Implementation of Data Affinity-based Distributed Parallel Processing on a Distributed Key Value Store

Naoko Hishinuma
Ochanomizu University
2-1-1, Otsuka, Bunkyo-ku
Tokyo 112-8610, JAPAN
naoko-h@ogl.is.ocha.ac.jp

Atsuko Takefusa and Hidemoto Nakada
National Institute of Advanced Industrial Science and Technology(AIST)
1–1–1, Umezono, Tsukuba
Ibaraki 305–8568, JAPAN
{atusko.takefusa, hide-nakada}@aist.go.jp

Masato Oguchi
Ochanomizu University
2-1-1, Otsuka, Bunkyo-ku
Tokyo 112-8610, JAPAN
oguchi@computer.org

ABSTRACT

The spread of cloud computing has increased the necessity of accumulating large amounts of data and performing high-speed data processing. Because strict consistency is not necessarily required for such large amount of data that cloud computing stores, a distributed Key Value Stores (KVS) is considered suitable for their data storage, based on an eventual consistency paradigm. In order to provide services such as SNS, mining and statistical processing of these data is indispensable. However because general distributed KVS systems are not designed for processing, these data must be transferred to distributed file systems such as HDFS, which enables data processing. The transfer cost issue has occurred in this case. To find a solution for this issue, we propose a method that performs high-speed data processing directly on a distributed KVS. In this paper, we extend the Apache Cassandra database, a distributed KVS that handles large amounts of data, to enable data affinity-based parallel processing. The parallel data processing mechanism runs the local processing on the stored values at each data node that stores the values, and it then returns only the results of the processing as an answer to a request. From the evaluation experiments, the proposed method is shown to be faster than the typically used Cassandra approach. In addition, even if the writing process is performed in the background while processing the data, the processing efficiency is appropriate for specific loads. The experimental results show that the data processing can be performed during the process of writing at approximately 10 Mbyte/sec if there are eight data nodes in the experiment environment.

Categories and Subject Descriptors

H.2.4 [**DATABASE MANAGEMENT**]: Systems—*Distributed databases*; D.1.3 [**Software**]: PROGRAMMING TECHNIQUES—*Concurrent Programming, Distributed programming*

General Terms

Measurement

Keywords

Distibuted KVS, Parallel and Distributed Processing, Data Affinity

Permission to make digital or hard copies of all or part of this work for personal or classroom use is granted without fee provided that copies are not made or distributed for profit or commercial advantage and that copies bear this notice and the full citation on the first page. To copy otherwise, to republish, to post on servers or to redistribute to lists, requires prior specific permission and/or a fee.

IMCOM(ICUIMC) '14 January 9-11, 2014, Siem Reap, Cambodia.

Copyright 2014 ACM 978-1-4503-2644-5 ...$15.00.

1. INTRODUCTION

With the popularity of cloud technology, applications services such as social networks and video sharing systems, which are available to many users and share a large amount of information, have been developed. Along with this advance, information that individuals produce will be stored on data center servers connected by the network in large amounts, and the amount of data on such servers is rapidly increasing. Because strict consistency is not necessarily required for such large amount of data that cloud computing stores, a distributed KVS, such as Apache Cassandra [1] [2] and Apache HBase [3], which is considered suitable for their data storage, based on an eventual consistency paradigm. To provide services such as SNS, techniques such as the mining and statistical processing of data are indispensable for taking advantage of the data cloud computing storage and extracting the information required by a user. However, because the distributed KVS that is used for data retention is not what has been designed for the processing, parallel processing as described above can perform high-speed processing by employing a parallel processing technique such as MapReduce [4]. Apache Hadoop [5] is generally used for these processes at present.

However, the use of Hadoop would increase the cost substantially, especially when the system transfers data from the distributed KVS to a distributed file system such as the Hadoop Distributed File System, which would perform the processing. As is conventional, it is best to not transfer the values to that point for the processing; it is necessary to allocate a process where the values are stored when large amounts of data are processed [7]. For this problem, the coprocessor [8] function, mounted on HBase Hadoop data

storage, let a simple process, such as counting and aggregation, execute on the server that holds the data. However, because Hadoop MapReduce has a batch processing system orientation, there is also a problem in that the system is not suitable for storing the data in real time.

Therefore, this study proposes and implements a method for high-speed data processing directly on a distributed KVS, which enables to prevent the data transfer costs. We have implemented the proposed method by using the Apache Cassandra database that can accumulate large-capacity data at a high speed [9]. Each process is executed on a Cassandra data node, which contains the requested large amounts of data, on the basis of the data affinity. In this paper, we extend the Apache Cassandra database, a distributed KVS that handles large amounts of data, to enable data affinity-based parallel processing. The parallel data processing mechanism runs the local processing on the stored values at each data node that stores the values, and it then returns only the results of the processing as an answer to a request. From the evaluation experiments, the proposed method is shown to be faster than the typically used Cassandra approach. In addition, even if the writing process is performed in the background while processing the data, the processing efficiency is appropriate for specific loads. Additionally, this paper also shows that this implementation is effective, even if maintaining a high consistency is required. In addition, even if the writing process is performed in the background while processing the data, the processing efficiency is good for the specific loads. The experimental results show that the data processing can be performed during the process of writing at approximately 10 Mbyte/sec if there are eight data nodes in the experiment environment.

The remainder of this paper is organized as follows. Section 2 introduces related research studies. Section 3 introduces an overview of Apache Cassandra. Section 4 describes our proposed method and the parallel data processing mechanism that is implemented. Section 5 shows the experimental results of the execution using our implementation, and Section 6 presents our concluding remarks.

2. RELATED WORK

Cassandra was used in this study and is a storage that has an emphasis on writing performance. Previous studies [10] have examined the use of Cassandra. Focusing on the performance cloud storage of the existing reading and writing performance, this study proposes and implements a method called MyCassandra cluster that has high reading and writing performance in the same cloud storage. This mechanism uses a selectable storage engine and is can distribute write and read requests appropriately.

ParaLite is [11] the like research associated with this study. ParaLite is a parallel Relational DataBase Management System (RDBMS) that is based on SQLite. It supports the parallel execution of SQL queries by using a function called Collective Query. In addition, ParaLite provides a function called UDX (User-Defined eXecutables), to make it possible to embed shell commands into the SQL query that was issued by the client. This feature is similar to that used in our proposed method: the feature has a function similar to User-Defined Function (UDF) and define processing which a user wants to execute as a plug-in. It can acquire the processing result only as an answer to the request.UDF is a function that allows users to define their own function as a plug-in, which can be applied to data in SQL.

Current distributed storage not be able to meet different requirements of two or more applications at the same time. Thus, Comet [12] has been studied as a distributed storage system to meet the needs of various applications. Multiple application can execute functions that were to application requirements such as adaptation to context, access log and tracking by extending the standard distributed KVS. Similar to our implementation, the user can execute any processing by using UDF over one distributed KVS.

Hive [13] and SQL/MapReduce [14] is a research to parallelize arbitrary processing by use of UDF and this research is related to our research. Hive supports a declarative language called HiveQL, which is SQL-Like. The HiveQL query is compiled into a MapReduce job and is executed on Hadoop. Thus, the user can operate data on Hadoop in such a way as to manipulate the data on RDBMS by use of SQL. In addition, HiveQL also support UDF function, therefore the user can parallelize any processing. SQL/MapReduce has enabled parallel data processing by extending the UDF and then useing MapReduce. These implementations are similar to our research in the point that users are able to parallelizes arbitrary processings by using a function to transmit jar files as UDF. These researches are also similar to our research in using CQL as a future extended direction. However, while SQL/MapReduce and Hive use MapReduce to perform parallelization, our implementation parallelizes by transferring processing to the data node.

Storm [15] is a real time data streaming framework that functions in memory. Storm constructs a processing graph, that feeds data from an input source through processing nodes. Storm aims event driven, high resolution processing, such as preprocessing of sensor data and word count of tweets. Our system focuses on medium to large resolution processing using stored data.

DataCutter [16] is a similar project, which provides a middleware infrastructure that enables processing of scientific datasets stored in archival storage systems across a wide-area network. DataCutter is integrated with Storage Resource Broker (SRB) [17], while we use Apache Cassandra as a storage system.

3. APACHE CASSANDRA

Apache Cassandra is a distributed database management system developed by Facebook Inc. and is an open-source Apache project [2]. Some of the main features of Cassandra include the following: the height of the fault tolerance, that there is no single point of failure in a non-centralized system, and that users can freely set the degree of consistency. It is specific features of Cassandra for a client to set the query level of consistency that it needs freely, which is uncommon not only in the RDBMS but also in other NoSQL. The consistency level of writing and reading can be set using the ConsistencyLevel option of Cassandra. Table 1 shows some of the settings and a consistent level. The consistency level of ONE is considered to be a generally weak level; to be consistent with a strong QUORUM, the setting of ALL is used. In Cassandra, client can specify the level of consistency in both writing and reading, and it is possible to control the strength of the consistency. We can determine the strength of the consistency using the expression $R + W > N$. R, W, and N stand for the number of read-replica, write-replica, and the replication number, respectively. For

example, if you specify two consistent levels of reading and writing, both the reading and writing replica numbers are set to two. If you do not satisfy this expression, it is a state of weak consistency (also called Eventual Consistency). In Eventual Consistency, in this case,e.g., when there is a read immediately after the a write, there is a possibility that the result of the read is that of the write is not reflected. The next level is referred to as Strong Consistency. Strong Consistency is when this formula is satisfied and when reading the result of the write-ahead is guaranteed.

It is good to store large amounts of data at a high speed because Cassandra employs Eventual Consistency. Using a YCSB to evaluate the basic performance, as in the existing studies [18], the RunTime of the workload for writing only is less than half of the RunTime of the workload for reading only, which increases by about three times in the throughput.

Table 1. Consistency level that can be set in the Cassandra.

Consystency level	Description
ONE	Write and Read process runs on a single target node. It is considered that the process has completed when there is a reply from one node.
QUORUM	Write and Read process runs on the target node. It is considered that the process has completed when there is a reply from the replica of the majority((number of replication/2)+1).
ALL	Write and Read process runs on multiple target node. It is considered that the process has completed when there is a reply from all of the replicas.

4. DESIGN OF THE PROPOSED METHOD

4.1 Overview of the proposed method

When we run data processing, e.g., mining or statistical processing, for the large amounts of data that are accumulated on the high-speed Cassandra, we obtain the values to be processed from Cassandra. The processing is then performed using MapReduce or a distributed file system such as HDFS. However, the read performance of Cassandra is not necessarily high. In addition, when Cassandra accumulates too much data, the Communication process for acquiring the value becomes too slow, and a large transmission delay could occur when transferring to a distributed file system from Cassandra.

Therefore, this study proposes a method for high-speed data processing directly on Cassandra, which accumulates the data to prevent the the transfer costs that would otherwise occur when the data are stored in real time.

Description of the performance of the proposed method. In the proposed method, it is possible to define the processing that the user wants to run as a plug-in using the UDF. This processing is performed locally on each data node that stores the processed data. Only the processing result is returned to the client, which reduces the transport costs, and high-speed data processing is achieved. If the value of the processing target is specified as multiple, parallel processing is allowed for different values, and a higher speed can be expected.

Figure 1 shows an overview of the proposed method.

(1) As in UDF, define the processing of any process X.

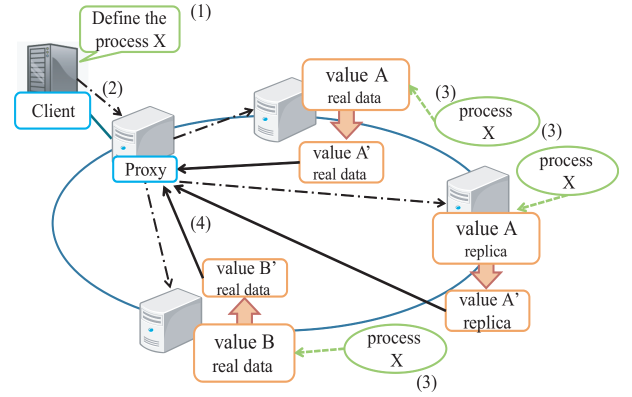

Figure 1. Overview of the proposed method.

(2) Send a request to each data node that contains the data from the client.

(3) Execute the process X that has been transferred, using the values A and B, in parallel. This implementation performs the same processing for each replica of values, but it omits the processing replica value B in Figure 1.

(4) Returns an answer to the requested value A' and B' which are the processing result.

4.2 Overview of the parallel data processing mechanism

Here, we describe the parallel data processing mechanism that is used to achieve the technique described in the previous section. The extended standard read commands of Cassandra (e.g., get, the multiget slice) do the following: after a request is sent for read processing, Cassandra runs the local processing for the stored value on each data node that stores that value. Then, it returns only the results of the processing as an answer to the request. This approach can perform any processing that the user specifies should be defined in Java UDF. In this implementation, jar files of processes that have been defined by a read request is transferred to the data node. Table 2 shows the contents of the main standard read command of Cassandra.

Table 2. List of commands.

Command	Function
get	get a specified value.
multiget slice	get multiple rows corresponding to the requested row keys.
get range slices	get multiple rows corresponding to the requested range of row keys.

An overview of the implementation is described with reference to Figure 1.

(1) Create a jar file of process X to be run.

(2) Transfer the jar file of the process X to be run along with the read request. If the jar file is not to be transferred, then a normal read is executed.

(3) Execute process X in parallel, which has been transferred for values A and B. The implementation performs the same processing for each replica of values, but it omits processing the replica of the value B in Figure 1.

(4) Return A' and B' that are the processing results as an answer to the request value.

The UDF is run even for a replica, and synchronization is performed in the background if the consistency is not maintained, checking by the hash value (Digest) answers to the requests.

```
/* public abstract class UDF {
     public abstract ByteBuffer
       processEach(ByteBuffer val)
                         Exception;
   } */
public class UDFImpl extends UDF {
  @Override
  public ByteBuffer
       processEach(ByteBuffer value)
  throws Exception {
    // write User defined
                  function body here.
    return result;
  }
}
```

Figure 2. A UDF example.

```
String udf =
"{\"classname\": \"<class name>\", ";
udf += "\"jar\":\"/<path>/my_udf.jar\", ";
udf += "\"option1\":\"<option1 value>\" }";

Map
<ByteBuffer, List<ColumnOrSuperColumn>>
results =
client.multiget_slice_udf(
        <row keys>, <column parent>,
<slice predicate>,
        <ConsistencyLevel>, udf);
```

Figure 3. An example of a requester program.

The application programming interface in Java for the parallel data processing mechanism is as follows. Figure 2 shows an example of defining a UDF. The parallel data processing mechanism extends the `getRow()` methods of `org.apache.cassandra.db.SliceByNamesReadCommand` and `org.apache.cassandra.db.SliceFromReadCommand`, which were extended to any processing to allow for each value. The `getRow()` method is a method that can be called to obtain a value on the side of the data node in Cassandra. The commands used to invoke it are listed in Table2. The user creates a `UDFImpl` class that extends an abstract class, a `UDF`, to implement the processing of any class `processEach ()` in a method. In addition, the pre-created jar file that contains the `UDFImpl.class` is kept.

Figure 3 shows an example of a program for executing the UDF. For the string `udf`, the jar file name and the Class name are specified in the JSON format. Furthermore, it is also possible to use the JSON format to specify that any options required by UDF should be performed, as defined by the user. The jar file that is specified by a command request is sent to the data node of Cassandra. It is possible to perform the processing specified by the data node by passing the jar file to the `multiget_slice_udf` function, which is an extension of the `multiget_slice` for this UDF.

5. PERFORMANCE EVALUATION OF THE PARALLEL DATA PROCESSING MECHANISM

We evaluate the performance of the parallel data processing mechanism by measuring the following: basic performance, performance when the consistency is changed and performance when write processing is performed in the background during data processing.

5.1 Experiment environment

To cluster up to eight nodes, we installed Cassandra with an extension. In this development, we used Cassandra version 1.2.0. Table 3 shows the performance of the node that was used for the measurements.

Table 3. Cassandra cluster node specification.

OS	Linux 2.6.32-5-amd64 Debian GNU/Linux 6.0.4
CPU	Intel(R) Xeon(R) CPU @ 2.66GHz x4 Intel(R) Xeon(R) CPU @ 3.10GHz x4
Memory	8GByte
HDD	500GB 7200RPM SAS Disk x 2
RAID Controller	SAS-6IR (RAID 0)
Network	1Gbps

5.2 Basic performance

To evaluate the basic performance of the present implementation, we investigated the performance of the data processing, in the case of the simplest form using Cassandra as mere storage.

5.2.1 Measurement overview: building a cluster using Cassandra

Next, We compare running the command word count (`wc`) after obtaining the values of the multiple-use standard commands for Cassandra, i.e., the `multiget slice` (hereinafter called Client-side processing), and when using the parallel data processing mechanism that is implemented (hereinafter called Server-side processing). We investigate the change in the execution time due to the change in the number of values and the number of data nodes to be used in processing. In this measurement, the result of running `wc` on a 20 MByte text is a value of 10 times `wc`, if the value is 10. Figures 4 and 5 shows the flow of processing for each when the value is two.

Flow of the client-side processing:

(1) Sending read request from the client.

(2) The data node that is responsible returns value A and B to the proxy as an answer to the request; the proxy returns the values to the client.

(3) The values A and B that are read out are written to file to perform a series `wc` on the file. Serial execution is performed that indicates the value obtained is processed by wc sequentially on one node.

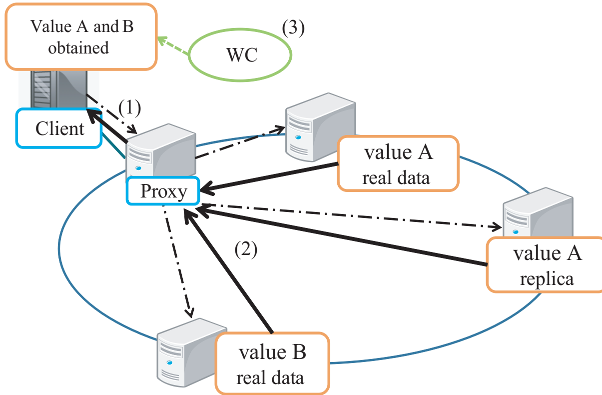

Figure 4. Flow of the client-side processing.

Flow of the server-side processing:

(1) The jar file is transmitted with read request from the client.

(2) The `wc` is executed in parallel for values A and B on the data nodes to respond to the request, and the new values A' and B' are produced as the processing result.

(3) The data node returns the values A' and B' to the proxy, and the proxy returns the values to the client.

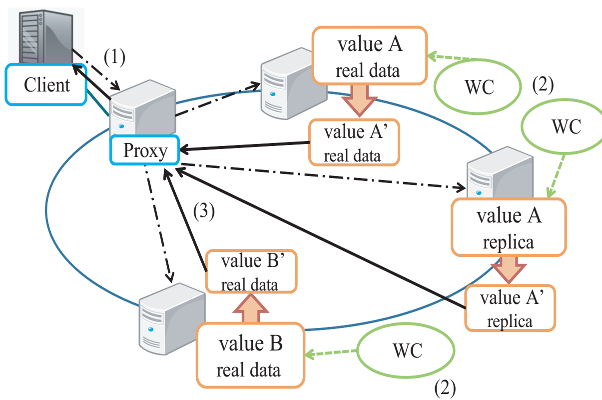

Figure 5. Flow of the server-side processing.

Both client- and server-side processing incorporate the processing of (1) to (3) as a single process. The number of data nodes was varied among three, five, and eight. We then measured the change in the execution time when were used the parameters shown in Table 4.

5.2.2 *Client-side processing: comparison of the execution time of server-side processing*

If you specify value numbers of 10 and 30 at consistency level ONE, then Figures 6 and 7 show the execution time for the client-side processing, the server-side processing and the `wc` time (the time spent only in wc in client-side processing). The vertical axis represents the execution time (sec), and the horizontal axis represents the number of data nodes.

Table 4. Cassandra cluster node specification

Number of replications	3
Number of data nodes	3, 5, 8
Number of values	10, 30
Consistency level	ONE, ALL

As observed in Figures 6 and 7, regardless of the number of values, the execution time of the server-side processing is reduced to less than one-fifth of the execution time of the client-side processing. This is because the server-side processing returns processing results of the `wc` as a response to the request and because the communication data volume could be reduced, by execution in parallel by dispersing the process, i.e., the series `wc`. This result occurs because it is possible to reduce the amount of time required for `wc` processing compared with the client-side process running time. Regarding the client-side processing in Figures 6 and 7, no changes are observed in the execution time because of the volume change. This is because the work to write the values to a file (that was obtained from Cassandra) temporary becomes overhead.

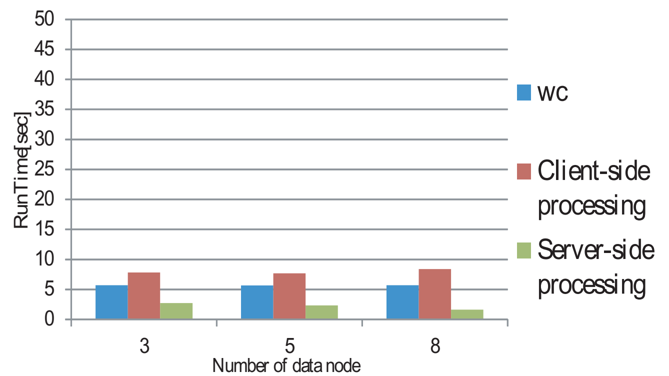

Figure 6. Execution time of word count (wc) processes. The number of values equals 10.

5.3 Change in the execution time associated with consistency

Cassandra contains a feature that can freely change the consistency according to the user's needs. Therefore, we must investigate the performance when the consistency of using this implementation is changed.

The execution time of each process is changed when the same experiment is performed in the previous section using the following values, as depicted in Figure 8: there are three or eight data nodes, the consistency level is ALL and the value is increased from 5 to 30. The vertical axis represents the execution time (sec), and the horizontal axis represents the number of values. In this figure, the result of client-side processing with three data nodes is omitted because it is almost the same with that with eight data nodes. When the given consistency level is ALL, the performance becomes low

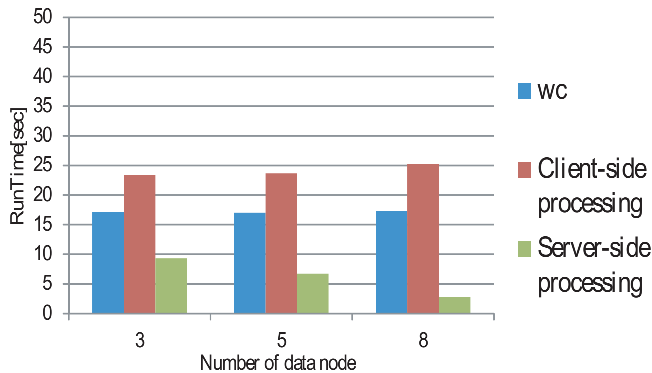

Figure 7. Execution time of word count (wc) processes. The number of values equals 30.

overall because the number of responses required for complete processing is increased. If a consistency level of ALL is specified, then the server-side processing is faster than the client-side processing. In addition, as a large number of nodes has become faster in the server-side processing, the same results are obtained as when a consistency level of ONE is specified. Therefore, in the proposed method, it is possible to process at a high speed even in a situation in which a high level of consistency is maintained

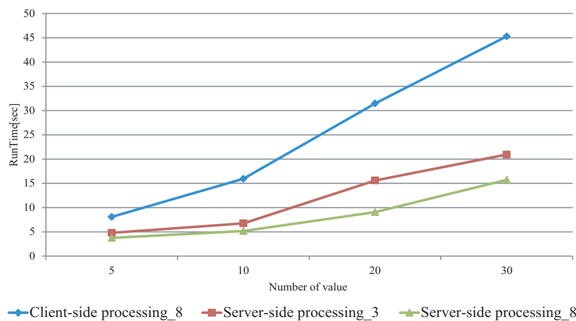

Figure 8. Execution time for the consistency level ALL.

5.4 Influence of the writing process on the execution time

This implementation is affected when the data processing is run during a write operation, when performing data processing and the writing process coexist on one system. In this section, to investigate the impact of the parallel data processing mechanism, we evaluate to what extent the performance is impacted when the write process is running in the background while the data is processed.

5.4.1 Measurement overview

In this measurement, the result of running the `wc` on a 20 MByte text is a value of 10 times `wc`, if the value is 10. This step is to investigate changes in the execution time when the throughput of the writing process is varied for concurrent data processing. The procedure of the experiment is as follows.

Processing flow:

(1) Send from the client jar files `wc` and the reading request. At the same time, write processing is continuously performed.

(2) The `wc` is executed in parallel for values A and B to suit the request, using the data nodes that are responsible for the new values A' and B' as the processing result.

(3) The data nodes returns values A' and B' to the proxy, and the proxy sends the values back to the client.

To the processing of (1) to (3), we determined the change in the execution time using the following conditions: eight and three data nodes, a consistency level of ALL for the writing process, a consistency level of ONE for the reading process and the parameters shown in Table 5.

Table 5. Measurement parameters.

Number of replications	3
Number of data nodes	3, 8
Number of values	50, 100, 200
Writing throughput (byte/sec)	0, 1K, 10K, 100K, 1M, 10M

5.4.2 Influence of the writing process on the execution time

We investigated the effect of the writing process to obtain the data processing execution time. Figures 9 and 10 indicate a change in the execution time for each of the cases in which the write throughput is changed 0 - 10 Mbyte/sec from eight and three data nodes. The vertical axis represents the execution time (sec), and the horizontal axis represents the write throughput (byte/sec). Each execution time is the average execution time when repeated 10 times via the process described in section 5.4.1.

Figures 9 and 10 indicate that when the tendency of the execution time to increase was observed, the write load affected the data processing performance. This finding was obtained by comparing the execution time of granting a write load of 10 Mbyte/sec to when there was no write load at all. In addition, the degradation of the data processing performance was found to be large given a write load of 100 Kbyte/sec if the data node number were small (three). On the contrary, even when the write load was 10 Mbyte/sec, if the data processing used a large number of nodes (eight), it was able to efficiently process even with the write load.

Since Disk-IO is considered to be the cause of increase in execution time, we investigated length of the queue of IO request. Figures 11 and 12 indicate a change in the length of queue of IO request for each of the cases in which 0 and 10 Mbyte/sec the write throughput on eight and three data nodes. Number of value is set to 200. The vertical axis represents the length of queue of IO request, and the horizontal axis represents execution time (sec).

Figures 11 and 12 indicate that length of queue of IO request is seventy-eight at a maximum if the data node number were small (three), 10 Mbyte/sec the write throughput. In addition, length of queue of IO request tends to be long in this case. The length of queue of IO request shorter than that when the data node number was three if the data processing used a large number of nodes (eight). Therefore it

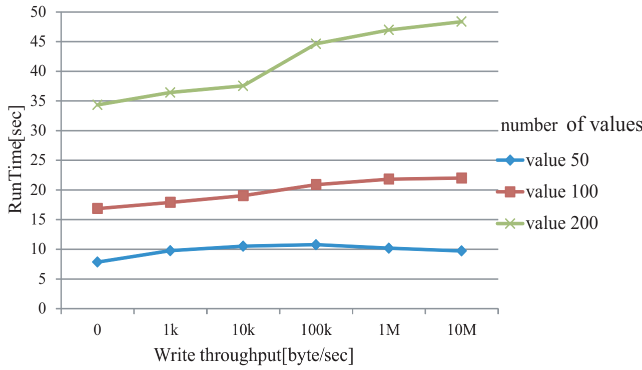

Figure 9. Execution time of word count (wc) processes under background write processes. The number of data nodes equals 3.

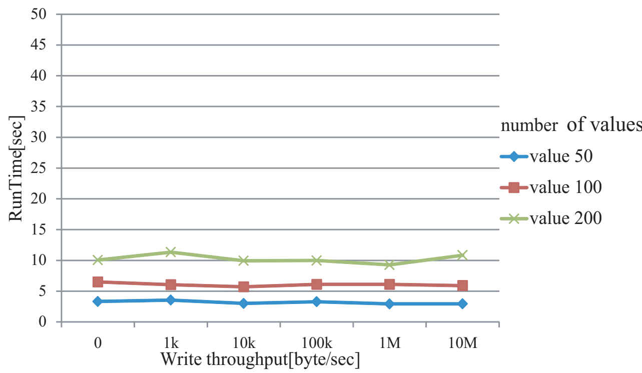

Figure 10. Execution time of word count processes (wc) under background write processes. The number of data nodes equals 8.

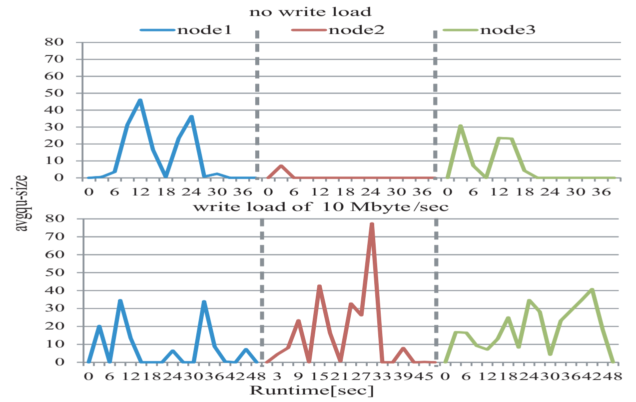

Figure 11. Length of queue of IO request. The number of data nodes equals 3.

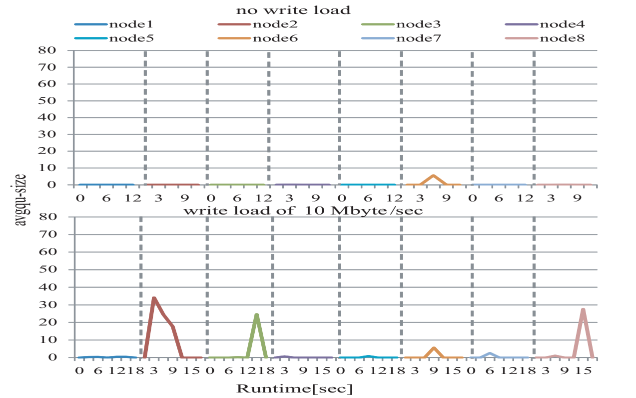

Figure 12. Length of queue of IO request. The number of data nodes equals 8.

was confirmed that disk IO was the cause of increase in execution time.

6. CONCLUSIONS AND FUTURE WORKS

In this research, to reduce the costs that are incurred and to speed up the processing of large volumes of data, we propose a method that performs high-speed data processing directly on a distributed KVS. In this paper, we extend the Apache Cassandra database, which is a distributed KVS for handling large amounts of data, to enable data affinity-based parallel processing. The parallel data processing mechanism runs the local processing on the stored value for each data node that stores the value. It then returns only the results of the processing as an answer to the request.

In the evaluation experiment, compared the case of a parallel data processing mechanism with that using Cassandra in its simplest form as a mere storage, we were able to improve the execution speed and proved the effectiveness of the proposed method. This result is caused by the following. Because the server-side processing returns a response to the request by processing results of the `wc` and because the communication data volume could be reduced, they were executed in parallel by dispersing the process, i.e., the series `wc`. This circumstance arises because the amount of time required to perform `wc` processing can be reduced compared with running the client-side process. In addition, it was found that the degradation of the data processing performance is large with a write load of 100 kbyte/sec if the number of data nodes is small (three). Even when the speed is 10 Mbyte/sec, if the data are processed by a large number of nodes (eight), it could be efficiently processed even with the write loads.

In our future work, we intend to conduct a performance evaluation with similar techniques to research the Hadoop cooperation function of Cassandra and compare it with the implementation using MapReduce and HBase. This step is intended to clarify that the performance of the proposed method is important. In addition, the situation closer to an actual environment should be evaluated, so we could consider the circumstance in which there are writes from many clients. The challenge surfaces that to have executable aggregation processing such as an average or sum, we must add the phase of the aggregate operation treatment. The method for the aggregate operation uses a language similar to SQL that is called Cassandra Query Language (CQL), which can be considered when adding such a function in the present implementation.

7. REFERENCES

[1] A.Lakshman and P.Malik. "Cassandra - A Decentralized Structured Storage System," The 3rd ACM SIGOPS International Workshop on Large Scale Distributed Systems and Middleware, October 2009.

[2] Eben Hewitt.,Cassandra: The definitive guide, trans. Shinpei Ohtani and Takashi kobayashi. O'Reilly JAPAN, 2011.

[3] The Apache software Foundation. Apache HBase. http://hbase.apache.org/hbase/.

[4] J.Dean and S.Ghemawat. "MapReduce: Simplified Data Processing on Large Clusters" In Proceedings of the 6th conference on Symposium on Opearting Systems Design & Implementation - Volume 6 (OSDI'04), Vol. 6. USENIX Association, Berkeley, CA, USA, 10-10.

[5] Tom White.,Hadoop: The definitive guide, trans. Ryuji Tamagawa. O'Reilly JAPAN, 2010.

[6] Dhruba Borthakur. "HDFS Architecture," 2008 The Apache Software Foundation.

[7] J.Gray, D.T.Liu, Maria Nieto-santisteban, A.S.Szalay, D.DeWitt, and G.Heber. "Scientific data management in the coming decade" Microsoft Technical Report, MSR-TR-2005-10.

[8] The Apache software Foundation. Apache HBase/coprocessor. https://blogs.apache.org/hbase/entry /coprocessor_introduction.

[9] B. F. Cooper, A. Silberstein, E. Tam, R. Ramakrishnan, R. Sears. "Benchmarking Cloud Serving Systems with YCSB" In Proc. SOCC2010, June 2010.

[10] S.Nakamura, K.Shudo. "A Cloud Storage Supporting Bath Read Heavy and Write Heavy Workloads" Proceedings of the 5th Annual International Systems and Storage Conference. ACM, 2012. p. 5.

[11] Ting Chen, Kenjiro Taura. "ParaLite: Supporting Collective Queries in Database System to Parallelize User-Defined Executable" In Proceedings of the 12th IEEE/ACM International Symposium on Cluster, Cloud and Grid Computing (CCGrid12), Ottawa, 2012:474-481

[12] R.Geambasu, A.A.Levy, T.Kohno, A.Krishnamurthy and H.M.Levy. "Comet: An active distributed key-value store" In Proc. OSDI. 2010, October. p.323-336.

[13] A.Thusoo, J.S.Sarma, N.Jain, Z.Shao, P.Chakka, S.Anthony, H.Liu, P.Wyckoff and R.Murthy. "Hive: a warehousing solution over a map-reduce framework" In Proc. VLDB Endow. 2, 2 (August 2009), 1626-1629.

[14] E.Friedman, P.Pawlowski, and J.Cieslewicz. "SQL/MapReduce: a practical approach to self-describing, polymorphic, and parallelizable user-defined functions" In Proc. VLDB Endow. 2, 2 (August 2009), 1402-1413.

[15] Storm: http://blog.gigaspaces.com /gigaspaces-and-storm-part-1-storm-clouds/.

[16] Michael Beynon, Renato Ferreira, Tahsin Kurc, Alan Sussman, Saltz. "DataCutter: Middleware for Filtering Very Large Scientific Datasets on Archival Storage Systems" http://www.cs.umd.edu/projects /hpsl/ResearchAreas/DataCutter.htm

[17] Chaitanya Baru, Reagan Moore, Arcot Rajasekar, Michael Wan. "The SDSC Storage Resource Broker" Proc. CASCON'98 Conference , Nov.30-Dec.3, 1998, Toronto, Canada.

[18] N.Hishinuma, A.Takefusa, H.Nakada, M.Oguchi. "A Study about the data volume and processing performance in KVS data processing by Cassandra" In Proc. DEIM Forum 2012, C2-5, March 2012.