Abstract of "Load Management Techniques for Distributed Stream Processing" by Ying Xing, Ph.D., Brown University, May 2006.

Distributed and parallel computing environments are becoming inexpensive and commonplace. The availability of large numbers of CPU's makes it possible to process more data at higher speeds. Stream-processing systems are becoming more important, as broad classes of applications require results in real-time. Since load can vary in unpredictable ways, exploiting the abundant processor cycles requires effective load management techniques. Although load distribution has been extensively studied for the traditional pull-based systems, it has not yet been fully studied in the context of push-based continuous query processing.

Push-based data streams commonly exhibit significant variations in their rates and burstiness over all time-scales. Even though medium-to-long term load variations can be dealt with using dynamic load migration techniques rather effectively, short-term variations are much more problematic - since capturing such variations and reactively migrating load can be prohibitively expensive or even impractical. We need robust solutions whose performances do not degrade significantly in the presence of time-varying load.

In this dissertation, we present both a static operator distribution technique and a dynamic operator re-distribution technique for a cluster of stream processing engines. First, for operators that are impractical to be moved between processing engines on the fly, we propose a resilient static operator distribution algorithm that aims at avoiding overload by maximizing the feasible space of the operator distribution plan. Second, for operators with relatively small load migration overheads, we propose a correlation-based dynamic operator distribution algorithm that aims at minimizing end-to-end latency by minimizing load variance and maximizing load correlation. Our experiment results quantify the effectiveness of the proposed approaches and demonstrate that they significantly outperform traditional operator distribution approaches.

Load Management Techniques for Distributed Stream Processing

by

Ying Xing

B. Eng., Automation Engineering, Tsinghua University, 1997

Sc. M., Computer Science, Tsinghua University, 1999

Sc. M., Computer Science, Brown University, 2001

Sc. M., Applied Mathematics, Brown University, 2004

A dissertation submitted in partial fulfillment of the
requirements for the Degree of Doctor of Philosophy
in the Department of Computer Science at Brown University

Providence, Rhode Island

May 2006

© Copyright 2006 by Ying Xing

This dissertation by Ying Xing is accepted in its present form by
the Department of Computer Science as satisfying the dissertation requirement
for the degree of Doctor of Philosophy.

Date _____ _____

Stan Zdonik, Director

Recommended to the Graduate Council

Date _____ _____

Uğur Çetintemel, Reader

Date _____ _____

John Jannotti, Reader

Approved by the Graduate Council

Date _____ _____

Sheila Bonde
Dean of the Graduate School

Acknowledgements

My deepest gratitude is due to my advisor, Professor Stan Zdonik, who taught me how to do research, guided me through the problems, helped me to improve my writing, presentation and many other things, watched me in all my progress, and most importantly, encouraged me and supported me to carry on through all these years. Without him, I would never have finished this dissertation.

I am profoundly indebted to Professor Uğur Çetintemel, who inspired much of this work and contributed both to the original papers and this dissertation with a major impact. I was highly impressed by his novel ideas in research and his ability to produce clear and concise descriptions of complicated algorithms. My dissertation would not be have been the same without him.

I am also very grateful to Professor John Jannotti for his valuable advice and support. Our discussions on the comparison and evaluation of different algorithms have been of great value to this dissertation.

I want to express my deep gratitude to Jeong-Hyong Wang who collaborated with me in all these works. His suggestions are a great source of ideas and improvements. He also developed the distributed stream processing simulator for evaluating the algorithms. In addition, he is a great officemate and friend!

I would also like to thank all the rest of the Borealis team: Daniel Abadi, Yanif Ahmad, Hari Balakrishnan, Magdalena Balazinska, Brad Berg, Mitch Cherniack, Wolfgang Lindner, Sam Madden, Anurag S. Maskey, Olga Papaemmanouil, Mike Stonbraker, Alexander Rasin, Esther Ryvkina, Nesime Tabul and Wenjuan Xing. In particular, I would like to thank Nesime Tabul, who entered Brown at the same time as I did. She gave me numerous helps in all these years and I learned so much from her! I owe my special thanks to Olga Papaemmanouil who is also my great officemate. I am impressed by her responsibility for every work she did. I also love shareing with her all the joys and sadness in the lives of PhD students.

Finally, I want to say 'thank-you' to all my friends and family, wherever they are, particularly to my parents Caiyun Wang and Shulin Xing, my brother Yashuai Xing, my friends Qingyan Meng, Lijuan Cai, Ye Sun, Danfeng Yao, Jue Yan, Zong Zhang, Liye Ma, and most importantly of all, to Ya Jin, for everything!

Contents

List of Tables

List of Figures

Chapter 1

Introduction

Recently, a new class of applications has emerged in which high-speed streaming data must be processed in real-time with very low latency. Prime examples of such applications include Financial data processing (e.g., high-frequency automated trading, real-time risk management), network traffic monitoring and intrusion detection, sensor-based environment monitoring (e.g. building temperature monitoring, highway traffic monitoring), moving object tracking (e.g., battlefield vehicle tracking), RFID-based equipment tracking (e.g., theft prevention, library book tracking), medical applications (e.g., patient health condition monitoring) and multiple user real-time on-line games.

It has been widely recognized that traditional Database Management Systems (DBMSs), which have been oriented towards business data with low update frequencies, are inadequate to process large volumes of continuous data streams with high speed [20, 11, 40, 8]. As a result, a new class of Stream Data Management Systems (SDMSs) has been developed, including Aurora [4], STREAM [58], TelegraphCQ [23] and Niagara [25]. SDMS breaks the traditional paradigm of request-response style processing, in which a finite answer is generated for each query sent to the system. Instead, users register queries in an SDMS that specifies their interests over potentially unbounded data streams. Based on these queries, SDMS continuously processes the incoming data streams from external data sources and delivers potentially unbounded results to the user. These specialized queries are also called *continuous queries*.

The *Aurora Stream Processing Engine*, which was developed at Brandies University, Brown University and M.I.T., is a data-flow-based continuous query processing engine. In Aurora, queries are represented by directed acyclic data-flow diagrams, as shown in Figure 1.1. Such queries are also called *query graphs* or *query networks*. The boxes in a query graph represent continuous query operators, which process and refine the input steams continuously to produce results for waiting applications. The arrows in a query graph represent data streams flowing through the continuous query operators. These operators are typically modifications of the familiar operators of the relational algebra to deal with the ordered and infinite nature of streams. In Aurora, queries are pre-registered in the processing engine; input data streams are pushed to the system from external data sources and get processed when they pass through the query graphs.

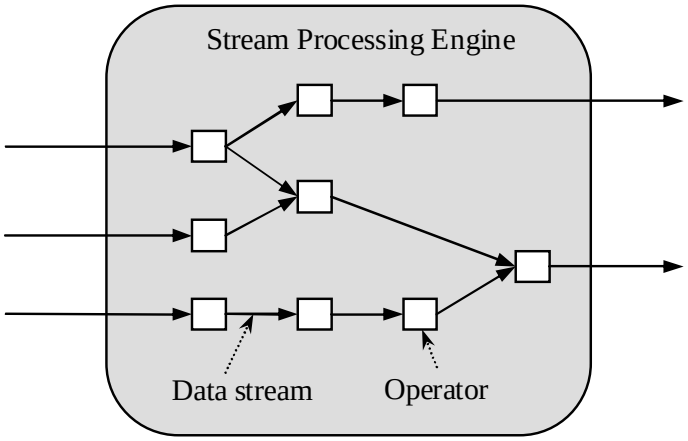


Figure 1.1: Stream processing query graph.

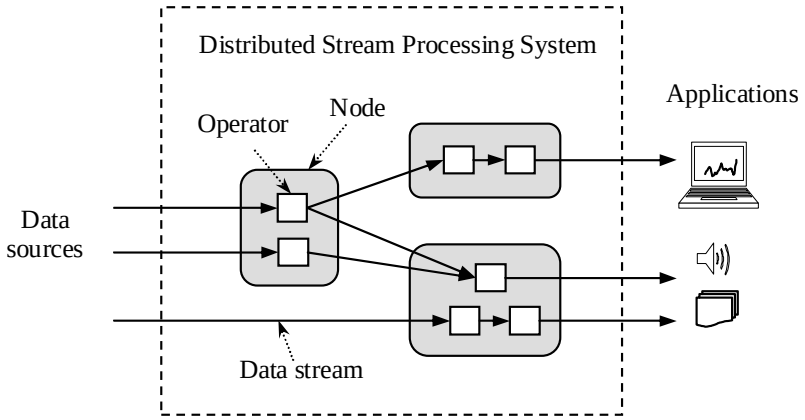


Figure 1.2: Distributed Stream Processing.

In stream processing domains, one observes increasing stream rates as more and more data is captured electronically, putting stress on the processing ability of stream processing systems. At the same time, the utility of results decays quickly demanding shorter and shorter latencies. Clusters of inexpensive processors allow us to bring distributed processing techniques to bear on these problems. Consequently, Distributed Stream Processing Systems (DSPSs) become a natural extension to the centralized stream processing engines to enable the scalability and availability that stream-based applications demand [26, 73, 13, 92].

Borealis [3] is a follow-on project of the Aurora system. It extends the core stream processing functionality from Aurora and enables queries to be processed in a distributed fashion. Figure 1.2 illustrates a typical configuration of the Borealis DSPS in which a query network is distributed across multiple machines. A Borealis DSPS contains multiple Borealis nodes (machines) with each node running a single Aurora engine. These nodes collaborate with each other for the benefit of the whole system. For example, if one node become overloaded, some operators can be moved out

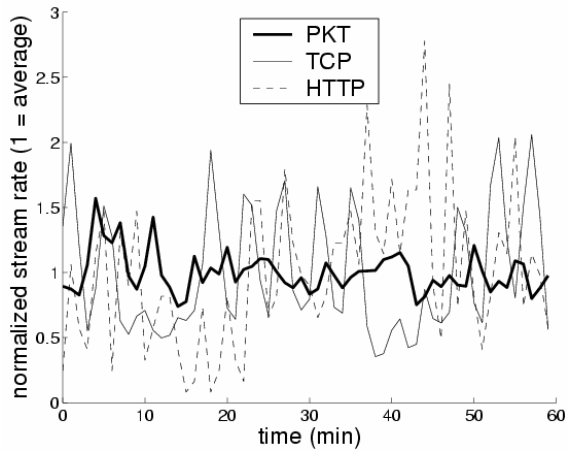


Figure 1.3: Stream rate variations in network traffic traces.

of the overloaded node to lightly loaded nodes.

The specific operator distribution plan has an enormous impact on the performance of the resulting system. As the load changes, this mapping will need to change in order to deal with new hot spots. The process of forming the initial mapping and of dynamically redistributing operators is the topic of this dissertation.

1.1 Challenges

While load distribution has been studied extensively in traditional parallel and distributed systems [34, 42, 15], the operator distribution problem has not yet been fully studied in the context of push-based stream processing. Distributed stream processing systems have two fundamental characteristics that differentiate them from traditional parallel database systems. First, stream processing tasks are long-running continuous queries rather than one-time parallel queries. In traditional parallel systems, the optimization goal is often minimizing the completion time of a finite task. In contrast, a continuous query has no completion time; therefore, we are more concerned with the latency of individual results.

Second, the data in stream processing systems is pushed from external data sources. Load information needed by task allocation algorithms is often not available in advance, not controllable, and varies significantly and over all time-scales. Medium and long term variations arise typically due to application-specific behavior; e.g., flash-crowds reacting to breaking news, closing of a stock market at the end of a business day, temperature dropping during night time, etc. Short-term variations, on the other hand, happen primarily because of the event-based aperiodic nature of stream sources as well as the influence of the network interconnecting data sources. Figure 1.3 illustrates such variations using three real-world traces [1]: a wide-area packet traffic trace (PKT), a wide-area TCP connection trace (TCP), and an HTTP request trace (HTTP). The figure plots

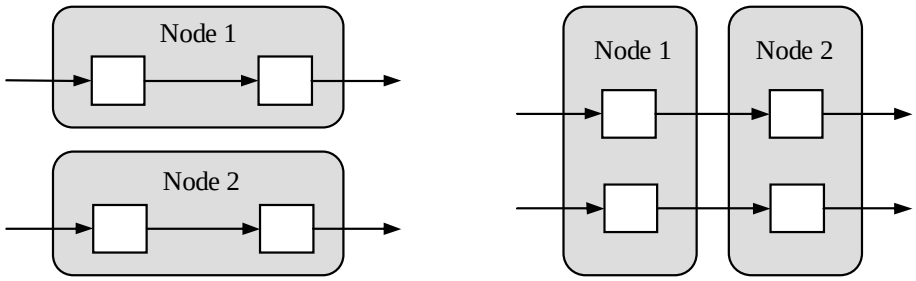
the normalized stream rates as a function of time and indicates their standard deviation. Note that similar behavior is observed at other time-scales due to the self-similar nature of these workloads [30, 51].

A common way to deal with time-varying, unpredictable load variations in a distributed setting is dynamic load distribution. This approach can effectively handle medium-to-long term variations as (1) these persist for relatively long periods of time and are thus rather easy to capture, and (2) the overhead of load redistribution is amortized over time. Neither of these properties holds in the presence of short-term load variations. Capturing such transient variations requires frequent statistics gathering and analysis across distributed machines. Reactive load distribution requires operator migration, a process which involves multi-node synchronization and state migration. As a concrete example of the cost of this process, in Borealis DSPS, run-time operator migration for stateless operators takes on the order of few hundreds of milliseconds in a LAN environment. Operators with larger state size, such as aggregate operators with many intermediate results and SQL read/write operators, may take much longer migration time depending on how much data has to be transferred. Even worse, some systems do not even provide support for dynamic migration and thus need to interrupt processing to re-distribute the operators. After an operator migration process finishes, the load burst that triggers this process might have disappeared. And very likely, another load burst may cause these operators to be moved again. As a result, dealing with short-term load fluctuations by moving operators frequently can be prohibitively expensive or even impractical.

Moreover, even for medium-to-long term load variations and operators with relatively small migration overheads, we should avoid moving operators as much as possible. Moving operators from one node to another node often incurs undesirable performance during the migration process because the execution of the operators to be moved needs to suspended and all their input data items must be delayed in this process. The jitters in the end-to-end latency caused by operator migration can be very bad for time-sensitive applications such as missile tracking and real-time stock trading. Therefore, we need a robust operator distribution plan that can sustain load bursts without moving operators or, at least, minimize the use of operator migration.

On the other hand, traditional dynamic load balancing strategies that only consider the average load of the processing operators do not yield good performance in the presence of bursty or fluctuating input stream rates. These algorithms are designed for pull-based systems where load fluctuation occurs as different queries are sent to the system. In a push-based system, load fluctuation occurs in the arrival rates of the streams. In this case, even when the average load of a node is not very high, a node may experience a temporary load spike and data processing latencies can be significantly affected by the duration of the spike.

Example 1: For instance, consider two operator chains with bursty input data. Let each operator chain contain two identical operators with a selectivity of one. When the average input rates of the two input streams are the same, the average load of all operators are the same. Now



(a) Connected plan. (b) Cut plan.

Figure 1.4: Two operator distribution plans of the same query graph.

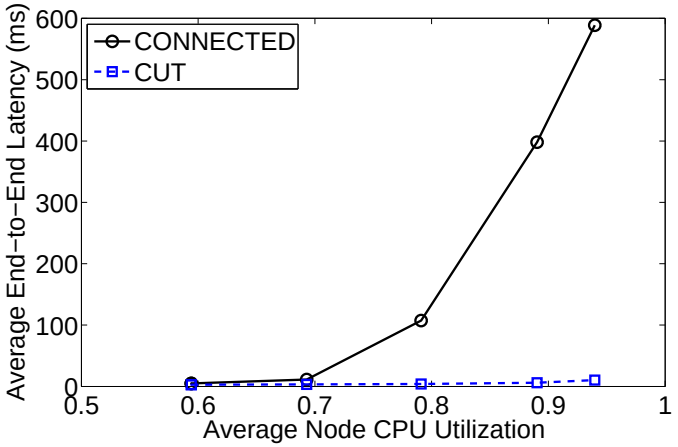


Figure 1.5: Comparison of different operator mapping plans with fluctuating load.

consider two operator mapping plans on two processing nodes. In the first plan, we put each of the two connected operator chains on the same node (call this the *connected plan*, as shown in Figure 1.4(a)). In the second plan, we place each component of a chain on different nodes (call this the *cut plan*, as shown in Figure 1.4(b)). Assume that the CPU overhead for data communication is negligible, and bandwidth is not a bottleneck of the system. There is no difference between these two plans if we only want to balance the average load of the two nodes. However, suppose the load burst of the two input streams happens at different times, i.e., when the input rate of the first chain is high, the input rate for the second chain is low and vice versa. Then very different performances will result from these two mapping plans. Figure 1.5 shows an example performance graph for this kind of workload in which the burst duration and the normal duration are both 5 seconds, and the high (bursty) input rate is twice the low (normal) input rate.

Putting connected operators on different nodes, in this case, achieves much better performance than putting them on the same node (ignoring bandwidth considerations for now). The main

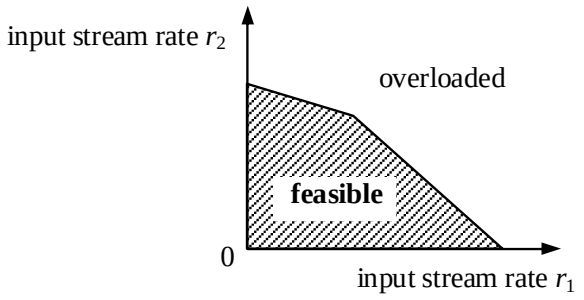


Figure 1.6: Feasible set on input stream rate space.

difference between these two mapping plans is that since the two input bursts are out of phase, the cut plan ensures that the load variation on each node is very small. In the connected plan, it is much larger. This simple example shows that the average load level is not the only important factor in load distribution. The variation of the load is also a key factor in determining the performance of a push-based system. Thus, to minimize data processing latencies we need an approach that can avoid temporary overload and minimize load variance on each node as much as possible.

1.2 Contributions

In this dissertation, we propose two novel operator distribution algorithms to cope with bursty and unpredictable load variations. One of the algorithms focuses on the static distribution of the operators with relatively large migration overheads (e.g., SQL read/write operators). We call these operators "sticky" operators. The optimization goal of this algorithm is to keep the system feasible for unpredictable load variations without the action of operator migration. The other algorithm focuses on the initial distribution and dynamic re-distribution of the operators with relatively small migration overheads (e.g., Filter, Union operators). We call these operators "floating" operators. The optimization goal of this algorithm is to minimize the average end-to-end data processing latency and avoid frequent operator migration under bursty workloads.

1.2.1 Static Operator Distribution Contributions

For the static distribution of sticky operators, we seek a resilient operator distribution plan that does not become overloaded easily in the presence of unpredictable input rates. Standard load distribution algorithms optimize system performance with respect to a single load point, which is typically the load perceived by the system in some recent time period. The effectiveness of such an approach can become arbitrarily poor and even infeasible when the observed load characteristics are different from what the system was originally optimized for. Resilient distribution, on the other hand, does not try to optimize for a single load point. Instead, it enables the system to "tolerate" a large set of load points without operator migration.

More specifically, we model the load of each operator as a function of the system's input

stream rates. For given input stream rates and a given operator distribution plan, the system is either feasible (none of the nodes are overloaded) or overloaded. The set of all feasible input rate combinations defines a *feasible set*. Figure 1.6 illustrate an example of a feasible set for two input streams. For unpredictable workloads, we want to make the system feasible for as many input rate points as possible. Thus, the optimization goal of resilient operator distribution is to maximize the size of the feasible set.

In general, finding the optimal operator distribution plan requires comparing the feasible set size of different operator distribution plans. This problem is intractable for a large number of operators or a large number of input streams. In this dissertation, we present a greedy operator distribution algorithm that can find suboptimal solutions without actually computing the feasible set size of any operator distribution plan.

More specifically, we formalize the resilient operator distribution problem for systems with linear load models, where the load of each operator can be expressed as a linear function of system input stream rates. We identify a tight superset of all possible feasible sets called the *ideal feasible set*. When this set is achieved, the load from each input stream is perfectly balanced across all nodes (in proportion to the nodes' CPU capacity). As a result, in this case, the loads of all nodes are always perfectly balanced no matter what the system's input stream rates are.

The ideal feasible set is in general unachievable, because the load of each input stream can not be arbitrarily partitioned. We propose two novel operator distribution heuristics to make the achieved feasible set as close to the ideal feasible set as possible. The first heuristic tries to balance the load of each input stream across all nodes. The second heuristic focuses on the combination of the "impact" of different input streams on each node to avoid creating system bottlenecks. We then present a resilient operator distribution algorithm that seamlessly combines both heuristics.

In addition, we present algorithm extensions that take into account the communications costs and knowledge of specific workload characteristics (i.e., lower bound on input stream rates) to optimize system performance.

Furthermore, we present a generalization of our approach that can transform a nonlinear load model into a linear load model by introducing new variables. Using this transformation, our resilient algorithm can be applied to any system.

Our study is based on extensive experiments that evaluate the relative performance of our algorithm against several other load distribution techniques. We conduct these experiments using both a simulator and the Borealis prototype on real-world network traffic traces. The results demonstrate that our algorithm is much more robust to unpredictable or bursty workloads than traditional load distribution algorithms.

1.2.2 Dynamic Operator Distribution Contributions

For the floating operators, we propose a correlation-based dynamic operator distribution algorithm that redistributes operators adaptively according to the most recent load patterns. Traditional dynamic load distribution algorithms balance the average load of processing nodes to eliminate hot

spots. However, for a push-based system with bursty workload, even when the average load of a node is not very high, a node may experience a temporary load spike and data processing latencies can be significantly affected by the duration of the spike. The correlation-based operator distribution algorithm, on the other hand, not only balances the average load of the nodes dynamically, but also minimizes the load variance on each node. The latter goal is achieved by exploiting the ways in which the stream rates correlate across the operators.

More specifically, we represent operator and node loads as fixed length time series. The correlation of two load time series is measured by the correlation coefficient, which is a real number between -1 and 1. Its intuitive meaning is that when two time series have a positive correlation coefficient, then if the value of one time series at certain index is relatively large (in comparison to its mean), the value of the other time series at the same index also tends to be relatively large. On the other hand, if the correlation coefficient is negative, then when the value of one time series is relatively large, the value of the other tends to be relatively small. Our algorithm is inspired by the observation that if the correlation coefficient of the load time series of two operators is small, then putting these operators together on the same node helps in minimizing the total load variance in the system. On the other hand, if the correlation coefficient of the load time series of two operators is large, then separating the operators to different nodes is better.

The intuition of correlation is also the foundation of the other idea in our algorithm; when making operator allocation decisions, we try to maximize the correlation coefficient between the load time series of different nodes. When the load correlation coefficient between two nodes is large, if the load of one node increases, the load of the other node also tends to increase, and if the load of one node decreases, the load of the other node also tends to decrease. As a result, the load of the two nodes are naturally balanced without operator migration. By maximizing the average load correlation between all node pairs, we can minimize the number of load migrations needed.

Later, we will see that minimizing the average load variance also helps in maximizing the average load correlation, and vice versa. Thus, the main goal of our load distribution algorithms is to produce a balanced operator mapping plan where the average load variance is minimized or the average node load correlation is maximized. Finding the optimal solution for such a problem requires exhaustive search and is, similar to the graph partitioning problem, NP hard [38]. In this dissertation, we propose a greedy algorithm that finds a sub-optimal solution in polynomial time. The key idea of our algorithm is to favor putting an operator on a node if the operator and the node have a small load correlation coefficient, and to avoid putting an operator on a node if they have a large load correlation coefficient.

We present both a global operator mapping algorithm and some pair-wise load redistribution algorithms. The global algorithm is mainly used for the initial placement of floating operators (after the static placement of the sticky operators). After global distribution, we will use pair-wise algorithms to move floating operators to adapt to load changes. The advantage of using pair-wise algorithms is that it does not require as much load migration as the global algorithm.

Our experimental results show that the performance of our algorithm is very close to the

optimal solution and the correlation based dynamic operator distribution algorithm significantly outperforms the traditional dynamic operator distribution algorithms under a bursty workload.

1.3 Organization

The remainder of this dissertation is organized as follows. In Chapter 2, we present an overview of the Aurora/Borealis stream processing prototype. In addition, we introduce the load models and other definitions and notations used throughout this dissertation. In Chapter 3, we formalize the static and dynamic operator distribution problem for distributed stream processing. We represent the detail load management techniques for static operator distribution and dynamic operator distribution algorithm in Chapter 4 and Chapter 5 respectively. We discuss the experimental results in Chapter 6 and present the related work in Chapter 7. Finally, in Chapter 8, we conclude and discuss possible areas for future work.

Chapter 2

Background

Our work is an integrated part of the Aurora/Borealis stream-processing project. In this chapter, we provide an overview of the Aurora Data Stream Processing Engine and the Borealis Distributed Stream Processing System.

2.1 The Aurora Stream Processing Engine

Aurora is a data stream processing engine on a single machine that is designed to process continuous queries over unbounded data streams. It is one of the pioneers of stream data management systems that were developed in recent years. In what follows, we provide an overview of the major features of Aurora.

2.1.1 Data Model

In Aurora, a data stream is defined as a sequence of data items ordered by some attributes of the data or by the sequence number of the data. Each data item (also called a *tuple*) in a stream constrains several fields called *attributes*. The data type and length of each attribute is defined as the *schema* of the stream. All data items of a stream must have the same schema. A data stream is an append-only data structure in that no data item can be deleted from a stream and newly arrived data are always appended to the end of the stream.

2.1.2 Query Model

Aurora uses acyclic boxes-and-arrows-based data-flow diagrams to represent data processing tasks, with boxes representing continuous query operators and arrows representing data streams. The output steam of an operator can split to multiple data streams and go to different downstream operators. Some operators have multiple input streams.

To make the applications easy to write, Aurora provides a set of primitive operators that can filter, aggregate, transform and merge input data streams to produce output of interest. In

addition, it also supports user defined operators to accomplish more complicated tasks. We now describe the core Aurora operators that are commonly used in stream processing applications. More detailed description of these operators appears in Abadi *et al.* [4].

- *Filter* extends the Selection operator in the relational algebra. The simplest Filter operator applies predicates to input tuples and outputs those that satisfy the predicates. It can also act like a switch statement that routes input tuples to alternative streams based on non-overlapping predicates.

- *Map* is a generalized Projection operator in the relational algebra. It transforms an input tuple into an output tuple by applying a function to the attributes of the input tuple.

- *Union* merges two or more input streams into a simple output stream. All its input streams must have the same schema. Union does not enforce any order on the output stream. Tuples are usually processed first-in-first-out.

- An *Aggregate* operator applies an aggregate function such as sum, count, average, maximum and minimum on sub-sequences of its input stream. Because data streams are potentially unbounded and stream processing operators cannot hold infinitely many data items to produce output results, Aurora defines *windows* over data streams so that operators can be applied on subsequences of data streams instead of entire data streams. A window on a data stream is defined as an interval over the ordering attributes of the stream or over the sequence number of the stream. A *sliding window* is defined as a sequence of windows with the same interval size but different starting values. The difference between the starting times of adjacent windows in a sliding widow is called the *advance step*. For example, suppose stream S has timestamp attribute T and another attribute A. If we want to know the average value of attribute A over the most recent one minute every second, we can use an Aggregate operator that computes the average value of A over a sliding with size one minute and advance step one second.

  The input stream of an Aggregate operator can be partitioned into sub-streams using the *group by* statement. Each sub-stream has its own sliding window. A window is opened in an Aggregate operator when the first tuple that falls into this window arrives. It is closed after all tuples in this window arrive, or when this window times out (the longest time a window can stays open is specified by the *timeout* statement.). An output tuple is generated when a window is closed. Before all the tuples of an open window arrive, an Aggregate operator must keep certain information about this open window (usually a summary of all the tuples that have arrived) in order to produce the output tuple. These data are collectively called the *state* of the Aggregate operator. The state size of an Aggregate operator is usually proportional to the number of open windows of the operator.

- *Join* extends the Join operator in the relational algebra. To deal with the unbounded nature of data streams, it enforces a window on the tuple pairs that can be matched against each

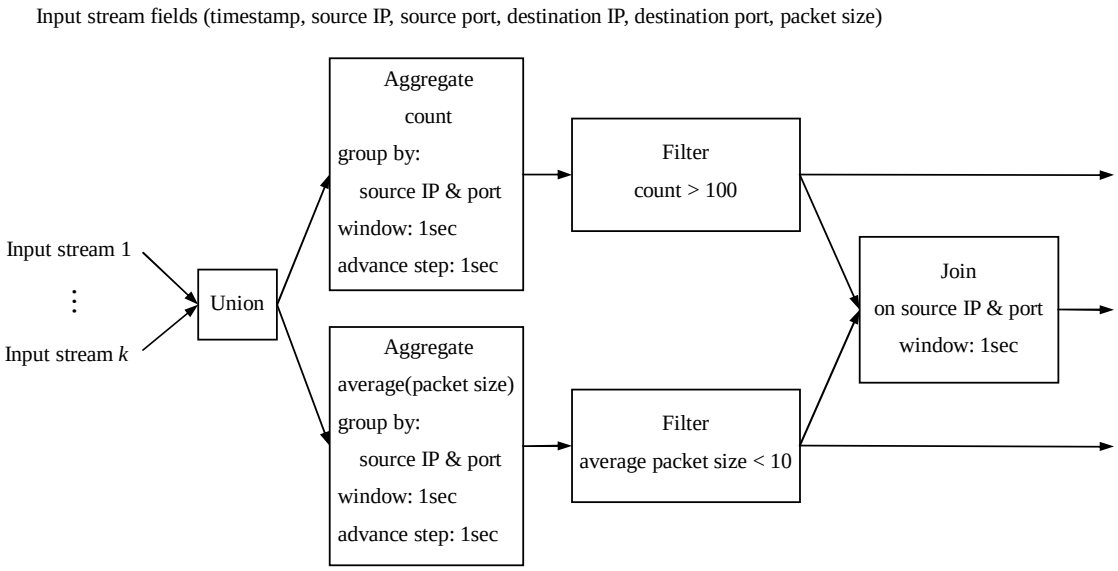


Figure 2.1: Network traffic monitoring query example.

other. For example, suppose a Join operator with predicate P has input stream S_1 ordered by attribute A, and input stream S_2 ordered by attribute B. It defines a window w on the ordering attributes. If tuple u from S_1 and tuple v from S_2 satisfy the condition

$$|u.A - v.B| \leq w$$

and predicate P, then an output tuple is produced by concatenating u and v. A Join operator must keep tuples that previously arrived available to match with tuples that will arrive in the future. A tuple can be discarded only if no future tuples can match with this tuple. The state of a Join operator consists of all history tuples that are kept in the operator. The state size of a Join operator is usually proportional to the product of its window size and input stream rates.

- *SQL-read* and *SQL-write* operators query and update relational database tables for each tuple received. The database tables can be considered as the states of these operators.

EXAMPLE 2: Figure 2.1 illustrates a network traffic monitoring query diagram, which is composed of commonly used operators introduced above. The input streams of this query diagram are the network packet traces collected from different network traffic monitors. Each input tuple contains the following fields: timestamp, source IP address, source port, destination IP address, destination port and packet size. The input streams from different monitors are merged together by a Union operator. The output of the Union operator splits into two streams. One streams goes to an aggregate operator that counts the number of packets from each source endpoint (IP and port) in each second. The other stream goes to an aggregate operator that computes the average

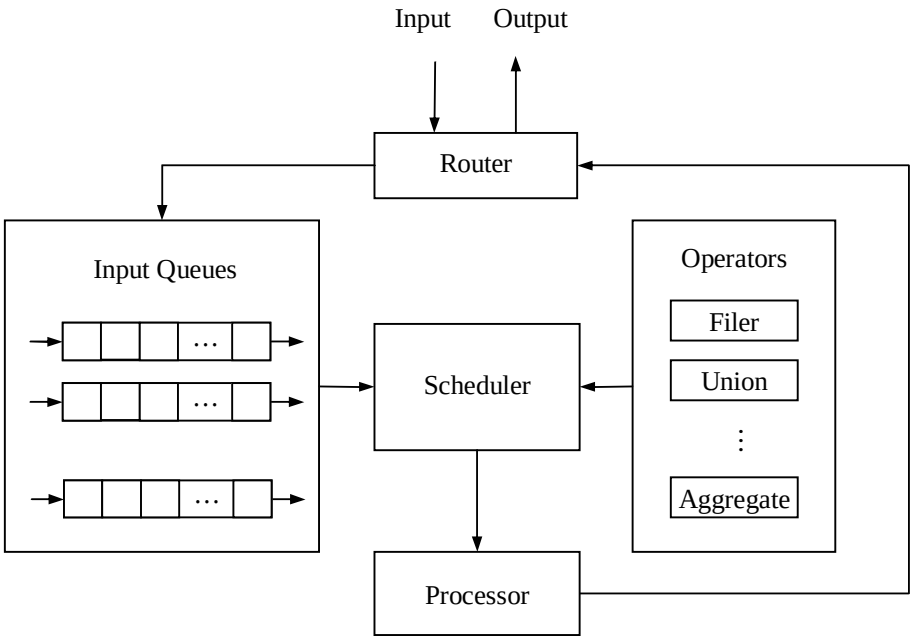


Figure 2.2: Aurora stream processing model.

packet size of each source endpoint in each second. The Filter operator following the first Aggregate operator is used to output the source endpoints that send many packets in each second. The Filter operator following the second Aggregate operators is used to output the source endpoints that send small packets on average in each second. The Join operator matches the output tuples of the two Filter operators to output the source endpoints that send many small packets in each second.

2.1.3 Processing Model

Figure 2.2 illustrates the basic stream processing model in Aurora. When an input tuple arrives in the system, it is first put into the input queue of the operator that processes this tuple. All operators with non-empty input queues are eligible for processing. The scheduler determines the processing sequence of all eligible operators based on certain scheduling policy [21]. The output tuple of an operator is either routed back to the input queues of its downstream operators or pushed out of the system.

2.2 The Borealis Distributed Stream Processing System

Borealis is a second-generation distributed stream processing engine. It inherits core stream processing functionality from Aurora and distribution functionality from Medusa [2, 95]. It extends both systems with the ability to

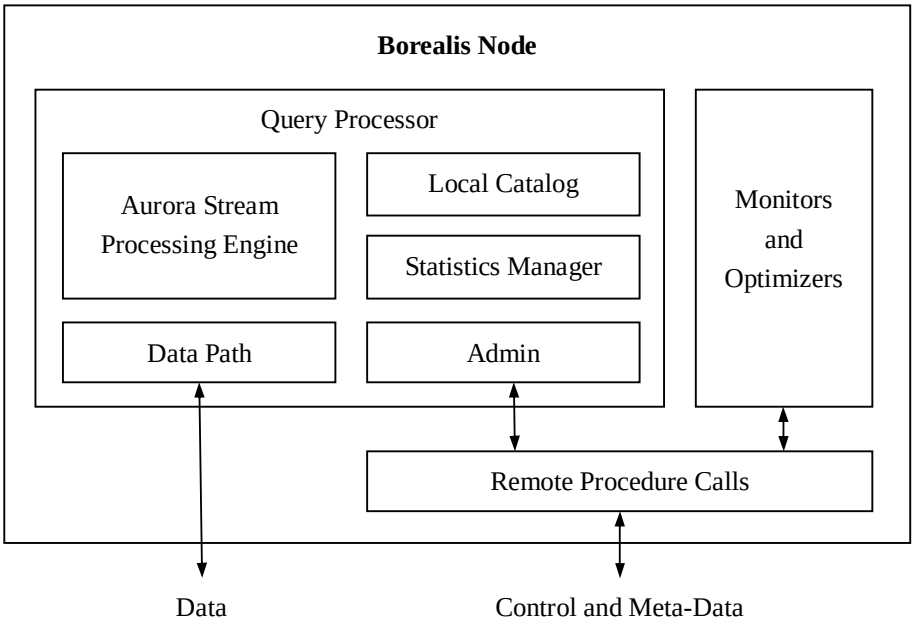


Figure 2.3: Borealis node architecture.

- Distribute queries across multiple machines and operates collaboratively.
- Support dynamic load migration by moving operators from one machine to another machine on the fly.
- Tolerate node and network failures and accomplish fast recovery.
- Integrate various optimization techniques such as load management, load shedding and high availability in a single framework.

More features and optimization techniques of Borealis are presented in Abadi *et.al.* [3]. In what follows, we present the high level architecture of the Borealis system to set the context of the work of this dissertation.

2.2.1 Borealis Node

The core components of the Borealis Distribute Stream Processing System are the Borealis nodes. Each Borealis node is a single stream processing engine on a physical machine. It collaborates with its peer Borealis nodes as well as other components in the system through Remote Procedure Calls (RPCs) [17].

Figure 2.3 illustrates the major components of a Borealis node. The actual stream processing takes place in the *Query Processor*. We now briefly introduce each of its components:

- The *Data Path* component is responsible for the data communication of the Borealis Node. It receives data streams from data sources or other Borealis nodes. It also sends data stream to application clients or other Borealis nodes.

- *Admin* is an interface between the local Query Processor and all other Borealis nodes, clients, and other components in the system. It handles incoming requests in the form of Remote Procedure Calls. These requests can modify the local query diagram by adding, deleting or moving operators. The Admin component handles all the detailed steps of these modifications.

- The *Local Catalog* holds the meta-data of the local query diagram. The meta-data include stream schemas, operator and stream descriptions and the topology of query diagrams.

- The *Statistics Manager* holds the runtime statistics about the local stream processing engine, such as data rates, CPU utilization of various operators etc.. Other Borealis nodes or other components in the system can request these statistics through the Remote Procedure Calls.

- The embedded Aurora stream processing engine is the core piece of a Borealis node. It receives input streams through the Data Path component, processes these data streams, and sends output streams to other nodes or clients through the Data Path component. It also collects runtime statistics and stores them in the Statistics Manager.

Other than the Query Processor, a Borealis node also has *monitors*, which monitor the state of the local Borealis node and its peer Borealis nodes, and *optimizers*, which communicate with their peers to take collaborative optimization actions. All the control messages between these components and between the Borealis nodes go through a transport independent RPC layer which enables the components to communicate in the same manner no matter whether they are local or remote.

2.2.2 System Architecture

Borealis nodes can be configured to fit different processing environments with processing elements as small as sensors and as large as servers. In this dissertation, we assume a physical architecture of a loosely coupled shared-nothing computer cluster. Each Borealis node runs on a server of the cluster. All Borealis nodes are connected by a high bandwidth local network.

Figure 2.4 illustrates a typical setup of a Borealis stream processing cluster with a centralized optimizer that coordinates the processing and resource allocation of the Borealis nodes. The *Global Optimizer* may contain components other than the ones shown in Figure 2.4. In this dissertation, we we only present the components that are related to the load management problem below:

- As its name indicates, the *Proxy* component acts as a proxy between users and Borealis nodes. Users send queries to the system through the Proxy server. They can also specify how to deploy the queries initially. The Proxy server parses the queries and deploys the operators over the Borealis nodes.

- The *Global Catalog* is part of the *Global Optimizer*. It holds the meta-data about all query diagrams in the system. Whenever an operator is added or removed from a node, the Local

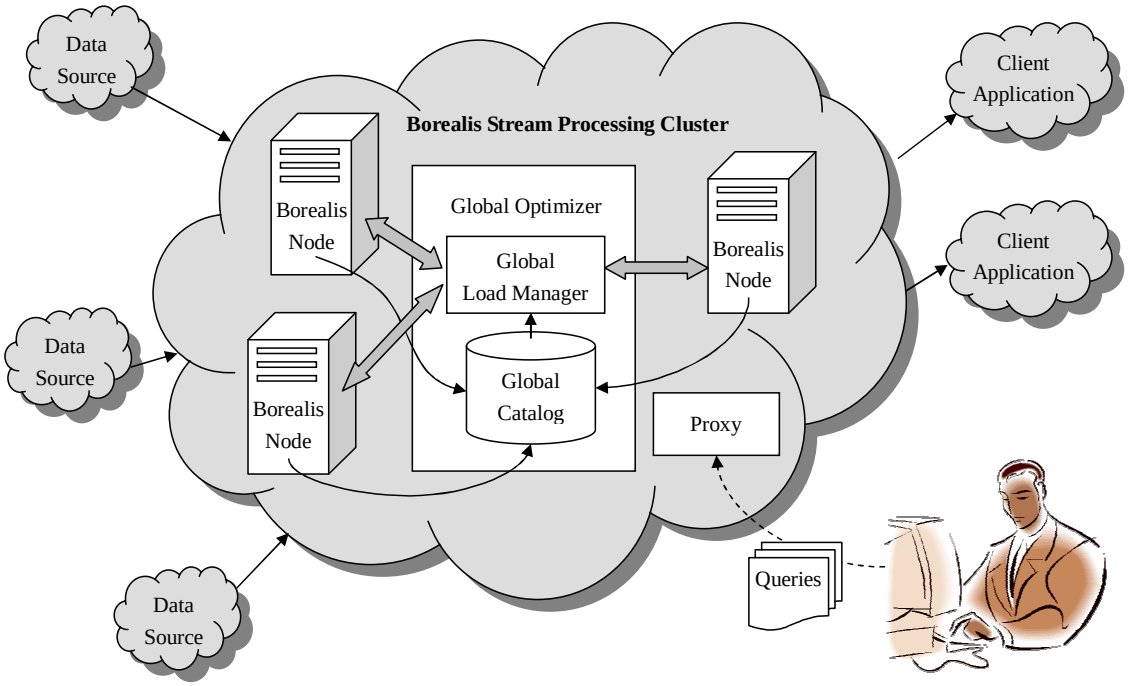


Figure 2.4: Borealis system architecture.

Catalog of the node sends an update message to the Global Catalog. Thus, the Global Catalog always contains the complete description of all the operators in the system as well as their locations with some delay.

- The *Global Load Manager* is part of the Global Optimizer. It reads the topology and deployment information of the query diagrams from the Global Catalog. It also requests runtime statistics of each node from the nodes' Statistics Manager components. After gathering all the necessary information in a warm up period, the Global Load Manager determines a new initial operator distribution plan. It then moves operators between the nodes according to the new plan. If there are operators that can not be moved between the nodes, it simply restarts the system with the new operator distribution plan as the initial plan. The Global Load Manager also monitors the load of the nodes and operators during the entire stream processing process. Periodically, it determines whether some operators should be moved from one node to another node. The algorithms that the Global Load Manager follows to place the opreators initially and to re-distribute the operators on the fly are introduce in the remainder of this dissertation.

2.2.3 Operator Migration

In Borealis, most operators (e.g., Filter, Map, Union, Aggregate, Join) provide interfaces that allow them to be moved on the fly. For practical purposes, we consider SQL-read and SQL-write

Figure 2.5: Operator migration time for moving a set of stateless operators.

operators to be immovable since their effective state can be huge. We also avoid moving Aggregate and Join operators if their state sizes are very large.

To move a set of operators from one node to another node, the Global Load Manager sends a PRC request to the sending node telling it to move a set of boxes to the receiving node. The sending node then coordinates with the receiving node through RPC calls for the rest of the operator migration process. The migration operation applies through the following steps:

1. The sending node first suspends the execution of the operators to be moved. It then serializes the states of these operators. After serializing all operators, it creates a package that contains the meta-data of the operators to be moved, the meta-data of the streams that connect to these operators, and the state of the operators. Then, it sends the whole package to the receiving node.

2. After receiving all the data, the receiving node first adds the meta-data of the operators and streams to its Local Catalog. Then it instantiates these operators with the given states and establishes data connection to appropriate nodes. After the execution of these operators can be resumed, the receiving node sends a request to the sending node telling it that it is now safe to remove the operators.

3. Upon receiving this message, the sending node deletes those suspended operators locally and updates its local catalog.

To avoid simultaneous operator migration among the nodes, which may cause inconsistency in the system, after moving operators between a node pair, the Global Load Manager waits sufficiently long before it schedules operator migration between the next node pair.

The operator migration process is not trivial because it involves multiple Remote Procedure Calls, serialization and deserialization of operators, modification of Local Catalogs and internal

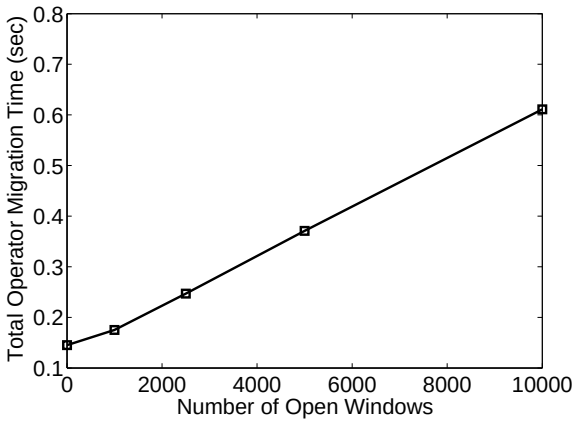


Figure 2.6: Operator migration time for moving a single aggregate operator with different state sizes.

data structures of query processors. Figure 2.5 shows the operator migration time to move a set of stateless operators from one node to another node in one single operator migration process in the Borealis prototype. Obviously, there is a linear relationship between the number of operators moved in this process and the operator migration time. The basic communication overhead between the two nodes takes about 60 milliseconds. Packing and unpacking the information of each stateless operators takes an additional 80 milliseconds.

The operator migration time for stateful operators usually depends on the state size of the operators. Figure 2.6 shows the operator migration time to move one aggregate operator from one node to another node with different state size. The x-axis shows the number of open windows of the aggregate operator being moved, which is in proportion to the state size of the operator. Figure 2.6 indicates that there is a linear relationship between the state size of the operator and its operator migration time.

Figure 2.7 shows the operator migration time to move different number of aggregate operators in one single operator migration process. Each of the aggregate operators has 10000 open windows. From Figure 2.7, we can see that the operator migration time to move a set of operators together is very close to the total operator migration time to move each of the operator separately. This is because the packing and unpacking time of each single operator dominates the basic communication time between the two nodes.

Note that operators with very large state size are very common in stream processing applications. For example, in network traffic monitoring applications, aggregate operators are often used to count the number of packets or compute the average packet size of packets from different sources. The number of groups in such aggregate operators can be very large. We fed a network traffic trace obtained from the Internet Traffic Archive [18] to an aggregate operator, which computes the average packet size grouped by source and destination IP address. The operator migration time of this operator on the trace takes about 1.4 second.

Because the execution of the operators to be moved are suspended during the migration process,

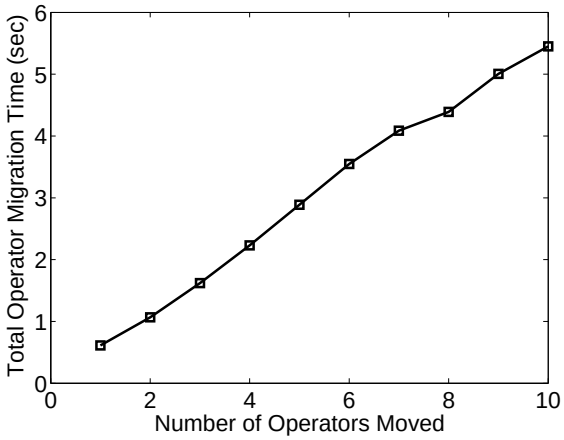


Figure 2.7: Operator migration time for moving a set of stateful operators.

moving operators can results in delays varying from several hundred milliseconds to several seconds. A lot of applications are very sensitive to such large delays in the output tuples. Therefore, avoiding moving operators frequently without scarifying performance is a crucial problem in distributed stream processing.

Chapter 3

Load Model and Problem Statement

In this chapter, we formally present the load management problems addressed by this dissertation.

3.1 Load Models

Before presenting the problems, we first introduced the basic definitions, notations, and the load models our problems are based on.

3.1.1 Basic Definitions and Notations

We assume there are n nodes (denoted by $N_i, i = 1, \cdots, n$), m operators (denoted by $o_j, j = 1, \cdots, m$), and d input streams (denoted by $I_k, k = 1, \cdots, d$) in the system.

Because the runtime statistics of the operators and nodes vary over time, we first define the average statistics based on a given time interval below:

- The *stream rate* is defined as the number of data items (tuples) that arrive at a data stream in the given time interval divided by the duration of the time interval.
- The *cost* of an operator is defined as the number of CPU cycles needed to process the input tuples arrived in the given time interval divided by the number of input tuples (or the number of tuple pairs for Join) arrived in that interval. It is also the average number of CPU cycles needed per tuple (or tuple pair for Join).
- The *selectivity* of an operator with a single input stream and a single output stream is defined as the number of output tuples of the operator divided by the number of input tuples of the operator in the give time interval. In general, an operator may have multiple input streams (e.g. Union, Join) and/or multiple output streams (e.g. switch style Filter). The selectivity

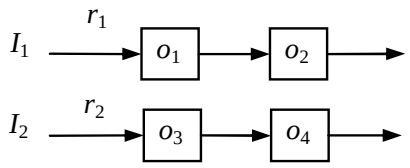


Figure 3.1: Example query graph (a).

of a Union operator is always one. The selectivity of a Join operator is defined as the number of output tuples divided by the number of input tuple pairs in the given time interval. Other operators with multiple inputs and outputs has multiple selectivities with each selectivity defined on each input-output stream pair as the number of output tuples divided by the number of input tuples. For the ease of presentation, in this dissertation, we assume that all the operators in the system are either Union operators or operators with with single input and single output, unless otherwise stated.

- The *CPU capacity* of a node is defined as the number of CPU cycles available for stream processing per unit time.
- The *load* of an operator is defined as the number of CPU cycles needed to process all input tuples of the operator in the given time interval divided by the duration of the time interval. It can be computed as the product of the input stream rate of the operator and cost of the operator.
- The *load* of a node is defined as the number of CPU cycles needed to process all input tuples of the node in the given time interval divided by the duration of the time interval. It can be computed as the sum of load of all operators allocated at the node.

We denote the the stream rate of system input stream I_k by r_k, denote the selectivity of operators o_j by s_j, denote the cost of operator o_j by c_j, denote the CPU capacity of node N_i by C_i, denote the load of operator o_j by $l(o_j)$ and denote the load of node N_i by $l(N_i)$. These notations are summarized in Table 3.1

To measure these statistics, the scheduler of the of the query processor in each Borealis node keeps track of number of CPU cycles used by each operator and the number of input and output tuples of each operator in each statistics measurement interval. The scheduler then computes the stream rates, operator costs and selectivities periodically and stores the statistics in the Statistics Manager of the local node. The global load manager periodically collects statistics from all nodes and the system. Using the system input rates, cost and selectivies of all operators, the global load manager can compute the load of all operators and nodes in the system.

Example 3: For instance, consider the simple query graph shown in Figure 3.1. The load of

Table 3.1: Notations used throughout the dissertation.

n	number of nodes
m	number of operators
d	number of system input streams
N_i	the ith node
o_j	the jth operator
I_k	the kth input stream
C_i	the CPU capacity of node N_i
r_k	the stream rate of system input stream I_k
c_j	the cost of operator o_j
s_j	the selectivity of operator o_j
$l(o_j)$	the load of operator o_j
$l(N_i)$	the load of node N_i

the operators can be easily computed as

$$\begin{aligned} l(o_1) &= c_1 r_1 \\ l(o_2) &= c_2 s_1 r_1 \\ l(o_3) &= c_3 r_2 \\ l(o_4) &= c_4 s_3 r_2 \ . \end{aligned}$$

3.1.2 Linear Load Model

We say a system has a *linear load model* if the load of all operators in the system can be written as linear functions, i.e.

$$l(o_j) = l_{j1} x_1 + \cdots + l_{jd} x_d, \quad j = 1, \cdots, m,$$

where $x_1, \cdots, x_d$ are variables and l_{jk} are constants, for $j = 1, \cdots, m, k = 1, \cdots, d$.

For instance, in Example 3, if c_1 to c_4 and s_1, s_2 are constant, then the system has a linear load model with variable r_1 and r_2.

For simplicity of exposition, we first assume that the system input stream rates are variables and the operator costs and selectivities are constant when we study static operator distribution algorithms. Under this assumption, all operator load functions are linear functions of system input stream rates. Assuming stable selectivity, operators that satisfy this assumption include Union,

Map, Aggregate and Filter, etc. We relax this assumption and discuss how to transform a non-linear load model into a linear load model in Section 4.4.2, in which we consider operators with varying costs and selectivities, as well as Join operators.

Definition and Notations

We now introduce the key definitions and notations that are related to the linear load model.

We express the load functions of the operators and nodes as:

$$\begin{aligned} l(o_j) &= l^o_{j1} r_1 + l^o_{j2} r_2 + \cdots + l^o_{jd} r_d \ , \quad j = 1, \cdots, m, \\ l(N_i) &= l^n_{i1} r_1 + l^n_{i2} r_2 + \cdots + l^n_{id} r_d \ , \quad i = 1, \cdots, n, \end{aligned}$$

where l^o_{jk} is the *load coefficient* of operator o_j for input stream I_k, and l^n_{ik} is the *load coefficient* of node N_i for input stream I_k (we use superscripts to distinguish load coefficients for operators and load coefficients for nodes), for $i = 1, \cdots, n,\ j = 1, \cdots, m$, and $k = 1, \cdots, d$. As shown in Example 3 above, these load coefficients can be computed using the costs and selectivities of the operators and are assumed to be constant unless otherwise specified.

Putting the load coefficients of the operators together, we get a m by d *operator load coefficient matrix*:

$$L^o = \begin{pmatrix} l^o_{11} & l^o_{12} & \cdots & l^o_{1d} \\ l^o_{21} & l^o_{22} & \cdots & l^o_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ l^o_{m1} & l^o_{m2} & \cdots & l^o_{md} \end{pmatrix}.$$

Putting the load coefficients of the nodes of together, we get a n by d *node load coefficient matrix*:

$$L^n = \begin{pmatrix} l^n_{11} & l^n_{12} & \cdots & l^n_{1d} \\ l^n_{21} & l^n_{22} & \cdots & l^n_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ l^n_{n1} & l^n_{n2} & \cdots & l^n_{nd} \end{pmatrix}.$$

We represent the distribution of operators on the nodes of the system by the n by m *operator allocation matrix*:

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1m} \\ a_{21} & a_{22} & \cdots & a_{2m} \\ \vdots & \vdots & \ddots & \vdots \\ a_{n1} & a_{n2} & \cdots & a_{nm} \end{pmatrix},$$

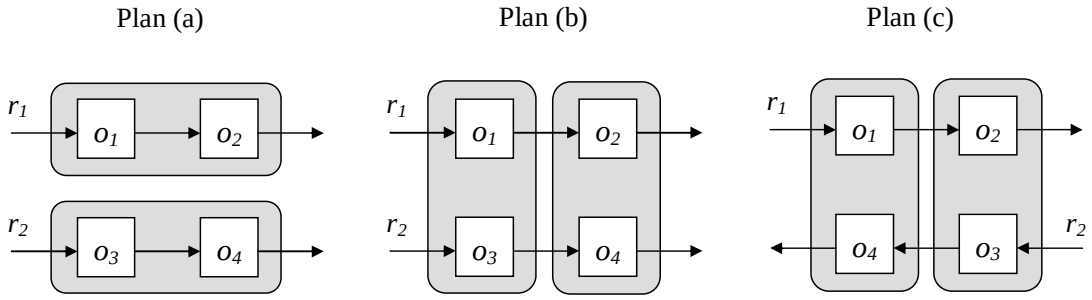


Figure 3.2: Three operator distribution plans for query graph (a).

where $a_{ij} = 1$ if operator o_j is assigned to node N_i and $a_{ij} = 0$ otherwise.

It follows from the definition of A that

$$L^n = AL^o.$$

Because the sum of the load coefficients of any input stream is fixed regardless of the operator allocation plan, we have that

$$\sum_{i=1}^{n} l_{ik}^n = \sum_{j=1}^{m} l_{jk}^o \ , \quad k = 1, \cdots d.$$

We also denote the total load coefficient of stream I_k by l_k.

Example 4: We now present a simple example of these notations using Query graph (a) (shown in Figure 3.1). Assume the following operator costs and selectivities: c_1= 14, c_2= 6, c_3= 9, c_4= 14 and s_1= 1, s_3= 0.5. Further assume that there are two nodes in the system, N_1 and N_2, with capacities C_1 and C_2, respectively. In Figure 3.2, we show three different operator distribution plans (Plan (a), Plan (b), and Plan(c)). In Table 3.2, we show the corresponding operator load coefficient matrix L^o and, for each of the above plans, the resulting operator allocation matrix A and node load coefficient matrix L^n.

Next, we introduce further notations to provide a formal definition for the feasible set of an operator distribution plan.

Let

$$C = \begin{pmatrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{pmatrix}$$

be the vector of CPU capacities of the nodes.

Table 3.2: Three operator distribution plans.

L^o	Plan	A	L^n
$\begin{pmatrix} 14 & 0 \\ 6 & 0 \\ 0 & 9 \\ 0 & 7 \end{pmatrix}$	(a)	$\begin{pmatrix} 1 & 1 & 0 & 0 \\ 0 & 0 & 1 & 1 \end{pmatrix}$	$\begin{pmatrix} 20 & 0 \\ 0 & 16 \end{pmatrix}$
	(b)	$\begin{pmatrix} 1 & 0 & 1 & 0 \\ 0 & 1 & 0 & 1 \end{pmatrix}$	$\begin{pmatrix} 14 & 9 \\ 6 & 7 \end{pmatrix}$
	(c)	$\begin{pmatrix} 1 & 0 & 0 & 1 \\ 0 & 1 & 1 & 0 \end{pmatrix}$	$\begin{pmatrix} 14 & 7 \\ 6 & 9 \end{pmatrix}$

Let

$$R = \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_d \end{pmatrix}$$

be the vector of system input stream rates.

Let L_i^n be the ith row of matrix L^n, i.e.

$$L_i^n = \begin{pmatrix} l_{i1}^n & l_{i2}^n & \dots & l_{id}^n \end{pmatrix}.$$

Then, the load of node N_i can then be written as

$$L_i^n R.$$

Node N_i is feasible if and only if

$$L_i^n R \leq C_i.$$

The system is feasible if and only if all nodes in the system are feasible, i.e.

$$L^n R \leq C$$

or equivalently

$$AL^o R \leq C.$$

The set of all possible input stream rate points is called the *workload set* and is referred to by D. For example, if there are no constraints on the input stream rates, then

$$D = \left\{ R : R \geq (0, \cdots, 0)^T \right\}.$$

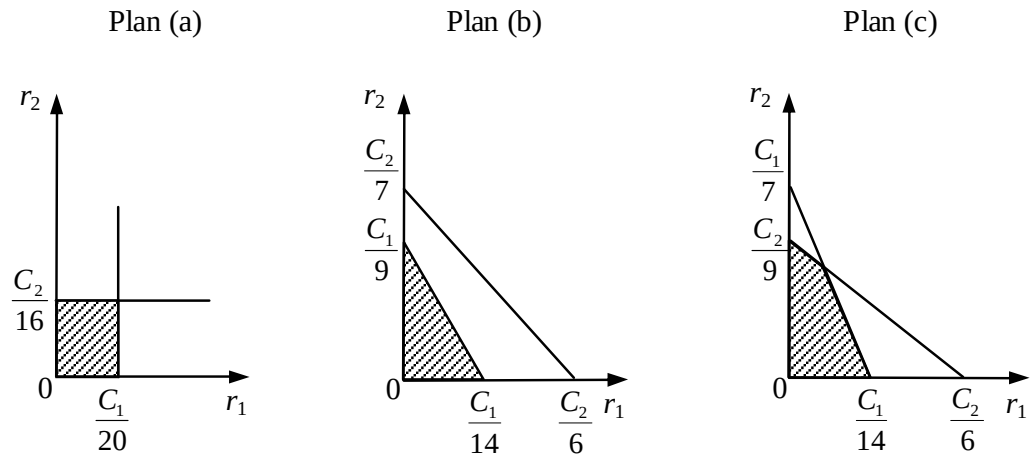


Figure 3.3: Feasible sets of various distribution plans.

DEFINITION 1 (**Feasible Set**): *Given a CPU capacity vector C, an operator load coefficient matrix L^o, and a workload set D, the feasible set of the system under operator distribution plan A (denoted by $F(A)$) is defined as the set of all points in the workload set D for which the system is feasible, i.e.,*

$$F(A) = \{R : R \in D,\ AL^oR \leq C\}.$$

In Figure 3.3, we show the feasible sets (the shaded regions) of the distribution plans of Example 4. We can see that different operator distribution plans can result in very different feasible sets.

The notations introduced in this section as well as some notations that are related to the linear load model and used in the later chapters are summarized in Table 3.3.

3.1.3 Time Series Load Model

For the dynamic operator distribution algorithm, we use the statistics collected in the most recent time intervals to make operator distribution decisions. The basic assumption is what has been observed in the most recent past can be used to predict what will happen in the near future.

In this load model, we measure the average system input stream rates, operator costs and selectivities periodically. The load of the operators and nodes are computedd in each time period using the measured statistics. Note that, in this model, we don't assume that operator costs or selectivities are constant.

The statistics measurement period should be long enough so that the computed load is independent of the scheduling policy, and any high frequency load fluctuation is smoothed out. The load values of the most recent h periods are kept in the system for decision making. The total time of the h statistics measurements is called the *statistics window* of the system. It should be selected large enough to avoid operator migration thrashing. The time series of the h load values

Table 3.3: Notations related to linear load model.

$C = (C_1, \cdots, C_n)^T$	available CPU capacity vector
$R = (r_1, \cdots, r_d)^T$	system input stream rate vector
l^n_{ik}	load coefficient of N_i for I_k
l^o_{jk}	load coefficient of o_j for I_k
$L^n = \{l^n_{ik}\}_{n\times d}$	node load coefficient matrix
$L^o = \left\{l^o_{jk}\right\}_{m\times d}$	operator load coefficient matrix
$A = \{a_{ij}\}_{n\times m}$	operator allocation matrix
D	workload set
$F(A)$	feasible set of A
C_T	total node CPU capacities
l_k	total load coefficients of I_k

of an operator or a node is called the *load time series* of the operator or the node.

Given a load time series $X = (x_1, x_2, \cdots, x_h)$, its mean $\mathsf{E}X$ and variance $\mathrm{var}X$ are defined as follows:

$$\mathsf{E}X = \frac{1}{h}\sum_{i=1}^{h} x_i,$$

$$\mathrm{var}X = \frac{1}{h}\sum_{i=1}^{h} x_i^2 - \left(\frac{1}{h}\sum_{i=1}^{h} x_i\right)^2.$$

Given two load time series $X_1 = (x_{11}, x_{12}, \cdots, x_{1h})$ and $X_2 = (x_{21}, x_{22}, \cdots, x_{2h})$, their covariance $\mathrm{cov}\,(X_1, X_2)$ and correlation coefficient $\mathrm{cor}\,(X_1, X_2)$ (often denoted by ρ_{12}) are defined as follows:

$$\begin{aligned}\mathrm{cov}\,(X_1, X_2) &= \frac{1}{h}\sum_{i=1}^{h} x_{1i}x_{2i} - \mathsf{E}X_1\mathsf{E}X_2,\\ \mathrm{cor}(X_1, X_2) &= \frac{\mathrm{cov}\,(X_1, X_2)}{\sqrt{\mathrm{var}X_1}\cdot\sqrt{\mathrm{var}X_2}}.\end{aligned}$$

In this dissertation, the variance of the load time series of an operator or node is also called the *load variance* of that operator or node. The correlation coefficient of the load time series of

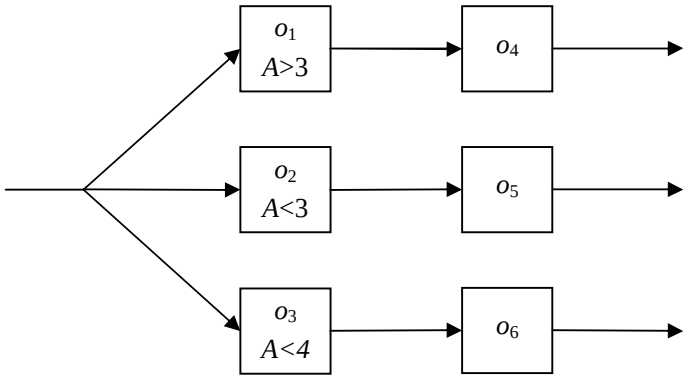


Figure 3.4: Stream with attribute A feeding different filters.

two operators or nodes is also called the *load correlation* of the two operators or nodes. The mean of the load time series of an operator or node is called the *average load* of that operator or node.

Traditional load balancing algorithms attempt to balance the average load of the nodes. Our algorithm not only tries to balance the average load, but also considers the load variances of the nodes as well as the load correlations between the node pairs.

Our algorithm is based on the observation that load correlations vary among operators. This variation is a result of more than the fluctuation of different input rates. It also results from the nature of the queries. This is best illustrated by an simple example:

EXAMPLE 5: Consider a stream with attribute A feeding three filter operators as depicted in Figure 3.4. The filtering predicates of o_1, o_2 and o_3 are $A < 3$, $A > 3$ and $A > 4$ respectively. It is not difficult to see that no matter how the input stream rate fluctuates, operators o_1, o_2 and o_3 always have a pair-wise load correlation 1. In addition, operator o_4 tends to have negative load correlations with operators o_5 and o_6. Operators o_5 and o_6 tend to have a positive load correlation.

Such query-determined correlations are stable or relatively stable in comparison to input-determined correlations. This feature is important to our operator distribution algorithms because we use the correlations to determine the locations of the operators. If the correlations are highly volatile, the decisions made may soon lose their effectiveness.

The notations related to the time series load model are summarized in Table 3.4.

3.2 Problem Statement

In this dissertation, we assume that each single query operator is the minimum unit for load migration. In other words, we can only move operators if we want to move some load from one node to another node. In this sense, the load management problem is actually the operator allocation problem. Therefore, we want to find good operator mapping plans such that the performance of

Table 3.4: Notations related to time series load model.

$X = (x_1, x_2 \cdots, x_h)$	load time series
$\mathsf{E}X$	average load
$\mathrm{var}X$	load variance
$\mathrm{cov}(X_1, X_2)$	load covariance
$\mathrm{cor}(X_1, X_2)$	load correlation

the system is optimized.

Our operator distribution problem can be further divided into two problems: a static operator distribution problem and a dynamic operator distribution problem. In the first problem, we want to find resilient or robust operator distribution plans for operators that are impractical to be moved between nodes. Its optimization goal is to keep the system feasible for unpredictable load variations without operator migration. In the dynamic operator distribution problem, we focus on operators with relatively small load migration overheads. The optimization goal is to minimize the average end-to-end latency of the output tuples and avoid frequent operator migration.

3.2.1 The Static Operator Distribution Problem

In this problem, we assume a linear load model with constant operator costs and selectivities. Therefore, the load variations of the operators are only the results of the variations of the system input stream rates. In order to be resilient to time-varying, unpredictable workloads and maintain quality of service (i.e., consistently produce low latency results), we aim to maximize the size of the feasible set of the system through intelligent operator distribution. We state the corresponding optimization problem more formally below:

PROBLEM 1 (**The Resilient Operator Distribution (ROD) problem**): *Given a CPU capacity vector C, an operator load coefficient matrix L^o, and a workload set D, find an operator allocation matrix A^* that achieves the largest feasible set size among all operator allocation plans, i.e., find*

$$A^* = \arg\max_{A} \int_{F(A)} \cdots \int 1 \, dr_1 \cdots dr_d,$$

where

$$F(A) = \{R : R \in D,\ AL^oR \leq C\}.$$

In the equation above, the multiple integral over $F(A)$ represents the size of the feasible set of A. Note that A^* may not be unique.

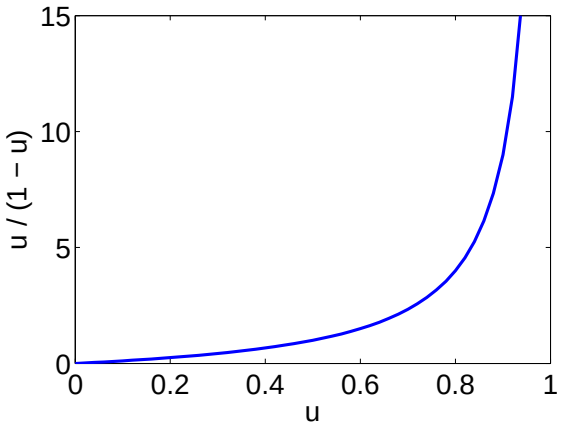


Figure 3.5: The curve of function $u/(1-u)$.

ROD is different from the canonical linear programming and nonlinear programming problems with linear constraints on feasible sets. The latter two problems aim to maximize or minimize a (linear or nonlinear) objective function on a fixed feasible set (with fixed linear constraints) [32, 64], whereas in our problem, we attempt to maximize the size of feasible set by appropriately *constructing* the linear constraints through operator distribution. To the best of our knowledge, our work is the first to study this problem in the context of load distribution.

3.2.2 The Dynamic Operator Distribution Problem

The dynamic operator distribution problem aims at the initial and dynamic re-distribution of operators with relatively small load migration overheads. In this problem, we use the time series load model to adapt to the most recent load changes dynamically.

A stream processing system is also a queuing system. Therefore, to minimize average end-to-end latency, we should try to minimize the average queueing latency in the system. Unfortunately, due to the complexity and variety of stream-based applications, it is very hard to solve this problem using precise mathematical queuing models. The commonly used models are either not suitable for stream processing systems (e.g. the M/M/1 model) or hardly can be used to estimate the average queueing time for different operator distribution plans (e.g. the G/G/1 models). However, despite the difficulty of finding precise optimal solutions, we can still use queueing theory as a guide to find approximate sub-optimal solutions.

From the G/G/1 model of queueing theory [82], we know that the average queueing latency of a single server is often approximately in proportional to

$$\frac{u}{1-u}.$$

where u is the CPU utilization of the server. As depicted by Figure 3.5, when u increases from 0 to 1, the curve of the function is very flat at the beginning and becomes very steep when the u approaches 1. From this curve, it is not hard to see that if two machines have the same average

CPU utilization, but different variances in CPU utilization, the one with smaller variance tends to have smaller average queuing latency. Therefore, the optimization goal of minimizing the average end-to-end latency can be approached by balancing the average load of the nodes and minimizing the load variance of each node.

In addition to this optimization goal, we also want to minimize the number of operator migrations to avoid jitters in end-to-end latency. As stated before, this goal can be approached by maximizing the load correlations of the node pairs in the system so that the loads of the nodes tend to be naturally balanced even when the load changes.

We will see later that, when the average load variance of the nodes is minimized, the average load correlation of the node pairs is maximized (assuming the total load of the system can be arbitrarily partitioned), i.e. the two optimization goals are achieved at the same time in the ideal case. Thus, the dynamic operator distribution problem can be formalized as the follows:

PROBLEM 2 (**The dynamic operator distribution problem**): *Let X_i denote the load time series of node N_i and ρ_{ij} denote the correlation coefficient of X_i and X_j for $1 \leq i, j \leq n$. We want to find and maintain operator mapping plans with the following properties:*

1. $\mathsf{E}X_1 \approx \mathsf{E}X_2 \approx \cdots \approx \mathsf{E}X_n$
2. $\frac{1}{n}\sum_{i=1}^{n} \mathrm{var}Xi$ *is minimized and*
3. $\sum_{1 \leq i < j \leq n} \rho_{ij}$ *is maximized*

When the load of the operators changes, the performance of an initially good operator distribution plan may deteriorate over time. In this case, operators need to be moved between the nodes to achieve a better load distribution plan. When moving an operator from on node to another node, the execution of the operator must be suspended. Without considering the CPU overhead for operator migration, if no tuple arrives at the operator being moved, then moving this operator does not result in any extra delay in the end-to-end latency. On the other hand, if the load of the operator being moved is large, then moving this operator may result in large delays in end-to-end latency, because not only the tuples arrived during the operator migration process are delayed, but also the tuples arrived after the operator migration process are delayed since the receiving node must first process the accumulated tuples of the received operators. Thus, for dynamic load re-distribution, we want to avoid moving too much load betweent the nodes.

There is usually a trade off between the performance of the system during the operator migration process and the performance after the migration process. Moving more load may result in a better load distribution plan than moving less load; thus, result in better performance after the load re-distribution process. However, moving more load also tends to result in larger end-to-end latency due to the operator migration process. How to move less load and achieve good performance is a common problem faced by many dynamic load distribution algorithms. In this

dissertation, we try to achieve a good balance between the amount of load moved in the dynamic load distribution process and the performance of the resulting load distribution plan.

Chapter 4

Static Operator Distribution

In this chapter, we present our Resilient Operator Distribution (ROD) algorithm, which aims at allocating operators that can not be moved between nodes on the fly or operators with very large migration overheads.

Our algorithm is guided by the theoretical analysis of an ideal load distribution plan, where the load of all nodes stays balanced no matter how the workload of the system varies over time. Such an ideal plan is usually unachievable given the constraint that each operator is the minimum load distribution unit. We then explore two operator distribution heuristics that aim at making the achieved feasible set close the feasible set in the ideal case as much as possible. Both heuristics try to maximize the size of of a subset of the achieved feasible set. The subset in the first heuristic is a polytope. The subset in the second heuristic is a partial hypersphere. Thus, these heuristics are actually maximizing the two lower bounds of the achieved feasible set size. Our operator distribution algorithm is a greedy algorithm the seamlessly combines both heuristics.

In this chapter, we first assume a linear load model with fixed operator costs and selectivities. We also assume that there is no constraint or extra information about the system input stream rates. In addition, we assume that the CPU overhead for data communication is negligible and network bandwidth is not a bottleneck of the system. These assumption are relaxed in Section 4.4, where we present extensions to the basic algorithm. More specifically, we present how to use the lower bound on system input stream rates to further maximize feasible set size, how to transform a non-linear load model into a linear load model, and how to cluster operators as a pre-processing step of ROD to reduce data communication CPU overhead and network bandwidth consumption.

At the end of this chapter, we discuss the computation complexity of the algorithm.

4.1 Optimization Fundamentals

A straightforward solution to ROD requires enumerating all possible allocation plans and comparing their feasible set sizes. Unfortunately, the number of different distribution plans is $n^m/n!$.

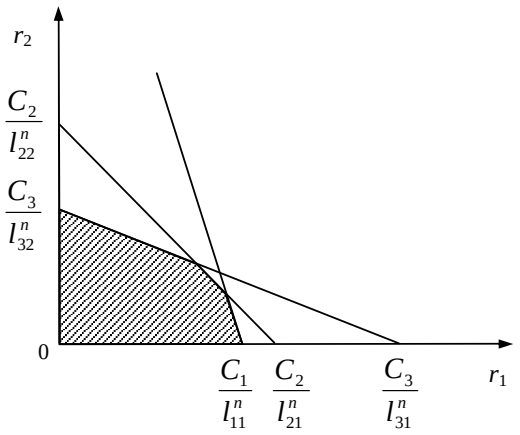


Figure 4.1: Feasible set and hyperplane example.

Moreover, even computing the feasible set size of a single plan (i.e., a d dimensional multiple integral) is expensive; since the Monte Carlo integration method, which is commonly used in high dimensional integration, requires at least $O(2^d)$ sample points [76]. As a result, finding the optimal solution for this problem is intractable for large d or large m.

Given the intractability of ROD, we explore a heuristic-driven strategy. We first explore the characteristics of an "ideal" plan using a linear algebraic model and the corresponding geometrical interpretation. We then use this insight to derive our solution.

4.1.1 Feasible Set and Node Hyperplanes

We here examine the relationship between the feasible set size and the node load coefficient matrix. Initially, we make no assumptions about the expected workload and assume that

$$D = \left\{ R : R \geq (0, \cdots, 0)^T \right\}.$$

(we relax this assumption in Section 4.4.1).

Given a node load coefficient matrix L^n, the feasible set determined by L^n is denoted by

$$F'(L^n) = \{ R : R \in D, L^n R \leq C \}.$$

This is a convex polytope [88] in the nonnegative space below n hyperplanes, where the hyperplanes are defined by

$$l_{i1}^n r_1 + l_{i2}^n r_2 + \cdots + l_{id}^n r_d = C_i, \quad i = 1, \cdots, n.$$

EXAMPLE 6: Figure 4.1 illustrates an example feasible set in a system with two input streams and three nodes. The lines correspond to the hyperplanes and the shaded area below the lines is the feasible set.

Note that the ith hyperplane consists of all points that render node N_i fully loaded. If a point is above this hyperplane, then N_i is overloaded at that point. The system is feasible at a point if and only if the point is on or below all of the n hyperplanes defined by

$$L^n R = C.$$

We refer to these hyperplanes as *node hyperplanes*.

For instance, in Figure 3.3, the node hyperplanes correspond to the lines above the feasible sets. Because the node hyperplanes collectively determine the shape and size of the feasible set, the feasible set size can be optimized by constructing "good" node hyperplanes or, equivalently, by constructing "good" node load coefficient matrices.

4.1.2 Ideal Node Load Coefficient Matrix

We now present and prove a theorem that characterizes an "ideal load distribution plan". The node load coefficient matrix that yields this ideal load distribution plan is called the *ideal* node load coefficient matrix.

THEOREM 1: *Given operator load coefficient matrix* $L^o = \left\{l^o_{jk}\right\}_{m\times d}$ *and node capacity vector* $C = (C_1, \cdots, C_n)^T$, *among all* n *by* d *matrices* $L^n = \{l^n_{ik}\}_{n\times d}$ *that satisfy the constraint*

$$\sum_{i=1}^{n} l^n_{ik} = \sum_{j=1}^{m} l^o_{jk} \doteq l_k, \quad \textit{for } k = 1, \cdots, d \tag{4.1}$$

the matrix $L^{n*} = \{l^{n*}_{ik}\}_{n\times d}$ *with*

$$l^{n*}_{ik} = l_k \frac{C_i}{C_T},$$

where

$$C_T = \sum_{i=1}^{n} C_i,$$

achieves the maximum feasible set size, i.e.,

$$L^{n*} = \arg\max_{L^n} \int \cdots \int_{F'(L^n)} 1 \; dr_1 \cdots dr_d,$$

Proof. All node load coefficient matrices must satisfy constraint 4.1. L^{n*} also satisfies this constraint because

$$\sum_{i=1}^{n} l^{n*}_{ik} = \sum_{i=1}^{n} l_k \frac{C_i}{C_T} = l_k \frac{\sum_{i=1}^{n} C_i}{C_T} = l_k.$$

Now, it suffices to show that L^{n*} has the largest feasible set size among all L^n that satisfy constraint 4.1.

From $L^n R \leq C$, we have that

$$(1\ 1 \cdots 1) \begin{pmatrix} l_{11}^n & l_{12}^n & \cdots & l_{1d}^n \\ l_{21}^n & l_{22}^n & \cdots & l_{2d}^n \\ \vdots & \vdots & \ddots & \vdots \\ l_{n1}^n & l_{n2}^n & \cdots & l_{nd}^n \end{pmatrix} \begin{pmatrix} r_1 \\ r_2 \\ \vdots \\ r_d \end{pmatrix} \leq (1 \cdots 1) \begin{pmatrix} C_1 \\ C_2 \\ \vdots \\ C_n \end{pmatrix},$$

which can be written as

$$l_1 r_1 + l_2 r_2 + \cdots + l_d r_d \leq C_T. \tag{4.2}$$

Thus, any feasible point must belong to the set

$$F^* = \{R : R \in D,\ l_1 r_1 + l_2 r_2 + \cdots + l_d r_d \leq C_T\}.$$

In other words, $F^*$is the superset of any feasible set. It then suffices to show that $F'(L^{n*}) = F^*$.

There are n constraints in $L^{n*} R \leq C$ (each row is one constraint). For the ith row, we have that

$$l_1 \frac{C_i}{C_T} r_1 + l_2 \frac{C_i}{C_T} r_2 + \cdots + l_d \frac{C_i}{C_T} r_d \leq C_i\ ,$$

which is equivalent to inequality 4.2. Since all n constraints are the same, we have that

$$F'(L^{n*}) = F^*.$$

□

Intuitively, Theorem 1 says that the feasible set size is maximized if the load coefficient of each input stream is *perfectly* balanced across all nodes (in proportion to the relative CPU capacity of each node). In this case, the load of all nodes are also perfectly balanced across all nodes (in proportion to the relative CPU capacity of each node) regardless of what the system input stream rates are. Therefore, no matter how the input stream rates vary, there is no need to move operators in the system at all. Note that the ideal coefficient matrix is independent of the workload set D.

Such a load coefficient matrix may not be realizable by using operator distribution, i.e., there may not exist operator allocation matrix A such that $AL^o = L^{n*}$ (the reason why L^{n*} is referred to as "ideal").

The largest feasible set achieved by the ideal load coefficient matrix is called the *ideal feasible set* (denoted by F^*). It consists of all points that fall below or on the *ideal hyperplane*

$$l_1 r_1 + l_2 r_2 + \cdots + l_d r_d = C_T.$$

When the ideal node load coefficient matrix is obtained, all node hyperplanes overlap with the ideal hyperplane.

We can compute the size of the ideal feasible set as:

$$V(F^*) = \int_{F^*} \cdots \int 1\ dr_1 \cdots dr_d = \frac{(C_T)^d}{d!} \cdot \prod_{k=1}^{d} \frac{1}{l_k}.$$

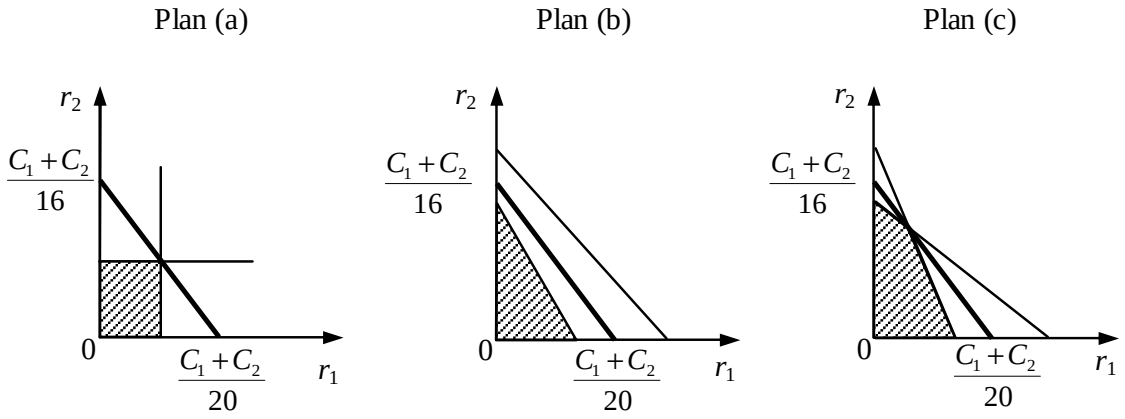


Figure 4.2: Ideal hyperplanes and feasible sets.

Figure 4.2 illustrates the ideal hyperplanes (represented by the thick lines) and the feasible sets of Plan (a), (b) and (c) in Example 4. It is easy to see that none of the shown distribution plans are ideal. In fact, all distribution plans for Example 4 can not achieve the ideal feasible set.

4.1.3 Optimization Guidelines

The key high-level guideline that we will rely on to maximize feasible set size is to make the node hyperplanes as close to the ideal hyperplane as possible.

To accomplish this, we first normalize the ideal feasible set by changing the coordinate system. The normalization step is necessary to smooth out high variations in the values of load coefficients of different input streams, which may adversely bias the optimization.

Let $x_k = l_k r_k / C_T$. In the new coordinate system with axis x_1 to x_k, the corresponding node hyperplanes are defined by

$$\frac{l_{i1}^n}{l_1} x_1 + \frac{l_{i2}^n}{l_2} x_2 + \cdots + \frac{l_{id}^n}{l_d} x_d = \frac{C_i}{C_T}\ , \quad i = 1, \cdots, n.$$

The corresponding ideal hyperplane is defined by

$$x_1 + x_2 + \cdots + x_d = 1.$$

By the change of variable theorem for multiple integrals [86], we have that the size of the original feasible set equals a constant c times the size of the normalized feasible set, where

$$c = \frac{(C_T)^d}{\prod_{k=1}^{d} l_k}.$$

Therefore, the goal of maximizing the original feasible set size can be achieved by maximizing the normalized feasible set size.

We now define our goal more formally using our algebraic model:

Let matrix

$$W = \{w_{ik}\}_{n \times d} = \left\{ \frac{l_{ik}^n}{l_{ik}^{n*}} \right\}_{n \times d}.$$

From $l_{ik}^{n*} = l_k C_i / C_T$, we have that

$$w_{ik} = \frac{l_{ik}^n / l_k}{C_i / C_T}.$$

In other words, w_{ik} is the percentage of the load from stream I_k on node N_i divided by the normalized CPU capacity of N_i (C_i/C_T). Thus, we can view w_{ik} as the "weight" of stream I_k on node N_i and view matrix W as a normalized form of a load distribution plan. Matrix W is also called the *weight matrix*.

Note that the equations of the node hyperplanes in the normalize space is equivalent to

$$w_{i1}x_1 + w_{i2}x_2 + \cdots + w_{id}x_d = 1, \quad i = 1, \cdots, n.$$

Our goal is then to make the normalized node hyperplanes close to the normalized ideal hyperplane, i.e. make

$$W_i = (w_{i1}\ w_{i2}\ \cdots\ w_{id}) \text{ close to } (1\ 1\ \cdots\ 1),$$

for $i = 1, \cdots, n$.

For brevity, in the rest of this paper, we assume that all terms such as hyperplane, feasible set and ideal feasible set refer to the ones in the normalized space, unless stated otherwise.

4.2 Heuristics

We now present two heuristics that are guided by the formal analysis presented in the previous section. For simplicity of exposition, we motivate and describe the heuristics from a geometrical point of view. We also formally present the pertinent algebraic foundations as appropriate.

4.2.1 Heuristic 1: MaxMin Axis Distance

Recall that we aim to make the node hyperplanes converge to the ideal hyperplane as much as possible. In the first heuristic, we try to push the intersection points of the node hyperplanes with each axis to the intersection points of the ideal hyperplane with each axis as much as possible. In other words, we would like to make the *axis distance* of each node hyperplane as close to that of the ideal hyperplane as possible. We define the axis distance of hyperplane h on axis a as the distance from the origin to the intersection point of h and a.

Example 7: Figure 4.3(a) shows the plane distance of two node hyperplanes. The axis distance of node N_i is $\frac{1}{w_{i1}}$ on axis x_1 and $\frac{1}{w_{i2}}$ on axis x_2. The plane distance of node N_j is $\frac{1}{w_{j1}}$ on axis x_1 and $\frac{1}{w_{j2}}$ on axis x_2. The axis distance of the ideal hyperplanes is one on both axis.

In general, the axis distance of the ith node hyperplane on the kth axis is

$$\frac{1}{w_{ik}}.$$

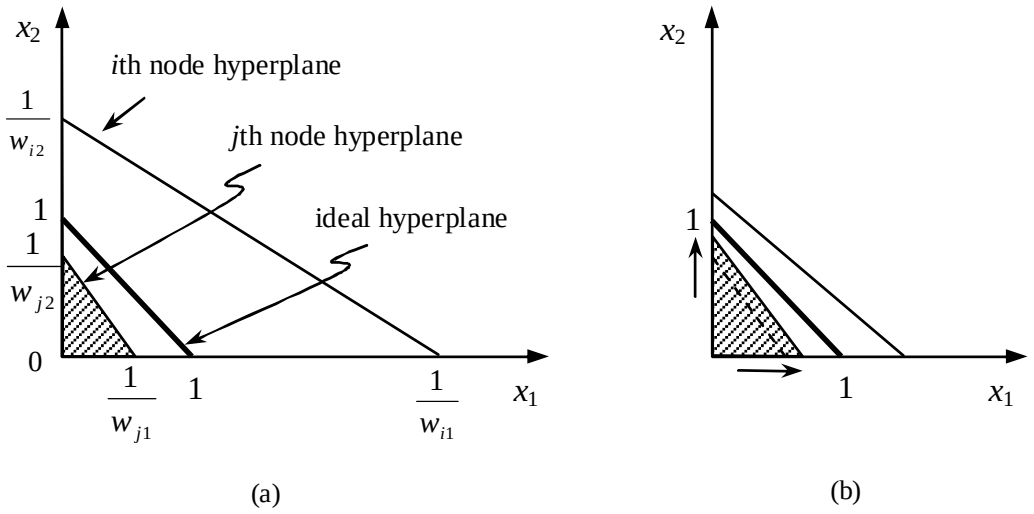


Figure 4.3: Heuristic 1 example.

The axis distance of the ideal hyperplane is one on all axis. Thus, from the algebraic point of view, by making each axis distance as close to one as possible, this heuristic is trying to making each entry of W_i as close to one as possible.

Because $\sum_i l_{ik}^n$ is fixed for each k, the optimization goal of making w_{ik} close to one for all k is equivalent to balancing the load of each input stream across the nodes in proportion to the nodes' CPU capacities. This goal can be achieved by maximizing the minimum axis distance of the node hyperplanes on each axis; i.e., we want to maximize

$$\min_i \frac{1}{w_{ik}}, \quad \text{for } k = 1, \cdots, d.$$

We therefore refer to this heuristic as *MaxMin Axis Distance* (MMAD). For example, this heuristic prefers the plan in Figure 4.3(b) to the one in Figure 4.3(a). The arrows in Figure 4.3(b) illustrate how MMAD pushes the intersection points of the node hyperplanes that are closest to the origin to that of the ideal hyperplane.

It can be proven that when the minimum axis distance is maximized for axis k, all the node hyperplane intersection points along axis k converge to that of the ideal hyperplane. This conclusion can be formally stated as the follows:

THEOREM 2: *The minimum axis distance of all node hyperplanes on a given axis is maximized if and only if the axis distance of all node hyperplanes on that axis is equal to the axis distance of the ideal hyperplane on that axis.*

To prove this theorem, we first state and prove the following lemma:

LEMMA 1: *Given n non-negative numbers a_1, a_2, $\cdots$, a_n. For all non-negative variables*

$b_1,\ b_2,\ \cdots,\ b_n$ that satisfy the constraint

$$\sum_{i=1}^{n} a_i b_i = 1, \tag{4.3}$$

the value of

$$\min_i \frac{1}{b_i}$$

is maximized if and only if

$$b_1 = b_2 = \cdots = b_n = \frac{1}{\sum_{i=1}^{n} a_i}.$$

Thus

$$\max_{b_1 \cdots b_n} \min_i \frac{1}{b_i} = \sum_{i=1}^{n} a_i.$$

Proof. Let

$$b^* \doteq \frac{1}{\sum_{i=1}^{n} a_i}.$$

Since

$$\sum_{i=1}^{n} a_i b^* = b^* \cdot \sum_{i=1}^{n} a_i = 1,$$

which means that $b_1 = b_2 = \cdots = b_n = b^*$ satisfy constraint 4.3, we have that

$$\max_{b_1 \cdots b_n} \min_i \frac{1}{b_i} \geq \frac{1}{b^*}. \tag{4.4}$$

Suppose there exist $1 \leq i_1 \leq i_2 \leq \cdots \leq i_v \leq n$ $(v \geq 1)$, such that

$$b_{i_1} = b_{i_2} = ... = b_{i_v} = \min_i b_i < b^*.$$

Then there must exist $1 \leq u \leq n$ such that $b_u > b^*$. Otherwise, we would have

$$\sum_{i=1}^{n} a_i b_i < \sum_{i=1}^{n} a_i b^* = 1.$$

Therefore we have that

$$\max_i b_i \geq b^* \tag{4.5}$$

and the equality holds if and only if

$$b_1 = b_2 = \cdots = b_n = b^*.$$

Inequality 4.5 can also be written as

$$\min_i \frac{1}{b_i} \leq \frac{1}{b^*}.$$

Thus we have

$$\max_{b_1 \cdots b_n} \min_i \frac{1}{b_i} \leq \frac{1}{b^*}. \tag{4.6}$$

Combining Inequalities 4.4 and 4.6, we have that

$$\max_{b_1 \cdots b_n} \min_i \frac{1}{b_i} = \frac{1}{b^*}. \tag{4.7}$$

The equality holds if and only if

$$b_1 = b_2 = \cdots = b_n = b^*.$$

□

Next, we give the proof of Theorem 2.

Proof. Without loss of generality, we consider the kth axis. Let

$$a_i = \frac{C_i}{C_T}, \quad b_i = w_{ik}.$$

Since

$$\sum_{i=1}^{n} a_i b_i = \sum_{i=1}^{n} \frac{C_i}{C_T} \cdot \frac{l_{ik}^n / l_k}{C_i / C_T} = \frac{\sum_{i=1}^{n} l_{ik}^n}{l_k} = 1,$$

following Lemma 1, we have that

$$\min_i \frac{1}{w_{ik}}$$

is maximized if and only if

$$w_{1k} = w_{2k} = \cdots = w_{nk} = \frac{1}{\sum_{i=1}^{n} C_i / C_T} = 1,$$

□

MMAD tries to maximize the minimum axis distance for each axis. When the minimum axis distance is maximized on all axes, we achieved the ideal feasible set.

From a geographic point of view, MMAP can be considered as maximizing the size of a polytope subset of the achieved feasible set. This subset is the polytope below the hyperplane

$$\left(\max_i w_{i1}\right) x_1 + \left(\max_i w_{i2}\right) x_2 + \cdots + \left(\max_i w_{ik}\right) x_k = 1. \tag{4.8}$$

In the non-negative space, any node hyperplane is above, or overlaps with, the hyperplane defined by Equation 4.8. Thus, the set below this hyperplane is a polytope subset of the feasible set. Figure 4.4 illustrates an example of this subset (denoted by G) in a system with three nodes and two input streams.

The size of this subset can be computed as

$$V(G) = \frac{1}{d!} \cdot \prod_{k=1}^{d} \frac{1}{\max_i w_{ik}} = \frac{1}{d!} \cdot \prod_{k=1}^{d} \min_i \frac{1}{w_{ik}}.$$

Notice that

$$\frac{1}{d!}$$

is the size of the normalized feasible set size. Therefore, the original feasible set size is bounded by

$$V(F^*) \cdot \prod_{k=1}^{d} \min_i \frac{1}{w_{ik}}$$

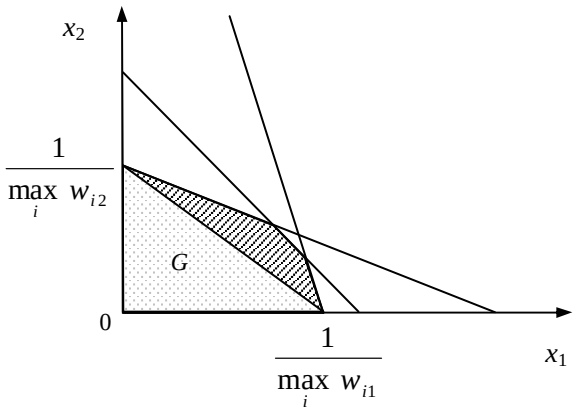


Figure 4.4: A polytope subset of the feasible set.

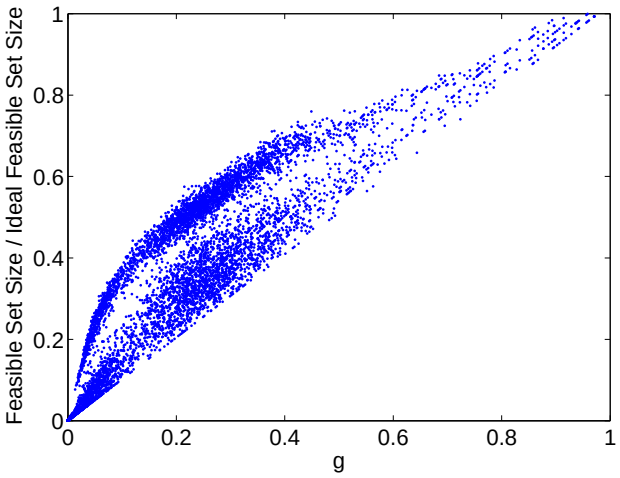


Figure 4.5: Relationship between g and feasible set size.

from below, where $V(F^*)$ is the size of the original ideal feasible set. In other words, the product of minimum axis distance on all axes determines a lower bound on feasible set size. Thus, MMAD is trying to maximize a lower bound for feasible set size. This lower bound is close to the ideal value when all axis distances can be well balanced.

Let

$$g \doteq \prod_{k=1}^{d} \min_i \frac{1}{w_{ik}}.$$

To further examine the relationship between g and the feasible set size, we generated random node load coefficient matrices and plotted the ratios of their feasible-set-size / ideal-feasible-set-size vs. g. Figure 4.5 shows the results of 10000 random load coefficient matrices with 10 nodes and 3 input streams. We see a trend: both the upper bound and lower bound of the feasible-set-size ratio increase when g increases.

On the downside, the key limitation of MMAD is that it does not take into consideration how

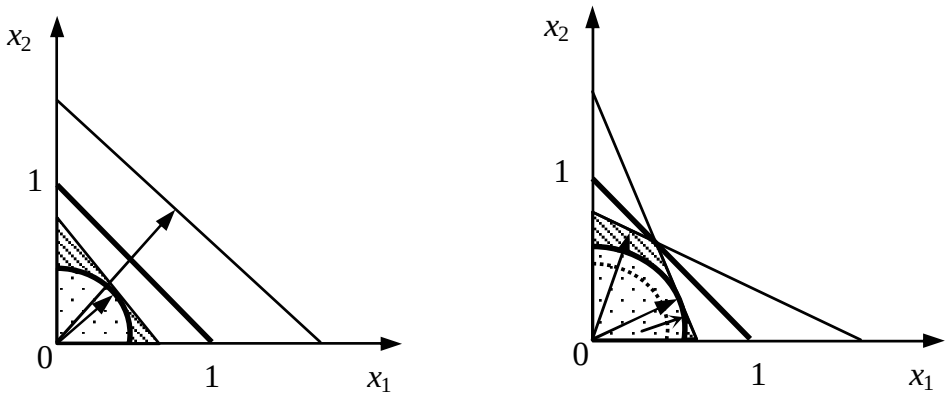


Figure 4.6: Heuristic 2 example.

to combine the weights of different input streams at each node. This is best illustrated by a simple example as depicted by Figure 4.6. Both plans in Figure 4.6 are deemed equivalent by MMAD, since their axis intersection points are exactly the same. They do, however, have significantly different feasible sets.

If we draw a vertical line in Figure 4.5, we can also see that plans with the same g have very different feasible set sizes. We should avoid those points that are close to the lower bound and try to achieved the points that are close to the upper bound.

Obviously, if the largest weights for each input stream are put on the same node (e.g., the one with the lowest hyperplane in Figure 4.6(a)), the corresponding node becomes the bottleneck of the system because it always has more load than the other node. The resulting feasible set size is equal to the lower bound determined by g. Using MMAD only can not avoid this situation. Next, we describe another heuristic that addresses this limitation.

4.2.2 Heuristic 2: MaxMin Plane Distance

Intuitively, MMAD pushes the intersection points of the node hyperplanes closer to those of the ideal hyperplane using the axis distance metric. Our second heuristic, on the other hand, pushes the node hyperplanes directly towards the ideal hyperplane using the *plane distance* metric. The plane distance of an hyperplane h is the distance from the origin to h. For example, the plane distance of a node hyperplane in Figure 4.6(a) is the length of the arrow that points to this plane. Our goal is to maximize the minimum plane distance of all node hyperplanes. We refer to this heuristic as *MaxMin Plane Distance* (MMPD).

Another way to think about this heuristic is to imagine a partial hypersphere that has its center at the origin and its radius r equal to the minimum plane distance (e.g. Figure 4.6). Obviously, MMAD prefers Figure 4.6(b) to Figure 4.6(a) because the former has larger r. The small arrow

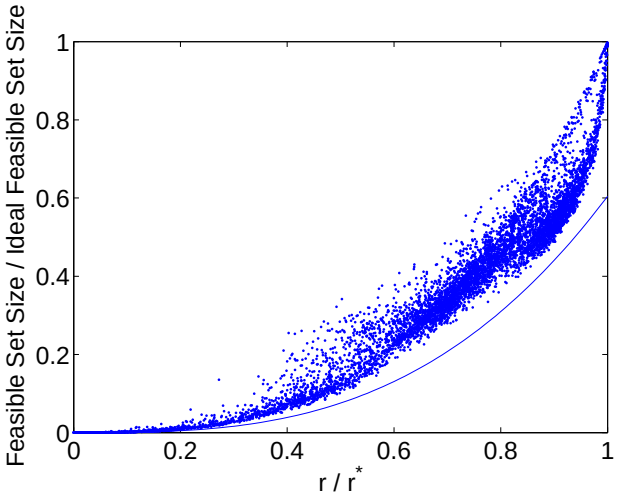


Figure 4.7: Relationship between r and feasible set size.

in Figure 4.6(b) illustrates how MMPD pushes the hyperplane that is the closest to the origin in terms of plane distance towards the ideal hyperplane.

The plane distance of the ith hyperplane is computed as:

$$\frac{1}{\sqrt{w_{i1}^2 + \cdots + w_{id}^2}} = \frac{1}{\|W_i\|},$$

where $\|W_i\|$ denotes the second norm of the ith row vector of W. Thus, the value we would like to maximize is:

$$r = \min_i \frac{1}{\|W_i\|} .$$

By maximizing r, i.e. maximizing the size of the partial hypersphere (a subset of the achieved feasible set), we are actually maximizing a lower bound of the achieved feasible set size. This lower bound is r^d [87] multiplied by a constant,

To further examine the relationship between r and the feasible set size, we plotted the ratios of their feasible-set-size / ideal-feasible-set-size vs. the ratios of $r/\ r^*$ for randomly generated node load coefficient matrices. Figure 4.7 shows the results of 10000 random load coefficient matrices with 10 nodes and 3 input streams. The curve in the graph is the computed lower bound using the volume function of hyperspheres [87]. Similar to the trend in Figure 4.5, both the upper bound and lower bound of the feasible-set-size ratio increase when $r/\ r^*$ increases. The lower bound in Figure 4.5 follows a linear function. The lower bound in Figure 4.5 is very close to a function in proportional to r^d because the volume of the subset hypersphere is a constant times r^d. For different n and d, the upper bound and lower bound differs from each other; however, the trend remains intact. This trend is an important ground for the effectiveness of MMPD.

Intuitively, by maximizing r, i.e., minimizing the largest weight vector norm of the nodes, we avoid having nodes with large weights arising from multiple input streams. Nodes with relatively larger weights often have larger load/capacity ratios than other nodes at many stream rate points.

Therefore, MMPD can also be said to balance the load of the nodes (in proportion to the nodes' capacity) *for multiple workload points*. Notice that this approach sharply contrasts with traditional load balancing schemes that optimize for single workload points.

4.3 ROD Algorithm

We now present a greedy operator distribution algorithm that seamlessly combines the heuristics discussed in the previous section.

In this section, we assume that the load coefficients of all operators are known. Operators that read or write the same table are grouped as a single allocation unit so that they are always assigned to the same node. In other words, we treat the operators that read or write the same table as a single operator in ROD algorithm. The load coefficients of this "grouped" operator are the summation of the load coefficients of all operators in the group.

The algorithm consists of two phases: the first phase orders the operators and the second one greedily places them on the available nodes.

4.3.1 Phase 1: Operator Ordering

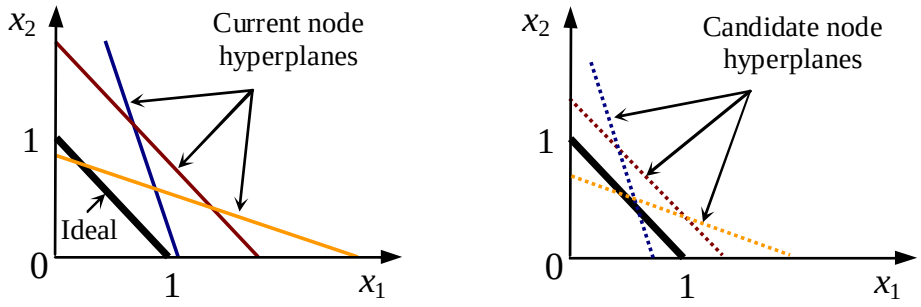
This phase sorts the operators in descending order based on the second norm of their load coefficient vectors. The reason for this sorting order is to enable the second phase to place "high impact" operators (i.e., those with large norms) early on in the process, since dealing with such operators late may cause the system to significantly deviate from the optimal results. Similar sorting orders are commonly used in greedy load balancing and bin packing algorithms [28].

4.3.2 Phase 2: Operator Assignment

The second phase goes through the ordered list of operators and iteratively assigns each to one of the n candidate nodes. Our basic destination node selection policy is greedy: at each operator assignment step, the assignment that *minimally* reduces the final feasible set is chosen.

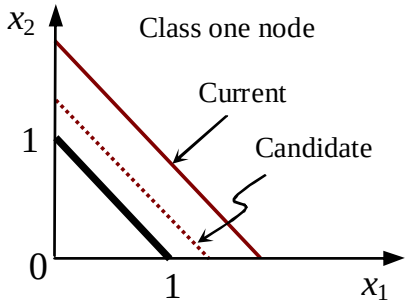
At each step, we separate nodes into two classes. Class I nodes consist of those that, if chosen for assignment, will not lead to a reduction in the final feasible set. Class II nodes are the remaining ones. If it exists, one of the nodes in Class I is chosen using a goodness function (more about this below). If not, then the operator is assigned to the Class II node with the maximum *candidate* plane distance (i.e., the distance after the assignment).

Let us now describe the algorithm in more detail while providing geometrical intuition. Initially, all the nodes are empty. Thus, all the node hyperplanes are at "infinity". The node hyperplanes move closer to the origin as operators get assigned. The feasible set size at each step is given by the space that is below all the node hyperplanes. Class I nodes are those whose candidate hyperplanes are above the ideal hyperplane, whereas the candidate hyperplanes of Class II nodes

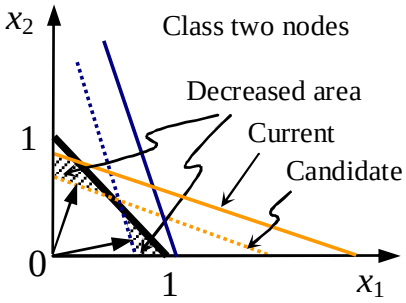


(a) Current node hyperplanes.

(b) Candidate node hyperplanes.



(c) Class I node hyperplanes.



(d) Class II nodes hyperplanes.

Figure 4.8: Node selection policy example.

are either entirely below, or intersect with, the ideal hyperplane. Figure 4.8(a) and 4.8(b) show, respectively, the current and candidate hyperplanes of three nodes, as well as the ideal hyperplane.

Since we know that the feasible set size is bounded by the ideal hyperplane, at a given step, choosing a node in Class I will not reduce the possible space for the final feasible set. Figure 4.8(c) shows an example of the hyperplane of a Class I node. Notice that as we assign operators to Class I nodes, we push the axis intersection points closer to those of the ideal hyperplane, thus follow the MMAD heuristic. If no Class I nodes exist, then we have to use a Class II node and, as a result, we inevitably reduce the feasible set. An example of two Class II nodes and the resulting decrease in the feasible set size are shown in Figure 4.8(d). In this case, we follow the MMPD heuristic and use the plane distance to make our decision by selecting the node that has the largest candidate plane distance.

As described above, choosing any node from Class I does not affect the final feasible set size in this step. Therefore, a random node can be selected or we can choose the destination node using some other criteria. For example, we can choose the node that results in the minimum number of inter-node streams to reduce data communication overheads for scenarios where this is a concern.

The pseudocode of the algorithm is shown in Figure 4.9.

Initialization

$C_T \leftarrow C_1 + \cdots + C_n$
for $k = 1, \cdots, d$, $l_k \leftarrow l^o_{1k} + \cdots + l^o_{mk}$
for $i = 1, \cdots, n$, $j = 1, \cdots, m$, $a_{ij} \leftarrow 0$
for $i = 1, \cdots, n$, $k = 1, \cdots, d$, $l^n_{ik} \leftarrow 0$

Operator Ordering

Sort operators by $\sqrt{{l^o_{j1}}^2 + , \cdots , + {l^o_{jd}}^2}$ in descending order
(let $h_1, \cdots, h_m$ be the sorted operator indices)

Operator Assignment

for $j = h_1, \cdots, h_m$
  (assign operator o_j)
  class I nodes $\leftarrow \phi$, class II nodes $\leftarrow \phi$
  for $i = 1, \cdots, n$
    (classify node N_i)
    for $k = 1, \cdots, d$, $w'_{ik} \leftarrow \left(\left(l^n_{ik} + l^o_{jk} \right) / l_k \right) / \left(C_i / C_T \right)$
    if $w'_{ik} \leq 1$ for all $k = 1, \cdots, d$
      add N_i to class I nodes
    else
      add N_i to class II nodes
  if class I is not empty
    select a destination node from class I
  else
    select the node with $\min_i \frac{1}{\sqrt{{w'_{i1}}^2 + \cdots + {w'_{id}}^2}}$ from class II
  Assign o_j to N_s (assume N_s is selected)
    $a_{sj} \leftarrow 1$;
    for $k = 1, \cdots, d$, $l^n_{sk} \leftarrow 1^n_{sk} + l^o_{jk}$

Figure 4.9: ROD algorithm pseudocode.

4.4 Algorithm Extensions

4.4.1 General Lower Bounds on Input Rates

We have so far leveraged no knowledge about the expected workload, assuming

$$D = \left\{ R : R \geq (0, \cdots, 0)^T \right\}.$$

We now present an extension where we allow more general, non-zero lower bound values for the stream rates, assuming:

$$D = \{ R : R \geq R_L,\ R_L = (r_{L1}, \cdots, r_{Ld})^T,\ r_{Lk} \geq 0 \text{ for } k = 1, \cdots, d \}.$$

This general lower bound extension is useful in cases where it is known that the input stream rates are strictly, or likely, larger than a workload point R_L. Using point R_L as the lower bound is equivalent to ignoring those workload points that never or seldom happen; i.e., we optimize the system for workloads that are more likely to happen.

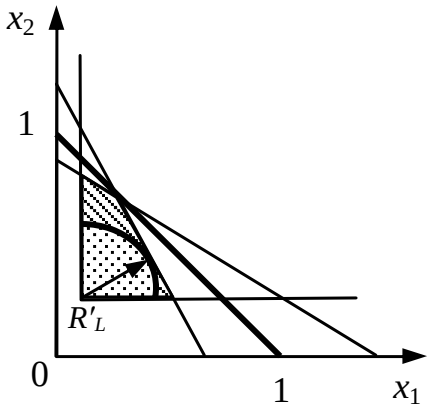


Figure 4.10: Feasible set with lower bound B'.

The operator distribution algorithm for the general lower bound is similar to the base algorithm discussed before. Recall that the ideal node load coefficient matrix does not depend on D. Therefore, our first heuristic, MMAD, remains the same. In the second heuristic, MMPD, because the lower bound of the feasible set size changes, the center of the partial hypersphere should also change. In the normalized space, the lower bound corresponds to the point

$$R_L^{'} = \begin{pmatrix} \frac{r_{L1} l_1}{C_T} \\ \vdots \\ \frac{r_{Ld} l_d}{C_T} \end{pmatrix}.$$

In this case, we want to maximize the radius of the partial hypersphere centered at $R_L^{'}$ within the normalized feasible set (e.g., Figure 4.10). The formula of the radius r is

$$r = \min_i \frac{1 - W_i R_L^{'}}{\|W_i\|}.$$

With this extension, the ROD algorithm needs to be changed slightly: when we choose a class II node, we simply replace the distance from the origin to the candidate node hyperplanes with the distance from the lower bound point to the candidate node hyperplanes.

Remark: This extension is necessary only when the real lower bound point is not close to the origin and when the weights of some input streams can not be well balanced. Recall that the ideal feasible set does not depend on the workload set D. If the weights of all input streams can be well balanced using the origin as the lower bound, the resulting plan will be close to the ideal load distribution plan, which is good regardless of the extra information of system input stream rates lower bound.

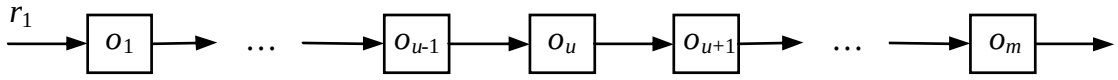


Figure 4.11: A chain query graph for Example 8.

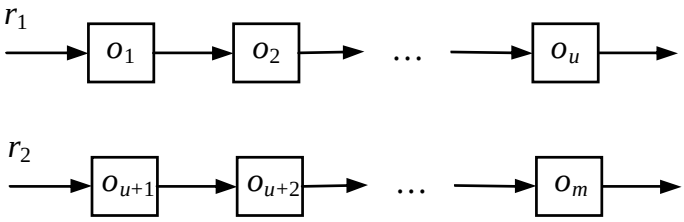


Figure 4.12: Linear pieces of the chain query graph in Example 8.

4.4.2 Nonlinear Load Models

Our discussion so far has assumed linear systems; i.e., those that use linear load models. In this section, we generalize our discussion to deal with nonlinear systems.

Our key technique is to introduce new variables such that the load functions of all operators can be expressed as linear functions of the actual system input stream rates as well as the newly introduced variables. Our linearization technique is best illustrated with two simple examples.

EXAMPLE 8: First, consider a chain of operators as depicted in Figure 4.11. The load functions of the operators in this chain can be computed as

$$\begin{aligned} l(o_1) &= c_1 \cdot r_1 \\ l(o_j) &= \left(\prod_{k=1}^{j-1} s_k\right) \cdot c_j \cdot r_1 \ , \quad j = 2, \cdots, m. \end{aligned}$$

Assume that all operator costs and selectivities except for the selectivity of operator o_u are constant. In this case, the load functions of operator o_1 to o_u are linear functions of r_1, but the load functions of o_{u+1} to o_m are not linear because they contain $s_u \cdot r_1$. To transform the load functions of o_{u+1} to o_m to linear functions, we add the input stream rate of operator o_{u+1} as a new variable r_2. The load functions of operator o_{u+1} to o_m are then written as

$$\begin{aligned} l(o_{u+1}) &= c_{u+1} \cdot r_2 \\ l(o_j) &= \left(\prod_{k=u+1}^{j-1} s_k\right) \cdot c_j \cdot r_2 \ , \quad j = u+2, \cdots, m, \end{aligned}$$

which are linear functions of r_2.

This linearization technique can also be considered as "cutting" a nonlinear query graph into linear pieces. For example, the above transformed load model is the same as the linear load model of the query graph in Figure 4.12.

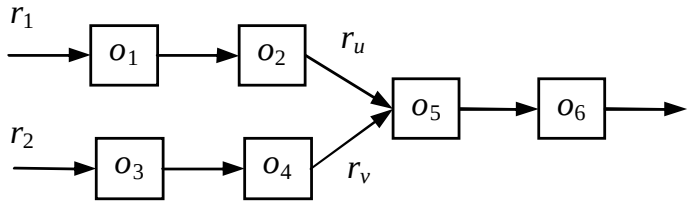


Figure 4.13: A query graph with a Join operator for Example 9.

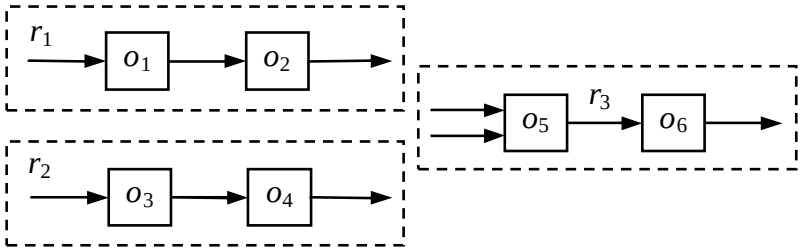


Figure 4.14: Linear pieces of the query graph in Example 9.

EXAMPLE 9: Consider the query graph in Figure 4.13. Assume that the selectivities and costs of all operators are constants. Operator o_5 is a Join operator that matches tuples whose timestamps are within a give time window w. Let o_5's input stream rates be r_u and r_v, its selectivity (per tuple pair) be s_5, and its processing cost be c_5 CPU cycles per tuple pair. The number of tuple pairs to be processed in unit time is wr_ur_v. The load of o_5 is thus $c_5wr_ur_v$ and the output stream rate of this operator is $s_5wr_ur_v$. As a result, it is easy to see that the load function of o_5 cannot be expressed as a linear function of r_1 and r_2. In addition, the input to o_6 cannot be expressed as a linear function of r_1 and r_2 either. The solution is to introduce the output stream rate of operator o_5 as a new variable r_3. It is easy to see that the load of operator o_6 can be written as a linear function of r_3. Less apparent is the fact that the load of operator o_5 can also be written as $(c_5/s_5)r_3$, which is a linear function of r_3 (assuming c_5 and s_5 are constant). Therefore, the load functions of the entire query graph can be expressed as linear functions of four variables r_1 to r_3. The linear pieces of this query graph is shown in Figure 4.14

This linearization technique is general; i.e., it can transform any nonlinear load model into a linear load model by introducing additional input variables. Once the system is linear, the analysis and techniques presented earlier apply. However, because the performance of ROD depends on whether the *weights* of each variable can be well balanced across the nodes, we aim to introduce as few additional variables as possible.

4.4.3 Extension with Communication CPU Overhead Consideration

So far, we have ignored the CPU overhead of data communication. We now address this issue by introducing *operator clustering* as a pre-processing step to be applied before ROD.

Table 4.1: Notations related to communication overhead.

g	number of streams
S_t	the tth stream
c_t^c	the communication cost of stream S_t
$l^c(S_t)$	the communication load of stream S_t
l_{tk}^c	the communication load coefficient of stream S_t for input I_k
r_{tu}^c	the communication ratio between S_t and o_u
r_t^c	the communication clustering ratio of S_t
w_t^c	the clustering weight of S_t
η^c	the communication clustering ratio threshold
ψ	the clustering weight upper bound

Before discussing how to cluster operators, we first introduce some definitions and notations that are related to the data communication overhead. These notations are also summarized in Table 4.1. We denote the streams in the system by $S_1, \cdots, S_g$. We define the following statistics based on a given time interval:

- The *communication cost* of a stream is defined as the average number of CPU cycles needed to transfer a tuple on this stream from one node to another node (sending CPU cycles plus receiving CPU cycles). The communication cost of stream S_t is denote by c_t^c.

- The *communication load* of a stream is defined as the number of CPU cycles needed to transfer all tuples of that stream in the given time interval divided by the duration of the time interval. It can be computed as the product of the communication cost and the data rate of the stream. The communication load of stream S_t is denoted by $l^c(S_t)$.

- In a linear system, the communication load of all streams can be written as linear functions of system input stream rates. We express the communication load function of stream S_t as

$$l(S_t) = l_{t1}^c r_1 + l_{t2}^c r_2 + \cdots + l_{td}^c r_d,$$

  where l_{tk}^c is called the *communication load coefficient* of stream S_t for input stream I_k. The communication load coefficients of the steams can be computed from the communication cost of the streams and the selectivities of the operators.

- Suppose stream S_t is connected to operator o_u and o_v. We define

$$r_{tu}^c \doteq \max_k \frac{l_{tk}^c}{l_{uk}^o}$$

as the *communication ratio* between S_t and o_u, and

$$r^c_{tv} \doteq \max_k \frac{l^c_{tk}}{l^o_{vk}}$$

as the *communication ratio* between between S_t and o_v.

- We define

$$r^c_t \doteq \max\left(r^c_{tu}, r^c_{tv}\right)$$

as the *communication clustering ratio* of stream S_t.

Obviously, if the communication ratios between all streams and their connected operators are very small, then the percentage of CPU cycles used for data communication is very small regardless of the system input stream rates and the operator distribution plan. In this case, operator clustering is not necessary because data communication overhead is negligible.

On the other hand, if the communication ratio between a stream and one of its connected operators is large, then separating the operators connected by this stream to different nodes may result in large data communication overhead. To avoid such situations, we cluster operators connected by such streams into *super-operators*, and then use ROD to distribute the super-operators. We studied two greedy operator clustering approaches. Both approaches follow the following steps:

1. Compute the communication clustering ratio of each stream.
2. Select a stream with communication clustering ratio larger than a given threshold η^c.
3. Cluster the end-operators of the selected stream as a new super-operator.
4. Update the communication clustering ratio of the streams connected to the new super-operator.
5. Repeat step 2 to 4 until no more operators can be clustered.

The two greedy approaches differ in how they select a stream in step 2. The first approach, called Largest-Ratio-First, selects the stream with the largest communication clustering ratio. A potential problem with this method is that it may create operator clusters with very large weights. Therefore, the second approach tries to avoid this by choosing the stream with the minimum *clustering weight*. The clustering weight of stream S_t that connects super-operator o_u and o_v is defined as

$$w^c_t \doteq \max_k \frac{n\left(l^o_{uk} + l^o_{vk}\right)}{l_k}.$$

This approach is then called Min-Weight-First. To avoid having super-operators with very large weights, we set an upper bound (ψ) on the clustering weight of the selected streams in both approaches. In other words, we do not cluster super-operators connected by a stream if the clustering weight of the stream is larger than ψ.

The pseudocode for these two operator clustering approaches is shown in Figure 4.15

Initialization
$\Omega \leftarrow \{o_1, \cdots, o_m\}$ (initialize the set of operator clusters or super-operators)
$\Gamma \leftarrow \phi$ (initialize the set of candidate streams as an empty set)
for $t = 1, \cdots, g$
(suppose stream S_t is connected to operator o_u and o_v)
$r^c_{tu} \leftarrow \max_k \frac{l^c_{tk}}{l^o_{uk}}$, $r^c_{tv} \leftarrow \max_k \frac{l^c_{tk}}{l^o_{vk}}$
$r^c_t \leftarrow \max(r^c_{tu}, r^c_{tv})$
$w^c_t \leftarrow \max_k \frac{n(l^o_{uk}+l^o_{vk})}{l_k}$
if $r^c_t > \eta^c$ and $w^c_t < \psi$, add S_t to Γ
Sream Ordering
if clustering approach is Max-Min-First
order streams in Γ by communication clustering ratio in descending order
if clustering approach is Min-Weight-First
order streams in Γ by clustering weight in ascending order
Operator Clustering
while Γ is not empty
let S_t be the first candidate stream in Γ
(suppose S_t is connected to super-operator o_u and o_v)
cluster the operators connected by S_t to a new super-operator o_c
for $k = 1, \cdots, d$, $l^o_{ck} \leftarrow l^o_{uk} + l^o_{vk}$
delete o_u and o_v from Ω
add o_c to Ω
for each stream (denoted by S_e) connected to o_c
(suppose stream S_e is connected to o_c and o_f)
delete S_e from Γ
$r^c_{ec} \leftarrow \max_k \frac{l^c_{ek}}{l^o_{ck}}$, $r^c_{ef} \leftarrow \max_k \frac{l^c_{ek}}{l^o_{fk}}$
$r^c_e \leftarrow \max\left(r^c_{ec}, r^c_{ef}\right)$
$w^c_e \leftarrow \max_k \frac{n(l^o_{ck}+l^{o'}_{vk})}{l_k}$
if $r^c_e > \eta^c$ and $w^c_e < \psi$
add S_e to ordered Γ
delete S_t from Γ

Figure 4.15: The operator clustering pseudocode for communication overhead reduction.

By varying η^c and ψ, we obtain different operator distribution plans. Our experimental analysis of these approaches did not yield a clear winner when considering various query graphs. Our current practical solution is to generate a small number of operator clustering plans for each of these approaches by systematically varying η^c and ψ. Then, we use ROD to distribute the super-operators without considering data communication overhead. After the operators are distributed, we update the node load functions by adding the communication load coefficients. Finally, we pick the operator distribution plan with the maximum plane distance.

4.4.4 Extension with Network Bandwidth Consideration

Our system consists of a cluster of Borealis interconnected by a high bandwidth local network. For most stream based applications, network bandwidth is usually not a bottleneck of the system.

Table 4.2: Notations related to bandwidth constraints.

C_i^b	the bandwidth capacity of node N_i
$C^b = \left(C_1^b,\ \cdots,\ C_n^b\right)$	the node bandwidth capacity vector
C_T^b	the total bandwidth capacity of the system
$b(S_t)$	the bandwidth of stream S_t
$b(N_i)$	the bandwidth consumption of node N_i
b_{tk}^s	the bandwidth coefficient of stream S_t for input I_k
b_{ik}^n	the bandwidth coefficient of node N_i for input I_k
B^n	the bandwidth coefficient matrix
r_{tu}^b	the bandwidth ratio between S_t and o_u
r_t^b	the bandwidth clustering ratio of S_t
η^b	the bandwidth clustering ratio threshold

However, if the system consists of many operators with very small costs (such as Filter and Union operators), the network bandwidth may become saturated before the nodes are overloaded. For example, we ran a single filter on a Borealis node on a machine with a Athlon 2800+ CPU, and 100Mbps Ethernet connection. When we increased the input stream rate of the operator, and when the tuples size was larger than 70 bytes, the network bandwidth became saturated before the CPU of the node was overloaded

For applications that could stress the network bandwidth resource, we also cluster operators as a pre-processing step of ROD to reduce network bandwidth consumption. The operator clustering algorithm is similar to the strategy introduced above. The difference is that we will extend the definition of feasible set to incorporate bandwidth constraints, and we will use different ratios in the Max-Ratio-First approach. In what follows, we first introduce the definitions and notations that are related to the bandwidth constrains. These notations are summarized in Table 4.2. Then we present the operator clustering algorithm.

Definitions and Notations

We define the *bandwidth* of a stream as the tuple size of the stream times its stream rate. We say a system has a *linear bandwidth model* if the bandwidth of all its streams can be written as linear functions. For example, if all operator selectivies are constants, then the bandwidths of all streams can be written as linear functions of system input stream rates.

Under the linear bandwidth model, we express the bandwidth functions of all stream as

$$b(S_t) = b_{t1}^s r_1 + b_{t2}^s r_2 + \cdots + b_{td}^s r_d, \quad t = 1,\ \cdots,\ g,$$

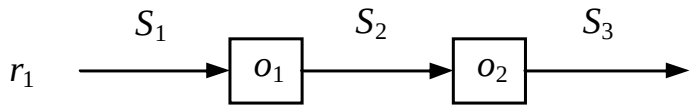


Figure 4.16: Example 10 query graph.

where b^s_{tk} is called the *bandwidth coefficient* of stream S_t for input stream I_k. The bandwidth coefficients can be computed from the selectivities of the operators and the tuple size of the streams.

EXAMPLE 10: Consider the query graph in Figure 4.16. Suppose the tuple size of stream S_t is h_t. Then the bandwidth of the streams are computed as the follows:

$$\begin{aligned} b(S_1) &= h_1 \\ b(S_2) &= s_1 \cdot h_2 \\ b(S_3) &= s_1 \cdot s_2 \cdot h_3. \end{aligned}$$

Given an operator distribution plan, the bandwidth consumption of a node is defined as the total bandwidth of all inter-node streams between the node and other nodes. We express the bandwidth functions of all nodes in the system as

$$b(N_i) = b^n_{i1} r_1 + b^n_{i2} r_2 + \cdots + b^n_{id} r_d, \quad i = 1, \cdots, n,$$

where b^n_{ik} is called the *bandwidth coefficient* of node N_i for input stream I_k. We express the total bandwidth consumption of all nodes in the system as

$$b_T = b_1 r_1 + b_2 r_2 + \cdots + b_d r_d, \quad i = 1, \cdots, n,$$

where b_i called the *bandwidth coefficient* of the system. The bandwidth coefficients of the nodes and the system can be computed from the bandwidth coefficients of the inter-node streams.

Putting the bandwidth coefficients in a matrix, we get a n by d *bandwidth coefficient matrix*

$$B^n = \begin{pmatrix} b^n_{11} & b^n_{12} & \cdots & b^n_{1d} \\ b^n_{21} & b^n_{22} & \cdots & b^n_{2d} \\ \vdots & \vdots & \ddots & \vdots \\ b^n_{n1} & b^n_{n2} & \cdots & b^n_{nd} \end{pmatrix}.$$

Let C^b_i denote the bandwidth capacity between node N_i and the local network, for $i = 1, \cdots,$

n. Let C_T^b denote the total bandwidth of the local network. Let

$$C^b \doteq \begin{pmatrix} C_1^b \\ C_2^b \\ \vdots \\ C_n^b \end{pmatrix},$$

and

$$B_T \doteq \begin{pmatrix} b_1 & b_2 & \cdots & b_d \end{pmatrix}.$$

Then the network of the system is feasible if and only if

$$B_i^n R \leq C_i^b \quad \text{for } i = 1, \cdots, n$$

and

$$B_T R \leq C_T^b.$$

We say the system is feasible if and only if all nodes in the system are feasible and the network of the system is feasible. Thus, we extend the definition of feasible set as follows:

DEFINITION 2 (**Extended Feasible Set**): *Given a CPU capacity vector C, an operator load coefficient matrix L^o, a workload set D, a bandwidth capacity vector C^b, a total bandwidth capacity C_T^b, and the bandwidth coefficients of all streams, the feasible set of the system under operator distribution plan A (denoted by $F(A)$) is defined as the set of all points in the workload set D for which the system is feasible, i.e.,*

$$F(A) = \left\{ R : R \in D,\ AL^o R \leq C,\ B^n R \leq C^b,\ B_T R \leq C_T^b \right\},$$

where B^n is the node bandwidth coefficient matrix under operator distribution plan A.

The bandwidth resource constraints add $n+1$ linear constraints to the definition of feasible set. The first n constraints define n *node bandwidth hyperplanes*. The last constraint defines an extra bandwidth hyperplane for the total bandwidth consumption of the shared network. The feasible set is the set bellow all of the $2n+1$ hyperplanes (n node load hyperplane and $n+1$ bandwidth hyperplane).

Figure 4.17 and 4.18 illustrate two example feasible sets when $n = 2$ and $d = 2$. The solid lines correspond to the node hyperplanes. The dashed lines correspond to the bandwidth hyperplanes. The thick solid line corresponds to the ideal hyperplane. The feasible sets are the shaded areas below all lines.

In Figure 4.17, all the bandwidth hyperplanes are above the node hyperplanes. Therefore, the shape of the feasible set is determined by the node hyperplanes only. We say a system is under *abundant bandwidth situation* if, no matter how the operators are distributed in the system, the

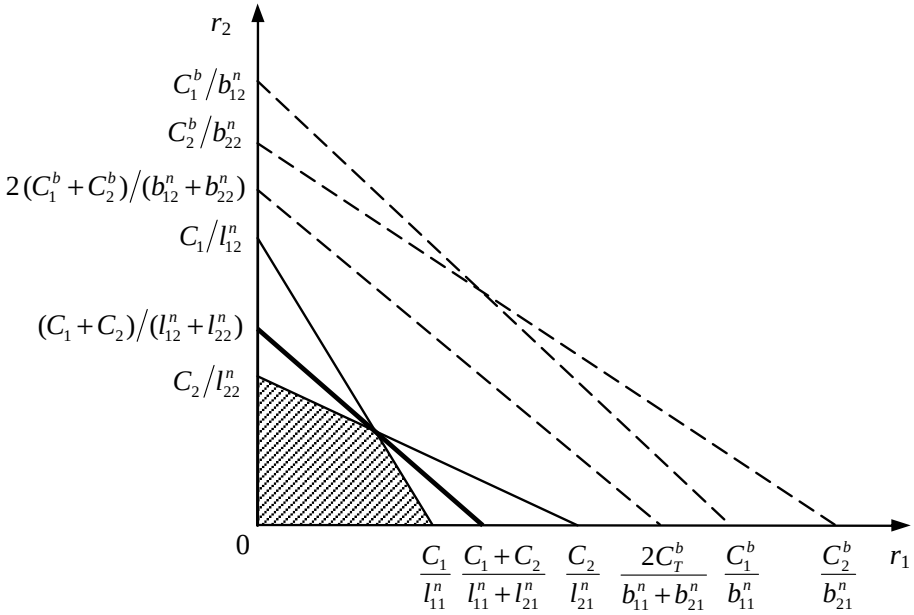


Figure 4.17: Feasible set under abundant bandwidth situation.

bandwidth hyperplane of any node in the system is always above the load hyperplane of that node, and the total bandwidth hyperplane is always above the ideal hyperplane. In this case, the feasible sets are always determined by the node hyperplanes only. Thus, we do not need to consider bandwidth constraints at all when distributing the operators. Next, we discuss how to determine whether a system is under abundant bandwidth situation.

Let b_{jk}^o denote the total bandwidth coefficients of all streams that connect to operator o_j for input stream I_k. Suppose o_j is put on node N_i and no other operators are on N_i. If

$$\max_k \frac{b_{jk}^o}{l_{jk}^o} < \frac{B_i}{C_i}, \tag{4.9}$$

then the CPU usage (percentage of busy time) of N_i is always larger then the bandwidth usage of N_i no mater what the input streams rate are. In addition, if all operators on node N_i satisfy condition 4.9, then the CPU usage of the node is always larger than the bandwidth usage of the node, and the node hyperplane of N_i is lower than the bandwidth hyperplane of N_i. Therefore, if all operators in the system satisfy

$$\max_k \frac{b_{jk}^o}{l_{jk}^o} < \min_i \frac{B_i}{C_i}, \tag{4.10}$$

no matter how the operators are distributed on the nodes, all node hyperplanes in the system are always below the bandwidth hyperplane for the same node. On the other hand, If some operator does not satisfy condition 4.10, then, there exist operator distribution plans that will make the bandwidth hyperplane of a node either be below, or intersect with, its node hyperplane.

In addition, the total bandwidth hyperplane of the system is above the ideal hyperplane if and

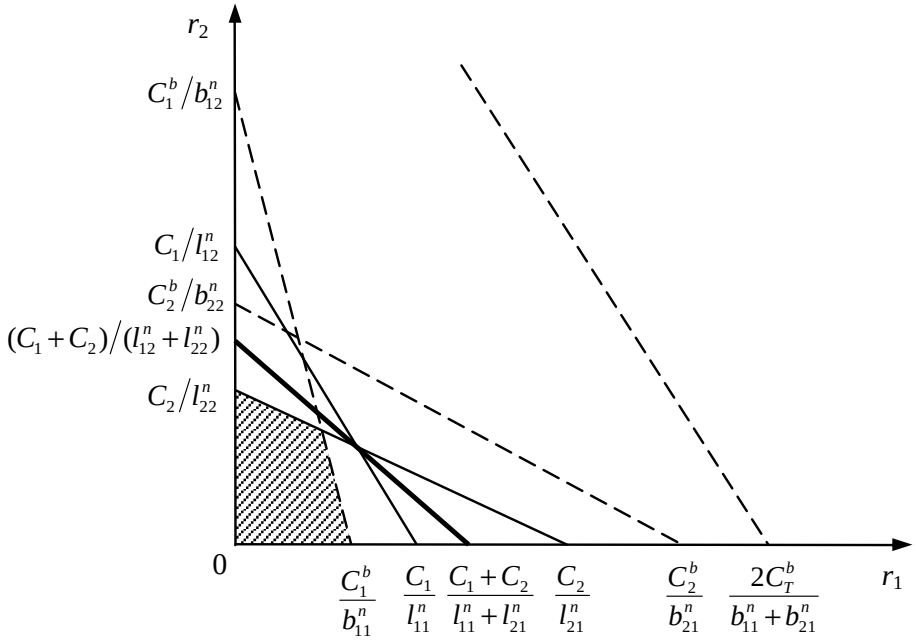


Figure 4.18: Feasible set under limited bandwidth situation.

only if

$$\max_k \frac{\sum_{t=1}^{w} b_{tk}^s}{\sum_{j=1}^{m} l_{jk}^o} < \frac{\sum_{i=1}^{n} C_i}{\sum_{i=1}^{n} B_i}.$$

Therefore, we have that

THEOREM 3: *A system is under abundant bandwidth situation if and only if*

$$\begin{aligned} \max_k \frac{b_{jk}^o}{l_{jk}^o} &< \min_i \frac{B_i}{C_i} \quad \textit{for } j = 1, \cdots, m, \\ \max_k \frac{\sum_{t=1}^{w} b_{tk}^s}{\sum_{j=1}^{m} l_{jk}^o} &< \frac{\sum_{i=1}^{n} C_i}{\sum_{i=1}^{n} B_i}. \end{aligned}$$

We call the other cases in which the above two inequalities do not hold the *limited bandwidth situation* (e.g. Figure 4.18). In this case, the bandwidth hyperplanes can be lower than the ideal hyperplane and affect the size of the achieved feasible set. In other words, the network bandwidth may become saturated before the CPU of the nodes are overloaded. In this situation, we must consider the bandwidth constraints in the operator placement algorithm.

Operator Clustering Under Limited Bandwidth Situation

The impact of operator placement on the bandwidth hyperplanes are very different from its impact on the load hyperplanes. By clustering connected operators, we can reduce network bandwidth consumption, and thus make the bandwidth hyperplanes far from the origin. However, such effort may lower the node load hyperplanes, because it may result in large node load coefficients.

Therefore, we need to find a good balance between the positions of the bandwidth hyperplanes and node load hyperplanes.

Our solution is to use an operator clustering algorithm, which is similar to the one introduced previously, to generate a set of operator clustering plans. Then we apply ROD on the clustered super-operators without considering the bandwidth constraints. Finally, we compare the minimum plane distance of the resulting plans and pick the one with the largest minimum plane distance. The minimum plane distance is now defined as the minimum distance from the lower bound point to all hyperplanes (including both the node load hyperplanes and the bandwidth hyperplanes).

The operator clustering algorithm for bandwidth usage reduction is very similar to the operator clustering algorithm that we introduced earlier for communication CPU overhead reduction.

In the operator clustering algorithm, we also use two greedy operator clustering approaches: Max-Ratio-First and Min-Weight-First. These approaches are similar to the approaches introduced previously for data communication CPU overhead reduction. The difference is that the communication clustering ratio used in the previous approaches is replaced by the *bandwidth clustering ratio*. We define the *bandwidth ratio* between stream S_t and its end-operators (or super-operators) o_u and o_v as

$$r^b_{ut} \doteq \max_k \frac{b^s_{tk}}{l^o_{uk}},$$

$$r^b_{vt} \doteq \max_k \frac{b^s_{tk}}{l^o_{vk}}.$$

The *bandwidth clustering ratio* of stream S_t is then defined as

$$r^b_t \doteq \max\left(r^b_{ut}, r^b_{vt}\right).$$

We also replace the communication clustering ratio threshold η^c in the previous Max-Ratio-First approach by the bandwidth clustering ratio threshold η^b in the new Max-Ratio-First approach. The rest of the Max-Ratio-First approach and the Min-Weight-First approach remain the same.

The pseudocode for these operator clustering approaches is shown in Figure 4.19

Combining the Two Operator Clustering Algorithms

We have introduced two operator clustering algorithms separately: one for reducing the communication CPU overhead, the other one for reducing the network bandwidth consumption. Next, we discuss how to integrate both operator clustering algorithms in a single framework.

Given η^c, η^b, and ψ, we now consider a stream as a candidate stream for operator clustering if its clustering weight is less than ψ and either its communication clustering ratio is larger than η^c or its bandwidth clustering ratio is larger than η^b. We use a greedy strategy to select a stream from the candidate streams and cluster the operators connected by the selected stream. This stream selection and operator clustering process is repeated until no more candidate streams are available.

We use two stream selection policies to generate operator clustering plans. In the Min-Weight-First policy, we always select the stream with the minimum clustering weight. In the Max-Ratio-First policy, we separate the candidate streams into three classes. The streams in the first class

Initialization
$\Omega \leftarrow \{o_1, \cdots, o_m\}$ (initialize the set of operator clusters or super-operators)
$\Gamma \leftarrow \phi$ (initialize the set of candidate streams as an empty set)
for $t = 1, \cdots, g$
(suppose stream S_t is connected to operator o_u and o_v)
$r^b_{tu} \leftarrow \max_k \frac{b^s_{tk}}{l^o_{uk}}$, $r^b_{tv} \leftarrow \max_k \frac{b^s_{tk}}{l^o_{vk}}$
$r^b_t \leftarrow \max\left(r^b_{tu}, r^b_{tv}\right)$
$w^c_t \leftarrow \max_k \frac{n(l^o_{uk}+l^o_{vk})}{l_k}$
if $r^b_t > \eta^b$ and $w^c_t < \psi$, add S_t to Γ
Sream Ordering
if operator clustering approach is Max-Min-First
order streams in Γ by bandwidth clustering ratio in descending order
if operator clustering approach is Min-Weight-First
order streams in Γ by clustering weight in ascending order
Operator Clustering
while Γ is not empty
let S_t be the first candidate stream in Γ
(suppose S_t is connected to super-operator o_u and o_v)
cluster the operators connected by S_t to a new super-operator o_c
for $k = 1, \cdots, d$, $l^o_{ck} \leftarrow l^o_{uk} + l^o_{vk}$
delete o_u and o_v from Ω
add o_c to Ω
for each stream (denoted by S_e) connected to o_c
(suppose stream S_e is connected to o_c and o_f)
delete S_e from Γ
$r^b_{ec} \leftarrow \max_k \frac{b^s_{ek}}{l^o_{ck}}$, $r^b_{ef} \leftarrow \max_k \frac{b^s_{ek}}{l^o_{fk}}$
$r^b_e \leftarrow \max\left(r^b_{ec}, r^b_{ef}\right)$
$w^c_e \leftarrow \max_k \frac{n(l^o_{ck}+l^o_{vk})}{l_k}$
if $r^c_e > \eta^c$ and $w^c_e < \psi$
add S_e to ordered Γ
delete S_t from Γ

Figure 4.19: The operator clustering pseudocode for network bandwidth consumption reduction.

have both a communication clustering ratio larger than η^c and a bandwidth clustering ratio larger than η^b. The streams in the second class only have a communication clustering ratio larger than η^c. The streams in the third class only have a bandwidth clustering ratio larger than η^b. We can either choose to consider the communication clustering ratio first or consider the bandwidth clustering ratio first. For example, if we consider the communication clustering ratio first, then we order the streams in the first class by the communication clustering ratio. When selecting streams, if the first class is not empty, we select the stream from this class with the with the largest communication clustering ratio. Otherwise, if the second class is not empty, we select the stream in the second class with the largest communication ratio. If both the first and the second classes are empty, we select the stream from the third class with the largest bandwidth clustering ratio.

Again, by varying η^c, η^b and ψ, we obtain multiple operator clustering plans. We then compare the minimum plane distance (considering both load hyperplanes and bandwidth hyperplanes) of different operator distribution plans after applying ROD and select the one with the largest minimum plane distance.

4.5 Complexity Analysis

The computation of ROD can be divided into four phases:

1. Statistics initialization, which include
   - Computing the load coefficients of the operators
   - Computing the communication load coefficients of the streams
   - Computing the bandwidth coefficients of the streams
2. Operator clustering.
3. Operator ordering.
4. Operator assignment.

The last three phases are repeated several times with different operator clustering parameters.

The computation complexity of the statistic initialization process is $O(md + gd)$, where computing the operator load coefficients takes time $O(md)$ and computing the stream communication load coefficients and bandwidth coefficients takes time $O(gd)$.

The computation complexity of the operator clustering phase is computed as the follows:

1. Checking whether the system is under abundant bandwidth situation takes time $O(md)$.
2. Computing the communication clustering ratios and the bandwidth clustering ratios of all streams takes time $O(gd)$.
3. Ordering the streams takes time $O(g \log g))$.
4. Clustering two operators takes constant time.
5. Updating the stream clustering ratio in the ordered stream queue takes time $O(\log g)$.
6. The above two steps are repeated at the most g times.

Thus, the computation complexity of the operator clustering phase is $O(md + gd + g \log g)$.

The computation complexity of the operator ordering phase is $O(md + m \log m)$, where computing the second norm of the load vectors of the operators takes time $O(md)$ and ordering the m operators takes time $O(m \log m)$. The pseudocode of the operator assignment phase contains three nested loops (see 4.9). Thus, its computation complexity is $O(mnd)$. Therefore, the total computation complexity of the basic ROD algorithm is $O(m \log m + mnd)$.

If operator clusterings are repeated p times, then the ROD algorithm is also repeated p times. Thus, the total computation complexity of whole process is $O\left(p(gd + w \log w + m \log m + mnd)\right)$.

Chapter 5

Dynamic Operator Distribution

In this chapter, we address the dynamic operator distribution problem. We assume that the operators are first clustered using the operator clustering algorithm introduced in the previous chapter so that the CPU overhead for data communication is negligible and the network bandwidth is not a bottleneck of the system. We further assume that all operator clusters that contain operators that are impractical to be moved on the fly are already distributed to the nodes by ROD. In this chapter, we only consider the the initial and dynamic re-distribution of operator clusters with relatively small load migration overheads. For simplicity of exposition, we treat each operator cluster as a single operator.

In order to adapt to the most recent load changes dynamically, our algorithms use the time series load model to make operator distribution decisions (the load time series of an operator cluster is the total load time series of all operators in the cluster). We refer the average load of a time series as its load for simplicity. Our optimization goal is to produce and maintain load balanced operator mapping plans where the average variance of the load time series of all nodes is minimized and the average load correlation of all node pairs is maximized.

In this chapter, we first discuss how to minimize the average load variance and maximize the average load variation in the ideal case, in which we assume that the total load time series of the system can be arbitrarily distributed to the nodes. Given the insights obtained from the ideal case, we then explore heuristics-based operator distribution algorithms that assign operators to nodes in a greedy manner. We present both a global algorithm for initial operator placement and some pair-wise algorithms for dynamic operator migration. The key idea underlying these algorithms is to separate operators with large load correlation to different nodes and put operators with small load correlation on the same node. For simplicity of exposition, we first assume that all nodes are homogeneous. This assumption is relaxed in Section 5.3, in which we extend our algorithms to deal with heterogeneous nodes. At the end of the chapter, we discuss the computation complexity of the presented algorithms.

5.1 Optimization Fundamentals

Before discussing our algorithm, it is beneficial to know how to minimize average load variance in the ideal case. In this section, we assume that the total load time series X of the system is fixed, and it can be arbitrarily partitioned across n nodes (this is usually unachievable). We want to find the load partition with minimum average load variance. The result is illustrated by the following theorem:

THEOREM 4: *Let the total load of the system be denoted by time series X. Let X_i be the load time series of node i, $1 \leq i \leq n$, i.e. $X = X_1 + X_2 + \cdots + X_n$. Then among all load balanced mapping plans with $\mathsf{E}X_1 = \mathsf{E}X_2 = \cdots = \mathsf{E}X_n$, the average load variance*

$$\frac{1}{n}\sum_{i=1}^{n} \mathrm{var}X_i$$

is minimized if and only if

$$X_1 = X_2 = \cdots = X_n = \frac{X}{n}.$$

Proof. Let ρ_{ij} be the correlation coefficient between X_i and X_j. Since $X = X_1 + X_2 + \cdots + X_n$, we have that

$$\mathrm{var}X = \sum_{i=1}^{n} \mathrm{var}X_i + 2 \sum_{1 \leq i < j \leq n} \rho_{ij} \sqrt{\mathrm{var}X_i} \sqrt{\mathrm{var}X_j}$$

Since

$$-1 \leq \rho_{ij} \leq 1$$

and

$$\sqrt{\mathrm{var}X_i} \sqrt{\mathrm{var}X_j} \leq \frac{\mathrm{var}X_i + \mathrm{var}X_j}{2} \tag{5.1}$$

(the equality holds if and only if $\mathrm{var}X_i = \mathrm{var}X_j$), we have that

$$\begin{aligned} \mathrm{var}X &\leq \sum_{i=1}^{n} \mathrm{var}X_i + \sum_{1 \leq i < j \leq n} (\mathrm{var}X_i + \mathrm{var}X_j) \\ &= \sum_{i=1}^{n} \mathrm{var}X_i + (n-1)\sum_{i=1}^{n} \mathrm{var}X_i \\ &= n\sum_{i=1}^{n} \mathrm{var}X_i, \end{aligned}$$

the equality holds if and only if $\rho_{ij} = 1$ and $\mathrm{var}X_i = \mathrm{var}X_j$ for all $1 \leq i, j \leq n$.

When

$$X_1 = X_2 = \cdots = X_n,$$

we have that

$$n\sum_{i=1}^{n} X_i = n\sum_{i=1}^{n} \mathrm{var}\frac{X}{n} = \mathrm{var}X,$$

i.e., var$X$ is maximized.

Now, it suffices to show that when $\rho_{ij} = 1$ and $\mathrm{var}X_i = \mathrm{var}X_j$ for all $1 \leq i, j \leq n$,

$$X_1 = X_2 = \cdots = X_n.$$

Given $\rho_{ij} = 1$, $\mathrm{var}X_i = \mathrm{var}X_j$ and $\mathsf{E}X_i = \mathsf{E}X_j$, we have that

$$\begin{aligned}
\mathrm{cov}\,(X_i, X_j) &= \sqrt{\mathrm{var}X_i} \cdot \sqrt{\mathrm{var}X_j} && \text{(since } \rho_{ij} = 1\text{)} \\
&= \mathrm{var}X_i && \text{(since } \mathrm{var}X_i = \mathrm{var}X_j\text{)} \\
&= \mathsf{E}X_i^2 - (\mathsf{E}X_i)^2 && \text{(by definition)} \\
&= \mathsf{E}X_i^2 - \mathsf{E}X_i\mathsf{E}X_j && \text{(since } \mathsf{E}X_i = \mathsf{E}X_j\text{)} \\
&= \mathsf{E}X_j^2 - \mathsf{E}X_i\mathsf{E}X_j && \text{(by symetry)} \\
&= \mathsf{E}X_iX_j - \mathsf{E}X_i\mathsf{E}X_j && \text{(by corvariance definition).}
\end{aligned}$$

Therefore, we have that

$$\mathsf{E}X_i^2 + \mathsf{E}X_j^2 - 2\mathsf{E}X_iX_j = 0,$$

i.e.

$$\begin{aligned}
& \frac{1}{n}\sum_{k=1}^{h} x_{ik}^2 + \frac{1}{n}\sum_{k=1}^{h} x_{jk}^2 - 2\frac{1}{n}\sum_{k=1}^{h} x_{ik}x_{jk} \\
= \;& \frac{1}{n}\sum_{k=1}^{h} \left(x_{ik}^2 + x_{jk}^2 - 2x_{ik}x_{jk}\right) \\
= \;& \frac{1}{n}\sum_{k=1}^{h} \left(x_{ik} - x_{jk}\right)^2 \\
= \;& 0.
\end{aligned}$$

This gives that $x_{ik} = x_{jk}$ for $k = 1, \cdots, h$, and it holds for any $1 \leq i, j \leq n$. Thus

$$X_1 = X_2 = \cdots = X_n.$$

□

Notice that in the ideal case, when the average load variance of the system is minimized, the average load correlation of the system is also maximized. Naturally, we want to know whether the average load variance is monotonically decreasing with the average load correlation. If so, minimizing average load variance and maximizing average load correlation are then the same. Unfortunately, such a conclusion does not hold in general. It is very easy to find an counter example through simulation. However, in the case of $n = 2$, we can prove that when $\rho_{12} > 0$, the lower bound of the average load variance is a monotone decreasing function of the load correlation coefficient and when $\rho_{12} < 0$, the upper bound of the average load variance is a monotone decreasing function of the load correlation coefficient. The conclusion is shown as follows:

THEOREM 5: *Given load time series X and X_1, X_2, with $X = X_1 + X_2$. If $\rho_{12} > 0$, then*

$$\frac{\mathrm{var}X}{1+\rho_{12}} \leq \mathrm{var}X_1 + \mathrm{var}X_2 \leq \mathrm{var}X.$$

If $\rho_{12} < 0$, then

$$\mathrm{var}X \leq \mathrm{var}X_1 + \mathrm{var}X_2 \leq \frac{\mathrm{var}X}{1+\rho_{12}}.$$

Proof. Given $X = X_1 + X_2$, we have that

$$\mathrm{var}X = \mathrm{var}X_1 + \mathrm{var}X_2 + 2\rho_{12}\sqrt{\mathrm{var}X_1}\sqrt{\mathrm{var}X_2}.$$

Obviously, when $\rho_{12} > 0$,

$$\mathrm{var}X_1 + \mathrm{var}X_2 \leq \mathrm{var}X,$$

and when $\rho_{12} < 0$,

$$\mathrm{var}X_1 + \mathrm{var}X_2 \geq \mathrm{var}X.$$

Since

$$\sqrt{\mathrm{var}X_1}\sqrt{\mathrm{var}X_2} \leq \frac{\mathrm{var}X_1 + \mathrm{var}X_2}{2},$$

if $\rho_{12} > 0$, we have that

$$\begin{aligned} \mathrm{var}X &\leq \mathrm{var}X_1 + \mathrm{var}X_2 + \rho_{12}\left(\mathrm{var}X_1 + \mathrm{var}X_2\right) \\ &= \left(1+\rho_{12}\right)\left(\mathrm{var}X_1 + \mathrm{var}X_2\right). \end{aligned}$$

Thus

$$\mathrm{var}X_1 + \mathrm{var}X_2 \geq \frac{\mathrm{var}X}{1+\rho_{12}}$$

when $\rho_{12} > 0$.

If $\rho_{12} > 0$, we have that

$$\begin{aligned} \mathrm{var}X &\geq \mathrm{var}X_1 + \mathrm{var}X_2 + \rho_{12}\left(\mathrm{var}X_1 + \mathrm{var}X_2\right) \\ &= \left(1+\rho_{12}\right)\left(\mathrm{var}X_1 + \mathrm{var}X_2\right). \end{aligned}$$

Thus

$$\mathrm{var}X_1 + \mathrm{var}X_2 \leq \frac{\mathrm{var}X}{1+\rho_{12}}$$

when $\rho_{12} < 0$.

□

The relationship between the correlation coefficient and the average load variance of two time series can be better illustrated by Figure 5.1. For a given positive time series X, we randomly generate positive time series pairs X_1 and X_2 with $X_1 + X_2 = X$. For each pair of time series, we plot their correlation coefficient vs. average variance in Figure 5.1. We also show $\mathrm{var}X$ and $\mathrm{var}X/(1+\rho_{12})$ in the figure as the bounds for the average variance.

Because correlation coefficients are bounded between $[-1, 1]$, it is very easy to use them to check whether a given mapping plan is near optimal and to determine whether redistributing operators between a node pair is necessary. This observation is a very important foundation for one of our optimization techniques.

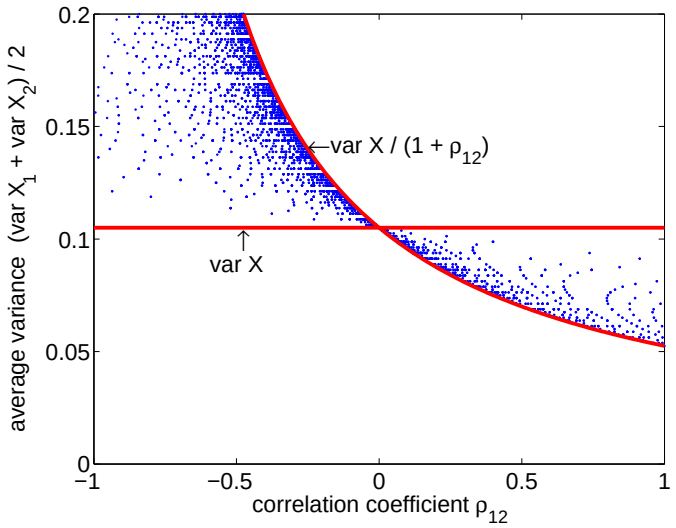


Figure 5.1: The relationship between the correlation coefficient and the average variance of two time series.

5.2 The Correlation-Based Algorithm

In this section, we present a greedy algorithm which not only balances the load of the system, but also tries to minimize the average load variance and maximize the average load correlation of the system. In addition, when moving operators between the nodes, we want to achieve a good balance between the amount of load moved between the nodes and the performance of the resulting load distribution plans.

Our algorithm can be divided into two parts. First, we use a global algorithm to make the initial operator distribution. Then, we switch to a dynamic load redistribution algorithm which moves operators between nodes in a pair-wise fashion. In the global algorithm, we only care about the quality of the resulting mapping plan without considering how many operators are moved. In the pair-wise algorithm, we try to find a good tradeoff between the amount of load moved and the quality of the resulting mapping plan.

Both algorithms are based on the basic load-balancing scheme. Thus, if the load of the system does not fluctuate, our algorithm reduces to a load balancing algorithm with a random operator selection policy. When the load of the system fluctuates, we can get load-balanced operator-distribution plans with smaller average load variance and larger average load correlation than the traditional load balancing algorithms.

Since it is easier to understand how to minimize the average load variance between a node pair than among all nodes in the system, we will first discuss the pair-wise algorithm, and then the global algorithm.

In all algorithms, we first obtain the operator distribution plan among all nodes without moving the physical operators. When we say move an operator from one node to another, we only move

the operators in the mapping plan. The physical operator are moved only after all changes to the operator distribution plans are finished.

5.2.1 Pair-wise Algorithms

For simplicity, we assume that there is a centralized coordinator in the system and the load information of all nodes is reported periodically to the coordinator. After each statistics collection period, the coordinator orders all nodes by their average load. Then the i^{th} node in the ordered list is paired with the $(n-i+1)^{th}$ node in the list. In other words, the node with the largest load is paired with the node with the smallest load; the node with the second largest load is paired with the node with the second smallest load, and so on. If the load difference between a node pair is greater than a predefined threshold ε, operators will be moved between the nodes to balance their average load.

Now, given a selected node pair, we will focus on how to move operators to minimize their average load variance. As we know that there is a tradeoff between the amount of load moved and the quality of the resulting mapping plan, we will first discuss an algorithm that moves the minimum amount of load, and then discuss an algorithm that achieves the best operator mapping quality, and finally, present an algorithm that balances the two goals well.

One-way Correlation-Based Load Balancing

In this algorithm, only the more loaded node is allowed to offload to the less loaded node. Therefore, the load movement overhead is minimized.

Let N_1 denote the more loaded node and N_2 denote the less loaded node. Let the load of N_1 be L_1 and the load of N_2 be L_2. Our greedy algorithm will select operators from N_1 one by one with total selected load less than

$$\frac{L_1 - L_2}{2}$$

until no more operators can be selected. The operator selection policy is inspired by the following observation:

Assume we have only two operators and two nodes. Let the load time series of the operators be Y_1 and Y_2 respectively and the load correlation coefficient of the two operators be ρ_{12}. Putting the operators on different nodes will result in an average load variance of

$$\left(\mathrm{var}Y_1 + \mathrm{var}Y_2\right)/2$$

and putting the operators on different nodes will result in an average load variance of

$$\mathrm{var}\left(Y_1 + Y_2\right)/2.$$

From the definition of correlation coefficient, we have that

$$\frac{\mathrm{var}(Y_1 + Y_2)}{2} - \frac{\mathrm{var}Y_1 + \mathrm{var}Y_2}{2} = \rho_{12}\sqrt{\mathrm{var}Y_1}\sqrt{\mathrm{var}Y_2}.$$

Initialization

$\Psi \leftarrow \phi$ (initialize the set of operators be moved as an empty set)
$\Omega \leftarrow \phi$ (initialize the set of candidate operators as an empty set)
$X_1 \leftarrow$ load time series of N_1
$X_2 \leftarrow$ load time series of N_2
$l \leftarrow \frac{\mathsf{E}X_1 - \mathsf{E}X_2}{2}$ (compute the total amount of load to be moved)
for each movable operator o_j on node N_1
  $Y_j \leftarrow$ the load time series of o_j
  if $\mathsf{E}Y_j < l$
    add o_j to Ω
    $S(o_j, N_1) \leftarrow \mathrm{cor}(Y_j, X_1 - Y_j)$
    $S(o_j, N_2) \leftarrow \mathrm{cor}(Y_j, X_2)$
    $S(o_j) \leftarrow \frac{S(o_j,N_1) - S(o_j,N_2)}{2}$

Operator Selection

while Ω is not empty
  let o_s be the operator in Ω with the largest score
  add o_s to Ψ
  delete o_s from Ω
  $X_1 \leftarrow X_1 - Y_s$
  $X_2 \leftarrow X_2 + Y_s$
  $l \leftarrow l - \mathsf{E}Y_s$
  for each operator o_j in Ω
    (update candidate operators and compute their score)
    if $\mathsf{E}Y_j < l$
      $S(o_j, N_1) \leftarrow \mathrm{cor}(Y_j, X_1 - Y_j)$
      $S(o_j, N_2) \leftarrow \mathrm{cor}(Y_j, X_2)$
      $S(o_j) \leftarrow \frac{S(o_j,N_1) - S(o_j,N_2)}{2}$
    else
      delete o_j from Ω

Figure 5.2: The pseudocode of the one-way correlation-based load balancing algorithm.

Obviously, to minimize average load variance, when $\rho_{12} < 0$, it is better to put the operators together on the same node, and when $\rho_{12} > 0$, it is better to separate them onto different nodes.

Now consider moving operators from N_1 to N_2 following this intuition. Let X_i denote the load time series of node N_i, and Y_j denote the load time series of operators o_j. Let $\rho(o_j, N_i)$ denote the correlation coefficient between the load time series of operator o_j and the total (sum of) load time series of all operators on N_i except o_j; i.e.

$$\rho(o_j, N_i) = \begin{cases} \mathrm{cor}(Y_j, X_i - Y_j) & \text{if } o_j \text{ is on } N_i \\ \mathrm{cor}(Y_j, X_i) & \text{if } o_j \text{ is not on } N_i \end{cases}$$

Then from N_1's point of view, it is good to move out an operator that has a large $\rho(o_j, N_1)$, and from N_2's point of view, it is good to move in an operator that has a small $\rho(o_j, N_2)$. Considering

both nodes together, we prefer to move operators with large

$$\rho(o_j, N_1) - \rho(o_j, N_2).$$

Define

$$S(o_j) = \frac{\rho(o_j, N_1) - \rho(o_j, N_2)}{2}$$

as the score of operator o_j with respect to N_2. Our greedy operator selection policy then selects operators from N_1 one by one with the largest score first.

As the score function in this algorithm is based on the correlation coefficients, and the operators can only be moved from one node to the other, this algorithm is called the *one-way correlation-based load balancing algorithm*. The pseudocode of this algorithm is shown in Figure 5.2.

Two-way Correlation-Based Redistribution

In this algorithm, we redistribute all operators on a given node pair without considering the former locations of the operators. With this freedom, it is possible to achieve better operator mapping quality than the previous algorithm.

The operator selection policy in this algorithm is also a score based greedy algorithm. We first start from two "empty" nodes (nodes with non-movable operators only), and then assign movable operators to these nodes one by one. In order to balance the load of the two nodes, for each assignment, we select the less loaded node as the receiver node. Then from all operators that have not been assigned yet, we compute their score with respect to the receiver node and assign the operator with the largest score to that node. This process is repeated until all operators are assigned. Finally, we use the above one-way algorithm to further balance the load of the two nodes.

The score function used here is the same as the score function used in the one way algorithm. It can also be generalized into the following form:

$$S(o_j, N_i) = \frac{\rho(o_j, N_1) + \rho(o_j, N_2)}{2} - \rho(o_j, N_i),$$

where $S(o_j, N_i)$ is called the score of operator o_j with respect to node N_i, $i = 1, 2$. The intuition behind the use of $S(o_j, N_i)$ is that the larger the score, the better it is to put o_j on N_i instead of on the other node.

As this algorithm will move operators in both directions, it is called the *two-way correlation-based operator redistribution algorithm*. The pseudocode of this algorithm is shown in Figure 5.3.

The final mapping achieved by this algorithm can be much better than the one-way algorithm. However, as it does not consider the former locations of the operators, this algorithm tends to move more load than necessary, especially when the former mapping is relatively good. In the following section, we present an algorithm that can get a good operator mapping plan by only moving a small fraction of operators from the existing mapping plan.

Initialization

$\Omega \leftarrow$ {all operators on N_1 and N_2 that can be moved }
$\Psi_1 \leftarrow \phi$ (initialize the set of operators to be assigned to N_1 as an empty set)
$\Psi_2 \leftarrow \phi$ (initialize the set of operators to be assigned to N_2 as an empty set)
$X_1 \leftarrow$ total load time series of all non-movable operators on N_1
$X_2 \leftarrow$ total load time series of all non-movable operators on N_2
for each operator $o_j \in \Omega$
    $Y_j \leftarrow$ the load time series of o_j
    $\rho(o_j, N_1) \leftarrow \text{cor}(Y_j, X_1)$
    $\rho(o_j, N_2) \leftarrow \text{cor}(Y_j, X_2)$

Operator Distribution

while Ω is not empty
    (determine the less loaded node)
    if $\mathsf{E}X_1 < \mathsf{E}X_2$
        $i \leftarrow 1$
    else
        $i \leftarrow 2$
    (compute operator scores with respect to the less loaded node)
    for each operator $o_j \in \Omega$
        $S(o_j, N_i) \leftarrow \frac{\rho(o_j, N_1) + \rho(o_j, N_2)}{2} - \rho(o_j, N_i)$
    let o_s be the operator in Ω with the largest score
    (assign selected operator to the less loaded node)
    add o_s to Ψ_i
    delete o_s from Ω
    $X_i \leftarrow X_i + Y_s$
    (update load correlation between N_i and all candidate operators)
    for each operator o_j in Ω
        $\rho(o_j, N_i) \leftarrow \text{cor}(Y_j, X_i)$

Further Load Balancing

balance the load of N_1 and N_2 using the one-way correlation-based load balancing algorithm

Figure 5.3: The pseudocode of the two-way correlation-based operator redistribution algorithm.

Two-way Correlation-Based Selective Exchange

In this algorithm, we allow both nodes to send load to each other. However, only the operators whose score is greater than a certain threshold δ can be moved. The score function used is the same as the one in the one-way algorithm. Recall that if the score of an operator on node N_i , $i = 1, 2$, is greater than zero, then it is better to put that operator on N_k ($k \neq i$) instead of on N_i. Thus, by choosing $\delta > 0$, we only move operators that are good candidates. By varying the threshold δ, we can control the tradeoff between the amount of load moved and the quality of the resulting mapping plan. If δ is large, then only a small amount of load will be moved. If δ is small (still greater than zero), then more load will be moved, but better mapping quality can be achieved.

One-way Load Balancing
balance the load of N_1 and N_2 using the one-way correlation-based load balancing algorithm

Operator Exchange Initialization
$\Psi_1 \leftarrow$ {all movable operators assigned to N_1 after one-way load balancing}
$\Psi_2 \leftarrow$ {all movable operators assigned to N_2 after one-way load balancing}
$X_1 \leftarrow$ total load time series of all operators assigned to N_1
$X_2 \leftarrow$ total load time series of all operators assigned to N_2
$count \leftarrow$ total number of operators on in Ψ_1 and Ψ_2

Operator Exchange
while $count > 0$
  if $\mathsf{E}X_1 > \mathsf{E}X_2$
    $s \leftarrow 1$, $r \leftarrow 2$
  else
    $s \leftarrow 2$, $r \leftarrow 1$
  for each operator $o_j \in \Psi_s$
    $Y_j \leftarrow$ load time series of o_j
    $\rho(o_j, N_1) \leftarrow \text{cor}(Y_j, X_1)$, $\rho(o_j, N_2) \leftarrow \text{cor}(Y_j, X_2)$
    $S(o_j, N_r) \leftarrow \frac{\rho(o_j, N_1) + \rho(o_j, N_2)}{2} - \rho(o_j, N_r)$
  let o_s be the operator in Ψ_s with the largest score
  if $S(o_s, N_r) > \delta$
    move o_s from Ψ_s to Ψ_r
    $X_s \leftarrow X_s - Y_s$, $X_r \leftarrow X_r - Y_s$
  else
    break
  $count \leftarrow count - 1$

Further Load Balancing
balance the load of N_1 and N_2 using the one-way correlation-based load balancing algorithm

Figure 5.4: The pseudocode of the two-way correlation-based selective operator exchange algorithm.

The details of the algorithm are as follows: (1) Balance the load of the two nodes using the above one-way algorithm. (2) From the more loaded node[1], check whether there is an operator whose score is greater than δ. If so, move this operator to the less loaded node. (3) Repeat step (2) until no more operators can be moved or the number of iterations equals to the number of operators on the two nodes. (5) Balance the load of the nodes using the one-way algorithm.

As this algorithm only selects good operators to move, it is called *two-way correlation-based selective operator exchange algorithm*. The pseudocode of this algorithm is shown in Figure 5.4.

[1]The load of the nodes cannot be exactly the same.

Pair-wise Load Balancing
balancing the load of the nodes pair-wisely using one of the two-way algorithms

Correlation Improved Initialization
$\Psi_i \leftarrow$ {all movable operators assigned to N_i after load balancing}
$X_i \leftarrow$ load time series of N_i after load balancing

Correlation Improvement
for $i = 1, \cdots, n$
if $\mathsf{E}X_i + \sqrt{\text{var}X_i} > 1$
(select the node with the minimum load correlation with N_i)
$s = 0$ (s is the index of the node to be selected)
$t \leftarrow \theta$
for $j = 1, \cdots, n$
if $j \neq i$ and $\text{cor}(X_i, X_j) < t$
$t \leftarrow \text{cor}(X_i, X_j)$
$s \leftarrow j$
if $s \neq 0$
(copy the old mapping plan)
$\Psi_i^{'} \leftarrow \Psi_i$, $\Psi_j^{'} \leftarrow \Psi_j$
$X_i^{'} \leftarrow X_i$, $X_j^{'} \leftarrow X_j$
apply one of the two-way algorithms between N_i and N_s and
update Ψ_i, Ψ_s, X_i, X_s
if $\text{cor}(X_i, X_s) \leq t$
(use the old operator distribution plan)
$\Psi_i \leftarrow \Psi_i^{'}$, $\Psi_j \leftarrow \Psi_j^{'}$
$X_i \leftarrow X_i^{'}$, $X_j \leftarrow X_j^{'}$

Figure 5.5: The pseudocode of the improved two-way algorithms.

Improved Two-way Algorithms

In all the above algorithms, operator migration is only triggered by load balancing. In other words, if an existing operator mapping plan is balanced, then no operator can be moved even if the load variance of some nodes is very large. To solve this problem and also maximize the average load correlation of the system, we add a correlation improvement step after each load balancing step in the above two-way algorithms.

Recall that if the load correlation coefficient of a node pair is small, then it is possible to further minimize the average load variance of the node pair. Thus, in the correlation improvement step, we move operators within a node-pair if their load correlation coefficient is below a certain threshold θ. Because we want to avoid unnecessary load migrations, the correlation improvement step is only triggered when some node is likely to get temporarily overloaded. The details of this step are as follows:

We define the "divergent load level" of each node as its average load plus its load standard deviation (i.e., square root of load variance). For each node with a divergent load level more than one (it is likely to get temporarily overloaded), apply the following steps: (1) compute the load

correlation coefficients between this node and all other nodes. (2) Select the minimum correlation coefficient. If it is less than θ, then apply one of the two way algorithms on the corresponding node pair (without moving the operators). (3) Compute the new correlation coefficient. If it is greater than the old one, then move the operators.

Notice that this is only for the two-way algorithms since no operators can be moved in the one-way algorithm when load is balanced. The resulting algorithms are called *improved two-way algorithms*. The pseudocode of this algorithm is shown in Figure 5.5.

5.2.2 Global Algorithm

In this section we discuss a global algorithm which distributes all operators on n nodes without considering the former location of the operators. This algorithm is used to achieve a good initial operator distribution when the system starts. Because we need load statistics to make operator distribution decisions, the algorithm should be applied after a statistics collection warm up period.

The algorithm consists of two major steps. In the first step, we distribute all operators using a greedy algorithm which tries to minimize the average load variance as well as balance the load of the nodes. In the second step, we try to maximize the average load correlation of the system.

The greedy algorithm is similar to the one used in the two-way operator redistribution algorithm. This time, we start with n "empty" nodes (i.e., nodes with non-movable operators only). The movable operators are assigned to the nodes one by one. Each time, the node with the lowest load is selected as the receiver node and the operator with the largest score with respect to this node is assigned to it. Finally, the load of the nodes is further balanced using one round of the pair-wise one-way correlation-based load balancing algorithm.

The major difference between the global algorithm and the former pair-wise algorithm is that the score function used here is generalized to consider n nodes together. The score function of operator o_j with respect to Node $N_i, i = 1, \cdots, n$, is defined as follows:

$$S(o_j, N_i) = \frac{1}{n} \sum_{k=1}^{n} \rho(o_j, N_k) - \rho(o_j, N_i),$$

The intuition behind $S(o_j, N_i)$ is that the larger the score, the better it is, on average, to put operator o_j on node N_i instead of putting it elsewhere. It is easy to verify that the score functions used in the pair-wise algorithms are just special cases of this form.

After all operators are distributed, a pair-wise correlation improvement step is then used to maximize the average load correlation of the system. First, we check whether the average load correlation of all node pairs is no less than a given threshold θ. If not, the node pairs with load correlation less than θ are selected as the candidate node pairs for operators redistribution. From this list, we first select the node pair with the minimum load correlation and use the two-way operator redistribution algorithm to obtain a new mapping plan among them. The new mapping plan is accepted only if the resulting correlation coefficient is greater than the old one. If operators are moved between the selected node pair, we must update their load correlations with the rest of

Initialization
$\Omega \leftarrow$ {all movable operators in the system}
for $i = 1, \cdots, n$
$\Psi_i \leftarrow \phi$ (initialize the set of operators to be assigned to N_i as an empty set)
$X_i \leftarrow$ total load time series of all non-movable operators on N_i
for each $o_j \in \Omega$
$Y_j \leftarrow$ the time series of o_j
for $i = 1, \cdots, n$
$\rho(o_j, N_i) \leftarrow \text{cor}(Y_j, X_i)$

Operator Distribution
order all nodes by average load ($\mathsf{E}X_i$)
while Ω is not empty
let N_r be the least loaded node
(compute operator scores with respect to the lest loaded node)
for each operator $o_j \in \Omega$
$S(o_j, N_r) \leftarrow \frac{1}{n}\sum_{i=1}^{n} \rho(o_j, N_i) - \rho(o_j, N_r)$
let o_s be the operator in Ω with the largest score
(assign selected operator to the lest loaded node)
add o_s to Ψ_r
delete o_s from Ω
$X_r \leftarrow X_r + Y_s$
(update load correlation between N_i and all candidate operators)
for each operator o_j in Ω
$\rho(o_j, N_r) \leftarrow \text{cor}(Y_j, X_r)$

Further Load Balancing
balancing the load of nodes pair-wisely using the one-way correlation-based load balancing algorithm

Correlation Improvement
(see pseudocode for pair-wise correlation improvement in Figure 5.7)

Figure 5.6: The pseudocode of the global correlation-based operator distribution algorithm.

the nodes and update the candidate node pair list accordingly. This process is repeated until no more candidate node pairs are available or the number of iterations reaches the number of node pairs in the system.

The pseudocode of the *global correlation-based operator distribution algorithm* is shown in Figures 5.6 and 5.7.

5.3 Extension

So far, we have assumed that all nodes are homogeneous. In this section, we generalize our algorithm to deal with nodes with different CPU capacities. The basic idea and approach of our algorithm remains the same. The only change here is related to the load measurement of the nodes.

Instead of balancing the absolute load (number of CPU cycles needed in a given time interval

```
Initialization
    Γ ← φ (initialize the set of candidate node pairs as an empty set)
    for i = 1, ··· , n
        for j = i + 1, ··· , n
            ρ_ij ← cor(X_i, X_j)
            if ρ_ij < θ
                insert ρ_ij to Γ and keep Γ in ascending order
    count ← n(n − 1)

Correlation Improvement
    while count > 0 and Γ is not empty
        let (N_i, N_j) be the node pair with the minimum load correlation in Γ
        delete ρ_ij from Γ
        (copy the old mapping plan)
        Ψ'_i ← Ψ_i, Ψ'_j ← Ψ_j
        X'_i ← X_i, X'_j ← X_j
        redistribute operators on N_i and N_j using the two-way correlation-based
        operator redistribution algorithm and update X_i and X_j
        if cor(X_i, X_j) > ρ_ij (ρ_ij is the old one)
            (apply the new plan and update the candidate node pairs)
            for k = 1, ··· , n
                if k ≠ i and k ≠ j
                    if old ρ_ik or ρ_ki or ρ_jk or ρ_kj ∈ Γ
                        delete them from Γ
                    ρ_ik ← cor(X_i, X_k), ρ_jk ← cor(X_j, X_k)
                    if ρ_ik < θ
                        add ρ_ik to Γ
        else
            (use the old plan)
            Ψ_i ← Ψ'_i, Ψ_j ← Ψ'_j
            X_i ← X'_i, X_j ← X'_j
        count ← count − 1
```

Figure 5.7: The pseudocode for the pair-wise correlation improvement step in the global operators distribution algorithm.

divided by the duration of the interval) of the nodes, we now balance the relative load of the nodes which is defined as the absolute load of the node divided by the CPU capacity of the node (number of CPU cycles the node has in the given time interval).

In the previous operator distribution algorithms, we always assign operators to the least load node or move operators from a more loaded node to a less loaded node. For heterogeneous nodes, we use relative load instead of absolute load for load comparison. Thus, by least loaded node, we mean the node with the minimum relative load; by more loaded node, we mean the node with larger relative load; and by less loaded node, we mean the node with smaller relative load.

When we balance the load of two homogeneous nodes with absolute loads L_1 and L_2, we move

operators with total absolute load

$$\frac{|L_1 - L_2|}{2}$$

from the more loaded node to the less loaded node.

Now suppose the two nodes N_1 and N_2 have CPU capacity C_1 and C_2 respectively, and assume $L_1/C_1 > L_2/C_2$. To balance the relative load of the nodes, we move absolute load ΔL from N_1 to N_2 such that

$$\frac{L_1 - \Delta L}{C_1} = \frac{L_2 + \Delta L}{C_2}.$$

Thus

$$\Delta L = \frac{L_1 C_2 - L_2 C_1}{C_1 + C_2}.$$

After making the above changes, the rest of the algorithms remain the same. We do not need to change the score functions because correlation coefficient is already a normalized term.

5.4 Complexity Analysis

In this section, we analyze the computation complexity of the above algorithms and compare it with a traditional load balancing algorithm. The basic load balancing scheme of the two algorithms are the same. The latter algorithm always selects operators with the largest average load first.

5.4.1 Statistics Collection Overhead

Assume each node has m/n operators on average and each load sample takes D bytes. Then the load statistics of each node takes $(m/n+1)hD$ bytes on average. Since the standard load balancing algorithm only uses the average load of each statistics window, the storage needed for statistics by the correlation based algorithm is h times that of the traditional load balancing algorithm.

On a high bandwidth network, the network delay for statistics transfer is usually negligible with regard to the load distribution time period. For example, we test the statistics transfer time on an Ethernet with 100M bps connection between the machines. Establishing the TCP connection takes 2ms on average. When $m/n = 20$, $h = 20$, the statistics transfer time is 1ms per node on average. Considering the TCP connection time together with the data transfer time, the difference between the correlation-based algorithm and the traditional load balancing algorithm is not significant.

5.4.2 Computation Complexity

First, consider the one-way correlation-based load balancing algorithm. In each load distribution period, it takes $O(n \log n)$ time to order the nodes and select the node pairs. For a given node pair, before selecting each operator, the scores of the candidate operators must be computed. Computing the correlation coefficient of a time series takes time $O(h)$. Thus, in a pair-wise algorithm, computing the score of an operator also takes time $O(h)$. Assume there are m_1 operators on the sender node, thus the total operator selection time is at most $O(hm_1^2)$. In the traditional load

balancing algorithm, it is not necessary to compute the scores of the operators, thus the operator selection time of the one-way correlation-based algorithm is $O(h)$ times that of the traditional load balancing algorithm. Similarly, for the two-way correlation-based load balancing algorithms, assume there are $m_1 + m_2$ operators on the two nodes, then it takes time $O(h(m_1 + m_2)^2)$ to redistribute the operators.

For the global algorithm, the score computation takes $O(nh)$ time for each operator. As there are m operators all together, its operator distribution time is $O(m^2nh)$. Thus the computation time of the greedy operator distribution step of the correlation-based global algorithm is $O(nh)$ times that of the traditional load balancing algorithm.

Finally, consider the computation complexity of the correlation improvement steps. In the pair-wise algorithms, computing the divergent load level of all nodes takes time $O(nh)$. If a node is temporarily overloaded, selecting a node pair takes time $O(nh)$, and to redistribute load between them takes time at the most $O(hm^2)$. There are at the most n temporarily overloaded nodes. Thus, the whole process takes time $O(n^2h + hm^2n)$

In the global algorithm, it takes time $O(n^2h)$ to compute the correlation matrix in the first iteration. In the following iterations, whenever operators are redistributed between a node pair, it takes $O(nh)$ time to update the correlation matrix. Selecting a node pair takes time $O(n^2)$. Redistributing operators on a node pair takes time $O(hm^2)$. Thus, each complete iteration takes time $O(nh + n^2 + hm^2)$. There are at most $n(n-1)$ iterations. Therefore, the total correlation improvement step takes time $O(n^3h + n^4 + m^2n^2h)$.

Although the correlation-based algorithms are in polynomial time, they can still be very expensive when m, n, h are large. Thus, we must work with reasonable m, n, h to make these algorithms feasible.

5.4.3 Parameter Selection

Obviously, the global algorithm and the centralized pair-wise algorithm cannot scale when n is large. However, we can partition the whole system into either overlapping or non-overlapping sub-domains. In each domain, both the global and the pair-wise algorithm can be applied locally.

In addition, as the pair-wise algorithm is repeated periodically, we must make sure that its computation time is small in comparison to the load distribution period. Obviously, when the number of operators on a node is large, a lot of operators must have very small average loads. As it is not necessary to consider each operator with a small load individually, the operators can be clustered into super-operators such that the load of each super-operator is no less than a certain threshold. By grouping operators, we can control the number of operators on each node.

Moreover, we can also choose h to achieve a tradeoff between the computation time and the performance of the algorithm. For a larger h, the correlation coefficients are more accurate, and thus the distribution plans are better. At the other extreme, when h is 1, our algorithm reduces to load balancing with a random operator selection policy.

Finally, we would like to point out that it is not hard to find reasonable m, h, and domain size

Table 5.1: Computation time with different number of nodes.

n	10	20	50
Computation Time	0.5sec	3.4sec	0.9min

n. For example, we tested the algorithms on a machine with an AMD Athlon^{TM} 3200+ 2GHz processor and 1GB memory. When $h = 10$ and the average number of operators on each node is 10, the computation time of the pair-wise operator redistribution algorithm is only 6ms for each node pair. If the load distribution interval is 1 second, the pair-wise algorithms only take a small fraction of the CPU time in each distribution period. Since the pair-wise algorithm can be easily extended to a decentralized and asynchronous implementation, it is potentially scalable. The computation time of the global algorithm with different n is shown in Table 5.1. Note that the global algorithm runs infrequently and on a separate node. It would only be used to correct global imbalances.

Chapter 6

Experimental Results

In this chapter, we evaluate the performance of our static operator distribution algorithm and dynamic operator distribution algorithms. We compare our algorithms with traditional operator distribution algorithms through various experiments with different parameters.

6.1 Static Operator Distribution Experimental Results

In this section, we study the performance of ROD by comparing it with several alternative schemes using the Borealis distributed stream-processing system [3] and a custom-built simulator. We use real network traffic data and an aggregation-heavy traffic monitoring workload, and report results on feasible set size as well as processing latencies.

6.1.1 Experimental Setup

Unless otherwise stated, we assume the system has 10 homogeneous nodes. We also assume that the operators can not be moved in this section. In addition to the aggregation-based traffic monitoring queries, we used random query graphs generated as a a collection of operator trees rooted at input operators. Because the maximum achievable feasible set size is determined by how well the weight of each input stream can be balanced, we let each operator tree consist of the same number of operators and vary this number in the experiments. For ease of experimentation, we also implemented a "universal" operator whose per-tuple processing cost and selectivity can be adjusted. To measure the operator costs and selectivities in the prototype implementation, we randomly distribute the operators and run the system for a sufficiently long time to gather stable statistics.

In Borealis, we compute the feasible set size by randomly generating 100 workload points, all within the ideal feasible set. We compute the ideal feasible set based on operator cost and selectivity statistics collected from trial runs. For each workload point, we run the system for a sufficiently long period and monitor the CPU utilization of all the nodes. The system is deemed

feasible if none of the nodes experience 100% utilization. The ratio of the number of feasible points to the total number of runs is the ratio of the achievable feasible set size to the ideal one.

In the simulator, the feasible set size of the load distribution plans are computed using Quasi Monte Carlo integration [60]. Due to the computational complexity of computing multiple integrals, most of our experiments are based on query graphs with five input streams (unless otherwise specified). However, the observable trends in experiments with different numbers of input streams suggest that our conclusions are general.

6.1.2 Algorithms Studied

We compare ROD with four alternative load distribution approaches. Three of these algorithms attempt to balance the load while the fourth produces a random placement while maintaining an equal number of operators on each node. Each of the three load balancing techniques tries to balance the load of the nodes according to the average input stream rates. The first one, called Connected-Load-Balancing, prefers to put connected operators on the same node to minimize data communication overhead. This algorithm is commonly used in traditional distributed systems. The second algorithm, called Largest-Load First (LLF) Load Balancing, orders the operators by their average load-level and assigns operators in descending order to the currently least loaded node. The third algorithm is our global Correlation-based Load Balancing algorithm. This algorithm is not designed for static operator distribution. However, because it tries to separate operators with large correlations to different nodes, operators from the same input stream tend to be separated to different nodes.

6.1.3 Experiment Results

Resiliency Results

First, we compare the feasible set size achieved by different operator distribution algorithms in Borealis. We repeat each algorithm except ROD 10 times. For the Random algorithm we use different random seeds for each run. For the load balancing algorithms, we use random input stream rates, and for the Correlation-based algorithm, we generate random stream-rate time series. ROD does not need to be repeated because it does not depend on the input stream rates and produces only one operator distribution plan. Figure 6.1 shows the average feasible set size achieved by each algorithm divided by the ideal feasible set size on query graphs with different number of operators.

It is obvious that the performance of ROD is significantly better than the average performance of all other algorithms. The Connected algorithm fares the worst because it tries to keep all connected operators on the same node. This is a bad choice because a spike in an input rate cannot be shared (i.e., collectively absorbed) among multiple processors. The Correlation-Based algorithm does fairly well compared to the other load balancing algorithms because it tends to do the opposite from the Connected algorithm. That is, operators that are downstream from a given input have high load correlation and thus tend to be separated onto different nodes. The Random

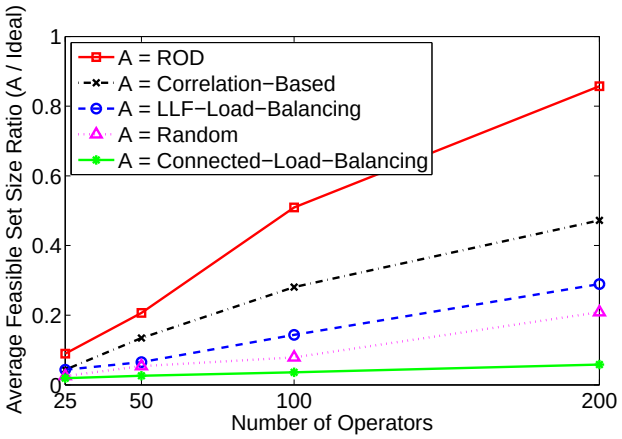


Figure 6.1: The ratio of the average feasible set size achieved by different algorithms to the ideal feasible set size (in Borealis).

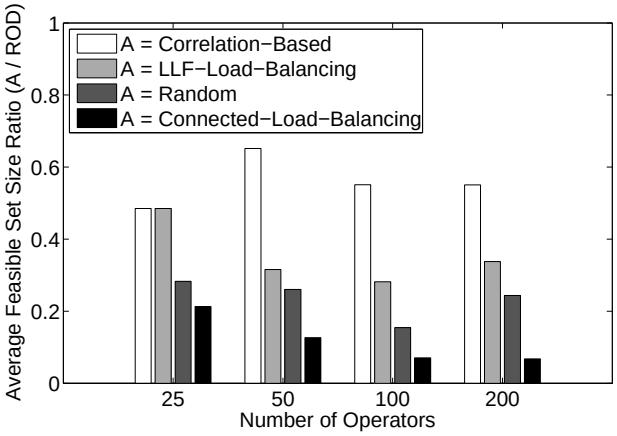


Figure 6.2: The ratio of the average feasible set size achieved by different algorithms to the feasible set size achieved by ROD (in Borealis).

algorithm and the LLF Load Balancing algorithm lie between the previous two algorithms because, although they do not explicitly try to separate operators from the same input stream, the inherent randomness in these algorithms tends to spread operators out to some extent. ROD is superior because it not only separates operators from each input stream, but also tries to avoid placing "heavy" operators from different input streams on the same node, thus avoiding bottlenecks.

As the number of operators increases, ROD approaches the ideal case and most of the other algorithms improve because there is a greater chance that the load of a given input stream will be spread across multiple nodes. On the other hand, even for fewer operators, our method retains roughly the same relative performance improvement (Figure 6.2).

Notice that the two hundred operators case is not unrealistic. It is important to realize that an operator graph does not correspond to a single query. In fact, it is typical for a given application

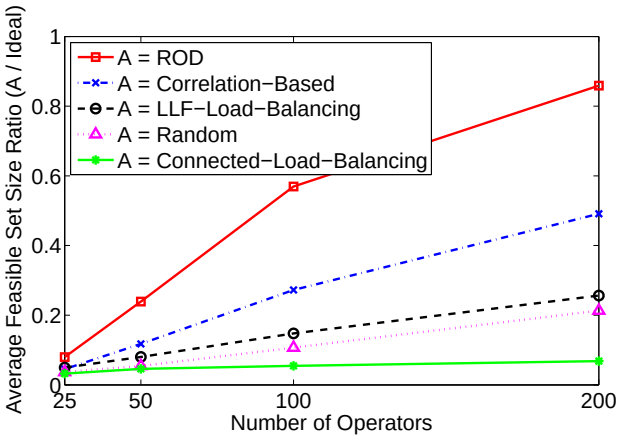


Figure 6.3: The ratio of the average feasible set size achieved by different algorithms to the ideal feasible set size (in the simulator).

to actually consist of many queries which, in total, account for large query graphs. In our experience with the financial services domain, applications often consist of related queries with common sub-expressions, so query graphs tend to get very wide (but not necessarily as deep). For example, a real-time proof-of-concept compliance application we built for 3 compliance rules required 25 operators. A full-blown compliance application might have hundreds of rules, thus requiring very large query graphs. Even in cases where the user-specified query graph is rather small, parallelization techniques (e.g., range-based data partitioning) significantly increase the number of operator instances, thus creating much wider, larger graphs.

We also ran the same experiments in our distributed stream-processing simulator. The ratio of the average feasible set size achieved by each algorithm to the ideal feasible set size is shown in Figure 6.3. Comparing this graph with the graph in Figure 6.1, we can see that the simulator results tracked the results in Borealis very closely, thus allowing us to trust the simulator for experiments in which the total running time in Borealis would be prohibitive.

We then run each algorithm except ROD 500 times in the simulator. Figure 6.4 shows the ratio of the feasible set size of these algorithms to the feasible set size of ROD in each experiment (each ratio correspond to a point in the graph). The ratios of each algorithm under a given query graph are shown in the same vertical line. The position of the markers in the graph shows the median ratio for each algorithm. The key thing this figure reveals is that the performance of different algorithms except ROD varies significantly given random input stream rates or random seed. Although these algorithms may result in feasible set size that are close to that of ROD sometimes, they are also likely to result in very small feasible set sizes, thus making the system very vulnerable. Having such uncertainty is even worse than what their average performance indicates, because time sensitive applications would be risk-averse.

In addition, we can see that the best-case performances of those algorithms are also worse than ROD in most cases. Thus, even if we can use those algorithms to generate a set of plans and choose

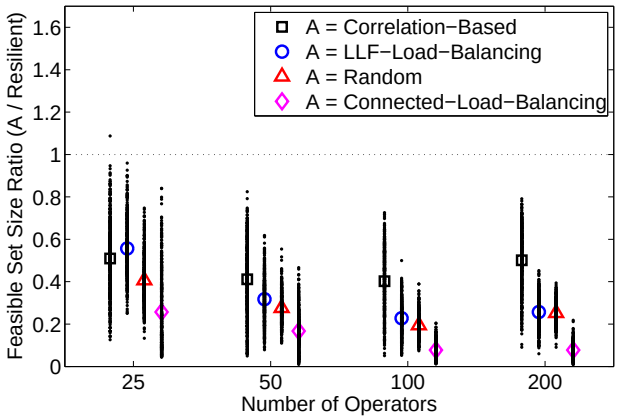


Figure 6.4: The ratio of the feasible set size achieved by different operator distribution algorithm to the feasible set size achieved by ROD (in the simulator).

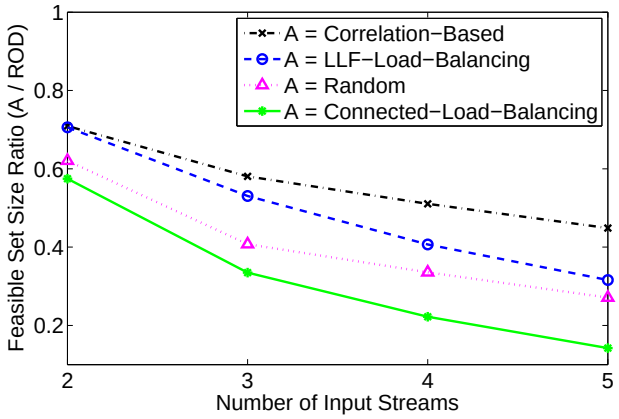


Figure 6.5: Impact of the number of input streams on feasible set size ratio.

the best one, they still can not beat ROD in most cases. Besides, for large numbers of input stream rates, comparing the feasible set size of different plans is itself computational infeasible.

Varying the Number of Inputs

Our previous results are based on a fixed number of input streams (i.e., dimensions). We now examine the relative performance of different algorithms for varying number of dimensions using the simulator.

Figure 6.5 shows the ratio of the feasible set size of the competing approaches to that of ROD, averaged over multiple independent runs. We observe that as additional inputs are used, the relative performance of ROD gets increasingly better. In fact, each additional dimension seems to bring to ROD a constant relative percentage improvement, as implied by the linearity of the tails of the curves. Notice that the case with two inputs exhibits a higher ratio than that estimated by the tail, as the relatively few operators per node in this case significantly limits the possible load

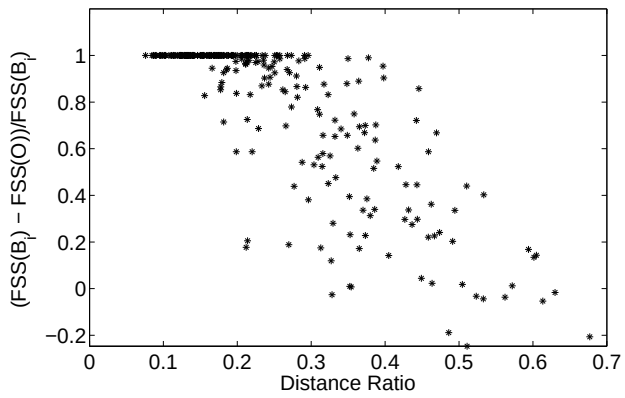


Figure 6.6: Penalty for not using the lower bound (using origin as the lower bound).

distribution choices. As a result, all approaches make more or less the same distribution decisions. For example, when the number of operators equals the number of nodes, all algorithms produce practically equivalent operator distribution plans by assigning one operator to each node.

Using a Known Lower-bound

As discussed in Section 4.4, having knowledge of a lower bound on one or more input rates can produce results that are closer to the ideal. We verify this analysis in this next set of experiments in the simulator.

We generate random points B_i in the ideal feasible space anchored at the origin to use as the lower bounds of each experiment. For each B_i, we generate two operator distribution plans, one that uses B_i as the lower bound and one that uses the origin. We then compute the feasible set size for these two plans relative to B_i. Let us call the feasible set size for the former plan FFS(B_i) and the feasible set size for the later FSS(O). We compute the penalty for not knowing the lower bound as $(\text{FFS}(B_i) - \text{FSS}(O))\,/\text{FSS}(B_i)$

We now run our experiment on a network of 50 operators. We plot the penalty in Figure 6.6 with the x-axis as the ratio of the distance from B_i to the ideal hyperplane to the distance from the origin to the ideal hyperplane. Notice that, when this ratio is small (B_i is very close to the ideal hyperplane), the penalty is large because without knowing the lower bound it is likely that we will sacrifice the small actual feasible set in order to satisfy points that will not occur. As B_i approaches the origin (i.e., the ratio gets bigger), the penalty drops off as expected.

The next experiment quantifies the impact of inaccurate knowledge of the lower bound values. In Figure 6.7, we run the same experiment as above except that, instead of using the origin as the assumed lower bound, we use another randomly generated point. As in the above experiment, we compute a penalty for being wrong. In this case, the penalty is computed as $(\text{FFS}(B_i) - \text{FSS}(Z_i))\,/\text{FSS}(B_i)$ where B_i is the real lower bound, as before, and Z_i is the assumed lower bound. The x axis is the distance between B_i and Z_i in the normalized space. As one might

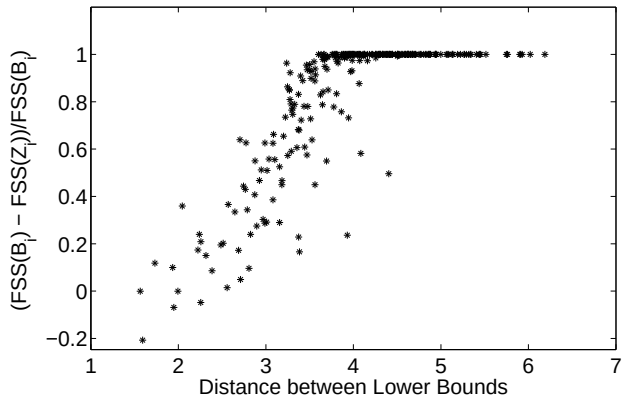


Figure 6.7: Penalty for using a wrong lower bound.

Table 6.1: Average penalty.

number of operators	25	50	100	200
origin as the lower bound	0.90	0.79	0.56	0.35
Z_i as the lower bound	0.89	0.83	0.55	0.32

expect, when the real and the assumed lower bounds are close to each other, the penalty is low. As the distance increases, the penalty also increases (Figure 6.7).

As one might expect, when the real and the assumed lower bounds are close to each other, the penalty is low. As the distance increases, the penalty also increases, reaching its maximum possible value at a normalized distance of about 4. Once the distance gets this big, the two points must be close to the ideal hyperplane because they would not be able to be that far from each other otherwise. With two such points, there would be little overlap in their feasible sets, so making them farther away has no further effect.

The penalty is also dependent on the number of operators in our query network. We redo our experiments for different networks with 25 to 200 operators leaving everything else unchanged. Looking at Table 6.1, we see that for these four different networks, the average penalty drops as we increase the number of operators. For very large numbers of operators, the penalty will converge to zero since, at that point, all the hyperplanes can be very close to the ideal case given the greater opportunity for load balancing.

Adding Data Communication Overhead

In this section, we address data communication overheads and study the impact of operator clustering in the simulator. For simplicity, we let each arc have the same per-tuple data communication cost and each operator have the same per tuple data processing cost. We vary the ratio of data

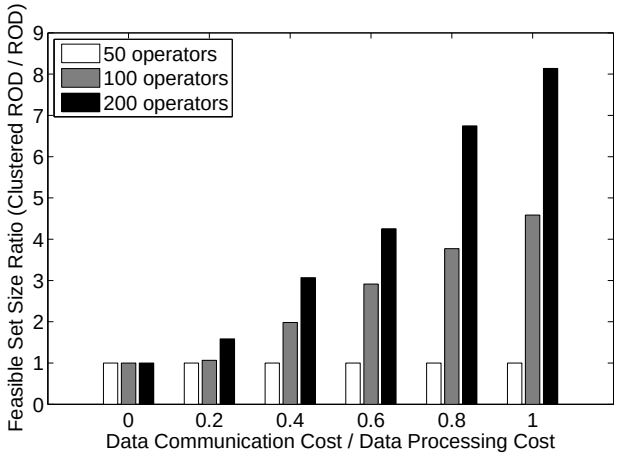


Figure 6.8: Feasible set size ratio of ROD with operator clustering vs. ROD without operator clustering after adding data communication overhead.

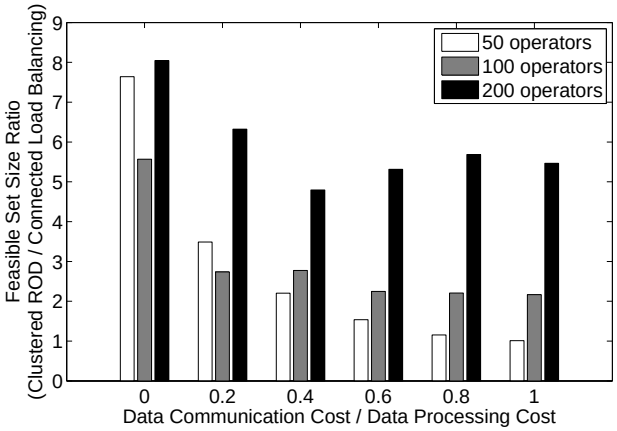


Figure 6.9: Feasible set size ratio of ROD with operator clustering vs. Connected-Load-Balancing after adding data communication overhead.

communication cost over data processing cost (from 0 to 1) and compare ROD with and without operator clustering. The results shown in Figure 6.8 are consistent with the intuition that operator clustering becomes more important when relative communication cost increases.

We also compare the performance of clustered ROD with Connected Load Balancing in Figure 6.1.3. Our first observation is that clustered ROD consistently performs better than Connected Load Balancing regardless of the data communication overhead. Secondly, we observe that clustered ROD can do increasingly better with increasing number of operators per input stream—more operators means more clustering alternatives and more flexibility in balancing the weight of each input stream across machines.

Adding Bandwidth Constraint

In this section, we add bandwidth constraints in the simulator. We assume that the bandwidth between each node the Ethernet network is 100M bps and the total bandwidth capacity of the Ethernet network is 1G bps. For simplicity, we let each stream have the same tuple size and let all operators have the same processing cost. The selectivities of all operators are one. In this configuration, the bandwidth ratios between all streams and their connected operators are the same. We then vary the bandwidth ratio in different experiments.

Figure 6.10 illustrates the ratio of the feasible set size achieved by ROD with operator clustering to the feasible set size of ROD without operator clustering in these experiments. For each experiment, we show, in the x-axis, the ratio of the bandwidth utilization to the CPU utilization of a node if a single operator (with a single input stream and a single output stream) is put on the node. When this ratio is low (e.g. 0.2, 0.8) , the bandwidth resources are not the bottleneck of the system. Thus, the performance of ROD with operator clustering is the same as that of ROD without operator clustering. When this ratio is one, if all operators in the system have a single input stream and a single output stream and no connected operators are put on the same node, then the bandwidth resource of a node and its CPU resource are always overloaded at the same time. This is the edge case between the abundant bandwidth situation and the limited bandwidth situation. In our randomly generated query graphs, an operator may have multiple output streams. Thus, when the ratio in the x-axis is one, the bandwidth resource of a node can become saturated before the CPU resource of the node is overloaded. Therefore, in this case, ROD with operator clustering can perform better than ROD without operator clustering. When the values in the x-axis are big (e.g. 2 and 10 in Figure 6.10), the bandwidth resource becomes the major bottleneck of the system. In these cases, ROD with operator clustering performs much better than ROD without operator clustering.

We also compared the performance of ROD with operator clustering vs. the Connected-Load-Balancing algorithm in Figure 6.11. We can see that whether the CPU resource is the bottleneck of the system or the bandwidth resource is the bottleneck of the system, ROD with operator clustering consistently performs better than the Connected-Load-Balancing algorithm.

Latency Results

While the abstract optimization of goal of this paper is to maximize the feasible set size (or minimize the probability of an overload situation), stream processing systems must, in general, produce low latency results. In this section, we evaluate the latency performance of ROD against the alternative approaches. The results are based on the Borealis prototype with five machines for aggregation-based network traffic monitoring queries on real network traces.

As input streams, we use an hour's worth of TCP packet traces (obtained from the Internet Traffic Archive [18]). For workload, we use 16 aggregation operators that compute the number of packets and the average packet size for each second and each minute (using non-overlapping time windows), and for the most recent 10 seconds and most recent one minute (using overlapping

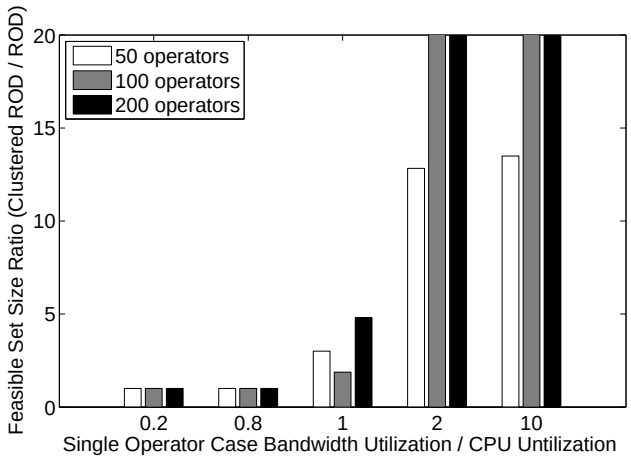


Figure 6.10: Feasible set size ratio of ROD with operator clustering vs. ROD without operator clustering after adding bandwidth constraints.

sliding windows), grouped by the source IP address or source-destination address pairs. Such multi-resolution aggregation queries are commonly used for various network monitoring tasks including denial of service (DoS) attack detection.

To give more flexibility to the load distributor and enable higher data parallelism, we partition the input traces into 10 sub-streams based on the IP addresses, with each sub-stream having roughly one tenth of the source IP addresses. We then apply the aggregation operators to each sub-stream and thus end up with 160 "physical" operators. Note that this approach does not yield perfectly uniform parallelism as the rates of the sub-streams are non-uniform and independent. It is therefore not possible to assign equal number of sub-streams along with their corresponding query graphs to different nodes and expect to have load balanced across the nodes (i.e. expect that the ideal feasible set can be achieved by pure data-partitioning based parallelism).

In addition to the algorithms described earlier, we introduce yet another alternative, Max-Rate-Load-Balancing, that operates similar to LLF-Load-Balancing, but differs from it in that the new algorithm balances the maximum load of the nodes using the maximum stream rate (as observed during the statistics collection period).

In order to test the algorithms with different stream rates, we scale the rate of the inputs by a constant. Figure 6.12 and 6.13 show the average end-to-end latency and the maximum end-to-end latency results for the algorithms when the input rate multiplier is 1, 2, 3, and 3.5, respectively. These multipliers correspond to 26%, 48%, 69% and 79% average CPU utilization for ROD. For this application, overall ROD performs better than all others not only because it produces the largest feasible set size (i.e., it is the least likely to be overloaded), but also because it tends to balance the load of the nodes under multiple input rate combinations. When we further increase the input rate multiplier to 4, all approaches except ROD fail due to overload (i.e., the machines run out of memory as input tuples queue up and overflow the system memory). At this point, ROD operates with approximately 91% CPU utilization.

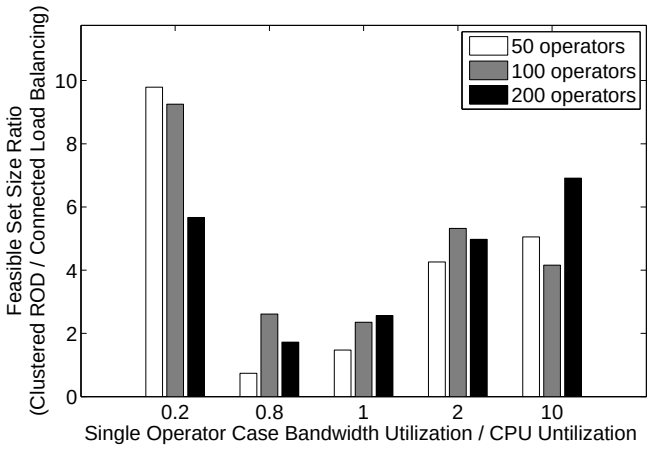


Figure 6.11: Feasible set size ratio of ROD with operator clustering vs. Connected-Load-Balancing after adding bandwidth constraints.

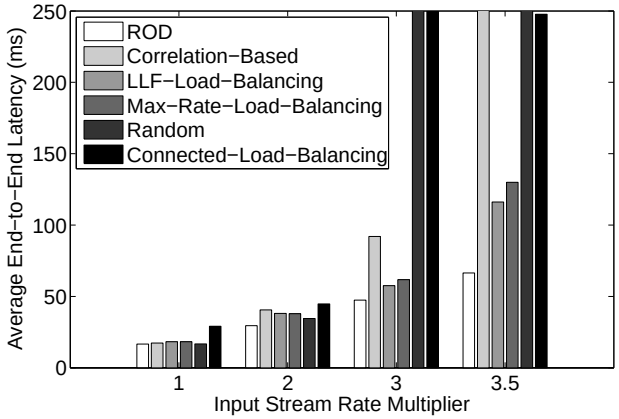


Figure 6.12: Average end-to-end latency achieved by different operator distribution algorithms.

The results demonstrate that, for a representative workload and data set, ROD (1) sustains longer and is more resilient than the alternatives, and (2) despite its high resiliency, it does not sacrifice latency performance.

6.2 Dynamic Operator Distribution Experimental Results

In this section, we study the performance of our dynamic operator distribution algorithms based on a simulator that we built using the CSIM library [56].

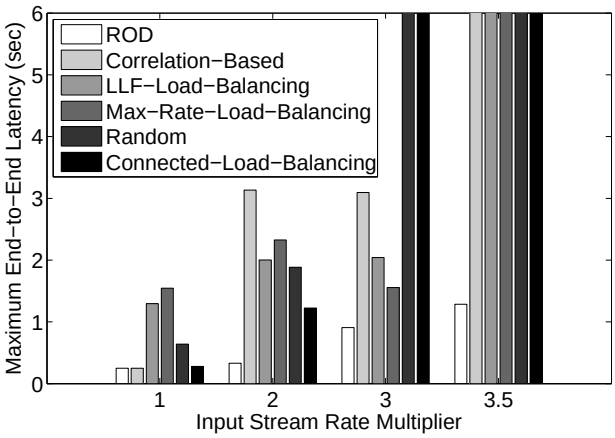


Figure 6.13: Maximum end-to-end latency achieved by different operator distribution algorithms.

6.2.1 Experimental Setup

Queries

For these experiments, we use several independent linear operator chains as our query graphs. The selectivity of each operator is randomly assigned based on a uniform distribution and, once set, never changes. The continuouse execution time of each operator is, at most, 0.1 second. We also treat all operators in the system as movable.

Workload

We used two kinds of workloads for our experiments. The first models a periodically fluctuating load for which the average input rate of each input stream alternates periodically between a high rate and a low rate. Within each period, the duration of the high rate interval equals the duration of the low rate interval. In each interval, the inter-arrival times follow an exponential distribution with a mean set to the current average data rate. We artificially varied the load correlation coefficients between the operators from –1 to 1 by aligning data rate modes of each input stream with a different offset.

The second workload is based on the classical On-Off model that has been widely used to model network traffic [5, 91]. We further simplified this model as follows: each input stream alternates between an active period and an idle period. During the active period, data arrives periodically whereas no data arrives during the idle period. The durations of the active and idle periods are generated from an exponential distribution. This workload models an unpredictably bursty workload. In order to get different load correlations from -1 to 1, we first generate some input streams independently and then let the other input streams be either the opposite of one of these streams (when steam A is active, its opposite stream is idle and vise versa) or the same as one of these streams with the initial active period starting at a different time.

We use the periodically fluctuating workload to evaluate the global algorithm alone and to

compare the pair-wise algorithms with the global algorithm. The bursty workload is used to test both algorithms together, as the global load distribution easily becomes ineffective under such workload.

Experiments

Unless specified, the operators are randomly placed on all nodes when a simulation starts. All experiments have an initial warm up period, when the load statistics can be collected. In this period, a node only offloads to another node if it is overloaded. After the warm up period, different load distribution algorithms are applied and the end-to-end latencies at the output are recorded.

We test each algorithm at different system load levels. The *system load level* is defined as the ratio of the sum of the busy time of all nodes over the product of the number of nodes and the simulation duration. For each simulation, we first determine the system load level, then compute the average rate of each input streams (to achieve the given load level) as follows: (1) Randomly generate a rate from a uniform distribution. (2) Compute the system load level using the generated steam rates. (3) Multiply each stream rate by the ratio of the given system load level over the computed system load level.

To avoid bias in the results, we repeated each experiment five times with different random seeds, and we report the average result. In order to make the average end-to-end latency of different runs comparable, we make each operator chain contain the same number of operators each with the same processing delay. In this setting, the end-to-end processing delay of all output tuples is the same. (i.e., no dependency on the randomly generated query graph).

Because the average end-to-end latency depends on the number of operators in each chain as well as the processing delay of each operator, we use the ratio of the average end-to-end latency over the end-to-end processing delay as the normalized performance measurement. This ratio is called the *latency ratio*.

Unless otherwise specified, all the experiments are based on the simulation parameters summarized in Table 6.2.

6.2.2 Algorithms Studied

We compare our correlation-based algorithms with a traditional load balancing algorithm which always selects the operator with largest load first, and a randomized load balancing algorithm which randomly picks the operators. Each of the latter two algorithms has both a global version and a pair-wise version. Operators are only moved from the more loaded nodes to the less loaded nodes.

Table 6.2: Simulation Parameters.

Number of nodes (n)	20
number of operators (m)	200
Number of operators in each chain	10
Operator selectivity distribution	U (0.8, 1.2)
Operator processing delay (per tuple)	1ms
Input rate generating distribution	U(0.8, 1.2)
Input rate fluctuation period	10sec
Input rate fluctuation ratio (high rate/low rate)	4
Operator migration time	200ms
Network bandwidth	100Mbps
Statistics window Size	10sec
# of samples in statistics window (h)	10
Load distribution period	1sec
Load balancing threshold (ε)	0.1
Score threshold for operator exchange (δ)	0.2
Correlation improvement threshold (θ)	0.8

6.2.3 Experiment Results

The Global Algorithms

First, we compare the three global operator allocation algorithms. They are the *correlation-based algorithm* (COR-GLB), the *randomized load balancing algorithm* (RAND-GLB) and the *largest-load-first load balancing algorithm* (LLF-GLB).

- **Static Performance of the Global Algorithms**

In the first experiment, the global algorithms are applied after the warm up period and no operator is moved after that. The latency ratios of these algorithms at different system load levels are shown in Figure 6.14. Obviously, the correlation-based algorithm performs much better than the other two algorithms. Figure 6.15 depicts the average load standard deviation of all nodes in the system after the global algorithms are applied. The COR-GLB algorithm results in a load variance that is much smaller than the other two algorithms. This further confirms that small load variance leads to small end-to-end latency. We also show the lower bound of the average load standard deviation (marked by MINIMUM) in Figure 6.15. It is the standard deviation of the overall system load time series divided by n (according to Theorem 1). The results show that the

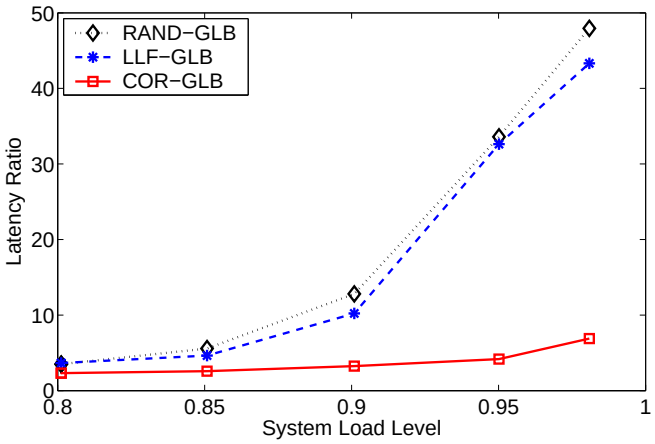


Figure 6.14: Latency ratio of the global algorithms.

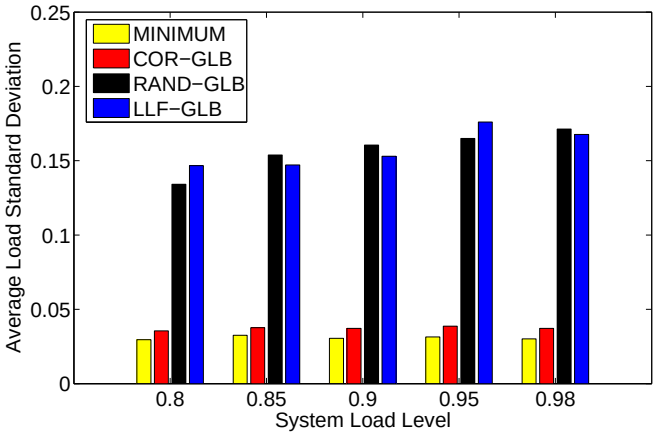


Figure 6.15: Average load standard deviation of the global algorithms.

average load variance of the COR-GLB algorithm is very close to optimal in this experiment.

In addition, we measured the average load correlation of all node pairs after the global distributions. The results of one algorithm at different load levels are similar to each other and the average results are shown in Table 6.3. Notice that the average load correlation of the RAND-GLB and the LLF-GLB algorithms are around zero, showing that their performance is not worst case. If an algorithm tends to put highly correlated operators (for instance, connected operators with fixed selectivity) together, it may result in an average load correlation close to -1. This would get much worse performance under a fluctuating workload.

The benefit of having large average load correlation is not obvious in the first experiment. The above results seem to indicate that when the system load level is lower than 0.5, it does not matter which algorithm is used. However, this is not true. In the second experiment, we show the effect of the different average load correlations achieved by these algorithms.

Table 6.3: Average load correlation of the global algorithms.

COR-GLB	RAND-GLB	LLF-GLB
0.65	-0.0048	-0.0008

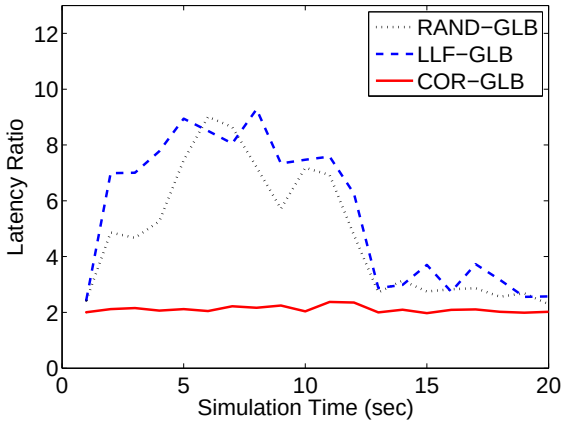


Figure 6.16: Latency ratio after load increase.

- **Dynamic Performance of the Global Algorithms**

In this experiment, we first set the system load level to be 0.5 and use different global algorithms to get initial operator distribution plans. Then, we increase the system load level to 0.8 and use the largest-load-first pair-wise load balancing algorithm to balance the load of the system. The latency ratios after the load increase is shown in Figure 6.16. The *amount of load moved* [1] after the load increase is shown in Figure 6.17. Because the COR-GLB algorithm results in large average load correlation, the load of the nodes is naturally balanced even when the system load level changes. On the other hand, the RAND-GLB and the LLF-GLB algorithms are not robust to load changes as they only have average load correlations around zero. Therefore, the correlation-based algorithm is still potentially better than the other two algorithms even if the current system load level is not high.

The Pair-wise Algorithms

For the pair-wise algorithms, we want to test how fast and how well they can adapt to load changes. Thus, in the following experiments, we let the system start from connected mapping plans where a connected query graph is placed on a single node. Different pair-wise algorithms are applied after the warm up period and the worse case recovery performance of these algorithms is compared.

[1] Whenever an operator is moved, its average load is added to the amount of load moved.

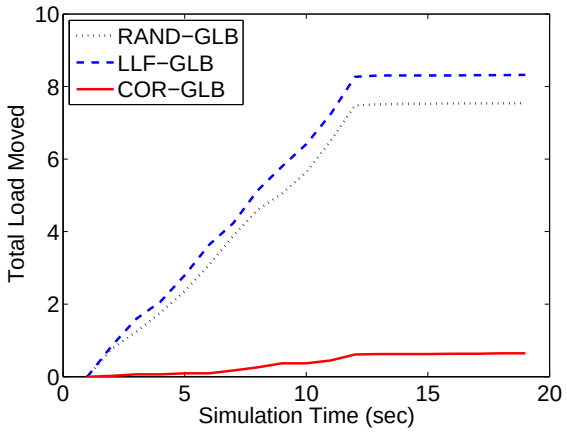


Figure 6.17: Amount of load moved after load increase.

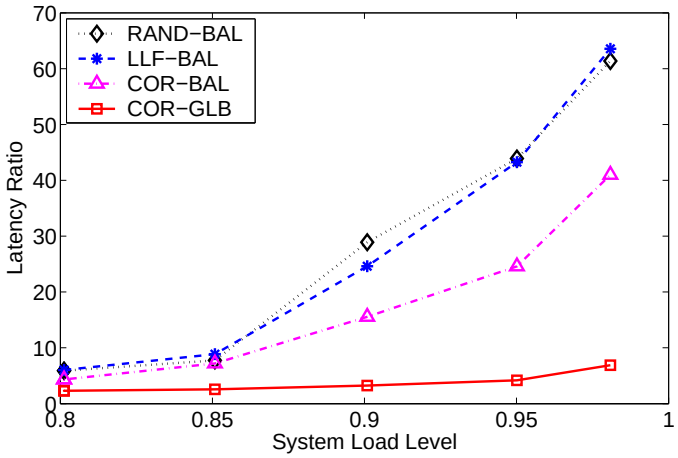


Figure 6.18: Latency ratio of the one-way pair-wise algorithms and the correlation-based global algorithm.

- **One-way Pair-wise Load Balancing Algorithms**

First the three one-way pair-wise algorithms are compared. They are the *correlation-based load balancing algorithm* (COR-BAL), the *randomized load balancing algorithm* (RAND-BAL) and the *largest-load-first load balancing algorithm* (LLF-BAL). Figure 6.18 depicts the latency ratios of these algorithms at different system load levels. Obviously, the COR-BAL algorithm has the best performance. Because the amount of load moved for these algorithms is almost the same, the result indicates that the operators selected by the correlation base algorithm are better than those selected by the other two algorithms. The latency ratios of the correlation-based global algorithm are added in Figure 6.18 for comparison. It shows that the performance of these pair-wise algorithms is much worse than that of the correlation-based global algorithm.

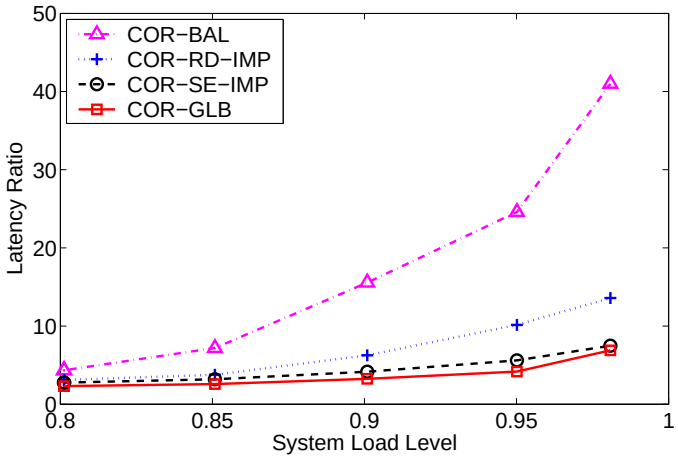


Figure 6.19: Latency ratio of correlation-based algorithms.

- **Improved two-way pair-wise algorithms**

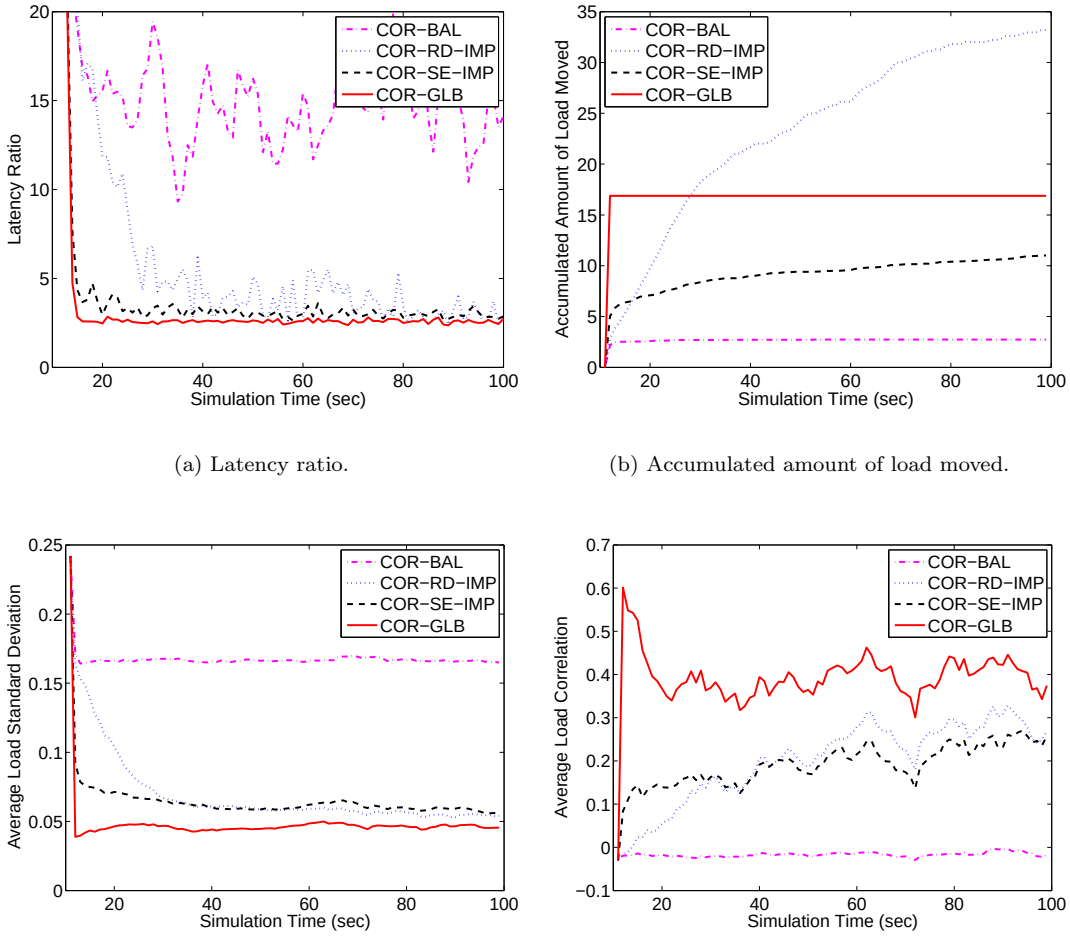
In this experiment, we compare two improved correlation-based two-way algorithms. They are the *improved operator redistribution algorithm* (COR-RE-IMP) and the *improved selective operator exchange algorithm* (COR-SE-IMP). The latency ratios of the COR-BAL and the COR-GLB algorithms are added in Figure 6.19 for comparison. The results show that the latency ratios of the improved two-way pair-wise algorithms are much smaller than the one-way algorithm. Thus, the benefit of getting better operator distribution plans exceeds the penalty of moving more operators.

To look at these algorithms more closely, we plot several metrics with respect to the simulation time when the system load level is 0.9 in Figure 6.20. Obviously, the COR-RE-IMP algorithm moves much more load than the COR-SE-IMP algorithm. Thus, although the quality of its final plan is closer to that of the global algorithm, its average performance is worse than that of the COR-SE-IMP algorithm. For different applications, which two-way algorithm performs better on average usually depends on the workload of the system and the operator migration time.

We can also see from Figure 6.20 that the global algorithm moves less load than the COR-RE-IMP algorithm but achieves better performance. Thus, although it is possible to use pair-wise algorithms only, it is still sensible to use a global algorithm for initial operator distribution.

Sensitivity Analysis

Here, we inspect whether the correlation-based algorithms are sensitive to different simulation parameters. In these experiments, the COR-GLB and the COR-SE-IMP algorithms are compared with the LLF-GLB and the LLF-BAL algorithms when the system load level is 0.9. We vary the number of nodes (n), the average number of operators on each node (m), the size of the statistics window, the number of samples in each statistics window (k), the input rate fluctuation period, and the input rate fluctuation ratio (high rate / low rate).



(a) Latency ratio.

(b) Accumulated amount of load moved.

(c) Average load standard deviation of the nodes.

(d) Average load correlation of the node pairs.

Figure 6.20: Dynamic performance of the correlation-based algorithms when system load level is 0.9.

The results in Figure 6.21 show that the correlation-based algorithms are not sensitive to these parameters except when m is very small, in which case, the load of the system cannot be well balanced. On the other hand, the largest-load-first load balancing algorithms are sensitive to these parameters. They perform badly, especially when the number of nodes is small, or the average number of operators on each node is small, or the load fluctuation period is long, or the load fluctuation ratio is large,

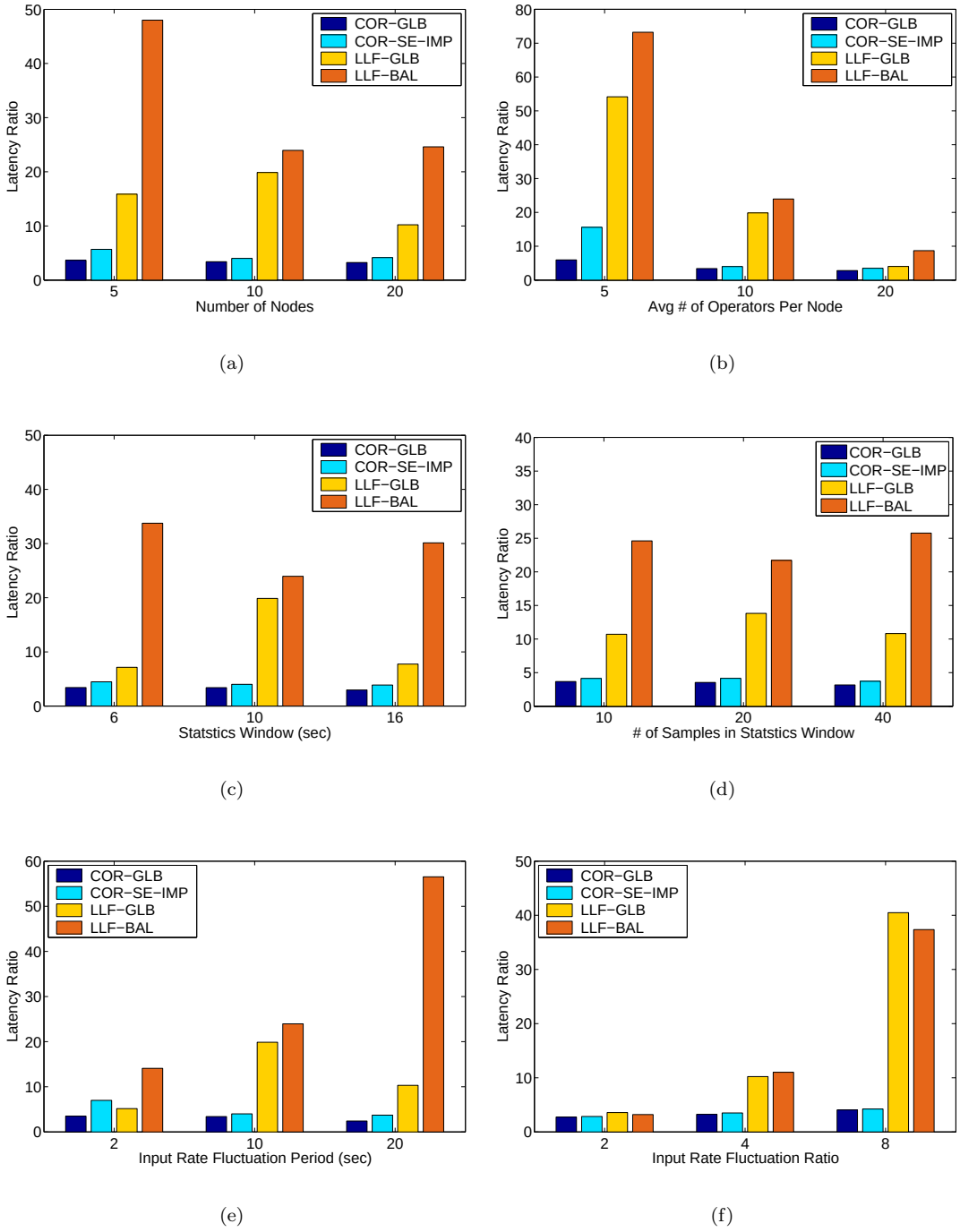
Notice that when m is large, the static performance of the largest-load-first algorithm is almost as good as the correlation-based algorithms. This is because when each operator has only a small amount of load and the load of all operators fluctuate differently, putting a lot of operators together can smooth out load variation. However, when the dynamic performance is considered, the correlation-based algorithm still performs better than the largest-load-first algorithm because

it results in a positive average load correlation and can naturally balance the load when the load changes.

In addition, these results show that the correlation-based algorithms are not very sensitive to the precision of the measured correlations. They work pretty well even when the size of the statistics window is only half of the load fluctuation period (i.e., when load fluctuation period is 20 in Figure 6.21(e)). Thus, when the precision of the load correlations must be sacrificed to reduce the computation overhead, we can still expect relatively good performance.

Bursty Workload

Finally, we use the bursty workload to test the robustness of our algorithms. The mean of the active period durations and the mean of the idle periods are both 5 seconds, and the statistics window size is still 10 seconds. As the duration of the active periods and the idle periods are exponentially distributed, the measured load correlations vary over time, and they are not precise. In this experiment, the global algorithms are combined with their corresponding pair-wise algorithms. The combined algorithms are identified by the names of the pair-wise algorithms with GLB inserted. The experimental results in Figure 6.22 confirm the effectiveness of the correlation-based algorithms under such workloads.



(a) (b)

(c) (d)

(e) (f)

Figure 6.21: Experiments with different parameters.

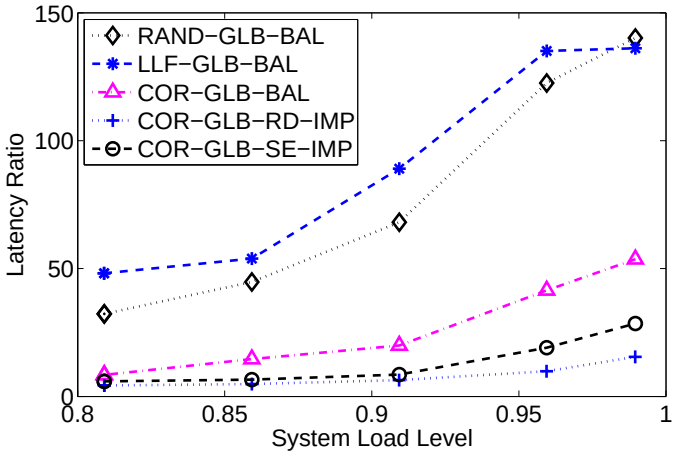


Figure 6.22: Bursty workloads.

Chapter 7

Related Work

In this chapter, we present the existing works that are related to our research. Our work is related both to the work on load distribution techniques in traditional distributed systems, and to the work on the load management techniques in distributed stream processing systems. In this chapter, we first give a brief overview of the existing work, and then discuss their relation to our work, as well as their difference from our work.

7.1 Load Distribution in Traditional Distributed Systems

Due to its critical role in high-performance computing, the load distribution issue has been studied extensively in traditional distributed systems [74, 15, 34]. Load distribution usually takes two forms: load balancing and load sharing [42]. Load balancing algorithms attempt to distribute the computation load across all nodes as evenly as possible. Load sharing is a weaker version of load balancing which only tries to avoid unshared states (some nodes are overloaded while some other nodes lies idle or underloaded) [54] .

7.1.1 Load Distribution Process

Generally, a load distribution process consists of three phases:

1. *Information Collection*, in which a load distributor gathers information about the load levels of the nodes.

2. *Decision making*, in which a load distributor determines a load distribution plan.

3. *Task routing or migration*, in which new tasks are routed to appropriate nodes or excess loads from heavily loaded nodes are transfered to underloaded ones.

7.1.2 Load Distribution Classification

Based on the different policies used in the three phases, load distribution schemes can be classified in the following dimensions:

1. **Centralized vs. Distributed Schemes**

   Load distribution schemes may be centralized, distributed or some hybrid form of both. In centralized schemes, there is only one load distributer in the system and it collects information from all nodes in the system. This load distributer is responsible for dispatching workload to the nodes or schedule load migration between the nodes.

   Distributed load balancing allows each node to collect load information and initiate the load migration process. It can be further divided into the following schemes:

   - *Sender initiated scheme*, in which heavily loaded nodes decide how much load to be moved to which underloaded node in the system.
   - *Receiver initiated scheme*, in which underloaded nodes request loads from heavily loaded nodes in the system.
   - *Sender and Receiver initiated scheme*, in which both heavily loaded nodes and underloaded nodes can initiate load migration.

   In a hybrid form of both centralized and distributed load distribution, the nodes in the system are partitioned into sub-domains or organized into a hierarchy. The load distribution within a sub-domain or a level is centralized. The load distribution between the sub-domains is distributed [46, 53].

2. **Global vs. Local schemes**

   In a global scheme, load distribution decisions are made based on the global knowledge of the system. In a local scheme, a load distributer only collects local information from its logical or physical neighbors. The decisions are thus based on partial knowledge of the system.

3. **Static vs. Dynamic Schemes**

   Static schemes assume that the total workload information is available a priori and a task is distributed only once before its execution [22]. In these schemes, tasks can be finished in a relatively short time, and, therefore, there is no need to consider time-varying load models and perform load migration. Static load distribution is advantageous in terms of low overheads, since the decision is made only once and there is no extra load migration overhead during task execution. Parallel computations, such as solving large-scale partial differential equations and finite element methods, often use static load balancing [37]. In these applications, the given computation can be divided into a large number of small computation tasks, and these tasks are distributed among the processors in a load-balanced manner. Much of the extensive literature on load distribution in fact belongs to the static schemes.

Dynamic load distribution receives attention for systems with long running tasks, such as large scientific computations and simulations. The static load distribution techniques are applicable for certain problems in which the computation can be divided into alternating balancing and processing phases. For most applications, however, it is desirable to have a process that monitors the system and redistributes the load continuously. For example, in many parallel mesh computations, the mesh regions are frequently refined or coarsened, which changes the distribution of load in the system continuously. Dynamic load distribution is required in a wide variety of applications, such as operating systems [61, 57, 62], parallel database systems [19, 69], and adaptive mesh partitioning [90, 27].

4. **System-Level vs. Application-Level Schemes**

   System-Level load distribution, also known as distributed scheduling, aims at maximizing the process throughput of the system or the overall utilization of the machines in the system. In this scheme, load distribution is often achieved by routing new jobs to underloaded machines [85] or transferring tasks from congested machines to other machines [36]. Application-level load distribution focuses on the mapping and re-mapping of parallel task graphs onto multiple processors. Its optimization goal is often to minimize the completion time of the parallel tasks.

7.1.3 Load Distribution Algorithms

Next, we briefly present some well known load distribution algorithms. Each of these algorithms may have many variations.

- **Balls-in-Bins**

  The most simple load balancing problem can be considered as a "balls-in-bins" problem, in which m balls have to be placed in n bins. The balls correspond to tasks with similar or unknown execution times, and the bins correspond to homogeneous machines. Commonly used distribution strategies include

  - *Round Robin* strategy, in which the fullest bin have $\lceil m/n \rceil$ balls.
  - *Random* strategy, which places each ball at a random bin and the fullest bin have $O(\frac{\log m}{\log \log m})$ balls with high probability [10].
  - *Semi-Random* strategy, in which k bins are asked for their load in each placement and a ball is placed into the lightest one. In this case, the fullest bin has $\frac{\log \log m}{\log k} + O(1)$ balls with hight probability [10].

- **Master/Slave**

  In this scheme, there is a single node in the system (called master node) that manages a work pool. All other nodes are called slave nodes and they request work from the master node if they are underloaded [36].

- **Centralized Load Balancing**

  Songnian Zhou [96] proposed a centralized algorithm in which each node sends their load information to a load distributor periodically. When the load distributor receives a request for a load transfer from a heavily loaded node, it select the least loaded node in the system and inform the originating node to send tasks there.

- **Random Location Load Sharing**

  Eager *et al.* [36] present a random load sharing scheme, in which a heavily loaded node forwards tasks to a randomly selected node in the system. To avoid the situation that tasks are transfered back and forth between nodes without being executed (which may happen if most nodes in the system are heavily loaded), an upper limit is set on on the number of times that a task can be transferred. This scheme has very small overhead because it does not require load information to be exchanged among the nodes. Yet, it yields substantial performance gain over the non-load sharing schemes.

- **Probing**

  This is a demand-driven policy, in which a node that wants to participate in a load transfer probes other nodes to find out whether it is beneficial to initiate a load transfer [74, 36, 96]. Both sender nodes and receiver nodes can probe other nodes. A node may probe either serially or in parallel. A node can select a probing node randomly, from a neighboring node, or based on the information collected in previous probes.

- **Bidding**

  There are two procedures in contract bidding algorithms: normal and reverse. In the normal case, a node that wants to distribute load first broadcasts a request for bids. The nodes that are willing to take the load then submit a bid based on the availability of their own resources. Later, the requesting node selects a best bid and sends some tasks to the bidding node [78]. In the reversed procedure, under loaded nodes send request-for-bid messages to broadcast their willingness to share load. The performances of the class of bidding-based algorithms depend on how well the bids are computed.

- **Diffusion Method**

  In diffusion methods [29, 79], any processor that invokes a load balancing operation compares its load with those of its nearest neighbors, and then offloads to, or takes in certain amount of load from, each neighbor. Depending on how to compute the amount of load to be moved between the neighboring nodes, diffusion methods can be divided into *local average diffusion*, which takes the average load of the neighboring nodes [45, 68], and *optimally tuned diffusion*, which tries to maximize efficiency in static and synchronous balancing [94, 31].

- **Dimension Exchange**

The Dimension Exchange strategy [31, 93] was conceptually designed for a hypercube system, but may be applied to other topologies with some modification. In the case of an n processor hypercube configuration, load balancing is performed iteratively in each of the $\log n$ dimensions.

- **Gradient Model**

  This model [52] classifies nodes into three classes: underloaded, normal, and highly loaded. It employs a gradient map of the "proximities" of each node to the underloaded nodes as a guide for the direction of load migration. When a node become highly loaded, it offloads to its nearest neighbor with the maximum proximity.

- **Hierarchical Load Balancing**

  In hierarchical load balancing [89], the nodes in the system are organized in a tree hierarchy where the root of the tree encompasses the entire system. Load balancing is initiated at the lowest levels and then ascends to higher levels. In each level, the load balancing is centralized.

- **Graph Partitioning**

  Graph partitioning problems involve partitioning a connected task graph into uniformly loaded subgraphs, while minimizing the total "weight" (often the data communication cost) of the cutting edges among the subgraphs. If changes in load lead to unbalanced subgraphs, the graph can be re-partitioned (often by moving boundary vertices of the subgraphs) to re-balance the overall system load.

  The graph partitioning problem is known to be NP-hard [38]. It is, therefore , not possible to compute optimal partition for graphs with large numbers of vertices in general. This fact has led to the development of several heuristic approaches. Static graph partitioning techniques are usually classified as geometric techniques [16, 39], combinatorial techniques [33, 43, 14], spectral techniques [66, 67, 44, 75], or multilevel methods [48, 49]. Dynamic graph repartitioning techniques include scratch-remap techniques [63], cut-and-paste methods [71], and diffusion-based schemes [35, 70]. A survey of these methods can be found in Schloegel *et al.* [71] and Walshaw *et al.* [27].

Our work is different from the work in traditional distributed systems both in terms of optimization goals and load distribution method. Our correlation-based algorithm is a centralized load balancing algorithm. However, it not only aims at balancing the average load of the nodes, but also at minimizing the average load variance and maximizing the average load correlation. To the best of our knowledge, this optimization goal has not been addressed by the previous load distribution works yet.

Our resilient operator distribution algorithm, on the other hand, does not try to balance the load of the system for any given input stream rates. Instead, it aims at achieving a load distribution plan that is resilient to load changes. We seek load distribution plans that are close to an ideal load distribution plan, in which the load of the system is balanced no matter what the input stream

rates are. In this sense, our algorithm can be considered as trying to balance the load of the system for multiple input stream rate points rather than one set of input stream rates. This is in sharp contrast with the traditional load balancing algorithms.

7.2 Stream Processing Engines

Our work is motivated by the new challenges brought by the emerging stream based applications. These applications are different from traditional parallel computation applications in that the input data are potentially unbounded and the input stream rates are determined by the external data sources. Many stream processing query languages and stream data management systems have been developed in recent years. We now provide an overview of the related projects on stream data processing.

Tapestry [83] is one of the first systems that proposed the notion of continuous queries. It adds *continuous* semantics to conventional database queries such that the queries are issued once and henceforth run continuously. It restricts to append-only relational databases where deletions and in place modifications are disallowed. Users issue queries for content-based filtering (e.g. filtering email and bulletin board messages). These queries are then rewritten into incremental queries, which are executed by the Tapestry query processing engine periodically. The contribution of this system is that it makes monitoring tasks no more difficult to write than traditional database queries, and only requires small changes from regular databases to process continuous queries. The limitation of this system is that it does not scale to high stream rates and the response time is relatively large due to its store-and-process execution nature.

Tribeca [80, 81] is a stream-oriented database management system which is designed to support network traffic analysis. It can read a stream of data from a single source and apply a data-follow based query language on the stream. It defines "Demultiplex" and "Multiplex" operators to separate and recombine sub-streams derived from the source. It also defines windows on data streams to perform aggregate and join operations. Although it is an application-specific engine, its query language is actually very general.

NiagaraCQ [25, 59] is an XML-based continuous query processing system which is designed to enable users to pose XML queries over the Internet. It supports a declarative high level query language which is transformed internally into a data-flow oriented query execution plan. In order to process a large number of queries registered by millions of users, NiagaraCQ explores the similarities between the query execution plans and share the computation of similar query plans.

TelegraphCQ [23, 50] is an adaptive stream processing engine which was developed at UC Berkeley. It grew out of the earlier Eddy project [9] on adaptive relational query processing and extends it to support shared processing over data streams [55, 24]. TelegraphCQ expresses continuous queries as SQL statements enhanced by windowing clauses. These queries are then translated into a set of commutable operators connected by an Eddy. The Eddy routes tuples between the operators based on changing conditions. The advantage of this system is that the

query execution plans are highly adaptive. However, the price of such high adaptivity is the high overhead in per tuple routing decision making.

STREAM [58, 41] is a continuous query processing engine on data streams developed at Stanford University. It both allows direct input of query plans (similar to the Aurora approach), and supports a declarative query language which is an extended version of SQL with sliding window specification and an optional sampling clause. In addition, STREAM introduces special operators that can transform streams into relations and relations to streams so that the continuous query in STREAM can be applied both on streams and relations [7].

The work presented in this dissertation is based on the Aurora/Borealis stream processing engine. Nevertheless, it is general enough to be applied to other distributed stream processing systems if they have data-flow based stream processing engines (such as Tribeca, NiagaraCQ, and STREAM) as processing nodes.

7.3 Distributed Stream Processing

The requirement for scalable and highly-available stream processing services led to the research on distributed stream processing [26]. Research in this area includes adaptive distributed continuous query processing [84, 97, 98], fault tolerance and high availability [47, 72, 12], load management [13, 73, 92] , and operator placement [77, 6, 65]. We now discuss in more detail the related work on load management and operator placement for distributed stream processing.

7.3.1 Load Management

Shah *et al.* [73] present a dynamic load distribution approach for a parallel continuous query processing system, called Flux. In Flux, a single continuous query operator is processed cooperatively by multiple shared-nothing servers. Each copy of this operator on a server is also called a consumer operator. The upstream operators of the consumer operators are also called the producer operators. A Flux middle-ware operator is interposed between the producer operators and consumer operators. The Flux operator partitions the output streams from the producer operators into sub-streams and feeds these sub-streams to the consumer operators. It also absorbs short term load imbalance by buffering and reordering tuples. When long term imbalance is detected (from buffer overflow), Flux re-partitions the data streams to re-balance the load of the consumer operators. This is done by balancing the load of the consumer operators (or servers) pair-wisely. The most loaded server is paired with least loaded server. The second most loaded server is paired with the second least loaded server, and so on and so forth. When certain data partitions are to be moved from one server to another server, Flux first moves the states of the data partition from the sending sever to the receiving server and then starts to send data of this partition to the receiving server. In summary, Flux performs dynamic "intra-operator" load balancing by repartitioning data streams on the fly. Our work is orthogonal to Flux, as we address the "inter-operator" load distribution problem.

Balazinska *et al.* [13] explore the dynamic load management problem for distributed stream processing in a federated environment. In this setting, each participant node owns and administers a set of resources that it uses to run tasks on behalf of its own clients. These autonomous participants do not collaborate for the benefit of the whole system, but aim to maximize their own benefit. Load management in this system is achieved by a *Bounded-Price Mechanism*. In this scheme, the nodes negotiate pairwise contracts, which set tightly bounded prices for migrating each unit of load between two participants and specify the set of tasks that each is willing to execute on behalf of the other. These contracts are negotiated off-line. Runtime load transfer occurs only between participants that have pre-negotiated contracts, and at a unit price within the contracted range. If a node finds that the local processing cost of a task is larger than the price that it would have to pay another participant for processing the same load (plus the migration cost), it will move the task to the other participant. The analysis and experiment of the work show that this mechanism provides enough incentives for selfish participants to handle each other's excess load, and thus improve the system's load distribution. Our scheme is different from this scheme in that all nodes in our system collaborate with each other for the benefit of the whole system. In addition, the Bounded-Price Mechanism only aims at converging to some feasible solution quickly by constructing good contracts, whereas, we try to optimize the performance of the system in terms of resiliency and end-to-end data processing latency.

7.3.2 Operator Placement

Srivastava *et al.* [77] address the problem of operator placement in a sensor network environment. In this system, data are collected from low-capability edge devices and then processed and transmitted through a hierarchy of network nodes and links with progressively increasing computation power and network bandwidth. In order to reduce network bandwidth consumption, this system pushes some or all of the query execution task to nodes lower in the hierarchy. They define the per tuple execution cost of an operator on a node as a function of the type of the operator and the processing power of the the node. An operator has a larger cost if it is placed on a node with less processing power than its cost if it is place on a node with more processing power. The network transmission cost of a tuple on a link is defined as a constant, which is assumed to include an appropriate multiplicative factor that can convert transmission cost into a quantity that can be treated at par with computation cost. The cost of a query execution plan is defined as execution cost of the query plus the network transmission costs on all links per input tuple. The optimization goal of the work is then to minimize the total cost of the query execution plan by proper operator placement. They provide an optimal solution for uncorrelated filters and an approximate solution for correlated filters. This work is different from our work in two major aspects: First, this work is based on a hierarchy of processing nodes with increasing processing power from leaf to root; while we consider a cluster of blade farm servers connected by a high bandwidth local network. Second, this work does not consider the constraint that each node can only process limited amount of work; in other words, they only consider scenarios that none of the nodes are overloaded. This

is in sharp contrast with our work, in which the feasibility and load balancing of the nodes are the major concerns.

Amand and Çetintemel [6] present a distributed query processing system for stream-based applications in a wide area environment. This system is built on top of a Distributed Hash Table that determines the locations of data sources. This work presents three algorithms to determine the location of the operators with the optimization goal of minimizing total mapping cost. The first algorithm models the cost of a link as a non zero constant if the end-operators of the link are placed at different nodes. The second algorithm extends the first algorithm by taking into account in the cost function the pair-wise transmission latencies between the nodes. Both algorithms only place operators either at the consumer side, or at the producer side. The third algorithm, called In-Network operator placement, considers a carefully selected subset of network locations, in addition to the locations of the procurers and consumers of the operators, when making placement decisions.

Pietzuch *et al.* [65] present another stream-based overlay network (called SBON) in a wide area environment. SBON is a layer between a stream processing system and the physical network that manages operator placement. In SBON, every node maintains its coordinates in a cost space. The distance between two nodes in this space represents the network overhead for routing data between them. The SBON determines the placement of query operators in this virtual space using a spring relaxation algorithm. This algorithm models the overlay network of queries as a collection of operators connected by springs (operator links). Some operators have a fixed location, and other operators can move freely. By using link latency as the spring extension factor, using link data rate as the spring constant, and minimizing the energy of the system, this algorithm minimizes the network usage of the system, where the network usage is defined as the total weighted latency of the used links used, with the weight of a link being the data rate of the link. After placing the operators in the virtual space, the locations are mapped back to physical nodes. When choosing physical nodes, the SBON selects nodes with sufficient resources to process the operators that are closest to the locations determined in the virtual space. The SBON nodes also re-evaluate operator placement plans dynamically and migrate operator to new hosts based on changing conditions. Our work is orthogonal to these works, in that we consider a cluster of servers connected by a high bandwidth local network, and the network transfer latencies in this environment is negligible.

Chapter 8

Conclusion

In this dissertation, we have addressed the load management problem for distributed stream processing. We have studied in-depth a class of algorithms that statically finds a good operator placement and that dynamically moves operators to adapt to load changes.

Unlike traditional databases where data can be pulled from secondary storage in a controlled and predictable manner, queries over streams must be prepared to deal with unpredictable spikes in input load. Our experience with the Borealis prototype reveals that moving operators can be quite expensive. We therefore sought a radically different solution for operator placement that is *resilient* in the sense that it produces load distribution plans that are highly tolerant to load variations, thereby resisting overload and, therefore, the need to reposition operators.

In addition, we also tried to adapt to load changes dynamically by strategically moving operators with relatively small load migration overheads. The traditional algorithm tends to reflect input spikes as spikes in end-to-end latency as a result of saturation of one or more of the component machines. Our approach, which is based on load correlation, effectively smoothes out the effects of the input load variations. Thus, load re-balancing is triggered less often and the associated short-term latency increase is avoided.

The main contribution and results of our techniques are summarized bellow:

8.1 Resilient Operator Distribution

We introduced the notion of a resilient operator placement plan that optimizes the size of the input space that will not overload the system. In this way, the system will be able to better withstand short term input bursts.

Our model is based on reducing the query processing graph to segments that are linear in the sense that the load functions can be expressed as a set of linear constraints. In this context, we identified an ideal load distribution plan, which balanced the load of each input stream perfectly across all nodes, and proved that such a plan achieves the maximum feasible set size. We presented two operator distribution heuristics that aim at making the achieved load distribution plan close

to the ideal load distribution plan as much as possible. Intuitively, the first heuristic balances the load of each input stream across all nodes, and the second tries to keep the load on each node evenly distributed. From a geometric point of view, the first heuristic tries to maximize the size of a polytope subset of the achieved feasible set, and the second heuristic tries to maximize the size of hypersphere subset of the achieved feasible set.

We presented a greedy operator distribution algorithm (ROD) that seamlessly combines both heuristics. The algorithm assigns operators to nodes one by one from the operator with the largest impact to the load distribution plan to the operator with the least impact. Each time an operator is assigned, the algorithm tries to make the reduction in feasible set size as small as possible.

We extended ROD to deal with known lower bounds on system input stream rates. We presented techniques that can transform a non-linear load model to a linear load model so that ROD can be applied to any systems. We also presented some operator clustering techniques to be applied before ROD to avoid unnecessary data communication overheads and network bandwidth consumption.

We have shown experimentally that there is much to be gained with this approach. It is possible to increase the size of the allowable input set over standard approaches. We also show that the average latency of our resilient distribution plans is reasonable. Thus, this technique is widely applicable to any modern distributed stream processor. Initial operator placement is useful whether or not dynamic operator movement is available. For systems that support dynamic operator migration, this technique can be used to place operators with large migration time to minimize the movement of those operators, and thus avoid big delays in end-to-end latency that results from operator migration.

8.2 Correlation-Based Operator Distribution

We have showed that for stream processing systems, average load is not the only important factor that determines the average end-to-end latency of the output tuples. The variance in CPU utilization also has a strong impact to the performance of the system too. Thus, in addition to the optimization goal of balancing the average load of the nodes, we proposed another novel optimization strategy that aims at minimizing the variance of the CPU utilization of the nodes. Furthermore, we try to maximize the average load correlations of all node pairs in the system to avoid unbalanced situation.

We proved that in the ideal case, where the total load of the system can be arbitrarily partitioned, the above optimization goals are achieved at the same time (when all nodes in the system have the same load time series). Such a load distribution is usually unachievable in practice. We then presented greedy operator distribution algorithms based on heuristics. We presented both a global operator distribution algorithm for initial operator placement and some pairwise algorithms for dynamic operator re-distribution. The key ideal underlying these algorithms is to separate operators with large load correlations to different nodes and put operators with small load correlations

on the same nodes.

Our experimental results have shown that our techniques are superier to conventional load management techniques. This illustrates how the streaming environment is fundamentally different from other parallel processing approaches. The nature of the operators and the way that data flows through the network can be exploited, as we have done, to provide a much better solution.

8.3 Combining the Two Approaches

The combination of the resilient operator distribution and the correlation-based operator distribution is straightforward. First, we separate the operators into two classes. One class contains the "sticky" operators that are impractical to be moved on the fly. The other class contains the "floating" operators that have relatively small load migration overheads. We used the resilient operator distribution algorithm to distribute the sticky operators first. After the sticky operators are distributed, we use the correlation-based global operator distribution algorithm to distribute the remaining floating operators and balance the load of the system. As the load varies over time and the system becomes unbalanced, we use the pairwise correlation-based algorithm to move the floating operators pair-wisely to re-balance the load of the system. Sticky operators are moved only if a node become overloaded and there are no floating operators on the node.

Note that, although the resilient algorithm can be used to place all operators initially, it is better to use it to place the sticky operators only. This is because we want to keep the feasible set size for sticky operators to be large so that those operators do not need to be moved frequently. If we maximize the feasible set size for all operators together, we may put many sticky operators together and have to move them in the future.

8.4 Open Issues

Our study of the load management problem in distributed stream processing opens several issues to be explored in this area.

We have assumed that each operator is the minimum load distribution unit and have explored pipelined parallelism in distributed stream processing in this dissertation. Data partitioning based parallelism is another technique that can be used in load management. For example, if an Aggregate operator has a "group by" statement, we can create several copies of this operator and put these copies on different nodes. The input stream of this operator is then partitioned into sub-streams based on the group-by attribute, where each sub-stream feeds one copy of the aggregate operator. Finally, the results of the operator replicas are merged together by an Union operator and then sent to the downstream operator.

We have shown in resilient operator distribution that the ideal load distribution plan is achieved when the weights of each input stream is perfectly balanced across all nodes. It is natural to think that such a load distribution plan may be achieved by creating replicas of all operators and

partitioning all input streams accordingly. However, such a plan is difficult to achieve in practice because: (1) some operators do not support data partitioning based parallelism, (2) even if all operators in the system support data partitioning based parallelism, it is difficult to partition the input steams evenly at all times. For example, in the network traffic monitoring experiment with aggregate queries, we partitioned the input stream into 10 sub-streams based on the group-by attribute. Each sub-stream contained the same number of randomly chosen group-by values. We observed that the stream rates of all sub-streams varied significantly and the variations of different sub-streams were independent from each other. Therefore, these sub-streams should be considered as ten independent input streams rather than a certain percentage of the same input stream.

Ideally, we would like to partition an input stream such that the stream rate time series of the sub-streams have a pairwise correlation equal to one. Such a partition is good both for static operator distribution and dynamic operator distribution. The partitioning of an input stream into sub-streams with large stream rate correlation is a very challenging problem and is application dependent. However, it is worth exploring given its potential gains. In addition, even if we can partition an input stream into positively correlated sub-streams, we still need to answer the following questions:

1. Is it beneficial to partition the data streams and create operator replicas?
2. How many sub-streams and operator replicas to create?
3. What is the data percentage on each sub-stream?

All these questions open up future directions for further exploration.

Another open issue in resilient operator distribution is how to utilize extra information, such as upper bounds on input stream rates, variations on input stream rates etc., to further optimize the operator distribution plan. This information can be thought of as an extra constrains on workload set D. Recall that the ideal feasible set is independent of D. Thus, it is not necessary to consider the shape of D if the weights of all input streams can be well balanced. Otherwise, considering the shape of D may result in a large improvement because we can ignore the areas that never happen and optimize for the right area. However, due to the complexity of computing multiple integrals and the large number of possible operator distribution plans, incorporating extra information in the operator distribution algorithm is not trivial. For each kind of new information, new heuristics need to be explored and integrated into the operator distribution algorithm.

Finally, it would be interesting to investigate whether there are better ways to deal with systems with non-linear load models, rather than transforming their load models into linear ones.

Appendix A

Computing the Ideal Feasible Set Size

LEMMA 2: *Let*

$$F = \{(x_1,\ x_2,\ \cdots,\ x_d)^T : a_1x_1 + a_2x_2 + \cdots + a_dx_d \leq b\}.$$

Then

$$\int \cdots \int_F 1\ dx_1 \cdots dx_d = \frac{b^d}{d!} \cdot \frac{1}{\prod_{k=1}^{d} a_k}.$$

Proof.

$$\int \cdots \int_F 1\ dx_1 \cdots dx_d = \int_0^{\frac{b}{a_d}} \int_0^{\frac{b-a_dx_d}{a_{d-1}}} \int_0^{\frac{b-a_dx_d-a_{d-1}x_{d-1}}{a_{d-2}}} \cdots \int_0^{\frac{b-a_dx_d-\cdots-a_2x_2}{a_1}} 1\ dx_1 \cdots dx_d$$

Let

$$\begin{aligned}
y_0 &= b - a_dx_d - \cdots - a_3x_3 - a_2x_2 - a_1x_1 \\
y_1 &= b - a_dx_d - \cdots - a_3x_3 - a_2x_2 \\
y_2 &= b - a_dx_d - \cdots a_3x_3 \\
&\vdots \\
y_{d-1} &= b - a_dx_d. \\
y_d &= b.
\end{aligned}$$

Then

$$\begin{aligned} x_1 &= \frac{y_1 - y_0}{a_1} \\ x_2 &= \frac{y_2 - y_1}{a_2} \\ &\vdots \\ x_d &= \frac{y_d - y_{d-1}}{a_d}. \end{aligned}$$

Thus

$$\begin{aligned} & \int_F \cdots \int 1 \, dx_1 \cdots dx_d \\ = & \int_0^{\frac{y_d}{a_d}} \int_0^{\frac{y_{d-1}}{a_{d-1}}} \cdots \int_0^{\frac{y_2}{a_2}} \int_0^{\frac{y_1}{a_1}} 1 \, dx_1 \cdots dx_d \\ = & \frac{1}{\prod_{k=1}^d a_k} \int_0^{y_d} \cdots \int_0^{y_1} 1 \, dy_0 \cdots dy_{n-1} \\ = & \frac{1}{\prod_{k=1}^d a_k} \int_0^{y_d} \cdots \int_0^{y_2} y_1 dy_1 \cdots dy_{n-1} \\ = & \frac{1}{2} \cdot \frac{1}{\prod_{k=1}^d a_k} \int_0^{y_d} \cdots \int_0^{y_3} y_2^2 dy_2 \cdots dy_{d-1} \\ = & \frac{1}{3!} \cdot \frac{1}{\prod_{k=1}^d a_k} \int_0^{y_d} \cdots \int_0^{y_4} y_3^3 dy_3 \cdots dy_{d-1} \\ = & \vdots \\ = & \frac{1}{(d-1)!} \cdot \frac{1}{\prod_{k=1}^d a_k} \int_0^{y_d} y_{d-1}^{d-1} dy_{d-1} \\ = & \frac{1}{d!} \cdot \frac{1}{\prod_{k=1}^d a_k} \cdot y_d^d \\ = & \frac{1}{d!} \cdot \frac{1}{\prod_{k=1}^d a_k} \cdot b^d \end{aligned}$$

□

The ideal feasible set is defined by

$$F^* = \{R : R \in D, \; l_1 r_1 + l_2 r_2 + \cdots + l_d r_d \leq C_T\}.$$

Thus, the ideal feasible set size is

$$V(F^*) = \int_{F^*} \cdots \int 1 \; dr_1 \cdots dr_d = \frac{(C_T)^d}{d!} \cdot \prod_{k=1}^d \frac{1}{l_k}.$$

Appendix B

Notations

Table B.1: System Parameters

n	number of nodes
m	number of operators
d	number of system input streams
g	number of streams
N_i	the ith node
o_j	the jth operator
I_k	the kth input stream
S_t	the tth stream
C_i	the CPU capacity of node N_i
C	the CPU capacity vector
C_T	total node CPU capacities
C_i^b	the bandwidth capacity of node N_i
C^b	the node bandwidth capacity vector
C_T^b	the total bandwidth capacity of the system

Table B.2: Runtime statistics.

r_k	the stream rate of system input stream I_k
R	system input stream rate vector
c_j	the cost of operator o_j
s_j	the selectivity of operator o_j
c_t^c	the communication cost of stream S_t
$l(o_j)$	the load of operator o_j
$l(N_i)$	the load of node N_i
$l^c(S_t)$	the communication load of stream S_t
$b(S_t)$	the bandwidth of stream S_t
$b(N_i)$	the bandwidth consumption of node N_i

Table B.3: Notations related to static operator distribution.

D	workload set
A	operator allocation matrix
$F(A)$	feasible set of A
$F*$	ideal feasible set
$V(F*)$	ideal feasible set size
l^n_{ik}	load coefficient of node N_i for stream I_k
l^o_{jk}	load coefficient of operator o_j for stream I_k
l_k	total load coefficient of stream I_k
L^o	operator load coefficient matrix
L^n	node load coefficient matrix
$F'(L^n)$	feasible set of of L^n
w_{ik}	$(l^n_{ik}/l_k) \,/\, (C_i/C_T)$, weight of I_k on N_i
W	weight matrix
r	minimum plane distance
r^*	ideal plane distance
l^c_{tk}	the communication load coefficient of stream S_t for input I_k
r^c_{tu}	the communication ratio between S_t and o_u
r^c_t	the communication clustering ratio of S_t
b^s_{tk}	the bandwidth coefficient of stream S_t for input I_k
b^n_{ik}	the bandwidth coefficient of node N_i for input I_k
B^n	the bandwidth coefficient matrix
r^b_{tu}	the bandwidth ratio between S_t and o_u
r^b_t	the bandwidth clustering ratio of S_t
η^c	the communication clustering ratio threshold
η^b	the bandwidth clustering ratio threshold
w^c_t	the clustering weight of S_t
ψ	the clustering weight upper bound

Table B.4: Notations related to dynamic operator distribution.

X	system total load time series
X_i	load time series of node N_i
$\mathsf{E}X_i$	average load of node N_i
$\mathrm{var}X_i$	load variance of node N_i
ρ_{ij}	load correlation of node N_i and node N_j
$\rho(o_j, N_i)$	correlation coefficient between load time series of operator o_j and the total load time series of all operators on N_i except o_j
$S(o_j, N_i)$	score of operators o_j with respect to node N_j
h	number of samples on load time series
ε	load balancing threshold
δ	score threshold for operator exchange
θ	correlation improvement threshold

Bibliography

[1] The internet traffic archive. http://ita.ee.lbl.gov/.

[2] The medusa project. http://nma.lcs.mit.edu/projects/medusa/.

[3] Daniel J. Abadi, Yanif Ahmad, Magdalena Balazinska, Ugur Cetintemel, Mitch Cherniack, Jeong-Hyon Hwang, Wolfgang Lindner, Anurag S. Maskey, Alexander Rasin, Esther Ryvkina, Nesime Tatbul, Ying Xing, and Stan Zdonik. The design of the borealis stream processing engine. In *Proc. of CIDR*, 2005.

[4] Daniel J. Abadi, Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Aurora: A new model and architecture for data stream management. *The VLDB Journal*, 2003.

[5] A. Adas. Traffic models in broadband networks. *IEEE Communications*, 35(7):82–89, July 1997.

[6] Yanif Ahmad and Ugur etintemel. Network-aware query processing for stream-based applications. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, 2004.

[7] Arvind Arasu, Shivnath Babu, and Jennifer Widom. An abstract semantics and concrete language for continuous queries over streams and relations. In *Processing of the 9th International Conference on Data Base Programming Languages (DBPL '03)*, September 2003.

[8] Arvind Arasu, Mitch Cherniack, Eduardo Galvez, David Maier, Anurag S. Maskey, Esther Ryvkina, Michael Stonebraker, and Richard Tibbetts. Linear road: A stream data management benchmark. In *Proceedings of the 30th International Conference on Very Large Data Bases (VLDB'04)*, 2004.

[9] Ron Avnur and Joseph M. Hellerstein. Eddies: continuously adaptive query processing. In *Proc. of the 2000 ACM SIGMOD*, May 2000.

[10] Y. Azar, A. Broder, A. Karlin, , and E. Upfal. Balanced allocations. In *Proceedings of the 26th ACM Symposium on the Theory of Computing*, pages 593–602, 1994.

[11] Brian Babcock, Shivnath Babu, Mayur Datar, Rajeev Motwani, and Jennifer Widom. Models and issues in data stream systems. In *Proc. of 2002 ACM PODS*, June 2002.

[12] Magdalena Balazinska, Hari Balakrishnan, Samuel Madden, and Michael Stonebraker. Fault-tolerance in the borealis distributed stream processing system. In *Proc. of the 2005 ACM SIGMOD*, June 2005.

[13] Magdalena Balazinska, Hari Balakrishnan, and Michael Stonebraker. Contract-based load management in federated distributed systems. In *Proc. of the 1st NSDI*, 2004.

[14] Earl R. Barnes, Anthony Vannelli, and James Q. Walker. A new heuristic for partitioning the nodes of a graph. *SIAM Journal on Discrete Mathematics*, 1(3):299–305, August 1988.

[15] Krishna M. Kavi Behrooz A. Shirazi, Ali R. Hurson, editor. *Scheduling and load balancing in parallel and distributed systems*. IEEE Computer Science Press, 1995.

[16] Marsha J. Berger and Shahid H. Bokhari. A partitioning strategy for nonuniform problems on multiprocessors. *IEEE Transactions on Computers*, 36(5):570–580, May 1987.

[17] John Bloomer. *Power Programming with RPC*. O'Rilley and Associates, 1992.

[18] Laura Bottomley. Dec-pkt, the internet traffic archive. http://ita.ee.lbl.gov/html/contrib/DEC-PKT.html.

[19] Luc Bouganim, Daniela Florescu, and Patrick Valduriez. Dynamic load balancing in hierarchical parallel database systems. In *Proceedings of 22th International Conference on Very Large Data Bases (VLDB'96)*, pages 436–447, September 1996.

[20] Don Carney, Uğur Çetintemel, Mitch Cherniack, Christian Convey, Sangdon Lee, Greg Seidman, Michael Stonebraker, Nesime Tatbul, and Stan Zdonik. Monitoring streams: A new class of data management applications. In *Proc. of the 28th VLDB*, August 2002.

[21] Don Carney, Uğur Çetintemel, Alexander Rasin, Stan Zdonik, Mitch Cherniack, and Michael Stonebraker. Operator scheduling in a data stream manager. Technical Report CS-03-04, Department of Computer Science, Brown University, February 2003.

[22] T.L. Casavant and J.G. Kuhl. A taxonomy of scheduling in general-purpose distributed computing systems. *IEEE Transactions on Software Engineering*, 14(2):141–154, Febrary 1988.

[23] Sirish Chandrasekaran, Amol Deshpande, Mike Franklin, and Joseph Hellerstein. TelegraphCQ: Continuous dataflow processing for an uncertain world. In *Proc. of CIDR*, 2003.

[24] Sirish Chandrasekaran and Michael J. Franklin. Psoup: a system for streaming queries over streaming data. *The VLDB Journal*, 12:140–156, 2003.

[25] Jianjun Chen, David J. DeWitt, Feng Tian, and Yuan Wang. Niagaracq: A scalable continuous query system for internet databases. In *Proc. of the 2000 ACM SIGMOD*, May 2000.

[26] Mitch Cherniack, Hari Balakrishnan, Magdalena Balazinska, Don Carney, Uğur Çetintemel, Ying Xing, and Stan Zdonik. Scalable distributed stream processing. In *Proc. of CIDR*, 2003.

[27] M. G. Everett Chris Walshaw, Mark Cross. Dynamic load balancing for parallel adaptive unstructured meshes. In *Processing of the 8th SIAM Conference on Parallel Processing for Scientific Computing*, October 1997.

[28] E. G. Coffman, M. R. Garey Jr., and D. S. Johnson. Approximation algorithms for binpacking - an updated survey. *Algorithm Design for Computer Systems Design*, pages 49–106, 1984.

[29] Antonio Corradi, Letizia Leonardi, and Franco Zambonelli. Diffusive load-balancing policies for dynamic applications. *IEEE Concurrency*, 7(1), 1999.

[30] Mark Crovella and Azer Bestavros. Self-similarity in world wide web traffic: Evidence and possible causes. *IEEE/ACM Transactions on Networking*, pages 835–846, 1997.

[31] G. Cybenko. Dynamic load balancing for distributed memory multiprocessors. *Journal of Parallel and Distributed Computing*, 2(7):279–301, 1989.

[32] George Bernard Dantzig. *Linear programming and extensions.* Princeton University Press, Princeton, 1963.

[33] R. Diekmann, B. Monien, and R. Preis. Using helpful sets to improve graph bisections. *Interconnection Networks and Mapping and Scheduling Parallel Computations, volume 21 of DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 57–73, 1995.

[34] Ralf Diekmann, Burkhard Monien, , and Robert Preis. Load balancing strategies for distributed memory machines. *Multi-Scale Phenomena and Their Simulation*, pages 255–266, 1997.

[35] Pedro Diniz, Steve Plimpton, Bruce Hendrickson, and Robert Leland. Parallel algorithms for dynamically partitioning unstructured grids. In *Proceedings of the 7th SIAM Conference of Parallel Processing in Scientific Computing*, 1995.

[36] Derek L. Eager, Edward D. Lazowska, and John Zahorjan. Adaptive load sharing in homogeneous distributed systems. *IEEE Transactions on Software Engineering*, 12(5):662–675, May 1986.

[37] Joseph E. Flaherty, R. M. Loy, P. C. Scully, M. S. Shephard, Boleslaw K. Szymanski, J. D. Teresco, and Louis H. Ziantz. Load balancing and communication optimization for parallel adaptive finite element methods. In *International Conference of the Chilean Computer Science Society*, pages 246–255, 1997.

[38] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Freeman, New York, 1979.

[39] John. R. Gilbert, Gary L. Miller, and Shang-Hua Teng. Geometric mesh partitioning: Implementation and experiments. *SIAM Journal on Scientific Computing*, 19(6):2091–2110, 1998.

[40] Lukasz Golab and M. Tamer Ozsu. Issues in data stream management. *ACM SIGMOD Record*, 32(2):5–14, 2003.

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

[42] Deepak Gupta and Pradip Bepari. Load sharing in distributed systems. In *Proceedings of the National Workshop on Distributed Computing*, January 1999.

[43] William W. Hager, Soon Chul Park, and Timothy A. Davis. *Blockexchange in graph partitioning*. Kluwer Academic Publishers, 1999.

[44] Bruce Hendrickson and Robert Leland. An improved spectral graph partitioning algorithm for mapping parallel computations. *SIAM Journal on Scientific Computing*, 16(2):452–469, 1995.

[45] Jiawei Hong, Xiaonan Tan, and Marina Chen. From local to global: An analysis of nearest neighbor balancing on hypercube. *Performance Evaluation Review*, 16(1):73–82, May 1988.

[46] G Horton. A multi-level diffusion method for dynamic load balancing. *Parallel Computing*, 19(2):209–218, 1993.

[47] Jeong-Hyon Hwang, Magdalena Balazinska, Alexander Rasin, Uğur Çetintemel, Michael Stonebraker, and Stan Zdonik. High-availability algorithms for distributed stream processing. Technical Report CS-04-05, Department of Computer Science, Brown University, June 2004.

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

[49] George Karypis and Vipin Kumar. A fast and high quality multilevel scheme for partitioning irregular graphs. *SIAM Journal on Scientific Computing*, 20(1):359–392, January 1999.

[50] Sailesh Krishnamurthy, Sirish Chandrasekaran, Owen Cooper, Amol Deshpande, Michael J. Franklin, Joseph M. Hellerstein, Wei Hong, Samuel R. Madden, Fred Reiss, and Mehul A. Shah. Telegraphcq: An architectural status report. *IEEE Data Engineering Bulletin*, 26(1):11–18, 2003.

[51] Will E. Leland, Murad S. Taqqu, Walter Willinger, and Daniel V. Wilson. On the self-similar nature of ethernet traffic. In *Proc. of SIGCOMM*, 1993.

[52] Frank C. H. Lin and Robert M. Keller. The gradient model load balancing method. *IEEE Transactions on Software Engineering*, 13(1):32–38, January 1987.

[53] J. Liu. A multilevel load balancing algorithm in a distributed system. In *Proceedings of the 19th ACM Annual Computer Science Conference*, March 1991.

[54] Miron Livny and Myron Melman. Load balancing in homogeneous broadcast distributed systems. *ACM Computer Network Performance Symposium*, pages 47–55, April 1982.

[55] Sam Madden, Mehul A. Shah, Joseph M. Hellerstein, and Vijayshankar Raman. Continuously adaptive continuous queries over streams. In *Proc. of the 2002 ACM SIGMOD*, 2002.

[56] Inc. Mesquite Software. Csim 18 simulation engine. http://www.mesquite.com/.

[57] Dejan S. Milojicic, Fred Douglis, Yves Paindaveine, Richard Wheeler, and Songnian Zhou. Process migration. *ACM Computing Surveys*, 32(3):241–299, September 2000.

[58] Rajeev Motwani, Jennifer Widom, Arvind Arasu, Brian Babcock, Shivnath Babu, Mayur Datar, Gurmeet Manku, Chris Olston, Justin Rosenstein, and Rohit Varma. Query processing, approximation, and resource management in a data stream management system. In *Proc. of CIDR*, 2003.

[59] Jeffrey F. Naughton, David J. DeWitt, David Maier, Ashraf Aboulnaga, Jianjun Chen, Leonidas Galanis, Jaewoo Kang, Rajasekar Krishnamurthy, Qiong Luo, Naveen Prakash, Ravishankar Ramamurthy, Jayavel Shanmugasundaram, Feng Tian, Kristin Tufte, Stratis Viglas, Yuan Wang, Chun Zhang, Bruce Jackson, Anurag Gupta, and Rushan Chen. The niagara internet query system. *IEEE Data Engineering Bulletin*, 24(2):27–33, 2001.

[60] Harald Niederreiter. *Random Number Generation and Quasi-Monte Carlo Methods*. Society for Industrial and Applied Mathematics, Philadelphia, 1992.

[61] Mark Nuttall. A brief survey of systems providing process or object migration facilities. *Operating System Review*, 28(4):64–80, October 1994.

[62] Mark Nuttall and Morris Sloman. Workload characteristics for process migration and load balancing. In *Proceedings of the IEEE International Conference on Distributed Computing Systems (ICDCS'97)*, May 1997.

[63] Leonid Oliker and Rupak Biswas. Plum: Parallel load balancing for adaptive unstructured meshes. *Journal of Parallel and Distributed Computing*, 52(2):150–177, 1998.

[64] Anthony L. Peressini, Francis E. Sullivan, and J.J. Jr. Uhl. *The Mathematics of Nonlinear Programming*. Springer-Verlag, 1988.

[65] Peter R. Pietzuch, Jonathan Ledlie, Jeffrey Shneidman, Mema Roussopoulos, Matt Welsh, and Margo I. Seltzer. Network-aware operator placement for stream-processing systems. In *Proc. of the 19th ICDE*, 2006.

[66] Alex Pothen, Horst D. Simon, and Kang-PuLiou. Partitioningsparse matrices with eigenvectors of graphs. *SIAM J. on Matrix Analysis and Applications*, 11(3):430–452, 1990.

[67] Alex Pothen, Horst D. Simon, Lie Wang, and Stephen T. Bernard. Towards a fast implementation of spectral nested dissection. In *Processings of Supercomputing'92*, 1992.

[68] X.-S. Qian and Q. Yang. Load balancing on generalized hypercube and mesh multiprocessors with lal. In *Proceedings of 11th International Conference on Distributed Computing Systems*, pages 402–409, 1991.

[69] Erhard Rahm and Robert Marek. Dynamic multi-resource load balancing in parallel database systems. In *Proceedings of the 21st International Conference on Very Large Data Bases (VLDB'95)*, pages 395–406, 1995.

[70] Kirk Schloegel, George Karypis, and Vipin Kumar. Multilevel diffusion schemes for repartitioning of adaptive meshes. *Journal of Parallel and Distributed Computing*, 47(2):109–124, 1997.

[71] Kirk Schloegel, George Karypis, and Vipin Kumar. Graph partitioning for high performance scientific simulations. In *CRPC Parallel Computing Handbook*. Morgan Kaufmann, 2000.

[72] Mehul A. Shah, Joseph M. Hellerstein, and Eric Brewer. Highly-available, fault-tolerant, parallel dataflows. In *Proc. of the 2004 ACM SIGMOD*, June 2004.

[73] Mehul A. Shah, Joseph M. Hellerstein, Sirish Chandrasekaran, and Michael J. Franklin. Flux: An adaptive partitioning operator for continuous query systems. In *Proc. of the 19th ICDE*, March 2003.

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

[75] Horst D. Simon, Andrew Sohn, and Rupak Biswas. Harp: A fast spectral partitioner. In *Proceedings of the Ninth ACM Symposium on Parallel Algorithms and Architectures*, pages 22–25, 1997.

[76] Ian H. Sloan and Stephen Joe. *Lattice Methods for Multiple Integration*. Clarendon Press, Oxford, 1994.

[77] Utkarsh Srivastava, Kamesh Munagala, and Jennifer Widom. Operator placement for in-network stream query processing. In *Proceedings of the 24th ACM SIGMOD-SIGACT-SIGART Symposium on Principles of Database Systems*, 2005.

[78] John A. Stankovic. Stability and distributed scheduling algorithms. *IEEE Transactions on Software Engineering*, 11(10):1141–1152, October 1985.

[79] R. Subramanian and I. D. Scherson. An analysis of diffusive load balancing. In *Proceedings of the 1994 ACM Symposium on Parallel Algorithms and Architectures*, pages 220– 225, June 1994.

[80] Mark Sullivan. Tribeca: A stream database manager for network traffic analysis. In *Proceedings of the 22th International Conference on Very Large Data Bases table of contents (VLDB'96)*, September 1996.

[81] Mark Sullivan and Andrew Heybey. Tribeca: A system for managing large databases of network traffic. In *Proceedings of the USENIX annual technical conference*, pages 15–19, June 1998.

[82] Mike Tanner. *Practical Queueing Analysis*. McGraw-Hill Book Company, London, 1995.

[83] Douglas Terry, David Goldberg, David Nichols, and Brian Oki. Continuous queries over append-only databases. In *Proc. of the 1992 ACM SIGMOD*, pages 321–330, January 1992.

[84] Feng Tian and David J. DeWitt. Tuple routing strategies for distributed eddies. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, September 2003.

[85] Yung-Terng Wang and Robert J. T. Morris. Load sharing in distributed systems. *IEEE Transactions on Computers*, 34(3):204–217, March 1985.

[86] Eric W. Weisstein. Change of variables theorem. MathWorld – A Wolfram Web Resource, http://mathworld.wolfram.com/ChangeofVariablesTheorem.html.

[87] Eric W. Weisstein. Hypersphere. MathWorld – A Wolfram Web Resource, http://mathworld.wolfram.com/Hypersphere.html.

[88] Eric W. Weisstein. Polytope. MathWorld – A Wolfram Web Resource, http://mathworld.wolfram.com/Polytope.html.

[89] Marc H. Willebeek-LeMair and Anthony P. Reeves. Strategies for dynamic load balancing on highly parallel computers. *IEEE Transaction on Parallel and Distributed Systems*, 4(9):979–993, September 1993.

[90] Roy D. Williams. Performance of dynamic load balancing algorithms for unstructured mesh calculations. *Concurrency*, 3:457–481, 1991.

[91] W. Willinger, M.S. Taqqu, R. Sherman, and D.V. Wilson. Self-similarity through high variability: statistical analysis of ethernet lan traffic at the source level. *IEEE/ACM Transactions on Networking*, 5(1):71–86, 1997.

[92] Ying Xing, , Stan Zdonik, and Jeong-Hyon Hwang. Dynamic load distribution in the borealis stream processor. In *Proc. of the 19th ICDE*, March 2005.

[93] C. Xu, B. Monien, R. Luling, and F. Lau. Nearest neighbor algorithms for load balancing in parallel computers. *Concurrency: Practice and Experience*, 9(12):1351–1376, 1997.

[94] Cheng-Zhong Xu and Francis C. M. Lau. Optimal parameters for load balancing with the diffusion method in mesh networks. *Parallel Processing Letters*, 4(1-2):139–147, 1994.

[95] Stan Zdonik, Michael Stonebraker, Mitch Cherniack, Ugur Cetintemel, Magdalena Balazinska, and Hari Balakrishnan. The aurora and medusa projects. *Bulletin of the Technical Committe on Data Engineering*, March 2003.

[96] Songnian Zhou. A trace-driven simulation study of dynamic load balancing. *IEEE Transactions on Software Engineering*, 14(9):1327–1341, September 1988.

[97] Yongluan Zhou. Adaptive distributed query processing. In *Proceedings of the 29th International Conference on Very Large Data Bases (VLDB'03)*, 2003.

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