QueueLinker: A Framework for Parallel Distributed Data-Stream Processing

QueueLinker: データストリームのための並列分散処理フレームワーク

March 2013

Waseda University
Graduate School of Fundamental Science and Engineering,
Major in Computer Science and Engineering,
Research on Parallel and Distributed Architecture

Takanori UEDA

Acknowledgements

Firstly, I would like to express my immense gratitude toward my supervisor, Prof. **Hayato Yamana**, of the Faculty of Science and Engineering at Waseda University. This research started as part of a big project led by him. Without the help and guidance of Prof. Yamana, the completion of this research would have been a far more difficult task. The state-of-the-art computer resources in our laboratory, which were essential for carrying out my research, are a result of his dedicated efforts. Prof. Yamana also encouraged me to give many conference presentations and conduct a variety of academic activities, and these gave me the opportunity to harness my skills and create personal connections.

I would also like to express my gratitude to Prof. **Yoichi Muraoka** and Prof. **Tatsuo Nakajima**, both of the Faculty of Science and Engineering, Waseda University, and Dr. **Sayaka Akioka**, of the Information Technology Research Organization, Waseda University. I received a great deal of advice for this doctor thesis from them. Their deep knowledge of operating systems and parallel distributed computing led to enlightening discussions that helped me to better understand my subject. In addition, international conferences and business trips with them provided many valuable experiences of foreign societies.

Dr. **Andrew Sohn**, associate professor at the New Jersey Institute of Technology, gave much-appreciated advice for my research. In addition to him, my research career has been supported by many people outside Waseda University. Dr. **Hideyuki Kawashima**, assistant professor at Tsukuba University, gave me a great deal of support and many academic opportunities. Drinking parties with him are always interesting. Prof. **Hiroyuki Kitagawa**, also of Tsukuba University, provided some immensely important advice for my research. Without their invaluable discussions, I could not have completed the key research of this thesis, and would not have attained the best paper award at DEIM 2012. Dr. **Yui Makoto**, of the National Institute of Advanced Industrial Science and Technology, provided many opportunities for my research career. A big project on distributed computing with him deeply affected my research.

During my internship at IBM Research Tokyo, I was supported by Dr. **Toshio Nakatani**, Dr. **Moriyoshi Ohara,** and Dr. **Hiroshi Horii**. This experience with actual products provided me with great experience that could not have been achieved in my university. Moreover, the intern experience gave me an understanding of the importance of database systems.

Here, I would also like to thank all of the members of Prof. Yamana's laboratory.

In particular, I would like to thank Mr. **Kousuke Morimoto** and Mr. **Kenji Uchida,** who have graduated from the laboratory, and Mr. **Kou Satoh,** who is currently my junior colleague. It would not have been possible to develop the Web crawler without their help. In addition, Mr. **Hiroaki Asai** provided me with valuable Web data, including the Twitter streams that were indispensable in developing and testing my framework. Mr. **Daichi Suzuki** and Mr **Yusuke Yamamoto** helped my research and managed the large number of servers. Mr. **Hiromasa Takei** has a deep knowledge of mathematics, and discussions with him provided many interesting research ideas.

I would also like to thank the alumni of Prof. Yamana's laboratory. First of all, I would like to thank my old senior colleague, Dr. **Yu Hirate**. His attitude toward research work and leadership qualities have always inspired me. He has consistently offered help wholeheartedly and been a friend to me during challenging times. Mr. **Takashi Tashiro**, who provided me with the opportunity to join Prof. Yamana's laboratory, had a positive effect on my research when I was a master's course student. I would also particularly like to thank Mr. **Hiroaki Katase** for his assistance. The all-night server-maintenance on which we worked together stands out in my memory as a pleasant experience. Mr. **Nobuyuki Kubota** and Mr. **Naoyoshi Aikawa** have also had a positive effect on my attitude towards research. I cannot mention all of their names, but I would like to thank each and every person working in Prof. Yamana's laboratory.

Finally, I would like to express my gratitude towards my parents, **Hirozo and Saeko Ueda**. They provided unconditional emotional support whenever it was needed, and this enabled me to complete these difficult research activities. I thank them from the bottom of my heart.

March 2013

Takanori Ueda

Abstract

Data stream analysis is required in a variety of situations. In particular, high-speed data stream analysis demands the use of parallel and distributed processing capability; however, implementing the necessary concurrency control and network communication to do so represents a challenge to developers. Some data stream applications such as algorithmic trading can benefit from the reduction of processing latency by even a few microseconds, and to support developers in realizing their requirements, a framework for parallel-distributed data-stream processing is indispensable.

The proposed QueueLinker framework enables the building of data-stream applications by implementing modules using a producer–consumer model and by specifying a logical directed graph representing the data-flow connection between modules. As QueueLinker executes modules in a parallel-distributed environment and performs data transfer between modules according to a logical directed graph, data streams can be processed in real-time.

One important application of QueueLinker is the execution of continuous queries. To cope with the latency requirements, in this thesis a low-latency parallelization technique for executing continuous queries in a multi-core environment is proposed. The proposed method attempts to assign the operators used in a query to threads in such a way that latency owing to inter-thread communication can be reduced, and it incorporates a dynamic operator reallocation mechanism that does not require that the framework stop processing the stream during operator reallocation.

QueueLinker can also be used to execute continuous queries within a commodity cluster environment. Chase Execution is a proposed backup method for reducing the latency of a query in an environment in which an operator must be replicated over multiple computers in order to cope with computational node failures. The proposed method attempts to reduce processing latency and manage changes in data stream speed by executing a secondary using set of operator deployments differing from those of the primary; correspondingly, query results are generated using the fastest tuples outputted by the respective deployments.

QueueLinker is useful for more than low latency data stream processing: a high-speed parallel-distributed Web crawler can be implemented using QueueLinker. Because each of its modules is implemented with data structures that are temporally and spatially efficient, this crawler would be able to crawl the Web on a large scale while conserving resources with improved load balancing and memory utilization between computers as compared to traditional site-based Web crawlers.

Contents

List of Figures

List of Tables

Chapter 1. Introduction

With the internet now a common component of our infrastructure and the connection of many mobile devices to the network, our lifestyles have changed dramatically. A constant stream of information is now posted to social network services with 400 million Twitter tweets generated worldwide per day[1] and the number of digital sensors generating 'data streams' that contain valuable information has increased considerably. The analysis of data streams in real-time is important in many applications including—among others—social analysis, stock market predicting, and healthcare monitoring. Several data stream applications are even affected by processing latency on the order of microseconds. Reducing latency in stock trading algorithms provides an advantage over trading competitors, as a trading order that reaches the central trading server first will be processed prior to those from other companies, and Wall Street algorithms will reportedly be uncompetitive if they experience a 5μs processing delay.[2]

Advances in information technology have corresponded to an increase in the number of mobile devices and sensors, which in turn has resulted in the generation of large numbers of data streams, and processing numerous large streams requires parallel-distributed computing. Recent advancements in commodity computer hardware have made parallel-distributed computing available for everyone. As CPU manufacturers have decided to increase the number of cores on a CPU instead of increasing individual core frequency, the resulting plummet in personal computer prices has allowed for the ownership of shared-nothing clusters at a low cost. In order to take full advantage of recent computer hardware performance, developers must now become familiar with parallel-distributed computing; however, most developers want to avoid implementing concurrency control and network communication procedures, as these have proven difficult to program.

Processing data streams in real-time presents special programming difficulties owing to the fact that applications must handle multiple data sources, wait for data arrival, and receive data via the network. In addition, the management of high-speed data streams demands parallel-distributed processing and, as described above, many applications require low-latency data stream processing. In addition, data stream

[1] Twitter hits 400 million tweets per day, mostly mobile, http://news.cnet.com/8301-1023_3-57448388-93/twitter-hits-400-million-tweets-per-day-mostly-mobile/

[2] K. Slavin, "How algorithms shape our world," http://www.ted.com/talks/lang/en/kevin_slavin_how_algorithms_shape_our_world.html

arrival speed and data content can change under certain conditions. Because it is difficult for many programmers to consider all of these factors, it is vital that a framework for real-time, parallel-distributed data-stream processing be developed.

1.1. Research Motivation

To summarize the issues and circumstances described above, the following are the requirements that this thesis attempts to address and the targets to which it tries to contribute:

- Requirements of a Framework for Parallel Distributed Data-Stream Processing
  - Analyzing data streams in real-time is required in a variety of situations.
  - To take full advantage of recent computer hardware developments in processing data streams, parallel-distributed computing is indispensable.
  - Developers want to be able to easily develop data stream applications without necessitating complicated implementations of network communication and concurrency controls.
- Requirements for Parallel Distributed Low Latency Data-Stream Processing
  - Several applications are affected by processing latencies that are now, in some case, on the order of microseconds.
  - Speed and content changes by data streams must be considered in order to achieve low-latency processing.

Conventional frameworks such as MapReduce [1] are not designed for data stream processing. Data stream management systems (DSMSs) [2,3,4,5,6,7] are designed to process data streams through continuous query; however, operator execution mechanisms such as Chain [8], Eddy [9], and Teddy [10] do not consider latency in parallel computing and fail to consider communication latency between threads or CPU cores. Operator execution mechanisms for distributed data-stream processing such as COLA [11], SODA [12], and RASC [13] assume network communication latency to be negligibly small. However, even a small amount of latency can seriously affect data stream applications. Recent frameworks such as Twitter Storm[1], Apache S4[2], and Esper[3] are designed to process data streams. However, they are not designed to realize low-latency processing in parallel-distributed computing.

[1] Storm, distributed and fault-tolerant realtime computation, http://storm-project.net/
[2] S4: Distributed Stream Computing Platform, http://incubator.apache.org/s4/
[3] Esper - Complex Event Processing, http://esper.codehaus.org/

Low-latency processing techniques are closely correlated to framework design. Therefore, we must consider both a framework design and low latency processing techniques.

1.2. Contributions

In this thesis, QueueLinker—a framework for parallel-distributed data-stream processing—is proposed. To cope with latency requirements, low-latency techniques for processing data streams are proposed. In addition, QueueLinker can also be used to execute applications such as Web crawlers. The contributions of this thesis are listed below and described in this section.

1. QueueLinker Development (Chapter 3)
   - QueueLinker is a framework for parallel-distributed data-stream processing.
   - The framework provides a producer–consumer programming model for data-stream processing.
   - Developers can implement data stream applications without implementing network communication and concurrency controls.
   - The framework supports a data parallel model.
2. Low Latency Processing of Continuous Queries (Chapters 4 and 5)
   - Chapter 4 describes a proposed low-latency, parallel execution method for continuous query on a multi-core processor.
   - Chapter 5 describes a proposed backup method for continuous query that achieves low-latency processing in a commodity cluster environment.
3. Application Demonstration (Chapter 6)
   - In addition to low-latency data stream processing, QueueLinker can execute practical applications such as Web crawlers.
   - The crawler can be executed using a data parallel model of QueueLinker.
   - The crawler achieves better load-balancing and memory utilization between computers than traditional site-based Web crawlers.

1.2.1. QueueLinker Development (Chapter 3)

The proposed framework, QueueLinker, will enable developers to build data stream applications by implementing JAVA modules that utilize a producer–consumer model and by specifying a logical directed graph representing the data-flow connection between modules. Each module is automatically instantiated and executed within a parallel-distributed environment, and data are automatically serialized and transferred

to other modules, even if these are running on other computers. Programmers do not need to implement multi-threaded program and network communication procedures. QueueLinker also provides an application programming interface (API) to realize data parallel computation. By using this data parallel model, developers can easily scale-up applications.

1.2.2. Low Latency Execution of Continuous Query (Chapters 4 and 5)

The mechanism of QueueLinker supports the implementation of useful applications, including "continuous query [14]," which has been studied in the field of database science since the early 2000s. A relational continuous query can usually be described using an SQL-like language [15] and compiled to a plan tree consisting of relational algebra operators. When a tuple arrives to the system, the tuple is pushed into a leaf of the plan tree and the plan tree generates the result of the query.

Continuous query is useful in data stream processing. Previously, data stream management systems (DSMSs) [2,3,4,5,6,7] had been developed to process data streams using continuous query, as DSMSs have scheduling algorithms for operator execution. However, operator execution mechanisms such as Chain [8], Eddy [9], Teddy [10], COLA [11], SODA [12], and RASC [13] do not take into account the communication latency associated with parallel-distributed computing.

QueueLinker can also be used to execute continuous query by implementing a relational algebra operator as a module, and the corresponding plan tree can be described using a logical directed graph. Thus, QueueLinker is useful in executing continuous query in a parallel-distributed environment; however, as the method of operator execution is quite relevant to the reduction in query latency, the following two methods are proposed in this thesis:

- Low-latency execution of a continuous query on a multi-core processor (Chapter 4)
  - A low-latency parallel execution method of continuous query on a multi-core CPU is proposed that attempts to assign threads to query operators in such a way that the latency of inter-thread communication is reduced. This thesis proposes a definition of the tuple latency of continuous query in parallel processing and proposes a dynamic programming algorithm that finds thread assignments for operators. In addition, a dynamic operator reallocation technique that does not require the framework to stop processing a stream during reallocation is proposed.

- Chase Execution – a backup method using latency reduction in a distributed environment (Chapter 5)
  - Chase Execution is a proposed backup method for achieving low-latency processing of continuous queries in a commodity cluster environment where operators must be replicated within multiple computers in order to cope with computation node failures. In this proposed method, a secondary is executed by a set of operator deployments that differs from those of the primary, and query results are generated from the fastest tuples outputted by the deployments. This method aims to reduce the processing latency while managing changes in data streams speed.

1.2.3. Parallel-distributed Web Crawler (Chapter 6)

In addition to its usefulness in low-latency data stream processing, QueueLinker can be used to implement a high-speed, parallel-distributed Web crawler. As Web crawlers must collect Web data while performing tasks such as the detection of crawled URLs and the prevention of consecutive access to a certain Web server, parallel and distributed crawling is necessary in achieving high-speed crawling of the extremely high number of URLs that exist on the Web.

The proposed Web crawler consists of fine-grained QueueLinker modules, all of which can be executed using the data parallel model. Because of this, the crawler can achieve improved load balancing and memory utilization between computers as compared to traditional site-based Web crawlers [16,17,18]. Moreover, because each module is implemented by data structures that are temporally and spatially efficient, crawling of the Web on a large scale while conserving resources is enabled. This crawler thus demonstrates that the data parallel model provided by QueueLinker is effective in the development of Web crawlers.

1.3. Outline of this Thesis

The remainder of this thesis is organized as follows:

- Chapter 2 describes data stream processing preliminaries.
- Chapter 3 describes QueueLinker, the proposed framework for parallel distributed data-stream processing.
- Chapter 4 describes the proposed method for low-latency parallel execution of continuous query on a multi-core CPU.

- Chapter 5 describes the proposed backup method for achieving low latency processing of continuous queries in a commodity cluster environment.
- Chapter 6 describes the proposed parallel-distributed Web crawler.
- Chapter 7 concludes this thesis and gives a viewpoint on potential future work.

Chapter 2. Preliminaries of Data Stream Processing

A data stream is a sequence of data items that arrive continuously and permanently. Analyzing data streams in real-time is essential to many applications, many of which also require very low latency. To fulfill these requirements, it is important to consider that the speed at which the data stream arrives, and the character of data contained in the stream, can vary widely under certain conditions.

To clarify the differences between data stream processing and traditional data processing, this chapter provides a review of traditional data processing software, such as Database Management Systems (DBMSs) and distributed computing frameworks like MapReduce [1]. It then provides a review of Data Stream Management Systems (DSMSs) [2,3,4,5,6,7] developed especially for executing so-called "continuous queries" [14] of data streams in real-time. A continuous query is compiled to a plan tree consisting of relational algebra operators. When a tuple arrives, it is pushed into a leaf of the plan tree and the result of the query is generated. The execution strategy of relational operators affects the latency of the query. This chapter reviews previous research about strategies of operator execution in continuous queries, and provides a point of reference for the remaining chapters.

2.1. Data Streams Characteristics

There are many different types of data streams. Everything from network packets and digital sensor outputs to game control events and social message channels can be considered data streams, regardless of data format. Even so, there are several characteristics that commonly apply to modern data streams:

- **Continuousness:** A data stream arrives continuously from outside data sources, and cannot be controlled or stopped, even when the receiving system has insufficient resources to process it. If the system cannot process the data as it arrives, it has to discard some part of it, or adopt an approximation technique, such as load shedding [19], to reduce computation time.
- **Large Data Size:** Advances in information technology have corresponded to an increase in the number of mobile devices and sensors, which in turn has resulted in the generation of large numbers of data streams, most of which cannot be processed outside a parallel-distributed computing context. In addition, as the data stream arrives continuously, it is usually impossible to store the data to secondary storage.

Thus, a system usually has to process a data stream in memory, in real time.

- **Arrival Rate Change:** The rate at which data arrives from the source may vary sharply and without notice. The receiving system must respond to rate changes by adapting computation resources, like CPU cores and memory, to reduce processing latency. In distributed data-stream processing, such adaptation is more difficult because internal state must migrate between computers.
- **Importance of Data Arrival Order:** A data item is usually represented by a tuple with a timestamp or a tuple ID that increases over time. For many applications, processing tuples that are not ordered by the time of their arrival will produce incorrect results.
- **Multiple Data Sources:** Some applications need multiple data sources to generate results, and must therefore manage multiple data streams in such a way that queries can be executed promptly.

The above characteristics are important to the design of software for processing data-streams. The next section provides actual examples of data stream applications.

2.2. Data Stream Applications and their Characteristics

Real-time data stream analysis is an essential component of many modern systems and platforms. All of the following applications rely on such analysis for their core function:

- **Financial Applications**: Market assets are often traded using algorithms that decide when to buy or sell depending on market prices, economic indices, and relevant events from newswires. These algorithms have to submit orders to the central server in an exchange as soon as possible to avoid arbitrage losses. To illustrate this point, the Tokyo Stock Exchange, Inc. states that its "arrownet" access to the core trading system can supply data at a delay of less than 32μs one-way[1], and algorithms on Wall Street are commonly supposed to lose if delayed by more than 5μs[2]. Thus, reducing latency in data stream analysis to an absolute minimum is crucial to success in algorithmic trading.
- **Social Analytics**: Using the connectivity of modern mobile devices, numerous users all over the world have started updating their moment-to-moment activities on

[1] TSE : arrownet, http://www.tse.or.jp/english/system/networkservices/arrownet.html
[2] K. Slavin, "How algorithms shape our world,"
http://www.ted.com/talks/lang/en/kevin_slavin_how_algorithms_shape_our_world.html

social network services. Analyzing these activity streams in real-time now forms the basis of a number of services, most notably those involving market analysis and targeted advertising.

- **Sensor Applications**: Modern micro-device technology has led to a proliferation of sensory apparatus. Temperature and tracking sensors in smart homes can reduce energy consumption, security cameras can detect criminal behavior in public and private spaces, and GPS locators can keep track of property and people.
- **Network Monitoring**: The most prevalent data stream of all is the one composed of network packets, and a number of applications are dedicated to direct analysis of this stream. An IPS (Intrusion Prevention System), for example, tries to detect and discard malicious packets in real time. Because of the nature of TCP communications, throughput between clients and servers can be noticeably degraded if an IPS cannot keep up with the speed of incoming packets.

2.3. Difficulties in Data Stream Processing

Applications such as those listed above must generally address three main problems in data stream processing:

- **Large Data**: Data streams want to flow freely, and carrier routers can be configured to switch packets at a speed of 322 Tbps[1]. With millions of financial transactions occurring every minute, users generating 400 million tweets per day[2], and security cameras producing untold amounts of pixel data, systems intended to process such data streams must be designed for massive throughput.
- **Low Latency**: Delays of several microseconds can cause critical deficiencies for some applications. If a trading algorithm or IPS incurs undue latency, significant damage may result.
- **High Availability**: Some applications cannot tolerate data loss. For example, if an IPS cannot process packets when a failure occurs, its network becomes unusable. For such applications, stream processing state must be replicated to multiple computers in order to prevent fatal losses.

Processing data streams in real-time presents a steep programming challenge. Applications must handle multiple data sources, remain available for data arrival, and

[1] Cisco Carrier Routing System Compare Models - Cisco Systems, http://www.cisco.com/en/US/products/ps5763/prod_models_comparison.html

[2] Twitter hits 400 million tweets per day, mostly mobile, http://news.cnet.com/8301-1023_3-57448388-93/twitter-hits-400-million-tweets-per-day-mostly-mobile/

respond to variations in stream speed promptly. To meet this challenge requires parallel-distributed processing that does not introduce significant latency, and can handle sharp fluctuations in the rate of data arrival. Because it is difficult for programmers to balance all of these factors through custom design, it is crucial that a framework for real-time, parallel-distributed data-stream processing be developed.

2.4. Previous Data Processing Systems

As described above, data stream processing is a high-throughput, low-latency undertaking. To clarify the differences between data stream processing and traditional data processing, this section will review traditional data processing technologies such as DBMS (Database Management System) and distributed processing frameworks like MapReduce.

2.4.1. DBMS (Database Management Systems)

DBMS (Database Management Systems) have been, and continue to be, widely used for processing and storage of data. DBMSs have superb query optimization, and their transaction management mechanism prevent the kind of data inconsistencies that are unacceptable for many systems, including financial and commercial platforms.

To use a DBMS, data must first be stored in a table, the structure of which is strictly defined by a schema. Thereafter, users or software programs can issue queries to the DBMS using a Structured Query Language (SQL), as shown in Figure 2.1. The

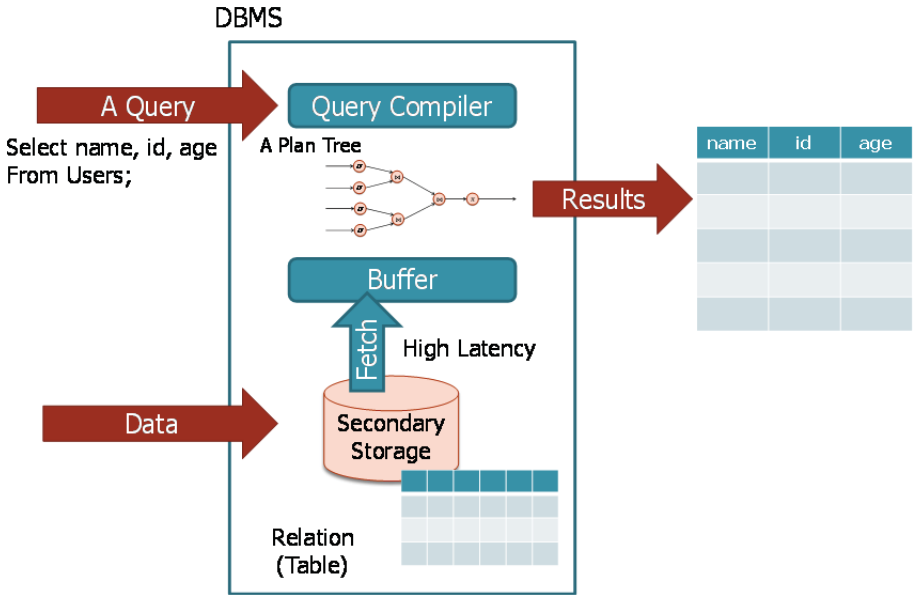

Figure 2.1 Typical Processing Procedure of a DBMS

advantage of this procedure is that clients can interactively process and modify data using a declarative language. Because the tables are well structured and DBMSs return results as a table, users are not bothered by unexpected data formats.

Unfortunately, secondary, table-based storage makes DBMSs unsuitable for data-stream processing, as it introduces significant latency to the system, and requires that SQL queries be constantly resubmitted to the DBMS to achieve real-time processing. Moreover, traditional query optimization is designed to minimize storage I/O, rather than to optimize data-stream processing.

2.4.2. Apache Hadoop (Google MapReduce)

Recent advancements in commodity computer hardware have made parallel-distributed computing available to everyone. Nevertheless, most developers want to avoid implementing concurrency controls and network communication procedures, as these have proven difficult to program in the past. These factors have led to the development of parallel-distributed computation frameworks such as MapReduce [1] and Dryad [20,21].

Apache Hadoop[1], an open source implementation of Google MapReduce [1], is now widely used for processing big data. It realizes distributed computing for large data without requiring user implementation of network communications. In the MapReduce programming context, developers are tasked with identifying and defining the data parallelism of their applications, after which the framework can scale out data processing by simply distributing job to available computation nodes. The effectiveness of this kind of parallelism is a major reason why Hadoop enjoys broad success in processing big data. Now, it is used to process variety of big data. Graph mining library [22] is built on Hadoop, and is supported by a number of recent extensions, including the graph mining library [22] and Pig Latin [23] a high level query language.

Figure 2.2 shows the procedure involved in MapReduce. Programmers implement the map and reduce functions. Each map function accepts data and processes it, and then outputs key-value data. Reduce functions accept values with the same key and output results. MapReduce executes these functions in parallel, by splitting distributing the input data to map functions running on available computation nodes. The outputs of the map functions are sorted based on shared key values and sent to a computer in the shuffle phase, where they are gathered into a list. Finally, the reduce functions process the list to generate results. Although all of these processing steps are executed in a parallel-distributed manner, programmers only have to implement the map and reduce

[1] Welcome to Apache Hadoop!, http://hadoop.apache.org/

functions, and can ignore issues of network communication.

Although Hadoop is a powerful tool for big data processing, it must store its data using the Hadoop Distributed File System (HDFS). As such, even if storage access is reduced [24], Hadoop ultimately relies on disk-based systems for its batch-processing, making it unsuitable for data stream processing.

2.5. Data Stream Management Systems (DSMS)

As described above, traditional disk-based processing software is not well suited for processing data streams due to the inherent latency in its batch-processing mechanisms. Hence, a number of DSMSs (Data Stream Management Systems) have been designed specifically for processing data streams using what are called "continuous queries". Aurora [2,25,26], Borealis [3,27,28,29], STREAM [4,30], TelegraphCQ [5], NiagaraCQ [6] and Gigascope [7] are examples of such systems. Figure 2.3 provides an overview of data stream processing by a DSMS. Queries for DSMSs can be usually described using a SQL-like language [15], after which the DSMS converts the query to a plan tree consisting of relational algebra operators. It then executes the query as data arrives and generates results in real-time.

Traditional relational algebra operators are not appropriate for continuous data because they assume that all data persists in secondary storage. To overcome this limitation, a stream is divided into windows and relational operators are applied to the data within a given window.

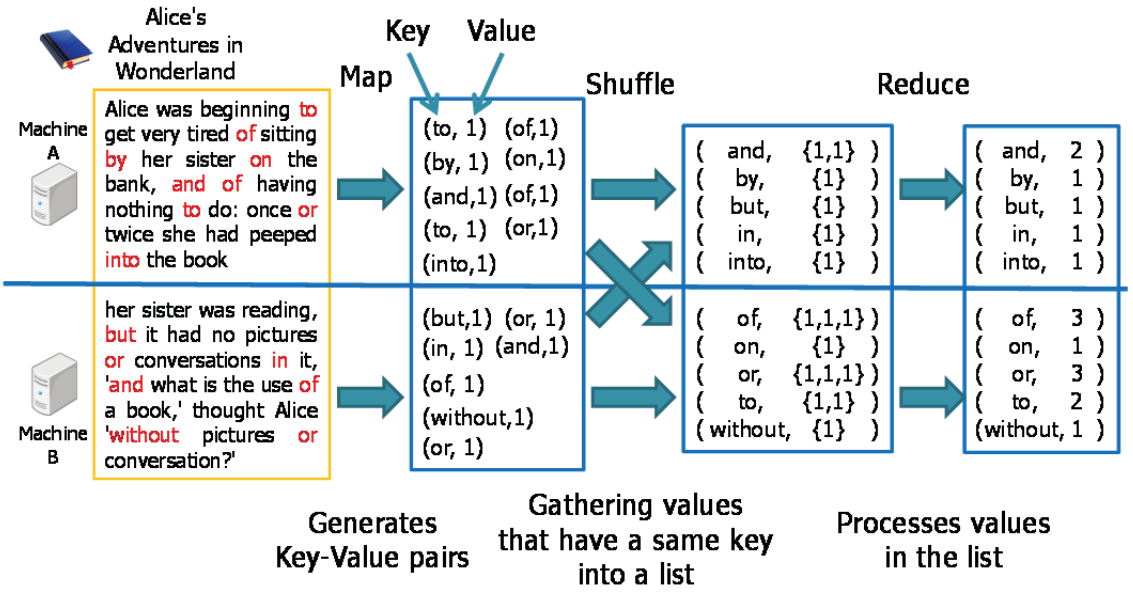

Figure 2.2 Word Count for a Novel "Alice's Adventures in Wonderland"

2.5.1. Continuous Queries

A continuous query is usually described by a SQL-like language [15] and compiled into a plan tree consisting of relational algebra operators that are traditionally used in DBMSs. Even so, the operator execution mechanism must be extended, since the computation of DSMSs differs from that of DBMSs in the following ways:

- Real-time Result Generation: In a DBMS, some relational operators such as join and sort have to process entire input data set(s). In contrast, a DSMS has to produce query results based on partial stream data, as no data stream can be considered finite. If a traditional operator implementation is used, operators will not be able to produce results until a stream ends. Thus, in DSMSs, blocking operators are converted to non-blocking operators through the use of a 'window' that defines a finite part of a stream.
- Different Operator Execution Mechanism: In a traditional DBSM, the pull model is a common implementation for executing a plan tree. In this model, an iterator-based pull execution mechanism is employed, such that, at the beginning of a query execution, the root operator of the plan tree is called, after which the children of the node are called recursively. The leaf nodes in a plan tree fetch data from a table that is typically persisted on secondary storage.

  In DSMSs, however, the arrival of data from a stream is unpredictable and leaf nodes may not yet have the necessary tuple, and a pull execution mechanism may block operator execution when there is no tuple. Thus, DSMSs must use a push

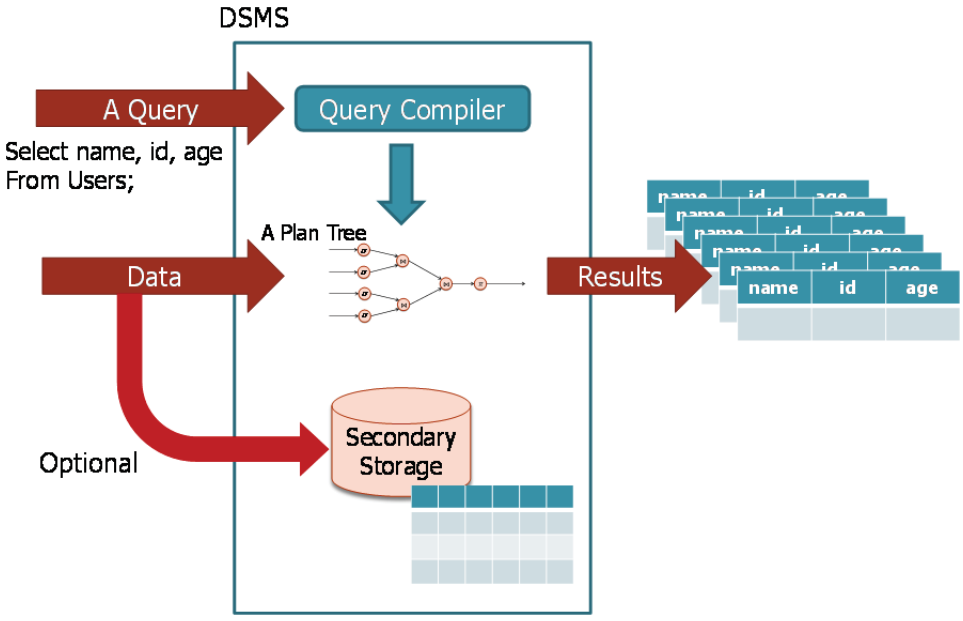

Figure 2.3 Processing Procedure of a DSMS

mechanism that calls operators when a tuple arrives. All operators are associated with an in-memory queue that buffers arriving tuples to be processed. The output of an operator is then sent to the queue of the next operator.

Figure 2.4 shows an example of a plan tree. Note that the query uses a time window specifying 24 hours, allowing blocking operators such as join to process tuples as they arrive. Here, the plan tree describes the query semantics representing the order in which operators are applied to tuples, but does not specify query execution strategy. This strategy is important because it can significantly affect query latency.

2.5.2. Operator Execution Strategies for Continuous Queries

DSMSs have to consider the execution strategy of each operator in a continuous query, as this strategy will have a significant impact on the query's latency and throughput. In this section, operator execution strategies are reviewed in order to establish the objectives for the remaining chapters in this thesis. Numerous studies have been devoted to operator execution strategies for continuous query [8,9,10,31,32,33,34,35,36,37].

Aurora [2] represents a continuous query using a 'box' that manages operators and an 'arrow' that manages connections between boxes. The scheduler used in Aurora [34] tries to minimize the number of I/O operations performed per tuple and to minimize

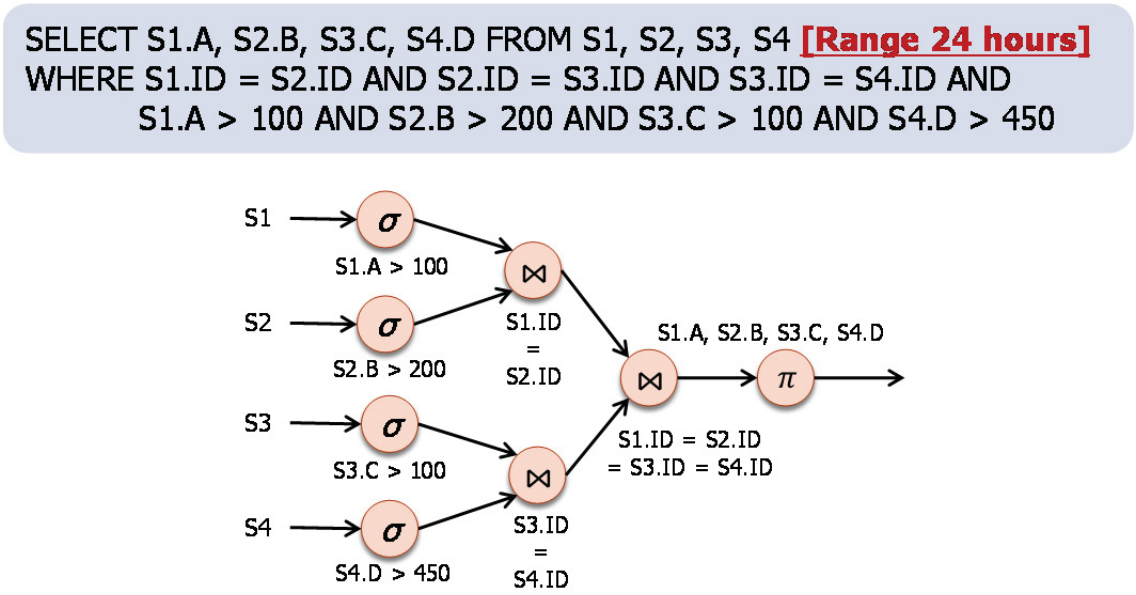

Figure 2.4 An Example of a Continuous Query and a Plan Tree

the number of box calls made per tuple.

STREAM [4] uses a scheduling algorithm called Chain [8,31], which executes operators in a sequence that is near-optimal to minimize memory consumption. As a side effect of this optimization, it can also reduce processing latency, but may also create high latency if the speed of the input stream suddenly increases, resulting in a buildup of unprocessed tuples. Moreover, Chain assumes the use of a single core CPU environment, and is thus unusable in modern multi-core processor environments.

TelegraphCQ [5] makes use of the Eddy [9] mechanism to reorder the operator execution sequence of a plan tree. With Eddy, each tuple has an execution history that records which operators the tuple has already visited. Based on this history and on the plan tree, an eddy transfers the tuple to the next appropriate operator. Unfortunately, doing so requires a call to the scheduler for every tuple, creating large overhead. This problem is addressed by Teddy [10], a refinement of Eddy, which processes a set of tuples at once, rather than one tuple at a time.

There are also several operator scheduling methods for distributed data-stream processing [11,12,13]. For example, COLA [11], a scheduler used in IBM System S [38,39], finds an operator assignment that can minimize the CPU resources used for network communication. While COLA helps increase the throughput of the processing, it does not consider processing latency. Similarly, SODA [12] uses mathematical optimization techniques to find operator assignments that do not violate the resource constraints of the computation nodes, but it too fails to consider processing latency in distributed environments. RASC [13] uses minimum cost flow to find operator assignments that minimize the number of tuples discarded under conditions of overload. Since RASC presupposes the possibility of lost data, it is focused on reducing computations, making it inappropriate for systems that are intolerant of data loss or require precision in query results.

None of the above provides a method for reducing the communication latency of a query executed in a parallel distributed environment. As mentioned previously, the arrival rate of a data stream may vary widely. To respond to such variations, a parallelized system must make difficult adjustments to the assignment of threads or computation nodes to operators. Because this adjustment involves transfer of the internal states of those operators, it typically creates new latency. The methods described above either fail to consider communication latency between threads and CPU cores, or assume this latency to be negligibly small. However, even a small amount of latency can seriously affect data stream applications, as described above, and must be addressed as a performance issue.

2.6. Other Systems

FPGAs can process data streams obviously in low latency and they are now used in algorithmic trading. However, due to the limitation of memory and the difficulty in programming the circuits, it is still arduous to put an algorithmic trading mechanism on an FPGA[1].

GPUs [40] can be used to process data in highly parallel. However, it requires transferring data from the main memory to the GPU memory or vice versa. In addition, GPUs use a large number of arithmetic logical units to achieve high throughput, which means they do not attempt to reduce the processing latency of each tuple.

Thus, software processing is still effective to process data streams. There are widely-used methods of research and systems to develop applications for parallel distributed computing. Grid projects such as Condor [41] attempt to leverage computation resources by batch processing. Obviously, they are not designed for data stream processing.

Parallel databases including DB2 [42] and Volcano [43] also should be considered. However, they are not designed for processing data streams in real-time, as is the case with other DBMSs.

MPI[2] is quite a usable programming interface for high performance computing. If a developer can make an effort to implement highly optimized applications, MPI may be the best way to achieve the highest performance. However, most developers want to avoid implementing concurrency control as these have proven to be difficult to program.

Recent frameworks such as Twitter Storm[3], Apache S4[4], and Esper[5] are designed to process data streams. However, they are not designed to realize low-latency processing in parallel-distributed computing. Low-latency processing techniques are closely correlated to framework design. Therefore, we must consider both a framework design and low latency processing techniques.

2.7. Summary and Milestones for the Remaining Chapters

This chapter described the common characteristics of data streams, several types of applications that use data streams, and a number of methods proposed for processing data streams. Overall, a modern data stream can be characterized as a large and

---

[1] FPGA & Hardware Accelerated Trading, Part Four - Challenges and Constraints, http://www.hftreview.com/pg/blog/mike/read/58189/fpga-hardware-accelerated-trading-part-four-challenges-and-constraints

[2] Open MPI: Open Source High Performance Computing, http://www.open-mpi.org/

[3] Storm, distributed and fault-tolerant realtime computation, http://storm-project.net/

[4] S4: Distributed Stream Computing Platform, http://incubator.apache.org/s4/

[5] Esper - Complex Event Processing, http://esper.codehaus.org/

unpredictable sequence of data that arrives from an outside data source. Traditional batch data processing systems, such as DBMS and MapReduce, were shown to be ill suited to data stream processing. Furthermore, recent advancements in commodity computer hardware have made clear the need for broadly accessible parallel-distributed computing frameworks, with concurrency controls and network communication procedures built in.

This chapter also reviewed several existing DSMSs and the conventional operator execution strategies they employ. It was shown that none of these systems provide methods for reducing the communication latency of a continuous query in a parallel-distributed environment. Low latency processing on parallel distributed environment is required because there are applications that are affected by latency on milliseconds or even on microseconds.

The balance of this thesis presents a framework for parallel-distributed data-stream processing that accomplishes the goal of reliably low latency even in response to large fluctuations in data stream speed.

- Requirements of a Framework for Parallel Distributed Data-Stream Processing
  - Analyzing data streams in real-time is required in a variety of situations.
  - To take full advantage of recent computer hardware developments in processing data streams, parallel-distributed computing is indispensable.
  - Developers want to be able to easily develop data stream applications without necessitating complicated implementations of network communication and concurrency controls.
- Requirements for Parallel Distributed Low Latency Data-Stream Processing
  - Several applications are affected by processing latencies that are now, in some case, on the order of microseconds.
  - Speed and content changes by data streams must be considered in order to achieve low-latency processing.

To meet with these requirements, Chapter 3 describes QueueLinker, a framework for parallel distributed data-stream processing. QueueLinker will enable developers to build data stream applications by implementing JAVA modules that utilize a producer–consumer model and by specifying a logical directed graph representing the data-flow connection between modules. Using QueueLinker, programmers will not need to implement multi-threaded concurrency and network communication procedures. Chapter 3 also provides an overview of software architecture of QueueLinker, including

its use of “thread units” for low-latency execution modules and its general model for parallel processing. QueueLinker also provides an application programming interface (API) to realize data parallel computation that is adopted by MapReduce. By using this data parallel model, developers can easily scale-up applications.

Chapter 4 describes a proposed method for assigning thread units of QueueLinker to the operators of a query so as to reduce the latency created by inter-thread communication. The chapter also proposes a low-overhead, dynamic operator reallocation technique that allows QueueLinker to continue processing during operator reallocation.

Chapter 5 describes an operator backup method for distributed processing. The method reduces network communication latency, maintains high availability, and responds promptly to changes in the speed of the input data stream. By executing a query on a set of different operator deployments, and generating results from the fastest tuples output by these deployments.

Chapter 6 describes a proposed Web crawler consisting of fine-grained QueueLinker modules, all of which can be executed using the data parallel model. It will be shown that this Web crawler achieves improved load balancing and memory utilization between computers as compared to traditional site-based Web crawlers.

Chapter 3. QueueLinker

The proliferation of mobile devices and sensors has resulted in a large numbers of new data streams, the processing of which requires robust parallel-distributed computing. Fortunately, advancements in commodity computer hardware have made this kind of computing available to a broad user base. As CPU manufacturers have begun to favor increases in the number of CPU cores over increases in CPU core clock speed, the costs associated with parallel computing hardware have plummeted. To take full advantage of these trends in the processing of data streams, developers must now become familiar with the essentials of parallel-distributed computing. However, most developers still want to avoid implementing concurrency controls and network communication procedures, as these have traditionally proven difficult to program.

This chapter describes a new framework, QueueLinker, developed especially for real-time data processing. QueueLinker enables programmers to build data stream processing applications by implementing application modules that use a producer–consumer model, and specifying a logical directed graph representing the data-flow connections between these modules. Each module is automatically instantiated and executed in parallel according to the logical directed graph. The data generated by a module is automatically serialized and transferred to other modules across the network, relieving the programmer of complex multi-threading and communication implementations. In addition, data parallel model of QueueLinker helps the developers to realize parallel processing without concurrency control.

3.1. Producer–Consumer Model

In QueueLinker, a module is a software processing unit implemented according to the producer–consumer model commonly described in design patterns for multi-threaded programming. Under the producer–consumer model, a module processes input item(s) and generates a result. Modules communicate with each other using queues. A module sending an item to another module puts the item into the input queue of the destination module. Because modules are not meant to share their internal states, an arbitrary number of threads can be assigned to each, and they can be executed in parallel on multiple computers to achieve greater speed.

3.2. Module Implementation Interfaces

QueueLinker presents a Java API for constructing producer–consumer modules of four major types: push modules, pull modules, source modules and sink modules. A push module is designed for non-blocking operations such as filtering and arithmetic operations. A pull module is designed for blocking operations such as file I/O and receiving data from external data sources. A source module is designed for providing data to other modules from external data sources. A sink module is designed for storing data in secondary storage or visualizing application window to a user.

3.2.1. Push Module Implementation

Figure 3.1 shows pseudo code for a push module. In this example, the module has two input queues and an output queue. An item transferred from another module is passed through the variable 'item'. The queue that the item was put into is identified by the variable 'queueId'. The module processes an input item and then returns a string as a result. Processing can differ depending on the input queue the item was put into. If the module returns null, QueueLinker sends no data to the next module. This mechanism helps us implement modules for filtering of data. A relational operator for continuous queries can be implemented using the interface for push modules.

Because QueueLinker uses a push thread unit (described in 3.5.1) to execute multiple push modules, a push module cannot use an infinite loop or perform blocking operations like file I/O. If a push module does not return, other push modules will not be executed.

3.2.2. Pull Module Implementation

Figure 3.2 shows pseudo code for a pull module. In this example, the module pulls an item from an input queue and then outputs a string toward an output queue. The system assigns a dedicated thread to each instance of the pull module. Thus, a pull module implementation can make use of an infinite loop, which is useful for implementing blocking operations, such as receiving data from external data sources. To do this, the blocking operation is simply written inside an infinite loop in a pull module.

3.2.3. Source Module Implementation

Figure 3.3 shows pseudo code for a source module. A source module has no input queue and has only one output queue. A data source is typically used to provide data to other modules from external data sources. For example, a data source may leverage the Twitter API to feed tweets to the system. QueueLinker can manage multiple data

sources and automatically duplicate the data if multiple modules need the data from a single data source.

3.2.4. Sink Module Implementation

QueueLinker also provides interfaces for data sink. A sink module has one input queue and no output queue, and is typically used to store data in secondary storage and present that data in visualize window to a user. Figure 3.4 shows pseudo code for a sink module.

```
public class ExamplePushModule extends PushModule<String, String>
{
    @Override
    public String execute(String item, int queueId)
    {
        if (queueId == 0)
            return "Waseda";
        else if (queueId == 1)
            return "University";
        return null;
    }
}
```

Figure 3.1 A Pseudo Code of a Push Module

```
public class ExamplePullModule extends PullModule<String, String> {

@Override
public void execute(InputStaff<String>  inputStaff,
                    OutputStaff<String> outputStaff,
                    QueueLinkerService  service)
{
    InputQueue<String>  input  = inputStaff.getDefaultInputQueue();
    OutputQueue<String> output = outputStaff.getDefaultOutputQueue();

    while (!service.stopRequested()) {
        try {
            String item = input.take();

            /* Something to do and generate newStr */

            output.put(newStr);
        } catch (InterruptedException e) {}
    }
}
```

Figure 3.2 A Pseudo Code of a Pull Module

3.3. Data Parallel Execution

This section describes how QueueLinker executes modules in a parallel-distributed way. Figure 3.5 shows an example consisting of three execution patterns for two modules, “Tweets Parse” and “Word Count”. A rectangle with a dashed line represents a computer and each rectangle represents a thread executing a module instance. The “Tweets Parse” module pares a tweet and outputs the extracted words from the tweet. The “Word Count” module counts the number of appearance of each word.

In pattern (1) of the figure, only one instance is created for each module and a thread is assigned to the module. Thus, “Tweets Parse” and “Word Count” run on different threads.

In the general producer–consumer model, an instance is executed by multiple threads. Thus, modules must be implemented for thread-safety by using concurrency control to avoid inconsistency. Concurrency control can be an especially difficult task, and even when it performed properly, the possibility of lock contention will increase with the number of threads executing the instance. To solve this problem, QueueLinker

```
public class ExampleSourceModule extends SourceModule<String>
{
    @Override
    public void execute(OutputStaff<String> staff, QueueLinkerService service)
    {
        OutputQueue<String> output = staff.getDefaultOutputQueue();

        while (!service.stopRequested()) {
            try {
                output.put("Output Something");
            } catch (InterruptedException e) {}
        }
    }
}
```

Figure 3.3 A Pseudo Code of a Source Module

```
public class ExampleSinkModule extends SinkModule<String> {
    @Override
    public void execute(String input, int queueId) {
        /* Something to do */
    }
}
```

Figure 3.4 A Pseudo Code of a Sink Module

uses data parallel execution with a hash partitioning technique to ensure that each instance of a module is executed by only one dedicated thread, allowing the programmer to implement modules without concurrency control.

For example, in pattern (2) of the figure, a word is transferred to one of the two "Word Count" instances depending on the hash value. QueueLinker automatically transfers words that have the same hash value to the same instance. This mechanism eliminates the need for concurrency control of the module because each instance is executed by only one thread. Note that developers must specify modules to be executed by this mechanism when they define an application; QueueLinker cannot infer automatically which modules can be parallelized in this way.

Modules executed in this way do not share their internal states with other modules and only communicate with other modules via queues. Thus, they can be run on any computer. Pattern (3) in the figure shows an example of parallel-distributed execution on three computers. QueueLinker automatically transfers items between modules, developers do not need to implement network communication procedures.

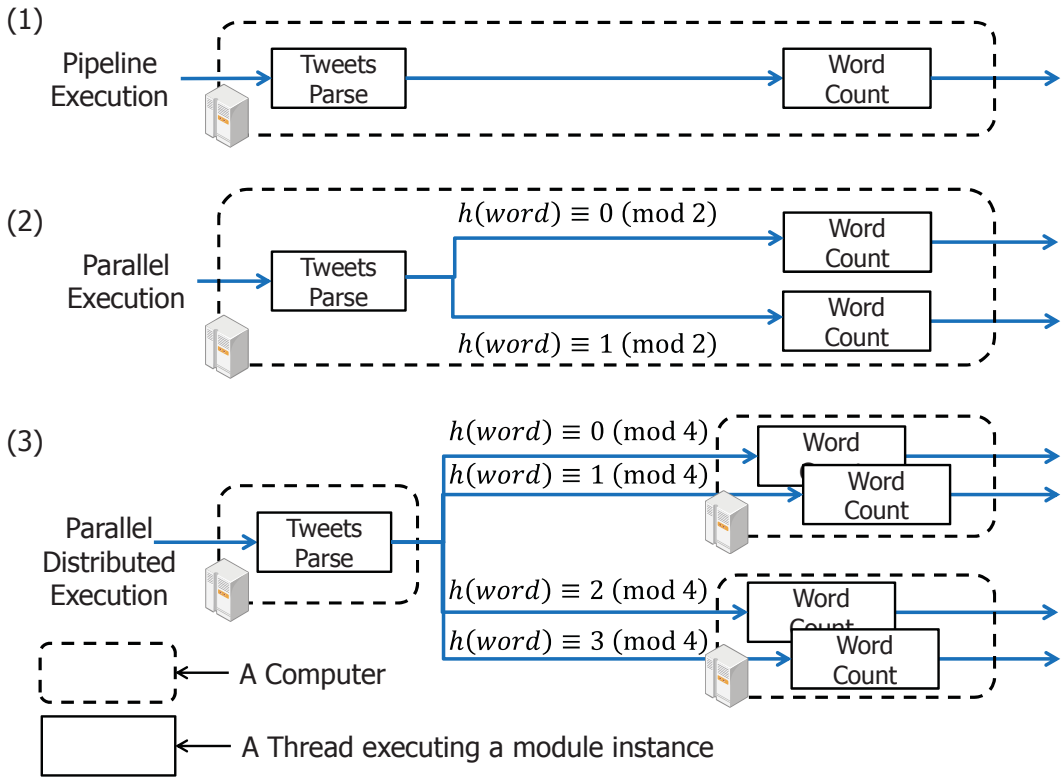

Figure 3.5 Parallel Distributed Execution Model of QueueLinker

3.4. Application Definition Using a Logical Directed Graph

A QueueLinker user can build an application by specifying connections between modules. The directed graph representing these connections is called a 'logical directed graph'. Figure 3.6 shows a logical directed graph (V, E) for the proposed Web crawler described in Chapter 6. Each node $v \in V$ is indicated by a rectangle and represents a module; each edge $e \in E$ is indicated by a line and represents a connection between two modules. A node in the graph is called a 'logical vertex' and an edge is called a 'logical edge'.

Users can specify the parallel execution mode of modules as well as connection settings. Figure 3.7 provides pseudo code describing a logical directed graph for the application shown in Figure 3.8. In the code, the execution mode of the 'Word Count' module is set to data parallel mode by hash partitioning the three instances. The function of the module is to count the number of appearances of each word in tweets. In this case, QueueLinker instantiates three instances on different threads and transfers each string output from the 'Morph Analyzer' module to the correct instance based on the hash value of the string. Note that the code does not specify data parallel mode for the 'Morph Analyzer' module. In this case, QueueLinker transfers each tweet to one of the two instances in round-robin fashion.

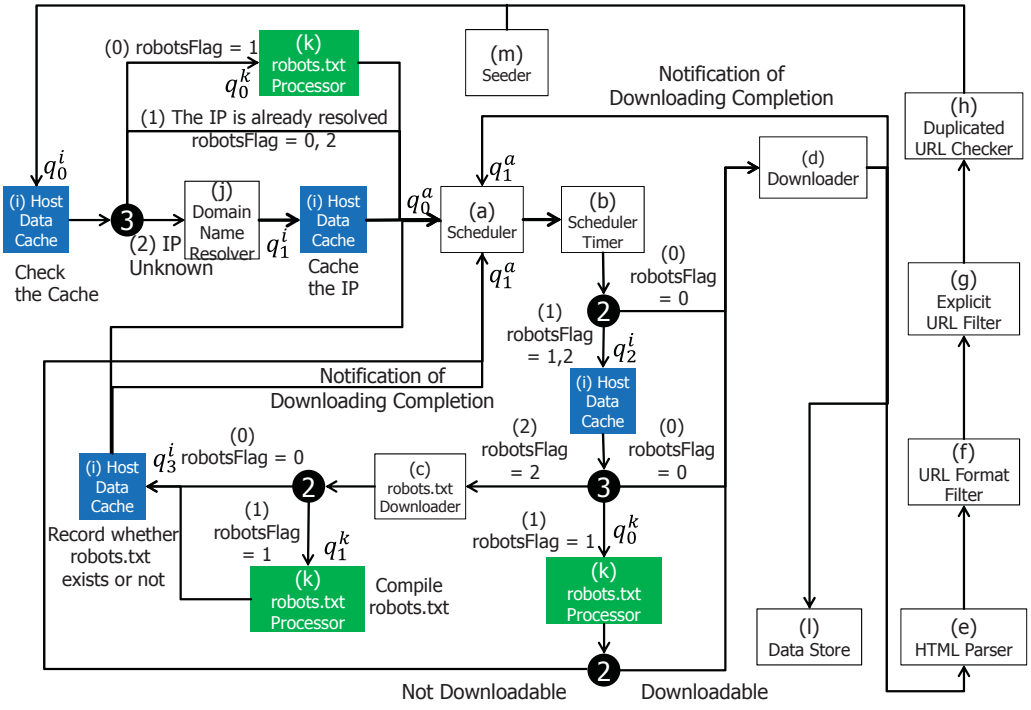

Figure 3.6 An Example of a Logical Graph

As described above, when QueueLinker accepts module implementations and a logical directed graph, it realizes parallel distributed execution by automatically instantiating the modules on available computers and transferring data items between the modules. The programmer does not need to know whether transfers between modules require network communication.

3.4.1. Switcher and Virtual Module

QueueLinker provides a mechanism called a 'switcher' for choosing a destination module based on the result data a module produces. It also offers a mechanism called a 'virtual module' that allows modules to be reused in different data flows.

The logical directed graph in Figure 3.9 includes a switcher, indicated by a circle containing the number of destination modules. Figure 3.10 provides pseudo code for a switcher. Note that the switcher returns an integer specifying the destination module. QueueLinker will send an item to a module based on this number. For example, in Figure 3.9, an output of module A is sent to B if the switcher returns 0, or to C if the

```
LogicalGraph graph          = new LogicalGraph();
LogicalVertex twitter       = graph.addLogicalVertex(TwitterDataSource.class);
LogicalVertex morphAnalyzer = graph.addLogicalVertex(MorphAnalyzer.class, 2);
LogicalVertex wordCount     = graph.addLogicalVertex(WordCount.class, 3, PMode.Hash);
LogicalVertex ui            = graph.addLogicalVertex(UI.class);

graph.addLogicalEdge(twitter,       morphAnalyzer);
graph.addLogicalEdge(morphAnalyzer, wordCount);
graph.addLogicalEdge(wordCount,     ui);

QueueLinkerClient client = QueueLinkerClientFactory.getClient();
QueueLinkerJob job = new QueueLinkerJob(graph);
JobHandle handle = client.startJob(job);
```

Figure 3.7 Pseudo Code Describing an Application

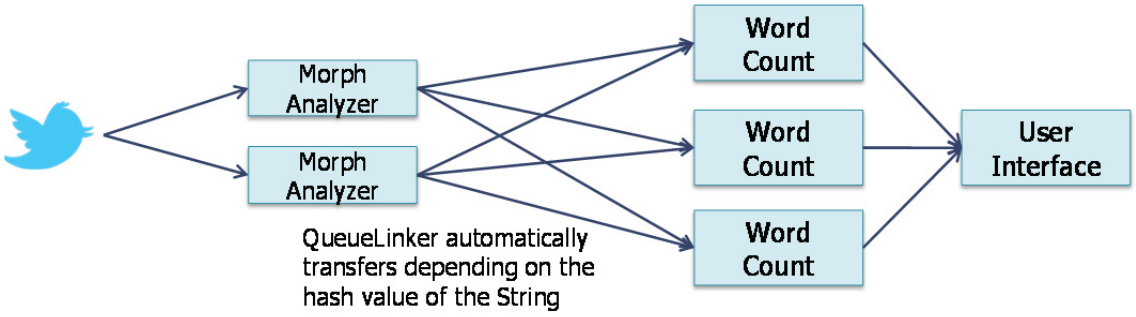

Figure 3.8 Logical Directed Graph Described by the Code in Figure 3.7.

switcher returns 1. Thus, the switcher provides control over data routing independent of module implementation.

The logical directed graph also includes a virtual module, indicated by a rectangle with a dashed line. In the logical directed graph, outputs of module B are sent to virtual module A. The virtual module is executed using the same instance of module A shown at the far left of the figure, but outputs of the virtual module are sent to module D. Thus, the virtual module makes it possible to reuse a module in a different data flow. For example, the Web crawler described in Chapter 6 uses multiple switchers and virtual modules. The logical directed graph of the crawler is shown in Figure 3.6. Despite being simple mechanisms, the switcher and virtual module are indispensable for describing a complex logical directed graph efficiently.

3.5. Software Architecture of QueueLinker

This section provides an overview of the software architecture of QueueLinker. QueueLinker uses several software mechanisms to execute modules and control execution.

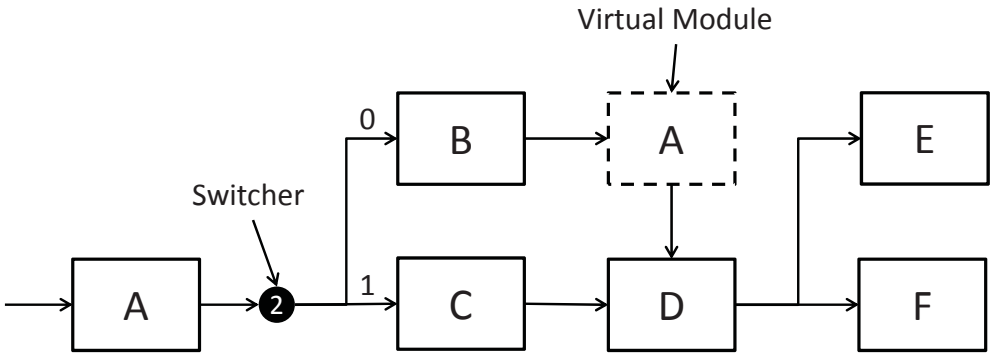

Figure 3.9 A Virtual Module and a Switcher

```
public class SwitcherExample extends FlowSwitcherModule<String>
{
  @Override
  public int execute(String input) {
    if (input.length() % 2 == 0)
      return 0;
    else
      return 1;
  }
}
```

Figure 3.10 A Pseudo Code of a Switcher

3.5.1. Push Thread Unit

A push thread unit is designed to execute multiple push modules and sink modules as illustrated in Figure 3.11. The push thread unit has a "thread local scheduler" and a "thread local router" for handling multiple modules. An item to be processed by a module in a thread unit is put into the "thread input queue". The thread unit fetches the item from the queue, and the thread local router, according to the logical directed graph, determines which module will process the item. It then sends the item to the input queue of that module. The thread local scheduler then chooses an executable module and executes it. When an item is produced from this module, the thread local router determines the destination of that item. If the destination module runs in data parallel mode, the local router calculates the hash value of the output item and transfers it to the appropriate thread unit. If the destination module is running in a thread unit on a remote computer, QueueLinker transfers the item to that computer, using a thread unit dedicated for network communication.

A thread unit can 'busy wait' for items to arrive in the thread input queue, and the CPU core that a thread unit runs on can be controlled using system calls like `sched_setaffinity` on Linux. 'Busy wait' is important for achieving low latency execution of continuous queries (described in Chapter 4). In addition, a push thread unit has a mechanism for collecting statistics on operator execution, such as the number of input/output items to/from, and the total CPU time consumed by, each operator. Note

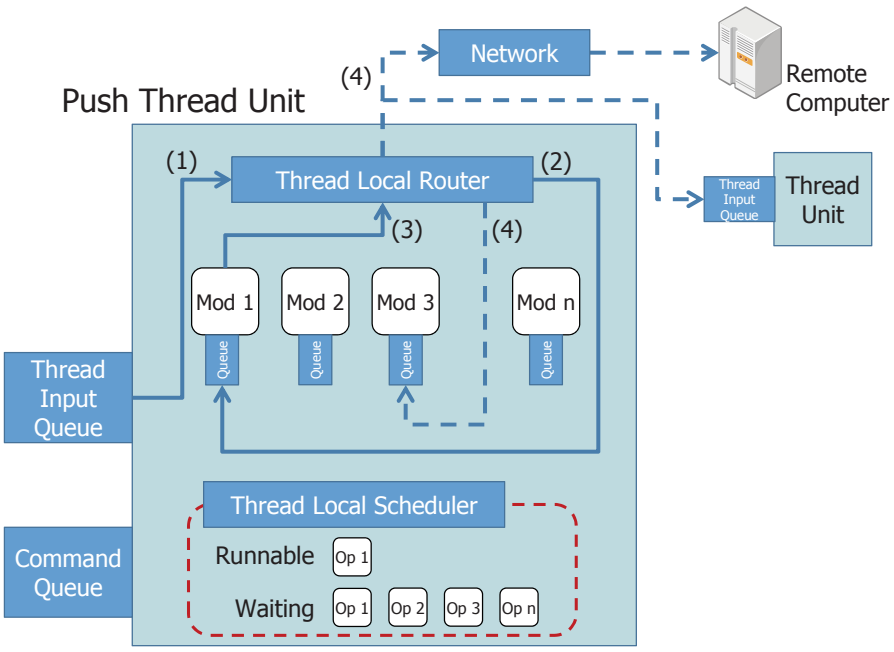

Figure 3.11 A Push Thread Unit

that the scheduler and the router in a push thread unit are only used by that thread unit, and thus do not require any concurrency control.

A number of optimizations should be considered for the thread local scheduler, since the strategy of the scheduler will affect the processing latency, throughput and memory consumption of applications. QueueLinker normally uses a FIFO scheduler, but other algorithms, such as Chain [8,31], can be substituted.

3.5.2. Pull Thread Unit

A pull thread unit executes only one pull or source module. It must execute that pull module on a single thread, since a pull thread unit may contain an infinite loop (as described in 3.2.2) and may therefore refuse to yield to other modules. Like the push thread unit, a pull thread unit has a thread local router to determine the transfer route of each result, but unlike the push thread unit, it does not have a local scheduler, since it does not execute multiple modules. Other mechanisms of the pull thread unit are nearly identical to those of the push module unit, and are therefore omitted.

3.5.3. Master Server and Worker Server

QueueLinker uses a master server to manage all computation nodes, or 'worker severs'. It accepts job requests from clients and sends commands to the worker servers, which in turn manage thread units. QueueLinker uses ZooKeeper[1] to communicate among master server, worker servers, and clients.

Figure 3.12 shows a worker server and its constituent thread units. A worker server has a worker local scheduler that collects operator statistics from thread units, such as the number of input/output items to/from, and the total CPU time consumed by, each operator. The proposed method in Chapter 4 is implemented by using this mechanism.

3.6. Continuous Query and QueueLinker

The relational operators of a continuous query can be implemented as a push module, and a query plan can be described as a logical directed graph. In this way, QueueLinker allows us to execute a continuous query in a parallel-distributed environment.

Recall that many data stream processing applications require extremely low latency. By assigning a dedicated push thread unit to each operator, QueueLinker

[1] Apache ZooKeeper – Home, http://zookeeper.apache.org/

achieves high parallelism. However, in this case, tuple transfer between the thread units adds latency to the query. By contrast, if a push thread unit executes too many operators, the thread unit may not be able to process all tuples and computation over load happens. The strategy of assigning operators to push thread units affects the latency of the query.

A query also can be executed in a distributed environment using the data parallel model of QueueLinker. In the distributed query execution, transferring tuples between computers incurs latency by network communication.

Thus, Chapters 4 and 5 propose a method for low latency execution of continuous queries using QueueLinker. Detailed in Chapter 4, this method attempts to reduce the frequency of communication between thread units, but it performs an operator reallocation when computational load changes. The method also uses a dynamic operator reallocation technique that does not require QueueLinker to stop stream processing during reallocation. In Chapter 5, a proposed backup method is shown to reduce latency by executing secondary processing on a set of alternative operator deployments, and generating query results from the tuples outputted fastest, either by primary or secondary deployments.

3.7. Summary

This chapter described the proposed QueueLinker framework. QueueLinker adopts a producer–consumer programming model, and accepts a Java module implementation along with a logical directed graph. Based on these, it automatically executes each module in the graph in parallel-distributed manner. Data generated by a module is automatically serialized and transferred to other modules across the

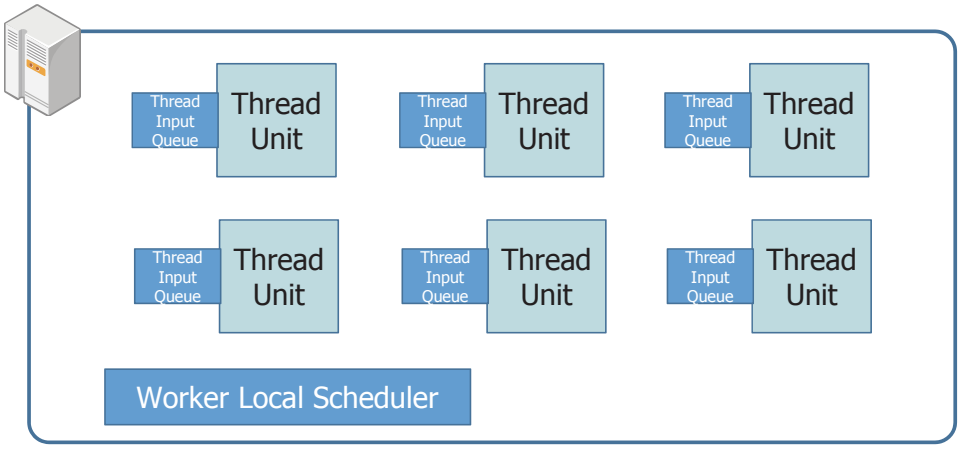

Figure 3.12 Thread Units on a Worker Server

computational network, even if they are running on other computers. Programmers do not need to write multi-threaded programs or network communication procedures.

The following chapters describe methods by which QueueLinker can be used to execute continuous queries efficiently. Chapter 4 presents a method for minimizing the processing latency of continuous queries. Chapter 5 presents a backup method that achieves low latency processing and handles computation node failures. Chapter 6 presents a proposed Web crawler consisting of fine-grained QueueLinker modules.

Chapter 4. Low-Latency Continuous Query Processing on Multi-core CPU Environments

The concept of a “continuous query” [14] has been studied in the field of database science since the early 2000s. A relational continuous query is usually described by an SQL-like language and compiled to a plan tree consisting of relational algebra operators. As each tuple arrives, it is pushed into a leaf of the plan tree and the result of the query is generated.

As a relational operator can be implemented as a QueueLinker push module, and a plan tree can be described as a logical directed graph of QueueLinker, the proposed framework can execute a plan tree in parallel on a multi-core CPU. The parallel computation power of multi-core CPUs allows us to process data streams at high speed. Reducing latency is important for data stream applications, with several, such as algorithmic trading and network packet monitoring, requiring very-low-latency stream processing. For such applications, we must consider the assignment of relational operators to QueueLinker push thread units, because this affects the query latency.

This chapter describes a proposed method for low-latency execution of continuous queries on multi-core environments. First, this chapter discusses the cause of processing latency in terms of the CPU architecture and thread context switch. Reducing the number of CPU cores used for processing and controlling the communication route between CPU cores are important factors in decreasing the latency derived from inter-core communication. This latency, along with the thread waiting overhead, becomes large if a thread executes only one operator. However, if a thread executes multiple operators, the throughput performance becomes limited.

The proposed method gives a definition of the latency of data stream processing and a dynamic programming algorithm to determine the optimal CPU core assignment problem for relational operators. In addition, the proposed method includes a dynamic operator reallocation technique that does not require QueueLinker to stop stream processing during the reallocation.

4.1. Background

Examples of applications that are highly affected by latency are those involved in algorithmic trading. The Tokyo Stock Exchange, Inc. says their “arrownet” system, an

access network to the trading system, supplies data within 32 μs one-way[1]. Moreover, Wall Street algorithms will reportedly be uncompetitive if they experience a 5μs processing delay.[2] Reducing latency provides advantages over other trading competitors, because if an order reaches the central trading server first, it will be processed prior to other companies' orders. Nowadays, algorithmic trading is competitive to the order of microseconds.

For such latency-aware applications, the system has to consider a delay of several microseconds when it assigns computational resources to the operators. In addition, multi-core CPUs are required to process high-speed data streams. However, existing research on data streams does not consider the processing latency in sufficient detail. Chain assumes a single-core CPU environment and cannot be applied to a parallel processing environment. In the multi-core CPU environment, the latency of inter-core communication and the overhead of thread contexts must be considered.

4.2. Causes of Processing Latency

This section describes the causes of processing latency from the viewpoint of inter-core communication and thread-waiting techniques for tuple arrival. This section gives an experimental evaluation to validate the discussion.

4.2.1. Latency Derived from Inter-Core Latency

The ccNUMA (cache-coherent Non-Uniform Memory Access) architecture is now common in multi-processer environments. Figure 4.1 shows the architecture diagram of the Intel Xeon 7500 Series[3]. A CPU has 8 cores, and HT (Hyper-Threading) provides 16 logical cores. Each CPU core has a 1st and a 2nd level cache. The CPU cores in a CPU package share a 3rd level cache with a capacity of 24 MB. Different CPUs are connected by QPI (QuickPath Interconnect). Therefore, the latency of communication within the 3rd level cache is less than that required for QPI communication.

Unless there is a risk of an overload, the number of CPU cores used for processing should be reduced to decrease the latency of inter-core communication. For example, Figure 4.2 shows an example in which a thread executes only one operator. If each thread can run on a dedicated CPU core, this execution receives the maximum benefit from pipeline processing and can process high-speed data streams. However, this

[1] TSE: arrownet, http://www.tse.or.jp/english/system/networkservices/arrownet.html
[2] K. Slavin, "How algorithms shape our world," http://www.ted.com/talks/lang/en/kevin_slavin_how_algorithms_shape_our_world.html
[3] Intel Xeon Processor 7500 Series, Datasheet, Volume 2: http://www.intel.com/Assets/PDF/datasheet/323341.pdf

approach requires a large amount of inter-core communication, which increases the latency. On the other hand, Figure 4.3 shows an example in which a thread executes multiple operators. This execution can reduce the amount of inter-core communication, decreasing the latency. However, if the input stream rate becomes large and an overload occurs, the thread cannot keep up with the newly arriving tuples. Having multiple tuples waiting to be processed causes considerable latency. This is a trade-off, so the proposed method attempts to minimize the communication latency, except in the case of an overload.

4.2.2. Latency Derived from Thread Waiting

In addition to inter-core communication, the thread-waiting method dramatically affects the latency. As an example, consider 100,000 tuples arriving at equally spaced intervals. In this case, the tuples arrive every 10 μs. A 2 GHz CPU runs 20,000 clock cycles during the 10 μs interval. If each tuple is processed before the next tuple arrives, the processing thread has to wait for the arrival of the next tuple.

There are two methods of waiting for the arrival of tuples. The first puts the thread to sleep, and the second uses a "busy wait" with an infinite loop. In the sleeping method, the thread requires a system call in order to awake when a new tuple arrives. The benefit of this method is that it saves CPU time. However, the overhead of waking the thread becomes large, because the thread needs the help of the operating system to change from the sleep state.

On the other hand, the busy wait method does not rely on the operating system,

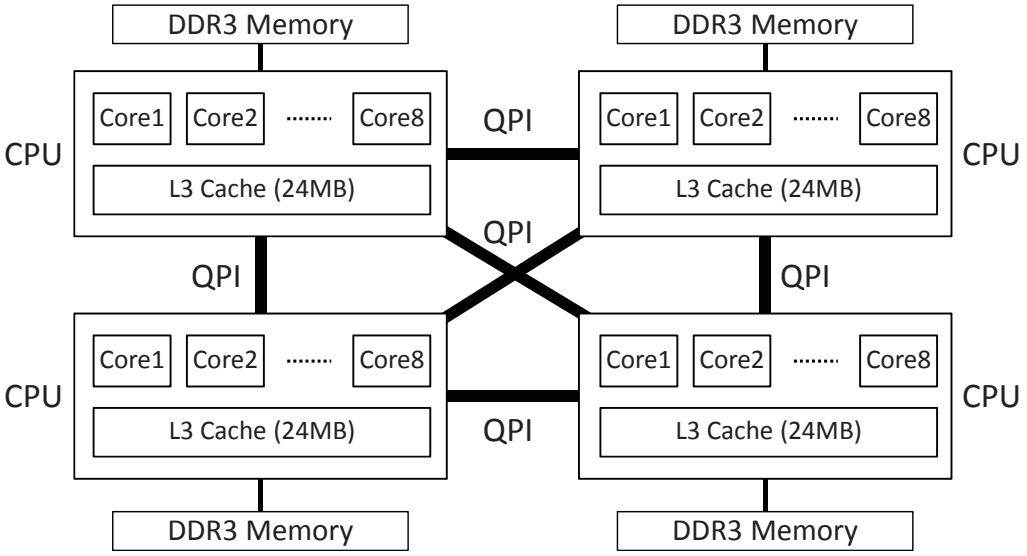

Figure 4.1 Architecture Diagram of Intel Xeon 7500 Series

and uses an infinite loop to await a new tuple. This method can reduce the latency, because there is no thread waking overhead[1]. However, CPU time and power are both consumed. The system administrator must determine the waiting method depending on the system requirements. The cache coherence protocol means that the busy wait does not consume the CPU bus while the read memory region is not being modified. Therefore, the busy wait does not affect other procedures running on other CPU cores if it is carefully implemented to only read from memory.

4.2.3. Experimental Validation

An experiment was carried out on a real computer, as described in Table 4.1, in order to validate the discussion in this section. In this experiment, 12 operators are connected to one another. Each operator executes 10 simple multiplication and addition calculations. Thus, the operators execute a total of 120 calculations for each tuple. The operators are assigned to thread units as shown in Figure 4.2 or Figure 4.3. Thus, the number of operators assigned to a thread unit was changed during the experiment. The latency of processing is the interval between the Data Source generating a tuple and the tuple arriving at the Sink. The Data Source generated 100,000 tuples per second, and sent a tuple to Thread Unit 1 every 10 μs. The tuples pass through the operators and then arrive at the Sink. Considering the CPU speed and the stream speed, each thread

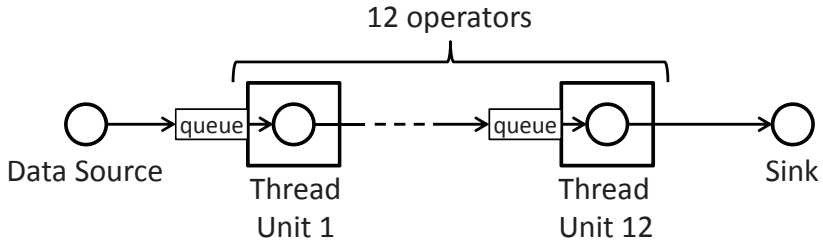

Figure 4.2 Thread Unit Executing Only One Operator

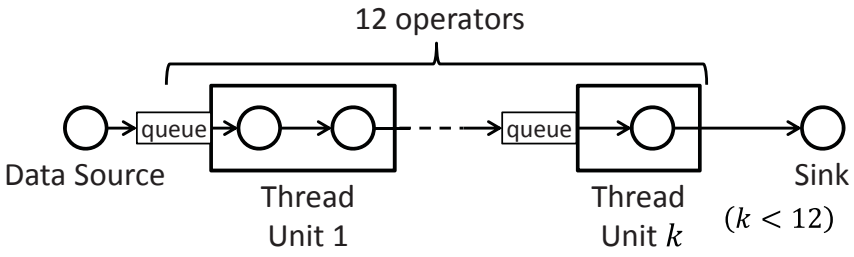

Figure 4.3 Thread Unit Executing Multiple Operators

[1] Note that even if the system uses the busy wait method, it can be interrupted and other applications may run on the CPU. This situation is not considered in this chapter.

unit is able to process a tuple before the next one arrives. Hence, there is a large amount of thread waiting.

Each thread unit uses LinkedBlockingQueue in java.util.concurrent as a FIFO queue for the inter-thread communication. The poll() method of LinkedBlockingQueue is used to perform a busy loop and the take() method is used to perform thread sleeping. The poll() method first checks the queue size, returning "null" if the queue is empty. Thus, it only causes a read operation, and this is important for implementing the busy loop without generating side-effects for the other CPU cores. Unfortunately, the operating system cannot assign appropriate CPU cores to the thread units. This is because the operating system does not have information about the connection between operators. To fix the CPU core on which each thread unit runs, the experiment calls sched_setaffinity, a Linux system call, through JNI (Java Native Interface).

Figure 4.4 shows the experimental evaluation. "Non-Fixed" means that CPU core assignment is completely controlled by the operating system. "Fixed" means that all threads, including the Data Source and Sink, run on the same CPU, and the CPU core on which each thread runs is controlled using sched_setaffinity. When fewer than six thread units are used in the experiment, each core runs only one thread. When twelve thread units are used in the experiment, each core runs two thread units with the help of HT. Note that the Data Source and Sink run on dedicated CPU cores throughout all of the experiments.

From Figure 4.4, we can see that the busy wait method and fixing CPU cores is the best way to stably reduce the latency. In the "Non Fixed" case with 12 thread units, the input tuples could not be processed, leading to an overflow. This result shows that the CPU cores on which thread units run should be fixed, even if the system cannot apply the busy wait method because of its power and resource consumption. Additionally, we can see that the latency increased with the number of threads used for processing. Therefore, for the same amount of processing, it is better to use the minimum number of threads in order to reduce the total processing latency. However, if

Table 4.1 Experimental Environment

CPU	4 Intel Xeon L7555 Processors (Total 32 Physical Cores and 64 Logical Cores)
Memory	512 GB
OS	CeontOS 5.5 (Linux Kernel: 2.6.18-238.19.1.el5)
Runtime	Java 1.6.0

the system uses too few threads, the data stream cannot be processed quickly enough. This means that tuples have to wait to be processed, which increases the latency dramatically and causes processing delays. Considering this trade-off, the proposed method attempts to minimize the number of CPU cores used, except when a processing overload occurs.

4.3. Latency Definition for a Continuous Query and Average Latency Minimization Problem

This section defines the latency of a continuous query and states the average latency minimization problem. This section only discusses an assignment method for CPU cores on a single CPU and communication inside the CPU. An assignment method for multiple CPUs, which causes communication over QPI, is beyond the scope of this thesis. In addition, the proposed method assumes the number of CPU cores in a CPU is sufficient that all operators can be assigned without a processing delay. The proposed minimization problem is an extended version of the "Average Path Length Minimization" described by Diwan et al. [44]. The paper considers how to minimize the number of disk I/Os when storing a tree structure on a secondary storage, which is different from the purpose of this thesis.

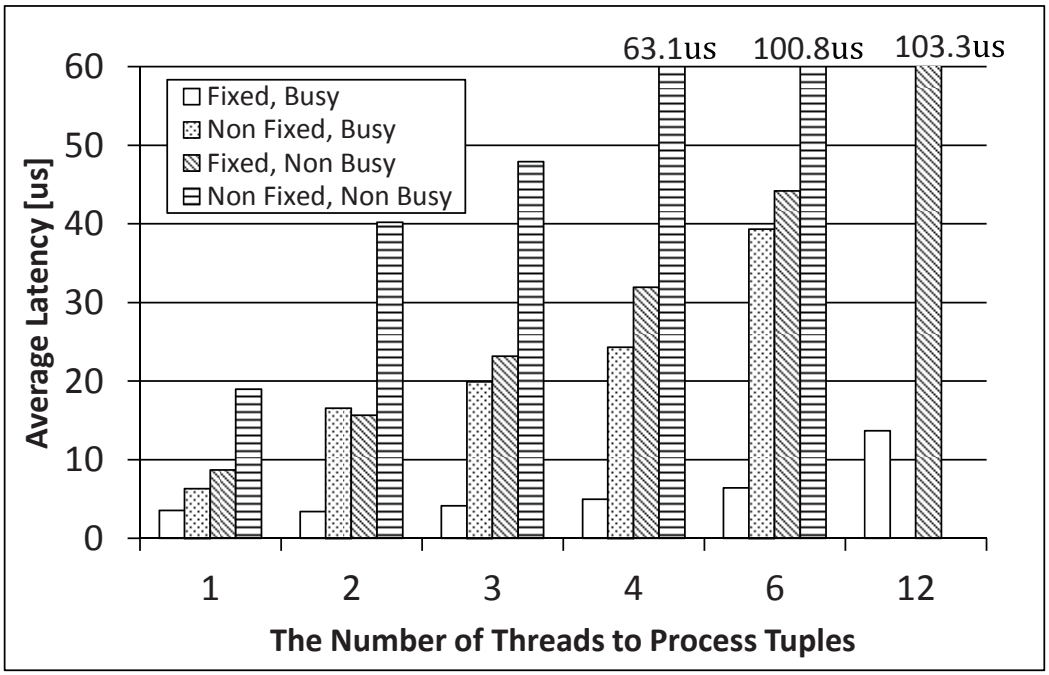

Figure 4.4 Experimental Validation of Processing Latency

4.3.1. Latency Definition and Problem Definition

A relational continuous query is represented by a plan tree $G = (V, E)$. A node $v \in V$ represents a relational operator, and an edge $e \in E$ represents a connection between two operators. Each operator has at least one input edge and only one output edge. An operator v processes a tuple using processor resource c_v, and then outputs a tuple or a tuple set with probability θ_v^{out}, which is equivalent to the selectivity of the relational operator v. A plan tree G has an input operator set V_{in} that includes at least one input operator and only one output operator v_{out}. An input operator accepts tuples from a data source in the data source set S. An output operator generates result tuples for the query.

For a given CPU core set U, consider a CPU core assignment strategy $\pi: v \to u$ that determines "which operator is executed on which CPU core $u \in U$." Let $\mathtt{C}$ be the processing power of a CPU core per unit of time. Let λ_v be the number of tuples processed by an operator v per unit time. We get

$$\forall u \in U, \quad \sum_{\{v|\pi(v)=u\}} c_v \lambda_v \leq \mathtt{C}$$

as a necessary and sufficient condition between the number of tuples that a CPU core processes per unit time and the processing power of the CPU core. In addition, assume that sending a tuple from a CPU core to another core causes latency w. The proposed method assumes that the CPU has a sufficient number of cores. Thus, communication via QPI does not occur. The method does not deploy an operator to multiple CPU cores, which means the method does not use data parallel computation techniques of QueueLinker.

The latency of an output tuple can now be defined. Let $t_{in}(\alpha)$ be the time at which the input tuple α from data source $s \in S$ arrives at an input operator. Let $t_{out}(\beta)$ be the time at which the output operator generates an output tuple β. Let $e(\beta)$ be a set of input tuples that contribute to the generation of the output tuple β. Let $l(\beta)$ be the latency of the output tuple β, described as

$$l(\beta) = t_{out}(\beta) - \max_{\forall i \in e(\beta)} t_{in}(i). \quad (1)$$

Formula (1) describes the time difference between an input tuple arriving at an input operator and an output tuple being generated from the output operator. In other words, it gives the response time following the point at which all the data required to produce the result tuple has arrived. Thus, smaller values of (1) imply a better latency performance, all else being equal.

Consider the path that an input tuple passes through after arriving at the input operator v_{in} until it is outputted from the output operator v_{out}. There is only one such

path if the plan is a tree. Thus, the path can be described as $P(v_{in}) = \langle v_1, v_2, \dots, v_{\mathrm{L}} \rangle$, $(v_1 = v_{in}, v_{\mathrm{L}} = v_{out})$. Let us assume that each operator can begin processing a tuple immediately upon arrival, and that no interruptions occur during the processing. In addition, operators that have two or more input queues process the tuples independently, and the operator does not wait for the different queues. Let $l\big(P(v_{in})\big)$ be the latency of a tuple going through a path $P(v_{in})$, described as

$$l\big(P(v_{in})\big) = w \times |\{i|\, \pi(v_i) \neq \pi(v_{i+1}), 1 \leq i \leq \mathrm{L}-1\}| + \sum_{v \in P(v_{in})} \frac{c_v}{\mathrm{C}}. \tag{2}$$

Let $\theta^{out}_{P(v_{in})}$ be the probability that an output tuple is generated through path $P(v_{in})$ after a tuple arrives at an input operator v_{in}. The probability is expressed as the product of the selectivities of the operators on the path:

$$\theta^{out}_{P(v_{in})} = \prod_{v \in P(v_{in})} \theta^{out}_{v}$$

Let $f(v_{in})$ be the ratio of tuples arriving at v_{in} to all input tuples. We can obtain the average latency of a query Q as

$$\overline{l(Q)} = \sum_{v_{in} \in V_{in}} \frac{f(v_{in})\theta^{out}_{P(v_{in})}}{\sum_{v' \in V_{in}} f(v')\theta^{out}_{P(v')}} l\big(P(v_{in})\big). \tag{3}$$

For example, the average latency of the plan tree in Figure 4.5 can be calculated as $\overline{l(Q)} = (0.125 \times 0.08 \times 0.185 + 0.25 \times 0.06 \times 0.185 + 0.625 \times 0.12 \times 0.035)/0.1 = 0.0725$. The problem can now be stated using the above definitions.

CPU Core Assignment Problem with Minimization of Average Latency

Find an optimal CPU core assignment strategy π_{OPT} that minimizes the average latency $\overline{l(Q)}$ subject to $\forall u \in U$, $\sum_{\{v|\pi(v)=u\}} c_v \lambda_v \leq \mathrm{C}$. There are enough CPU cores to assign all operators.

Theorem

The "CPU Core Assignment Problem with Minimization of Average Latency" is an NP-hard problem.

Proof

Consider the knapsack problem of capacity C. There are I items, the sizes of which are given by $S = \{s_1, s_2, \ldots, s_I\}$ and the values of which are given by $L = \{l_1, l_2, \ldots, l_I\}$. Consider a plan tree that has I input operators $c_{v_i} = s_i/l_i$, $\lambda_{v_i} = l_i$, $f(v_i) = l_i / \sum_j l_j$, and an output operator $c_{v_{out}} = 0$. The processing power of a CPU core is C (Figure 4.6). Now, consider the solution of the core assignment problem with $\forall v \in V,\ \theta_v^{out} = 1$. Input operators assigned to the same core as the output operator obviously give the solution to the knapsack problem. Therefore, the problem is NP-hard,

$$f(v_1) = 10/(10+20+50) = 0.125$$
$$f(v_2) = 20/(10+20+50) = 0.25$$
$$f(v_3) = 50/(10+20+50) = 0.625$$

$$\theta^{out}_{P(v_1)} = 0.4 \times 0.5 \times 0.4 = 0.08$$
$$\theta^{out}_{P(v_2)} = 0.3 \times 0.5 \times 0.4 = 0.06$$
$$\theta^{out}_{P(v_3)} = 0.6 \times 0.5 \times 0.4 = 0.12$$

$$l\big(P(v_1)\big) = 0.1 + (1.5+5+2)/100 = 0.185$$
$$l\big(P(v_2)\big) = 0.1 + (1.5+5+2)/100 = 0.185$$
$$l\big(P(v_3)\big) = (0.5+1+2)/100 = 0.035$$

$$\sum_{v\prime \in V_{in}} f(v')\theta^{out}_{P(v\prime)} = 0.125 \times 0.08 + 0.25 \times 0.06 + 0.625 \times 0.12 = 0.1$$

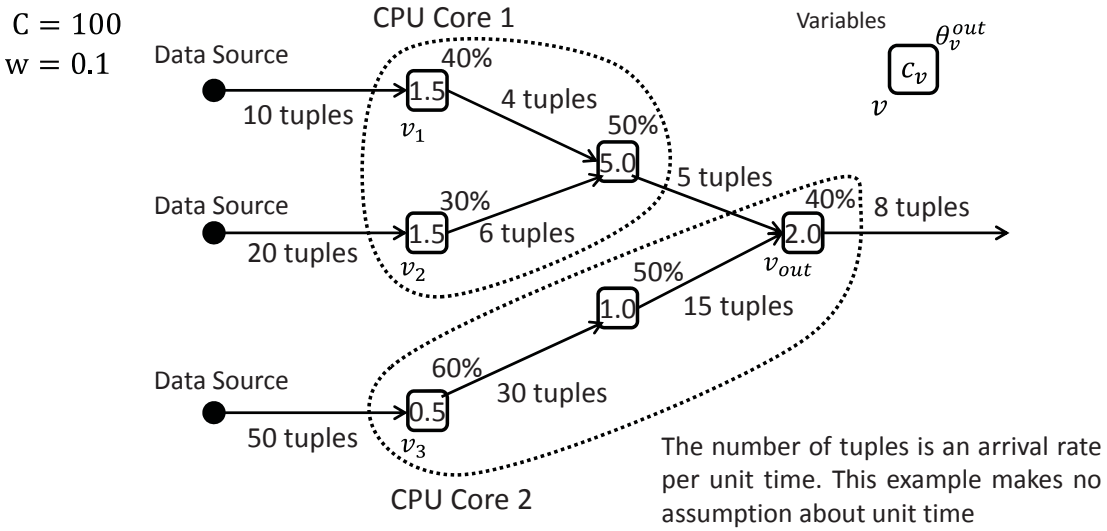

Figure 4.5 Calculation Example of Average Latency

because it can be transformed into a reduction of the knapsack problem, which is known to be an NP-hard problem. ■

Note that "Average Path Length Minimization" [44], which finds a mapping between nodes of a tree structure and disk pages while minimizing the average number of disk seeks, is equivalent to a core assignment problem if $\forall v \in V\ (\theta_v^{out} = 1,\ c_v\lambda_v = 1)$ and $w = 1$.

4.3.2. A Solution using Dynamic Programming

We can minimize (3) by minimizing the first term of (2). The proposed method finds the optimal solution π_{OPT} by modifying the dynamic programming solution of "Average Path Length Minimization" [44] to consider the case that $\forall u \in U,\ \sum_{\{v|\pi(v)=u\}} c_v\lambda_v \leq \mathrm{C}$.

Consider n operators $vp_1, vp_2, \ldots, vp_n$ that send tuples to an operator v. and i sub-trees, the roots of which are $vp_1, vp_2, \ldots, vp_i\ (i \leq n)$. Consider operators that are assigned to the same CPU core as v and in the same sub-trees. Assuming v is an output operator, let $S[v,i,j]$ be the minimal average latency when the total resource consumption of the operators is exactly equal to j (Figure 4.7). To simplify, the description below uses $S_1[v,j] = S[v,n,j],\ S_2[v] = \min_{1\leq j\leq \mathrm{C}} S_1[v,j]$. Let $F[v]$ be the ratio of the number of output tuples from v to the number of output tuples from v_{out}. This value can be calculated recursively by $F[v] = \sum_{1\leq i\leq n} F[vp_i]$ for non-input operators $V_{mid} = \{v|v \in V, v \notin V_{in}\}$. The initial values for input operators $v \in V_{in}$, are

$$S[v,0,c_v\lambda_v] = S_1[v,c_v\lambda_v] = S_2[v] = F[v] = \frac{f(v)\theta_{P(v)}^{out}}{\sum_{v'\in V_{in}} f(v')\theta_{P(v')}^{out}}$$

$$S[v,0,j] = S_1[v,j] = \infty \quad (j \neq c_v\lambda_v)$$

For non-input operators $v \in V_{mid}$,

$$S[v,0,c_v\lambda_v] = 0$$

$$S[v,0,j] = \infty \quad (j \neq c_v\lambda_v),$$

which defines the values recursively as below.

$$S[v,i,j] = \min(\mathrm{A},\mathrm{B})$$

$$\mathrm{A} = S_2[vp_i] + F[vp_i] + S[v,i-1,j]$$

$$\mathrm{B} = \min_{1\leq m<j} (S_1[vp_i,m] + S[v,i-1,j-m]).$$

We can obtain the minimal average latency $\overline{l(Q)_{\mathrm{OPT}}}$ of π_{OPT} as

$$\overline{l(Q)_{\mathrm{OPT}}} = w \times (S_2[v_{out}] - 1) + \sum_{v_{in} \in V_{in}} \frac{f(v_{in})\theta^{out}_{P(v_{in})}}{\sum_{v' \in V_{in}} f(v')\theta^{out}_{P(v')}} \sum_{v \in P(v_{in})} \frac{c_v}{\mathrm{C}}. \quad (4)$$

If $c_v\lambda_v$ and C are handled by integers, the time complexity is $\mathrm{O}(\mathrm{C}^2N)$ and the space complexity is $\mathrm{O}(\mathrm{C}N)$, where N is the number of operators. As usual in dynamic programming methods, the optimal CPU core assignment π_{OPT} can be recovered by tracking back the calculations. This method finds an optimal static CPU core assignment. The next section discusses dynamic operator reassignment to respond to

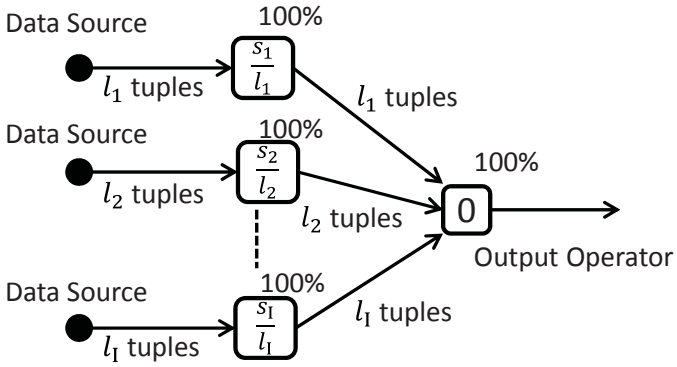

Figure 4.6 Reduction from the Knapsack Problem

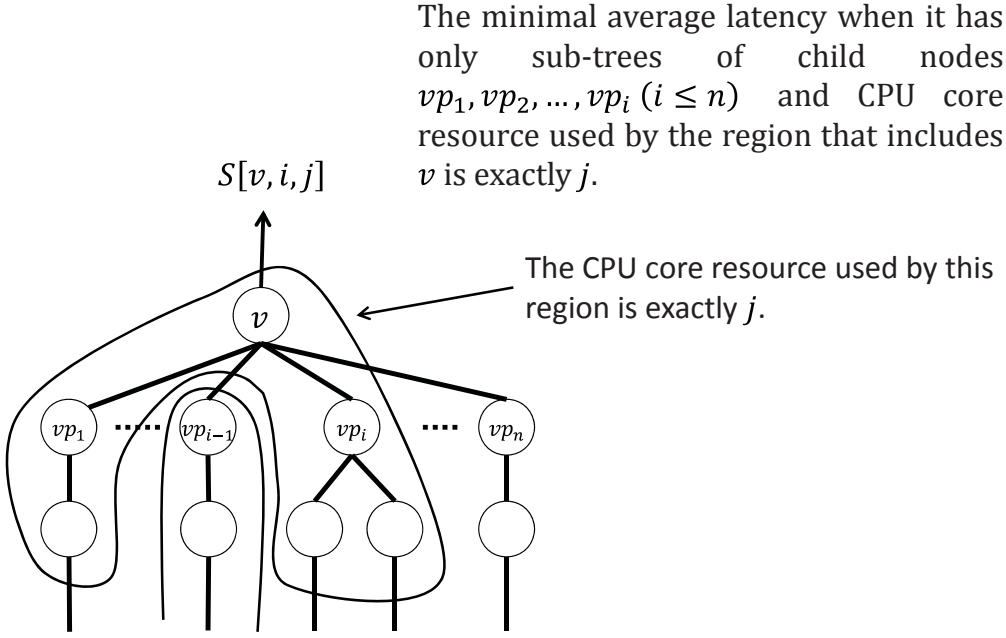

Figure 4.7 Definition of $S[v, i, j]$

changes in the computational load.

4.4. Dynamic Operator Reassignment and Statistics Collection

This section describes the proposed method of operator reassignment, and also discusses how to collect statistics regarding the computational load of the operators. Although the proposed method in Section 4.3.2 gives a static optimal solution, the computational loads of the operators change depending on both the content and arrival speed of data streams. Therefore, CPU cores should be reassigned to respond to these changes. However, operators that have internal states, such as join operators, have to process tuples in order of arrival. Therefore, it is necessary to migrate operators among thread units with low overhead to reduce the processing latency.

4.4.1. Thread Units and the Collection of Statistics

The proposed method assigns only one push thread unit (TU in this section) of QueueLinker to a CPU core. A TU has only one FIFO queue, and communication between TUs is conducted by messages placed in the queue. A TU runs on a dedicated CPU core, and thus communication between the queues is completed inside a CPU. TUs should adopt the busy wait while waiting for a new message, though the method described below is independent of the waiting methodology.

In addition to the given operators, a scheduler is run on a dedicated CPU core using the method described in Section 4.3.2. For each operator, the TU records the total CPU time consumed, the total number of input tuples, and the total number of output tuples. TUs update these statistics after processing a tuple[1]. The selectivity of an operator can be calculated by dividing the number of output tuples by the number of input tuples. Each TU sends the collected statistics to the scheduler at regular intervals. The interval should not be smaller than the time needed to solve the dynamic programming described in Section 4.3.2. Therefore, the overhead associated with sending statistics to the scheduler is not large, because the interval is long enough compared to the tuple arrival interval. The calculation time under dynamic programming is shown in Section 4.5.1. Depending on the statistics, the scheduler finds the core assignment by the method described in Section 4.3.2. When operator reassignment is required, the scheduler requests TUs to reorganize the CPU core assignment. Note that, at the beginning of the query execution, the scheduler assigns enough CPU cores to operators because it does not know any of the operator statistics.

[1] rdtsc instruction can be used to get the current CPU clock count with a low overhead. The total CPU time consumed can be calculated from the time before and after an operator execution.

After the scheduler collects a sufficient amount of statistics, it reassigns the cores.

4.4.2. Operator Reassignment Procedure

Relational operators usually have to process tuples in order of arrival, because processing tuples in a different order gives a different result. The proposed method restricts the direction of operator migration and realizes low-overhead operator reassignment without suspension of stream processing.

Consider the case of an operator v running on TU1 being migrated to TU2, as in Figure 4.8. The FIFO queue of TU2 has tuples generated by v, and the tuples are waiting to be processed by operator d. In the proposed method, TU1 adds a migration instruction message to the TU2 queue in order to migrate operator v. This ensures that the tuples, which are put in the TU2 queue before the migration message for operator v arrives, have already been processed by operator v, because the queue is first-in-first-out. This also ensures that tuples arriving after the message are not processed by operator v.

On the other hand, if the system moves operator d from TU2 to TU1, there is no such guarantee. In this case, the queue must be locked to prevent further tuples from arriving, and then operator d must be applied to all tuples in the TU2 queue. This causes a high overhead concurrency control and increases the processing latency. Thus, the proposed method restricts the direction of operator migration to the same as the

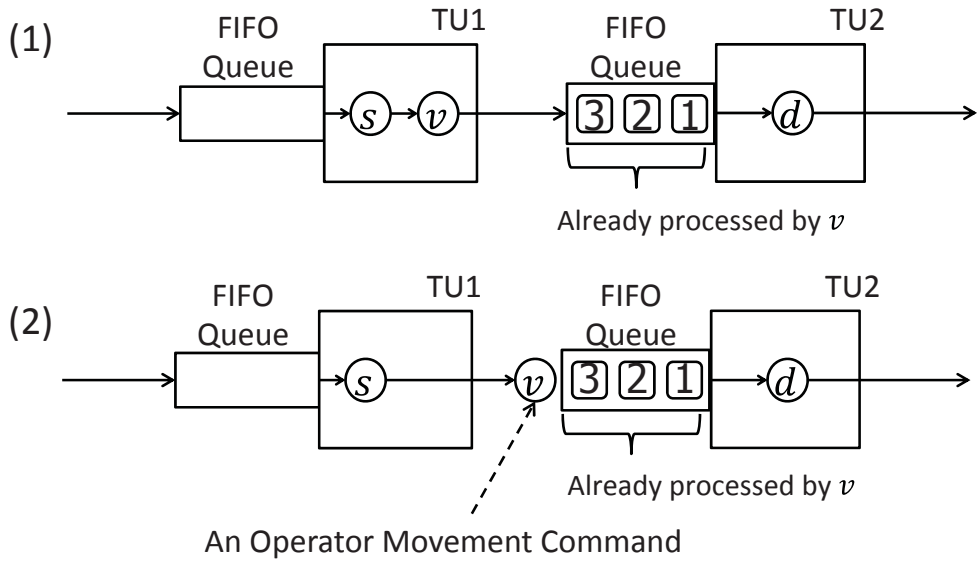

Figure 4.8 Operator Movement Operation

data transfer to avoid such high overhead controls. Note that the internal state of an operator can be seen from the TU that the operator moved to, because the system runs on a shared memory computer.

The next consideration is how to maintain one-hop communication, as in the model described in Section 4.3, while ensuring that the tuples are processed in order of arrival. Consider the migration of operator v to TU3 from state (1) in Figure 4.9. To maintain one hop communication, TU1 must send the result tuples of s directly to TU3. When TU1 changes the destination of tuples, the FIFO queue of TU2 contains tuples from s, and thus TU2 has to send them to TU3, as shown in state (2) in the figure. In this case, TU3 may accept the tuples without preservation of tuple arrival order, because it accepts them from both TU1 and TU2. Therefore, TU3 has to sort them according to the tuple IDs and then apply v to the tuples. Note that TU2 does not have to do anything when TU1 sends a migration message for s to the TU2 queue in the

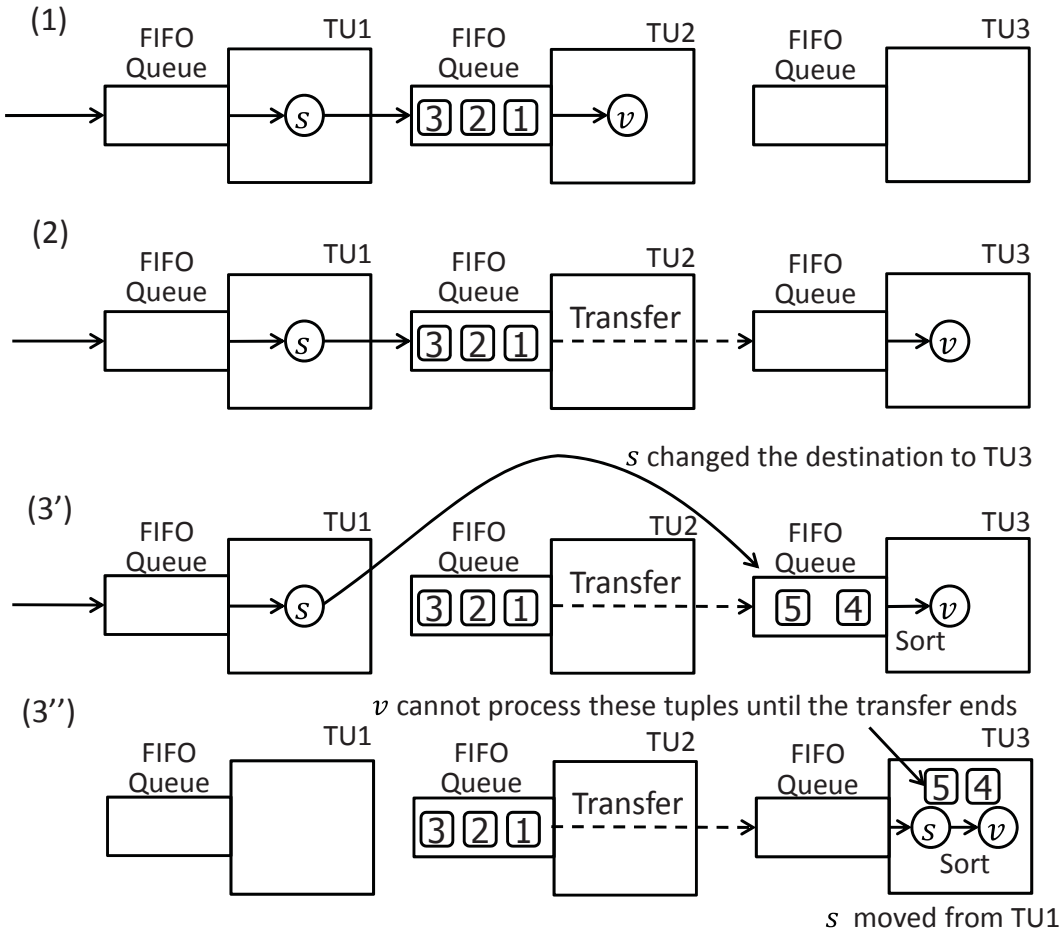

Figure 4.9 Operator Movement Operation with Tuple Transfer

situation of (2) or (3') in Figure 4.9. This is because it is guaranteed that TU2 has finished transferring the tuples when TU2 recognizes the migration message.

Figure 4.9 (3") shows the state after operator s has moved to TU3 from state (2). In this case, the tuples generated by s remain in the TU2 queue. Therefore, even if TU3 processes the input tuples for s, and s generates new tuples, v must not process the new tuples. Operator v has to wait until the transfer from TU2 finishes, and then process the transferred tuples prior to the tuples that s generated in TU3.

The above description can be applied to a plan tree to give the state transition diagram shown in Figure 4.10. This state transition defines the state of an edge from operator src to operator dest. Each TU manages a state for each edge. NONE is the initial state in which TUs execute neither src nor dest. SORT is the state in which the TU starts to execute dest and has to sort the input tuples. After the sorting finishes, the state is changed to RECV. Migrating src to the TU changes the state from RECV to RUN, whereby the TU executes both the src and dest operators. After this, migration of the dest operator to other TUs changes the state to SEND, whereby the TU has to transfer the tuples generated by src to the TU to which the dest operator moved. In

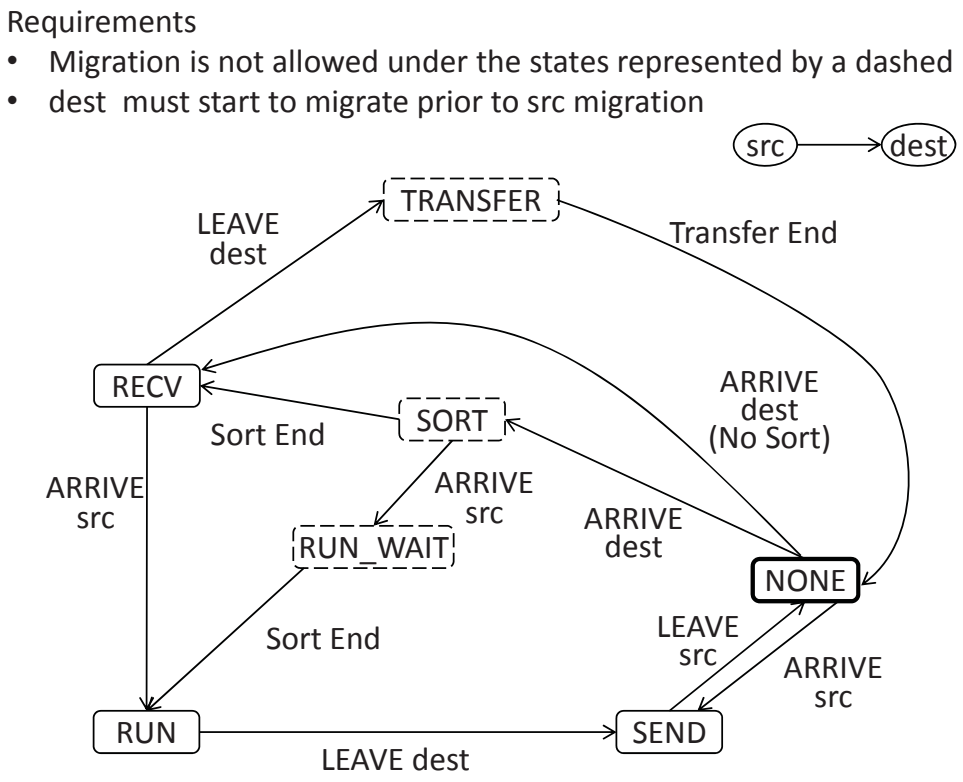

Figure 4.10 State Transition Diagram

addition to these states, TRANSFER corresponds to TU2 transferring the tuples as shown in Figure 4.9 (2), (3'), and (3"). RUN_WAIT corresponds to TU3 in Figure 4.9 (3"). In this case, the TU waits until the transfer has completed.

As described above, restricting the direction of operator migration allows us to realize operator migration and communication route changes without concurrency control.

4.5. Performance Evaluation

This section presents a performance evaluation of the proposed method. The experimental environment is given in Table 4.1.

4.5.1. Computation Time of the Dynamic Programming

This subsection confirms the computation time of the dynamic programming method described in Section 4.3.2. Figure 4.11 shows the time taken to find an optimal solution under dynamic programming. From the figure, we can see that the computation time increases linearly with the number of operators. The algorithm took about 3 ms to calculate the solution for a plan tree consisting of 100 operators, which

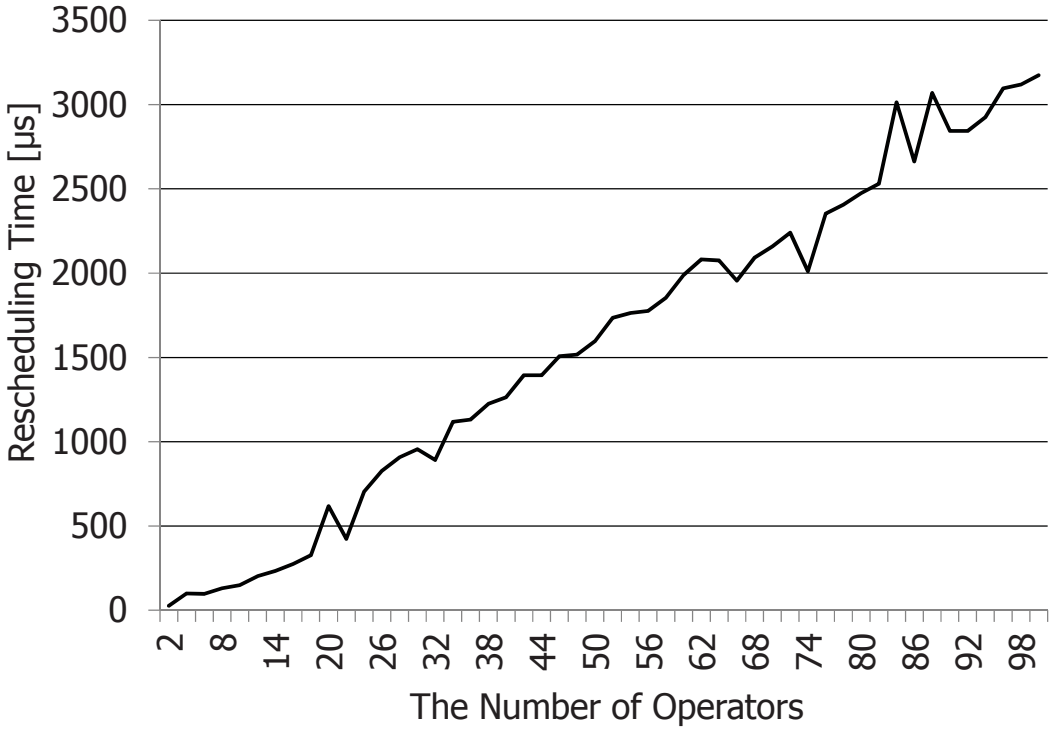

Figure 4.11 Scheduling Time and the Number of Operators

means that the scheduler can re-calculate the solution over 300 times a second. The graph provides a guideline to determine the time interval for sending statistics from thread units to the scheduler. Operator reassignment takes place after the scheduler finds an optimal solution. Therefore, the time interval of sending statistics to the scheduler can be determined from Figure 4.11.

4.5.2. Experiment for a Query Plan Tree

The next experiment used a plan tree consisting of relational operators. Figure 4.12 shows the plan tree used in this experiment. The query received input tuples from eight data sources and outputted join results. The window size of the join operators was 10 s. A tuple from a data source was composed of four integers with attribute names A, B, C, and D. Four data sources supplied 5,000 tuples per second and the other four sources supplied 1,250 tuples per second. In the experiment, the latency was measured under a change of input speed every 5 s between patterns A and B, as shown in Figure 4.12. This figure also shows an optimal CPU core assignment for pattern A. The experiment compared the proposed method of changing the assignment dynamically depending on the pattern with the optimal core assignment for pattern A and no change

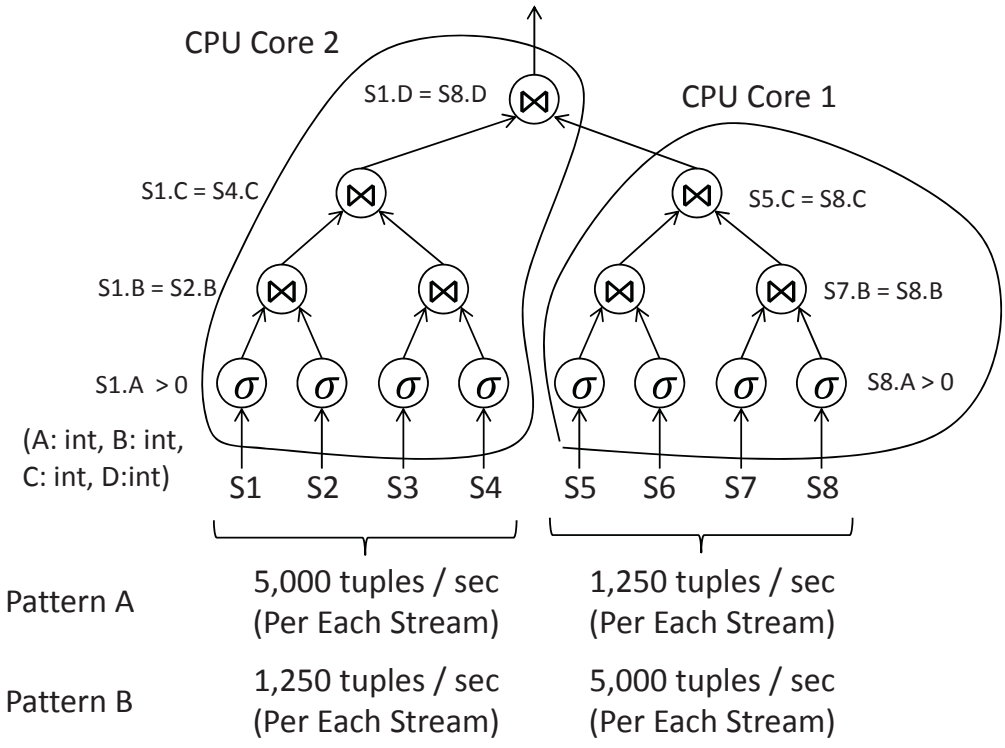

Figure 4.12 Query Plan Tree and CPU Core Mapping For Pattern A

of assignment. The thread units used the busy wait method. The experiment used pattern A from 5 s to 10 s, and then changed between patterns A and B every 5 s.

Figure 4.13 shows the results of the experiment, where the average latency is taken over 100 ms. The figure omits results from before 10 s because the window of the join operators is not filled during this time. 'Static' denotes a static method that did not change the optimal core assignment for pattern A, resulting in increased latency when the stream speed changed to pattern B. The figure shows that the proposed method maintained a similar latency after the stream speed change because it changed the core assignment. For example, we can see the benefits of the proposed method from 10–15 s, 20–25 s, and 30–35 s. The reassignment of cores after the stream speed changed caused the temporary latency increase shown in Figure 4.13. However, the increased latency did not exceed that of the static method, demonstrating that the proposed method can reassign CPU cores with low overhead. Note that the increased latency of the proposed method around 10 s was caused by the Java JIT compiler, because the Java runtime performed the JIT compilation the first time cores were reassigned.

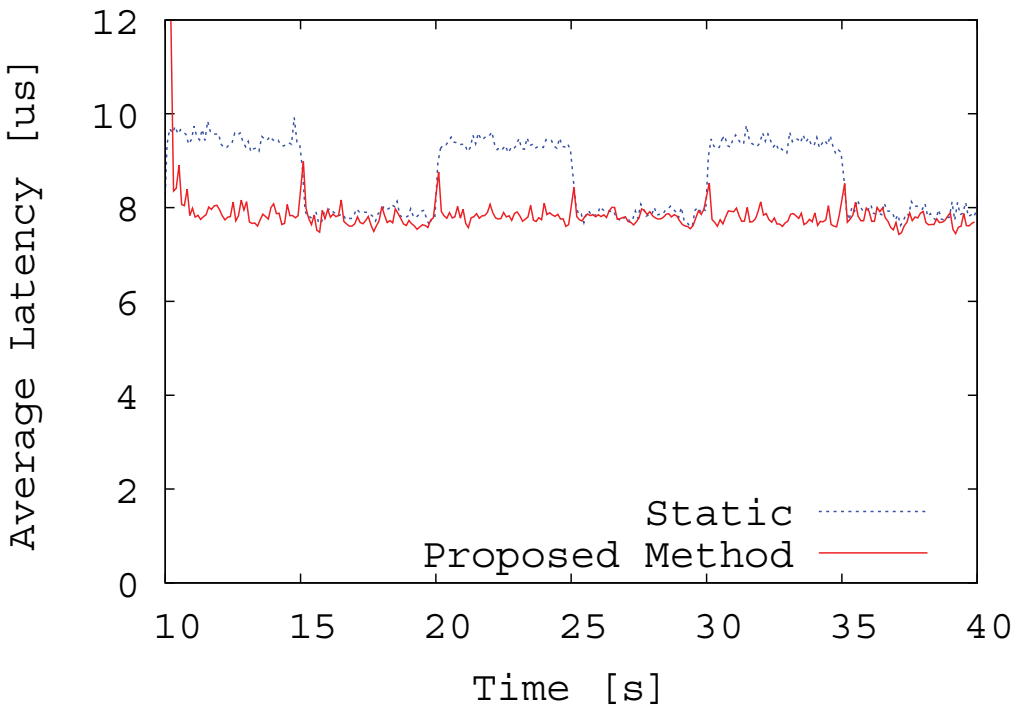

Figure 4.13 Experimental Result of Average Latency

4.6. Discussion of Applicable Conditions for the Proposed Method

This section discusses the conditions necessary for the proposed method to achieve the maximum benefit. The effectiveness of the proposed method is determined by the relationship between operator computation time and the communication latency. For a given plan tree, consider the worst-case CPU core assignment, whose communication latency is the largest possible. Obviously, the worst case is the assignment in which every CPU executes only one operator. The average latency $\overline{l(Q)_{\text{worst}}}$ of the worst case is

$$\overline{l(Q)_{\text{worst}}} = \sum_{v_{in} \in V_{in}} \frac{f(v_{in})\theta^{out}_{P(v_{in})}}{\sum_{v' \in V_{in}} f(v')\theta^{out}_{P(v')}} \left\{ w(|P(v_{in})| - 1) + \sum_{v \in P(v_{in})} \frac{c_v}{\mathrm{C}} \right\}.$$

If we use

$$A\big(g(v_{in})\big) = \sum_{v_{in} \in V_{in}} \frac{f(v_{in})\theta^{out}_{P(v_{in})}}{\sum_{v' \in V_{in}} f(v')\theta^{out}_{P(v')}} g(v_{in})$$

to simplify, the ratio R of the optimal value $\overline{l(Q)_{\text{OPT}}}$ given by the proposed method to the worst case is

$$\begin{aligned} R = \frac{\overline{l(Q)_{\text{OPT}}}}{\overline{l(Q)_{\text{worst}}}} &= \frac{w(S_2[v_{out}] - 1) + A\left(\sum_{v \in P(v_{in})} \frac{c_v}{\mathrm{C}}\right)}{A\big(w(|P(v_{in})| - 1)\big) + A\left(\sum_{v \in P(v_{in})} \frac{c_v}{\mathrm{C}}\right)} \\ &\geq \frac{A\left(\sum_{v \in P(v_{in})} \frac{c_v}{\mathrm{C}}\right)}{A\big(w(|P(v_{in})| - 1)\big) + A\left(\sum_{v \in P(v_{in})} \frac{c_v}{\mathrm{C}}\right)} \\ &= \frac{1}{\frac{A\big(w(|P(v_{in})| - 1)\big)}{A\left(\sum_{v \in P(v_{in})} \frac{c_v}{\mathrm{C}}\right)} + 1}. \end{aligned} \tag{5}$$

R can be considered as the performance ratio of the proposed method to the worst case. Formula (5) is an optimistic estimation of R, and gives the highest performance improvement of the proposed method when it can reduce all communication latencies. The formula represents the contribution of the method as the sum of operator computation time c_v and inter-core communication latency w. The smaller is c_v, the greater the ratio of communication latency to query latency $\overline{l(Q)}$, providing more benefits from reducing the communication latency by the proposed method.

The next consideration is the cost of reassignment. The computation time of the dynamic programming gives a suggestion of the cost. The proposed method finds an

optimal solution using a dynamic programming method depending on the input stream speed and the CPU consumption of operators, reassigning operators when the optimal assignment changes. Figure 4.11 shows that the computation time of the dynamic programming increases linearly with the number of operators. Therefore, if the scheduler manages a large number of operators, the computation time becomes long and the scheduler cannot reassign CPU cores to respond to changes in computational load. This is a weakness of the proposed method, as it is unavoidable in finding a solution. Thus, to use the method most efficiently, the computation time of the dynamic programming must be shorter than the intervals at which reassignment is required.

The optimization target of the method is a plan tree consisting of relational operators that process a tuple in memory over a short time period. The purpose of the method is to reduce the inter-core communication latency w, which is usually a small value of the order of several microseconds. Therefore, it is not clear whether the method can produce positive effects for a plan tree including relational operators, such as selection or join, that may have a small c_v.

The experimental evaluation showed that the method produces positive effects in terms of reducing the communication latency when selection operators and join operators are used in a plan tree. Moreover, the comparison in Figure 4.13 is an optimal assignment for pattern A. This is a more severe condition than the worst assignment where a CPU core executes only one operator, which is used to give the optimistic estimation of formula (5). Under this severe condition, the method can reduce the latency from 10 μs to 8 μs, as shown in Figure 4.13.

The effectiveness of the proposed method is determined by the calculation interval of the dynamic programming, the timing of operator reassignment, and the number of operators that must migrate. However, the degree to which these aspects affect performance is still an open question. The main contribution of this chapter is to show the possibility of reducing communication latency by operator reassignment, an approach that has not been considered in previous research in the field of continuous queries.

4.7. Summary and Future Work from this Chapter

This chapter detailed a low-latency execution method for continuous queries on a multi-core CPU. A definition of the latency of continuous queries was given. The proposed dynamic programming algorithm gives an optimal CPU core assignment for the relational operators of a plan tree. This method includes a dynamic operator reallocation technique that does not require stream processing to be interrupted during

the reallocation.

There are many areas of future work, including an evaluation of the frequently changing arrival rate of streams and communication over QPI, which is required when operators are assigned to multiple CPUs. Communication over QPI makes the problem difficult, because the latency is different from the inter-core communication latency w and the system has to handle the two different latency values. Another area of work involves considering future computer environments in which a few hundred or thousand CPU cores are available. In such an environment, each CPU core may have to adjust its assignment without a centralized scheduler, which cannot handle the statistics from so many cores. In addition, the arrival of data over networks should be considered when applying the method to an actual system. Data can be sent to other computers in several microseconds using InfiniBand's RDMA (Remote Direct Memory Access). The proposed method should work well under such a low-latency network environment.

Chapter 5. A Backup Method for Reducing Latency of a Continuous Query

High-speed data streams and/or queries that consume large amounts of memory represent significant opportunities for distributed processing. To seize this opportunity, QueueLinker can be used to run continuous queries on commodity clusters, executing the operators of those queries in a data parallel model. As described in previous chapters, there are a number of applications that require low latency query processing.

In the distributed processing of a continuous query, the deployment of operators to computation nodes introduces a certain amount of latency. While increasing the number of computation nodes can improve processing throughput, it will increase query latency due to the greater amount of network communications between operators running on different nodes. By the same token, while reducing the number of computation nodes can reduce query latency, it can decrease processing throughput. Migration of operators is required when the computational load is heavy, or when sudden variations in stream speed occur.

The method introduced in this chapter is based on the notion that a stream processing system has to make backups of operators. The proposed backup method is called “Chase Execution”, and it ensures that the internal states of all operators must be replicated to multiple computers in order to cope with computation node failures. Under Chase Execution, the secondary query execution uses a different set of operator deployments than the primary execution, and results are generated from the fastest tuples output by both deployments. This helps reduce overall processing latency and handle changes in the speed of incoming data streams. Experimental results show that the chase execution method can achieve a lower latency than deployments that do not perform backup.

5.1. Background and Example

In this chapter, an IPS (Intrusion Prevention System) is combined with a DSMS to form an example application. IPS protects networks from malicious activities by dropping malicious packets. As shown in Figure 5.1, the DSMS monitors packet headers and lets the IPS determine whether each packet should be allowed for transmitting. The IPS makes this determination based on complicated analytic queries rather than simple pattern matching techniques.

When a DSMS is used with an IPS in this way, it must balance the following priorities:

- Reducing the latency of a query
- Responding to changes in the rate of incoming data
- Handling computation node failures

The DSMS ought to process packets with the lowest possible latency. Note that if packets are sent under the TCP protocol, the packet sender must wait for an 'ack' (acknowledgment) signal to return from the receiver if the TCP window size is full. Thus, throughput will be degraded if the packet round trip time becomes large. The DSMS must respond to changes in the rate of incoming packets while maintaining low latency, and it must be able to handle computation node failures. If the DSMS fails, the IPS cannot process incoming packets, and the network as a whole will fail as a result.

This chapter will consider operator deployments for the following simple analytic query of a packet monitoring system:

Example Query: For each arriving packet, calculate (count(srcIP), count(destIP)) for the last 3 minutes of activity.

Each packet carries a 'Source Address' (srcIP) field and a 'Destination Address' (destIP) field in its header. On packet arrival, the above query calculates the total number of appearances of the 'Source Address' and the 'Destination Address' in all packets arriving over the previous 3 minutes. This query can be executed by two grouping operators. A grouping operator maintains the number of appearances of each

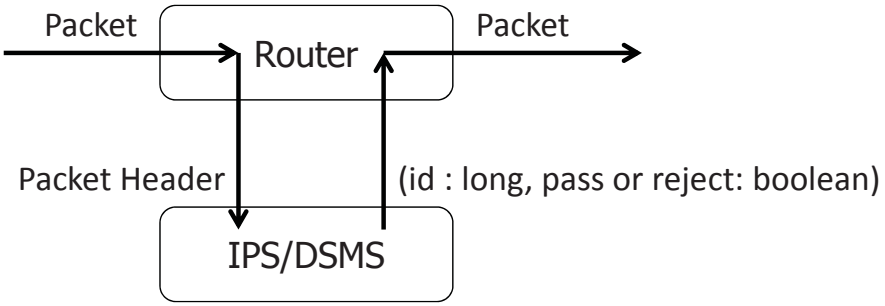

Figure 5.1 Conceptual Diagram of an IPS with a DSMS

IP address using a hash table or some similar data structure. More formally, let γ_{srcIP} be an operator to count the appearance of each srcIP and let γ_{destIP} be an operator to count the appearance of each destIP.

The following section addresses operator assignment to available computation nodes. The latency of the above query and its responsiveness to fluctuations in incoming data rates will be determined by this assignment.

5.1.1. Operator Deployment Strategy and Latency

Figure 5.2 shows two examples of operator deployment for the operators γ_{srcIP} and γ_{destIP}. The top of the figure shows a deployment that executes the query using only one computer (Node A). In this deployment, the output tuple incurs no network latency because no network communication is required. However, the length of the input queue will grow if the arrival rate exceeds the processing rate of Node A, resulting in increased latency. If queue growth persists, Load Shedding [19], which discards tuples or sacrifices the accuracy of the computation, must be employed to prevent network failure.

By contrast, the bottom of the Figure 5.2 shows a deployment that executes the query through data parallel processing, using hash partitioning between two computation nodes (Node A and B). The system determines to which of the two nodes an input tuple should be routed, depending on the hash value of srcIP. After the receiving

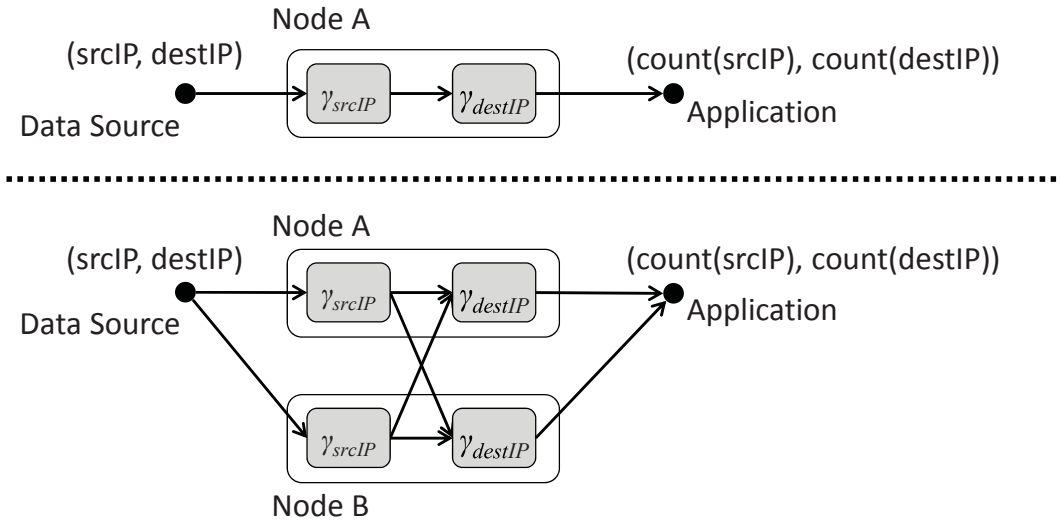

Figure 5.2 Deployment Examples for Operators in the Example Query

node applies γ_{srcIP} to the tuple, it forwards it to the next computation node depending on the hash value of its destIP. Assuming a uniform distribution of srcIPs and destIPs, Node A and B will process the same number of tuples, and transfer half of those tuples to one another, between the γ_{srcIP} and γ_{destIP} operations. If the system uses more computation nodes, the processing load per node decreases, but the number of tuples transferred between nodes increases. In either case, new latency is introduced.

Under the above scenario, the best way to maintain low latency is to use one computation node when the data arrival rate is low and multiple nodes when the arrival rate is high. So long as the arrival rate is not too volatile, adjusting the number of computation nodes in this way should help the system execute the query with consistently lower latency. Note, however, that the internal states of operators γ_{srcIP} and γ_{destIP} must also be transferred when the number of computation nodes is adjusted. If the data arrival rate changes very sharply, there may be insufficient time to transfer internal state without creating new latency.

5.1.2. Operator Backup

The technique of Operator Backup provides a means of handling computation node failures. Under this technique, only those operators with internal state are replicated, as stateless operators (e.g. those that perform selections) can be redeployed and restarted at any time. In the scenario above, the γ_{srcIP} and γ_{destIP} operators would be replicated ("backed up") to prevent loss of that state when a computation node fails.

Active Standby [45] is a previously proposed method of achieving high availability for DSMSs. Figure 5.3 shows how Active Standby can be applied to the deployment depicted at the top of Figure 5.2. Replicated data is sent to secondary processing, but the results of this processing only sent back to the application if the primary node fails. Note that this does not, in itself, reduce the latency described in 5.1.1, since primary and secondary processing must still be configured with the same deployment. Further, note that Active Standby requires twice the number of computation nodes for stateful operations. For example, four computation nodes will be required for the deployment depicted at the bottom of the Figure 5.2.

5.1.3. The Purpose of the Proposed Method

Chiefly, DSMSs need to reduce latency in response to changes in data arrival rate, and thereby preserve high availability. Many applications can be seriously affected by latency, even on the order of several milliseconds, and in the case of IPS applications, security precludes use of Load Shedding because all packets must be examined.

Using Chase Execution allows a query to be executed on a set of alternative operator deployments and to get results based on the fastest tuple outputs from the deployments. The proposed method simultaneously tries to reduce processing latency, handle node failures, and respond to changes in the speed of a data stream.

5.2. Proposed Method: Chase Execution

This section describes Chase Execution, the proposed backup method. Under Chase Execution, a query is executed by a set of alternative operator deployments and the results of the query are generated from the fastest tuples output by the deployments. Alternative query deployments are called "chasers". Arriving tuples of data are duplicated and send to these chasers for processing.

Figure 5.4 shows the deployment of the example query using Chase Execution. While Chaser 1, consisting of Node A, executes the query, Chaser 2, consisting of Nodes B, C and D will also execute the query. As described in 5.1.1, Chaser 1 can produce the results faster than Chaser 2 as it will require no network communication between the γ_{srcIP} and γ_{destIP} operators. Meanwhile, Chaser 2 uses three computation nodes and network communication between the operators, increasing latency. Only if the data arrival rate becomes too large and Chaser 1 cannot process the data due to the shortage of computational resources, will Chaser 2 produce the results faster than Chaser 1. When the data arrival rate goes back down, Chaser 1 will again produce the results faster than Chaser 2.

For robustness, Chase Execution always organizes alternative chasers using different numbers of computation nodes. The Chase Operator (Figure 5.5) produces the results of the query from the fastest tuples output by the chasers. Even if a node failure

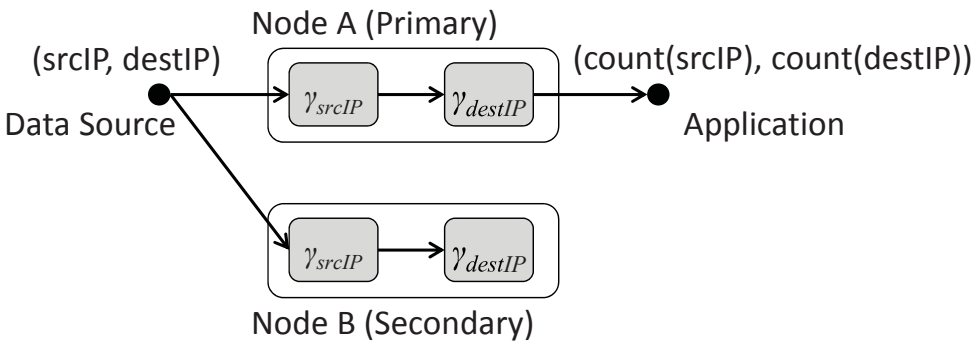

Figure 5.3 An Example of Active Standby for the Example Query

occurs in a chaser, it maintains other chaser(s) consisting of normal status nodes and can achieve high availability by using tuples from the normal status chaser(s). Previous methods for DSMS, such as Active Standby and hot standby (for duplex systems), construct secondary processing with the same deployment as primary processing, and do not use the secondary under normal conditions. This is in contrast to Chase Execution, which uses multiple chasers even under normal (i.e. non-failure) conditions.

5.2.1. Chase Operator

Figure 5.4 shows a Chase Operator that generates results for a query by consuming the earliest tuples output by multiple chasers. A tuple with a duplicate ID that arrives later is discarded. For Chase Operator to work correctly, each tuple must have a tuple ID representing its ascending order of arrival. In addition, output tuples of operators must hold the same tuple IDs as input tuples. Some operators may produce a single tuple based on multiple input tuples. In such a case, the latest tuple ID among the input tuples consumed by the operator is used as the ID of the output tuple.

Pseudo-code for the Chase Operator is given in Figure 5.5. Note that the procedure does not depend on the number of chasers and has a single input queue Q_{in}. $id(\tau)$ denotes a function that returns the tuple ID of a tuple τ. From a chaser running with multiple computers, such as Chaser 2 in Figure 5.4, tuples may not arrive in ascending order of their IDs. To address this, the Chase Operator detects and discards

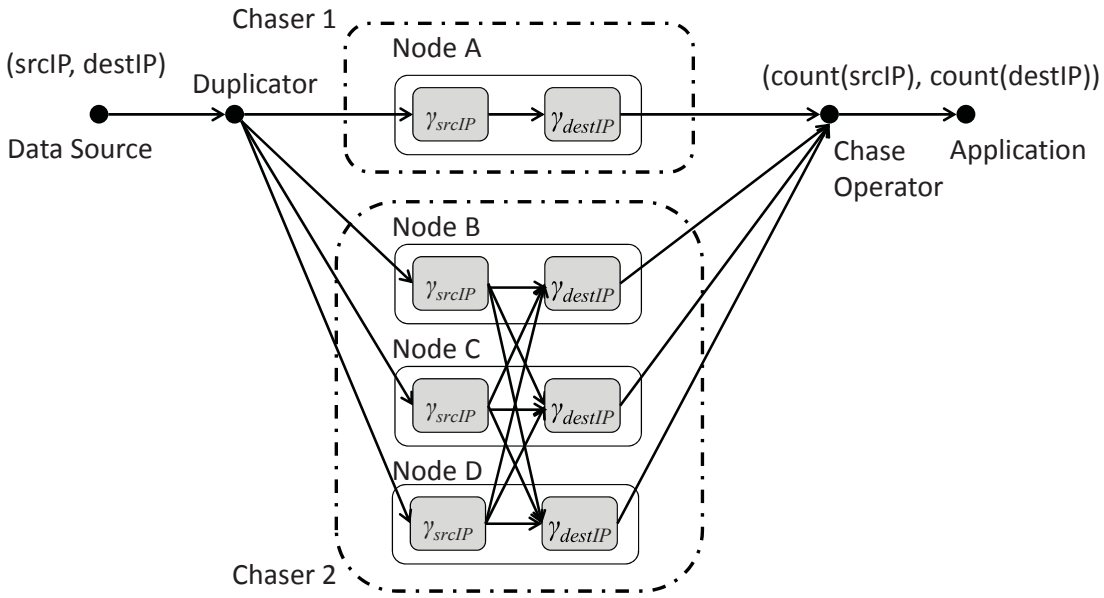

Figure 5.4 Chase Execution of the Example Query

redundant tuples using a variable n to hold the smallest tuple ID among the tuples that have not yet arrived at the Chase Operator, as well as a set S to hold the IDs that have arrived too early. As long as the processing proceeds normally in at least one of the chasers, the size of S will not continue to grow even when processing delays arise among the chasers, since the members of S that are no longer needed will be discarded.

The tuples that are output by the Chase Operator are a mixture of outputs from multiple chasers. Thus, there is a possibility that the result will be different from that output by a single computation node. For instance, a join operator, which has two or more input queues and a window to delimit the data stream, may produce different results depending on the order of acquisition from the multiple input queues. Results will also differ if an operator uses a non-deterministic algorithm.

Thus, applications using a Chase Operator must tolerate mixed tuples and unguaranteed ordering. For applications that need guaranteed ordering, such as those involving event processing, the Chase Operator must use a buffer to arrange tuples in

Algorithm: Chase Operator

Variables
Q_{in}: Input queue
Q_{out}: Output queue
n: Minimum ID of tuple that is not arrive
S: A set of Tuple IDs that already arrived

Repeat until System Halt

```
1   τ ← Q_in
2   if id(τ) ≥ n and id(τ) ∉ S then
3       Q_out ← τ
4       if id(τ) = n then
5           n = n + 1
6           while ∃ i ∈ S, i = n then
7               S = S \ i
8               n = n + 1
9           end
10      else then
11          S = S ∪ id(τ)
12      end
13  end
```

Figure 5.5 Pseudo Code for Chase Operator

the proper order. Note that an IPS, as discussed earlier, can tolerate mixed ordering because it only needs to determine whether packet should be permitted. Thus, a reordering buffer is not used in the experimentation in this chapter.

5.2.2. Handling a Node Failure

In the proposed method, even if a computational node fails, the system can continue processing so long as there is more than one normal chaser with sufficient computational resources. When such a failure occurs, the chaser with the failed computational node will not function anymore and must be recovered. Operator recovery can be achieved using general methods such as internal state transfer and check pointing, and is beyond the scope of this thesis.

5.3. Performance Evaluation

This section presents a performance evaluation of the proposed method using QueueLinker. Though the evaluation experiment is run against data constructed specifically to test packet monitoring, the results are expected to generalize to all kinds of stream processing applications. The experiment was conducted on a Linux cluster using the example query described in 5.1.

5.3.1. Data for the Experimental Evaluation

The experiment used simulated packet data generated for a hypothetical 1,000 servers and 1,000 thousands clients. Each client selected a server on Zipf distribution and accessed that server. Each server accepted the request and sent a response to the client immediately. After the client accepted the response, it immediately started server selection again and sent a request to the newly selected server. The experiment measured performance by varying the number of packets from 10,000 to 100,000 per second. The packets contained only headers (no payloads). Thus, a simulated data stream of 100,000 packets per second for 9 minutes contained 54 million packets, totally about 3.6GB. The amount of data for a one-minute packet trace was about 7MB, which was easily transferred via a 1000Base-T Ethernet network. All the packets were processed by QueueLinker.

5.3.2. Experimental Environment

Table 5.1 shows the experimental environment. All computation nodes had the same computation resources and were connected to a same network switch. The purpose of the experiment is to test whether the proposed method can perform in a commodity

network environment. A single computation node acted as a virtual router, sending packets to QueueLinker and measuring the latency, that is, the time difference between sending a packet to QueueLinker and receiving the corresponding result tuple. This setup is same as depicted in Figure 5.1. The Chase Operator and Duplicator also run on the router node.

5.3.3. Experimental Results

To evaluate the performance of Chase Execution under dynamic fluctuation of the data arrival rate, the data source was configured to generate 10,000 tuples (packets) per second for the first 3 minutes, then 100,000 tuples per second for the next 3 minutes, and again 10,000 tuples per second for the last 3 minutes. Chase Execution used four computers as computation nodes: one computer for Chaser 1 and the other three for Chaser 2. To establish comparative references for Chase Execution, the query was first executed using just one computer and then again using three computers. Note that because these reference setups did not actually replicate internal states, they could not have coped with any node failures.

Figure 5.6 shows the average latency of the query. Although the proposed method has to replicate incoming packets to multiple chasers, its latency performance is the best among the three experimental settings. Figure 5.7 shows the cumulative percentage of latency when the arrival rate was 10,000 tuples per second. Figure 5.8 shows the result at 100,000 tuples per second. As discussed in 5.1.1, the performance of one computer was better than that of three computers when the arrival rate was small, and vice versa when the arrival rate was high. Chase Execution outperformed both of these configurations, and showed the lowest overall latency.

5.4. Summary

This chapter described a proposed backup method that achieves low latency for distributed processing of continuous queries. The primary advantage of this method is that backup chasers do not add significant latency to normal, failure-free function. The method executes a query on a set of different operator deployments, or “chasers”, and collects the fastest result tuples from these deployments. Experimental results show that this method can achieve lower latency than deployments for which achieving high availability is not an objective.

Possible directions for future work include investigation of possible bottlenecks in applications that require an ordering mechanism, and evaluation of the Chase Operator as a single point of failure, especially where more than two chasers are used. One

promising approach to these kinds of issues is the introduction of a backup mechanism to the upstream side of the system.

Table 5.1 Experimental Environment

CPU	Intel Xeon E5530 x2 (Total 16 Logical Cores)
Memory	16GB
OS	CentOS 5.5
Runtime	Java HotSpot 64-Bit Server VM (Java version 1.6.0_27)
Network	1000Base-T Ethernet

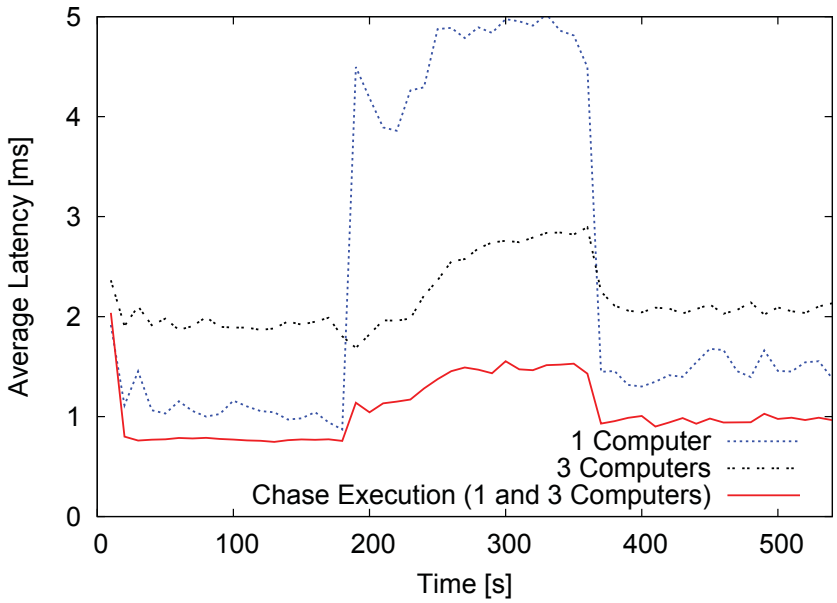

Figure 5.6 Average Latency with Data Rate Change

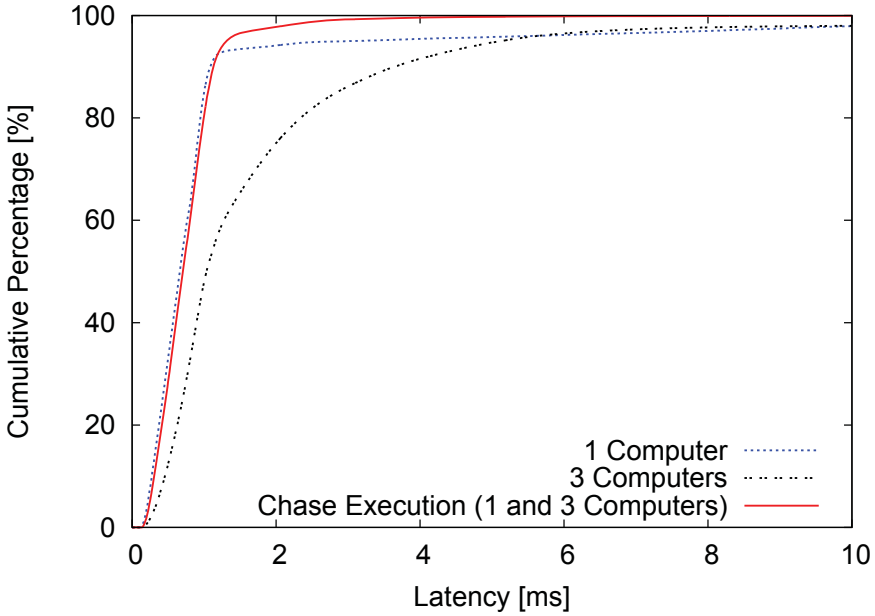

Figure 5.7 Cumulative Percentage of Latency (10,000 tuples / sec)

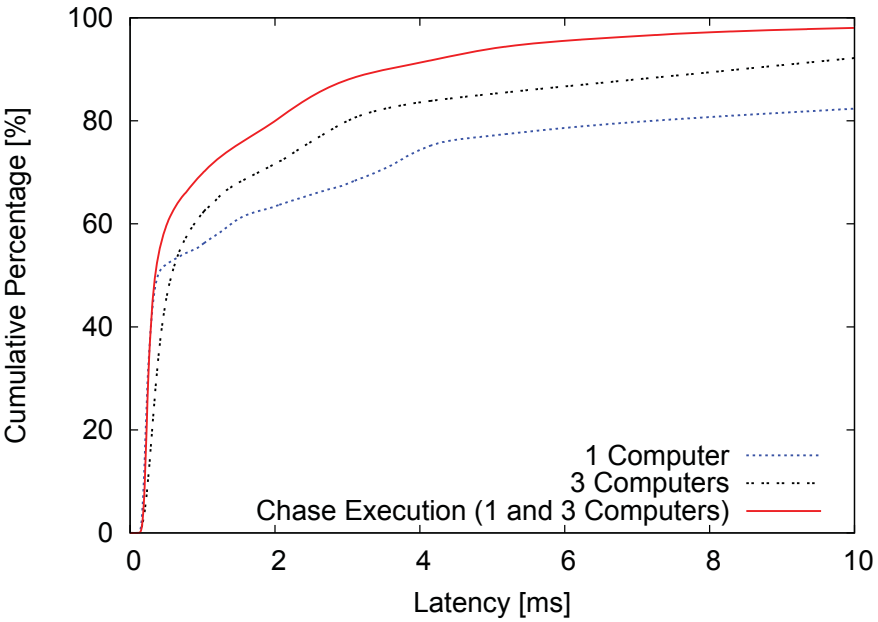

Figure 5.8 Cumulative Percentage of Latency (100,000 tuples / sec)

Chapter 6. A Parallel Distributed Web Crawler Consisting of QueueLinker Modules

This chapter describes a parallel distributed Web crawler implemented using QueueLinker. Web crawlers collect Web data while performing tasks such as detecting crawled URLs and preventing consecutive access to a certain Web server. Parallel and distributed crawling is required to realize high-speed crawling for the massive number of URLs that exist on the Web.

QueueLinker can help build a parallel distributed Web crawler. The proposed Web crawler consists of QueueLinker modules, and its tasks are distributed to computers by QueueLinker. The Web crawler consists of fine-grained modules, allowing it to realize better load balancing and memory utilization between computers than traditional site-based Web crawlers. Moreover, it becomes possible to crawl the Web on a large scale while conserving resources, because each module is implemented by data structures that are temporally and spatially efficient.

6.1. Background

Web crawlers collect Web pages from the Internet, performing an indispensable task for research, services, and businesses. A Web crawler starts crawling from a given seed page, and follows hyperlinks from one page to another in order to gather Web pages, images, and videos. High-speed crawling is important to quickly gather the large amount of data on the Web. However, intensively accessing the same Web server causes it to become overloaded. In a 2012 report [46], it was stated that, although Web crawlers represent only 6.68% of access to a Web server, they are responsible for 31.76% of its computational load. This is because the access pattern used by a Web crawler is different from that of a human. Thus, Web crawlers should be polite as they crawl the Web.

There are many Web servers and URLs on the Internet. For example, Google reports that there are over a trillion URLs[1]. Thus, Web crawlers manage a large number of URLs as they collect Web data, performing concurrent tasks such as detecting crawled URLs and preventing consecutive access to the same Web server. Parallel distributed Web crawling is expected to allow us to gather more pages, but the load balancing and scalable distributed crawling necessary to construct such a crawler are

[1] We knew the web was big..., http://googleblog.blogspot.jp/2008/07/we-knew-web-was-big.html

difficult issues. In addition, it is necessary to decrease the time and space complexity in order to reduce the number of computers required for crawling. Polite crawling, scalable distributed crawling, and resource saving are all important factors for incorporation in a parallel distributed Web crawler. In particular, parallel distributed computing mechanisms are tightly correlated to algorithms and data structures. Thus, we must discuss the whole software architecture of Web crawlers. Although commercial Web crawlers work very well, their technical details are not published. Therefore, a parallel distributed Web crawler is still worthy of discussion in research literature.

In existing research papers on distributed crawling, each computer executes a crawler, and all of the URLs of a Web site are assigned to a certain computer. This site-based distribution method easily realizes parallel distributed Web crawling. However, the number of URLs on a Web site varies, and thus a site such as Twitter, which has a large number of URLs, may cause a particular computer to become overloaded.

The proposed Web crawler consists of QueueLinker modules. It realizes polite crawling by ensuring that access to a certain Web server does not occur more than once in a given interval. Every module is designed along the data parallel model of QueueLinker, and thus every module can run on any number of computers and any number of threads. In other words, the crawler can assign any computational resources to each module independently. In addition, the crawler uses data structures that are temporally and spatially efficient, which allows us to crawl a large number of URLs with a small amount of computational resources. Another positive effect of the QueueLinker model is that it enables us to analyze Web data in real-time using the flow of data between modules. We can also easily customize the crawler by changing the module implementation.

6.2. Requirements for Web Crawlers and Existing Crawlers

This section describes the requirements of a Web crawler. Many existing papers on Web crawlers [16,47,48,49,50] discuss their scalability and crawling manner. To crawl the large number of URLs on the Web, crawlers must be scalable and able to crawl more URLs by simply adding crawler nodes. Moreover, crawlers must be polite to each Web server. In addition, some papers discuss re-crawling, fault tolerance, ease of configuration, and ease of customization for each purpose. This thesis does not consider these aspects, but such functions could be implemented by adding new modules to the proposed crawler. Thus, this chapter describes the proposed Web crawler from the standpoint of polite crawling and scalability. Firstly, this section discusses the following

aspects of politeness and scalability.

1. Requirements for polite crawling
   (a) Ensuring access intervals to a certain Web server are sufficiently long
   (b) Crawling pages by following instructions in 'robots.txt' and 'meta tags'
   (c) Not frequently accessing same Web pages
   (d) Reducing the load on outside DNS servers
   (e) Handling errors when accessing Web servers
2. Requirements for scalability
   (a) High-speed parallel distributed Web crawling with load balancing
   (b) Crawling with small computational resources
   (c) Priority crawling on good quality pages

6.2.1. Crawling Politeness

A Web page usually has hyperlinks to pages on the same Web server. If the crawler accesses the destination page given by these links without any interval, the Web server may be overloaded. As described above, 31.76% of the computational load of a Web server derives from Web crawlers [46]. To accomplish polite Web crawling, pages on the same server should only be accessed after a sufficient interval.

A Web server can give instructions to Web crawlers that access it by placing a robots.txt file in the top directory of the Web site. Using the 'disallow' description, robots.txt can prevent Web crawlers from accessing all or part of the Web site. According to some research papers [51,52], over 30% of Web servers have a robots.txt file, which means that Web crawlers should consider this file to realize polite crawling. Moreover, robots.txt may have a 'Crawl-delay' description that specifies the shortest interval for re-accessing the server. Some commercial crawlers do not consider Crawl-delay [46]. However, because the proposed crawler is for research purposes, and does not offer the benefits to Web sites that commercial Web crawlers do, the crawler proposed in this chapter should take account of Crawl-delay to make it acceptable to Web site administrators.

In addition, accessing the same page not only causes an increased load for the server, but also degrades the performance of the crawler. The large number of URLs on the Web makes it difficult to manage the crawled URLs. After the crawler resolves the IP address of a host, it should cache the address for a given period to reduce the overhead of name resolution and the load for external DNS servers. The cache must use a space-efficient data structure to manage a large number of URLs. Some limitation on

the number of retries is also required to avoid continuous access attempts when an error occurs.

6.2.2. Scalability

Realizing high-speed crawling with politeness and scalability is a challenging problem in terms of a suitable data structure. We should choose carefully between memory- and storage-based data structures. Papers on IRLbot [49,50] point out that previous storage-based structures had scalability problems due to storage access, and thus they proposed DRUM, an efficient batch-based processing mechanism. Storing data in physical memory allows crawlers to easily achieve scalability. However, the crawler must discard URLs or other data when there is no available memory space.

Parallel crawling must also gather Web data efficiently. The response time and throughput of Web servers varies. If the crawler does not perform parallel crawling, it has to wait until each download is completed, and thus slow Web servers degrade the crawling performance. In addition, distributed crawling should achieve better performance, because parsing Web pages and detecting crawled URLs require a large amount of computation time and space. Thus, a parallel distributed system is indispensable for high-performance Web crawling.

The method and flexibility of computational resource assignment is also important in parallel distributed Web crawling. In research papers, a full-set crawler is executed on each computer, and all URLs on a Web site are assigned to a certain computer. This site-based distribution method easily realizes parallel distributed Web crawling. However, the number of URLs on a Web site varies, and thus a Web site such as Twitter, which has a large number of URLs, may cause certain computers to be overloaded.

Finally, the crawler performance is evaluated in terms of the download speed and the quality of pages that the crawler traverses. There are a large number of spam Web pages that may cause large amounts of traffic, and these can be obstacles to gathering good quality pages.

6.2.3. Download Speed of Published Crawlers

The development of Web crawlers is a research topic that has produced a number of papers [16,17,18,49,50,53,54]. For example, Mercator [54] was reported to crawl 112 pages/s using only one computer. A report on Mercator [16] in 2001 discusses an attempt at distributed crawling with four computers. Several papers have described distributed Web crawlers. In 2002, Ubicrawler [17] crawled 10 million pages/day (116 pages/s), and

another crawler was reported to have crawled 140 pages/s. IRLbot [49,50] is the fastest of the published Web crawlers that use only one computer, reaching a performance level of 1,789 pages/s. A research paper on WebBase [18], which was developed at Stanford University, argues that it can gather up to 6,000 pages/s with 40 processes.

6.3. Characterization of the Proposed Web Crawler

Previous distributed Web crawlers assign each Web site to be crawled to a certain computer. This site-based distribution approach causes computers assigned Web sites with a large number of URLs to be overloaded. In addition, resource consumption varies according to the functions of a crawler. Thus, an efficient crawler should be organized into fine-grained modules, enabling it to assign computing resources more flexibly. Previous research papers on distributed Web crawlers have not considered the problem of resource assignment. To solve these problems, the proposed Web crawler consists of QueueLinker modules.

Table 6.1 shows a solution for the Web crawler requirements described in Section 6.2. The proposed crawler stores data structures in the main memory. The data space of each module can be divided using the hash values of host names, URLs, or IP addresses. The modules do not share their internal states. Thus, every module can be executed according to the data parallel model of QueueLinker, and can run on any number of threads and computers. Therefore, the proposed system can assign computational resources to each crawler function more flexibly than existing site-based crawlers.

Figure 6.1 explains the benefit of the load balancing offered by the proposed crawler. The figure compares the proposed Web crawler with a site-based Web crawler in terms of load balancing the DNS cache module and duplicated URL detection module. The site-based Web crawler assigns each Web site to a certain computer, which causes a bias in the number of URLs processed among the computers, as shown in (1a) and (1b), because the number of URLs on a Web site varies. The DNS cache module keeps the Web site IP addresses, and thus there is no bias in memory consumption between (1a) and (1b). On the other hand, a duplicated URL detection module keeps URLs that the crawler has already visited, resulting in the load bias between (1a) and (1b). The proposed crawler executes the DNS cache task and duplicate detection task in different modules. Thus, the proposed crawler can realize load balancing in the duplicate detection task, because the URLs to be processed by the module are partitioned by the hash value of each URL. Load balancing is an important factor for Web crawlers, because several Web sites have a large number of URLs.

Moreover, the proposed crawler uses a Bloom Filter [55] and HAT-trie [56] to

reduce the space complexity of data. A Bloom Filter is a space-efficient probabilistic data structure that can determine whether an element is already contained in a set with a certain false positive. HAT-trie is a trie data structure that stores a set of strings within a small data space. Additionally, another positive effect of the QueueLinker model is that it enables us to analyze Web data in real-time using the flow of data between modules. Thus, the crawler can be used as the backend of applications like [57]. QueueLinker offers a simple way to add new modules and change the connections between modules.

Previous papers discuss other aspects, including re-crawling, fault tolerance, and ease of configuration. These aspects are important in actual operation. This thesis does not consider these aspects, but existing re-crawling methods could be implemented by adding new modules to the proposed crawler.

Table 6.1 Solutions for the Requirements of Web Crawlers

Requirements	Solutions
(1a) Ensuring Access Intervals	Crawling scheduler guarantees that the minimum access interval to a certain Web server is always longer than a given value.
(1b) Dealing with robots.txt and meta tag	The crawler can analyze them.
(1c) Duplicated URL Detection	Duplicated URLs are filtered using LRU cache and a Bloom Filter.
(1d) Reducing DNS Overhead	The crawler caches IP addresses of Web sites using HAT-trie.
(1e) Error Handling	The crawler suspends crawling to error pages.
(2a) Parallel Distributed Web Crawling	QueueLinker can automatically execute modules in a parallel distributed way.
(2b) Reducing Memory Consumption	The crawler uses a Bloom Filter and HAT-trie, which are space efficient data structures.
(2c) Crawling High Quality Pages	The Scheduler can gather high PageRank pages.

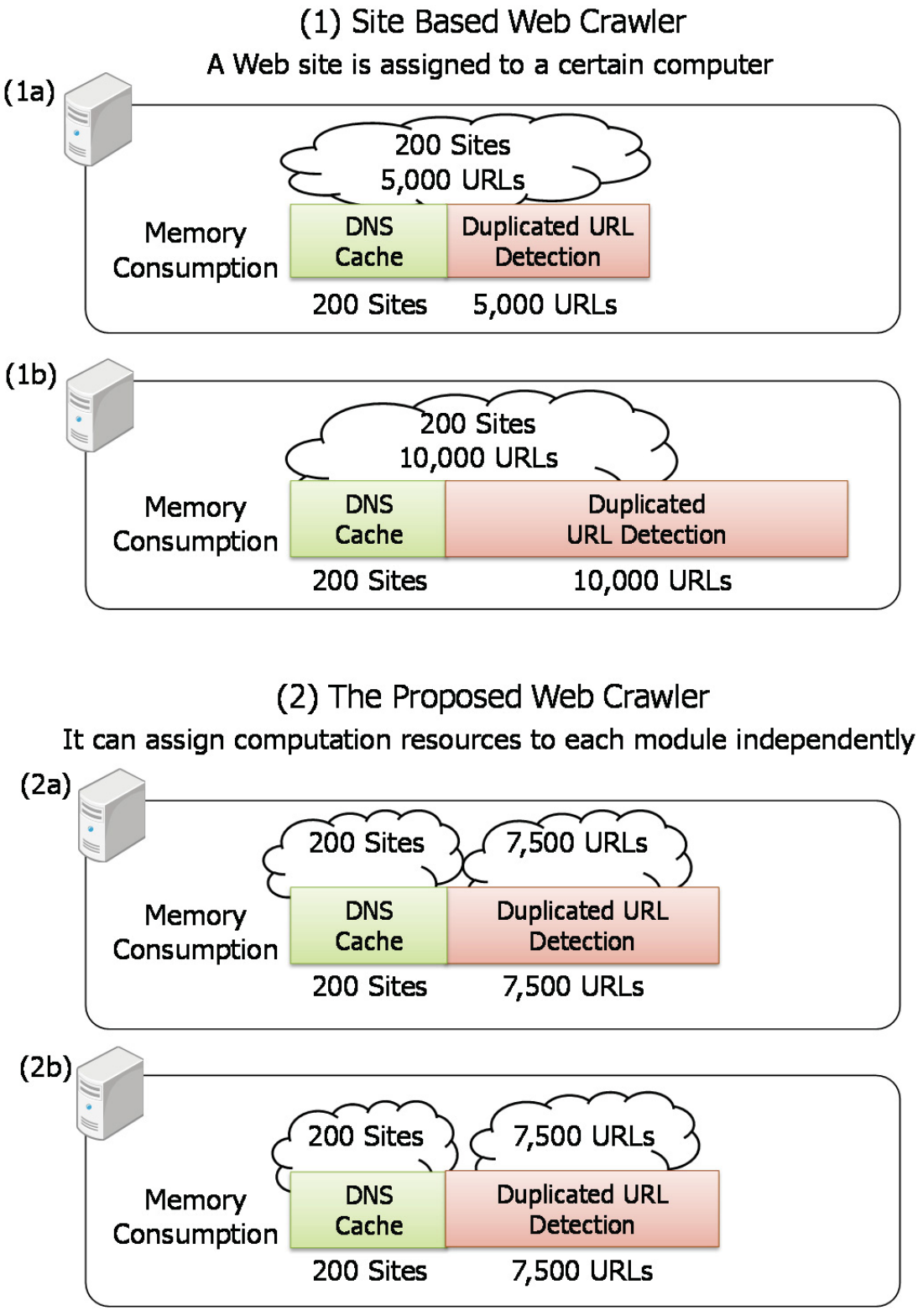

Figure 6.1 Load Balancing Benefit of the Proposed Crawler

6.4. Execution Model of the Proposed Web Crawler

This section describes the execution model of the proposed Web crawler. The proposed crawler is implemented by QueueLinker. Thus, QueueLinker automatically realizes parallel distributed Web crawling. This section describes the execution model. The implementation details of each module are described in Section 6.5.

6.4.1. Data Structure for Crawling

Table 6.2 shows the data structure that is transferred between modules of the crawler. QueueLinker automatically transfers the data structure between modules. In the following explanation, this structure is referred to as CrawlingData. A URL found during crawling is managed by an instance of CrawlingData. Each module accepts CrawlingData, performs a process, and then outputs CrawlingData. Thus, CrawlingData can be modified if needed. For example, the downloading module accepts CrawlingData, and then downloads the Web page indicated by its 'url' field. The downloading module then stores the downloaded data in the 'data' field of CrawlingData, and finally outputs the modified CrawlingData.

6.4.2. Hash Functions

As described in Chapter 3, QueueLinker provides a hash partitioning technique to realize data parallel execution. The data space of every module of the crawler can be

Table 6.2 Main Components of CrawlingData, a Data Structure for Communication between Modules

Type	Name	Role
String	url	A URL to be processed.
byte[ ]	ipAddress	An IP address of the Web sever of the URL
byte[ ]	data	Downloaded data
int	robotsFlag	Robots.txt Existence (0: Does Not Exist, 1: Exists, 2: Unknown)
boolean	downloadable	If true, the page can be downloaded. Otherwise, not downloadable. This flag is affected by the robots.txt or the meta tag in the Web page.
int	crawlDelay	Crawl-delay of robots.txt of the Web site. Default value is 5 seconds.
Exception	exception	Exception occurred when downloading

hash partitioned, and the following hash functions are used to determine a partition.

- $h(\mathrm{Host})$: A function calculating the hash value of a host name
  - Used by host cache module and robots.txt processor module
- $h(\mathrm{URL})$: A function calculating the hash value of a URL
  - Used by duplicated URL detection module
- $h(\mathrm{IP})$: A function calculating the hash value of an IP address
  - Used by crawling scheduler module

For example, QueueLinker always transfers URLs with the same hash value $h(\mathrm{Host})$ to the same instance of the host data cache module that caches each IP address corresponding to a host. The cache works correctly if and only if URLs with the same hash value are always processed by the same instance. The duplicated URL detection module also correctly detects already-crawled URLs if and only if those with the same hash value $h(\mathrm{URL})$ are always processed by the same instance. Each process is completed inside an instance of each module, and does not require access to any other instances. Therefore, every module can run on any number of threads and any number of computers. This is the benefit of the data parallel model of QueueLinker.

6.5. Implementation of Web Crawler Modules

This section describes the implementation of each module in the proposed crawler using the QueueLinker model described in Chapter 3. The modules listed in Table 6.3 are connected to each other as shown in Figure 6.2. Each rectangle in the figure represents a QueueLinker module, and the colored rectangles represent virtual modules. In the figure, queue names are shown beside modules that have multiple input queues.

The crawler uses a crawling scheduler whose time complexity is O(1). The crawling scheduler ensures that the access intervals for a certain Web server are always longer than a specified time interval. The scheduler can place high priority on good quality pages, and the total PageRank value of crawled pages is higher than that of breadth-first order crawling for the same number of crawled pages. The crawler interprets robots.txt and meta tags in HTML. The results of DNS resolution are cached using HAT-trie. The LRU cache and Bloom Filter are used to detect duplicate URLs, allowing the crawler to detect crawled URLs efficiently. HAT-trie and the Bloom Filter are spatially efficient data structures, which mean the crawler can gather a large number of Web pages with a small amount of computational resources.

6.5.1. Overview of the Modules and Strategy of Distributed Crawling

Each module of the crawler can be grouped into one of two types according to whether the module has internal states. The distribution strategy of the modules that do not have an internal state is 'Free,' as shown in Table 6.3. For example, '(d) Downloader' just downloads data and outputs the downloaded data, and this does not require any internal states inside the module.

The distribution strategy of modules that have internal states is one of 'IP,' 'Host,' or 'URL.' As described in Chapter 3, QueueLinker selects which instance of a module to transfer CrawlingData to based on its hash value. For example, CrawlingData must be transferred to an instance of '(a) Scheduler' depending on the value of $h(\mathrm{IP})$, because the module realizes the scheduling by grouping URLs according to their IP addresses. As another example, '(i) Host Data Cache' caches IP addresses of Web sites, and CrawlingData must be transferred depending on the value of $h(\mathrm{Host})$. '(k) robots.txt Processor' also uses the 'Host' strategy, because robots.txt exists in the top directory of each Web site. '(h) Duplicated URL Checker' detects already crawled URLs, and thus an instance must be chosen according to the value of $h(\mathrm{URL})$. Note that the strategy of '(m) Seeder' is 'Single,' because it must be executed only once whilst crawling to avoid

Table 6.3 List of Modules Composing the Proposed Crawler

Module Name	Distribution Strategy	Number of Input Queues
(a) Scheduler	IP	2
(b) Scheduler Timer	Free	1
(c) robots.txt Downloader	Free	1
(d) Downloader	Free	1
(e) HTML Parser	Free	1
(f) URL Format Filter	Free	1
(g) Explicit URL Filter	Free	1
(h) Duplicated URL Checker	URL	1
(i) Host Data Cache	Host	4
(j) Domain Name Resolver	Free	1
(k) robots.txt Processor	Host	2
(l) Data Store	Free	1
(m) Seeder	Single	0

seeding URLs multiple times.

6.5.2. Crawling Scheduler Modules

The crawling scheduler is the most important part of the crawler, affecting the performance and the politeness of the Web crawler [58,59]. This scheduler ensures that the access intervals on a certain Web site are always longer than a given time interval. The time complexity of the scheduler is O(1). In this thesis, the minimum interval is set to 5 s. The scheduler consists of two modules, '(a) Scheduler' and '(b) Scheduler Timer,' as shown in Table 6.3

The 'Scheduler' module has two input queues, q_0^a and q_1^a. During crawling, queue q_0^a receives CrawlingData containing a newly found URL. Queue q_1^a receives CrawlingData containing URLs whose download procedure has been completed or failed for some reason. In addition, as shown in Figure 6.3, the scheduler has L lists $l_0, l_1, \ldots, l_{L-1}$ containing URLs to be downloaded. These lists do not contain multiple URLs from a single Web server. The scheduler controls the crawler so as to always download URLs belonging to only one list. After downloading a list, the crawler waits

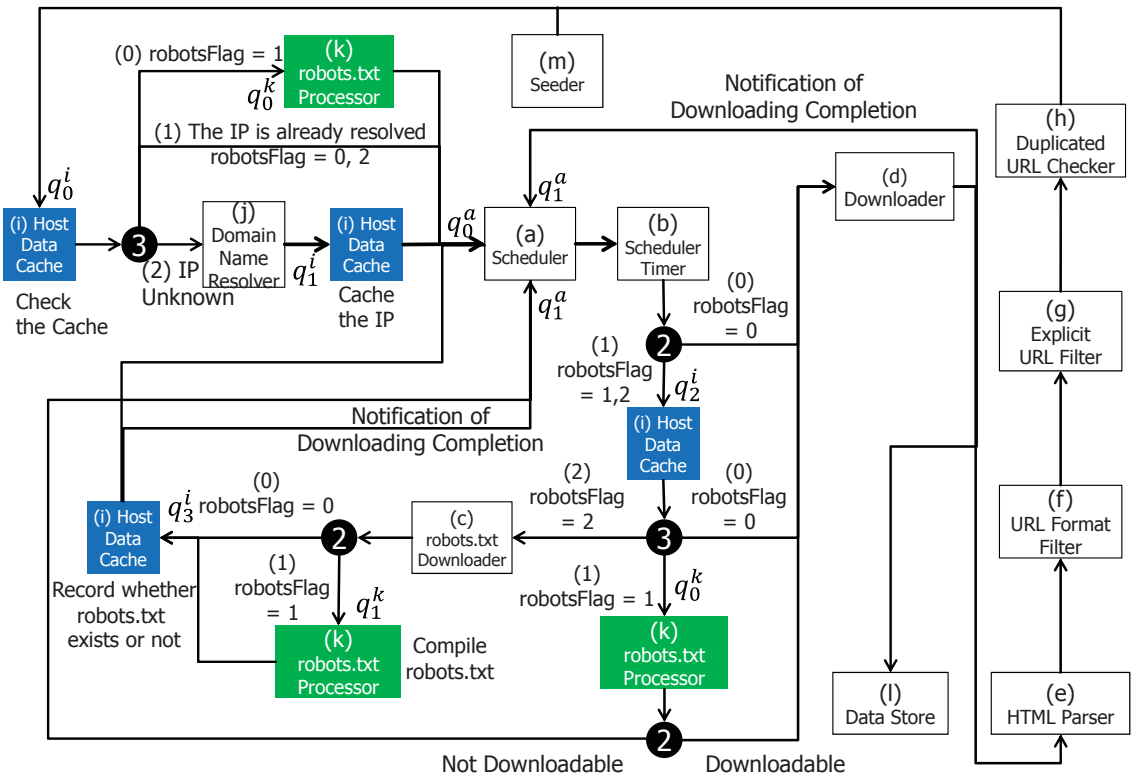

Figure 6.2 The Logical Directed Graph of the Proposed Crawler. The Black Circles Represent Switchers

for 5 s before it starts to download URLs in the next list, which ensures the crawling interval for a given Web server is always longer than 5 s.

The scheduler algorithm is shown in Figure 6.4. After the scheduler receives a URL via q_0^a, it chooses which list to add the URL to by referencing an integer array $(n_0, n_1, \ldots, n_{\mathrm{H}-1})$. A list $l_{n_i(mod\,\mathrm{L})}$ is the candidate for the URL, where i is the remainder of the division of the IP address by H, where the IP address corresponding to the URL is considered as an integer. Although different IP addresses may have the same remainder, the lists do not have multiple URLs from a single Web server, even if such collisions are neglected. In other words, the scheduler stores the URLs of a certain Web server in lists $\{l_{a_m} | 1 \leq m \leq \mathrm{L}-1\}$ in order of increasing m, where $a_m \equiv c + m \pmod{\mathrm{L}}$ and l_c is the list that is currently being crawled. A new URL is discarded when there is no available list in which to store the new URL. However, if the discarded page is important, it is expected to appear again during the crawl, because important Web pages tend to be linked from many other pages. Note that the scheduler has to store URLs with an extra interval to adhere to the given interval when the robots.txt file defines 'Crawl-delay' (lines 15 and 21 in Figure 6.4). If a Web site does not use 'Crawl-delay', the 'crawlDelay' field of CrawlingData has the default value 5 s, and site URLs are successively stored in the list.

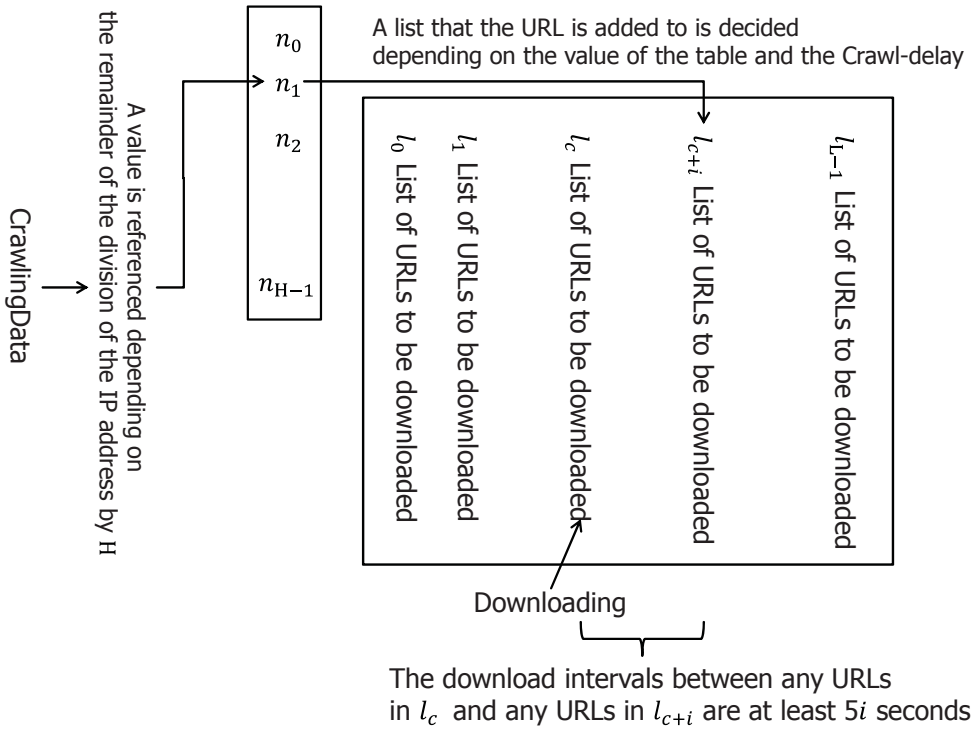

Figure 6.3 Conceptual Diagram of the Scheduler

Algorithm 1 (a) Scheduler

Lists of URLs to be downloaded: $l_0, l_1, ..., l_{\mathrm{L}-1}$
Integers: $n_0, n_1, ..., n_{\mathrm{H}-1}$ (initialized by 0)
Current step: c (initialized by -1)
The number of URLs currently downloaded: p (initialized by 0)

```
 1: procedure SCHEDULER(data, id)
 2:     if id = 0 then
 3:         NewURL(data)                          ▷ Process an input from q_0^a
 4:     else if id = 1 then
 5:         p = p − 1                             ▷ Process an input from q_1^a
 6:     end if
 7:     if p = 0 then
 8:         return GoNext()
 9:     end if
10: end procedure
11:
12: procedure NEWURL(data)
13:     r ≡ 32bit unsigned integer of data.ipAddress(modH)
14:     if n_r ≤ c then
15:         n_r = c + ⌈data.crawlDelay/5⌉
16:     end if
17:     if n_r ≥ c + L then
18:         return null                           ▷ Discard the URL
19:     end if
20:     i ≡ n_r(modL)
21:     n_r = n_r + ⌈data.crawlDelay/5⌉
22:     l_i = l_i ∪ data
23: end procedure
24:
25: procedure GONEXT
26:     m = 1
27:     while m ≤ L − 1 do
28:         c = c + 1
29:         i ≡ c(modL)
30:         if |l_i| ≠ 0 then
31:             l = l_i
32:             l_i = φ
33:             p = |l|
34:             return l
35:         end if
36:         sleep(5 seconds)                      ▷ Suspend for 5 seconds
37:         m = m + 1
38:     end while
39:     return null                               ▷ There is no URL to be downloaded
40: end procedure
```

Figure 6.4 Pseudo-Code of the Scheduler Module

The scheduler can determine the number of URLs that have been crawled by counting those arriving at q_1^a. When all of the URLs in l_c have been crawled, the scheduler outputs the next list. More precisely, it selects a list from $\{l_{a_m} | 1 \leq m \leq \mathrm{L} - 1\}$, where m is the smallest number such that the chosen list has at least one URL. After the scheduler outputs a list, the Scheduler Timer receives the list and waits for 5 s before it is output. The pseudo-code for Scheduler Timer is shown in Figure 6.5. This procedure ensures the crawling intervals for a certain Web server are always longer than 5 s. Note that, even if the list is empty, the scheduler has to wait in order to ensure the minimum interval (line 36 in Figure 6.4). This line is rarely executed, because URLs of Web sites that have no 'Crawl-delay' definition almost always fill the lists without gaps.

The proposed scheduler has the drawback that no downloading is performed while a list waits in the Scheduler Timer. In addition, URLs of slow Web servers can cause a bottleneck, because the scheduler does not output a new list until all of the

Algorithm 2 (b) Scheduler Timer

```
1: procedure SCHEDULERTIMER(data:List<CrawlingData>)
2:     sleep(5 seconds)                          ▷ Suspend for 5 seconds
3:     return data.toArray()                     ▷ Output each CrawlingData
4: end procedure
```

Figure 6.5 Pseudo-Code of Scheduler Timer

Algorithm 3 (h) Duplicated URL Checker

```
1: procedure DUPLICATEDURLCHECKER(data)
2:     if data.url is contained in the LRU list then
3:         Move data.url in the LRU list to the head of the list
4:         return null                           ▷ No crawl the URL
5:     end if
6:     Add data.url to the head of the LRU list
7:     if The LRU list overflows then
8:         Remove the URL in the tail of the LRU list and add it to BloomFilter
9:     end if
10:    if Bloom Filter contains data.url then
11:        return null                           ▷ No crawl the URL
12:    else
13:        return data                           ▷ Crawl the URL
14:    end if
15: end procedure
```

Figure 6.6 Pseudo-Code of Duplicated URL Checker

URLs in the current list have downloaded. However, these drawbacks can be resolved by increasing the number of parallel executions of the scheduler, because this increases the number of CrawlingData instances that are being downloaded while several instances are waiting in the Scheduler Timer.

Our scheduler places a high priority on good quality pages, and the total PageRank value of crawled pages is higher than that of breadth-first order crawling for the same number of crawled pages. Although RankMass [60], a previous method, may be faster at collecting high quality pages, the proposed scheduler has a better time complexity than that of RankMass. This is because RankMass calculates the estimated PageRank values while crawling, and it has to access the link structure in order to estimate the PageRank.

The time complexity of the proposed scheduler is O(1). The scheduler can be distributed by a hash partitioning technique with the help of QueueLinker, and the hash value of each IP address is used to determine which instance processes each URL.

6.5.3. Duplicated URL Detection Module

The '(h) Duplicated URL Checker' detects duplicated URLs using the LRU cache and Bloom Filter. It is known that cache algorithms can be used to detect duplicated URLs efficiently [61]. The proposed crawler uses an LRU cache as the first-level cache for detecting duplicated URLs. A Bloom Filter is used as the second-level cache to store evicted URLs from the LRU cache. This is necessary because of the size limitation of the LRU cache. Bloom Filters are space-efficient probabilistic data structures that determine whether an element is contained in a set while allowing false positives. Thus, the crawler can detect already-crawled URLs because the Bloom Filter does not cause false negatives, whereas the crawler mistakenly assumes a URL has already been crawled if a false positive occurs.

The algorithm for this module is shown in Figure 6.6. First, it checks the LRU cache. If the LRU cache does not contain the URL, it checks the Bloom Filter. If the URL is contained in either data structure, the module discards the URL as not to be crawled. Otherwise, if neither data structure contains the URL, the module outputs the URL for future crawling. When a URL overflows from the LRU cache, the URL is inserted into the Bloom Filter, and thus the duplication of the URL will be correctly checked without error.

6.5.4. Host Data Cache Module

To avoid the overheads associated with name-resolution and accessing robots.txt

multiple times, the crawler must store a flag denoting the presence or absence of robots.txt and IP addresses corresponding to host names. The '(i) Host Data Cache' module uses a modified version of the original HAT-trie [56] in order to store key-value data. In this case, the key is a host name and the value is an IP address and a flag denoting the presence or absence of robots.txt.

Figure 6.7 illustrates the performance of the HAT-trie key-value storage. This experiment was carried out on the assumption that the structure is used to cache IP addresses of Web sites. Host names were inserted as keys, and a 4-byte integer corresponding to an IP address was inserted as a value for each key. The experiment used 590 million host names taken from published data[1]. The total data size of the host names was about 16 GB. The total size of the inserted data was about 18 GB, because a 4-byte integer was inserted for each host name. The memory consumption and throughput of inserting key-value data were measured from when the data structure was empty, at the beginning. After the data had been inserted, the throughput of searching for values was measured for all the host names that had been inserted in the structure.

As shown in Figure 6.7, the memory consumption was almost the same as the

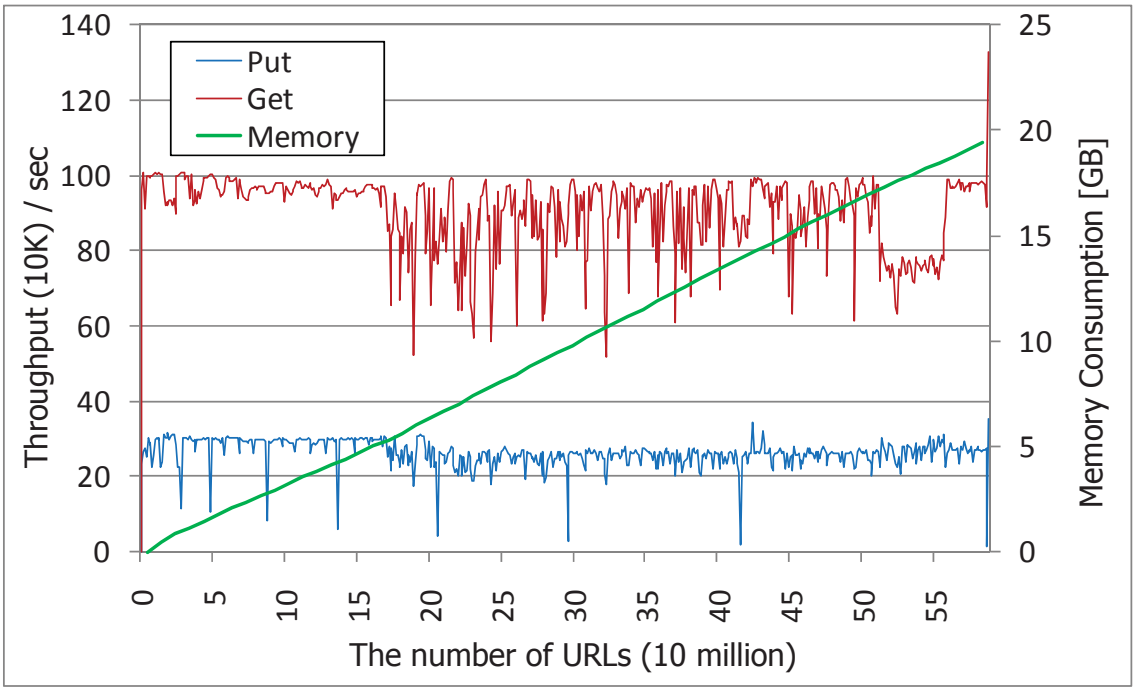

Figure 6.7 Performance of HAT-trie for Storing and Searching Host Names

[1] Laboratory for Web Algorithmics, http://law.di.unimi.it/index.php

total size of the inserted data. The trie data structure compresses common URL prefixes, and this acts to reduce the memory consumption. The memory consumption is one-fifth of that of HashMap in the Java API. The peak throughput of inserting data is over 300,000/s and the peak throughput of searching is over 1,000,000/s.

The Host Data Cache module uses four input queues $q_0^i, q_1^i, q_2^i, q_3^i$. Queue q_0^i receives CrawlingData from the Duplicated URL Checker module, and this contains a URL that has not been crawled. This module searches for a value including the IP address from the HAT-trie. The module then stores the IP address in the 'ipAddress' field of CrawlingData. The 'ipAddress' field is set to 'null' if HAT-trie does not have the IP address of the Web server. In addition, this module also checks whether HAT-trie caches the presence or absence of robots.txt on the site and sets the 'robotsFlag' in CrawlingData. The switcher connected to this module transfers CrawlingData according to the status of the 'ipAddress' and 'robotsFlag' fields. The switcher transfers the data to the Domain Name Resolver in order to resolve the IP address when 'ipAddress' is null. The switcher transfers the data to the robots.txt Processor module to check whether the URL can be downloaded when the 'robotsFlag' indicates the site contains a robots.txt file. Otherwise, the data is transferred to the Scheduler module so that it can be scheduled for future crawling.

The remaining input queues q_1^i, q_2^i, q_3^i are used in the virtual modules of this module. The input queue q_1^i receives CrawlingData from the Domain Name Resolver. This contains the resolved IP address of the site. In this case, the module inserts the IP address in HAT-trie and simply outputs the data. When a URL from the same site appears, the inserted IP address will be used to avoid name resolution. The output is transferred to the input queue q_0^a of the Scheduler module, and will thus be crawled during a future crawling.

The input queue q_2^i receives CrawlingData from the Scheduler module. This means the URL contained in the CrawlingData is now going to be crawled. This module checks again whether HAT-trie has cached the presence or absence of robots.txt on the site. This check is required because there is a possibility that other URLs from the site may check the existence of 'robots.txt' while the URL is waiting to be crawled in the Scheduler. For example, two URLs for a newly found Web site can be added to lists l_{c+1} and l_{c+2} in the Scheduler module when URLs in l_c are being downloaded. In this case, when the URL in l_{c+1} is crawled, the crawler checks whether the robots.txt file exists on the server. Thus, the existence of robots.txt has been checked before the URL in l_{c+2} has begun to be crawled.

The input queue q_3^i receives CrawlingData containing a URL of a Web site that

has robots.txt. This module records the robots.txt existence in HAT-trie and simply outputs the data to the Scheduler module for future crawling.

6.5.5. robots.txt Processing Module

The '(k) robots.txt Processor' module has two input queues, q_0^k and q_1^k, and processes robots.txt. This module uses the Pattern class of the Java API to check whether URLs are allowed or disallowed by robots.txt.

When the input queue q_0^k receives CrawlingData, it checks whether the URL has been disallowed for crawling by the robots.txt file. This module sets the 'downloadable' flag of CrawlingData to false if robots.txt disallows crawling on the URL. Otherwise, this module sets the flag to true. The switcher connected to this module checks the 'downloadable' flag. If the URL is downloadable, CrawlingData is transferred to the Downloader module. If not, CrawlingData is transferred to the input queue q_1^a of the Scheduler module and discarded.

The input queue q_1^k is used in the virtual module of robots.txt Processor. It receives CrawlingData containing robots.txt from a Web site. When this module receives a newly found robots.txt, it generates Pattern instances of the rules in the robots.txt file.

6.5.6. Downloading Modules

This crawler has two modules for downloading. The '(c) robots.txt Downloader' module downloads robots.txt, and the '(d) Downloader' module downloads other Web data. Data indicated by a URL are downloaded after the modules receive CrawlingData. The data is stored in the 'data' field of CrawlingData. These modules use Apache HttpClient library[1] to download data.

If a download error occurs, the exception is stored in the 'exception' field of CrawlingData. In this case, the Switcher connected to this module transfers CrawlingData to the input queue q_1^a of the Scheduler in order to stop the page downloading.

6.5.7. Link Extraction Module and Filtering Modules

The '(e) HTML Parser' module analyzes HTML pages and extracts linked URLs from them. After the URLs are extracted, the '(f) URL Format Filter' module checks the format of the extracted URLs. The '(g) Explicit URL Filter' module checks each URL to determine whether it is marked as uncrawlable by the crawler user.

[1] HttpComponents HttpClient Overview, http://hc.apache.org/httpcomponents-client-ga/index.html

The HTML Parser module uses the jsoup library[1] to parse HTML pages, from which it extracts linked URLs. Web pages often include incorrect or illegally formatted URLs. Thus, URL Format Filter module discards URLs that are not included in the Public Suffix List[2], a list of worldwide domains. The Explicit URL Filter module then filters the URLs of sites marked as uncrawlable by the user. These uncrawlable sites include electronic journals such as ACM Digital Library and IEEE Xplorer, which prohibit the crawling of their sites. In addition, some Web sites whose administrators have complained about crawling may be marked as uncrawlable.

6.5.8. Storing Downloaded Data Module

The '(l) Data Store' module stores downloaded data in secondary storage. This module receives CrawlingData and serializes it into a BSON-format file. This module packs multiple Web data into a file, because large numbers of files are generated if the data is stored in separate files. The large number of files strongly reduces the read throughput due to random accesses to the files, resulting in difficulties analyzing the crawled data. To cope with file corruptions, this module limits the file size to 2 GB and creates a new file when this limit is reached.

6.5.9. URL Seed Module

The Web crawler requires a seed set of URLs as the starting point for crawling. The '(m) Seeder' module runs once at the beginning of a crawling, and supplies seeds to the Scheduler module. This thesis does not consider how to choose the seeds.

6.6. Performance Evaluation

This section describes the performance evaluation of the proposed crawler. In this evaluation, computers were placed in Waseda University and NII (National Institute of Informatics). A virtual Web space was built on the NII computers. Those in Waseda University accessed the virtual Web space. In this experiment, resource assignment to the modules was manually controlled. To evaluate the maximum throughput of the proposed crawler, the minimum interval for accessing a certain Web server was set to zero. Java 1.7 was used for this experiment.

6.6.1. Experimental Environment and Module Deployment

Figure 6.8 and Table 6.4 show the experimental environment. One master node

[1] jsoup Java HTML Parser, with best of DOM, CSS, and jquery, http://jsoup.org/
[2] Public Suffix List, http://publicsuffix.org/

and eight crawling nodes were placed in Waseda University. A virtual Web space was built on four computers in NII. The virtual Web space was built using the actual Web link structure of uk-2007-05[1], and virtual Web pages were created according to the structure. In this experiment, the microhttpd library[2], a lightweight Web server for embedded devices, was used as the Web server of the virtual space. Microhttpd can generate Web pages in parallel using multi-threads. The networks of Waseda University and NII were connected by SINET4[3], and the upstream network switches of the computer were connected to SINET4 by 10 Gbps lines.

In this experiment, the Scheduler module was assigned to the master node with the largest memory among the computers. The Data Store module was assigned to the storage node. Other modules were assigned to crawling nodes. Each module, except the Downloader module, was executed by one thread. The Downloader module on a crawling node was executed by 200 threads. To reduce the amount of network communication, Web pages that were downloaded by the Downloader module on a node were parsed by the HTML Parser module on the same node.

The above configuration allows the Scheduler module, which requires a large memory space to store URLs, to use 512 GB of the master node's memory space. At the same time, the Duplicated URL Checker, Host Data Cache, and robots.txt Processor, which also require a large amount of memory space to store data, can use a total of 128 GB of memory when eight crawling nodes are used. As described above, the crawler implementation using QueueLinker allows us to assign computational resources to the modules flexibly. This is one advantage of the proposed crawler.

6.6.2. Experimental Results

Figure 6.9–Figure 6.11 show the number of pages downloaded per second. The experiment was performed on 2, 4, and 8 crawling nodes. The figures show that the download speed increased with the number of crawling nodes. This result demonstrates the scalability of the crawler.

Figure 6.12 shows the number of URLs and Web sites that the crawling nodes handled when all of the pages in the virtual space were downloaded. The proposed Web crawler consists of fine-grained modules, and QueueLinker executes each module using a hash partitioning technique. As a result, the load bias between computers is reduced. There was a 0.30% difference between the maximum and minimum number of URLs handled by the crawling nodes, and a difference of 3.0 % in the case of Web sites.

---

[1] Laboratory for Web Algorithmics, http://law.di.unimi.it/index.php
[2] GNU libmicrohttpd, http://www.gnu.org/software/libmicrohttpd/
[3] SINET4: Science Information NETwork 4, http://www.sinet.ad.jp/index_en.html?lang=english

Figure 6.13 shows the simulated result of a site-based crawler that assigns all pages in each Web site to a certain computer. For Web sites, the result was similar to that in Figure 6.12. However, there is a load bias in the number of URLs handled, because the number of URLs on a Web site varies. For the site-based crawler, the difference between the maximum and minimum number of URLs handled by a computer was 12.4%. If the number of URLs that each Web site has is known prior to crawling, the site-based crawler can optimally assign Web sites to computers such that they handle almost the same number of URLs. However, in reality, it is difficult to adjust the Web site assignment, because such numbers of URLs cannot be achieved until the crawling has finished.

In contrast, the proposed Web crawler can distribute every process depending on

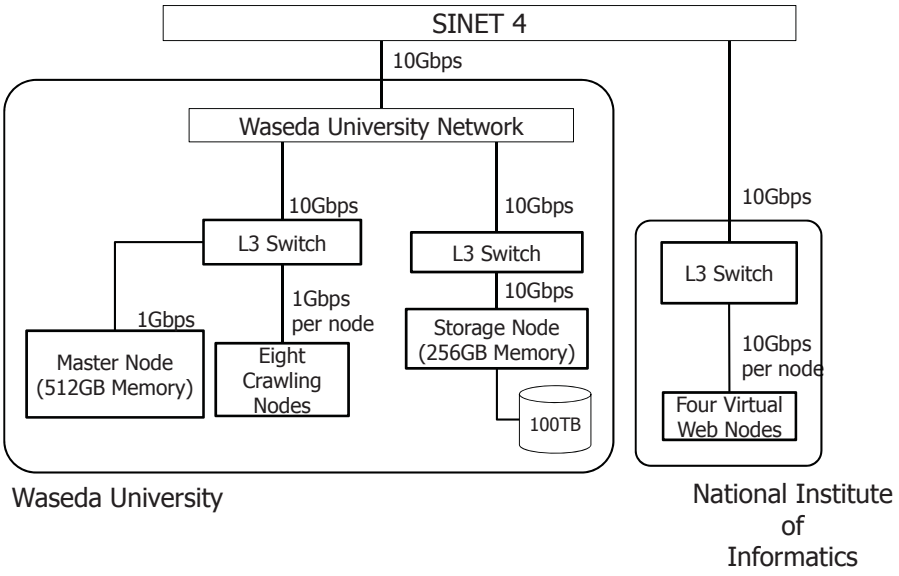

Figure 6.8 Experimental Environment

Table 6.4 Experimental Environment

	Master Node	Storage Node	Crawling Nodes	Virtual Web Nodes
CPU	Intel Xeon L7555 x4	AMD Opteron 8381 HE x4	Intel Xeon E5530 x2	Intel Xeon E5530 x2
Memory	512 GB	256 GB	16 GB	16 GB
HDD	24 TB	100 TB	2 TB	2 TB
OS	CentOS 5	CentOS 5	CentOS 5	CentOS 5
Network	1 Gbps	10 Gbps	1 Gbps	10 Gbps

the hash value. This characteristic allows for load balancing during Web crawling. Recently, Web sites have been constructed with a large number of dynamic pages. The load balancing mechanism of the proposed crawler is indispensable for such Web sites.

6.7. Summary

In this chapter, a parallel distributed Web crawler was described as an application of QueueLinker. Web crawling is a difficult task, and a large number of computers are required to download and analyze Web pages from the Internet. The proposed crawler was implemented as a series of QueueLinker modules. The crawler realizes better load balancing and memory utilization between computers than traditional site-based Web crawlers. Moreover, it becomes possible to crawl the Web on a large scale while conserving resources, because each module is implemented by data structures that are temporally and spatially efficient.

The performance of actual Internet crawling is currently being evaluated. QueueLinker enables monitoring of the crawling progress. Figure 6.14 shows a visualization of a crawling for the Internet with 4 computers. Statistics on the number of items processed by each module and the amount of resources consumed by each module can be obtained with the help of the QueueLinker statistics mechanism.

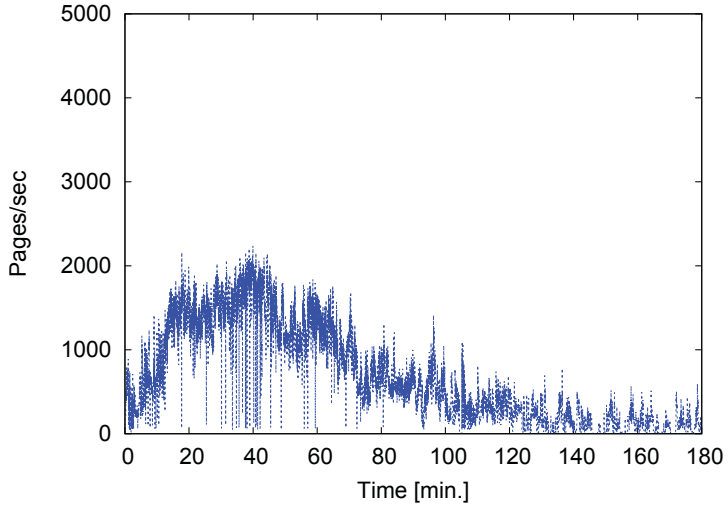

Figure 6.9 Number of Pages Downloaded per Second (2 Computers)

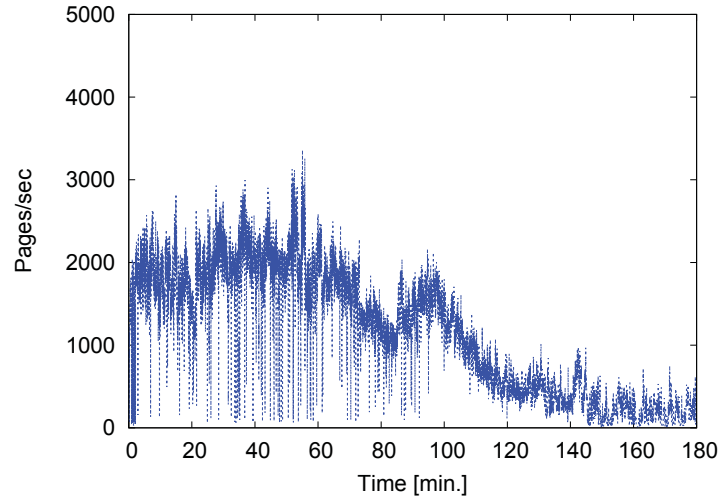

Figure 6.10 Number of Pages Downloaded per Second (4 Computers)

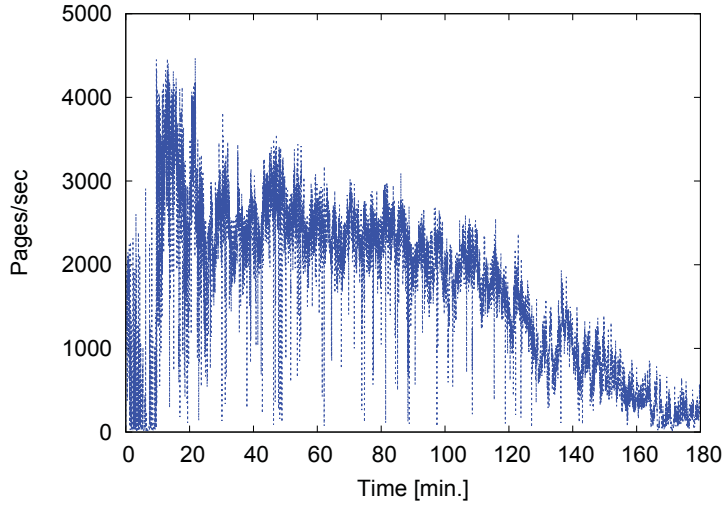

Figure 6.11 Number of Pages Downloaded per Second (8 Computers)

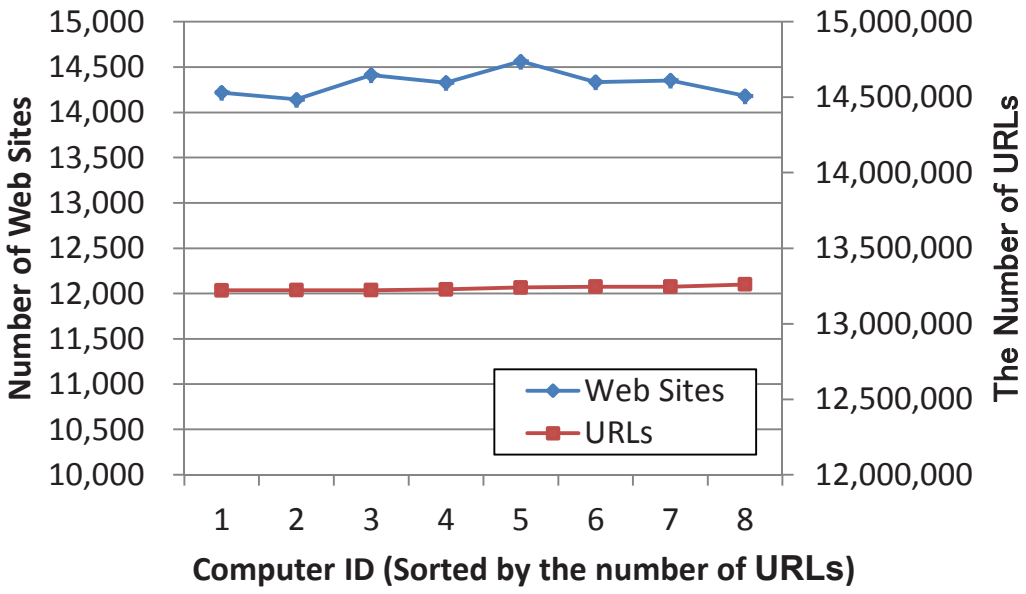

Figure 6.12 Number of Web Sites and URLs Handled
(Proposed Web Crawler, 8 Computers)

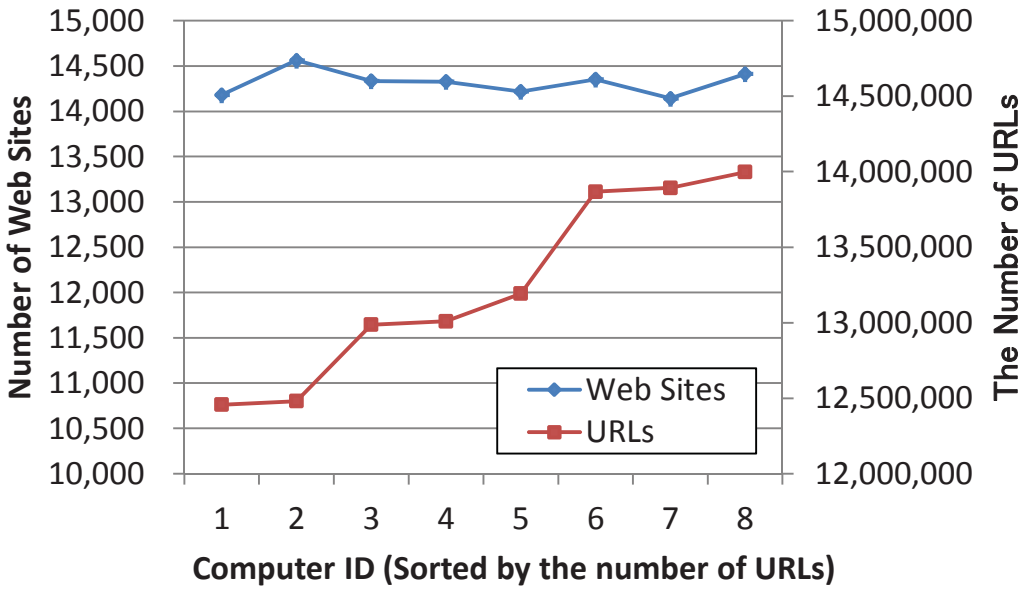

Figure 6.13 Number of Web Sites and URLs Handled
(Site-based Crawler, 8 Computers)

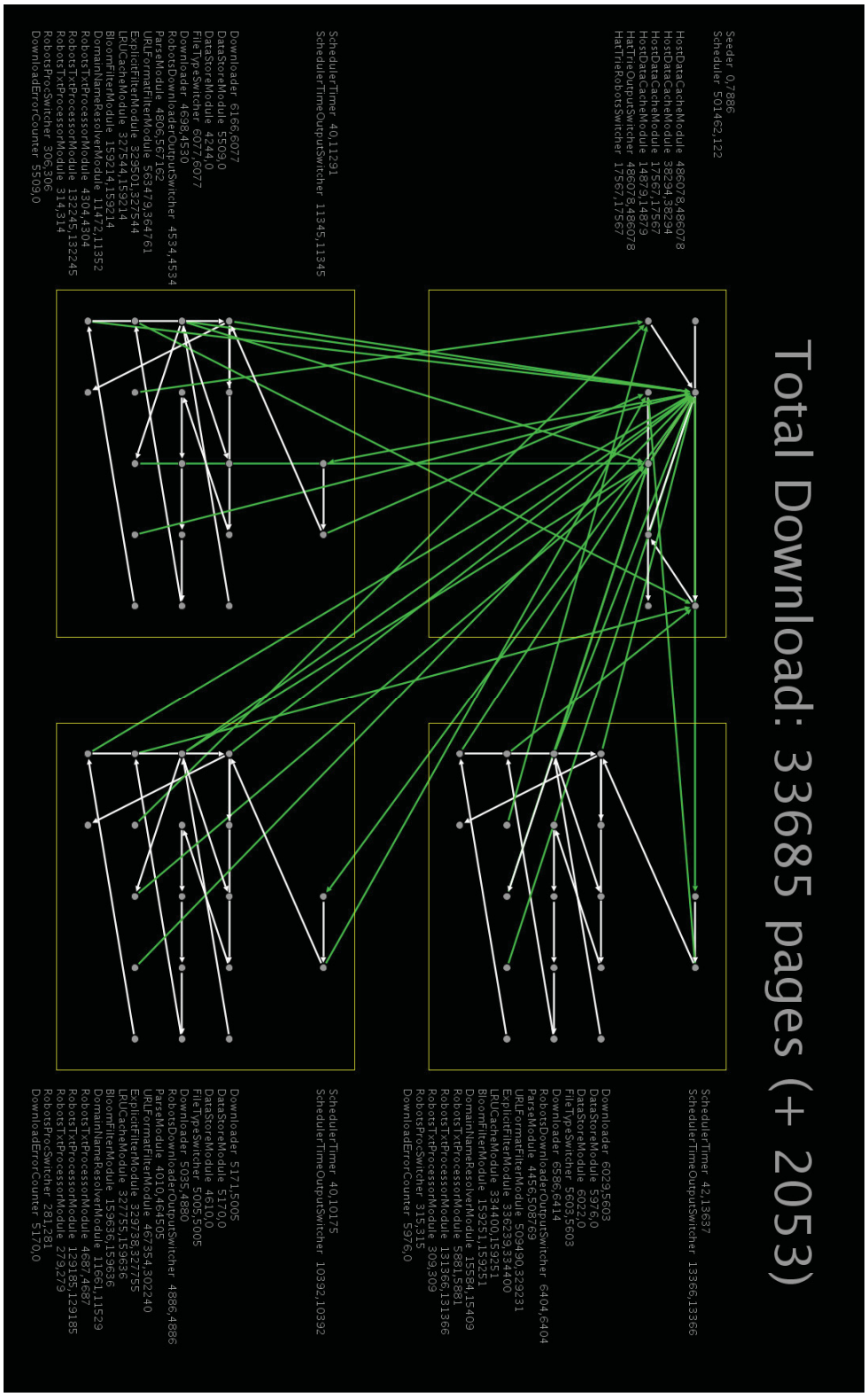

Figure 6.14 Crawling for the Internet with 4 Computers
(Each Computer is Represented as a Yellow Rectangle)

Chapter 7. Conclusion

Analyzing data streams in real-time contributes to many information services. The advance in information technology society increases the number of mobile devices and sensors, which results in generating large amount of data streams. Processing such large streams requires parallel distributed computing. The recent advancement in commodity computer hardware makes parallel distributed computing available for everyone. However, most developers want to avoid implementing concurrency control and network communication procedure, which are difficult to program. Moreover, several data stream applications are affected by processing latency even on order of microseconds. It is difficult for many programmers to consider these aspects, and so a framework for parallel distributed data-stream processing is indispensable for the real-time world.

7.1. Contributions of This Thesis

This thesis proposed QueueLinker, a framework for parallel distributed data-stream processing. QueueLinker adopts a producer–consumer approach as a programming model, and helps programmers to implement data-stream applications. The contributions of this thesis are listed below.

1. QueueLinker Development (Chapter 3)
   - QueueLinker provides producer–consumer programming model for data-stream processing and developers can implement applications for data streams without network communication and concurrency control.
2. Low Latency Processing of Continuous Query (Chapter 4 and 5)
   - A low latency execution method for continuous query on a multi-core processor is proposed in Chapter 4.
   - A proposed backup method for continuous query with realizing low latency processing on a distributed environment is proposed in Chapter 5.
3. Application Demonstration (Chapter 6)
   - QueueLinker can execute a practical application like Web crawler. The crawler can be executed with data parallel model of QueueLinker. It can realize better load balancing and memory utilization between computers compared to traditional site-based Web crawlers.

7.2. Future Works and Discussion

There are many remaining avenues for future work. QueueLinker is a simple but general-purpose framework; thus, it should be evaluated in more widely used cases. For example, QueueLinker may be used to execute machine learning algorithms in a parallel distributed environment. However, the programming interface may not be convenient for implementing machine learning algorithms.

Another interesting research topic is a programming language to describe data stream applications. Several languages can be used to describe data stream applications; for example, Continuous Query Language (CQL) was developed as an extension of SQL and has been adopted in STREAM; and IBM System S has adopted SPADE [38] as a dedicated language. Although QueueLinker currently uses Java to implement applications, the abilities of its simple programming interface were demonstrated in this thesis through continuous query execution and the Web crawler.

Another future work is comparing the performance with other frameworks, including Twitter Storm, Apache S4, and Esper. The proposed methods in this thesis provide a runtime complement to the QueueLinker package, especially for low-latency data stream processing; thus, QueueLinker can be a unique and valuable framework.

Chapter 4 described the proposed low-latency parallel execution of a continuous query with QueueLinker in a multi-core processor environment. An alternative should be considered to QPI communication, which is required when operators are assigned to multiple CPUs. Furthermore, future computer environments may have several hundred or even thousand CPU cores. In this environment, each CPU core may have to adjust its assignment without a centralized scheduler because the centralized scheduler cannot handle the statistics from the many cores, which causes bottlenecks.

Chapter 5 described a backup method to realize low-latency processing in a distributed environment. The impact on applications when the ordering mechanism is used in Chase Operator should be evaluated. Additionally, the concentration of tuples in a Chase Operator can result into a bottleneck when the data streams have a high arrival rate. Chase Operator can be a single point of failure. As a measure against bottlenecks, Chase Operator can—when three or more chasers are running in total—receive tuples from only two chasers and run the remaining chaser(s) in a manner similar to the active standby; that is, it switches from one chaser to another in response to data rate changes and failure occurrences. To address the single point of failure, an upstream backup in which a backup is deployed at the upstream side can be applied.

Chapter 6 described the proposed Web crawler. High-performance crawling and

better load balancing between computers were presented. The performance of actual Internet crawling is currently being evaluated. QueueLinker enables monitoring of the crawling progress. Statistics on the number of items processed by each module and the amount of resources consumed by each module can be obtained with the help of the QueueLinker statistics mechanism. QueueLinker currently has no resource scheduler for the Web crawler; an automated resource scheduler may be indispensable for global crawling.

While many future works remain, I believe this thesis provides valuable methodologies for processing data streams. I have developed QueueLinker with strong confidence that data streams will become ubiquitous in the future. I am hopeful that this thesis will prove helpful in data stream processing.

References

[1] J. Dean and S. Ghemawat, "MapReduce: Simplified Data Processing on Large Clusters," In *Proceedings of the 6th Symposium on Operating Systems Design and Implementation* (OSDI), San Francisco, US-CA, Dec. 2004.

[2] D. J. Abadi, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, M. Stonebraker, N. Tatbul, and S. Zdonik, "Aurora: a new model and architecture for data stream management," *The VLDB Journal*, vol.12, pp.120-139, Aug. 2003.

[3] D. Abadi, Y. Ahmad, M. Balazinska, U. Çetintemel, M. Cherniack, J. Hwang, W. Lindner, A. Maskey, A. Rasin, E. Ryvkina, et al., "The Design of the Borealis Stream Processing Engine," In *Proceedings of CIDR 2005*, pp.277-289.

[4] B. Babcock, S. Babu, M. Datar, R. Motwani, and J. Widom, "Models and issues in data stream systems," In *Proceedings of the 21st ACM SIGMOD-SIGACT-SIGART symposium on Principles of database systems* (PODS), Jun. 2002.

[5] S. Chandrasekaran, O. Cooper, A. Deshpande, M. J. Franklin, J. M. Hellerstein, W. Hong, S. Krishnamurthy, S. Madden, V. Raman, F. Reiss, and M. Shah, "TelegraphCQ: Continuous Dataflow Processing for an Uncertain World," In *Proceedings of the 1st Conference on Innovative Data Systems Research* (CIDR), Asilomar, US-CA, Jan. 2003.

[6] J. Chen, D. J. DeWitt, F. Tian, and Y. Wang, "NiagaraCQ: A Scalable Continuous Query System for Internet Databases," In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (SIGMOD), pp.379-390, Dallas, US-TX, May 2000.

[7] C. Cranor, T. Johnson, O. Spataschek, and V. Shkapenyuk, "Gigascope: a stream database for network applications," In *Proceedings of the 2003 ACM SIGMOD international conference on Management of data* (SIGMOD), Jun. 2003.

[8] B. Babcock, S. Babu, R. Motwani, and M. Datar, "Chain: Operator Scheduling for Memory Minimization in Data Stream Systems," In *Proceedings of the 2003 ACM SIGMOD International Conference on Management of Data* (SIGMOD), pp.253-264, San Diego, US-CA, Jun. 2003.

[9] R. Avnur and J. M. Hellerstein, "Eddies: Continuously Adaptive Query Processing," In *Proceedings of the 2000 ACM SIGMOD International Conference on Management of Data* (SIGMOD '00), pp.261-272, Dallas, US-TX, Mar. 2000.

[10] K. Claypool and M. Claypool, "Teddies: Trained Eddies for Reactive Stream Processing," In *Proceedings of DASFAA 2008*, pp.220-234.

[11] R. Khandekar, K. Hildrum, S. Parekh, D. Rajan, J. Wolf, KL. Wu, H. Andrade, and B. Gedik, "COLA: Optimizing Stream Processing Applications via Graph Partitioning," In *Proceedings of the ACM/IFIP/USENIX 10th International Middleware Conference* (Middleware), pp.308-327, Urbana Champaign, US-IL, Dec. 2009.

[12] J. Wolf, N. Bansal, K. Hildrum, S. Parekh, D. Rajan, R. Wagle, KL. Wu, and L. Fleischer, "SODA: An Optimizing Scheduler for Large-Scale Stream-based Distributed Computer Systems," In *Proceedings of the 9th ACM/IFIP/USENIX International Conference on Middleware* (Middleware), pp.306-325, Leuven, Belgium, Dec. 2008.

[13] Y. Drougas and V. Kalogeraki, "RASC: Dynamic Rate Allocation for Distributed Stream Processing Applications," In *Proceedings of IPDPS 2007*, pp.1-10.

[14] S. Chakravarthy and Q. Jiang. *Stream Data Processing: A Quality of Service Perspective*. Springer, 2009.

[15] A. Arasu, S. Babu, and J. Widom, "The CQL continuous query language: semantic foundations and query execution," *The VLDB Journal*, vol.15, pp.121-142, Jun. 2006.

[16] M. Najork and A. Heydon, "High-Performance Web Crawling," *COMPAQ Systems Research Center*, Sep. 2001.

[17] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "UbiCrawler: A Scalable Fully Distributed Web Crawler," In *Proceedings of AusWeb 2002*.

[18] J. Cho, H. Garcia-Molina, T. Haveliwala, W. Lam, A. Paepcke, S. Raghavan, and G. Wesley, "Stanford WebBase Components and Applications," *ACM Transactions on Internet Technology (TOIT)*, vol.6, no.2, pp.153-186, May 2006.

[19] N. Tatbul, U. Çetintemel, S. Zdonik, M. Cherniack, and M. Stonebraker, "Load Shedding in a Data Stream Manager," In *Proceedings of VLDB 2003*, pp.309-320.

[20] M. Isard, M. Budiu, Y. Yu, A. Birrell, and D. Fetterly, "Dryad: Distributed Data-Parallel Programs from Sequential Building Blocks," In *Proceedings of the 2nd ACM SIGOPS/EuroSys European Conference on Computer Systems* (EuroSys), pp.59-72, Lisbon, Portugal, Mar. 2007.

[21] Y. Yu, M. Isard, D. Fetterly, M. Budiu, Ú. Erlingsson, P. K. Gunda, and J. Currey, "DryadLINQ: A System for General-Purpose Distributed Data-Parallel Computing

Using a High-Level Language," In *Proceedings of the 8th USENIX Symposium on Operating Systems Design and Implementation* (OSDI), pp.1-14, San Diego, US-CA, Dec. 2008.

[22] U. Kang, C. E. Tsourakakis, and C. Faloutsos, "PEGASUS: A Peta-Scale Graph Mining System - Implementation and Observations," In *Proceedings of the 9th International Conference on Data Mining* (ICDM), pp.229-238, Miami, US-FL, Dec. 2009.

[23] C. Olston, B. Reed, U. Srivastava, R. Kumar, and A. Tomkins, "Pig Latin: A Not-So-Foreign Language for Data Processing," In *Proceedings of International Conference on Management of Data* (SIGMOD '08), pp.1099-1110, Vancouver, Canada, Jun. 2008.

[24] T. Condie, N. Conway, P. Alvaro, J. M. Hellerstein, K. Elmeleegy, and R. Sears, "MapReduce Online," In *Proceedings of the 7th USENIX Conference on Networked Systems Design and Implementation* (NSDI), San Jose, US-CA, Apr. 2010.

[25] H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, M. Cherniack, C. Convey, E. Galvez, J. Salz, M. Stonebraker, N. Tatbul, R. Tibbetts, and S. Zdonik, "Retrospective on Aurora," *The VLDB Journal*, vol.13, pp.370-383, Dec. 2004.

[26] S. B, Zdonik, M. Stonebraker, M. Cherniack, U. Çetintemel, M. Balazinska, and H. Balakrishnan, "The Aurora and Medusa Projects," *IEEE Data Engineering Bulletin*, vol.26, no.1, Mar. 2003.

[27] M. Cherniack, H. Balakrishnan, M. Balazinska, D. Carney, U. Çetintemel, Y. Xing, and S. B. Zdonik, "Scalable Distributed Stream Processing," In *Proceedings of the 1st Biennial Conference on Innovative Data Systems Research* (CIDR), Jan. 2003.

[28] Y. Xing, S. Zdonik, and JH. Hwang, "Dynamic Load Distribution in the Borealis Stream Processor," In *Proceedings of the 21st International Conference on Data Engineering* (ICDE), Apr. 2005.

[29] Y. Ahmad, B. Berg, U. Çetintemel, M. Humphrey, JH. Hwang, A. Jhingran, A. Maskey, O. Papaemmanouil, A. Rasin, N. Tatbul, et al., "Distributed operation in the Borealis stream processing engine," In *Proceedings of the 2005 ACM SIGMOD international conference on Management of data* (SIGMOD), Jun. 2005.

[30] S. Babu and J. Widom, "Continuous queries over data streams," *SIGMOD Record*, vol.30, pp.109-120, Sep. 2001.

[31] B. Babcock, S. Babu, M. Datar, R. Motwani, and D. Thomas, "Operator scheduling

in data stream systems," *The VLDB Journal*, vol.13, pp.333-353, Dec. 2004.

[32] S. D. Viglas and J. F. Naughton, "Rate-based Query Optimization for Streaming Information Sources," In *Proceedings of the 2002 ACM SIGMOD International Conference on Management of Data* (SIGMOD '02), pp.37-48, Madison, US-WI, Jun. 2002.

[33] R. H. Arpaci-Dusseau, "Run-Time Adaptation in River," *ACM Transactions on Computer Systems*, vol.21, no.1, pp.36-86, Feb. 2003.

[34] D. Carney, U. Çetintemel, A. Rasin, S. Zdonik, M. Cherniack, and M. Stonebraker, "Operator Scheduling in a Data Stream Manager," In *Proceedings of the 29th International Conference on Very Large Data Bases* (VLDB), pp.838-849, Berlin, Germany, Sep. 2003.

[35] J. Chen, D. J. DeWitt, and J. F. Naughton, "Design and Evaluation of Alternative Selection Placement Strategies in Optimizing Continuous Queries," In *Proceedings of the 18th International Conference on Data Engineering* (ICDE), pp.345-356, San Jose, US-CA, Mar. 2002.

[36] M. A. Shah, J. M. Hellerstein, S. Chandrasekaran, and M. J. Franklin, "Flux: An Adaptive Partitioning Operator for Continuous Query Systems," In *Proceedings of the 19th International Conference on Data Engineering* (ICDE), pp.25-36, Bangalore, India, Mar. 2003.

[37] D. Carney, U. Çetintemel, M. Cherniack, C. Convey, S. Lee, G. Seidman, M. Stonebraker, N. Tatbul, and S. Zdonik, "Monitoring Streams – A New Class of Data Management Applications," In *Proceedings of the 28th International Conference on Very Large Data Bases* (VLDB '02), pp.215-226, Hong Kong, China, Aug. 2002.

[38] B. Gedik, H. Andrade, KL. Wu, P. S. Yu, and M. Doo, "SPADE: The System S Declarative Stream Processing Engine," In *Proceedings of the 2008 ACM SIGMOD International Conference on Management of Data* (SIGMOD'08), pp.1123-1134, Vancouver, Canada, Jun. 2008.

[39] L. Amini, H. Andrade, R. Bhagwan, F. Eskesen, R. King, P. Selo, Y. Park, and C. Venkatramani, "SPC: A Distributed, Scalable Platform for Data Mining," In *Proceedings of the 4th International Workshop on Data Mining Standards, Services and Platforms* (DMSSP '06), pp.27-37, Philadelphia, PA-US, Aug. 2006.

[40] D. Tarditi, S. Puri, and J. Oglesby, "Accelerator: Using Data Parallelism to Program GPUs for General-Purpose Uses," In *Proceedings of the 12th*

International Conference on Architectural Support for Programming Languages and Operating Systems (ASPLOS), pp.325-335, Oct. 2006.

[41] D. Thain, T. Tannenbaum, and M. Livny, "Distributed Computing in Practice: The Condor Experience," *Concurrency and Computation: Practice and Experience*, vol.17, no.2-4, pp.323-356, 2005.

[42] C. Baru and G. Fecteau, "An overview of DB2 parallel edition," In *Proceedings of the 1995 ACM SIGMOD International Conference on Management of Data* (SIGMOD), 1995.

[43] G. Graefe, "Encapsulation of parallelism in the Volcano query processing system," In *Proceedings of the 1990 ACM SIGMOD International Conference on Management of Data* (SIGMOD), 1990.

[44] A. A. Diwan, S. Rane, S. Seshadri, and S. Sudarshan, "Clustering Techniques for Minimizing External Path Length," In *Proceedings of VLDB 1996*.

[45] JH. Hwang, M. Balazinska, A. Rasin, U. Çetintemel, M. Stonebraker, and S. Zdonik, "High-Availability Algorithms for Distributed Stream Processing," In *Proceedings of ICDE 2005*, pp.779-790.

[46] A. Koehl and H. Wang, "Surviving a Search Engine Overload," In *Proceedings of WWW 2012*.

[47] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "UbiCrawler: A scalable fully distributed web crawler," *Software: Practice & Experience*, vol.34, pp.711-726, Jul. 2004.

[48] V. Shkapenyuk and T. Suel, "Design and Implementation of a High-Performance Distributed Web Crawler," In *Proceedings of ICDE 2002*.

[49] H. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 Billion Pages and Beyond," In *Proceedings of the 17th International World Wide Web Conference* (WWW), pp.427-436, Beijing, China, Apr. 2008.

[50] H. Lee, D. Leonard, X. Wang, and D. Loguinov, "IRLbot: Scaling to 6 billion pages and beyond," *ACM Transactions on the Web*, vol.3, Jun. 2009.

[51] Y. Sun, Z. Zhuang, and C. L. Giles, "A Large-Scale Study of Robots.txt," In *Proceedings of WWW 2007*.

[52] S. Kolay, P. D'Alberto, A. Dasdan, and A. Bhattacharjee, "A Larger Scale Study of Robots.txt," In *Proceedings of WWW 2008*.

[53] P. Boldi, B. Codenotti, M. Santini, and S. Vigna, "Trovatore: Towards a Highly Scalable Distributed Web Crawler," In *Proceedings of the 10th International*

World Wide Web Conference (WWW), Hong Kong, May 2001.

[54] A. Heydon and M. Najork, "Mercator: A Scalable, Extensible Web Crawler," *World Wide Web*, vol.2, no.4, pp.219-229, 1999.

[55] B. H. Bloom, "Space/time trade-offs in hash coding with allowable errors," *Communications of the ACM (CACM)*, vol.13, Jul. 1970.

[56] N. Askitis and R. Sinha, "HAT-trie: A Cache-conscious Trie-based Data Structure for Strings," In *Proceedings of ACSC 2007*.

[57] J. M. Hsieh, S. D. Gribble, and H. M. Levy, "The Architecture and Implementation of an Extensible Web Crawler," In *Proceedings of NSDI 2010*.

[58] R. Baeza-Yates, C. Castillo, M. Marin, and A. Rodriguez, "Crawling a Country: Better Strategies than Breadth-First for Web Page Ordering," In *Proceedings of the 14th International World Wide Web Conference* (WWW), pp.864-872, Chiba, Japan, May 2005.

[59] C. Castillo, A. Nelli, and A. Panconesi, "A Memory-Efficient Strategy for Exploring the Web," In *Proceedings of the 2006 IEEE/WIC/ACM International Conference on Web Intelligence* (WI), Hong Kong, Dec. 2006.

[60] J. Cho and U. Schonfeld, "RankMass Crawler: A Crawler with High Personalized PageRank Coverage Guarantee," In *Proceedings of VLDB 2007*.

[61] A. Z. Broder, M. Najork, and J. L. Wiener, "Efficient URL Caching for World Wide Web Crawling," In *Proceedings of The 12th International World Wide Web Conference* (WWW), pp.679-689, Budapest, Hungary, May 2003.

Publications

Journals

1. **Takanori Ueda**, Sayaka Akioka, and Hayato Yamana, "Low Latency Data Stream Processing on Multi-core CPU Environments," IEICE Transaction D, vol. 96, no. 5, May 2013 (in Japanese).
2. **Takanori Ueda**, Koh Satoh, Daichi Suzuki, Kenji Uchida, Kousuke Morimoto, Sayaka Akioka and Hayato Yamana, "A Parallel Distributed Web Crawler Consisting of Producer-Consumer Modules," IPSJ Transactions on Database, vol. 57, Mar. 2013 (in Japanese).
3. **Takanori Ueda**, Kenji Uchida, Sayaka Akioka, Hayato Yamana, "An Operator Execution Method for Data Stream Processing to Reduce Latency and Achieve High Availability," DBSJ Journal, vol.10, no.3, pp.1-6, Feb. 2012 (in Japanese).
4. Nobuyuki Kubota, **Takanori Ueda** and Hayato Yamana, "Efficient Duplicated URL Detection for Web Crawlers," DBSJ Journal, vol.8, no.1, pp.83-88, Jun. 2009 (in Japanese).
5. Takuya Funahashi, **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Reliability Verification of Search Engines' Hit Count," DBSJ Journal, vol.7, no.3, pp.31-36, Dec. 2008 (in Japanese).
6. **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Disk Access Pattern Mining with System Call Level Access Log," DBSJ Journal, vol.7, no.1, pp.145-150, Jun. 2008 (in Japanese).
7. Takuya Funahashi, **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Gathering and Analysis of Unlisted Search Engines' Results," DBSJ Journal, vol.7, no.1, pp.37-42, Jun. 2008 (in Japanese).
8. Hiroaki Katase, Taku Matsunaga, **Takanori Ueda**, Takashi Tashiro, Yu Hirate and Hayato Yamana, "Web-Link Structure Reduction for Accelerating Link Analysis Algorithms," DBSJ Journal,vol.7, no.1, pp.245-250, Jun. 2008 (in Japanese).

International Conferences, Symposiums, and Workshops

1. Hiroki Asai, **Takanori Ueda** and Hayato Yamana, "Legible Thumbnail: Summarizing On-line Handwritten Documents based on Emphasized Expressions," In *Proc. of the 13th International Conference on Human-Computer Interaction with Mobile Devices and Services* (MobileHCI), Stockholm, Sweden, Aug. 2011 (Poster).

2. **Takanori Ueda**, Yu Hirate and Hayato Yamana, "The Challenge of Eliminating Storage Bottlenecks in Distributed Systems," In *Proc. of the 1st International Workshop on Software Technologies for Future Dependable Distributed Systems* (STFSSD), Tokyo, Japan, Mar. 2009.

3. Sayaka Akioka, Junichi Ikeda, **Takanori Ueda**, Yuki Ohno, Midori Sugaya, Yu Hirate, Jiro Katto, Shigeki Goto, Yoichi Muraoka, Hayato Yamana and Tatsuo Nakajima, "Scalable Monitoring System for Distributed Environment," In *Proc. of the 1st International Workshop on Software Technologies for Future Dependable Distributed Systems* (STFSSD), Tokyo, Japan, Mar. 2009.

4. **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Exploiting Idle CPU Cores to Improve File Access Performance," In *Proc. of the 3rd International Conference on Ubiquitous Information Management and Communication* (ICUIMC), Suwon, Korea, Jan. 2009.

5. Yasuaki Yoshida, **Takanori Ueda**, Takashi Tashiro, Yu Hirate and Hayato Yamana, "What's going on in search engine rankings?," In *Proc. of the 2008 IEEE International Symposium on Mining And Web* (MAW), Okinawa, Japan, Mar. 2008.

6. **Takanori Ueda**, Yu Hirate and Hayato Yamana, "EReM-DiCE: Exploiting Remote Memory for Disk Cache Extension," In *Proc. of the 1st International Workshop on Storage and I/O Virtualization, Performance, Energy, Evaluation and Dependability* (SPEED), Salt Lake City, US-UT, Feb. 2008.

7. Takashi Tashiro, **Takanori Ueda**, Taisuke Hori, Yu Hirate and Hayato Yamana, "EPCI: Extracting Potentially Copyright Infringement Texts from the Web," In *Proc. of the 16th International World Wide Web Conference* (WWW), Banff, Canada, pp.1151-1152, May 2007 (Poster).

Domestic Forums, Symposiums and Workshops

1. **Takanori Ueda**, Koh Satoh, Daichi Suzuki, Sayaka Akioka, and Hayato Yamana, "QueueLinker: A Framework for Parallel Distributed Processing of Data Streams," DEIM 2013, Mar. 2013.
2. Koh Satoh, **Takanori Ueda**, and Hayato Yamana, "An Accuracy Evaluation for Search Engine's Hit Count – Comparison with Document Frequency in Large-Scale Crawl Data–," DEIM 2013, Mar. 2013 (in Japanese).
3. Yusuke Yamamoto, Hiroki Asai, **Takanori Ueda**, Sayaka Akioka, and Hayato Yamana, "Real-time Detection of Twitter Users who Have Opinions for TV Programs," DEIM 2013, Mar. 2013 (in Japanese).
4. Shino Fujiki, **Takanori Ueda**, and Hayato Yamana, "A Method for Extracting Information from Twitter by Query Expansion Considering Continuous Variations of Correlated Words," DEIM 2013, Mar. 2013 (in Japanese).
5. Daichi Suzuki, **Takanori Ueda**, and Hayato Yamana, "A Consideration of I/O Parallelization of DBMS Queries on a High Performance Storage Environment," DEIM 2013, Mar. 2013 (in Japanese).
6. **Takanori Ueda**, Hiroki Asai, Shino Fujiki, Yusuke Yamamoto, Hiromasa Takei, Sayaka Akioka and Hayato Yamana, "Information Extraction by Analyzing Multiple Media Big Data Including Social Media," IPSJ SIG Technical Report (DBS), vol.2012-DBS-15, no.8, Dec. 2012 (in Japanese).
7. **Takanori Ueda**, Koh Satoh, Daichi Suzuki, Kenji Uchida, Kousuke Morimoto, Sayaka Akioka, and Hayato Yamana, "A Parallel Distributed Web Crawler Consisting of Producer-Consumer Modules," WebDB Forum, Nov. 2012 (in Japanese).
8. **Takanori Ueda**, Sayaka Akioka, and Hayato Yamana, "A Method of Thread Assignment for Low Latency Stream Processing on Multi-core Environment," DEIM 2012, Mar. 2012 (in Japanese).
9. Kenji Uchida, **Takanori Ueda**, and Hayato Yamana, "Design and Implementation of the Web Crawler Focusing on Customizability and Offering Real-time Stream Data," DEIM 2012, Mar. 2012 (in Japanese).
10. Daichi Suzuki, Koh Satoh, **Takanori Ueda**, and Hayato Yamana, "Constructing and Evaluation of RDF-Store using a Key-Value Database," DEIM 2012, Mar. 2012 (in Japanese).

11. Yuki Tanaka, Yusuke Yamamoto, **Takanori Ueda**, and Hayato Yamana, "A Similarity Video Search that can Adjust Search Time and Recall," DEIM 2012, Mar. 2012 (in Japanese).

12. **Takanori Ueda**, Kenji Uchida, Sayaka Akioka, and Hayato Yamana, "An Operator Execution Method for Data Stream Processing to Minimize Latency and Achieve High Availability," WebDB Forum 2011, Nov. 2011 (in Japanese).

13. Kosuke MORIMOTO, **Takanori Ueda**, Kenji UCHIDA, Hayato YAMANA, "An O(1) Time Complexity Web Crawling Scheduler with Guarantee of Minimum Interval of Accesses to a Web Server," In *Proc. of the 3rd Forum on Data Engineering and Information Management* (DEIM), Feb. 2011 (in Japanese).

14. **Takanori Ueda**, Hiroaki Katase, Kousuke Morimoto, Kenji Uchida, Makoto Yui and Hayato Yamana, "QueueLinker: A Distributed Framework for Pipelined Applications," In *Proc. of the 2nd Forum on Data Engineering and Information Management* (DEIM), Feb. 2010 (in Japanese).

15. Hiroaki Katase, **Takanori Ueda** and Hayato Yamana, "LittleWeb: Web Graph Compression Method using Similar Nodes Aggregation," In *Proc. of the 2nd Forum on Data Engineering and Information Management* (DEIM), Feb. 2010 (in Japanese).

16. **Takanori Ueda**, Hiroaki Katase, Kousuke Morimoto, Kenji Uchida and Hayato Yamana, "QueueLinker: Distributed Producer/Consumer Queue Framework," In *WebDB Forum* (Poster), Nov. 2009 (in Japanese).

17. Nobuyuki Kubota, **Takanori Ueda** and Hayato Yamana, "Efficient Duplicated URL Detection for Web Crawlers," In *Proc. of the 1st Forum on Data Engineering and Information Management* (DEIM), Mar. 2009 (in Japanese).

18. Taku Matsunaga, Hiroaki Katase, **Takanori Ueda**, Nobuyuki Kubota, Kosuke Morimoto, Yu Hirate and Hayato Yamana, "Implementing and Evaluating Graph Engine for Large Scale Graphs," In *WebDB Forum*, IEICE Technical Report, vol.108, no. 329, DE2008-69, pp. 43-43, Dec. 2008 (in Japanese).

19. Takuya Funahashi, **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Reliability Verification of Search Engines' Hit Count," In *iDB Forum*, IPSJ SIG Technical Report (DBS), vol.2008, no.88, pp.139-144, Sep. 2008 (in Japanese).

20. **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Dynamic I/O Optimization with Access Pattern Mining at OS Level," In *iDB Forum*, IPSJ SIG Technical Report

(DBS), vol.2008, no.88, pp.73-78, Sep. 2008 (in Japanese).

21. **Takanori Ueda**, "OS Level I/O Optimization in the Many-Core Era," In *jDB Workshop*, IPSJ SIG Technical Report (DBS), vol.2008, no.56, p.133, Jun. 2008 (in Japanese).

22. Takuya Funahashi, **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Gathering and Analysis of Unlisted Search Engines' Results," In *Proc. of the 19th Data Engineering Workshop* (DEWS), Mar. 2008 (in Japanese).

23. Hiroaki Katase, Taku Matsunaga, **Takanori Ueda**, Takashi Tashiro, Yu Hirate and Hayato Yamana, "Web-Link Structure Reduction for accelerating Link-Structure Analysis Algorithms," In *Proc. of the 19th Data Engineering Workshop* (DEWS), Mar. 2008 (in Japanese).

24. Takashi Tashiro, **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Evaluation of EPCI: Extracting Potentially Copyright Infringement texts by using a Search Engine," In *Proc. of the 19th Data Engineering Workshop* (DEWS), Mar. 2008 (in Japanese).

25. Sayaka Kuroki, **Takanori Ueda**, Yu Hirate and Hayato Yamana, "A Similar Code Search System using Abstraction of Program Codes," In *Proc. of the 19th Data Engineering Workshop* (DEWS), Mar. 2008 (in Japanese).

26. **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Disk Access Pattern Mining for System Call Level Access Log," In *Proc. of the 19th Data Engineering Workshop* (DEWS), Mar. 2008 (in Japanese).

27. Yasuaki Yoshida, Takuya Funahashi, Hiroaki Katase, **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Support System for Analysis of Commercial Search Engines' Rankings," In *DBWeb* (Poster), Nov. 2007 (in Japanese).

28. **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Exploiting Remote Memory to Speed-up Random Disk Access," In *Summer United Workshops on Parallel, Distributed and Cooperative Processing* (SWoPP), IPSJ SIG Technical Report (ARC), vol.2007, no.79, pp.151-156, Aug. 2007 (in Japanese).

29. Yasuaki Yoshida, **Takanori Ueda**, Takashi Tashiro, Yu Hirate and Hayato Yamana, "Quantitative Evaluation and Feature Analysis of Search Engine Rankings," In *Database Workshop* (DBWS), IPSJ SIG Technical Report (DBS), vol.2007, no.65, pp.441-446, Jul. 2007 (in Japanese).

30. **Takanori Ueda**, Yu Hirate and Hayato Yamana, "Performance Evaluation of using

Machines on a Network as Disk Cache," In *Proc. of the 18th Data Engineering Workshop* (DEWS), Mar. 2007 (in Japanese).

31. Takashi Tashiro, **Takanori Ueda**, Taisuke Hori, Yu Hirate and Hayato Yamana, "Copyright violation detection system for Web texts," In *Database Workshop* (DBWS), IPSJ SIG Technical Report (DBS), vol.2006, no.78, pp.27-33, Jul. 2006 (in Japanese).