QueueLinker: A Framework for Parallel Distributed Processing of Data Streams

Takanori UEDA †,‡‡ Koh SATOH ‡ Daichi SUZUKI ‡ Sayaka AKIOKA † and Hayato YAMANA ††,‡‡

† Information Technology Research Organization, Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
‡ Graduate School of Science and Engineering, Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
†† Faculty of Science and Engineering, Waseda University, 3-4-1 Okubo, Shinjuku-ku, Tokyo, 169-8555 Japan
‡‡ National Institute of Informatics, 2-1-2 Hitotsubashi, Chiyoda-ku, Tokyo, 101-8430 Japan
E-mail: {t-ueda, kohsatoh, dsuzuki, akioka, yamana}@yama.info.waseda.ac.jp

Abstract With the development of computer systems, many more devices are being connected to the network and generating 'data stream.' Analyzing data streams in real-time offers valuable information about human activities and contributes to many information services. 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.

Keyword Parallel Distributed Processing, Data Stream, Data Parallel, Producer-Consumer

1. Introduction

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.

These factors have led to the development of parallel-distributed computation frameworks such as Google MapReduce [1] and Dryad [2]. Apache Hadoop[1], an open source implementation of MapReduce, 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.

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 since MapReduce was introduced in 2004. A constant stream of information is now posted to social network services with 400 million Twitter tweets generated worldwide per day[2] 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.

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. Processing data streams in real-time presents programming difficulties owing to the fact that applications must handle multiple data sources, wait for data arrival, and receive data via the network. Because it is difficult for many programmers to consider these factors, it is vital that a framework for real-time, parallel-distributed data-stream processing be developed.

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

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

With this in mind, we have been developing QueueLinker, a framework for parallel distributed data-stream 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.

This paper is organized as follows. Section 2 describes the programming interface of QueueLinker. Section 3 describes the data parallel model of QueueLinker. Section 4 describes how to define a logical directed graph representing data-flow between modules. Section 5 describes the software architecture of QueueLinker. Section 6 explains several applications implemented by using QueueLinker. Section 7 concludes this paper.

2. Programming Interface of QueueLinker

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.

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

2.1. Push Module Interface

Figure 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.

Because QueueLinker uses a push thread unit (described in 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.

2.2. Pull Module Interface

Figure 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. QueueLinker assigns a dedicated thread to each instance of the pull module. Thus, a pull module implementation

```
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 1 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 2 Pseudo Code of a Pull Module

can make use of an infinite loop, which is useful for implementing blocking operations. To do this, the blocking operation is simply written inside an infinite loop in a pull module.

2.3. Source Module Interface

Figure 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.

2.4. Sink Module Interface

QueueLinker also provides an interface for data sink. A sink module has one input queue and no output queue, and is typically used to store data in secondary storage or present that data in visualize window to a user. Figure 4 shows pseudo code for a sink module.

3. Data Parallel Execution

This section describes how QueueLinker executes modules in a parallel-distributed way. Figure 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 parses 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 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

```
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 Pseudo Code of a Source Module

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

Figure 4 Pseudo Code of a Sink Module

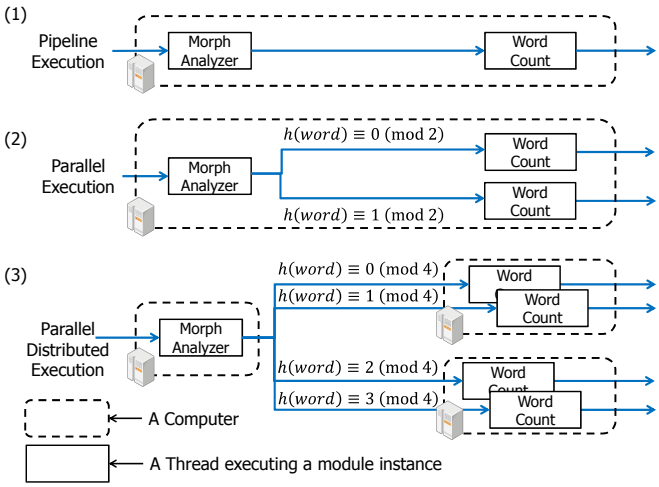

Figure 5 Data Parallel Execution Model of QueueLinker

modules, developers do not need to implement network communication procedures.

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 6 shows a logical directed graph (V, E) for the proposed Web crawler described in [5]. 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 7 provides pseudo code describing a logical directed graph for the application shown in Figure 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.

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.

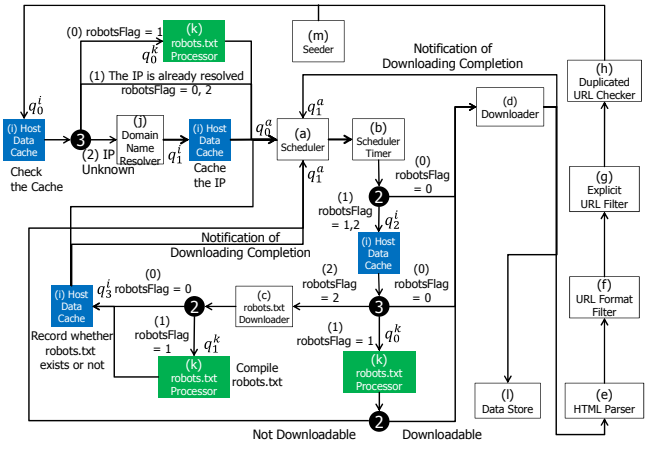

Figure 6 An Example of a Logical Directed Graph (Our Proposed Web Crawler Described in [5])

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 9 includes a switcher, indicated by a circle containing the number of destination modules. Figure 10 provides pseudo code for a switcher. 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 9, an output of module A is sent to B if the switcher returns 0, or to C if the 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 and the thread 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 [5] uses multiple switchers and virtual modules. The logical directed graph

```
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 7 Pseudo Code Describing an Application and Submitting the Job

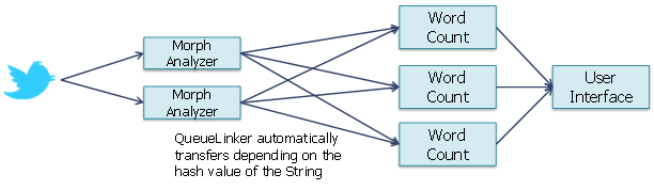

Figure 8 A Logical Directed Graph Described by the Code in Figure 7

of the crawler is shown in Figure 6. Despite being simple mechanisms, the switcher and virtual module are indispensable for describing a complex logical directed graph efficiently.

5. Software Architecture

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

5.1. Push Thread Unit

A push thread unit is designed to execute multiple push modules and sink modules as illustrated in Figure 11. A 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 the executed 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 [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 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 [3], can be substituted.

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 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.

5.3. Master 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

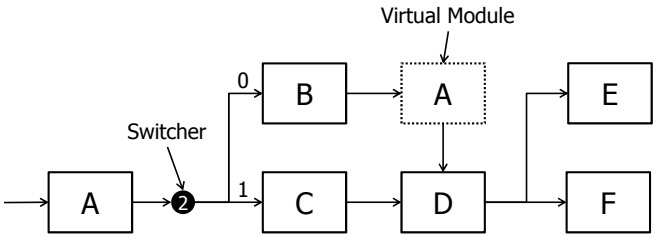

Figure 9 A Virtual Module and a Switcher

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

Figure 10 Pseudo Code of a Switcher

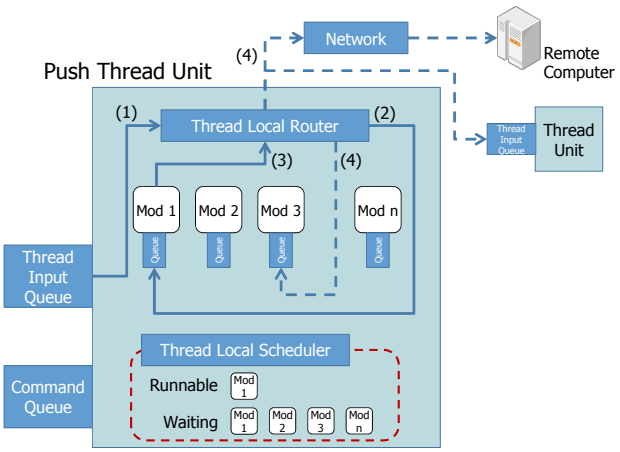

Figure 11 A Push Thread Unit

uses ZooKeeper[3] to communicate among master server, worker servers, and clients.

Figure 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 [4] can be implemented by using this mechanism.

6. QueueLinker Applications

QueueLinker can be used to execute widely applications such as Web crawlers, Web analytics applications and continuous queries. This section describes the overview of these applications.

6.1. A Parallel Distributed Web Crawler

QueueLinker can be used to implement a high-speed, parallel-distributed Web crawler [5]. 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 QueueLinker modules. The logical directed graph of the crawler is shown in Figure 6. 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.

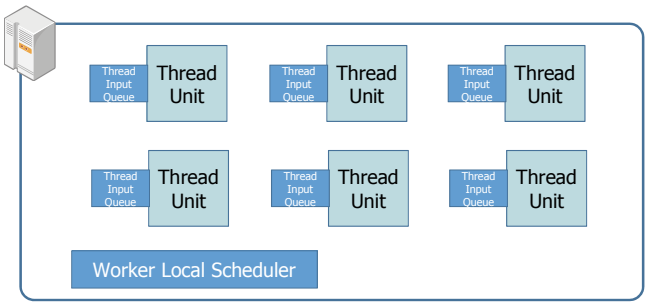

Figure 12 Thread Units on a Worker Server

QueueLinker enables monitoring of the crawling progress. Figure 13 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.

6.2. Web Analytics Application

As other applications, we have been developing Web data analytics applications. One of those applications is TV chatter extraction from Twitter. People now post their opinions about TV programs for Twitter when they are watching TV. Such tweets can be used for audience analysis. We have been implementing such application by using QueueLinker and Hadoop to extract them in real-time as shown in Figure 14.

6.3. Continuous Queries

The mechanism of QueueLinker supports the implementation of useful applications, including "continuous query [6]," 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 [7] 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.

QueueLinker can also be used to execute continuous

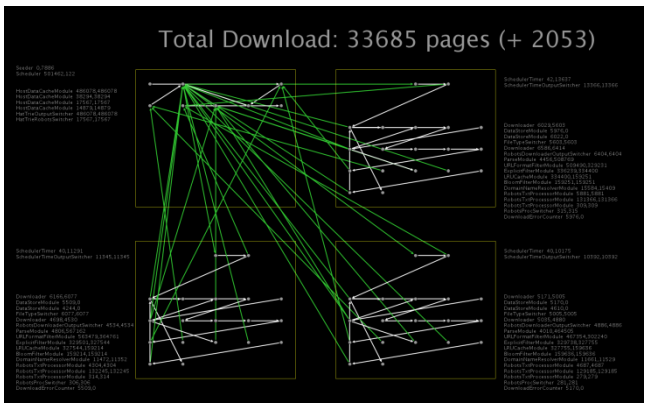

Figure 13 Crawling for the Internet with 4 Computers (Each Computer is Represented as a Yellow Rectangle)

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

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.

7. Conclusion

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

Acknowledgement

This research was supported by grant from JST "Multimedia Web Analysis Framework towards Development of Social Analysis Software"

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), pp.137-250, San Francisco, US-CA, Dec. 2004.

[2] 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.

[3] 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, 2004.

[4] T. Ueda, S. Akioka and H. Yamana, "Low Latency Data Stream Processing on Multi-core CPU Environments," IEICE Transactions on Information and Systems, Vol. 96, no. 5, May 2013 (Japanese, To Appear).

[5] T. Ueda, K. Satoh, D. Suzuki, K. Uchida, K. Morimoto, S. Akioka, H. Yamana, "A Parallel Distributed Web Crawler Consisting of Producer-Consumer Modules," IPSJ Transaction Database, vol. 57, Mar. 2013 (Japanese, To Appear).

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

[7] 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.

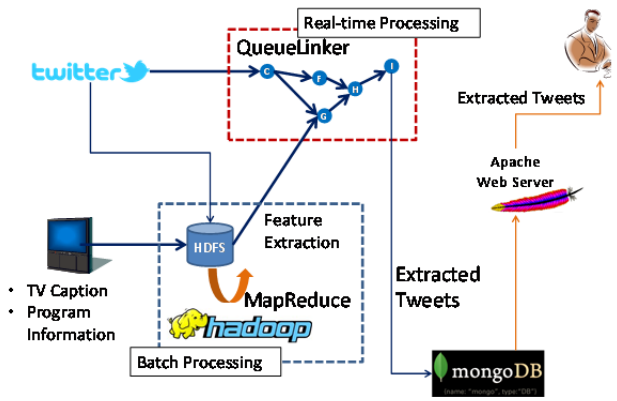

Figure 14 TV Chatter Extraction