Sebo: Selective Bulk Analysis Optimization in Big Data Processing

Jiangling Yin, Jun Wang, Tyler Lukasiewicz, Jian Zhou, Xuhong Zhang and Dan Huang
Department of Electrical Engineering and Computer Science
University of Central Florida, Orlando, FL
{jyin, jwang, xzhang, dhuang}@eecs.ucf.edu

***Abstract*—Selective bulk analyses, or the process of analyzing specific sub data sets for various purposes is fundamental to a wide range of contemporary data analyses. However, with the increasing size of data-sets, data organization/processing becomes more challenging for selective bulk analysis in current big data processing frameworks such as Spark. This is because these frameworks provide methods based on coarse-grained transformation, i.e. they need to filter through the entire data set for every selective bulk analysis regardless of which data is actually needed. In this paper, we propose a content-aware method to optimize selective bulk analysis in big data processing referred to as Sebo. Sebo maintains in-memory data organization metadata so as to support fast lookup and selection. Through Sebo, selective analysis programs can efficiently identify their needed data. Sebo is able to save memory as well as computation in comparison to current data processing methods.**

***Index Terms*—Scientific Data, In-memory Processing, Index**

I. Introduction

Many temporal/spatial data analyses usually execute on subsets of a given dataset, this is referred to as selective bulk data analysis in this paper. Selective bulk analysis plays an important role in many analysis activities such as periods comparison/evaluation, knowledge discovery, event reasoning, and prediction. For instance, in selective bulk analysis, statistical methods [**?**] such as Moving Average or Stationarity Computation are usually applied to investigate how the data changes within a period of time. Also, methods such as pattern extraction or distance comparison [**?**] could be employed to perform trends/seasonality analysis, which can be used to predict future trends, e.g, weather forecasts or stock price predictions. Moreover, machine learning algorithms [**?**] such as model training usually group data into different subsets in order to capture a precise prediction model.

In today's big data era, data organization and processing has become more challenging. Currently, MapReduce [**?**] is becoming the de-facto programming model for large data processing. Spark [**?**] is a popular open-source framework, which implements the MapReduce data processing model and provides Resilient Distributed Datasets (RDDs) that are partitioned in memory collections of data items. Spark allows the partitioned data to reside in memory for repeatedly or interactively processing. This can greatly improve the execution performance for interactive or iterative data processing.

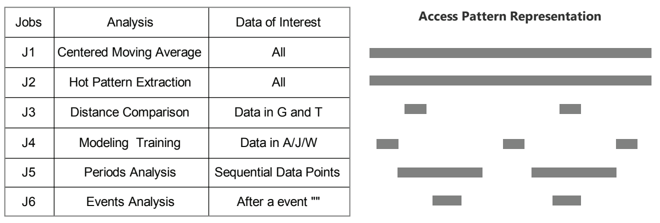
Unfortunately, Spark provides interfaces based on coarse-grained transformations, i.e, it will filter through the entire data set for every selective bulk analysis regardless of which data is actually needed. Such a coarse-grained data processing model is inefficient for selective bulk data analysis. This is because selective bulk analysis programs may only need to access part of the data throughout the entire execution. For instance, periods analysis such as Distance Comparison will only need the data in two or three specific periods. Thus, in order to prepare the data for selective bulk analysis, an extra cost on computation and memory will be needed to scan/filter the whole data sets. One main challenge involved in selective bulk analysis is finding a method such that the required data partitions can be efficiently targeted and accessed.

In this paper, we implement Spark as the basic implementation platform for our method. This is because selective bulk analysis often involves interactive analysis and data partitions need to be accessed multiple times. This type of analysis is much more efficient when the data are resident in memory. We propose a content-aware method to optimize selective bulk analysis within Spark Our method can efficiently identify and access the needed data for selective bulk analysis programs without filtering through the whole data set, thus enabling us to save memory and computation costs in comparison to the default data processing method.

II. Selective Bulk Analysis and In-Memory Data Processing

Selective Bulk Analysis In practice, given temporal/spatial data sets such as time series, multiple interactive analyses could be involved. Specifically, we illustrate several commonly used methods in selective bulk analysis as follows.

- *Distance Comparison* is used to study how two or more time series differ at specific periods of the time. It could be used in seasonality trends analysis or pattern extraction. For instance, in Meteorology, to compare the temperatures in Florida throughout 1940 and 2014, the high and low temperatures on each date of 1940 would be compared with that date in 2014.
- *Modeling Training* is to build a prediction model with the use of existing data. In model training, data are usually grouped into different subsets, e.g, Training, Tests and Validation. For example, we can randomly select 10 years of weather data to train a model and use the remaining years' data for Tests and Validation.



Jobs	Analysis	Data of Interest
J1	Centered Moving Average	All
J2	Hot Pattern Extraction	All
J3	Distance Comparison	Data in G and T
J4	Modeling Training	Data in A/J/W
J5	Periods Analysis	Sequential Data Points
J6	Events Analysis	After a event ""

Fig. 1: Common access patterns in selective bulk analysis.

- *Events Analysis* is used to investigate the cause/effect of a special event. For instance, in telephone security, fraud can be detected by comparing the distributions of typical phone calls and of calls made from a stolen phone.

We illustrate the typical data access patterns for these data analysis methods in figure 1. In real execution, all of these methods could access the entire data set or subsets of the data set.

In-memory Big Data Processing Currently, Spark is a popular in-memory MapReduce processing framework. Spark introduces resilient distributed datasets (RDDs) to facilitate the programming of parallel applications. Each RDD represents a collection of data partitions. We present a typical example of how to run a spark application. The application reads data from a file system, filter the error messages and then counts them using the map and reduce interface.

```
val file = spark.textFile("//data...")
val errs = file.filter(_.contains("ERROR"))
val ones = errs.map(_ => 1)
val count = ones.reduce(_+_)
```

The data flow is shown in Figure 2. As we can see, a new RDD will be generated after applying each data operation. To collect the lines including the text of *error*, all n data partitions (rdds) will call the filter operation and find the valid data items.

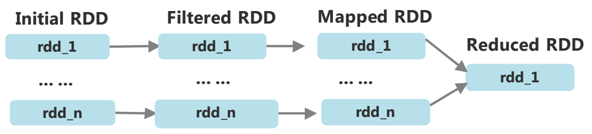


Fig. 2: A example of data flow.

However, since selective bulk data analyses usually involve analysis of sub data sets and the default processing workflow in Spark can cause inefficiencies during execution. For instance, to perform a period analysis, a filter operation is usually needed to execute on all data partitions in order to select the valid period data. This requires the program to scan and filter the whole data set, which creates an extra memory cost to store the new filtered data. Such a filter operation is necessary in coarse-grained data processing frameworks. This is because the content in each data partition is unknown without scanning through it.

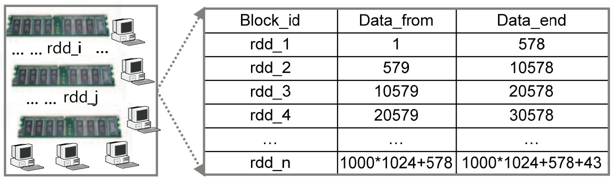
III. Content-aware In-Memory Data Organization

In order to efficiently support selective bulk data analysis in spark, we propose a content-aware method to allow analysis programs to efficiently access their needed data without thorough scanning/filtering all the data partitions. There are two advantages to a content-aware method. (1) Memory efficiency: we don't need extra memory to store unneeded filtered dataset, e.g. *_filterRDD* (2) Computational efficiency: data selection with a content-aware method is much faster than filtering through all partitions.

A. *Table_based Content_Aware Data Organization*

To support selective bulk data analysis efficiently, we record the metadata of each data partition (rdd). In this paper, the metadata mainly refers to the data index that is the major filter condition used in temporal/spatial data such as time property. An intuitive way to maintain the metadata for each data partition is to use a table, similar to the technique adopted in databases. The key and the value are the id of data partitions and the data range of each partition respectively, as shown in Figure 3. With the help of this metadata table, we can identify the data range for all data partitions. If we need to select the data items ranging from index i to j, we can use a binary search to find which partition contains the data item with index of i and j respectively, then all the partitions between them in the table are the targeted data items.

However, such an intuitive method may encounter some challenges with the increasing number of data partitions (rdds). Firstly, the space complexity of a table-based method is $O(m)$, where m is the number of data partitions. This implies that the memory space will grow linearly with the increasing number of blocks. Secondly, the lookup time in table-based design is related to the size of table and the average lookup time should be $O(log\ m)$. With the increasing size of data, this could place a substantial burden on the application driver/scheduler, which is based on a centralized architecture.



Block_id	Data_from	Data_end
rdd_1	1	578
rdd_2	579	10578
rdd_3	10579	20578
rdd_4	20579	30578
…	…	…
rdd_n	1000*1024+578	1000*1024+578+43

Fig. 3: Table-based content-aware data organization for selective bulk analysis.

B. *Compressed Index with Associated Search List (CIAS)*

As discussed, the table-based method could be inefficient regarding time and space. In this section, we propose a more efficient method to capture the mapping relationship between partitions' IDs and their data ranges. Our goal is to find a method such that the overhead of metadata organization and lookup does not increase with the growing size of data or number of data partitions. We format our problem as, given the id of data partitions and their data ranges as shown in

Figure 3, find a method to capture their relationship such that the memory cost is not affected by the size of the table. Because of this relationship, the table does not need reside in memory.

To solve the problem, we propose a data structure called Compressed Index with Associated Search List (CIAS) to record the relationship between the data range with the id of data partitions. The design is based on the following facts, (1) the distributed partitions (rdds) in Spark usually have the same size, e.g, 32 MB or 64 MB, thus each partition contains the same number of data items. (2) data with a time property could have a fixed size on each period, for instance, the weather data or stock prices. CIAS represents the table in a compact way and performs data range lookup in a computational fashion. For instance, the table in Figure 3 could be represent as following. To find which partition contains the i^{th} item, we firstly identify i in the ASL and then compute the partition id with the use of compressed index.

```
Compressed Index: 578, 10000^(n-2), 43
AssociatedSearchList(ASL): 578, (n-2)X1000+578, 1**621
```

IV. Experiment

We have conducted preliminary experiments on Marmot. *Marmot* is a cluster of the PRObE on-site project [**?**] that is housed at CMU in Pittsburgh. The system has 128 nodes / 256 cores and each node in the cluster has dual 1.6GHz AMD Opteron processors, 16GB of memory, Gigabit Ethernet, and a 2TB Western Digital SATA disk drive. For our experiments, all nodes are connected to the same switch. On *Marmot*, Spark [1.0.2] is installed as big data in-memory processing framework on the cluster nodes running CentOS55-64 with kernel 2.6.

A. Sebo Evaluation

To test Sebo, we use a benchmark application which interactively processes a data set on different periods. The experiments data is a time series, which has the similar data format to the climate data with the properties of *time, temperature, humidity, wind speed and direction*. The size of our dataset is around 480 MB and the data is divided into 15 partitions after loading into memory. We process the dataset via two methods. The first method is to use the default data processing interface in Spark, in which we firstly *load/reside* the data into memory, then we apply *filter* operation to obtain our target period data and finally we perform statistic learning operations on the filtered bulk data. The second method is our proposed method: Sebo, which records the content range for each data partition. Instead of scanning all data partitions during filter operation, we can directly identify the data partition that contain our target data with the use of Sebo. In our experiment, 5 bulk data from different periods are selected to do analysis, as shown in Figure 4. For each period, we do three basic statistic analysis on *temperature* property: computing the *max, mean* and *standard deviation* of the selected data. We mainly present the performance comparison on memory and processing time in this paper.

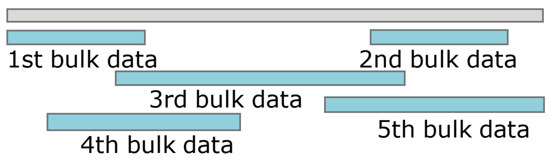


Fig. 4: The pattern of five bulk data are selected during periods analysis.

The execution includes five phases according to the selected five periods. After finishing each phase, we monitor the total used memory. The memory comparison is shown in Figure 5. With the use of default method, we can find that the memory usage is increasing after each analysis. This is because in each analysis phase, more RDDs are created and they are resident in memory by default. The final accumulated used memory is around 1800MB, which is about 3.8X to the raw input data. On the other hand, with the use of Sebo, we can achieve a much lower memory cost than the default method. The used memory is almost not increasing. This is because we don't need to save "filtered" RDDs in comparison to the default method. In general, we can find the memory cost is half that of without Sebo after the analysis on the third phase, and a third for the fifth phase. This shows our method is efficient during bulk data analysis.

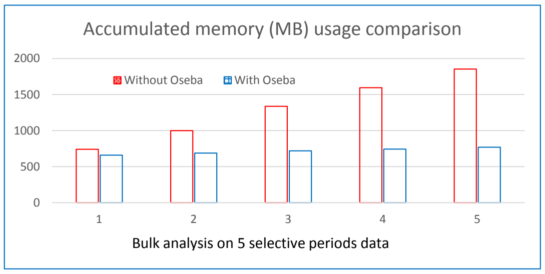


Fig. 5: The memory cost comparison for selective bulk analysis on five periods.

We also collected the accumulated time based on the five phases and the result is shown in Figure 6. Clearly, we can find less time is spent with the use of Sebo in comparison to that of without Sebo. There is a little improvement during the first phase. After that, the processing time gap become much bigger. The total processing time is more than 120 seconds without the use of Sebo while that is around 70 seconds with the use of Sebo. This could be explained since thorough scanning the data during *filter* operation is expensive for selective bulk data analysis without the use of Sebo. In fact, a larger size of raw data can result in a bigger time consumption during data selection.

V. Related work

There are a great deal of frameworks or systems that are proposed to manage and process big data. The Hadoop File System is an open source community response to the Google

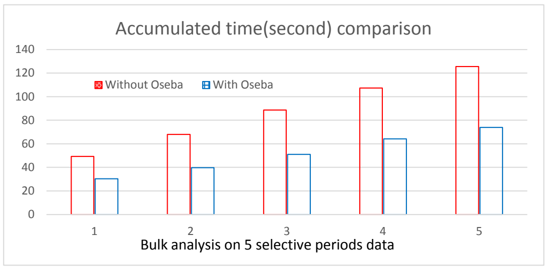


Fig. 6: The time cost comparison for selective bulk analysis on five periods.

File System (GFS), specifically for the use of MapReduce style workloads [?]. Dryad [?] and Spark [?] are two other frameworks to support big data processing with the similar interface to MapReduce. Sparkler [?] extends Spark to support distributed stochastic gradient descent. However, these systems or frameworks usually adopt coarse-grain data processing, e.g, applying operations to all data items, which is inconvenient for selective bulk data analysis due to a subset of data being involved. There are systems that can support fine-grained data processing. Example of these systems are keyvalue stores [?], databases, and Piccolo [?] and they provide interfaces to support fine-grained data items/cells updates/processing. However, these frameworks and systems need extra cost for maintaining reliability as discussed in [?], while bulk data analysis are more about coarse-grained data processing on the sub data set.

VI. Conclusion

In this paper, we investigate the problems of selective bulk analysis on big data in-memory processing frameworks, e.g, Spark. Due to the missing information of blocks' content, selective bulk analysis programs needs to scan thorough the whole data set in order to find the valid data partitions, resulting in extra computation and memory overheads. To address this problem, we propose a content-aware data organization method to help selective bulk analysis. With the use of our method (Sebo), selective bulk data analysis program can easily access their needed data. We conduct preliminary experiments for our proposed method on PRObEs Marmot and the experimental results show the promising performance of Sebo.

Acknowledgments

This material is based upon work supported by the National Science Foundation under the following NSF program: Parallel Reconfigurable Observational Environment for Data Intensive Super-Computing and High Performance Computing (PRObE).