Storage-Side Processing for Spark with Tiered Storage

Kaihui Zhang[1,a)] Yusuke Tanimura[2,1] Hidemoto Nakada[2,1]
Hirotaka Ogawa[2]

Abstract: Apache Spark is a parallel data processing framework that executes fast for iterative calculations and interactive processing, by caching intermediate data in memory with a lineage-based data recovery from faults. However, Spark still needs to load input data from a persistent storage at the beginning of main analytics and store the final result on the storage at the end of the analytics. In this study, we use a memory-based, tiered storage called Alluxio for the persistent storage of Spark and implement the active storage concept, which utilizes processing resources on the storage side and reduces the amount of I/O between Spark and Alluxio. As our first step, we implemented filtering on the Alluxio worker and examined performance improvement of reading data. The results showed that the performance was worse than we expected, due to inefficiency of storage-side filtering in our implementation.

1. Introduction

Apache Spark (Spark) [1] is an open source parallel data processing framework and highly used in industry due to attractive performance and capabilities for big data analytics and processing for artificial intelligence. One of the biggest advantages of Spark against Hadoop MapReduce [2,3] is that Spark can hold intermediate data in memory which eliminates the need to read and write the data on disks and fault tolerance is supported by the lineage-based recovery mechanism. Thus Spark is suitable for data mining and machine learning that require iterative calculation. However, Spark still needs to load input data from a persistent storage at the beginning of main analytics and store the final result on the storage at the end of the analytics. Spark can use local disks of execution nodes, where analytics are executed, to store persistent data by using Hadoop DFS, etc. While this architecture benefits data affinity, some resources such as CPU, memory, network, etc. on the analytics environment are consumed and applications may have a negative impact on performance. If the Spark environment is shared by multiple applications, one application might be affected by workloads of other applications.

Separating the execution nodes of Spark from the persistent storage, that is having an external storage, is a solution of the above problem. Although there is a network latency between Spark nodes and the external storage, Spark applications can fully use the resources (CPU, memory, etc.) on the execution nodes. In addition, the applications can utilize extra processing resources available on the external storage for pre/post processing of the main analytics. This study aims at achieving the concept, which is also well known as "Active Storage" [4], in the Spark architecture, with a tired memory/disk storage component called Alluxio [5]. As the first step of the study, filtering function on the external storage is implemented and evaluated in order to investigate the feasibility of storage-side processing on external storage. The following investigation is performed by using a synthetic benchmark.

(1) Investigate how the size and structure of different data sets can affect read performance on the Alluxio client, in the case of a data retrieve request with a query, when filtering was implemented on the Alluxio worker nodes.
(2) Compare the performance between when filtering was applied on the Alluxio worker nodes and when filtering was applied on the Spark executor level. The total execution time including send of the request and data return from the Alluxio worker node to the Spark node was measured.

Based on these results, feasibility of storage-side processing on the external storage and potential improvements of efficiently using external storage in Spark was discussed.

2. Background

2.1 External Storage for Spark

Spark can perform distributed processing on a cluster consisting of nodes having both compute and storage functions. In a typical use case, Spark loads data from Hadoop DFS, performs memory-based main analytics and then outputs

[1] University of Tsukuba
[2] National Institute of Advanced Industrial Science and Technology (AIST)
[a)] neilyo.chou@aist.go.jp

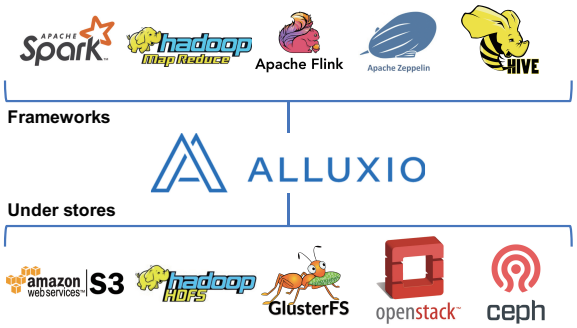

Fig. 1 The position of Alluxio in the big data processing architecture.

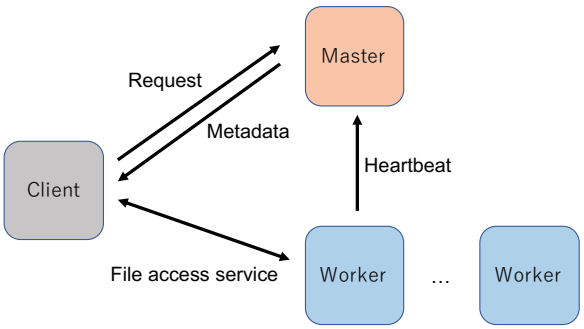

Fig. 2 Alluxio's distributed system design.

results to Hadoop DFS. In the main analytics, intermediate data is usually hold in memory except in a few causes such as shuffle operation, users' intention to use disks, etc. As a replacement of HDFS, other storage systems or services which provide the Hadoop-compatible interface are available.

2.2 Alluxio

2.2.1 Overview

Alluxio [5] is a middleware between the underlying distributed file system and the upper distributed computing framework as Figure 1 shown. The primary responsibility is to provide data in memory or other storage facilities as a files access service. Alluxio is positioned between traditional big data storage such as Amazon S3, Apache HDFS and OpenStack Swift and big data processing frameworks such as Spark and Hadoop MapReduce. The frameworks can directly read/write data from/to the storage but the storage is a potential bottleneck when large amounts of data are read or written. Because Alluxio works as a memory-based distributed storage system, it provides an order of magnitude acceleration for big data applications and achieves storage tiering. The file access interface to the underlying storage through Alluxio is transparent but configurable for caching options, etc.

2.2.2 Architecture

Alluxio has a master-slave architecture similar to the HDFS implementation. The Alluxio master is responsible for managing the global file system metadata, such as the file system tree. The Alluxio worker is responsible for providing a data storage service. The Alluxio client provides users with a unified file access service interface.

Figure Figure 2 shows Alluxio's distributed system design.

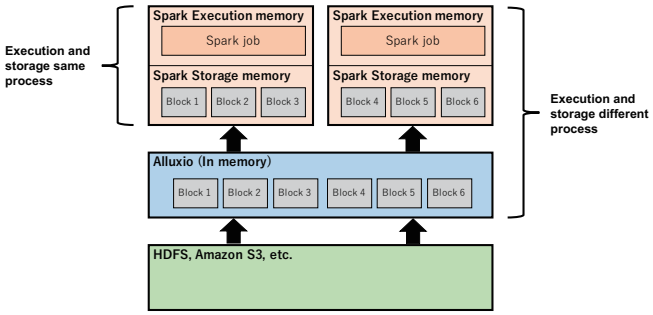

Fig. 3 The system configuration diagram of running Spark with Alluxio.

When the application needs to access Alluxio, the client first communicates with the master to obtain the metadata of the corresponding file, and then communicates with the corresponding worker to perform the actual file access operation. The file's metadata maintained by the master contains information about location (i.e., worker node) of divided blocks of the file. All workers periodically send a heartbeat to the master and maintain file system metadata information with ensuring that they are aware of the master and the entire storage service still works normally in the cluster. The master does not actively initiate communications with other components. Instead, it just communicates with them in its reply for the requests from them.

2.3 Running Spark with Alluxio

When Spark is used with Alluxio as an external storage, Alluxio mediates between Spark and the underlying storage in general, as shown in Figure 3. Spark applications can load and store any data on Alluxio including intermediate data, but using Alluxio as a cache instead of the internal storage such as local memory and/or disks causes large overhead in I/O, which is confirmed in our preliminary experiment. Therefore Alluxio is currently more suitable for load of the input data before main analytics and store of the results after the analytics.

It is possible to setup Spark and Alluxio on either same or different node groups. When they are separately setup, the Spark nodes has no storage memory consumption and more execution memory is available for applications, as the data is stored in the Alluxio nodes. The drawback is that the data transfer between Spark and Alluxio might be a bottleneck due to network delay, serialization/deserialization cost, etc. and that system resources on the Alluxio nodes might not be fully utilized, too.

The write/read operation for data in DataFrame and RDD is programmed in Spark as follows.

Write operation:

- RDD:
  saveAsTextFile("alluxio://...")
  saveAsObjectFile("alluxio:/...")
- DataFrame:
  write.format("⟨file format⟩").save("alluxio://...")

Read operation:

- RDD: sc.textFile("alluxio://...")

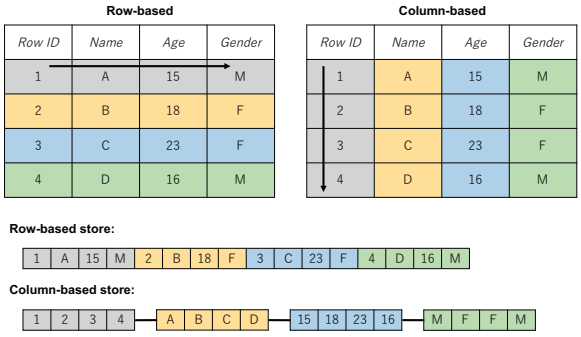

Fig. 4 The difference between the row-based store and the column-based store.

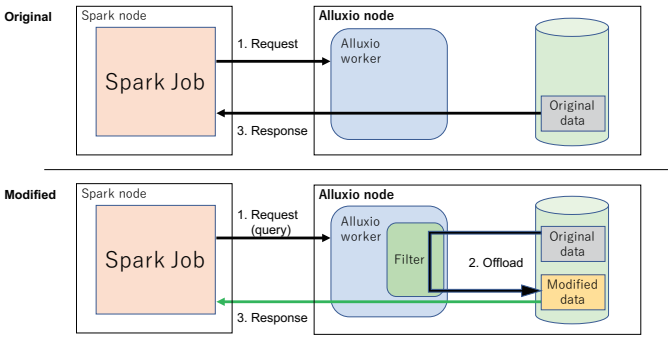

Fig. 5 Storage-side processing: Offload the request data for Spark.

- DataFrame: sqlContext.read.text("alluxio://...")

2.4 Processing with Data Format

2.4.1 Columnar Storage

The columnar storage stores data like the relational table in columns. The concept has already appeared in C-Store [6]. Figure 4 shows the difference between the row-based store and the column-based store. The columnar storage can improve the query performance compared with row storage because the columnar storage uses these optimizations:

- Compression to save storage space: Because the same column has the same data type, more efficient compression coding, such as Run Length Encoding and Delta Encoding, can be used to save larger storage space.
- Skip of non-conforming data: Only the required data can be read to reduce storage I/O.
- Support of vector operation: The verctor operation achieves better scanning performance.

2.4.2 Parquet Format

Apache Parquet (Parquet) [7] is a columnar storage format available to any project in the Hadoop ecosystem, regardless of the choice of data processing framework, data model and programming language. Parquet is just a storage format and independent from languages and platforms. Parquet does not require binding with any kind of data processing frameworks, either. Currently, the Parquet format is adapted by the query engine like Hive, Impala, Pig, Presto, Drill, the computing framework like MapReduce, Spark, and the data model like Avro, Thrift, Protocol Buffers. The data generated by other serialization tools can be easily converted to the Parquet format.

3. Big Data Analytics Platform with External Active Storage

3.1 A Concept of Active Storage

As computer processing performance and data density become higher and higher, large-scale computing is more and more limited by I/O performance. The data transmission technology can not keep up with the requirements of large-scale distributed computing. In order to improve overall performance, there is a way to move processing to the data side to reduce unnecessary I/O and the data transfer between storage-side and main processing nodes. This will also reduce the processing-side CPU resource consumption and achieve better performance, which allows the processing nodes to handle larger data.

3.2 Our Proposed Architecture and Current Implementation

Based on the concept of the active storage, we assume the separation architecture between the big data analytics platform and the backend storage system and have proposed the efficient data staging design [8]. Then we are trying to implement an external active storage for Spark by using Alluxio. In this paper, off-loading of filtering unnecessary data on the Alluxio storage side has been implemented as shown in Figure 5. When the Alluxio client sends a data request with a query to the Alluxio, the Alluxio worker filters the data based on the query. Then the filtered data is returned to the client. In the case of accessing the data from Spark applications, the Spark executor behaves as an Alluxio client. Due to the off-loading of filtering, the amount of I/O from the Alluxio worker to the client as well as the Spark executor will be reduced. The data size that Spark needs to process becomes lighter, which helps the Spark node save processing resources.

In our implementation, a query can be speficied in the format, "⟨file path⟩?⟨column name⟩:⟨value⟩". For example, "alluxio://foo.example.org/test.parquet?name:Ben" reads data from the test.parquet file and only reads records that match the value of the "name" column equals to "Ben." Internally, the filtering function is implemented with Parquet. More specifically, parquet-mr that consists of several modules including the ability to read and write Parquet files is used.

4. Evaluation

4.1 Experiments Setup

In order to evaluate our current implementation, how much the data size and selectivity affect on performance of read with a filtering query. At the first experiment, time of reading was measured on the Alluxio client node and the case with filtering was compared with the case without it. In the former case, filtering was applied on the Alluxio worker nodes. At the second experiment, time of reading was mea-

Table 1 Machine specification used for the Alluxio system.

Master	CPU	AMD Opteron 6128 2.0GHz, 8 cores
	Memory	32GB
	Network	10 Gbps
Worker	CPU	Intel Xeon CPU E31230 3.20GHz, 4 cores
	Memory	8 GB
	Network	10 Gbps

Table 2 Machine specification used for the Alluxio client and the Spark execution.

CPU	Intel Xeon CPU E5-2620v3 2.40GHz, 6 cores x2
Memory	128 GB
Network	10 Gbps (for HDFS connection)
NVMe-SSD	Intel SSD DC P3700
SSD	OCZ Vertex3 (240GB, SATA6G I/F)
HDD	Hitachi Travelstar 7K320 (SATA3G I/F)
OS	Ubuntu 14.04 (Kernel v.3.13)
File System	Ext4

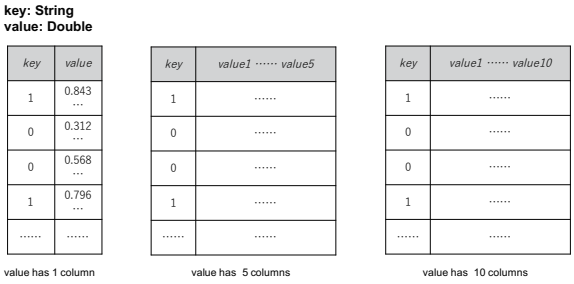

Fig. 6 Structure of three different parquet format dataset.

sured on the Spark node. Then our implementation was compared with the case where filtering was specified in the Spark program. Spark provides a function called pushdown filter but the function was implemented in the internal of Spark.

Table 1 shows the machine specification used in our Alluxio system. The system consists of 1 master node and 2 worker nodes. Our implementation of storage-side filtering was based on the Alluxio version 1.7.0. The Alluxio client was executed in the machine shown in Table 2. At the second experiment, the machine shown in Table 2 was also used in our Spark node and Spark was executed in a local mode. The version of Spark was 2.2.0. The version 1.9.0 of parquet-mr was used in both of our Alluxio and Spark systems.

Three datasets were generated by Spark for our experiments and all of the file formats were Parquet as mentioned in Section 2.4.2. Each file has,

- 1 column of "key" + 1 column of "value"
- 1 column of "key" + 5 columns of "value"
- 1 column of "key" + 10 columns of "value"

and the number of rows is variant. As Figure 6 shown, the "key" column is String type and the all "value" columns are Double type.

4.2 Performance Effect of Filtering

In this experiment, time of sending a request and receiving data was measured on the Alluxio client node. Enabling

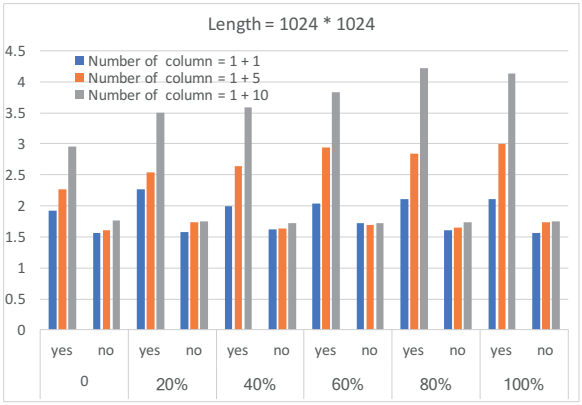

Fig. 7 Read time when the number of rows = 1024 * 1024. The number of column = 1+1 (8.1 MB), 1+5 (41 MB) or 1+10 (81 MB).

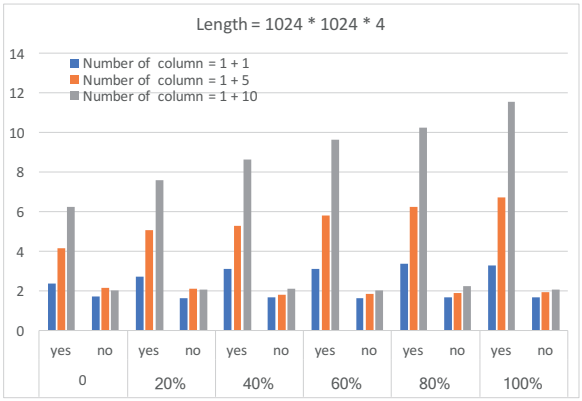

Fig. 8 Read time when the number of rows = 1024 * 1024 * 4. The number of columns = 1+1 (33 MB), 1+5 (161 MB) or 1+10 (321 MB).

and disabling the query were tested. As explained in Section 4.1, each dataset has a different number of columns and a different number of rows. In addition, the ratio of matched data is controlled for the experiment. The ratios were 0%, 20%, 40%, 60%, 80% and 100%. When the maching ratio is 0, an empty Parquet file which only contains schema information is returned.

Figure 10 shows the results of read time, when the number of rows was 1024 * 1024 and the number of column was 1+1 (8.1 MB), 1+5 (41 MB) or 1+10 (81 MB). In the figure, *yes* indicates the query is enabled. From the results, we can see that when disabling the query, the increase of the number of columns in the dataset did not affect on its performance because the extra query operation was not added. When enabling the query, the extra query operation was added and the performance was gradually degraded with the increase of the number of columns.

In order to investigate the filter performance as the dataset size becomes larger, we increased the number of rows in each dataset from the values in the previous experiment. Figure 11 shows the results when the number of rows was 1024 * 1024 * 4 and the number of columns was 1+1 (33 MB), 1+5 (161 MB) or 1+10 (321 MB). Figure 11 shows the results when the number of of rows was 1024 * 1024 * 4 and the number of columns was 1+1 (129 MB), 1+5 (643

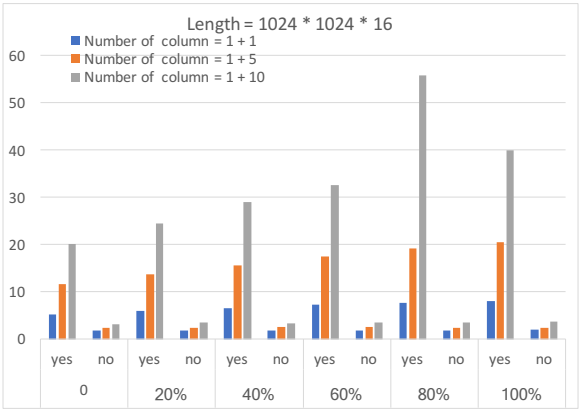

Fig. 9 Read time when the number of rows = 1024 * 1024 * 16. The number of columns = 1+1 (129 MB), 1+5 (643 MB) or 1+10 (1284 MB).

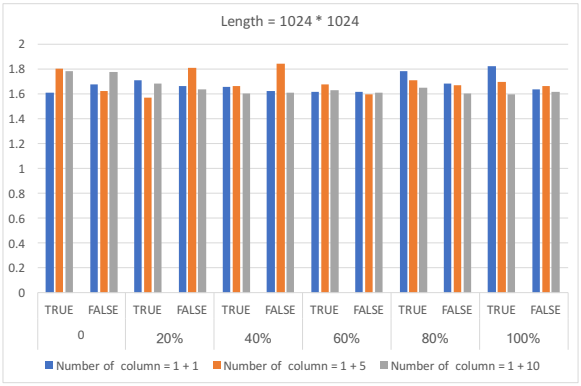

Fig. 10 Read time when the number of rows = 1024 * 1024. The number of columns = 1+1 (8.1 MB), 1+5 (41 MB) or 1+10 (81 MB). FilterPushdown = true / false.

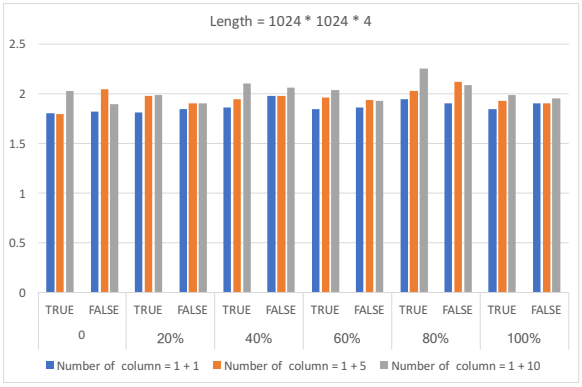

Fig. 11 Read time when the number of rows = 1024 * 1024 * 4. The number of columns = 1+1 (33 MB), 1+5 (161 MB) or 1+10 (321 MB). FilterPushdown = true / false.

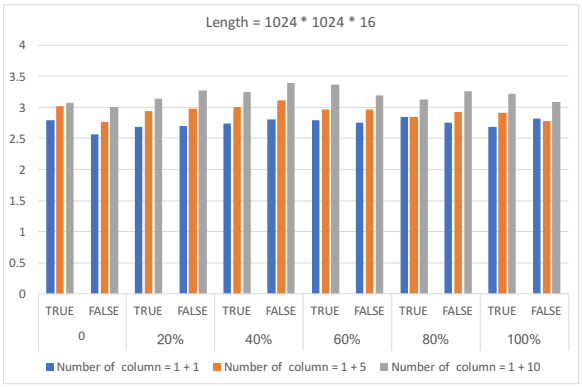

Fig. 12 Read time when the number of rows = 1024 * 1024 * 16. The number of columns = 1+1 (129 MB), 1+5 (643 MB) or 1+10 (1284 MB). FilterPushdown = true / false.

MB) or 1+10 (1284 MB). These experiment results basically have the same tendency as the previous experiment result. We could also see that the performance of read without the query was better than read with the query. As the size of the dataset increased to nearly 1.3 GB, read with the query took more than a minute. The overhead was caused by reading, matching and writing operation on the Alluxio worker but filtering cost in our implementation seems to be dominant as far as we investigated.

4.3 Performance Comparison with Filtering on Spark Executor

In the next experiment, time of sending a request and receiving data on the Spark node was measured. Then the time was compared with the case where filtering was applied on the Alluxio worker. The case was estimated from the experiment results in the previous section because our implementation was incomplete with Spark at the time of writing this paper. Here, we chose the same dataset as the previous experiment. When using filtering in Spark, "spark.sql.parquet.filterPushdown = true (default) or false" was examined. The flag enables or disables the Parquet filter push-down optimization.

Figure 10 shows the results when the number of rows was 1024 * 1024 and the number of columns was 1+1 (8.1 MB), 1+5 (41 MB) or 1+10 (81 MB). Because DataFrame of Spark has good compatibility with the Parquet file format, Parquet files can be quickly converted to the DataFrame data. The results show that filtering in Spark is clearly faster than the estimation of filtering in Alluxio. Enabling the filter push-down did not show the advantage, possibly because of the dataset size was too small.

Figure 11 shows the results when the number of rows was 1024 * 1024 * 4 and the number of columns was 1+1 (33 MB), 1+5 (161 MB) or 1+10 (321 MB). Figure 12 shows the results when the number of rows was 1024 * 1024 * 16 and the number of columns was 1+1 (129 MB), 1+5 (643 MB) or 1+10 (1284 MB). From these results, even when the dataset increased from 12MB to 1284MB, the time cost increased only 0.5 - 1 second. Compared with the experimental results of Section 4.2, which filters data on the Alluxio worker, performance of filter with Parquet format data in Spark is outstanding. Enabling the filter push-down shows slightly better performance than disabling it in some results but the advantage looks weak.

5. Discussion

We implemented the filter on the Alluxio worker node and analyzed the performance effect of it, by comparing with filtering on the Spark executor. The following were learned from the results of our experiments.

- The proposed filtering on the Alluxio worker was not

faster than filtering on the Spark side. The reason seems to be that our implementation needs decoding and encoding by the parquet-mr library and they took much time than we expected before the actual experiments.
- Filtering on the Spark side also maintained the performance against the increase of the dataset size.
- DataFrame and Catalyst optimization of Spark were implemented well with the Parquet file format. They are fast and scalable to the data size. However, this also means that our implementation could be improved with the similar techniques adopted in Spark.
- The filter push-down of Spark did not show significant performance improvement in our experiment. However, additional experiments would be necessary with other types of datasets which have different rows, columns and data types.

For encoding of the Parquet format, "snappy" was used for input datasets and "uncompressed" was used for temporarl data on the Alluxio worker. Using other encoding would be one of our future works.

6. Related Work

The idea of adding more intelligence to storage system to help processing system for better performance was mentioned by D. Slotnick in 1970 [9]. Then the concept of moving processing closer to memory or storage has been studied for decades. The active disk [10], the active storage [4] and the intelligent disk (IDISK) for decision support on the database server [11] were proposed and these concepts were attentioned repeatedly as hardware was upgraded and the processing power of the storage-side increased.

Diamond [12] aims at lowering the load on the processing-side by reducing unnecessary data using domain-specific knowledge and dynamically allocating computation to storage devices to accommodate changes in the system and network conditions. The processing task is mainly sent from the processing-side in a form of the query. The query is translated into a set of machine executable tasks to filter data. Then the filtered data was sent back to the processing-side. The goal of Diamond is to discard irrelevant data items as quickly and efficiently as possible at the storage-side rather than close to the processing-side. Other studies [13–15] address filtering out unnecessary data on the storage side as much as possible, in order to reduce the processing load before transferring data to the process side. These studies are similar to our current effort in this paper but our study targets other storage-side processing rather than filtering.

7. Conclusion

Our study aims at moving processing to the storage-side to reduce unnecessary data transfer and reducing CPU resource consumption on the processing side to improve the overall performance. In this paper, we showed our implementation and evaluation results for the filtering function on the storage side, and faced the performance problem in handling the Parquet format. Improvement and further evaluation is our first priority in future works. As a long term goal, we would like to implement other processing rather than filtering, such as data conversion, encryption, repartition, etc., and apply the similar technique to write operation.

Acknowledgments This paper is based on results obtained from a project commissioned by the New Energy and Industrial Technology Development Organization (NEDO). This work was partly supported by JSPS KAKENHI Grant Numbers JP16K00116, JP16K21675.

References

[1] Zaharia, M., Chowdhury, M., Das, T., Dave, A., Ma, J., McCauly, M., Franklin, M. J., Shenker, S. and Stoica, I.: Resilient Distributed Datasets: A Fault-Tolerant Abstraction for In-Memory Cluster Computing, *Proceedings of the 9th USENIX Symposium on Networked Systems Design and Implementation (NSDI'12)*, pp. 15–28 (2012).
[2] Dean, J. and Ghemawat, S.: MapReduce: Simplified Data Processing on Large Clusters, *Proceedings of the 6th USENIX Symposium on Operating Systems Design and Implementation (OSDI'04)*, pp. 1–13 (2004).
[3] Apache Hadoop: http://hadoop.apache.org/.
[4] Riedel, E., Gibson, G. A. and Faloutsos, C.: Active Storage for Large-Scale Data Mining and Multimedia, *Proceedings of the 24rd International Conference on Very Large Data Bases*, VLDB '98, San Francisco, CA, USA, pp. 62–73 (1998).
[5] Alluxio: https://www.alluxio.org.
[6] Stonebraker, M., Abadi, D. J., Batkin, A., Chen, X., Cherniack, M., Ferreira, M., Lau, E., Lin, A., Madden, S., O'Neil, E., O'Neil, P., Rasin, A., Tran, N. and Zdonik, S.: C-store: A Column-oriented DBMS, *Proceedings of the 31st International Conference on Very Large Data Bases*, VLDB '05, pp. 553–564 (2005).
[7] Apache Parquet: https://parquet.apache.org.
[8] Tanimura, Y.: Towards Efficient Data Staging for Multi-Tenant Big Data Analytics, *Poster session presented at the ACM Symposium on High-Performance Parallel and Distributed Computing (HPDC'16)* (2016).
[9] Slotnick, D.: Logic Per Track Devices, *Academic Press*, p. 291296 (1970).
[10] Acharya, A., Uysal, M. and Saltz, J. H.: Active Disks: Programming Model, Algorithms and Evaluation, *SIGPLAN Not.*, Vol. 33, No. 11, pp. 81–91 (1998).
[11] Keeton, K., Patterson, D. A. and M.Hellerstein, J.: A Case for Intelligent Disks (IDISKs), *ACM SIGMOD Record*, p. 4252 (1998).
[12] Larry, H., Rahul, S., Wickremesinghe, R., Mahadev, S., R., G. G., Erik, R. and Anastassia, A.: Diamond: A Storage Architecture for Early Discard in Interactive Search, *FAST '04 Conference on File and Storage Technologies*, VLDB '05, pp. 73–86 (March 31 - April 2, 2004).
[13] Bairavasundaram, L. N., Sivathanu, M., Arpaci-Dusseau, A. C. and Arpaci-Dusseau., R. H.: X-RAY: A Non-Invasive Exclusive Caching Mechanism for RAIDs, *International Symposium on Computer Architecture*, ISCA '04 (2004).
[14] Sivathanu, M., Bairavasundaram, L., Arpaci-Dusseau, A. C. and Arpaci-Dusseau, R. H.: Database-Aware Semantically-Smart Storage, *The Fourth USENIX Conference on File and Storage Technologies*, FAST '05 (December 2005).
[15] Buck, J. B., Watkins, N., Maltzahn, C. and Brandt, S. A.: Abstract Storage: Moving File Format-specific Abstractions into Petabyte-scale Storage Systems, *The Second International Workshop on Data-Aware Distributed Computing*, HDPC '09 (2009).