)$ between the parent n_k and n_i is less than $d(s, n_k) + d(s, n_i)$. Because the nodes are added to T in ascending order of d_i, we can get $d(s, n_k) + t_k^p + t_k^c < d(s, n_i) + t_i^p + t_i^c$ and hence $d(s, n_k) < d(s, n_i) + t_i^p + t_i^c$, i.e. $d(s, n_k) < d_i$. We can obtain the following expression:

$$\left.\begin{array}{rcl} d(n_k, n_i) & < & d(s, n_k) + d(s, n_i) \\ d(s, n_k) & < & d_i \\ d(s, n_i) & < & d_i \end{array}\right\} \Rightarrow d(n_k, n_i) < 2d_i$$

We can also derive that the transfer delay of each edge in the path from the root to n_i is at most $2d_i$. Because the height of the tree is at most h, then the number of edges in the path from the root to n_i is at most h. That means the

worst case communication delay for n_i is $2d_i \cdot h$.

Now we look at the worst case processing delay of messages sent to n_i. Since $t_k^p + t_k^c < d_i$, we have $t_k < C(t_k^p + t_k^c) < C \cdot d_i$ (from Equations (4.3) and (4.2)). Furthermore, from Equation (4.4) we have the following:

$$\begin{aligned} g(n_k, n_i) &= \frac{1 + r_k^m \cdot t_k}{2(1 - r_k^m \cdot t_k)} t_k + t_k^p + t_k^c - \frac{1}{2} t_{ki} \\ &< \frac{1 + r \cdot C \cdot d_i}{2(1 - r \cdot C \cdot d_i)} C \cdot d_i + d_i \end{aligned} \tag{4.8}$$

This is the worst case processing delay in the parent n_k. Since for any ancestor n_j, $t_j^p + t_j^c < d_i$ is also true, Inequality (4.8) is also applicable to n_j. Again, the number of ancestors of n_i is at most h. Hence we can derive that the worst case total processing delay of a message sent to n_i is at most h times the worst case processing delay in each ancestor of n_i. □

Simulated Annealing

Since the Min-AvgLF problem is NP-Hard, we use a probabilistic approach, Simulated Annealing[49](SA), to approximate an optimal solution. This approach has been shown to generate very efficient solutions for hard problems, such as large join query optimizations [47]. The algorithm is illustrated in Algorithm 4.3. It starts from a random scheme S_0 and an initial temperature T_0. In the inner loop, a new scheme $newS$ is chosen randomly from the neighbors of the current scheme S. If the cost of $newS$ is smaller than that of S, the transition will happen. Otherwise, the transition will take place with probability of $e^{-\Delta C/T}$. (With the decrease of T this probability would be reduced.) Meanwhile, it also records the minimum-cost scheme that has been visited. Whenever it exits the inner loop, the current temperature would be reduced. Based on our experimental tuning and

past experiences[48, 47], we select the parameters as follows: (1) T_0: $2 * cost(S_0)$; (2) $frozen$: $T < 0.001$ and $minS$ unchanged for 10 iterations; (3) *equilibrium*: $64 \times \#nodes$; (4) $reduceTemp$: $T \leftarrow 0.95T$; (5) $RandomNeighbor$: randomly select one node and move its subtree to another random node.The cost of the new scheme can be computed similar to the incremental cost computation presented in Section 4.2.2. Given a static environment and accurate system parameters, we believe this algorithm can derive the best dissemination scheme over all the other algorithms. However, its optimization overhead may be high. Moreover, such a centralized scheme will incur too large a communication overhead in a dynamic context.

Algorithm 4.3: Simulated Annealing

```
1  begin
2      S ← S_0; T ← T_0; minS ← S;
3      while !frozen do
4          while !equilibrium do
5              newS ← RandomNeighbor(S);
6              ΔC ← cost(newS) − cost(S);
7              if ΔC ≤ 0 then S ← newS;
8              else S ← newS with probability e^{−ΔC/T};
9              if cost(S) < cost(minS) then minS ← S;
10         T ← reduceTemp(T);
11     return minS;
12 end
```

4.3 Multi-Object Dissemination

In the above discussion, we only consider single object dissemination. To disseminate multiple objects, there are two possible solutions: (a) the single-tree approach (to build one tree for multiple data objects), and (b) the multi-tree approach (to build one dissemination tree for each data object). In the following subsections,

we will look into these two approaches in detail.

4.3.1 The Single-Tree Approach

In the single-tree approach, a single dissemination tree T is built to disseminate a set of objects. Note that if an object of interest to a child is not requested by the parent itself, the parent's requesting object set would be enlarged to include this object. Hence there is an effective object set O_i^m for a node n_i which is the union of all the interesting objects of the nodes in the subtree rooted at n_i. In this section, we first develop the cost model for this approach, and then present the dissemination tree construction scheme.

Cost Model

The derivation process is similar to the single object case, except that we have to deal with more than one object. The delay of a message for a node n_i can still be divided into two parts: the transmission delay and the processing delay in the path from the root to n_i. The transmission delay is the same as the single object case which is $D(s, n_i)$. Before estimating the processing delay of a message in each node, we extend some of the above notations as follows. The message arrival rate of n_k from object o_x is $r_{k,x}$ and its corresponding effective update arrival rate is $r_{k,x}^m$. The sum of $r_{k,x}^m$ over all objects is denoted as $r_k^m = \sum_{o_x \in O_k^m} r_{k,x}^m$. We assume the expected per-child filtering time and the transmission time for a message in n_k is equal over all of the objects, which are still denoted as t_k^p and t_k^c respectively.

Now we are ready to derive the cost function of the processing delay. Recall that the delay is equal to the sum of the queuing time and the processing time. For a message from object o_x, the expected processing time in n_k for a child n_j

interested in o_x is

$$t_{kj,x} = t_k^p + t_k^c \frac{r_{j,x}^m}{r_{k,x}^m}.$$

Hence the total processing time of a message from object o_x should be

$$t_{k,x} = \sum_{n_j \in C_{k,x}} t_{kj,x}.$$

The average processing time of a message from all the objects in O_k^m is

$$t_k = \frac{\sum_{o_x \in O_k^m} r_{k,x}^m t_{k,x}}{r_k^m}.$$

Then the workload of n_k can be computed as $\rho_k = r_k^m \cdot t_k$. Therefore, the expected queuing time of a message should be $\frac{\rho_k}{1-\rho_k} t_k$. Similar to the analysis in the single object case, the message received by a child n_j has to experience an average processing time of $\frac{1}{2}(t_{k,x} - t_{kj,x}) + t_k^c + t_k^p$. Summing up the queuing time and the processing time, we have the expected processing delay in n_k of a message for one of its child n_j on object o_x:

$$g(n_k, n_j, o_x) = \frac{\rho_k}{1-\rho_k} t_k + \frac{1}{2}(t_{k,x} - t_{kj,x}) + t_k^c + t_k^p. \tag{4.9}$$

In Equation (4.9), the cost function distinguishes the processing cost on different objects. That means if the number of objects is large, the computational cost of our algorithm would be very large. Therefore, we provide an approximation on the cost model as follows. First, we approximate $t_{kj,x}$ for all values of x by using

$$t_{kj} = \frac{\sum_{o_x \in O_j^m} r_{k,x}^m t_{kj,x}}{r_k^m}.$$

Then we use t_k to approximate $t_{k,x}$. In this way, we can approximate Equation

(4.9) as follows:

$$\begin{aligned} g(n_k, n_j) &= \frac{\rho_k}{1-\rho_k} t_k + \frac{1}{2}(t_k - t_{kj}) + t_k^c + t_k^p \\ &= \frac{1+\rho_k}{2(1-\rho_k)} t_k + t_k^c + t_k^p - \frac{1}{2} t_{kj} \end{aligned} \tag{4.10}$$

Note that this equation is of the same form as Equation (4.4) in the single object cost model. Similar to the approximation we have done in the single object case, which uses the average processing delay over all the children to approximate that of every child of n_k, we have:

$$g(n_k) = \frac{1+\rho_k}{2(1-\rho_k)} t_k + t_k^c + t_k^p - \frac{1}{2|C_k|} t_k \tag{4.11}$$

Hence we can calculate the expected LF of n_i on object o_x, $LF(n_i, o_k)$ and then the expected LF of n_i averaging over all its interesting objects, $LF(n_i)$.

$$\begin{aligned} LF(n_i) &= \frac{1}{|O_i|} \sum_{o_x \in O_i} LF(n_i, o_x) \\ &= u_i [D(s, n_i) + g(p(n_i)) + g(p(p(n_i))) \\ &\quad + \cdots + g(s)] \end{aligned} \tag{4.12}$$

where

$$u_i = \frac{\sum_{o_x \in O_i} r_{i,x}}{|O_i|}.$$

Furthermore, the adaptation cost can be incorporated by adding $t_k^a \cdot |O_k^m|$ to $g(n_k)$, where t_k^a can be computed using Eq. (4.7).

Dissemination Tree Construction

As in the single object case, we also design an adaptive scheme and a static scheme. For the adaptive scheme, the transformation rules as well as the adaptation mechanism are also the same as the single object case. However, we need to extend the information collection strategy to include the new information that are required by the new cost model. More specifically, in the list in Section 4.2.2, the 1st and 4th points remain unchanged, while the 2nd and 3rd points are revised as follows:

- The values of O_j^m, $u_j^c, r_{j,x}^m, t_j^p$ and t_j^c of each of its children or grandchildren n_j for each object o_x in n_j's effective object set O_j^m.
- The value of u_i^c of node n_i, where u_i^c aggregated u_j values of all the nodes in the subtree T_i rooted at n_i, i.e. $u_i^c = \sum_{n_j \in T_i} u_i$.

Both the Greedy and SA Algorithm can be used here by employing the new cost model. The complexity of Algorithm 4.2 becomes $O(|O| \cdot |N|^2 \cdot \log |N|)$. Theorem 4.1 can also be applied to this scheme. Note that, in this case, the parameter r in the theorem should be the sum of the maximum message rate among all the nodes for each data object.

4.3.2 The Multi-Tree Approach

In this approach, one dissemination tree is created for each data object, which is similar to DiTA. Each tree only covers those servers that are interested in the corresponding data object. By doing so, update messages of an object will not be routed through the uninterested nodes.

The operations in each node is similar to the single-tree approach. When an update message arrives, the node checks the children that are involved and forward the message if necessary. Therefore, the cost model is similar to the single-tree approach.

Furthermore, we can perform the adaptive transformation of each tree independently and concurrently. Unfortunately, these trees are not independent. Two trees are correlated through those nodes that appear in both of them. Hence the change of one tree may affect the other trees through their common nodes. In particular, when a node n_i is making its adaptation decision for a tree, one of its children n_j may be performing the adaptation in another tree. Hence n_i's decision may not be based on the right information. Simply sequencing the transformation of the trees would slow down the adaptation.

Algorithm 4.4: Process Message

```
1  begin
2      while true do
3          wait for a new message msg;
4          HandleMsg(msg);
5          if state = IDLE||WAIT then
6              for each wait ∈ Q_ready do
7                  wait ← Q_ready.Dequeue();
8                  PerformAdapt(wait.tree);
9                  send a token message to each node in Child[wait.tree];
10                 continue;
11             for each msg in Q_token do
12                 remove msg from Q_token;
13                 HandleMsg(msg);
14             for each msg in Q_hold do
15                 remove msg from Q_hold;
16                 HandleMsg(msg);
17                 if state = HOLD then break;
18 end
```

To solve this problem, extra mechanism has to be incorporated. In our scheme, each node has three possible states: IDLE, WAIT and HOLD. As in the single-tree approach, the root node of each tree generates the token which is passed around the tree in a top-down manner. Each node that receives the token, before making the adaptation decision, sends out a "hold" message to all its children and enters

Algorithm 4.5: Helper Functions

```
1  Function HandleMsg(msg)
2  begin
3      switch msg.type do
4          case HOLD_MSG
5              if state = HOLD then Q_hold.Enqueue(msg);
6              else if state = WAIT then
7                  if msg.num > NUM then
8                      PerformHold(msg.tree) ;  /* deadlock prevention */
9                  else Q_hold.Enqueue(msg);
10             else if state = IDLE then
11                 PerformHold(msg.tree)
12         case TOKEN_MSG
13             if state = HOLD then
14                 if msg.tree = hold.tree then state ← WAIT;      /* this
                   token unlocks the hold state */
15                 else Q_token.Enqueue(msg);       /* put it in the token
                   queue */
16             if state = IDLE||WAIT then
17                 if ∃wait, wait.tree = msg.tree then break;  /* ignore this
                   msg */
18                 create a new object wait and put it into waitPool;
19                 wait.tree ← msg.tree ; /* initialize the wait object */
20                 wait.count ← Child[msg.tree].length;
21                 state ← WAIT;
22                 create a new hold message hmsg;
23                 hmsg.num ← NUM;
24                 send one copy of hmsg to each node in Child[msg.tree];
25         case ACK_MSG
26             Look up wait in waitPool s.t. wait.tree = msg.tree;
27             wait.count − −;
28             if wait.count = 0 then
29                 waitPool.Remove(wait);
30                 if state = WAIT||IDLE then
31                     PerformAdapt(wait.tree);
32                 else Q_ready.Enqueue(wait);
33 end
```

Algorithm 4.6: Helper Functions (Cont.)

```
1  Function PerformAdapt(tree)
2  begin
3      perform adaptation of tree;
4      if waitPool = φ then  state ← IDLE;
5      else state ← WAIT;
6  end
7  Function PerformHold(tree)
8  begin
9      send an ack message to msg.source;
10     hold.tree ← tree;
11     state ← HOLD;
12 end
```

the WAIT state. A child node that receives a hold message will reply with an acknowledgement message and enter the HOLD state when possible. The parent node that receives all the acknowledgements from its children, will perform the adaptation as usual if and only if it is not in the HOLD state. The details of this mechanism are presented in Algorithms 4.4,4.5 and 4.6.

Note that without careful considerations, the above algorithm may incur deadlock. Consider two nodes n_i and n_j. n_i is the parent of n_j in one tree while it is the child of n_j in another tree. It is possible that n_i and n_j will send a "hold" message to each other at about the same time. If they keep waiting for acknowledgement from each other, deadlock occurs. Furthermore, they should not both enter the HOLD state. To solve the deadlock problem, we assign a unique integer number NUM to each node, which is implemented by using the unique IP address of every node. When a node in the WAIT state receives a hold message, it enters the HOLD state only when its number is smaller than that of the hold message's origin. Lines 6 - 9 in Algorithm 4.5 implement this scheme.

Now let us analyze the effectiveness of our algorithm in solving the distributed deadlock problem. First, to model the problem, a directed graph, called a parent-

child graph (or P-C graph), can be generated, where a vertex represents a network node and a directed edge from n_i to n_j represents the fact that n_j is a child of n_i in at least one dissemination tree. Moreover, without any deadlock prevention scheme, a deadlock would happen if there is a cycle, $n_{i_1} \rightarrow n_{i_2} \rightarrow \cdots \rightarrow n_{i_p} \rightarrow n_{i_1}$, in the P-C graph and each node in the cycle is kept waiting for the acknowledgement from its immediate next node, i.e. n_{i_1} waits for n_{i_2}, n_{i_2} waits for n_{i_3} and so on. By using our proposed scheme, we have the following theorem:

Theorem 4.2 *The system is deadlock-free.*

Proof: Without loss of generality, assume there is a cycle $n_{i_1} \rightarrow n_{i_2} \rightarrow \cdots \rightarrow n_{i_p} \rightarrow n_{i_1}$ in the P-C graph. If a deadlock happens in this cycle, then $NUM_1 < NUM_2 < \cdots < NUM_p < NUM_1$ has to be satisfied, where NUM_j is the NUM value of node n_{i_j}. Otherwise, if say $NUM_1 > NUM_2$ (note that NUM_j is unique so $NUM_1 \neq NUM_2$), then, when n_{i_2} receives a hold message from n_{i_1}, n_{i_2} will enter the HOLD state and hence the dead lock will not happen. However $NUM_1 < NUM_2 < \cdots < NUM_p < NUM_1$ would not be true at anytime. Therefore, deadlock will not exist. □

In addition, when a node n_i is ready to perform adaptations, the workload statistics of a child n_j may have changed due to the adaptation of the other trees. In order to let n_i make decisions based on updated statistics, such statistics will piggyback onto the acknowledgement message sent to n_i. This includes the change of the number of n_j's children as well as the change of the update rates of its children.

4.4 Experiments

In this section, we present a performance study of the proposed techniques, and report our findings.

4.4.1 Experiment Configurations

The simulator is implemented using ns-2, a popular discrete-event simulator for networking research. The topology is generated using the GT-ITM topology generator. The Transit-Stub model, which resembles the Internet structure, is used. We generate a network topology with 1500 nodes, of which one node is chosen as the source, 256 nodes are selected as the SPs, and the remaining nodes act as routers. The average communication delay between any two SPs is about 20ms.

The expected filtering time and transmission time of each node is derived by using two respective uniform distributions. In our basic configuration, we set the average values of these times as 5ms and 1ms respectively (which may vary in our experiments), and set the minimum values as 1ms and 0.125ms respectively. The source node's expected filtering time and transmission time are always set to the minimum value to model an enterprise class server. Given the expected filtering time t_i^p and transmission time t_i^c for a node, the exact filtering time and transmission time of each message are drawn from two respective exponential random variable with expected values as t_i^p and t_i^c respectively. Recall that each SP in our system has to process local user queries (probably complex queries) and disseminate data to the child SPs, and only a limited resource can be allocated for the dissemination task. Hence we use a relatively long filtering time and transmission time which capture the load of processing user queries in the SPs.

In addition, the adaptation interval of our adaptive scheme is set to 200 seconds

and the information update window is set to 50 seconds. These values are chosen such that the system would not be over reactive to short term variances in our experimental setup. With higher data volumes, these intervals could be set shorter. We model the time used to transmit the statistical information to be the same as t_i^c. All the experiments are conducted in a Linux server with an Intel 2.8GHz CPU. We also implemented the optimization algorithms and the adaptation functions in C to study their performance. The adaptation overhead would be studied and modelled in the experiments.

To evaluate the performance of the proposed techniques, we compare them with the following approaches:

1. **DiTA**[75]. In DiTA, a tree is constructed for each data object. Fanout constraint is set for each node to avoid overloading. In our experiments, this is done by trial-and-error by repeatedly trying with different parameters and to pick the set that gives the optimal performance. (We find that this is the only way to find good fanout constraints and we believe this is a disadvantage of schemes relying on predetermined fanout constraints.) The nodes are added to the trees one by one. A node can serve another node only when its coherency requirement is at least as stringent as that of the other. A node n_i is added to each tree for each of its requesting data objects. Heuristics are applied to ensure that the level of n_i is as small as possible and secondarily the communication delay between n_i and its parent is also as small as possible. However, since DiTA is a distributed algorithm, these heuristics cannot guarantee the above objective. Hence we use a centralized version of DiTA which has the guarantees. Note that this is biased towards DiTA. It first sorts the nodes in ascending order of the values of their coherency requirements and then adds them one by one into the tree in the sorted order. When adding a node n_i, another node within the partial tree, which has

the smallest communication delay to n_i and still has available fanout degree, is selected to act as the parent of n_i.

2. **Source-Based Approach**. The distributed nodes do not cooperate and all the nodes are connected to the source. This provides a base line to evaluate all the schemes.

3. **Random Tree**. The nodes are added in random order. For each joining node, randomly select a node to act as its parent. This scheme provides a base line to evaluate all the tree-based schemes.

Furthermore, in the experiments, we use two types of datasets: synthetic data and real data. In the synthetic dataset, we set a specific expected message rate $r_{i,x}$ for each node on every object based on a uniform distribution. The source is of the largest $r_{i,x}$ for all the objects. Given the $r_{s,x}$ of the source, the interval of each update message is an exponential distributed variable with an average value of $1/r_{s,x}$. The synthetic data set provides relatively steady message rates, which offers opportunities for us to study the properties of the different algorithms. For the real dataset, we continuously poll stock traces from `http://finance.yahoo.com`. The polling is done in an interval of one second. In the experiments, we use 100 traces as our basic dataset which would be varied.

4.4.2 Adaptation Cost

In this section, we study the cost of performing adaptations using our C implementation. To examine the cost of making adaptation decisions, we use a node that serves 100 objects and try estimating 100 possible decisions. We found that $t_k^d \approx 0.6\mu s$ for both the single-tree and multi-tree approach. To keep the adaptation cost affordable, we have to set an appropriate adaptation period T_d. For example, if we can afford 5% of the CPU time for adaptation, we can set the

adaptation period of this testing node as shown in Figure 4.2. For example, if this node serves 10,000 objects, we have to set the adaptation period larger than or equal to 12 seconds. Therefore, to keep the adaptation responsive, the number of objects served by each node and the number of children and grandchildren should be kept to a certain limit. Note that constructing the tree using our extended cost model inherently consider this effect. The cost of collecting information is analyzed similarly. In the following experiments, we set both t_k^d and t_k^e as $1\mu s$ in the cost model and the simulation.

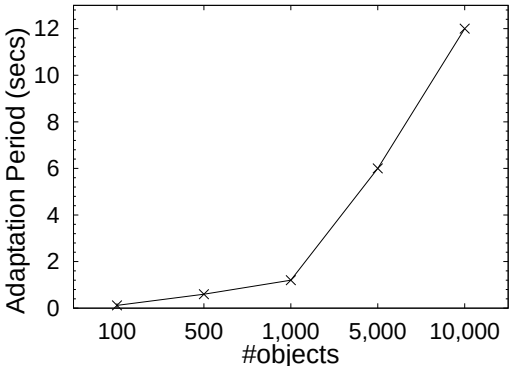
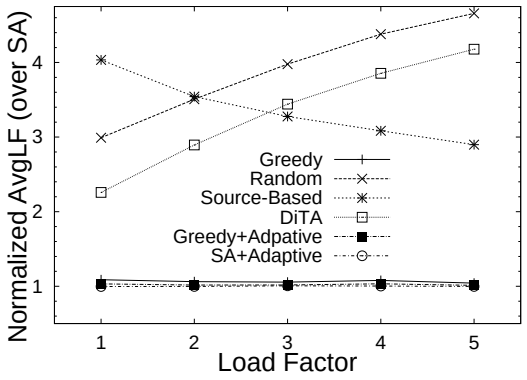


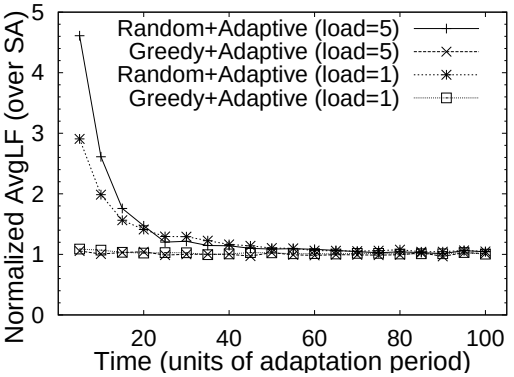
Figure 4.2: Adaptation period selection

4.4.3 Single Object Dissemination

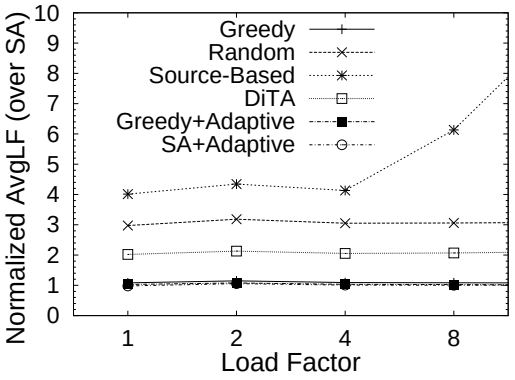
In this subsection, we examine the algorithms in a single object dissemination situation. We utilize the synthetic dataset. The expected message rate of each node is selected from a uniform distribution with the average value of 1 messages/second and a minimum value of 0.5 messages/second. (Note that the message rate models the coherency requirement at each node - a small coherency requirement implies a high message rate, and vice versa.)



(a) Sensitivity to processing time



(b) Online improvement



(c) Sensitivity to message rate

Figure 4.3: Performance on single object dissemination in static environment

Static Environment

In the first experiment, we vary the average filtering time and transmission time by multiplying them with a parameter *load*. The parameter *load* ranges from 1 to 5 in our simulation. The minimum values of filtering time and transmission time are not changed. This models two effects: (1) Various load conditions of the whole system. When more clients are connected or more queries are submitted to a node, its load would become higher and hence it takes a longer time to disseminate messages to its child nodes. The filtering and transmission times of these nodes would be increased. (2) Various degrees of heterogeneity of the system. With a

higher value of *load*, the filtering time and transmission time of the nodes would differ to a higher degree. No matter which is the case, nodes with higher filtering and transmission time would be deemed as less capable nodes and hence a good plan should be able to identify this kind of nodes and put them at a lower level of the dissemination tree. We run each algorithm for $20,000$ seconds and record the average $AvgLF$ over the whole simulation period as well as the values within every $1,000$ seconds time window. To ease the comparison, we normalize the $AvgLF$ values of all the other algorithms over that of the SA algorithm, which is (as expected) the best dissemination scheme.

Figure 4.3 shows the results of our experiment. From Figure 4.3(a), we can see that when $load = 1$, Greedy and the adaptive counter-part (Greedy + Adaptive) perform as well as SA, while the adaptive algorithm slightly improves over the initial scheme. Due to the optimality of SA, the adaptive scheme has few opportunities to further optimize the scheme. On the other hand, DiTA has more than two times $AvgLF$ than SA. That is because it can neither differentiate the capabilities of the different nodes nor utilize information of the communication delays between the nodes. The source-based algorithm performs the worst. In this scheme, all nodes are connected to the source node. Although the source node in our settings is not overloaded, the messages would still experience very long delay in the source node because of the high workload of the source. The random tree algorithm on the contrary scatters the workload randomly over all the nodes, and hence has a smaller $AvgLF$ value.

However, with the increase of the *load* parameter, we can see from Figure 4.3(a) that the relative performance of the source-based scheme improves. This is because, in our study, increasing the *load* parameter increases the processing time of all the nodes except the source node. Since the source-based approach dis-

seminates the messages directly from the source, it is not influenced by the *load* parameter. On the contrary, all the tree-based schemes would suffer from the increase of *load*. Furthermore, with the increase of *load*, DiTA and the random tree scheme become much worse while our static algorithms with/without adaptation scheme remains effective. This is because our scheme can identify the different capabilities of the nodes and reorganize them in a more cost-effective way.

Although our static schemes work well as shown above, they rely on accurate system statistics. To examine the performance of our adaptive mechanisms without these statistics, we use the random scheme to model an initial scheme that would be generated without accurate statistics. Figure 4.3(b) shows the result of this experiment. To ease viewing, we only depict the results of $load = 1$ and $load = 5$ for the Random+Adaptive and Greedy+Adaptive algorithms. The curves of the other *load* values would be between these two cases. It can be seen that when there are accurate system statistics, Greedy would result in a good dissemination scheme that works as well as SA. Hence there are not many opportunities for the adaptation scheme to improve. On the contrary, the random scheme works far worse than SA. Our adaptation algorithm iteratively improves this initial scheme. After about 30 adaptation periods, the random scheme has been improved from more than 3 and 4 to only 1.3 times of the performance of SA. And after more adaptation periods, the random scheme is improved to the extent that it performs as well as SA. This clearly shows the need for adaptive strategy, as well as the effectiveness of our adaptive scheme.

Another type of load change of the system is the change of message rates. With the increase of message rates, the dissemination load of the system is increased. In this experiment, we fix the processing time of each node to its basic value and multiply each node's basic message rate with the *load* parameter. The results are

depicted in Figure 4.3(c). With increasing message rate, Source-Based deteriorates rapidly. This is because with a high message rate, the workload of the source node largely increases due to its large number of children, and this incurs long queuing time for the messages in the source node. On the other hand, the relative performances of all the tree-based algorithms are not sensitive to message rate changes. This is due to the moderate number of child nodes in a tree-based scheme. Furthermore, our schemes steadily outperform the others under various message rates.

Dynamic Environment

In this subsection, we study our adaptive algorithm under a dynamic environment. In the experiments, we study how the algorithms perform when the workloads of the nodes are changed. The first experiment studies the single object dissemination schemes using the synthetic dataset. The parameters are set as in the first experiment in the last subsection where $load = 1$. Since Source-Based and Random have been shown to perform worse than the others in this situation, we only examine the results of the other algorithms. We run the system for $20,000$ seconds, and at the $10,000$th second, we increase the processing time of 10 nodes that are the first 10 nodes (except the source node) in a breadth-first search of the dissemination tree. These nodes are at the top of the dissemination tree. Their filtering time and transmission time are increased to 10 times of the previous values. This models the situation that the workloads of some nodes at the higher level of the tree increase as more clients are connected or more queries are submitted.

The result is depicted in Figure 4.4. In order to examine the optimality of the algorithms before and after the state transitions, we also executed two special runs of the SA algorithm: (a) Run the SA algorithm based on statistics before

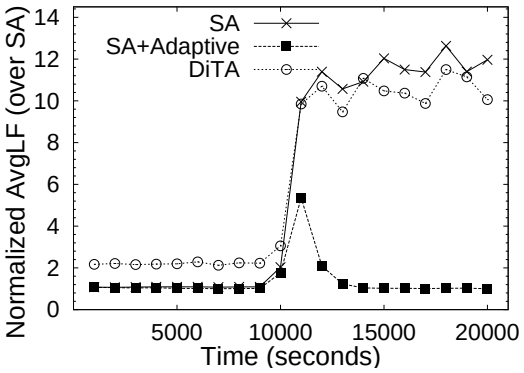


Figure 4.4: Performance on single object dissemination in dynamic environment

the change. Let the $AvgLF$ value of this run be SA1. (b) Run the SA algorithm based on statistics after the change. Let the $AvgLF$ value of this run be SA2. We then normalized the $AvgLF$ value of each algorithm under each condition by the corresponding $AvgLF$ of the SA algorithm. For example, consider the DiTA scheme. Let the $AvgLF$ be D. Then, before the change, its normalized value will be D/SA1, and after the change, its normalized value will be D/SA2. We compute the average of the normalized $AvgLF$ values over a $1,000$ seconds window and then report the 20 resulting values. In figure 4.4, one can see that at the first $10,000$ seconds, SA and SA+Adaptive perform as well as SA, while DiTA is two times worse than them. After the $10,000$th second, the $AvgLF$s of both DiTA and SA drastically increase. That is because the 10 nodes whose processing times are increased become the bottleneck of the whole dissemination tree. Furthermore because they are at the top of the tree, their processing delays dominate the delays of the messages sent to all their descendant nodes. On the other hand, our adaptive mechanism can detect this change and hence reorganize the dissemination tree to adapt to the new situation. Therefore, it only has a short term increase in the $AvgLF$ and then drops back to the original state. That is because the highly loaded nodes have been put to lower levels of the tree and

then their high processing times have little effect on the dissemination efficiency.

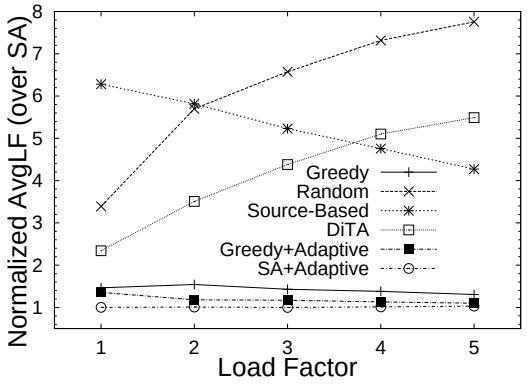
4.4.4 Multiple Object Dissemination

In the second set of experiments, we use our collected stock traces to examine the efficiency of our multiple object dissemination scheme. For each object, a probability that it is of interest to a node is set to 0.6, which will be varied in the experiments. The $cr_{i,x}$ values of each node n_i on each object o_x is chosen using a uniform random variable between 0.1 to 0.01. 100 traces are used as our basic configuration. For the ease of exposition, in the following experiments we first compare our single-tree approach with other approaches and then compare the single-tree approach with the multi-tree approach.

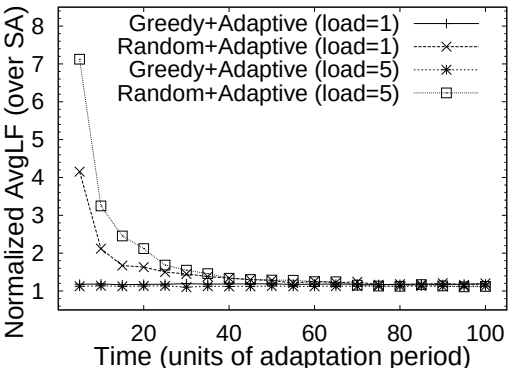
Single-Tree Approach

In the first experiment, we use a parameter *load* to vary the average filtering time and transmission time as we have done in the single object experiments. Figure 4.5(a) shows the results of this experiment. The relative performance of the algorithms is similar to the single object case. All our techniques perform as well as SA. Random and DiTA perform worse with larger *load* due to their inability to differentiate the capabilities of the various nodes. Source-Based is insensitive to the parameter *load*. Figure 4.5(b) again shows that our adaptive mechanism can improve a random tree, which models a tree built on inaccurate statistics, to perform as well as SA.

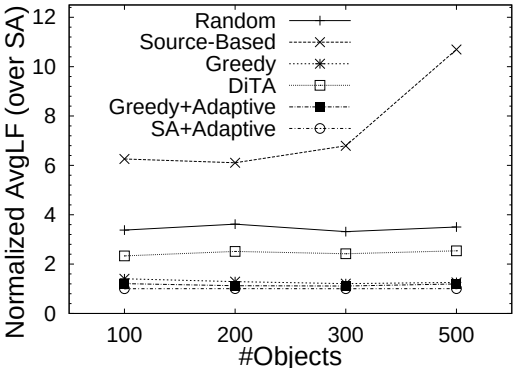
In another experiment, we examine the sensitivity of the algorithms to different number of data objects. We vary the number of data objects to be disseminated from 100 to 500. The results are depicted in Figure 4.5(c). With different number of data objects, Greedy, Greedy+Adaptive and SA+Adaptive persistently outper-



(a) Sensitivity to processing time



(b) Online improvement

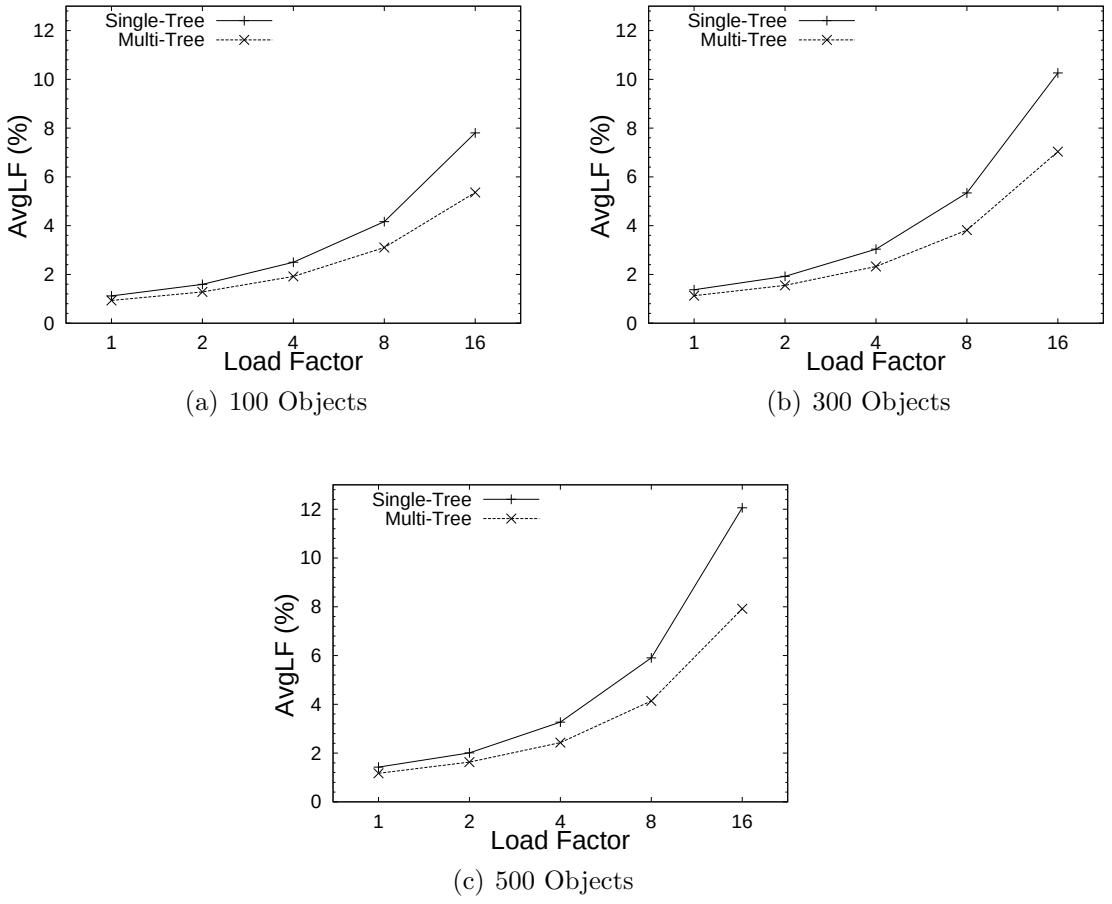


(c) Sensitivity to number of objects

Figure 4.5: Performance on multiple object dissemination

form all the other algorithms. We can also see that the relative performance of the Source-Based algorithm deteriorates with increasing number of data objects. This is because the source's workload largely increases with increasing number of data objects and hence its processing delay increases. Furthermore, the absolute values of the $AvgLF$s of all the other tree-based algorithms only increase by around 15% when the number of objects is increased from 100 to 500. However, for the $AvgLF$ of Source-Based, the increase is around 200%. This shows that the tree-based approaches have better scalability with respect to the number of objects.

Multi-Tree Approach



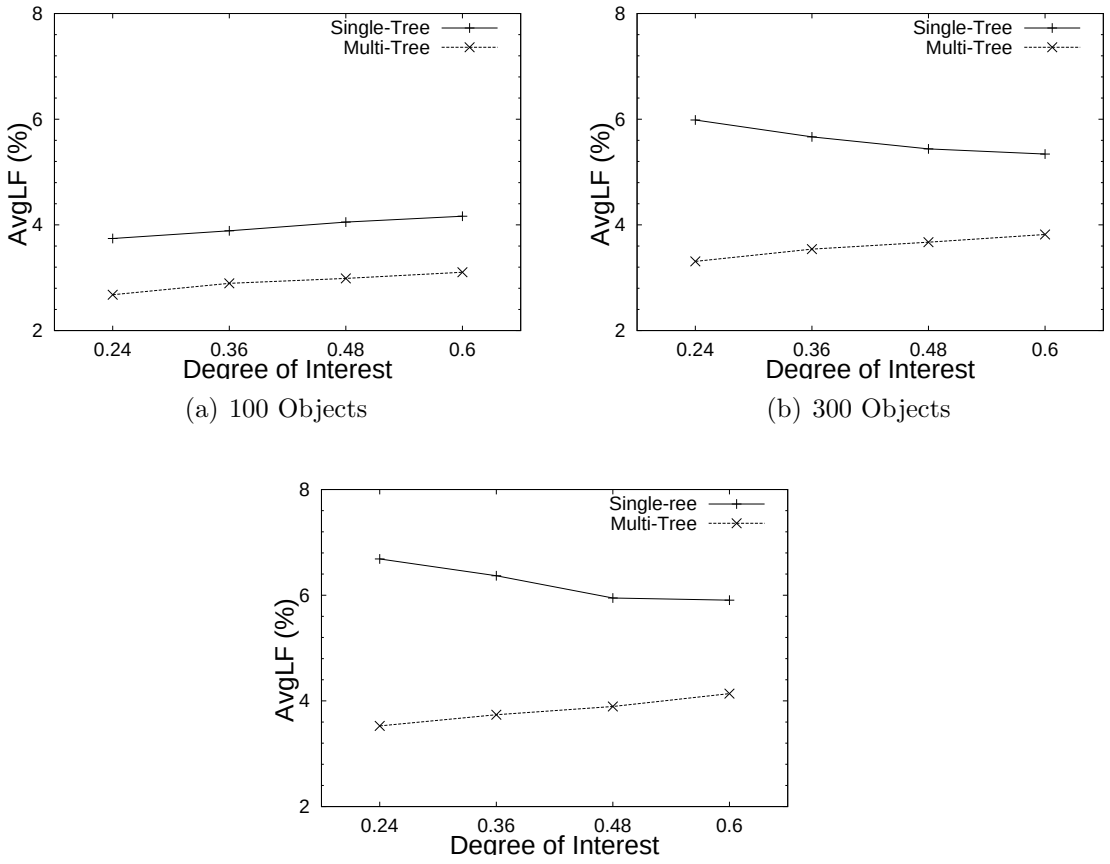
(a) 100 Objects

(b) 300 Objects

(c) 500 Objects

Figure 4.6: Sensitivity on system workload

Now we study our multi-tree approach. From the results in the previous experiments, it is clear that our proposed single-tree method is superior to other methods. Thus, we shall only compare our multi-tree approach against our proposed single-tree method. Furthermore, for conciseness, only the results of SA for both approaches are presented. In the first experiment, we use a parameter *load* to vary the average filtering time and transmission time of the nodes as we have done in Section 4.4.3 and 4.4.4. Figure 4.6 shows the result. It can be seen that the multi-tree approach outperforms the single-tree approach consistently.



(a) 100 Objects

(b) 300 Objects

(c) 500 Objects

Figure 4.7: Sensitivity on the number objects of interest to each node

Furthermore, with higher workload on the nodes, their performance difference is larger. This is because the update messages in the multi-tree approach are transferred through fewer number of nodes and this benefit is more obvious with larger workload on the nodes.

In the second experiment, we fix the *load* parameter at value 8. Instead, we vary the probability that a node is interested in an object for each pair of node and object. We refer to this probability as the degree of data interest. The smaller the degree of interest, the fewer are the number of objects of interest to each node. From the results shown in Figure 4.7, it is obvious that the multi-tree approach

consistently outperforms the single-tree approach. Moreover, when each node has a smaller number of interesting objects, we can achieve more benefit by using the multi-tree approach. The reason is the number of nodes in each individual dissemination tree is smaller and the update messages experience less processing delays in the nodes. This effect is more obvious with a larger number of objects.

Running Time of SA and Greedy

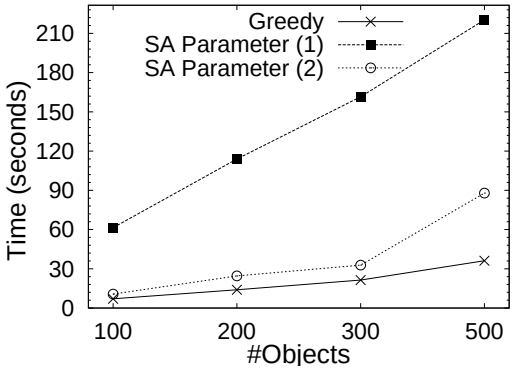


Figure 4.8: Running time of Greedy and SA

From the above experiment results, we can conclude that in a static environment, Greedy and SA perform the best among all the static algorithms given accurate statistics. In this experiment, we evaulate their running time. We use two sets of parameters of SA: (1) the parameters listed above and (2) changing $64 \times \#nodes$ to $16 \times \#nodes$ and $T < 0.001$ to $T < 0.0015$. Since the running time of the single-tree and the multi-tree approach is similar, only the results of the single-tree approach is presented here. Figure 4.8 shows the running time of both algorithms with different number of objects. Obviously, Greedy persistently outperforms SA in running time for both sets of parameters of SA. However, SA with parameters (2) comes with a plan whose cost is more than 2 times of that of Greedy. SA with parameters (1) can derive the best plan; however, the running

time is significantly increased. We also tested a lot of other parameters of SA and cannot find a case that SA outperforms Greedy both in runtime and tree cost. For a static environment, SA is superior to Greedy due to its ability and robustness to find a low cost scheme. However Greedy is more suitable for a dynamic environment, because it provides a cheaper way to construct a good initial tree and devoting more time to construct the initial tree does not make much sense as a previously optimal plan would become sub-optimal when the system state is changed. We can see this effect in Section 4.4.3 and the next section.

Dynamic Environment

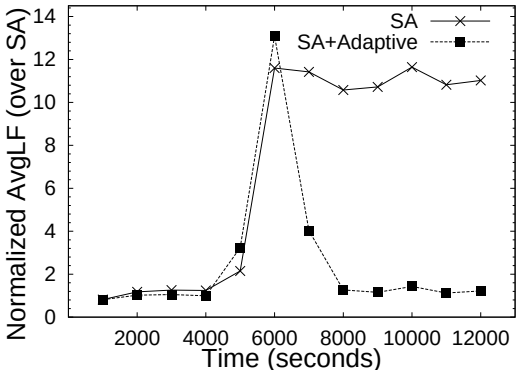


Figure 4.9: Performance on multi-object dissemination in dynamic environment

This experiment is similar to the one in Section 4.4.3, except that it is performed on multiple object dissemination. Since DiTA builds one tree for each object and DiTA has been shown above that it is not adaptable to system changes for any one of its dissemination trees, we only compare the SA and SA+Adaptive in this experiment. The other settings are similar to Section 4.4.3. At the $5,000$th second, we shift the filtering time and transmission time of 10 nodes, which are at the top of the dissemination tree, to 10 times of their original values. The result is reported in Figure 4.9. We can see that before the change, SA works

slightly worse than Adaptive. At the $5,000$th seconds, both SA and SA+Adaptive increase in their $AvgLF$s. However, our adaptive mechanism successfully detects the shift and then reorganizes the dissemination tree to adapt to the new situation. Hence SA+Adaptive restores back to its original state in terms of $AvgLF$ while the bad performance of SA persists. We also performed experiments on runtime change of transmission delays and coherency requirements. The results show that our adaptive scheme can also adapt to these changes and re-optimize the scheme incrementally.

4.5 Summary

In this chapter, we studied the problem of optimizing the overlay network at the data overlay of COSMOS. We proposed a cost-based approach to construct dissemination trees to minimize the average *loss of fidelity* of the system. Based on our cost model, a novel adaptation scheme was proposed and is experimentally shown to be able to adapt to inaccurate statistics and changes of system states. Two static algorithms: Greedy and SA, have also been proposed for relatively static environments and for constructing initial trees under dynamic environments. The Greedy algorithm is useful for dynamic environments due to its faster speed to build a relatively good initial tree, while SA is superior for static environments due to its robustness. Furthermore, the multi-tree approach was shown to be more robust to the number of objects, the degree of data interest as well as system workload.

Chapter 5

Adaptive Operator Ordering

Having seen the design of the inter-provider layer, we shall turn to the intra-provider layer in the following two chapters. We study the problem of operator ordering in this chapter and the problem of operator placement in the next one. In this chapter, we introduce a new highly adaptive distributed query processing mechanism, called SwAP (Scalable & Adaptable query Processor), that facilitates efficient adaptation of operator orders at runtime. The architecture can quickly detect fluctuations in selectivities of operations, as well as transmission speeds and workloads of servers, and accordingly change the operation order of a distributed query plan during execution. We have implemented a prototype based on the Telegraph system [64]. Experimental study shows that the proposed mechanism can adapt itself to the changes in the environment and hence approach to an optimal plan during execution.

The rest of this chapter is organized as follows. In the next section, we introduce the background and challenges. Details of the query execution mechanisms are given in Section 5.2. In Section 5.3, the scheme to construct a distributed query plan for multi-join queries is presented. Experimental results are given in Section 5.4.

5.1 Background and Challenges

In this section, we present the background of our work and discuss the challenges.

5.1.1 Background

Our work is based on several pieces of work of the Telegraph Project. Eddy [8] is an tuple routing operator interposed between data sources and query operators such as selections and joins. An eddy operator continuously pushes tuples into the queue of the query operators and the query operators may return the result tuples to the eddy operator. By adjusting the routing orders of tuples through operators under a tuple routing scheme, eddy is able to adaptively approach the optimal order of operations at runtime.

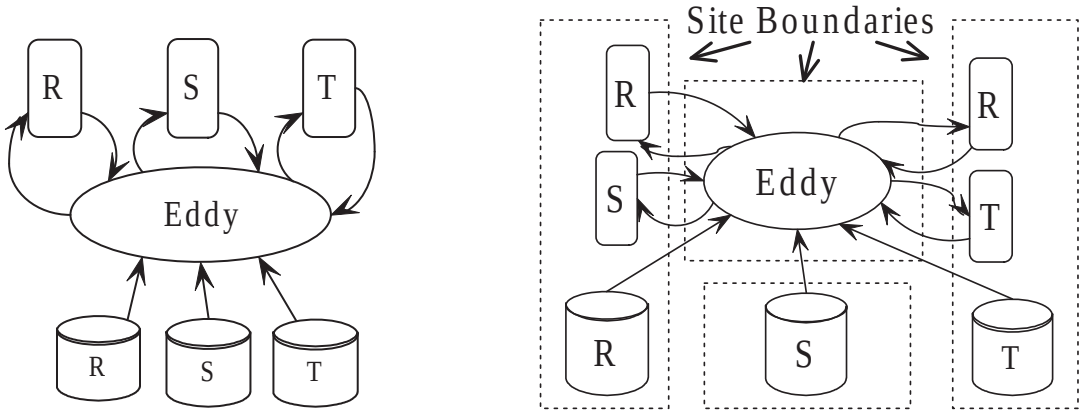
The authors introduced two tuple routing schemes, *back-pressure effect* and *lottery routing* scheme, that enable eddy to observe an operator's behavior (cost and selectivity) and accordingly route tuples through the operator in an order approaching the optimal plan. The idea of the back-pressure effect is as follows: operators of higher costs take more time to finish the processing of a tuple and hence they consume tuples more slowly than those of lower costs. This results in larger input queue sizes for high cost operators. Hence by fixing the lengths of the operators' input queues, the eddy operator is forced to route tuples to an operator of lower cost before routing to those of higher costs. Under the lottery routing scheme, each operator is assigned a number of tickets. An operator gets a ticket when a tuple is routed to it and looses a ticket when it returns a tuple to the eddy. Thus the number of tickets can be used to roughly estimate the selectivity of an operator. The more selective an operator is, the more tickets it holds. When two operators vie for a tuple, the operator with more tickets has higher probability

to "win" the tuple. By combining the back-pressure effect and lottery routing scheme, an eddy generally routes tuples to a faster and more selective operator before routing to those slower and less selective ones.

[66] extends eddies by splitting up a symmetric join operator [43] into two first order operators called SteMs. A SteM can be viewed as a half symmetric join operator. It is implemented as an indexed repository built on tuples from a base stream using a particular attribute. One SteM is created for each attribute of each base stream addressed in the join predicates. For example, a two-way equi-join can be evaluated using two SteM operators each is implemented as a hash table built on the joining attribute of the two base streams. Tuples arriving from each base stream are first built into their own SteM(s) and then used to probe the other streams' SteMs to get the join results. By exposing the normally hidden data structures (for example hash tables), SteMs enable eddies to have more control over the normally hidden physical operations: build and probe within a join algorithm. By probing SteMs in different orders, the join ordering, join algorithm and the spanning tree (for cyclic queries) can be adapted. SteM also provides a shared data structure for data from a given table, regardless of the number of access methods or join algorithms. This facilitates the access method adaptation by avoiding redundant work during competition between access methods.

Figure 5.1(a) is an example execution plan for a three-way join $R \bowtie S \bowtie T$. In this example, all the operations are located at a single site, while data sources could be remote data sources. The join operators are all equi-joins. In the figure, and all figures throughout this chapter, we use the rounded rectangles to denote SteMs. Tuples from the sources are routed through the Eddy operator to the processing operators. The internal data structure in the SteM operator is a hash table built on the joining attributes. We can see that the SteM of R is shared

by both join operations. Tuples of S are first inserted into the hash table of the SteM of S and then probe the hash tables of the other two SteMs. The order in which the other two SteMs are probed is determined based on the routing schemes stated above. Tuples of R and T are routed similarly.



(a) Single site processing (b) Multi-site processing

Figure 5.1: Centralized eddy.

5.1.2 Challenges

Unfortunately, the above techniques are not readily applicable to our context. In our intra-provider system, there are multiple locally distributed processors and hence the operators of a query may be executed in multiple processors. Moreover, there are two kinds of parallelism in a distributed query plan that can be exploited between processors: *horizontal* (or intra-operator) and *vertical* (or pipelined) parallelism. In horizontal parallelism, different processors are running independently on different partitions of data. In vertical parallelism, processors are running in a pipelined manner, i.e., results of operations running in one site may be piped to another site for further processing. It turns out that vertical parallelism offers greater opportunities for adaptivity.

Consider the following example, which we shall use as a *running example* throughout this chapter to illustrate the mechanisms of SwAP. Suppose we want to evaluate a three-way join $R \bowtie S \bowtie T$. Recall that we assign a delegation processor for each stream (Section 1.1.3. Therefore, R, S and T may be delegated to three different processors: Site 1, Site 2 and Site 3 respectively. Assume that the two join operations are to be evaluated at Site 1 and Site 3 respectively. Then we can form two different pipelined query plans each corresponding to a different join order. The first one is to route tuples of S to Site 1 to evaluate $R \bowtie S$ whose results are piped to Site 3 to join with T. This plan actually corresponds to executing the join operations in the order $(R \bowtie S) \bowtie T$. The other plan is to route the tuples of S to Site 3 first and then to pipe the results to Site 1. This corresponds to $(T \bowtie S) \bowtie R$. To determine the best plan to use is essentially the operation ordering problem. A good choice of the order should consider both the selectivities and costs of the distributed operations, as well as the network transmission speeds and workloads of the processors. We believe this is where traditional query optimization is inadequate as a static query plan that fixes the order in which tuples are routed would be unable to adapt to inaccurate estimations or runtime changes. Instead, we need robust query processing schemes that can dynamically adjust the join order on-the-fly based on characteristics of the queries and data, as well as the system resources at runtime.

A naive strategy that serves the purpose is to extend the mechanism of eddies by placing an eddy operator on a single processor and connecting all operators and data sources to the eddy no matter where they are located. The query plan for the three-way join query in our running example using this method is shown in Figure 5.1(b). The eddy is located on a server that could be one of the three data source sites or an alternative site. Then the eddy can dynamically reorder the

distributed operations. However, this centralized eddy architecture suffers from the problem of scalability, reliability and large communication overhead and the eddy operator can become a bottleneck during query processing. SwAP targets on solving the above problems.

5.2 Query Execution Mechanism

In SwAP, when a new query is submitted to a server, that server becomes the coordinator for that query. The coordinator site compiles the query, chooses the sites for processing the query and determines the degree of parallelism for the operations required by the query. Then, an algorithm (described in Section 5.3) is employed to generate the distributed query plan.

In this chapter, we assume the placement of operators is determined. At runtime, SwAP adapts the operation orders, join algorithms and access methods (as well as spanning trees for cyclic queries) on the fly. (The choices of join algorithms, access methods and spanning trees are done by using the adaptive capability provided by SteMs.)

In the following subsections, we assume that a query plan has been set up, and look at how the query plan is processed in SwAP. We shall consider both vertical parallelism and horizontal parallelism. We defer the discussion on how a distributed query plan can be generated to the next section.

5.2.1 Scheme for Vertical Parallelism

Given a particular layout of query operations, there are two kinds of parallelism between the processing sites: vertical parallelism and horizontal parallelism. We shall focus on vertical parallelism in this subsection and discuss horizontal paral-

lelism later. Under vertical parallelism, sites are running in a pipelined manner: the output of one site is piped to another site(s). A result tuple can only be output as an answer when it has gone through all sites. (A result tuple is said to have gone through a site if and only if at least one of its component tuples has gone through that site.)

An interesting problem here is that the output of one site may have the choice of being routed through other sites in different order. Recall our running example: $R \bowtie S \bowtie T$ mentioned in Section 5.1. The result tuples of Site 2 have two possible routing order, either Site 2 $\rightarrow$ Site 1 $\rightarrow$ Site 3 or Site 2 $\rightarrow$ Site 3 $\rightarrow$ Site 1, which correspond to the two join orders: $(R \bowtie S) \bowtie T$ and $(T \bowtie S) \bowtie R$. A good choice of the order should balance the workloads of servers while minimizing the cost of communication and other system resources. Instead of fixing this order, our scheme makes the routing decision at runtime and thus can potentially balance the workloads of servers, and minimize the communication cost and response time. Rather than using a centralized eddy operator as illustrated in Figure 5.1(b), our scheme employs multiple eddies - one at each site. The query plan for the sample query under our scheme is shown in Figure 5.2. There is one eddy operator at each processing site. The rounded squares are the SteMs used to evaluate the join operations. To choose a site to transmit the results of Site 2, the eddy operator in Site 2 continuously measures the selectivities of operations, as well as the transmission speeds and the workloads at Site 1 and Site 3. The exact mechanisms will be introduced in the following sub-subsections. Moreover, this is done in a distributed manner, i.e., each site is making the decision for the transmission of its own results.

To realize the above framework, we have identified three issues: (a) How does each eddy efficiently collect statistics from remote sites and make routing decisions

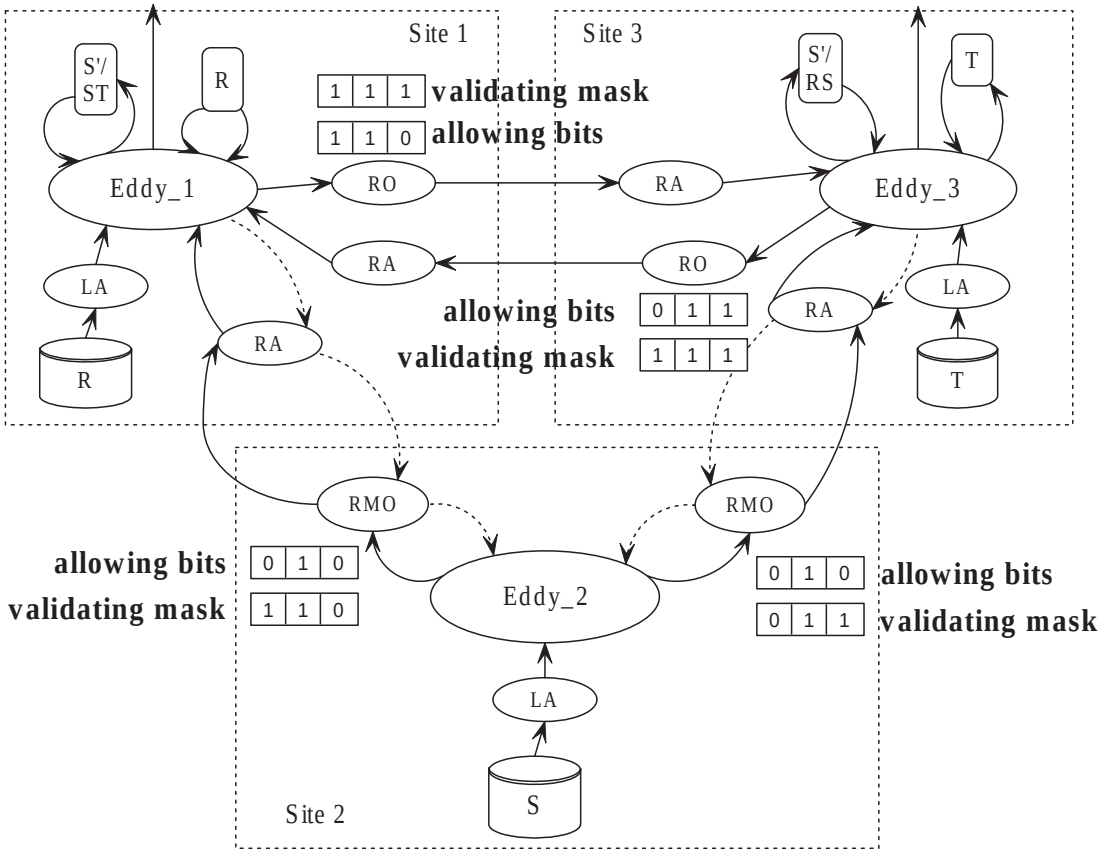


Figure 5.2: An example of the scheme for vertical parallelism. LA denotes local access operator; RA denotes remote access operator.

for its intermediate result tuples? (b) Since the routing order of tuples is not fixed, what kind of mechanisms should we use to facilitate the routing of tuples? (c) How does the system efficiently process the intermediate result tuples received from the remote sites? In the remaining part of this section, we shall present our solutions to these issues.

Collecting Statistics and Making Routing Decisions

In SwAP, we use a *Remote Output* (RO) operator to transmit intermediate results from a local site to a remote site for further processing. Each RO connects to an RA (Remote Access) operator at its corresponding remote site. An RO operator

continuously sends intermediate results of the local site to its corresponding RA operator. For example, in Figure 5.2, we attach an RO to Eddy_1 at Site 1, which transmits output tuples from Site 1 to its corresponding RA operator at Site 3.

However, when a site needs to choose the routing order of its results, such as Site 2 in the running example, we have to collect statistical information from the candidate remote sites so that we can make an appropriate routing decision. Therefore, in this situation, we extend the functionality of an RO operator such that it not only sends out local result tuples but also collects statistics from its corresponding remote site. This extended version of the RO operator is called the *Remote Meta-Operator* (RMO). In some sense, the RMO together with the statistical information it collects forms a *local abstraction* of the operations running at the remote site. For example, in Figure 5.2, Site 2 needs to choose between Site 1 and Site 3 to transmit its results. Therefore, we attach two RMOs to the eddy at Site 2. Based on the statistics collected by the RMOs, the local eddy can determine the RMO through which it should route its results.

Let us now look at the kind of statistics an RMO should collect from a remote site. As stated above, to make an appropriate decision, we have to know the selectivities, as well as the transmission speeds and workloads of the candidate remote sites. We can adapt to the latter two parameters by using the back pressure effect introduced in centralized eddy [8]. In other words, if a remote site's transmission speed is slow or its workload is high, then it consumes tuples very slowly so that its corresponding RMO also consumes tuples very slowly from the local eddy. Therefore, it forces the local eddy to route the result tuples to another RMO which connects to a faster remote site.

To estimate the first parameter, i.e., selectivity of a remote site, we have to know how many result tuples the remote site generates given the number of tuples

it receives from an RMO. Therefore, the remote site has to send this number back to its corresponding RMO. Our scheme works as follows. At a remote site, when its eddy routes a tuple to an output operator (RO or RMO), it will check whether the tuple contains any data fetched from an RA that is connected to an RMO. If so, it tells the RA that a result tuple has been generated. The RA operator accumulates the number of result tuples generated, and when the number reaches a threshold, it sends the *number* (as an integer) back to its corresponding RMO. The threshold is a tunable system parameter, which determines how sparingly the information is sent. There is clearly a trade-off here: the transmission overhead and the responsiveness of the system. The higher the threshold, the lower the transmission overhead, but the less responsive the system will be. Let us look at the example in Figure 5.2. When Eddy_1 routes a result tuple of $R \bowtie S$ to the RO operator, it will detect that the tuple contains data fetched from an RA which is connected to an RMO (the left RMO of Eddy_2). Then the eddy tells the RA module that a result tuple has been generated. The RA accumulates this number until it reaches a threshold, after which it sends this number to the corresponding RMO. The dotted curves in the figure indicate the flow of this information. Similar processing is performed at Site 3. Actually, we can compute the transmission overhead between two sites arose from this mechanism as

$$\frac{s \times t \times sizeof(int)}{threshold}\ bytes,$$

where s is the selectivity of the site receiving tuples from the other site, and t is the number of tuples sent to that site.

Now that RMO can collect statistical information, the eddy needs to be able to use this information to determine the correct join order. In this thesis, we adopt

the lottery routing scheme implemented in the original eddy implementation [8]. In the lottery routing scheme, query operators will return result tuples to the eddy so that eddy can calculate the tickets for them and hence make routing decision. In order for the lottery routing scheme to work in our context, an RMO mimics actions of other ordinary query operators by returning *Virtual Tuples* to its eddy. Virtual Tuples are not typical data tuples. In fact, they do not contain any data, i.e., have zero data length.[1] The number of Virtual Tuples an RMO returns to the eddy equals to the number it receives from the remote site (which corresponds to the number of result tuples the remote site generates). From an eddy's point of view, an RMO is like an ordinary query operator that continuously fetches tuples from the eddy and returns "tuples" to the eddy.[2] The virtual tuples are used to calculate tickets of the RMO in the lottery scheme. Therefore, under the lottery routing scheme, the number of tickets held by an RMO can be used to roughly estimate the selectivities of the operations of its corresponding remote site. For example, in Figure 5.2, the tickets held by the left RMO of Eddy_2 reflect the selectivities of the operations running at Site 1. In this sense, the RMO together with the Virtual Tuples it generates form a *local abstraction* of the operations running at the remote site. Therefore, by using the lottery routing scheme, we can adaptively make the decision on which site to transmit the intermediate results according to the selectivities of the candidate remote sites.

By combining the back pressure effect and the lottery routing scheme, we can generally route tuples to the site with lower selectivity, faster transmission speed and lighter workload first. (Note that this combination, unlike only using the

[1] In Telegraph, every tuple passing through operators is a message, and every message has a header indicating the message type. Thus, virtual tuples are tuple messages without the message body.

[2] The virtual tuples are sent in bulk to the eddy. Only one message is sent for a bundle of virtual tuples once the RMO receives a statistical message from the remote site.

lottery routing scheme, can distinguish between a slow site and a very selective site. Because a slow site consumes tuples very slowly while a very selective but fast site would consume tuples very fast.) The intuition is that a site with lower selectivity can eliminate larger number of irrelevant tuples, and a site with faster transmission speed and lighter workload can finish its processing earlier. Hence routing to this kind of sites first can speed up the processing. Moreover, all these decisions in our scheme are done in a distributed way, i.e. sites are making decisions for the transmission of their own results.

In addition, more routing strategies can be incorporated into our proposed processing architecture. For example, in [82], the authors proposed several routing strategies which also can be incorporated into our system by extending the RMO to collect more statistical information from the remote sites. Since our main contribution is building the new processing architecture, we do not consider the incorporation of them in this thesis.

Routing of Tuples

In the case of vertical parallelism, a final result tuple must have undergone the processing of all sites. To execute the query efficiently and effectively we have to avoid two types of routing of tuples: (1)*Redundant routing* and (2)*void routing*. Redundant routing occurs when tuples are routed through the same site more than once, while void routing occurs when tuples are routed to a site that have no operations over them, e.g. routing tuples of stream R to Site 3 for the example in Figure 5.2. Since no operation in Site 3 involves tuples of R, we refer to this kind of routing as void routing. To prevent these two kinds of routing, we have to know two pieces of information: a tuple's routing history (which sites the tuple has been routed through and which it has not) and the tuple's possible next "stations"

(given a tuple's routing history, which sites the tuple can be routed to).

To record a tuple's routing history, we associate with each tuple a bit vector called *Global Footprint*, whose length equals to the number of participating sites. Each bit in the Global Footprint corresponds to one participating site. Turning on a bit (setting it to 1) in the global footprint means the tuple has been routed through the corresponding site. Whenever a tuple has completed the local processing on one site, the corresponding bit in the global footprint is set. When joining two tuples, the global footprint of the new tuple is an OR result of the two global footprints. Note that there is an extra constraint on the routing of tuples. In order to minimize the communication overhead, tuples must have been fully processed by the local operations before being routed to an RMO/RO operator. Otherwise, the result tuples of the remote site have to be sent back for processing, which will significantly increase the communication overhead. In the example of Figure 5.2, each tuple is associated with a Global Footprint of three bits corresponding to the three processing sites. Consider an S tuple in Site 2, its Global Footprint will be set to [0,1,0] after it has been retrieved and is ready to be transmitted to Site 1 or Site 3.

To efficiently store and use the second piece of information mentioned above (the tuple's possible next "stations"), we attach to each RMO/RO a compact descriptor that contains two bit vectors: *AllowingBits* and *ValidatingMask*. These bit vectors are of the same length as the Global Footprint. The AllowingBits indicates which type (in terms of the Global Footprint) of tuples can be routed through the RMO/RO, and the ValidatingMask indicates which bits of the AllowingBits are valid. *Only if a tuple's global footprint matches the AllowingBits under the ValidatingMask of an RMO/RO can the tuple be routed through that RMO/RO.*

Let us use Figure 5.2 to illustrate. The bit vectors drawn around the RMO/ROs

are their AllowingBits and ValidatingMasks. The ith bit in these bit vectors corresponds to Site i. Consider the left RMO of Eddy_2, whose AllowingBits indicates that only those tuples that have been fully processed by Site 2 but have not been routed through Site 1 can be routed through this RMO. The corresponding ValidatingMask indicates that only the first and the second bits of the AllowingBits are valid. Now, the global footprint of an S tuple retrieved at Site 2 would be set to [0,1,0] once it is ready to be transmitted. This tuple can be routed through the left RMO since the first two bits of the AllowingBits (the ValidatingMask says that we only need to consider the first two bits) match the global footprint of the tuple.

As another example, consider the RO attached to Eddy_1. Result tuples of Site 1 can be routed through this RO only when they have been fully processed both by Site 1 and Site 2, and have not undergone Site 3. Hence we have to consider all the three bits of a tuple's global footprint (all the three bits of the ValidatingMask are turned on). Here, we have three possible scenarios: (a) Site 1 retrieves a local R tuple. In this case, upon retrieval, the R tuple initially has a global footprint of [0,0,0]. Since the valid bits of the AllowingBits do not match the global footprint of this tuple, it is prevented from being routed to Site 3 (which is what we wanted otherwise we will end up with void routing). (b) Site 1 receives an S tuple from Site 2. Initially, upon arrival, the global footprint of the tuple would be [0,1,0]. When there is a matching R tuple, the resultant tuple RS will have a global footprint of [0,1,0] (recall that we OR-ed the global footprints of joining tuples). Now, the eddy detects that this tuple has been fully processed at Site 1, so it turns on the first bit of RS's global footprint making it [1,1,0]. Based on the ValidatingMask and the AllowingBits, RS can now be routed to Site 3. (c) Site 1 receives an intermediate result tuple from Site 3. The initial global

Table 5.1: Storage overhead of the affiliated data structures

Structure	Size Expression
Global Footprint	$\frac{\#tuples \times \#sites}{8}$ bytes
AllowingBits	$\frac{\#sites \times (\#RMO + \#RO)}{8}$ bytes
ValidatingMask	$\frac{\#sites \times (\#RMO + \#RO)}{8}$ bytes

footprint of this tuple would be [0,1,1], which means it has been fully processed by Site 2 and Site 3 and has not been processed by Site 1. Then, if it matches a tuple from R the resultant tuple's global footprint would still be [0,1,1]. Since it has been fully processed at Site 1, eddy turns on the first bit of its global footprint resulting in a vector as [1,1,1]. Again, since it cannot match all the three bits of the RO's AllowingBits, it is prevented from being routed to Site 3 (to prevent the redundant routing). In fact, since the global footprint is [1,1,1], that means the tuple has been fully processed at all sites, and is an answer tuple that can be output to the user.

Table 5.1 summarizes the storage overhead of the above auxiliary data structures. In the table, $\#tuples$ denotes the number of tuples that have been loaded into the system and are still under processing. The number of RMO and RO at every site is at most $\#sites$, which is the number of the processing sites. Therefore the total number of these operators in all sites, $\#RMO + \#RO$, is at most $\#sites^2$. However, as we will see in Section 5.3 tuples should not be transmitted between some sites. A simple example is Site 1 in Figure 5.2 should not route its intermediate result tuples to Site 2. Therefore, $\#RMO + \#RO$ is often less than $\#sites^2$ in practice.

We shall defer the discussion on how these bit vectors are initialized to section 5.3 where we present the algorithm to generate the distributed query plan.

Processing of Intermediate Results

In the centralized processing framework of Eddies and SteMs, the processor does not store the intermediate join results. For example, in Figure 5.1(a), a possible routing order for tuples of R after they are inserted into their own SteM is to probe the SteM of S first and then the intermediate join tuples RS are used to probe the SteM of T. In this case, the intermediate join tuples RS are not stored in the processor. When a new tuple of T is received, it has to probe the other two SteMs to get the final result tuples. Instead, if the intermediate tuples RS were stored, then the new T tuple only needs to probe the intermediate tuples. The reasons why centralized Eddies and SteMs do not store the intermediate tuples are because (a) probing a hash table is a very fast operation and (b) storing the intermediate tuples may require a lot of memory.

However, in a distributed processing context, there are further considerations. Not storing the intermediate results from a remote site may incur a high communication overhead. For example, in Figure 5.2, if we do not store in Site 3 the intermediate result tuples RS from Site 1, then every new T tuple has to be sent to Site 1 to probe the two SteMs in Site 1. That is because even if the T tuple does not match the tuples in the S's SteM in Site 3, we cannot guarantee that the T tuple will not match the S and R tuples in Site 1. On the contrary, if we store the intermediate result tuples RS in Site 3, then we can just use the newly received T tuple to probe the RS and S tuples that are stored locally. Only those tuples that join with S need to be transmitted to Site 1 to probe the SteM of R. Therefore, we choose to store the intermediate result tuples in a processing site.

On the other hand, storing the intermediate results gives rise to another problem. In our scheme, the processing sites may receive different types of intermediate tuples. For example, in Figure 5.2, Site 1 receives two types of tuples, one is from

Site 2 and another is from Site 3. They are different because those from Site 2 are only tuples from Relation S while those from Site 3 are results of $S \bowtie T$. Hence there may be more than one "sub-query" running at a single site. For example, in Figure 5.2, there are actually two joins running at Site 1: $R \bowtie S$ and $R \bowtie ST$, where ST is the join results of a portion of stream S and stream T. Similarly, there are two sub-queries running at Site 3. Actually the number of sub-queries at one site is related to the number of possible global footprint of tuples routed through that site. To evaluate a number of sub-queries efficiently, one solution is to adopt the multi-query processing scheme proposed in CACQ [58]. Applying the techniques of CACQ in our scenario would result in creating a separate SteM for each type of intermediate result tuples. Hence each sub-query running at a site would require different operators, and this would result in overheads to maintain a lot of query information, which are necessary for queries with different operations. However, the situation in our scheme is different from that of CACQ, i.e., all these queries require the same operations, while queries in CACQ may require different operations!

To avoid the above unnecessary overhead, we adopt another approach. We use only one SteM for all types of tuples containing data from a particular stream involved in a join operation. For example, in Figure 5.2, we use only one SteM for both the intermediate tuples sent from Site 2 and Site 3 and tuples are built into the SteM using the same fields from the base stream S. In this way, all sub-queries require tuples to undergo the same operators and hence there is no need to maintain sub-query completion information and additional tuple routing information as is done in CACQ [58].

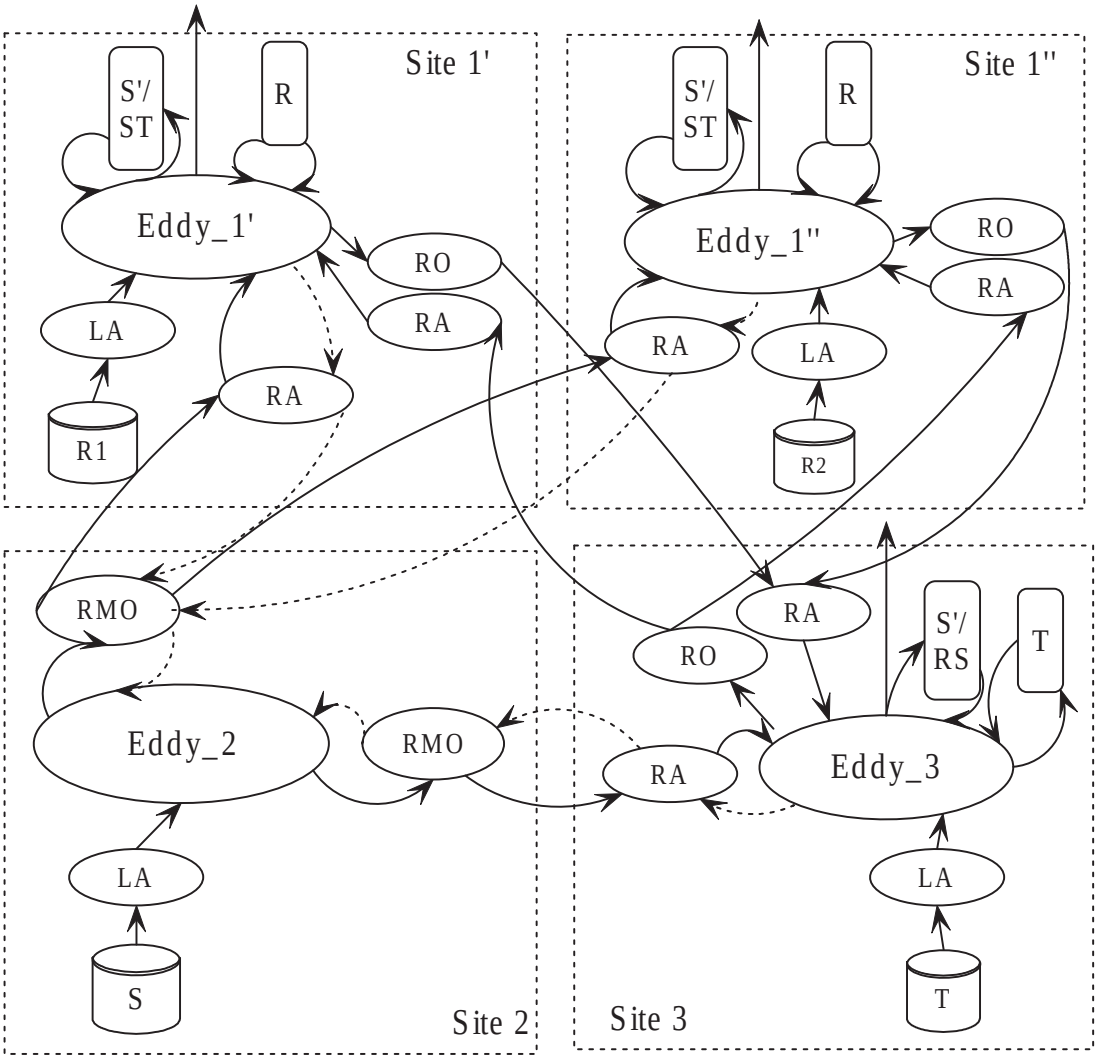


Figure 5.3: An example of scheme for horizontal parallelism. LA denotes local access operator; RA denotes remote access operator.

5.2.2 Scheme for Horizontal Parallelism

When one or some streams are composed by the union of multiple substreams which are delegated to multiple sites, it is natural to horizontally parallelize the operations. In this scheme, different sites exploit intra-operator parallelism to independently perform the same operation on different substreams. In the horizontally parallelization scheme of SwAP, each processing site has its own eddy to manage the required operators running there. Each eddy has the same number of operators but operating on different substreams. Complete results can be obtained by performing a union operation on the output of all the processing sites. The complete results may be further processed if they are only intermediate results or output to the user if they are the final answers. Since eddies between the horizontally parallelized sites are running independently to one another and provide adaptivity of operations running at their own sites, there is no need to introduce any extra mechanism.

For example, if stream R in our running example is fragmented onto two sites: Site $1'$ and Site $1''$. The execution scheme in SwAP is illustrated in Figure 5.3. The operators in Site 2 and Site 3 are not changed. The operators of Site 1 in Figure 5.2 are replicated onto Site $1'$ and Site $1''$. Site $1'$ and Site$1''$ are running in a horizontally parallelized manner. The two eddies in these two sites are running independently to each other.

An alert reader may note that now the left RMO of Eddy_2 (in Figure 5.3) corresponds to two remote sites rather than one remote site. It is reasonable since the mechanism of RMO in this example is actually used to choose the join order between $(R \bowtie S) \bowtie T$ and $(T \bowtie S) \bowtie R$. Therefore, there is no need for Eddy_2 to distinguish the two sites: Site $1'$ and Site $1''$. Now we have to refine the definition of RMO/RO and RA under the case of horizontal parallelism. Each RMO/RO

is connected to all the RA operators of the horizontally parallelized down stream partner, and vice versa. The RMO-RA (or RO-RA) pair encapsulates the distribution of operations, the corresponding communications/flow details as well as the collection of statistics of the remote sites. Hence it separates the distribution details from the local eddies and operators. This feature eases the development of our system. Few modifications need to be added into the existing centralized system. Furthermore, the RMO/RO would send tuples using the partition information of the partitioned streams. For example, if R is partitioned based on the join attribute, then the RMO connected to Site $1'$ and Site $1''$ would send a tuple from S based on its join attribute value to either Site $1'$ or Site $1''$ accordingly. Otherwise the tuple have to be sent to both sites.

5.2.3 Cyclic Queries

We note that the above discussions and examples only focus on acyclic queries. For cyclic queries, a traditional query optimizer will statically choose a spanning tree and only create join operations in the spanning tree. The remaining predicates are enforced by using selection operations. In [66], the author has addressed the issue of making the choice of spanning trees adaptive in a centralized processing environment. This is done by adaptively changing the order of tuples routed through SteMs. In a distributed processing context, if all the join operations involved in the cycle are to be evaluated at a single site, then the spanning tree can be adapted as in a centralized processing context. If these join operations are running across multiple sites, a spanning tree can only be chosen statically under the current scheme. (Note that we are not arguing against adapting the spanning trees. In fact, we plan to explore this as our future work.) Hereafter, we focus on acyclic queries.

5.3 Query Plan Generation for Multi-Join Queries

In the above discussion, we have assumed that a distributed join plan is available, and its operations have been set up on the various processing sites. We can also see that SwAP works on a distributed query plan that is different from a traditional plan. Since SwAP does not fix the tuple routing order through the processing sites, we have to find out all the candidate routing orders for the result tuples of each site and accordingly add the RMO-RA (or RO-RA) operators to transmit tuples between the sites. In this section, we will present a scheme to generate a distributed plan that supports both horizontal and vertical parallelism. Given a query, the plan generation is done by a preparatory phase in SwAP, which involves three steps. In the first step, the query is parsed into a query parse tree. Then a join graph [53] (JG), is generated. A JG is essentially an undirected graph where nodes represent streams and an edge exists between two streams when there is a join predicate between them. Figure 5.4(a) is an example of a JG. Here, we have 6 streams R1 - R6, and a join predicate exists between R1 and R2, R2 and R4, and so on.

In the second step, the optimizer selects the processing sites and the degree of parallelism of the operations. The end result is a distributed extension of the join graph, called *Distributed Join Graph* (DJG). Finally, in the last step, communication operators (i.e, RMO/RO-RA pairs) are added into the DJG to produce the distributed query plan.

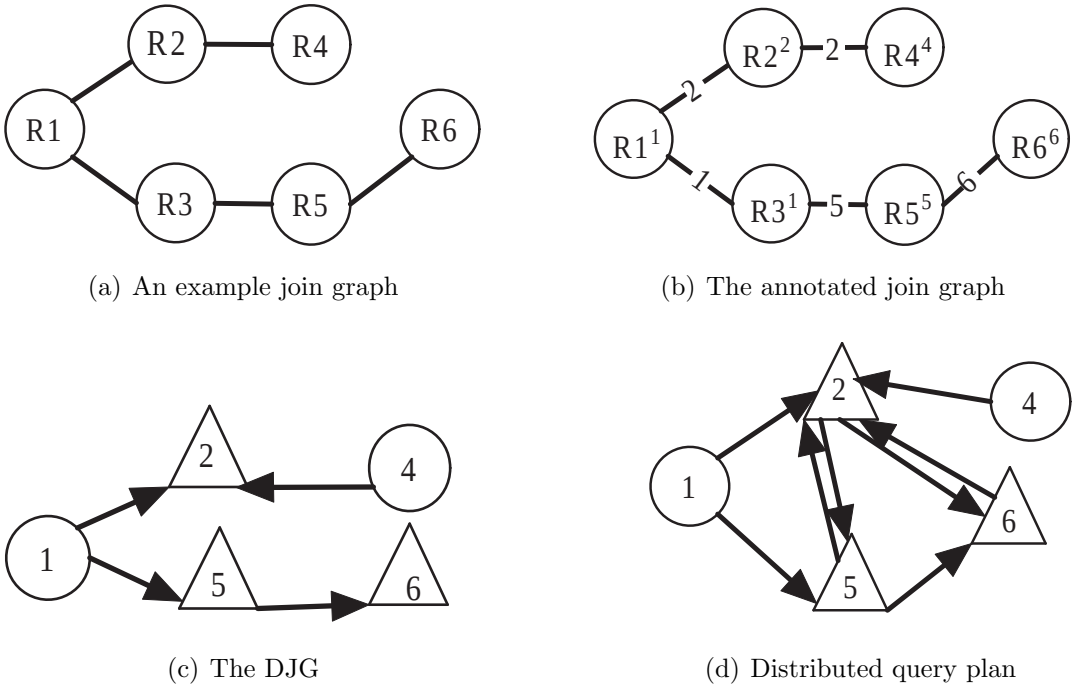
In the following subsections, we first introduce the DJG and its properties, and present the algorithm to generate the DJG from a JG. Then, we present the algorithm to incorporate the communication operators into the query plan.

5.3.1 Distributed Join Graph

In the second step of the preparatory phase, the optimizer first annotates the join graph (from the first step) to reflect the processing sites and the degree of parallelism of the operations. (Since we do not address operator placement problem in this chapter, we just use the simple strategy mentioned at the beginning of Section 5.2.) In the annotated join graph, nodes are labeled with the delegation sites of the corresponding streams, and edges are labeled with the processing sites. Note that more than one join operation can be assigned to a processing site. Figure 5.4(b) is an example annotated join graph. Superscripts of the stream names are the locations of the streams. In the figure, we have streams R1 and R3 being co-located at Site 1, and the join operation between these two streams is also to be performed at Site 1.

We shall refer to a *maximum* connected sub-graph in an annotated join graph as a *sub-query* if all the edges have the same label (i.e, the operations are to be performed at a single site). For example, in Figure 5.4(b), the subgraph involving streams $R1$ and $R3$ corresponds to a subquery; similarly, the subgraph involving streams $R1$, $R2$ and $R4$ also forms a subquery. We note that a sub-query can be a single node in a join graph, i.e. the sub-query is only a stream access, e.g., $R4$ in Figure 5.4(b). (For ease of presentation, we assume that each site processes exactly one sub-query. But in fact, we can treat multiple sub-queries running at one site separately when setting up the plan, and then run them separately at the same site.)

The annotated join graph is finally converted to a *Distributed Join Graph* (DJG). A DJG is an acyclic directed graph where nodes represent sub-queries and edges represent cooperation relations between the nodes. Figure 5.4(c) is the DJG transformed from the annotated join graph 5.4(b). We note that there are two



(a) An example join graph

(b) The annotated join graph

(c) The DJG

(d) Distributed query plan

Figure 5.4: Example join graph, distributed join graph and the distributed plan.

kinds of nodes in a DJG:

Self-contained Node A node is *self-contained* when all streams involved in the sub-query are co-located. The circular nodes in Figure 5.4(c) are self-contained nodes. In Figure 5.4(b), the sub-query $R1 \bowtie R3$ is to be run at site 1, and $R1$ and $R3$ are also located at site 1. So the node for this sub-query is a self-contained node. Self-contained nodes may be evaluated in parallel.

Partial Node A node is a *partial* node if one or more streams involved in the sub-query are located at other nodes. In Figure 5.4(b), the sub-query involving streams $R1$, $R2$ and $R4$ contains two join operations ($R1 \bowtie R2$ and $R2 \bowtie R4$) to be run at site 2, but $R1$ and $R4$ are not located at Site 2. So this sub-query forms a partial node in DJG. Partial nodes can cooperate with other nodes (could be partial nodes or self-contained nodes) in a pipelined

manner, which corresponds to vertical parallelism.

A node u has an outgoing edge pointing to a partial node v, if a stream involved in one or more operations of v is contained in the result of u (either as part of the joining result with other streams or as the result of filters if any). For example, in Figure 5.4(c), node 1 has an outgoing edge pointing to node 2, since stream $R1$ is contained in the result of node 1 (as part of the joining result with $R3$) and $R1$ is involved in the join operation $R1 \bowtie R2$ to be executed at node 2. We can see that *the direction of an edge actually indicates the direction of tuple transmission.*

We note that we did not address the fragmentation of streams. As stated in the previous section, when a stream is composed by several substreams delegated to a few sites, the operations assigned to the delegation sites are parallelized onto these sites using the mechanism addressed in section 5.2.2. This scenario can be represented in the DJG by replicating the node which is to be parallelized. For brevity, we only present the scenario where the whole stream is delegated to a single site.

Algorithm 5.1 describes the scheme to transform an annotated join graph into a DJG. The annotated join graph is described by an adjacency list (variable *adj*), an edge list (variable *edge*) and also the labels of the nodes and edges (variable *site*). The output DJG is also represented as an adjacency list. The algorithm first creates an DJG with empty edge set and then initializes the two auxiliary variables: *visitedList* and Q (lines 1,2). It then performs a breadth-first traversal through the input annotated join graph AJG to create the DJG (lines 3-19). In lines 10-11, the operations about the visiting node v are added into the corresponding node in the DJG. Lines 12-15 creates the necessary edges incurred by the visiting node v in the DJG. Line 16 adds v to the *visitedList* to avoid subsequent duplicate visits, while line 17 adds it into the FIFO queue Q to further

Algorithm 5.1: AJG_To_DJG(AJG): Transforming an Annotated Join Graph AJG to a Distributed Join Graph

```
1  Create a DJG with empty edge set and one self-contained node for
   each processing site;
2  visitedList ⇐ Q ⇐ ∅;
   // visitedList: a list of visited nodes; Q is a FIFO queue.
3  visitedList.add(s); // s is an arbitrary node in AJG.
4   Q.enqueue(s);
5  add local access operation of s to the node corresponding to site[s] in
   DJG;
6  while !Q.isEmpty() do
7      u ⇐ Q.dequeue();
8      for each v ∈ adj[u] do
9          if v ∈ visitedList then continue;
10         add local access operation of v to the node in DJG
           corresponding to site[v];
11         add join operation u ⋈ v to the node corresponding to
           site[edge[u, v]] in DJG;
12         if site[edge[u, v]] ≠ site[u] AND site[edge[u, v]] ∉ adj'[site[u]]
           then
13             adj'[site[u]].add(site[edge[u, v]]);
               // adj' is the adjacency list of the resulting
                  DJG.
14             Change the node of site[edge[u, v]] in DJG to partial node;
15         visitedList.add(v);
16         Q.enqueue(v);
17 return DJG;
```

process its adjacent nodes.

Properties of DJG

To generate the final query plan, we need to incorporate the communication operators. For each edge in the DJG, we should add one RMO(/RO)-RA pair to the two connected sites. However, this is not enough. For example, in Figure 5.4(c), it is possible to route the intermediate result tuples of Site 5 to Site 2 first and then to Site 6. Therefore, even though there is no edge between node 5 and node 2, we

should add on RMO(/RO)-RA pair to transmit tuples from Site 5 to Site 2. The final distributed plan is shown in Figure 5.4(d) (the bit vectors are not shown). In our set up algorithm, we have to consider all the possible routing paths while leaving out the impossible ones to avoid redundant routing and void routing as stated in Subsection 5.2.1.

Before we look at the algorithm to incorporate the communication operators to produce the distributed query plan, let us present several interesting properties of a DJG that the algorithm is based upon.

Property 5.1 *A self-contained node has only outgoing edges.*

This is because a self-contained node involves only those operations whose streams are co-located.

In a DJG, if a partial node v has an incoming edge from another node u, u is called a parent of v. Moreover, if there is a path from node u to node v, then u is an ancestor of v and v is a descendant of u. Obviously, self-contained nodes have no parent or ancestors.

Property 5.2 *Descendants of all the self-contained nodes contain all the partial nodes.*

This can be proven by noting that there is at least one parent, say u, for any partial node. If u is not a self-contained node, then it should have a parent. This process can be continued until it reaches a self-contained node. That means for each partial node, there is a path originated from a self-contained node. Property 5.2 means that we can traverse all nodes in the DJG by traversing the descendants of all self-contained nodes.

Property 5.3 *For a partial node, only those tuples that have been routed through at least one of its parents should be routed to it.*

A partial node u actually joins the intermediate results of its parents and the local streams if any. So only those tuples that have been routed through one of its parents can be used in the operations of u. Based on this property, we can make the following observation:

Observation 5.1 *Ancestors of a node u, except its parents, should not route their intermediate result tuples to u, since these tuples could not have been routed through any of u's parents. Symmetrically, a node u should not route its intermediate result tuples to its ancestors, since they had already processed these tuples.*

5.3.2 Incorporating the Communication Operators

Algorithm 5.2: SETUP(DJG): Setting up the distributed plan

```
1  for each self-contained node s do
2      pre[s] ⇐ null; // pre:  an array of parents
3      anc[s] ⇐ null; // anc:  an array of lists of ancestors
4      visitedList ⇐ Q ⇐ ∅;
       // visitedList:  a list of visited nodes; Q is a FIFO
          queue.
5      visitedList.add(s);
6      Q.enqueue(s);
7      while !Q.isEmpty() do
8          u ⇐ Q.dequeue();
9          for each v ∈ adj[u] do
10             pre[v] ⇐ u;
11             anc[v].addAll(anc[u]);
               /* ancestors of u are also ancestors of v, since u
                  is v's parent.                                  */
12             anc[v].add(u);
13             for each t ∈ visitedList do
14                 if t ∉ anc[v] OR t = u then CreateOutputModule(t, v);
15                 if t ∉ anc[v] then CreateOutputModule(v, t);
16             visitedList.add(v);
17             Q.enqueue(v);
```

Algorithm 5.3: CreateOutputModule(u, v): Creating output operators to transmit tuples from one site to another site

```
1  if u already has an output module pointing to v then return;
2  if (u already has output modules) then
3      for each output module r of u do
4          CreateOutputModule(v,the remote node of r);
5          CreateOutputModule(the remote node of r, v);
           /* tuples routed from u to v may need to route to
              those nodes later and vice versa.                */
6      replace the existing RO module, if any, with an RMO module;
7      add an RMO module r to u outputting to v;
8  else
9      add an RO module r to u outputting to v;
10 add an RA module to v to connect with r;
11 for the AllowingBits of r, set the bits corresponding to u and pre[v],
   and clear the bit corresponding to v;
12 create a ValidatingMask for r with the above bits set;
```

We are now ready to describe the algorithm to set up the processing plan. Algorithm 5.2 gives an algorithmic description of how to setup the processing plan given a DJG. Algorithm 5.3 is a routine used in Algorithm 5.2. For brevity, we do not address the set up of regular operators (such as LA, SteMs, etc.) in the algorithm, which are straightforward [3], and only focus on the newly introduced operators: RMO/RO and RA. The setup algorithm assumes that the input graph is represented using adjacency list and the variable *adj* is the adjacent list for the input DJG. As shown in line 1, the algorithm begins from each self-contained node to traverse the DJG. Lines 2-3 set the parent and ancestors of the self-contained node, and based on Property 5.1 they are both null. From line 4 to line 20, it traverses all of the current self-contained node's descendants in a breadth-first way. The completeness of the traversal is based on Property 5.2. The lines from 10 to 12 set the parent and ancestors of node v. Then lines 13-16 create the necessary

[3]One LA is created to access a local source. One SteM is created in a site for each source's attribute that appears in the join predicates to be evaluated in that site.

output operators (RO or RMO) between the visited nodes and the current node v. Due to Observation 5.1, we avoid routing a node's results to its ancestors and vice versa. After these processing, line 17 and line 18 end the current iteration by adding the current node v to the visited list and the FIFO queue Q. Then a new iteration is started. After applying the algorithm, every node has the necessary output operators created.

Algorithm 5.3 creates the module pair: RO-RA or RMO-RA to transmit tuples from site u to site v and creates AllowingBits and the ValidatingMask accordingly. Most part of the algorithm is self-explanatory, except for lines 3-6. We note that if a node u already has one or more RO/RMO to transmit tuples to other sites, then u can choose to transmit its result tuples to those sites first and then to v or the reverse. Therefore, we need to create the transmission relationship between these nodes and node v. That is what the code segment lines 3-6 does.

After running the setup algorithm, all the operators would have been created. In Figure 5.4(d), we illustrate the final query plan for the example query, where arrows represent the transmission directions for the created transmission operators.

5.4 Experiments

In this section, we describe our experimental setup and present the results of various experiments conducted to evaluate our proposed SwAP. For horizontal parallelism, an eddy runs independently in each site. Since this is similar to running an eddy in a single site context, we only focus on the evaluation of SwAP for vertical parallelism.

All experiments are performed on four machines, interconnected with a 100M LAN. Like the running example on vertical parallelism, there are three streams:

Table 5.2: Configuration of processing sites.

Name	CPU	Memory	Operating System
Site 1	P4 2.4G	512M	MS Windows XP Pro.
Site 2	P3 1G	256M	MS Windows 2000
Site 3	P4 2.4G	256M	MS Windows XP Pro.

R, S and T, delegated to three different sites (machines): Site 1, Site 2 and Site 3 respectively. Queries are submitted on the fourth site. The main configuration of the three processing sites are listed in Table 5.2. To examine the efficiency of our mechanism, we measure the response time for processing a number of tuples from three streams. We set the number of tuples of streams R, S and T as 10000, 100000 and 10000 respectively. All streams have one attribute a and the values in that attribute are uniformly distributed. By changing the range of this attribute, we can control the selectivities of the joins. For example, based on the cardinalities and the value ranges of two streams, we can calculate the average number of tuples at each value for each stream. Multiplying the product of these two numbers and the overlapped range, we can get the cardinality of the join result. In all the experiments, $R \bowtie S$ and $S \bowtie T$ are run at Site 1 and Site 3 respectively. We choose the relatively large cardinality of stream S to better show the effect of the choice of tuple routing orders. Although the choice of processing sites in this case may not be optimal, it does not affect the validity of the experiments. Unless explicitly stated, both joins are implemented as symmetric pipelined hash joins using SteMs.

We have implemented a prototype of SwAP based on the Java code of Telegraph system [64]. There is an instance of the distributed version of the telegraph server running on each processing site. For the network communication, we used java.nio package, which provides efficient unblocking I/O API. For the virtual tuple transmission, we adopt an aggressive approach: once the network allows the trans-

mission, we will transmit the virtual tuple immediately, otherwise we accumulate the number needed to transmit.

5.4.1 Learning Static Characteristics

In the first set of experiments, we compare the performance of SwAP with static plans when all the characteristics are static. The static plans are implemented by employing a fixed join ordering and employing only the RO-RA pair of operators, i.e., there is no RMO-RA pair of operators so that no runtime adaptivity is supported. In this way, by comparing with the static plans we can evaluate the effectiveness of SwAP as well as the overhead of introducing the Virtual Tuple mechanism in RMO. In all the experiments of this subsection, the running query is a three-way join: $R \bowtie S \bowtie T$.

First, we study how well SwAP can learn the selectivities of operations. In this experiment, we fix the selectivity of $S \bowtie T$ with respect to S (i.e., $|S \bowtie T|/|S|$) to 100%, and change the selectivity of $R \bowtie S$ w.r.t. S so that it is 200% in one version and 20% in the other. The transmission speeds and workloads of the two processing sites are about the same. Under this scenario, the best static plan in the first version of the experiment is to evaluate $S \bowtie T$ before $R \bowtie S$, while the reverse is true for the second case. This is because evaluating $R \bowtie S$ in the first case may increase the join size to be further processed to 200%, while it may reduce the join size to 20% in the other case. Figure 5.5 shows the response time of the three different schemes for both cases. We can see that the response time of the SwAP scheme is very close to the best static plan in both cases, while the worst static plan took much more time to complete. This not only implies that SwAP is effective, but it also shows that the overhead of SwAP is not significant. Figure 5.6 shows that nearly 80% of tuples are routed through the optimal order

in both cases in SwAP. Figure 5.7 shows the percentage of tuples routed under back-pressure effect and ticket routing scheme. We can see that when both of the selectivities w.r.t. S are larger than or equal to 1, nearly 90% of the tuples are routed under the back-pressure effect. That is because neither operators can accumulate positive tickets and hence the ticket routing scheme cannot be applied. The effectiveness of back-pressure effect in this case is due to the fact that sites with higher selectivities may take more time to finish than sites with lower selectivities do. On the contrary, when one of the selectivities w.r.t. S is less than 1, then tuples are mainly routed under the ticket routing scheme.

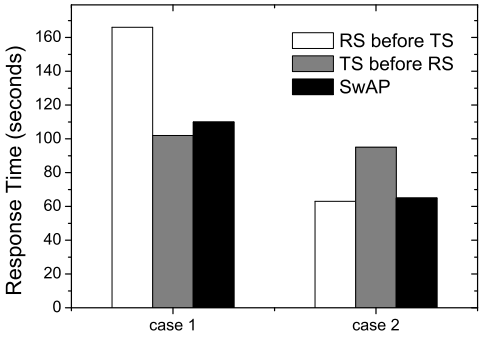


Figure 5.5: Performance on static selectivity

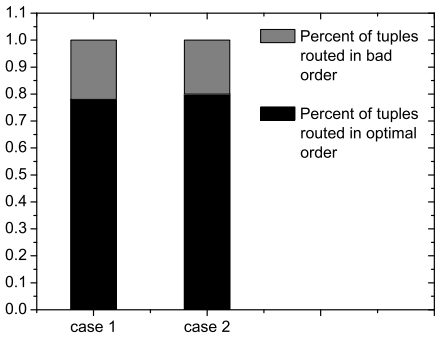


Figure 5.6: Percent of tuples routed in either way

In the second experiment, we study how SwAP adapts to the transmission speeds of the sites. In this experiment, both joins are implemented as index joins to facilitate the changing of selectivities. First, we fix both selectivities of the two joins w.r.t. S to 10%. But Site 3 has a slow connection to the other two processing sites. To simulate the slow transmission speed, output modules take 10 ms to send a tuple to the site with slow connection. Under this situation, the best static plan is to send tuples of S to Site 1 to perform the join with stream R first, and then route the resulting tuples to Site 3 to perform the remaining join

operation. Figure 5.8 is the resulting response time of the three schemes. As the figure shows, SwAP turns out to outperform the optimal static plan slightly. This is because even though the transmission speed of Site 3 is slow, Site 3 is lightly loaded. For the best static plan, at some time, the memory of Site 1 may be filled with join results that are waiting to be output to Site 3. At this moment Site 1 cannot process more tuples, while Site 3 is idle. SwAP can exploit this idle time by sending some tuples of S to Site 3. In this way, Site 3 can utilize the idle time to evaluate the join $(S \bowtie T)$ before sending the result tuples to Site 1 to produce the final answers.

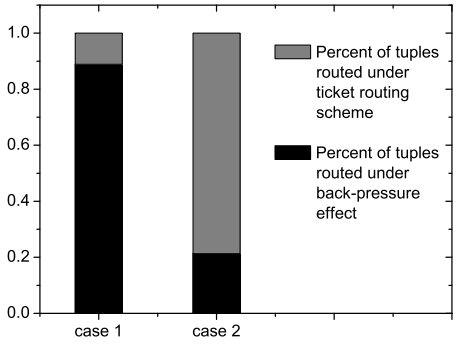


Figure 5.7: Effect of routing strategies

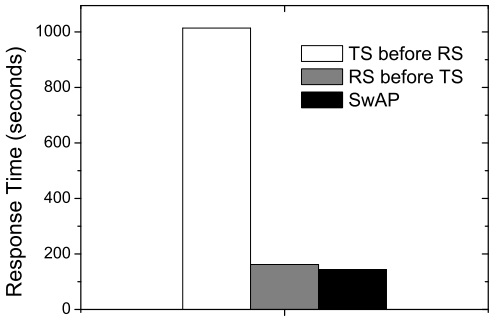


Figure 5.8: Performance on static transmission speed

The above result inspires us to do another experiment to see how well SwAP can outperform the best static plan when the selectivities of the joins varied. Figure 5.9 shows the response time of both schemes when the selectivities of both joins w.r.t. S are varied between 0 to 0.9. We can see that the response time of both schemes increases linearly while the line for SwAP grows more slowly. Figure 5.10 shows the change in the percentage of tuples routed through Site 1 first when the selectivities of the joins varied in SwAP. When the selectivities are 0, nearly all the tuples are routed through Site 1 first. This is because operations in Site 1 can eliminate all tuples to be sent to Site 3 which has a slow transmission

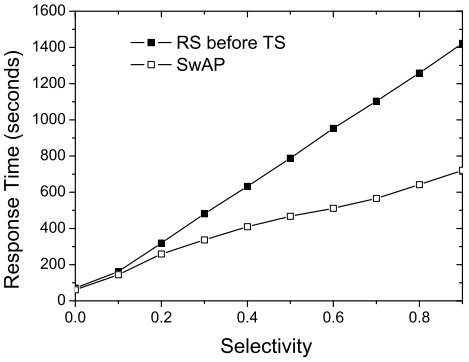


Figure 5.9: Different selectivity

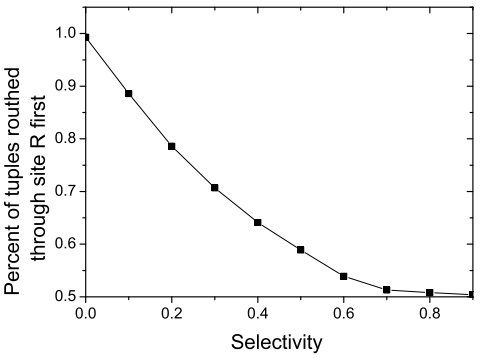


Figure 5.10: Percent of tuples routed through site R first

speed. When the selectivities become higher, more and more tuples are routed through the other way. This is because Site 1 can eliminate fewer tuples when the selectivities are higher. When the selectivities w.r.t. S approach 1, nearly the same number of tuples are going in either way. Although tuples going through Site 3 first will be transmitted two times through the slow connection while those going through Site 1 first only need to be transmitted once through the slow connection, the completion time of either way is about the same due to the pipelined effect.

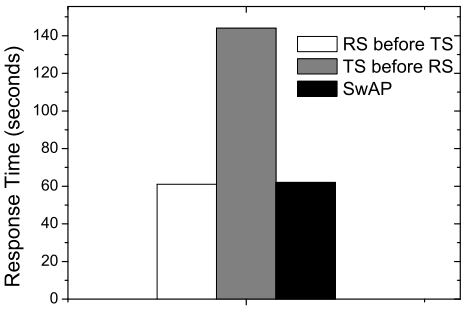


Figure 5.11: Performance on static workload

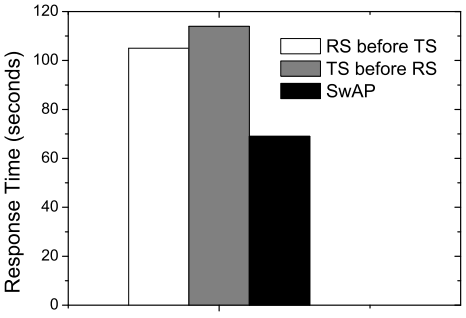


Figure 5.12: Performance of adapting to fluctuations of selectivity

The last experiment in this subsection is to study how well SwAP can learn the static workloads of processing sites. To simulate the high workload, we created a thread that ran a spin loop that may cost a lot of CPU cycles. In this experiment, we make Site 3 the overloaded site. The selectivities of the two joins w.r.t. S are both 10%. Again, the best static plan is to perform $R \bowtie S$ first. We can see from Figure 5.11 that SwAP approaches the optimal static plan.

5.4.2 Adapting to fluctuations

In this set of experiments, we study how well SwAP adapts to the fluctuation of selectivity, transmission speeds and workloads of servers. The running query is the same as that used in the previous set of experiments.

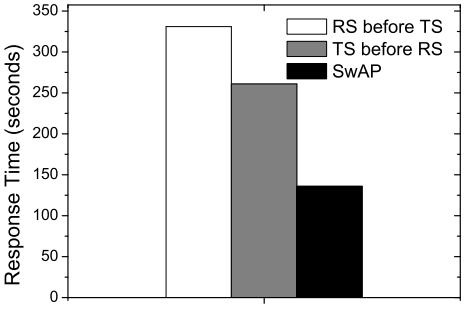


Figure 5.13: Performance of adapting to fluctuations of transmission speed

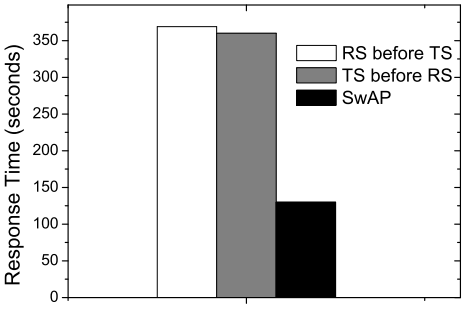


Figure 5.14: Performance of adapting to fluctuations of workloads of servers

First, we consider the fluctuation of selectivity. For the first 50,000 tuples of stream S, the selectivity of $R \bowtie S$ w.r.t. S is 20%, while the selectivity of $S \bowtie T$ w.r.t. S is 200%. For the remaining tuples, we toggle the selectivities of the two joins. As stated in [8], the benefits of an adaptive scheme for changing only the selectivities of two operators are not very dramatic. But the benefits become larger if there are more operators and the changing of selectivity becomes more dramatic.

Figure 5.12 shows the performance of the two static plans compared with SwAP. As we would hope, SwAP outperforms the two static plans. In this query, for the first 50,000 tuples of S, the best join order is $(R \bowtie S) \bowtie T$; however, for the remaining tuples of S, the best join order becomes $(T \bowtie S) \bowtie R$. Since the static plans employed only one join order, they are unable to adapt to the change in selectivities resulting in overall poorer performance. SwAP, on the other hand, can cope with the change in selectivities to adapt to the best join order during runtime.

Second, we study how well SwAP adapts to the fluctuation of transmission speeds of servers. For this experiment, we fix both the selectivities of the two sites w.r.t. S to be 50%. Initially the connection to Site 3 is slow and the connection to Site 1 is fast. After 250 seconds, the two sites swap transmission speeds. Here, output operators take 10 ms to send a tuple over a slow connection. As shown in Figure 5.13, SwAP is much more efficient than both the static plans for reasons similar in logic to the earlier experiments, i.e., the static plans only perform best for a limited time period, while SwAP is optimal most of the time.

A similar experiment is done for studying the adaptivity of SwAP to the workload fluctuations of servers. In this experiment, we created five delay threads for the overloaded site. And the selectivities of the two sites w.r.t. S are fixed at 10%. Initially Site 3 is overloaded while Site 1 has normal workload. After 70 seconds, we toggle the workload of the two sites. The results, shown in Figure 5.14, indicate that SwAP is superior over the two static plans. The superiority is attributed to SwAP's ability to adapt the tuple routing orders according to the workload fluctuations in the midst of processing.

5.5 Summary

In this chapter, we have presented a novel distributed query processing mechanism, namely SwAP, which can optimize the query operation orders by adaptively learning the operator selectivities, as well as the transmission speeds and workloads of processing servers. When these properties changed during runtime, SwAP can also adapt its behavior accordingly to approach an optimal plan. Moreover, all runtime decisions are made in a distributed manner. Hence it is scalable. Furthermore, more routing strategies can be easily incorporated.

Chapter 6

Dynamic Operator Placement

The previous chapter presents a runtime adaptive operator ordering mechanism executed over an adaptable distributed query plan. However, such adaptable query plans would become very complicated once the number of joins as well as the number of queries increases. Hence, in this chapter, we propose another mechanism to optimize the performance of the intra-provider layer by dynamically and optimally placing the operators. This mechanism is more scalable to the number of queries as well as the complicity of the queries. More specifically we make the following contributions:

- We formally define the metric *Performance Ratio (PR)* to measure the relative performance of each query and the objective for the whole system (informally, we want to minimize the worst relative performance among all queries).
- By building a new cost model, we identify the heuristics that can be used to approach the objective. More specifically, the heuristics (1) balance the load among all the processing nodes; (2) restrict the number of nodes that the operators of a query can be distributed to; (3) and minimize the total communication cost under conditions (1) and (2).
- The design objective of a platform independent (independent on the underlying stream processing engines) and non-intrusive load management scheme distin-

guishes our approach from existing ones (e.g. [86]). The proposed techniques are meant to allow the leveraging of exiting well developed single-site stream processing engines without much modifications. This is reflected throughout the design of the whole system and especially reflected in the load selection strategy.

- To support heuristic (1), we focus on new architectural design that allows us to tap on existing well studied load balancing algorithms instead of proposing new ones. The architectural design includes constructing the load migration unit, load management partner selection, online collection of load statistics, selection of operators to be migrated, operator migration mechanisms.

- To reduce the overhead of employing heuristic (2), unlike existing proposals [32, 82, 86] where load (re)distribution is done at the operator level, we adopt the notion of *query fragments* (a subset of operators) as the finest migration unit. It also helps reduce the overhead of making load balancing decisions.

- To employ heuristic (3), we propose the data flow aware load selection strategy to select the query fragments to be migrated. It effectively maintains data flow locality so that the communication cost is minimized.

- We conducted an extensive simulation study to evaluate the proposed strategy. Results show that the proposed strategy can effectively adapt to the runtime changes of the system to approach our objective.

The rest of this chapter is organized as follows. Section 6.1 formulates the problem and presents our analysis. We present the details of our system design in Section 6.2. Experiment results are presented in Section 6.3. Finally Section 6.4 summarize the chapter.

6.1 Problem Formulation and Analysis

In this section, we formulate the problem setting and define the metric to measure the system performance, followed by a formal presentation of the problem statement. Finally, we analyze the problem by building a new cost model and present the proposed heuristics.

6.1.1 Problem Formulation

In the system there is a set of geographically distributed data stream sources $S = \{s_1, s_2, \cdots, s_{|S|}\}$ and a set of distributed processing nodes $N = \{n_1, n_2, \cdots, n_{|N|}\}$ interconnected by a local network. As mentioned in Section 1.1.3, each source stream is routed to other processing nodes through a delegation node. We denote the delegation scheme as Ω. Users impose a set of continuous queries $Q = \{q_1, q_2, \cdots, q_{|Q|}\}$ over the system. The set of operations $O_k = \{o_1, o_2, \cdots, o_{|O_k|}\}$ of query q_k might be distributed to a set of nodes $N_k \subseteq N$ for processing. The operators we consider include filters, window joins and window aggregations. In addition, we denote the set of streams that a query q_k operates on as S_k.

Like previous work on continuous processing of streams [23, 82], we are concerned about the delay of resulting data items, which is also one of the main concerns of end users in terms of system performance. More formally, if the evaluation of query q_k on a source tuple $tuple_l$ from stream s_l generates one or more result tuples, then the delay of $tuple_l$ for q_k is defined as $d_k^l = t_{out} - t_{in}$, where t_{in} is the time that $tuple_l$ arrived at the system and t_{out} is the time that the result tuple is generated. If there are more than one result tuples, then t_{out} is the time that the last one is generated. A similar metric was used in [82]. We focus on this metric because users in a continuous query system typically make decisions based

on the results arrived so far. Shorter delay of result tuples would enable a user to make more timely decisions.

At a closer look, d_k^l includes the time used in evaluating the query (denoted as p_k^l), the time waiting for processing as well as the time it is transferred over the network connections. For a specific processing model and a particular query q_k, we regard the evaluation time p_k^l as the inherent complexity of q_k. Since different queries may have different inherent complexities, the value of d_k^l cannot reflect correctly the relative performance of different queries. For example, a query may experience a long delay because its evaluation time is long. We cannot conclude that the relative performance of this query is worse than another one which has a shorter evaluation time. However, in a multi-query and multi-user environment, we wish to tell the relative performance of different queries. Hence we propose a new metric *Performance Ratio (PR)* to incorporate the inherent complexity of a query. Formally, the PR_k^l of the processing of $tuple_l$ for q_k is defined as $PR_k^l = \frac{d_k^l}{p_k^l}$. And the performance ratio of q_k is defined as $PR_k = \max_{s_l \in S_k} PR_k^l$. PR_k reflects the relative performance of q_k. Our objective is to minimize the worst relative performance among all the queries.

The formal problem statement is as follows: *Given a set of queries Q, a set of processing nodes N, a set of data stream sources S and a delegation scheme Ω, according to the change of system state, dynamically distribute the operators of each query to the $|N|$ processing nodes so that the maximum performance ratio $PR_{max} = \max_{1 \le k \le |Q|} PR_k$ is minimized.*

6.1.2 Problem Analysis

In this section we develop a cost model to estimate the values of d_k^l and p_k^l. Note that our cost model is meant to be simple for us to figure out the main factors that

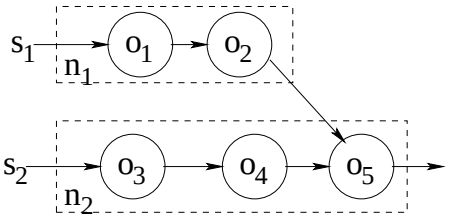


Figure 6.1: An example query plan

affect these values and to allow us to analyze the problem complexity. Finding that the problem is NP-hard, we design some heuristics to help solve the problem.

Cost Model

In our cost model we adopt the following simplifications and assumptions:

1. Operators of each query compose a separate processing tree. They are grouped into query fragments and distributed to the processing nodes. Figure 6.1 shows an example processing tree for a query whose operators are grouped into two query fragments and distributed to two nodes: n_1 and n_2. Tuples arrived at each node are processed in a FIFO manner. Only when an input tuple[1] is fully processed would a new input tuple be processed. The cost of delivering the final results to the users is not considered.

2. For an operator o_j, we assume its per-tuple evaluation time t'_j is independent of its location. And we define its average per-tuple selectivity sel_j as the average number of tuples that would be generated for a given input tuple.

3. Workload ρ_i of a node n_i is defined as the fraction of time that the node is busy.

Given these assumptions, we now look at how to estimate p_k^l and d_k^l. In a particular execution plan of a query, for source tuples from each querying source, there is a path composed by some operators and possibly some network connec-

[1] A tuple here could be a batch of individual tuples in a batch processing mode.

tions. For example, in Figure 6.1, the path for source tuples from s_1 consists of o_1, o_2, o_5 and the connection between n_1 and n_2, while the path for those from s_2 comprises o_3, o_4 and o_5. Hence, roughly speaking, the p_k^l and d_k^l of a source tuple are respectively equal to the total processing time of the operators in its path and the total time that the tuple stays in its path. In the following paragraphs we will compute them one by one.

For query q_k, assume the path for source tuples from s_l comprises a set O_k^l of operators and some network connections. Furthermore, let O_k^l be distributed to a set N_k^l of nodes and $O_{k,i}^l \subseteq O_k^l$ be the subset of operators of O_k^l assigned to node n_i (where $n_i \in N_k^l$). Let the average per-tuple evaluation time of operator $o_{l_j} \in O_k^l$ be t'_j and its average per-tuple selectivity be sel_j. Without loss of generality, assume o_{l_j} is processed before $o_{l_{j+1}}$. Note that only those source tuples that would be output as result tuple(s) are counted in our metric (hence, each operator's selectivity on these particular tuples is at least 1). Assume $tuple_l$ from s_l is such a tuple, then the average processing time of o_{l_j} incurred by $tuple_l$ is $t_j = t'_j \prod_{h=1}^{j-1} \max(sel_h, 1)$. Hence we have

$$p_k^l = \sum_{o_{l_j} \in O_k^l} t_j. \tag{6.1}$$

In our model every processing node is a queueing system. From queueing theories [50], in all solvable single task queueing systems, the time that a data item spends in a system can be calculated as $t = g(\rho) * t_s$, where t_s is the processing time of a data item and $g(\rho) \geq 1$ is a monotonically increasing concave function of the system's workload ρ. The exact form of $g(\rho)$ depends on the type of system, e.g. $g(\rho) = \frac{1}{1-\rho}$ in an M/M/1 system.

This inspires us to model the delay of $tuple_l$ as

$$d_k^l = (\sum_{n_i \in N_k^l} (f(\rho_i) \times \sum_{o_{l_j} \in O_{k,i}^l} t_j)) + t_c \times m, \tag{6.2}$$

where t_c is the communication delay of a tuple and m is the number of times that a tuple is transferred over the network. $f(\rho_i)$ is a monotonically increasing concave function. Note that $f(\rho_i)$ is different from $g(\rho)$ mentioned above and may have a much higher value than $g(\rho)$. That is because there are multiple tasks running on each node. We assume $f(\rho_i)$ is identical for all nodes. Hence the first term of the right-hand side of Equation (6.2) summarizes the delay in the processing nodes while the second term summarizes the delay caused by the communications.

Based on Equations (6.1.1), (6.1) and (6.2), we have

$$PR_k^l = PPR_k^l + CPR_k^l, \tag{6.3}$$

where

$$PPR_k^l = \frac{\sum_{n_i \in N_k^l} (f(\rho_i) \times \sum_{o_{l_j} \in O_{k,i}^l} t_j)}{\sum_{o_{l_j} \in O_k^l} t_j}, \tag{6.4}$$

and

$$CPR_k^l = \frac{t_c \times m}{\sum_{o_{l_j} \in O_k^l} t_j}. \tag{6.5}$$

We call PPR_k^l the processing performance ratio (PPR) and CPR_k^l the communication performance ratio (CPR). Analogously, $PPR_k = \max_{s_l \in S_k} PPR_k^l$ and $CPR_k = \max_{s_l \in S_k} CPR_k^l$.

Problem Complexity

Given the cost model, let us examine the complexity of the problem. We can observe that the total number of possible allocation schemes is $|N|^{|O|}$ where $O = \bigcup_{1 \leq k \leq |Q|} O_k$. Even worse, we can derive that the problem is actually NP-hard. To see this let us first ignore the communication cost and only consider minimizing $PPR_{max} = \max_{1 \leq k \leq |Q|} PPR_k$. It is easy to see from Equation (6.4) that PPR_k^l is a weighted sum of the $f(\rho_i)$ values, where the weight for $f(\rho_i)$ is the fraction of evaluation time p_k^l allocated to node n_i. Assume we can migrate the load between nodes in the finest granularity. Then we have the following observation.

Observation 6.1 *To minimize PPR_{max}, PPR_k is equal for all queries and ρ_i is equal for all nodes.* □

The intuition behind it is when PPR_k of a query q_k is higher than the others, we can always allocate more resources to q_k (i.e. reducing the workload of some of the processing nodes for q_k by load migration to the other nodes) so that PPR_k is still the largest but is reduced. When the load is balanced then PPR_k equal to $f(\overline{\rho})$ for all queries, where $\overline{\rho}$ is the uniform workload of all nodes. However, we cannot migrate the load in the finest granularity in practice and hence the best plan is to minimize the difference of loads among all the nodes. By restricting our problem to ignore the communication cost, it is equivalent to a MULTIPROCESSOR SCHEDULING problem which is NP-hard. Hence our problem is NP-hard.

Heuristics

In view of the complexity of the problem, we opt to designing heuristics instead of finding an optimal algorithm. From the estimation equation d_k^l, we know that the extra delay is caused by the communication and the workload of the system.

Hence, we adopt the following heuristics. (1) Dynamically balance the workload of the processing nodes. This heuristic is inspired by Observation 6.1. (2) Distribute operators of a query to a restricted number of nodes so that communication overhead of a query is limited. We call the maximum of this number as the distribution limit of that query. Note that always distributing all the operators of every query to a single node is impractical, because it would incur excessive data flow over the network.(3) Minimize the communication cost under conditions (1) and (2). In short, we have to design a dynamic load balancing scheme where the operations of each query should not be distributed to too many nodes and the total communication traffic is minimized.

Besides employing the heuristics stated above, the scheme should also satisfy the following objectives in the perspective of system design:

1. *It is fast and scalable.* Because dynamic re-balancing could happen frequently at runtime, the overhead of making re-balancing decisions should be kept low. Furthermore, a distributed scheme is preferred to enhance scalability and avoid bottleneck.

2. *It does not rely on any specific processing model.* There are different single-node processing models that are currently under development such as TelegraphCQ [31], Aurora [22] and STREAM [81]. Our system is not restricted to any processing model because it separates the stream processing engine in each node from the distributed processing details. Queries are compiled into logical query plans which consist of *logical operators*. The logical operators are distributed to the processing nodes by our placement scheme. Then the logical operators would be mapped into *physical operators* by the stream processing engine for processing. Different engines under different processing models could map a logical operator into a different physical operator.

6.2 System Design

In our dynamic operator placement scheme, we adopt a local load balancing strategy. Each node would select its load management partners and dynamically balances the load between its partners. To implement this, there are several issues to be addressed: (1) initial placement of operators; (2) load management partner selection; (3) workload information collection; (4) load balance decision-making; (5) selection of operators for migration; and (6) migration strategy. We address these issues in the following subsections.

6.2.1 Initial Placement of Operators

In our initial placement scheme, we only consider minimizing the communication cost and leave the load balancing task to our dynamic scheme. The scheme generates one query fragment for each participating stream and then distributes the query fragments to the delegation nodes of their corresponding streams. More specifically, the scheme comprises the following steps:

1. When a query is submitted to the system, it is compiled and optimized into a logical query plan without considering the distribution of the data streams. The logical query plan, which is represented as a traditional query plan tree, determines the required logical operators such as filters, joins, aggregation operators and their processing orders. Existing optimization techniques [83, 9] can be applied at this step. Figure 6.2(a) is an example of the resulting query tree of this step.

2. For each stream involved in the query, generate one query fragment which is initially set to empty. Add each leaf node (i.e. the stream access operators) to its corresponding query fragment QF_i and then replace it with QF_i.

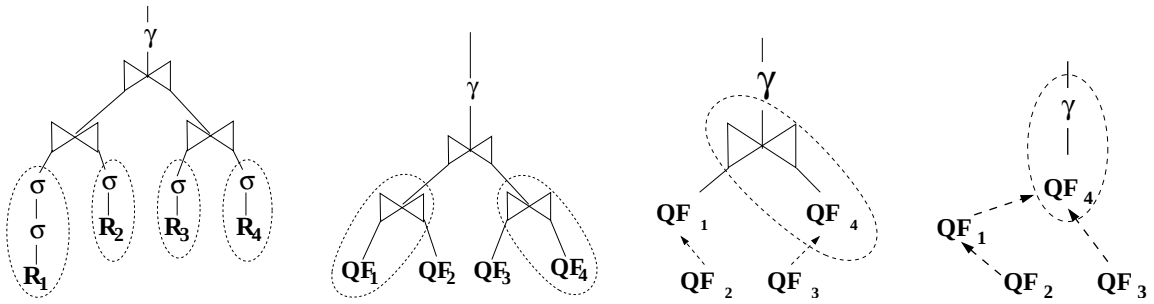
3. For each query fragment, if the parent operator is a unary operator, the

operator would be added to the query fragment and removed from the query tree. The step is repeated until all the operators are removed or the parent operator for every query fragment is a binary operator. Figure 6.2(b) is an example of the resulting query tree of this step. The intuition is to place each stream's filters at its delegation node to reduce the amount of data to be transferred.

4. Now we have a query tree in which all the next-to-leaf nodes are binary operators. Add each next-to-leaf binary operator to one of its two child query fragments, say QF_i, whose estimated resulting stream rate is higher than the other one. Then remove the other query fragment from the tree and push QF_i up a level to replace that binary operator. A binary operator is added to the query fragment of higher (estimated) resulting stream rate to reduce the volume of data that needs to be transmitted through the network if the two fragments of the two involved streams are to be evaluated at two different nodes. This process continues until all operators are removed or the parents of one or more of the remaining query fragments are unary operators. For the latter case, the algorithm goes back to step (3). Figures 6.2(c) and (d) illustrate the procedure of this step.

5. Distribute the query fragments to the delegation nodes of their corresponding streams.

Based on the operator ordering, there is a downstream and upstream relationship between some of the query fragments. For example, in Figure 6.2, results of QF_2 should be further processed by the binary operator of QF_1 and hence we call QF_2 the upstream query fragment of QF_1. Similarly, QF_1 is the upstream query fragment of QF_4. Symmetrically, we call QF_1 (or QF_4) the downstream query fragment of QF_2 (or QF_1). We call a query fragment's downstream or upstream query fragments its neighbors. For instance, QF_2 and QF_4 are neighbors of QF_1. Furthermore, if a query fragment QF_i's corresponding data stream is delegated to



(a) Phase 1 (b) Phase 2 (c) Phase 3 (d) Phase 4

Figure 6.2: Query Fragments Generation

Algorithm 6.1: PARTNER SELECTION

```
1  Function PartnerSelect()
2  begin
3      sort neighbors in descending order of neighboring factor;
4      for (i ← 0; |g1| < max1 AND i <|neighbors|+ MaximumTry; i++) do
5          if i < |neighbors| then  n ← neighbors[i];
6          else  n ← a random node ∉ neighbors ∪ g1;
7          if n ∈ g2 then
8              move it from g2 to g1;
9          else if n ∉ g1 then
10             send a request to n;
11             if the request is accepted then
12                 add n to g1;
13 end
```

a node n_j then QF_i is called a *native query fragment* of n_j and n_j is a *native node* of QF_i. Otherwise, QF_i is called a *foreign query fragment* of n_j and n_j is a *foreign node* of QF_i.

Furthermore, the native nodes of two neighboring query fragments are called *neighbors* to each other. And the number of neighboring query fragments between two nodes is called the *neighboring factor*.

6.2.2 Partner Selection Strategy

As stated above, our dynamic load balancing scheme is a local strategy. Each node n_i has a number of load management partners (abbreviated as partners). The partner relationship is symmetric, i.e. if n_i is a partner of n_j, then n_j is also a partner of n_i. In this section, we discuss the partner selection strategy for each node.

In our scheme, each node sends out requests to some other nodes to initiate the partner relationships and receives such requests from its peers. We separate the partners of each node into two groups : (1) g_1, the relationship is created by the (explicit) request of this node; (2) g_2, the rest. There is a maximum bound for each group of partners denoted as max_1 and max_2 respectively. Each node would use Algorithm 6.1 to send out requests. Neighbors with higher neighboring factors with the current node have higher priority to be selected. That is to enhance the opportunity of reducing communication cost during load redistribution, which is can easily be seen in Section 6.2.5. Algorithm 6.1 is implemented in asynchronous mode in our system. It does not wait for a remote response but instead returns once all requests have been sent out. After a node receives a response message, the algorithm is called to resume the processing. Furthermore, a node n_i which receives a request will check whether the sender n_j is also being requested by n_i or is already in g_1. If so, n_i accepts the request and adds n_j into g_1 if necessary. Otherwise it adds n_j into g_2 if $|g_2| < max_2$ or sends back a reject message otherwise. A node will update its partners periodically.

6.2.3 Information Collection Strategy

The information collection strategy determines when and how workload information of nodes in the system is collected and also what information is to be collected.

We adopt a window based and asynchronous workload collection approach. Time is divided into windows which have static lengths τ. Each node accumulates the total processing time t of all its physical operators within each window and the workload with respect to a window is computed by dividing t by τ. Each node asynchronously collects its workload within each window and updates its workload once the current time window elapsed. It broadcasts the workload information to all its partners if its workload increases to κ or decreases to $1/\kappa$ times of the last broadcast value.

The above strategy performs well only if the input rate and the processing time are constants. But in practice they are random variables. The resulting workload may fluctuate over time, which renders the system unstable. As stated before, we only focus on adaptation to long term system changes which would bring long term benefits and alleviate the short term adaptation overhead. To prevent the system from reacting to short term fluctuations, we use a low pass filter to remove the high frequency noises (caused by the short term changes of stream rates, tuple processing time, etc.) in workload collection. In particular, workload is computed as $\rho_{i+1} = \alpha \times \rho_i + (1-\alpha) \times \rho_c$, where ρ_{i+1} and ρ_i are the workload information used for load balancing after $i+1$ and i time windows, and ρ_c is the collected workload within the $(i+1)$th time window. α is a parameter to determine the responsiveness of the estimated value to the workload changes. The purpose of using this formula in previous work is to give more weight to recent collected statistics. Here we analytically show that it can also smooth out short term fluctuations.

Figure 6.3 shows the effect of low pass filter in the estimation of workload of an M/M/1 machine. The average input rate of data is 10 tuples/ms before the fifth second and becomes 14 tuples/ms after the fifth second. The average processing time is 1ms/tuple. The workload collection window is set to $10ms$ and $\alpha = 0.9$.

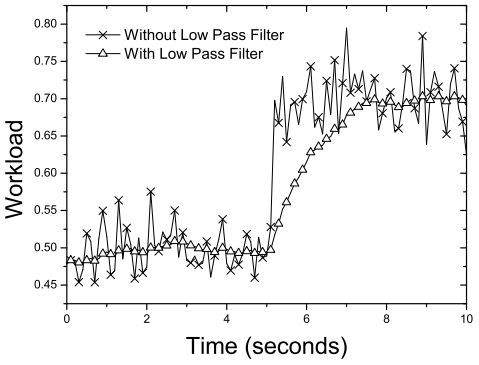


Figure 6.3: Effect of low pass filter

As shown in the figure, workload estimation without using the low pass filter is more fluctuant than the one with low pass filter.

We now consider how α would be set in a system. Without loss of generality, we assume the workload is increasing. Given the initial workload ρ_0 and that we want to filter out transient workload fluctuation where the workload is changed to $l\rho_0$ $(l > 1)$ within $m_1\tau$ time and last for $m_2\tau$ time, we should choose α such that the estimated workload after $(m_1+m_2)\tau$ time $\rho_{m_1+m_2}$ should satisfy $\rho_{m_1+m_2} \leq \kappa\rho_0$. In practice, the values of m_1, m_2 and l reflect the typical range and time span of short term fluctuations. They can be adaptively tuned by collecting the characteristics of the system. In our calculation, we assume the workload increases $(l-1)\rho_0/m_1$ within each τ time during the $m_1\tau$ period. After the $m_1\tau$ time, the estimated workload is

$$\begin{aligned}
\rho_{m_1} &= \rho_0 + \sum_{i=1}^{m_1}(\alpha^{m_1-i}(1-\alpha)\frac{(l-1)\rho_0}{m_1}i) \\
&= \rho_0 + (1-\alpha)\alpha^{m_1}\frac{(l-1)\rho_0}{m_1}\sum_{i=1}^{m_1} i(\frac{1}{\alpha})^i \\
&= \rho_0 + \frac{(l-1)\rho_0}{m_1}(m_1 + 1 + \frac{\alpha^{m_1+1}-1}{1-\alpha})
\end{aligned}$$

The last step in the derivation is based on the fact:

$$\sum_{i=1}^{m_1} ix^i = \frac{x + (m_1 x - m_1 - 1)x^{m_1+1}}{(1-x)^2}.$$

Then after $m_2\tau$ time, the estimated workload $\rho_{m_1+m_2}$ can be calculated as:

$$\alpha^{m_2}\rho_0 + \frac{\alpha^{m_2}(l-1)\rho_0}{m_1}(m_1 + 1 + \frac{\alpha^{m_1+1}-1}{1-\alpha}) + (1-\alpha^{m_2})l\rho_0.$$

Substitute the above equation into the inequality $\rho_{m_1+m_2} \leq \kappa\rho_0$, we have

$$\alpha^{m_2} + \frac{\alpha^{m_2}(l-1)}{m_1}(m_1 + 1 + \frac{\alpha^{m_1+1}-1}{1-\alpha}) + (1-\alpha^{m_2})l \leq \kappa.$$

Hence we can calculate the lower bound of α by solving the above inequality given the values of m_1, m_2 and l. For example, given $m_1 = m_2 = 1$, $l = 2$ and $\kappa = 1.2$, we can get $\alpha \geq 0.9$. The case for short term workload decrease can be analyzed similarly.

On the other hand, if α is too high, the estimated workload may not be able to reflect the current workload, hence the system would response too slowly to the workload changes. To show this effect, we do the following calculations. Assume that the workload is changed from ρ_0 to $l\rho_0$ and then remain steady. Further assume that the change happens within 0 time. Then similar to the above calculation, we have the estimated workload after $m\tau$ time as

$$\rho_m = \alpha^m \times \rho_0 + (1-\alpha^m) \times l \times \rho_0.$$

If we want the estimated workload to reflect k percent of the actual workload, we

have to solve the inequality $\rho_m \geq k \times l \times \rho_0$, which results in

$$m \geq \frac{\ln \frac{(k-1)l}{1-l}}{\ln \alpha}.$$

Let $l = 2$, $k = 1 - (1/2e) = 0.92$. $m \geq 10$ for $\alpha = 0.9$, and $m \geq 20$ for $\alpha = 0.95$. While the optimal value of α depends on the specific situation, in this chapter, we fix it at 0.9 which is shown to be efficient under our experimental configuration in the performance study.

6.2.4 Load Balance Decision Strategy

Algorithm 6.2: Generate Load Requests

```
1 Function GenRequest()
2 begin
3     Compute the average workload ρ̄ within itself and its partners;
4     if the local workload κρ_l < ρ̄ then
5         Find the partner n_i whose workload ρ_i is the largest;
6         Compute the load request ρ_r = (ρ_i − ρ_l)/2;
7         Request ρ_r amount of workload from n_i;
8 end
```

The load balance decision strategy determines whether it is beneficial to initiate a load balance attempt and how much workload should be transmitted between the nodes. Our strategy is adapted from the local diffusive load balancing strategy introduced in [84]. It is a receiver-initiated strategy, which is found to be more efficient in [84]. It works in rounds. The length of each round is denoted as Δ. Each node maintains its own value of Δ. At the start of each round, Algorithm 6.2 is run to generate one workload request if necessary. In this algorithm, the load request is generated by the potential load receiver (i.e., the node with smaller load initiates load balancing). Since we focus on continuous queries, load migration can bring long term benefits. As such our decision strategy does not consider the short

term migration overhead. Once a node receives a workload request, it satisfies the request as much as possible, provided the workload to send out within each Δ time window is no more than half of its total workload at the beginning of the current window.

It is possible that the nodes in the system are separated into several non-overlapping groups and the workloads are not balanced between groups. Hence once a node in our system detects that itself and all of its partners are overloaded, it will randomly probe the other nodes until it finds an underloaded node to add it as a partner or the probe limit is reached.

6.2.5 Load Selection Strategy

As stated above, once a potential load sender receives a load request, it will select the *victim* query operators to satisfy the request as far as possible. When multiple such requests are received, the sender processes them in descending order of the workload amount requested. The sender will estimate its resulting workload after each migration, and if it detects that half of the workload has been exported within the current Δ interval, it will stop processing any request until the start of the next round. In this subsection, we explore how to select the victim operators for migration and discuss how to migrate them in the next subsection.

Migration Unit.

The first question to be answered is what is the smallest task unit used for load migration. We consider the following choices:

1. Using *the whole logical query* as the migration unit is easy to implement. However, a good evaluation plan often distributes the operations across multiple nodes in order to minimize the communication overheads. So migrating in the

unit of logical query is inappropriate.

2. *Logical operator* as another candidate is a fine-grained unit. Migrating at this level may result in better balance state. However, it is hard to implement our second heuristic which imposes a distribution limit on the query operators (see Section 6.1.2). When we are trying to move an operator, we have to know the location of the other operators belonging to the same query. Otherwise, we do not know if the distribution limit is violated. This results in high update overhead and is not compatible to our local strategy as a node cannot make decisions based on local information.

3. Another candidate *physical operator* that is proposed in previous work [32, 82, 86], has similar pros and cons. Furthermore, as stated above, physical operators may be shared by multiple queries [58]. Hence migrating a physical operator affects all the queries sharing that operator. It becomes harder to maintain good plans for all the queries sharing that operator. Another shortcoming is that the underlying stream processing engines need to be tightly coupled with the load balancing strategy. That means excessive complexity has to be introduced into the existing stream processing engines.

4. *Query fragments*. Based on the above analysis, a good candidate for migration unit should render the maintenance of good query plans easy and allow the separation of load balancing strategy from the underlying stream processing engine and hence introduce less complexity to the existing processing techniques. Furthermore, this unit should not be too coarse to restrict the adaptive ability of the load management module. For the above purposes, we would like to find a subset of operators that is of appropriate size and would be processed in the same site in most cases for a good query plan. Furthermore, we consider only candidates in the logical level. We adopt the notion of query fragment - a subset of logical

operators of a query. We set the number of query fragments of a query as its distribution limit. This exempts the task of keeping track of the distribution of all the operators of a query while we are implementing heuristic (2). The distribution limit would always be met no matter where we allocate the query fragments.

While a query can be fragmented in a lot of ways, we simply use the query fragments generated in our initial placement scheme as the migration units. Operators in each of such query fragments would be allocated to the same processing node in a good query plan generated by applying traditional optimization heuristics. Furthermore, by doing so, the distribution limit of a query is set to the number of streams involved by the query. Here, we assume that queries involving more streams are more complicated and hence can afford a higher distribution limit.

Note that migrating logical level query operators/fragments may sometimes create more physical operators than migrating physical operators. E.g., we might have to create an additional physical operator when we migrate a logical operator that shares a physical operator with another non-migrated logical operator. However, this is the price we have to pay for a platform independent and non-intrusive load management scheme.

Data Flow Aware Load Selection.

The choice of query fragments to be migrated is critical in maintaining data flow locality. A poor choice may cause streams to be scattered across too many nodes and result in network congestion. In this subsection, we propose a lightweight query fragment selection strategy which makes decisions only based on local information.

In our strategy, for each request, the sender chooses the query fragments in the following order until the request is satisfied or half of the workload of this node

has been exported within the current Δ interval.

1. *Query fragments that are foreign to the sender but native to the receiver.* This kind of query fragments is considered to be of highest priority to migrate because migrating them has the potential to reduce the data flow.

2. *Other query fragments that are foreign to the sender.*

3. *Query fragments that are native to the sender.* This kind of query fragments is considered of lowest priority for migration because migrating them tends to scatter the streams delegated to this node.

The above heuristics are reasonable in maintaining data flow locality. However, its categorization is too coarse. The migrations of the query fragments within each category may still have different effects on the data flow locality and the delay of the queries. For example, migrating a query fragment QF_i to a node that is evaluating a neighbor of QF_i may bring less increase of data flow than migrating it to other nodes. This is because it avoids the transfer of the data flow between QF_i and its neighbor. Hence, within each of the above categories, we further classify the query fragments into one of the following categories and we list them in the order of descending migration priorities.

1. *Query fragments that have neighbors being evaluated at the receiver but none at the sender.* The migration of this class of query fragments eliminates the transmission of the data flow between the sender and the receiver caused by the migrated query fragment. Figure 6.4(a) shows a possible situation in this case. The situations before and after migration are plotted on the left and the right respectively. Solid arrows in the figure indicate the data flows between the query fragments. For brevity, the other query fragments being evaluated in the two nodes are not shown. In this example QF_1 is a neighbor of QF_2. n_i is the sender while n_j is the receiver. After migration, the data flow introduced by QF_1 and QF_2 between n_i

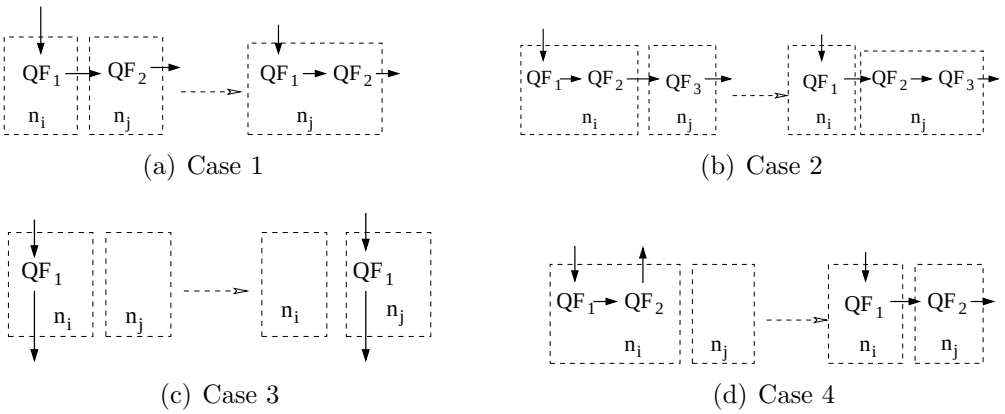


Figure 6.4: Query fragments migration cases

and n_j is eliminated.

2. *Query fragments that have neighbors at both nodes.* This class of query fragments has lower migration priority than the above-mentioned one because the migration eliminates one data flow but also creates another one between the sender and the receiver. For example, in Figure 6.4(b), the transmission of the data flow introduced by QF_2 and QF_3 is eliminated while the one incurred by QF_1 and QF_2 is created by the migration.

3. *Query fragments have neighbors at neither node.* Figure 6.4(c) is an example situation.

4. *Query fragments that have neighbors at the sender but none at the receiver.* This class has lower priority than the third one because the migration may introduce extra data flow between the sender and the receiver. An example of this case can be found in Figure 6.4(d). The migration in this example creates the data flow between n_i and n_j caused by QF_1 and QF_2.

If there is more than one query fragment in the above subcategories, we will compute the migration priority for each of them and will migrate those with higher priorities first. The migration priority of a query fragment is computed as $\frac{\rho}{\max(size,1)}$, where ρ is the workload it incurs, and $size$ is its state size in bytes.

We call this value the *load density* of the query fragment as it means the amount of workload will be migrated for each byte of state transmission. Furthermore, ρ is estimated by summing up the estimated workload incurred by each of its logical operator, which is estimated as $1/n$ of the workload caused by its corresponding physical operator. n is the number of logical operators sharing that physical operator.

6.2.6 Migration Strategy

After the sender had chosen the query fragments for migration, it would perform the migration operation in the following steps. First, it redirects the input stream(s) of the migrating query fragments to the receiver. After the stream processing engine has drained the data currently in the system for the migrating query fragments, the query fragments are removed and their intermediate state information (such as joins, aggregations), if any, are extracted from the stream processing engine. Then the query fragments and their state information are shipped to the receiver. The receiver will add the query fragments and install their state information into the stream processing engine. Here we assume the stream processing engine has a data buffer strategy under which a query can specify when to start or resume the evaluation [12, 31]. Furthermore, we assume there is a unique timestamp associated with each tuple from each stream. The sender records with the shipped query fragments the timestamp(s) of the last processed tuple(s) so that the receiver node will resume the query fragments at that point of time.

6.3 A Performance Study

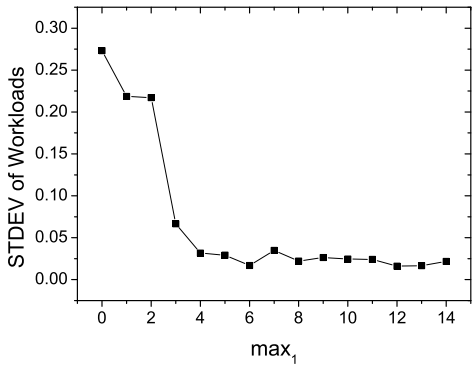
Our experiments are performed on a Linux server with Intel Xeon 2.8GHz CPU and 2.5GB RAM. The stream processing engine in each node is an emulation of the TelegraphCQ system and implemented based on the Java code of the TelegraphCQ system, where joins are evaluated using SteM operators while filters are implemented using group filters to share computations between queries. We choose the TelegraphCQ system model because it is able to add or remove queries efficiently at very frequent moments (the moment that a tuple is fully processed). Following existing work [74, 86], we use a simulator to simulate the communication among the processing nodes. The simulator is implemented in JAVA using the JavaSim discrete event simulation package. We use 32 simulation nodes and an additional sink node as our basic configuration. Each processing node is delegated 3 streams. Tuples from every stream are of 100 bytes and consist of 10 attributes. The bandwidth of the network connecting the nodes is modeled as 100Mbps.

We use 500 queries and a total of 5750 logical operators, to measure our system performance. Each query q_k is generated in the following steps: (1) randomly choose the number of querying streams $|S_k|$; (2) pick the set of querying streams S_k conforming to a particular distribution which will be stated in the following experiments; (3) create 4 filters on 4 randomly chosen attributes for each querying streams; (4) create $|S_k| - 1$ equi-joins and ensure that each stream is involved in at least one join. The sliding window size for window joins is randomly selected from 5000 to 20000. The selectivities of the operators are from 0.5 to 0.8. We set the average data inter-arrival time to be $4ms$ and the mean processing time for each filter and join operation to be $20\mu s$ and $80\mu s$ respectively. Besides, we use the following algorithm parameters: the workload collection window $\tau = 100ms$, the length of load management round $\Delta = 1s$, and the threshold to broadcast

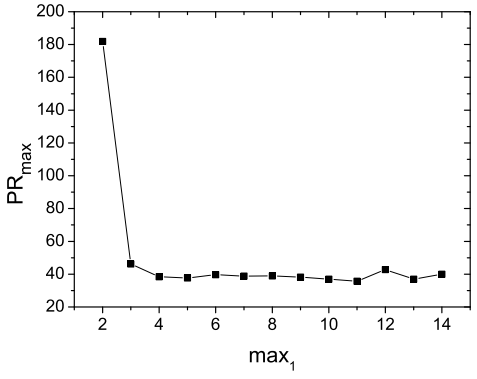
workload $\kappa = 1.2$. The real values of p_k^l and d_k^l were collected online and the PR_k values were computed by the sink node when it received a result tuple. In all the computations, we do not consider the cost of transmitting results to the sink node and the cost of transmitting data streams from their sources to their delegation nodes. That is because these costs are fixed given a fixed problem setting and are irrelevant to our scheme.

6.3.1 Partner Selections

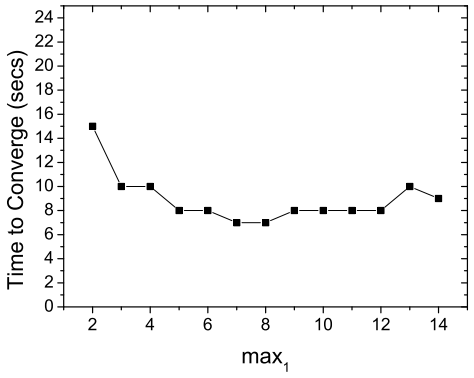
We have two parameters for our partner selection strategy: max_1 and max_2. In this experiment, we set $max_2 = \lceil \frac{1}{2} max_1 \rceil$ and vary the value of max_1. The system is initially in an unbalance state generated by using a zipfian distribution ($\theta = 0.95$) to select the querying streams S_k for all queries. We use the standard deviation (STDEV) of the ρ_i for all processing nodes to measure the load imbalance, i.e. $\sqrt{\frac{\sum_i (\rho_i - \bar{\rho})^2}{|N|-1}}$. Figure 6.5(a) shows the final load distribution for different values of max_1. $max_1 = 0$ means that dynamic load balancing is disabled. We can see when $max_1 >= 4$ the load is well balanced. No significant improvement can be made by using a larger max_1 value. Figure 6.5(b) illustrates the PR_{max} after the system is stable. It is computed by averaging on the values within 10 seconds. It is clear that the PR_{max} values are also similar when $max_1 >= 4$. Figure 6.5(c) shows the time it takes to converge to the final load distribution. There is not much difference between small and large number of partners. The above comparisons show that our system works well with a small max_1 value. As a larger number of partners would increase the runtime cost (such as transferring workload update messages, making load balancing decisions), we could keep the number to a small value and hence keep the cost low. In the subsequent experiments, we set $max_1 = 5$ and $max_2 = \lceil \frac{1}{2} max_1 \rceil$.



(a) Load imbalance



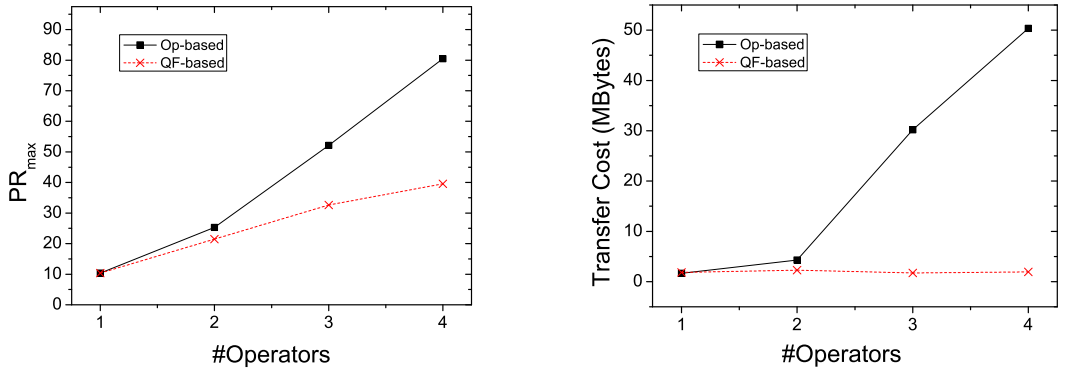
(b) PR_{max}



(c) Time to Converge

Figure 6.5: Effect of various partner selection parameter

6.3.2 Load Selection Heuristics



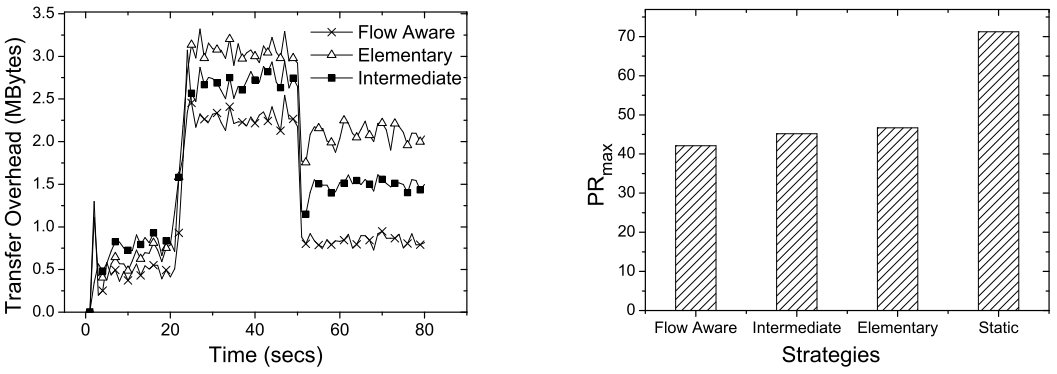
(a) PR_{max} (b) Transfer cost

Figure 6.6: QF-based vs. OP-based

The first experiment examines the necessity of imposing a distribution limit. This is done by comparing the QF-based (query fragment based) load balancing strategy with the OP-based (operator based) strategy proposed by reference [86]. The latter approach does not impose any distribution limit. To generate an imbalanced workload, the streams that a query operates on are chosen according to a Zipfian distribution ($\theta = 0.95$). Initially the query fragments are distributed using the static allocation scheme. Then the system would detect the imbalance and hence initiate load balance actions. We varied the number of operators per query fragment in our experiment. We ran the experiment under each case for 60 seconds simulation time and report the average values.

Figure 6.6 presents the result. We can see that when the number of operators in each query fragment is fewer or equal to 2, both PR_{max} are nearly identical. However, when the number reaches 3, the PR_{max} of the OP-based scheme increases to a very high value. Figure 6.6(b) may be able to explain this phenomenon. The data transfer volume of the OP-based scheme increases quickly with more opera-

tors. That is because operators of a single query are migrated to too many sites in the OP-based scheme and hence the data streams are scattered over the network and leads to network congestion. On the other hand, the QF-based strategy still maintains small transfer overhead and hence it still performs well in data delay. Note that, by employing a distribution limit, an OP-based strategy can achieve better performance. However, as analyzed before, the cost to maintain such a limit would be higher than a QF strategy and such a scheme does not fit into a local load management strategy.



(a) Transfer Overhead

(b) PR_{max}

Figure 6.7: On load selection strategies

The second experiment examines the effectiveness of our flow-aware load selection strategy in maintaining good data flow locality. We impose an initially balanced load distribution over the processing nodes and use a uniform distribution to choose the querying streams S_k for every query q_k. At time $t = 20s$ we randomly select 4 nodes and then increase the input rates of the streams delegated to those nodes to 3 times of their initial values. At $t = 50s$, the increased input rates drop back to their initial values.

To show the effect of load selection strategy, we design another two approaches

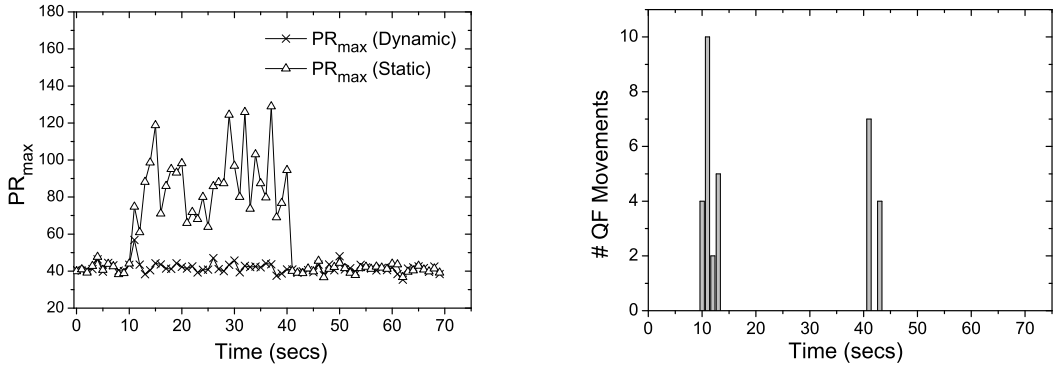
for comparison: (1) Elementary: the query fragments are selected in descending order of their load density. (2) Intermediate: the same as Elementary except foreign query fragments are given higher migration priorities than the native query fragments. In previous work, such as [74, 86], data flow relationship is not considered. Hence their effects on the communication cost can be well represented by the Elementary algorithm. We compare the transfer overhead introduced by the three strategies against the static query fragment allocation strategy, i.e. the initial placement scheme. The static strategy allocates the query fragments to their native nodes, hence its data flow transfer cost is minimum though it may incur very high data delay due to the unbalanced load allocation. We subtract the amount of transfer cost of the static strategy from those of the other three and then compare the extra transfer overheads of the three dynamic strategies over the static one.

From figure 6.7(a), we can see that the data flow aware strategy outperforms the other two at all stages of the experiment. Both Intermediate and Elementary , unlike the data flow aware strategy, fail to identify the neighborhood relationship of the query fragments. Intermediate is better than Elementary because it can differentiate between foreign query fragments and native query fragments and to some degree can help maintain data flow locality. At $t = 50s$ when the perturbed stream rates dropped back to the original value, all three strategies' transfer overheads are reduced. However, both Intermediate and Elementary cannot restore back to the state prior to the change. This is because both strategies are unable to identify their native nodes when migrating foreign query fragments. That means they would become worse and worse with the evolution of the system state while the data flow aware strategy is able to maintain a more stable state over time.

Figure 6.7(b) shows the PR_{max} for all the four strategies. The values are calcu-

lated by averaging over the whole simulation time. The static strategy performed the worst simply because of the absence of load balancing strategy. Furthermore, the three dynamic strategies performed similarly. This is attributed to our heuristic to maintain a distribution limit for every query. Since processing load are similar for the three dynamic strategies due to the balanced load distribution, PR_{max} was similar for the three strategies. However, in the case when network traffic is so high that it approaches the bandwidth limit, the data flow aware strategy will do much better to avoid network congestion situation.

6.3.3 Adapting to Changes of System State

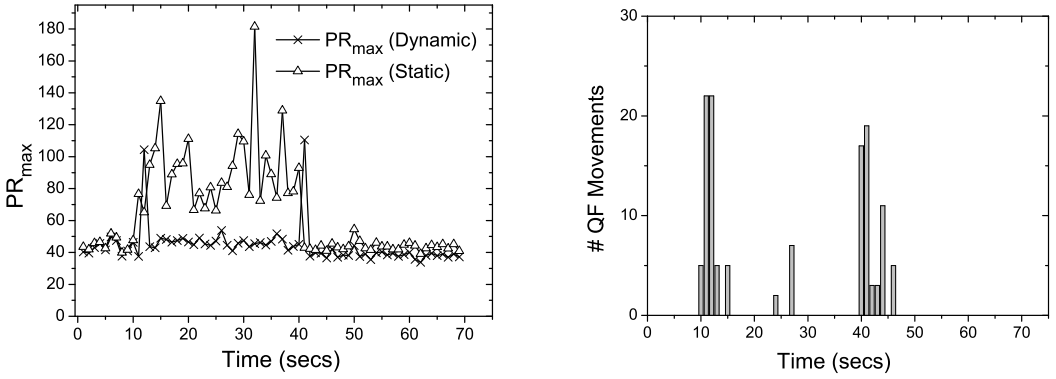


(a) PR_{max} (b) #Query fragment migrations

Figure 6.8: Small perturbation on stream rates

In this subsection we examine the system performance when the system state changes. A good system should be able to quickly adapt to changes and then remains steady. For the first experiment, we examine the stability of our mechanism under changes of workload. The settings are similar as the previous experiment. Figure 6.8 shows the result of this experiment. As shown in Figure 6.8(a), the PR_{max} values for both static and dynamic strategies are identical before the per-

turbation of stream rates. At time $t = 10s$, the input rates of the streams of a random node were increased and hence rendered some nodes' workload increased. The query fragments running on the perturbed nodes suffer long delay, hence the PR_{max} in the static case increased significantly. However users in the dynamic case nearly have no sense of the changes in PR_{max}, except a temporary increase during the load migration period. The temporary increase is due to the stalled processing of the migrated query fragments. This good performance can be attributed to the load balancing strategy's ability to amortize the workload of the processing nodes. At $t = 40s$ the stream rates dropped back to the original value and the PR_{max} of the static case is restored back to the state prior to the change. Now both the static and dynamic schemes behave identically again.



(a) PR_{max} (b) #Query fragment migrations

Figure 6.9: Large perturbation on stream rates

Figure 6.8(b) also shows the number of query fragments migrated over time. At $t = 10s$, the dynamic strategy detected the load imbalance and began to migrate the query fragments in order to balance the load distribution. A lot of query fragment migrations occurred within 5 seconds. After that the system became stable and few migrations occurred until $t = 40s$. Some migrations occurred again

within 5-6 seconds and then the system became stable again. We also conducted another experiment on perturbing the stream rates from 4 nodes. As shown in Figure 6.9. The results suggest similar conclusions.

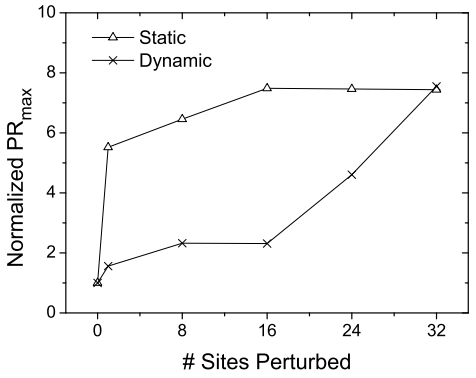
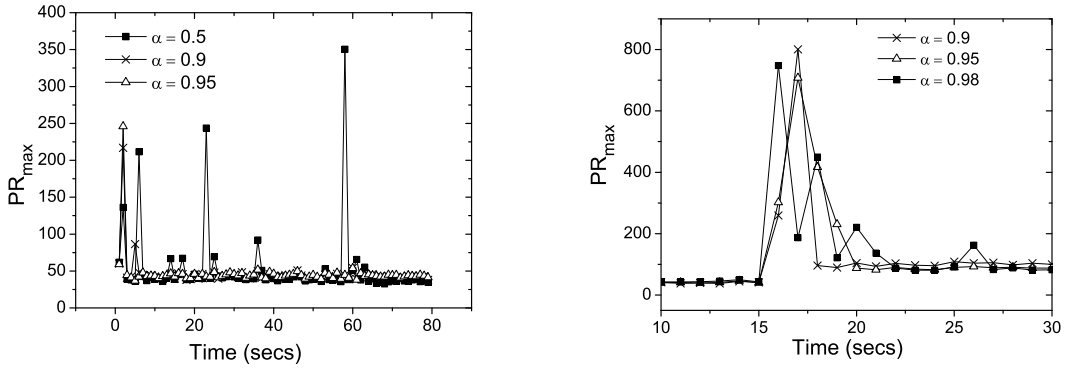


Figure 6.10: On change of workloads

The second experiment shows how the system behaves with external turbulence introduced by ad-hoc queries, which only last for a short time. However, in a multi-user environment, we can expect such ad-hoc queries follow some access patterns over certain period. Once that pattern changes, the system should re-distribute the workload to achieve new balance. The configuration of this experiment is similar to the previous one, except extra workloads of ad-hoc queries are introduced at the $15th$ second instead of the change in stream rates. The workload of a perturbed node is about 0.9. We varied the number of perturbed nodes and ran each case for 60 seconds, then calculated the average PR_{max} values over the whole period. For ease of comparison, we normalized the figures by dividing them by the one without perturbation (whose absolute value can be found from Figure 6.8). From Figure 6.10, we can see that the static case degrades much faster than the dynamic one. This is because the load balancing algorithm amortizes the workload by migrating query fragments and hence alleviates the influence of the workload imbalance

introduced by ad-hoc queries.

6.3.4 Sensitivity to α



(a) No external load

(b) With external load

Figure 6.11: Sensitivity to α

In this section, we examine the sensitivity of our load balancing strategy to the value of α, the parameter of the low pass filter. We conducted two experiments. In one experiment we did not impose any change in the system, while in the other one we increase the external workloads of 8 nodes similar to the experiment in the previous subsection. Figure 6.11(a) illustrates the change of PR_{max} over simulation time under three values of α. It is clear when α is too low, the PR_{max} becomes very fluctuant. A larger α can alleviate this problem by filtering out the high frequency components (i.e. the short term changes). In the second experiment we impose external workloads to 8 nodes at $t = 15s$. To ease viewing, Figure 6.11(b) only presents the portion when the workload varies. After $t = 15s$, all three cases started to react to the change and hence the PR_{max} is increased due to the temporary stalling of the query fragments being migrated. But the larger the value of α, the longer the system takes to react to the change. These two

experiments suggest a trade-off in selecting a proper α value. We leave this issue for our future work.

6.4 Summary

Distributed processing of continuous queries over data streams suffers from run time changes of system resource availability and data characteristics. Dynamic operator placement techniques are desirable for a locally distributed stream processing system. In this chapter, we formalized the problem and analyzed it by building a cost model. We also proposed a load management architecture, which dynamically balances the workload of the locally distributed processors and maintain good data flow locality. As shown in our experiments, load imbalance can cause severe performance degradation and our techniques can alleviate such degradation by dynamic load balancing. Our data flow aware load selection strategy can help restrict the scattering of data flows and lead to lower communication cost.

Chapter 7

Conclusion

In this thesis, we presented several mechanisms that enhance the scalability and adaptability of a large-scale distributed stream processing system. To conclude, we first review the contributions we have made and then propose a few interesting problems for future work.

7.1 Review of Contributions

We began our journey at Chapter 1 by observing that a lot of stream processing applications need to process a large number of complex continuous queries and hence building a scalable and adaptable distributed stream processing system is critical for them. Then we investigated a large-scale system that is composed of a number of autonomous service providers and proposed a two layer architecture to integrate the power of all these service providers.

We observed a few challenges in the two layers respectively. In the inter-provider layer, due to the autonomous and widely distribution of the SPs, loosely coupling and communication efficiency should be stressed in the architecture design. Two overlays were built in this layer to handle the query streams and the data streams respectively.

At the query overlay, we first addressed how to manage the queries to leverage the power of the data overlay to efficiently disseminate both the source data and the result data around the network. Then we studied the problem of distributing the queries among the SPs to achieve load balancing and minimum communication cost. The query distribution problem was modeled as a graph partitioning problem. To enhance the scalability, hierarchical algorithms were proposed to distribute the initial query workload, to route the incoming new queries and redistribute the queries in according to the change of the system.

The data overlay employed a content-based network to disseminate the data throughout the whole network. The SPs were organized into multiple dissemination trees to avoid communication bottlenecks at the sources and to break the coupling between the sources and the destinations. Furthermore, common communication of different destinations was naturally shared in such an architecture. Here an optimization algorithm is required to construct optimal dissemination trees. Hence we proposed an adaptive algorithm and two static algorithms to solve this problem. The adaptive algorithm was shown to be robust the inaccurate statistics and runtime change of system parameters, while the static algorithms worked close to the optimum by given accurate apriori statistics.

After investigating the inter-provider layer, we then concentrated on the intra-provider layer. An SP could employ a cluster of locally distributed processors to enhance its scalability. Queries allocated to an SP could be divided into multiple fragments and evaluated in parallel at multiple processors. We proposed two mechanisms to harness these processors. The first mechanism deployed multiple eddy [8] operators at multiple processors to adaptively optimize the ordering of the distributed operators. As shown in this thesis, such an mechanism can quickly detect the changes of operator selectivities, transmission speed as well as processor

workload and adapt the query plan accordingly.

On the other hand, the second mechanism proposed in the intra-provider layer enabled the dynamic placement of query operators among the processors. After formally analyzing the problem, we identified several heuristics to achieve the optimization objective. To implement these heuristics, queries were partitioned into multiple query fragments and a local algorithm was proposed to dynamically (re)distribute them to the processors. A low pass filter was applied on the collected statistics to filter out short term variances hence our decisions can be made on long term changes. Furthermore, the proposed data flow aware load selection strategy was shown to be effective in maintaining data flow locality and hence help avoid network congestion.

7.2 Future Work

Based on the proposed architecture, we believe we have only studied a small portion of the problem. There are a few interesting future directions to be explored.

Sharing of computation resources. In the current scheme of the inter-provider layer, we only considered the sharing of communications. As the widely distributed SPs would be running queries with similar operations, exploiting the sharing of computation resources among these queries is beneficial. This is a very hard problem. First, we need a mechanism to efficiently discovery the similarities among the widely distributed queries. Second, an algorithm is required to generate a processing plan that maximizes the sharing of computation resources but does not impair the communication efficiency. Last but not the least, these algorithms should be scalable and adaptable.

QoS management. In such a service oriented system, QoS management is an

important issue. It is desirable to allow the users to specify their QoS preferences. Example QoS parameters are: delay and accuracy of query results, the sensitivity of the query to system failures, and the cleaness of data etc. The problem is how to allocate the resources to maintain the QoS requirements. For example, more resources should be allocated to process queries that have higher requirements in delays and accuracies. Queries are more sensitive to failures should be allocated to more stable nodes for processing. Furthermore, queries' preferences on data cleaness also affect the amount of resource to be put to clean the data.

Fault tolerance. The SPs are autonomous and hence they can join or leave the system anytime. It is hard to actively control the their availability. Therefore, it is interesting to investigate both proactive and reactive approaches to handle any unexpected leave of SPs. This problem should be addressed in both the query overlay and the data overlay. The fault tolerance mechanism at query overlay is responsible to resume the interrupted queries upon any failures, while the one at the data overlay should recover the broken overlay network as well as retransmit the lost messages.

Bibliography

[1] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Çetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag Maskey, Alex Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stanley B. Zdonik. The design of the borealis stream processing engine. In *CIDR*, pages 277–289, 2005.

[2] Yanif Ahmad and Ugur Çetintemel. Networked query processing for distributed stream-based applications. In *VLDB*, pages 456–467, 2004.

[3] M. Altherr, M. Erzberger, and S. Maffeis. iBus – a software bus middleware for the java platform. In *International Workshop on Reliable Middleware Systems*, pages 43–53, 1999.

[4] Lisa Amini, Navendu Jain, Anshul Sehgal, Jeremy Silber, and Olivier Verscheure. Adaptive control of extreme-scale stream processing systems. In *ICDCS*, page 71, 2006.

[5] Arvind Arasu, Shivnath Babu, and Jennifer Widom. The CQL continuous query language: semantic foundations and query execution. *VLDB J.*, 15(2):121–142, 2006.

[6] Arvind Arasu and Jennifer Widom. Resource sharing in continuous sliding-window aggregates. In *VLDB*, pages 336–347, 2004.

[7] Andrea C. Arpaci-Dusseau, Remzi H. Arpaci-Dusseau, David E. Culler, Joseph M. Hellerstein, and David A. Patterson. High-performance sorting on networks of workstations. In *SIGMOD Conference*, pages 243–254, 1997.

[8] Ron Avnur and Joseph M. Hellerstein. Eddies: Continuously adaptive query processing. In *SIGMOD Conference*, pages 261–272, 2000.

[9] Ahmed Ayad and Jeffrey F. Naughton. Static optimization of conjunctive queries with sliding windows over infinite streams. In *SIGMOD Conference*, pages 419–430, 2004.

[10] Shivnath Babu, Rajeev Motwani, Kamesh Munagala, Itaru Nishizawa, and Jennifer Widom. Adaptive ordering of pipelined stream filters. In *SIGMOD Conference*, pages 407–418, 2004.

[11] Shivnath Babu, Kamesh Munagala, Jennifer Widom, and Rajeev Motwani. Adaptive caching for continuous queries. In *ICDE*, pages 118–129, 2005.

[12] Shivnath Babu and Jennifer Widom. Continuous queries over data streams. *SIGMOD Record*, 30(3):109–120, 2001.

[13] Guruduth Banavar, Tushar Deepak Chandra, Bodhi Mukherjee, Jay Nagarajarao, Robert E. Strom, and Daniel C. Sturman. An efficient multicast protocol for content-based publish-subscribe systems. In *ICDCS*, pages 262–272, 1999.

[14] Suman Banerjee, Bobby Bhattacharjee, and Christopher Kommareddy. Scalable application layer multicast. In *SIGCOMM*, pages 205–217, 2002.

[15] Suman Banerjee, Christopher Kommareddy, Koushik Kar, Samrat Bhattacharjee, and Samir Khuller. Construction of an efficient overlay multicast infrastructure for real-time applications. In *INFOCOM*, 2003.

[16] Philip A. Bernstein, Nathan Goodman, Eugene Wong, Christopher L. Reeve, and Jr. James B. Rothnie. Query processing in a system for distributed databases (sdd-1). *ACM Transactions on Database Systems (TODS)*, 6(4):602–625, 1981.

[17] Pedro Bizarro, Shivnath Babu, David J. DeWitt, and Jennifer Widom. Content-based routing: Different plans for different data. In *VLDB*, pages 757–768, 2005.

[18] Avrim Blum, Prasad Chalasani, Don Coppersmith, William R. Pulleyblank, Prabhakar Raghavan, and Madhu Sudan. The minimum latency problem. In *STOC*, pages 163–171, 1994.

[19] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. Dynamic load balancing in hierarchical parallel database systems. In *VLDB'96*, 1996.

[20] Eli Brosh and Yuval Shavitt. Approximation and heuristic algorithms for minimum delay application-layer multicast trees. In *IEEE INFOCOM'04*, 2004.

[21] Diego Calvanese, Giuseppe De Giacomo, and Maurizio Lenzerini. On the decidability of query containment under constraints. In *PODS*, pages 149–158, 1998.

[22] Donald Carney, Ugur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stanley B.

Zdonik. Monitoring streams - a new class of data management applications. In *VLDB*, pages 215–226, 2002.

[23] Donald Carney, Ugur Çetintemel, Alex Rasin, Stanley B. Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. In *VLDB*, pages 838–849, 2003.

[24] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Achieving scalability and expressiveness in an internet-scale event notification service. In *PODC*, pages 219–227, 2000.

[25] Antonio Carzaniga, David S. Rosenblum, and Alexander L. Wolf. Design and evaluation of a wide-area event notification service. *ACM Trans. Comput. Syst.*, 19(3):332–383, 2001.

[26] Antonio Carzaniga, Matthew J. Rutherford, and Alexander L. Wolf. A routing scheme for content-based networking. In *INFOCOM*, 2004.

[27] Antonio Carzaniga and Alexander L. Wolf. Content-based networking: A new communication infrastructure. In *Infrastructure for Mobile and Wireless Systems*, pages 59–68, 2001.

[28] Antonio Carzaniga and Alexander L. Wolf. Forwarding in a content-based network. In *SIGCOMM*, pages 163–174, 2003.

[29] Ashok K. Chandra and Philip M. Merlin. Optimal implementation of conjunctive queries in relational data bases. In *STOC*, pages 77–90, 1977.

[30] Badrish Chandramouli, Junyi Xie, and Jun Yang. On the database/network interface in large-scale publish/subscribe systems. In *SIGMOD*, 2006.

[31] Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Sailesh Krishnamurthy, Samuel R. Madden, Vijayshankar Raman, Fred Reiss, and Mehul A. Shah. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *First Biennial Conferece on Innovative Data Systems Research(CIDR 2003)*, 2003.

[32] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Cetintemel, Y. Xing, and S. Zdonik. Scalable distributed stream processing. In *Proceedings of the First Biennial Conference on Innovative Database Systems(CIDR'03)*, January 2003.

[33] Yang-Hua Chu, Sanjay G. Rao, and Hui Zhang. A case for end system multicast. In *SIGMETRICS*, pages 1–12, 2000.

[34] Sara Cohen, Werner Nutt, and Yehoshua Sagiv. Containment of aggregate queries. In *ICDT*, pages 111–125, 2003.

[35] Stephen E. Deering. Multicast routing in internetworks and extended lans. In *SIGCOMM*, pages 55–64, 1988.

[36] Alan J. Demers, Johannes Gehrke, Mingsheng Hong, Mirek Riedewald, and Walker M. White. Towards expressive publish/subscribe systems. In *EDBT*, pages 627–644, 2006.

[37] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Trans. Softw. Eng.*, 12(5):662–675, 1986.

[38] Robert S. Epstein, Michael Stonebraker, and Eugene Wong. Distributed query processing in a relational data base system. In *SIGMOD Conference*, pages 169–180, 1978.

[39] Patrick Th. Eugster, Pascal Felber, Rachid Guerraoui, and Anne-Marie Kermarrec. The many faces of publish/subscribe. *ACM Comput. Surv.*, 35(2):114–131, 2003.

[40] Bugra Gedik and Ling Liu. Peercq: A decentralized and self-configuring peer-to-peer information monitoring system. In *ICDCS*, pages 490–499, 2003.

[41] Goetz Graefe. Encapsulation of parallelism in the volcano query processing system. In *SIGMOD Conference*, pages 102–111, 1990.

[42] Goetz Graefe. Query evaluation techniques for large databases. *ACM Comput. Surv.*, 25(2):73–170, 1993.

[43] Peter J. Haas and Joseph M. Hellerstein. Ripple joins for online aggregation. In *SIGMOD Conference*, pages 287–298, 1999.

[44] Alon Y. Halevy. Answering queries using views: A survey. *VLDB J.*, 10(4):270–294, 2001.

[45] B. Hendrickson and K. Devine. Dynamic load balancing in computational mechanics. *Computational Methods in Applied Mechanics & Engineering*, 184:485–500, 2000.

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

[47] Yannis E. Ioannidis and Younkyung Cha Kang. Randomized algorithms for optimizing large join queries. In *SIGMOD Conference*, pages 312–321, 1990.

[48] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: an experimental evaluation. part i, graph partitioning. *Oper. Res.*, 37(6):865–892, 1989.

[49] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[50] Leonard Kleinrock. *Queueing Systems, volume I:Theory*. Wiley, 1975.

[51] Phokion G. Kolaitis, David L. Martin, and Madhukar N. Thakur. On the complexity of the containment problem for conjunctive queries with built-in predicates. In *PODS*, pages 197–204, 1998.

[52] Donald Kossmann. The state of the art in distributed query processing. *ACM Computing Surveys (CSUR)*, 32(4):422–469, 2000.

[53] Ravi Krishnamurthy, Haran Boral, and Carlo Zaniolo. Optimization of non-recursive queries. In *VLDB*, pages 128–137, 1986.

[54] Sailesh Krishnamurthy, Chung Wu, and Michael J. Franklin. On-the-fly sharing for streamed aggregation. In *SIGMOD Conference*, pages 623–634, 2006.

[55] Vibhore Kumar, Brian F. Cooper, Zhongtang Cai, Greg Eisenhauer, and Karsten Schwan. Resource-aware distributed stream management using dynamic overlays. In *ICDCS*, pages 783–792, 2005.

[56] Guy M. Lohman, C. Mohan, Laura M. Haas, Dean Daniels, Bruce G. Lindsay, Patricia G. Selinger, and Paul F. Wilms. Query processing in R*. In Won Kim, David S. Reiner, and Don S. Batory, editors, *Query Processing in Database Systems*. Springer, 1985.

[57] Hongjun Lu and Kian-Lee Tan. Load-balanced join processing in shared-nothing systems. *Journal of Parallel and Distributed Computing*, 23(3):382–398, December 1994.

[58] Samuel Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *SIGMOD Conference*, pages 49–60, 2002.

[59] Manish Mehta and David J. DeWitt. Managing intra-operator parallelism in parallel database systems. In *VLDB*, pages 382–394, 1995.

[60] Werner Nutt, Yehoshua Sagiv, and Sara Shurin. Deciding equivalences among aggregate queries. In *PODS*, pages 214–223, 1998.

[61] Chris Nyberg, Tom Barclay, Zarka Cvetanovic, Jim Gray, and David B. Lomet. Alphasort: A risc machine sort. In *SIGMOD Conference*, pages 233–242, 1994.

[62] Olga Papaemmanouil and Ugur Çetintemel. SemCast: Semantic multicast for content-based data dissemination. In *ICDE*, pages 242–253, 2005.

[63] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-aware operator placement for stream-processing systems. In *ICDE*, page 49, 2006.

[64] Telegragraph project. `http://telegraph.cs.berkeley.edu/`.

[65] Erhard Rahm and Robert Marek. Dynamic multi-resource load balancing in parallel database systems. In *VLDB'95*, 1995.

[66] Vijayshankar Raman, Amol Deshpande, and Joseph M. Hellerstein. Using state modules for adaptive query processing. In *ICDE*, pages 353–, 2003.

[67] David S. Rosenblum and Alexander L. Wolf. A design framework for internet-scale event observation and notification. In *ESEC / SIGSOFT FSE*, pages 344–360, 1997.

[68] Yehoshua Sagiv and Mihalis Yannakakis. Equivalences among relational expressions with the union and difference operators. *J. ACM*, 27(4):633–655, 1980.

[69] Kirk Schloegel, George Karypis, and Vipin Kumar. Graph partitioning for high-performance scientific simulations. pages 491–541, 2003.

[70] Donovan A. Schneider and David J. DeWitt. A performance evaluation of four parallel join algorithms in a shared-nothing multiprocessor environment. In *SIGMOD Conference*, pages 110–121, 1989.

[71] Patricia G. Selinger and Michel E. Adiba. Access path selection in distributed database management systems. In *Proceedings International Conference on Data Bases*, pages 204–215, July 1980.

[72] Patricia G. Selinger, Morton M. Astrahan, Donald D. Chamberlin, Raymond A. Lorie, and Thomas G. Price. Access path selection in a relational database management system. In *SIGMOD Conference*, pages 23–34, 1979.

[73] Mehul A. Shah, Joseph M. Hellerstein, and Eric A. Brewer. Highly-available, fault-tolerant, parallel dataflows. In *SIGMOD Conference*, pages 827–838, 2004.

[74] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *ICDE*, pages 25–36, 2003.

[75] Shetal Shah, Shyamshankar Dharmarajan, and Krithi Ramamritham. An efficient and resilient approach to filtering and disseminating streaming data. In *VLDB*, pages 57–68, 2003.

[76] Shetal Shah, Krithi Ramamritham, and Prashant J. Shenoy. Maintaining coherency of dynamic data in cooperating repositories. In *VLDB*, pages 526–537, 2002.

[77] Niranjan G. Shivaratri, Phillip Krueger, and Mukesh Singhal. Load distributing for locally distributed systems. *Computer*, 25(12):33–44, 1992.

[78] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for in-network stream query processing. In *PODS*, pages 250–258, 2005.

[79] Michael Stonebraker, Paul M. Aoki, Witold Litwin, Avi Pfeffer, Adam Sah, Jeff Sidell, Carl Staelin, and Andrew Yu. Mariposa: A wide-area distributed database system. *VLDB Journal*, 5(1):48–63, 1996.

[80] Liying Tang and Mark Crovella. Virtual landmarks for the internet. In *Internet Measurement Comference*, pages 143–152, 2003.

[81] The STREAM Group. STREAM: The stanford stream data manager. *IEEE Data Engineering Bulletin.*

[82] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *VLDB*, pages 333–344, 2003.

[83] Stratis Viglas and Jeffrey F. Naughton. Rate-based query optimization for streaming information sources. In *SIGMOD Conference*, pages 37–48, 2002.

[84] M. H. Willebeek-LeMair and A. P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Trans. Parallel Distrib. Syst.*, 4(9):979–993, 1993.

[85] Eugene Wong and Karel Youssefi. Decompositiona strategy for query processing. *ACM Transactions on Database Systems (TODS)*, 1(3):223–241, 1976.

[86] Ying Xing, Stanley B. Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *ICDE*, pages 791–802, 2005.

[87] Yongluan Zhou. Adaptive distributed query processing. In *VLDB PhD Workshop*, 2003.

[88] Yongluan Zhou. Scalable and adaptable distributed stream processing. In *ICDE Workshops*, page 148, 2006.

[89] Yongluan Zhou, Beng Chin Ooi, and Kian-Lee Tan. Dynamic load management for distributed continuous query systems. In *ICDE*, pages 322–323, 2005.

[90] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Wee Hyong Tok. An adaptable distributed query processing architecture. *Data Knowl. Eng.*, 53(3):283–309, 2005.

[91] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Ji Wu. Efficient dynamic operator placement in a locally distributed continuous query system. In *CoopIS*, 2006.

[92] Yongluan Zhou, Beng Chin Ooi, Kian-Lee Tan, and Feng Yu. Adaptive reorganization of coherency-preserving dissemination tree for streaming data. In *ICDE*, 2006.

[93] Yongluan Zhou and Kian-Lee Tan. Architecting a large-scale cbn-based stream processing system. Submited for Publication.

[94] Yongluan Zhou, Kian-Lee Tan, and Feng Yu. A continuous query overlay on top of a distributed publish/subscribe system. Submited for Publication.

[95] Yongluan Zhou, Kian-Lee Tan, and Feng Yu. Leveraging distributed publish/subscribe system for scalable stream processing. In *VLDB 06 Workshop on Business Intelligence for the Real Time Enterprise (BIRTE)*, 2006.

[96] Yongluan Zhou, Ying Yan, Feng Yu, and Aoying Zhou. PMJoin: Optimizing distributed multi-way stream joins by stream partitioning. In *DASFAA*, pages 325–341, 2006.

[97] Yali Zhu, Elke A. Rundensteiner, and George T. Heineman. Dynamic plan migration for continuous queries over data streams. In *SIGMOD Conference*, pages 431–442, 2004.