Stream Processing with Bigdata
by SSS-MapReduce

Hidemoto Nakada, Hirotaka Ogawa and Tomohiro Kudoh
National Institute of Advanced Industrial Science and Technology,
1-1-1 Umezono, Tsukuba, Ibaraki 305-8568, JAPAN

Abstract—We propose a MapReduce based stream processing system, called SSS, which is capable of processing stream along with large scale static data. Unlike the existing stream processing systems that can work only on the relatively small on-memory data-set, SSS can process incoming streamed data consulting the stored data. SSS processes streamed data with continuous Mappers and Reducers, that are periodically invoked by the system. It also supports merge operation on two set of data, which enables stream data processing with large static data. This poster shows overview of SSS stream processing and preliminary evaluation results.

I. INTRODUCTION

Recently, more and more data is getting easily available as streams. To efficiently analyse the streamed data, stream processing systems are proposed and getting popular.

Existing stream processing systems [2] [6] [11][9] are mainly targeting on the low-latency data processing and work only on the relatively small on-memory data-set, however. This kind of systems are very effective for specific class of applications, such as algorithm trading, but applicable area is not so large.

Although there are plenty numbers of MapReduce system that are designed to process stream data processing[3] [8] [7] [4], they just employ MapReduce as the programming API and do not provide methods to handle stream data and static large data simultaneously.

We propose SSS [10], that can process streamed data along with the stored large data, as shown in fig. 1. SSS is basically a KVS based MapReduce System, but can handle streamed data with Continuous Mapper and Reducer process which is periodically invoked by the system. SSS provides a special capability called MergeReducer to merge the streamed data with the data already stored in the system.

Note that SSS is not meant for low-latency stream processing. It cannot handle streamed data instantly, but can process streamed data based on large scaled static data store on the system. We believe that there are a wide area of applications that require this nature.

Fig. 1. Streaming Computation with Bigdata.

II. IMPLEMENTATION

A. Overview of SSS-MapReduce

1) Server Configuration: Fig. 2 shows components of SSS. Two separate services, namely SSS server and unit KVS, operate on each worker node. The SSS server is the service that executes Mappers and Reducers provided by the users. The unit KVSs from all the worker nodes compose a distributed KVS. The two services are not only share the resources, but also have a special relationship. The SSS server reads data only from the co-existing unit KVS too reduce input data latency.

SSS employs Owner Computes Rule; i.e., each Mapper/Reducer worker is responsible only for the data that resides on the same node.

2) Implementation of Distributed KVS: The distributed KVS in SSS are composed of unit KVSs on all the worker nodes. We employed simple hashing for data distribution management. When SSS servers put key-value pair to the distributed KVSs, it determines unit KVS to put with hashed value of the key. All the SSS servers shares the same hash function to guarantee that key-value pairs with the same key go to the same unit KVS.

As unit KVSs we employed Tokyo Cabinet[5], which is substantially modified so that it can handle well bulk writes for sorted key-value pairs and bulk read for a range of key. We have implemented a network service layer that wraps TokyoCabinet so that I can be accessed remotely.

3) Tuple Group: In SSS, data space is divided into several sub namespaces called 'Tuple Group'. Mappers and Reducers read input from tuple group(s) and write the output into tuple group(s).

Fig. 2. Overview of SSS.
B. Stream Processing in SSS

1) Stream Input and Output: In SSS, stream input is represented as a continuous writes to a specific tuple group. The tuple group works as input buffer for the input stream. Processing Mapper / Reducer will read from the tuple group.

Output stream is also represented as a continuous reads from a specific tuple group.

2) Periodic Mapper / Reducer: We implemented streamed data processing by invoking Mappers and Reducers continuously and periodically. The interval of periodical invocation can be specified by the user. The Mappers and Reducers reads and delete Key Value Pairs from the specified tuple Group, to ensure that one Key Value Pair is not processed more than once.

3) MergeReducer: The MergeReducers are special Reducer that can handle inputs more than one tuple groups. MergeReducers work just like merge sort. The inputs for the MergeReducers are a Key and more than two Value lists. By assigning one input to the stream input buffer, we can describe algorithms that refers streamed data and static large data (fig. 3).

III. PRELIMINARY EVALUATION

We have performed a preliminary evaluation to know data stream handling throughput of SSS on one node.

The input data was randomly generated so that they mimic the Apache Web Server log records. The record size was about 300 bytes. We repeatedly put 10000 records with 10ms interval. We used a server equipped with Intel(R) Xeon(R) W5590 3.33GHz and Fusion-io ioDrive Duo 320GB.

We set continuous Mapper interval as 2, 5, and 10 seconds. The results are shown in fig. 4. As we can see, in all the cases, 0.14Mi records/s are accomplished. We can also observe that there are periodical throughput degradation. This is obviously caused by the periodically invoked Continuous Mapper.

IV. CONCLUSION

We described overview of the streamed data processing with SSS: a KVS based MapReduce system and results of preliminary evaluation.

We confirmed that periodic reads by Mappers/Reducers interfere streamed data writes. This is caused by the data lock that is necessary to avoid race condition. We are planning to re-design the datafile management of SSS so that it can handle multiple files with file rotation. This modification will allows read and write

ACKNOWLEDGEMENT

This work was partly funded by the New Energy and Industrial Technology Development Organization (NEDO) Green-IT project.

REFERENCES