Journal of Theoretical and Applied Information Technology
15th November 2020. Vol.98. No 21
© 2005 – ongoing JATIT & LLS

ISSN: **1992-8645** www.jatit.org E-ISSN: **1817-3195**

SCALABLE PARALLEL BIG DATA SUMMARIZATION TECHNIQUE BASED ON HIERARCHICAL CLUSTERING ALGORITHM

[1]VERONICA S. MOERTINI and [2]MATTHEW ARIEL

Informatics Department, Parahyangan Catholic University, Bandung Indonesia

[1]moertini@unpar.ac.id, [2]matthew.ariel.wangsit@gmail.com

ABSTRACT

Data reduction or summarization techniques can be applied to obtain a reduced representation of the data set that is much smaller in volume, yet closely maintains the integrity of the original data. For summarizing data, Agglomerative hierarchical clustering algorithms, has few advantages. It is quite simple, can produce summaries at specific level (in the form of cluster patterns) with simple adjustment, and can be paralyzed.

Apache Spark is a data processing framework that can quickly perform processing tasks on very large data sets, and it is currently a standard tool for analyzing big data. In processing data, Spark can distribute data processing tasks across multiple machines. Spark can run on top of Hadoop with its distributed file system (HDFS) and resources management (YARN) such that it can access HDFS files and uses network resources efficiently. To achieve high performance and scalability of the data analysis technique in Spark environment, stages, narrow and wide transformation, cost of I/O and network must be considered.

In this research, we develop a data summarization technique by employing Agglomerative algorithm for Spark. To avoid biased results, records in the given big data are randomly split into bags of dataset stored as Resilient Distributed Datasets (RDD) partitions in the worker machines. To reduce network and I/O cost, we employ one wide transformation that involves data shuffling across the network. In addition, the RDD partitions are then processed locally to produce cluster patterns by worker tasks. Functions with complex computations are designed as Spark parallel tasks.

A series of experiments were conducted on a Spark cluster with one driver and ten worker nodes by varying data size (5 to 20 Gb), machine cores used (10 to 50) and the application variables (data split and maximum objects/dendrogram). The results show that the technique is scalable and efficient. The execution time is mostly determined by the parallel tasks run locally on the workers.

Keywords: *Big Data Summarization, Parallel Agglomerative, Apache Spark Application Design*

1. INTRODUCTION

Big data is known with its volume, velocity, variety, veracity and value characteristics [1, 2]. The main characteristic that makes data "big" is the volume, which can be from gigabytes up to zettabytes and beyond. The high frequency of incoming data, that is the velocity, causes the big data to "grow" fast. In other words, big data volume increases quickly. Values can be obtained from big data by employing data analysis techniques, which are based on algorithms, statistics, Machine Learning, pattern recognition, database systems, visualization, and so on [3]. However, to the best of our knowledge, to these days many technologies running on non-distributed system, despite its rich features for analyzing data, have not supported big data analysis technique.

Data reduction or summarization techniques can be applied to obtain a reduced representation of the data set that is much smaller in volume, yet closely maintains the integrity of the original data [3, 4, 5]. The smaller dataset can further be analyzed using data analysis technologies to obtain the insights.

Data reduction strategies include dimensionality reduction, numerosity reduction, and data compression. Numerosity reduction techniques replace the original data volume by alternative, smaller forms of data representation, which can be generated by nonparametric methods such as histograms, clustering, sampling, and data cube.

Hierarchical clustering algorithms work by grouping data objects into a hierarchy or "tree" of clusters. Representing data objects in the form of a hierarchy is useful for data summarization and visualization [3]. Thus, these algorithms can be adopted to summarize big data having numerical attributes, for instance, data gathered by sensors or other measuring instruments. Numerical big data can also be produced by transforming non-numerical data (i.e. texts, images, etc.) into numerical one.

Apache Spark is a data processing framework that can quickly perform processing tasks on very large data sets [2]. It is currently the most actively developed open source engine for these tasks, making it a standard tool for any developer or data scientist interested in big data [6]. Spark can distribute data processing tasks across multiple machines such that it is able to process large quantities of data, beyond what can fit on a single machine, with a high-level API. It allows us to write the logic of data analysis algorithms in a way that is parallelizable [7]. So, it is often possible to write computations that are fast for distributed storage systems of varying kind and size. However, despite its advantages, the simplest implementation of many common data science routines in Spark can be much slower and much less robust than the best version. Since the computations may involve data at a very large scale, the time and resources that gains from designing the algorithms and tuning code for performance are enormous. In this context, I/O cost and shuffling data objects across machines should be minimized.

Apache Hadoop is a platform for storing and analyzing big data in distributed systems [8, 9]. Its distributed file system, namely Hadoop Distributed File System (HDFS), is designed to reliably store very large files across machines in a large cluster. Each HDFS file is stored as a sequence of blocks, which are replicated for fault tolerance. Apache Hadoop resource management and job scheduling, namely YARN (Yet Another Resource Negotiator), supports application scalability. In doing so, YARN employs two independent daemons. The first is a resource manager to manage the use of resources across the cluster, the second is an application master to manage the lifecycle of applications running on the cluster [9].

One of the basic hierarchical clustering algorithms, is Agglomerative [3]. This algorithm is quite simple. It produces hierarchy of clusters as tree, namely dendrogram, from a given dataset. This dendrogram can then be "cut" using a certain cut-off distance, to obtain clusters from the dataset. This cut-off can be adjusted to obtain less or more clusters. In term of data reduction or summarization, in which a summary is computed from each cluster, the cut-off can be used to "tune" the reduction level. Here, the cluster patterns [10] and feature trees [11] can be used as the summary. Although due to its time complexity, Agglomerative is suitable for small dataset, we find that it can be parallelized. Spark parallel tasks of Agglomerative can be designed to process small dataset read from HDFS. As each core in a machine can run one executor, and each executor may run more than one task, we find that scalability can be achieved.

In our past research, we have developed big data reduction technique based on Hadoops MapReduce framework [5]. It reduces big data based on Agglomerative clustering algorithm. However, its scalability is not satisfying. When the objects in a given big data is randomly split into a large number of partitions, then a dendrogram is built from each partition, its performance decreases significantly. In this research, the technique is re-designed based on Spark framework that runs on top of Hadoop YARN and HDFS. To reduce network and I/O cost, we employ one wide transformation that involves data shuffling across the network, and the RDD partitions are then processed locally to produce cluster patterns by worker tasks. Functions with complex computations are designed as Spark parallel tasks.

Within our research results, we intend to contribute a framework of a scalable application for Spark. It involves one wide transformation that involve data shuffling across the network. Functions with complex computation are implemented as parallel tasks.

This paper is organized as follows: Literature review of Hadoop with its HDFS and YARN; Spark concept, hierarchical clustering and Agglomerative algorithm, evaluation of the technique discussed in

[5]; our proposed technique with is high level concept and detailed designs, experiments for evaluating the technique with its results and analysis, and conclusions.

2. LITERATURE REVIEW

2.1. Hadoop Distributed Storage and YARN

Hadoop is a platform for storing and analyzing big data in distributed systems [8, 9]. It comes with master-slave architecture and support commodity machines. Hadoop has two main components, which are MapReduce and Hadoop Distributed File System (HDFS). MapReduce is a programming model for data processing.

The Hadoop Distributed File System (HDFS) is a distributed file system designed to run on commodity hardware. It is designed to reliably store very large files across machines in a large cluster. Each HDFS file is stored as a sequence of blocks where all blocks except the last block are the same size. The blocks of a file are replicated for fault tolerance (the default is three replicas). For each file, the block size and replication factor are configurable. An application (client) may specify the number of replicas of a file. Files in HDFS are write-once (except for appends and truncates) and have strictly one writer at any time.

With its with master-slave architecture, in the Hadoop cluster, datanodes process and store data blocks, while Namenode manages the datanodes (see Figure 1). Namenode makes all decisions regarding replication of blocks, maintains data block metadata, and control client access. It periodically receives a heartbeat and a block report from each datanode in the cluster. Upon receiving a heartbeat, Namenode knows that the datanode (who send that heartbeat) is functioning properly. A block report contains a list of all blocks on a Datanode.

Namenode stores the file metadata, which include the file name, file permissions, IDs, locations, and the number of replicas. These are stored in the its local memory. Should a Namenode fail, HDFS would not be able to locate any of blocks distributed throughout the datanodes. This vulnerability is addressed by implementing a Secondary Namenode or a Standby Namenode.

A datanode communicates and accepts instructions from Namenode roughly twenty times in a minute. Once an hour, each datanode reports the status and health of its data blocks. Based on the provided information, Namenode can request a datanode to create additional replicas, remove them, or decrease the number of data blocks present on the node.

Apache Hadoop YARN (Yet Another Resource Negotiator) is the resource management and job scheduling employed in MapReduce 2. YARN remedies the scalability shortcomings of “classic” MapReduce 1 by splitting the responsibilities of the jobtracker into separate entities [9]. That jobtracker takes care of both job scheduling and task progress monitoring.

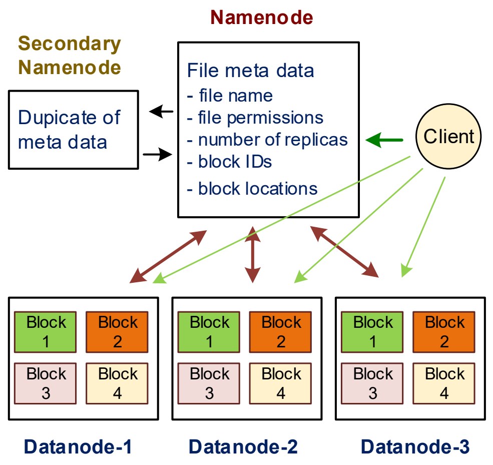

Figure 1. The Hadoop Distributed File System (Hdfs).

YARN separates these two roles into two independent daemons: a resource manager to manage the use of resources across the cluster and an application master to manage the lifecycle of applications running on the cluster. An application master negotiates with the resource manager for cluster resources—described in terms of a number of containers, each with a certain memory limit—and then runs application-specific processes in those containers. The containers are overseen by node managers running on cluster nodes, which ensure that the application does not use more resources than it has been allocated.

YARN involve entities, which are: (a) The client, which submits the job, which can be Spark Job. (b) The YARN resource manager, which coordinates the allocation of compute resources on the cluster. (c) The YARN node managers, which launch and monitor the compute containers on machines in the cluster. (d) The Application Master, which coordinates the tasks running the job. The application master and the tasks run in containers that are scheduled by the resource manager and managed by the node managers. (e) The distributed filesystem (normally HDFS), which is used for sharing job files between the other entities

The per-application Application Master is a framework specific library having task for negotiating resources from the Resource Manager and working with the node manager(s) to execute and monitor the tasks.

2.2. Spark

High Level Overview

Spark provides a high-level query language to process data. Spark Core, the main data processing framework in the Spark ecosystem, has APIs in Scala, Java, Python, and R. Spark is built around a data abstraction called Resilient Distributed Datasets (RDD).

On its own, Spark is not a data storage solution. It performs computations on Spark JVMs (Java Virtual Machines) that last only for the duration of a Spark application [6, 7]. Spark can be run locally on a single machine with a single JVM (called local mode). More often, Spark is used in tandem with a distributed storage system (e.g., HDFS, Cassandra, or S3) and a cluster manager—the storage system to house the data processed with Spark, and the cluster manager to orchestrate the distribution of Spark applications across the cluster. Spark currently supports three kinds of cluster managers: Standalone Cluster Manager, Apache Mesos, and Hadoop YARN. In this research, Spark is configured to run using Hadoop YARN (Figure 2).

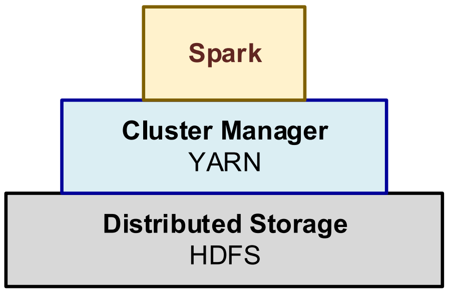

Figure 2. Spark On YARN And HDFS.

Spark Model of Parallel Computing: RDD

An RDD in Spark is simply an immutable distributed collection of objects. Each RDD is split into multiple partitions, which may be computed on different nodes of the cluster [2]. RDDs can contain any type of Python, Java, or Scala objects, including user defined classes. Users create RDDs in two ways: by loading an external dataset, or by distributing a collection of objects (e.g., a list or set) in their driver program.

Once created, RDDs offer two types of operations, which are transformations and actions**.** Actions are functions that return values that are not an RDD, and transformations are functions that return another RDD. Each Spark program must at least contain an action, since actions either bring information back to the driver or write the data to stable storage. The first include collect, count, collectAsMap, sample, reduce, and take. The second include saveAsTextFile, saveAsSequenceFile, and saveAsObjectFile. In short, actions compute a result based on an RDD, and either return it to the driver program or save it to an external storage system (e.g., HDFS). Actions are what force evaluation of a Spark program.

Transformations construct a new RDD from a previous one. Most of the power of the Spark API is in its transformations. Spark transformations are "coarse-grained" transformations used to sort, reduce, group, sample, filter, and map distributed data. RDDs have a number of predefined coarse-grained transformations (functions that are applied to the entire dataset), such as map, join, and reduce to manipulate the distributed datasets, as well as I/O functionality to read and write data between the distributed storage system (such as HDFS) and the Spark JVMs [7].

Transformations and actions are different because of the way Spark computes RDDs. Although new RDDs can be created at any time, Spark computes them only in a lazy fashion—that is, the first time they are used in an action.

Spark allows users to write a program for the driver (or master node) on a cluster computing system that can perform operations on data in parallel [7]. The Spark cluster manager handles starting and distributing the Spark executors across a distributed system according to the configuration parameters set by the Spark application. The Spark execution engine itself distributes data across the executors for a computation (see Figure 3).

Spark can keep an RDD loaded in-memory on the executor nodes throughout the life of a Spark application for faster access in repeated computations. RDDs are immutable, so transforming an RDD returns a new RDD rather than the existing one [7].

Spark Application

A Spark application corresponds to a set of Spark jobs defined by one SparkContext in the driver program [7]. A Spark application begins when a SparkContext is started, which cause a

Journal of Theoretical and Applied Information Technology
15th November 2020. Vol.98. No 21
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

driver and a series of executors are started on the worker nodes of the cluster (Figure 3). Each executor corresponds to a Java Virtual Machine (JVM), and an executor cannot span multiple nodes although one node may contain several executors. The SparkContext determines how many resources are allotted to each executor. When a Spark job is launched, each executor has slots for running the tasks needed to compute an RDD. When RDD are created by loading an external dataset from HDFS, the default configuration is an RDD partition correspond to a block of HDFS. An illustration of a worker node with 2 executors that process 3 HDFS blocks stored in this worker node is presented on Figure 4.

One Spark cluster can run several Spark applications concurrently. The applications are scheduled by the cluster manager and correspond to one SparkContext. Spark applications can run multiple concurrent jobs. Jobs correspond to each action called on an RDD in an application.

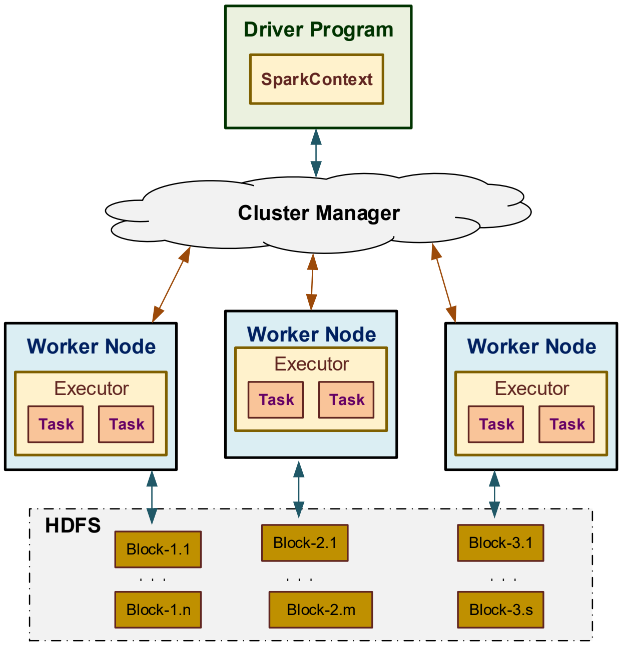

Figure 3. Spark Cluster Running On Top Of HDFS Distributed Storage.

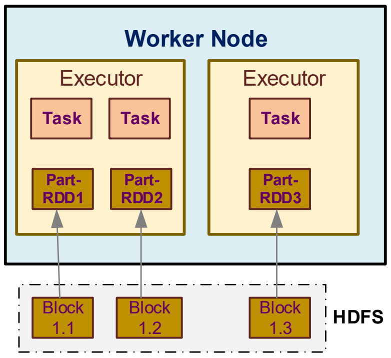

Figure 4. A Worker Node With 2 Executors That Process 3 HDFS Blocks.

The Anatomy of a Spark Job

Spark evaluates RDDs lazily. A Spark application does not “do anything” until the driver program calls an action. With each action, the Spark scheduler builds an execution graph (a directed acyclic graph or DAG) and launches a Spark job. Each job consists of stages, which are steps in the transformation of the data needed to materialize the final RDD. Each stage consists of a collection of tasks that represent each parallel computation and are performed on the executors [2, 7].

A DAG is built based on the dependencies between RDD transformations. Hence, Spark evaluates an action by working backward to define the series of steps it has to take to produce each object in the final distributed dataset. Then, using these series of steps or the execution plan, the scheduler computes the missing partitions for each stage until it computes the result of the action.

Figure 5 shows a tree of the different components of a Spark application and how these correspond to the API calls. An application corresponds to starting a SparkContext/SparkSession. Each application may contain many jobs that correspond to one RDD action. Each job may contain several stages that correspond to each wide transformation. Each stage is composed of one or many tasks that correspond to a parallelizable unit of computation done in each stage. There is one task for each partition in the resulting RDD of that stage. More discussion of stage and task is as follows.

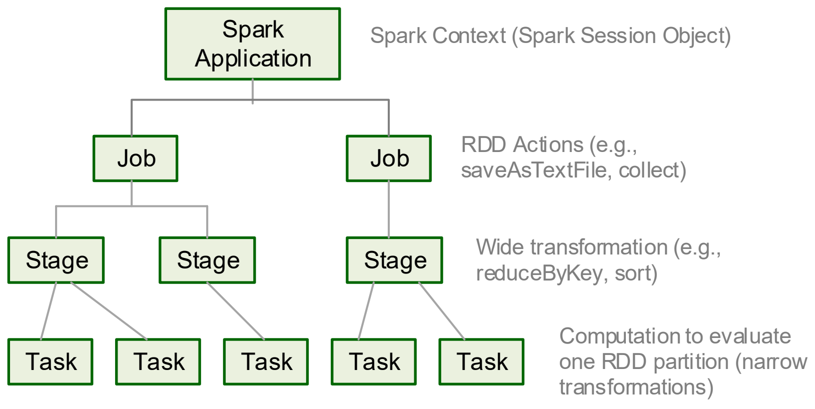
Spark Application
Spark Context (Spark Session Object)
Job
Job
RDD Actions (e.g., saveAsTextFile, collect)
Stage
Stage
Stage
Wide transformation (e.g., reduceByKey, sort)
Task
Task
Task
Task
Task
Computation to evaluate one RDD partition (narrow transformations)

Figure 5. The Spark Application Tree.

Stages: Each stage corresponds to a shuffle dependency created by a wide transformation in the Spark program. One stage can be thought of as the set of computations (tasks) that can each be computed on one executor without communication with other executors or with the driver. Thus, a new stage begins whenever network communication between workers is required, such as in a shuffle. (Shuffles are caused by those wide transformations, such as sort or groupByKey, which require the data to be redistributed across the partitions.) Several transformations with narrow dependencies can be grouped into one stage.
Because the stage boundaries require communication with the driver, the stages associated with one job generally have to be executed in sequence rather than in parallel. (It is possible to execute stages in parallel if they are used to compute different RDDs that are combined in a downstream transformation such as a join.) To achieve better performance, it is necessary to design Spark's program to require fewer shuffles.

Tasks: A stage consists of tasks. The task is the smallest unit in the execution hierarchy, and each can represent one local computation. All of the tasks in one stage execute the same code on a different piece of the data. One task cannot be executed on more than one executor. However, each executor has a dynamically allocated number of slots for running tasks and may run many tasks concurrently throughout its lifetime. The number of tasks per stage corresponds to the number of partitions in the output RDD of that stage.

Wide Versus Narrow Dependencies

The narrow versus wide distinction has significant implications for the way Spark evaluates a transformation and for its performance [6, 7]. When designing algorithm for Sparks, thus, it becomes important consideration.

Narrow transformations are those in which each partition in the child RDD has simple, finite dependencies on partitions in the parent RDD. Dependencies are only narrow if each parent of RDD partition has at most one child partition. Specifically, partitions in narrow transformations can either depend on one parent (such as in the map operator), or a unique subset of the parent partitions that is known at design time (coalesce). Therefore, narrow transformations can be executed on an arbitrary subset of the data without any information about the other partitions. Figure 6 illustrates dependency graph for transformations with narrow dependencies.

(a)

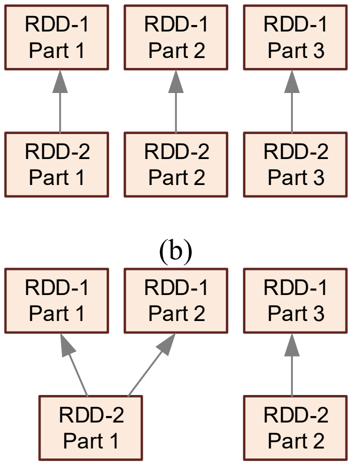

Figure 6. Dependencies Between RDD Partitions For Narrow Transformations.

In contrast, *transformations with wide dependencies* require the data to be partitioned in a particular way (Figure 7). For instance, records from the whole data have to be partitioned so that keys in the same range are on the same partition. Transformations with wide dependencies include sort, reduceByKey, groupByKey, join, and anything that calls the rePartition function. These transformations involve shuffling records among RDD partitions (that could be spread in worker nodes). As shuffles are expensive, this transformation becomes more expensive with more data and when a greater proportion of that data has to be moved from particular RDD partitions to other partitions during the shuffle.

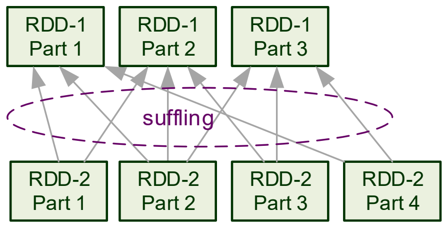

Figure 7. Dependencies Between RDD Partitions For Wide Transformations.

In-Memory Persistence and Memory Management

Spark's performance advantage over MapReduce is greatest in use cases involving repeated computations. Much of this performance increase is due to Spark's use of in-memory persistence. Rather than writing to disk between each pass through the data, Spark has the option of keeping the data on the executors loaded into memory, and is available in-memory each time it needs to be accessed.

Spark offers three options for memory management: in-memory as deserialized data, in-memory as serialized data, and on disk. In-memory as deserialized data is the fastest, since it reduces serialization time, however, it may not be the most memory efficient, since the data is stored as objects (see [7] for more discussion). The persist() function in the RDD class lets the user control how the RDD is stored. By default, persist() stores an RDD as deserialized objects in memory, but the user can pass one of numerous storage options to the persist() function to control how the RDD is stored.

2.3. Hierarchical Clustering and Agglomerative Algorithm

Clustering techniques consider data tuples as objects. They partition the objects into groups, or *clusters*, so that objects within a cluster are "similar" to one another and "dissimilar" to objects in other clusters. In data reduction, the cluster representations of the data are used to replace the actual data [3]. The effectiveness of this technique depends on the data's nature. It is much more effective for data that can be organized into distinct clusters than for smeared data.

A hierarchical clustering method works by grouping data objects into a hierarchy or "tree" of clusters. Representing data objects in the form of a hierarchy is useful for data summarization and visualization [3]. For instance, in the case of handwritten character recognition, a set of handwriting samples may be first partitioned into general groups where each group corresponds to a unique character. Some groups can be further partitioned into subgroups since a character may be written in multiple substantially different ways. If necessary, the hierarchical partitioning can be continued recursively until a desired granularity is reached.

There are a number of hierarchical clustering algorithms, such agglomerative or divisive hierarchical clustering, Balanced Iterative Reducing and Clustering using Hierarchies (BIRCH), and Chameleon: Multiphase Hierarchical Clustering Using Dynamic Modeling. An agglomerative hierarchical clustering method uses a bottom-up (merging) strategy, while the divisive hierarchical clustering method uses top down (splitting) approach.

An agglomerative hierarchical clustering method uses a bottom-up strategy. It starts by letting each object form its own sub-cluster and iteratively merges sub-clusters into larger and larger sub-clusters, until all the objects are in a single cluster or certain termination conditions are satisfied (Figure 8 (left)). The single cluster becomes the hierarchy's root. For the merging step, it finds the two clusters that are closest to each other (according to some similarity or distance measure), and combines the two to form one cluster. Because two clusters are merged per iteration, where each cluster contains at least one object, an agglomerative method requires at most *n* iterations. As in each step involve computation for finding the minimum distance between the newly added object and the existing clusters/objects, the complexity of this algorithm is *O*(n^3). The agglomerative hierarchical clustering algorithm is therefore suitable for small dataset.

A tree structure called a dendrogram is commonly used to represent the process of hierarchical clustering. In an agglomerative method, it shows how objects are grouped together step-by-step.

A dendrogram can then be "cut" to form clusters. For instance, on Figure 8 (right) it is shown that the cutting with cut-off distance, *co*, has formed 3 clusters. In general, the larger value of cut-off distance leads to fewer clusters that are formed.

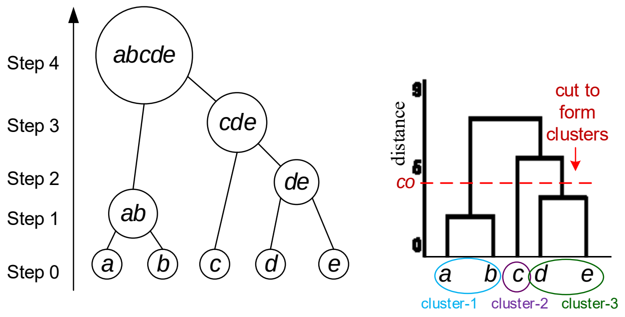

Figure 8. Steps Of The Agglomerative (Left), The Dendrogram Tree With Its Clusters Formed By Cut-Off Distance, Co (Right).

The discussion of the three distance measures, which are single, mean and maximum distance, that can be used in forming the dendrogram can be found in [5].

BIRCH is designed for clustering a large amount of numeric data by integrating hierarchical clustering (at the initial micro clustering stage) and other clustering methods such as iterative partitioning (at the later macro clustering stage) [11]. It is more scalable compared to agglomerative clustering methods.

BIRCH uses the notions of clustering feature to summarize a cluster, and clustering feature tree (*CF-tree*) to represent a cluster hierarchy. A clustering feature is essentially a summary of the statistics for a cluster. CF is a 3-D vector summarizing information about clusters of objects. It is defined as $CF = (n, LS, SS)$, where *n* is the objects count, *LS* is the linear sum of the *n* points, and *SS* is the square sum of the data points. Using a clustering feature, *CF*, many useful statistics of a cluster, such as the cluster's centroid, x_o, radius, *R*, and diameter, *D* can easily be computed.

An important consideration in BIRCH is to minimize the time required for input/output (I/O). BIRCH applies a multiphase clustering technique: A single scan of the data set yields a basic, good clustering, and one or more additional scans can optionally be used to further improve the quality. The primary phases are:
Phase 1: BIRCH scans the database to build an initial in-memory CF-tree, which can be viewed as a multilevel compression of the data that tries to preserve the data's inherent clustering structure.
Phase 2: BIRCH applies a (selected) clustering algorithm to cluster the leaf nodes of the CF-tree, which removes sparse clusters as outliers and groups dense clusters into larger ones. A CF-tree is a height-balanced tree that stores the clustering features for a hierarchical clustering.

In Chameleon, cluster similarity is assessed based on (1) how well-connected objects are within a cluster and (2) the proximity of clusters. That is, two clusters are merged if their interconnectivity is high and they are close together. The merge process facilitates the discovery of natural and homogeneous clusters and applies to all data types

Journal of Theoretical and Applied Information Technology
15th November 2020. Vol.98. No 21
© 2005 – ongoing JATIT & LLS

ISSN: 1992-8645 www.jatit.org E-ISSN: 1817-3195

as long as a similarity function can be specified.

Chameleon uses a *k*-nearest-neighbor graph approach to construct a sparse graph, where each vertex of the graph represents a data object, and there exists an edge between two vertices (objects) if one object is among the *k*-most similar objects to the other. The edges are weighted to reflect the similarity between objects. Chameleon uses a graph partitioning algorithm to partition the *k*-nearest-neighbor graph into a large number of relatively small subclusters such that it minimizes the edge cut. That is, a cluster C is partitioned into subclusters C_i and C_j so as to minimize the weight of the edges that would be cut should C be bisected into C_i and C_j. It assesses the absolute interconnectivity between clusters C_i and C_j.

Chameleon then uses an agglomerative hierarchical clustering algorithm that iteratively merges subclusters based on their similarity. Chameleon has been shown to have greater power at discovering arbitrarily shaped clusters [3, 12].

In selecting the hierarchical clustering algorithm to be adopted for summarizing big data in Spark environment, the following are our considerations:
(1) BIRCH and Chameleon involve two methods and two phases in constructing the final cluster tree. First, some computation is used to prepare the CF or sub-clusters. Second, they employ other algorithms to construct the final hierarchical trees. Therefore, the computation is complex.
(2) Agglomerative is simple, the dendrogram trees can be constructed in one pass from objects having numerical attributes. The degree of summarization can be materialized by adjusting the distance used to cut the tree. Although it is suitable for small dataset, using HDFS and RDD partitions, the big data can be split into small datasets, where each one can be handled by this algorithm. Furthermore, the objects in the partitions can be randomized to avoid biased of the final result.

In this research, big data containing objects with numerical attributes is summarized or reduced into cluster patterns. BIRCH uses *CF* containing a number of statistical measures to represent a cluster. In [10], it is also discussed that cluster patterns can be formed by computing statistical measures from the cluster members. The measures include the number of objects in each cluster, the average (means), minimum, maximum, standard deviation of each attribute values and percentage of objects having each of the attribute values. Thus, in the proposed technique, the patterns include statistical measures of each cluster.

2.4. Evaluation of MapReduce Based Big Data Reduction Technique

In our past research, we have developed big data reduction technique based on Hadoops MapReduce framework, namely BDRT-ParAgglo [5]. It reduces big data based on agglomerative clustering algorithm. The main stages of the technique are as follows (Figure 9).
(a) The big data (*TO*) is randomly divided the objects into *n* disjoint bags of dataset (TO_i), then dendrogram trees are built from each bag (in every slave node of Hadoop cluster) using agglomerative hierarchical clustering method. The maximum number of objects in a dendrogram is defined as a parameter (*maxObject*). Thus, if a bag contains more than *maxObject,* the algorithm will create more than one dendrogram. To construct dendrograms, there are three choices of distance that can be selected, which are single (minimum), complete and means (average).
(b) Once a complete dendrogram is constructed, it is “cut” using certain distance value (defined by users) to form clusters.
(c) Cluster patterns (***CP***) are computed from each cluster formed, collected then written to HDFS.

By selecting the agglomerative method and MapReduce, the advantages are:
(a) Reducing the volume of “dense objects” with their cluster patterns having a lot smaller size.
(b) Allowing flexibility in defining cut off distance to adjust the granularity (fine grain) of the the reduced dataset.
(c) Permitting flexibility in using single, complete or mean distance in constructing dendrogram such that users can select the most suitable one for a certain dataset.
(d) Outliers are preserved (as patterns) as they may be needed by the techniques analyzing the reduced data.
(e) Numbers of bags (*n*) can be adjusted for optimal computation in the distributed network (by considering the slave/data nodes number and their specification, i.e. available memory).

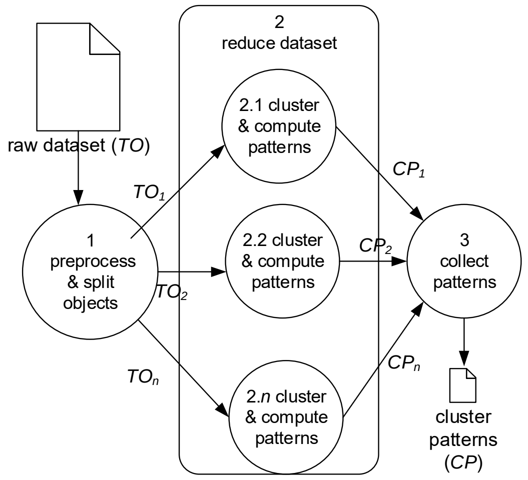
2
reduce dataset
raw dataset (TO)
2.1 cluster & compute patterns
2.2 cluster & compute patterns
2.n cluster & compute patterns
1 preprocess & split objects
3 collect patterns
TO1
TO2
TOn
CP1
CP2
CPn
cluster patterns (CP)

Figure 9. Hadoop Mapreduce BDRT-Paragglo [5].

As discussed in [10] and [13], patterns generated from clusters may include the number of objects in each cluster, the average (means), minimum, maximum, standard deviation of each attribute values and percentage of objects having each of the attribute values. The pattern components computed by BDRT-ParAgglo are count of objects in each cluster, the average (means), minimum and maximum of each attribute values. The standard deviation of each attribute values is excluded as the computation is complex, which leads to significantly reduce its performance.

The technique is implemented as Hadoop Map and Reduce functions as described as follows:
(1) Each Mapper read local blocks of HDFS containing consecutive of records/objects (O_i) then distributes every records randomly in the form of <key, value> to every Reducer, where key is a random number (1… *nSplit*) and value is a record.
(2) Every Reducer receives <key, list of values> from (possibly) all of the Mapper for a certain value of key, then perform data reduction computations and write the results (cluster patterns, CP_i) to HDFS files.
The detailed steps of Mapper and Reducer functions can be found in [5].

The conclusions of the experiments that was conducted to evaluate BDRT-ParAgglo are as follows:
(1) The influence of BDRT-ParAgglo variables: (a) The counts of patterns created are affected by the value of *maxObj* (larger *maxObj* – fewer patterns), *nPar* (larger *nPar* –more patterns), *distType* (descending order: means-single/complete) and *co* (smaller *co* – more patterns); (b) On a specific configured Hadoop cluster, the larger value of *maxObj* or *nPar,* the slower the execution; (c) The size of big data that can be processed on a specific configured Hadoop cluster is influenced by the selection of *maxObj* and *distType* values.
(2) The influence of Hadoop cluster configuration: (a) BDRT-ParAgglo execution times are affected by number of slave nodes and HDFS block size; (b) Adding slave nodes will increase the size of the data that can be processed.
(3) The main objective of partitioning the input dataset (using *nPar* value in Mapper) is to randomize the objects fed into Reducer. Hence, the larger the *nPar* value, the more randomized the data partitions received by Reducer that will produce patterns, which lead to patterns that are less biased towards the objects sequence**.** Thus, it is important to use large value of *nSplit*, such as 1000. It turns out that Hadoop MapReduce BDRT-ParAgglo handles larger splitting poorly. The execution time increases significantly along with increasing the value of *nSplit*. In other words, Hadoop MapReduce BDRT-ParAgglo is not scalable.

3. PROPOSED TECHNIQUE

Our intention is to enhance and redesign Hadoop MapReduce BDRT-ParAgglo towards a technique for Spark that is scalable, namely **Spark BDRT**. We preserve the input parameters, add standard deviation for the cluster pattern component, and design high level concept, data structure and algorithms for Spark.

3.1. High Level Concept of Spark BDRT

In general, to process big datasets, the technique must address:
(1) Scalability issue: The technique should be designed to handle the growing of the data input that will be processed. It must be able to process large size of big data as needed by the users.
(2) Efficiency issue: In the distributed system, there are three main things contributing to this, which are I/O cost, time complexity, and network cost. Reading and writing data from/to disk is slow, thus, the I/O cost should be minimized. The best I/O cost can be achieved by reading the big data once only. In the distributed system, tasks run parallelly in the cluster machines should be design with time complexity lower than *$O(n^3)$*. To minimize the

network cost, shuffling data across the network cluster should be kept minimum.

In term of designing a technique for distributed system, we define that a scalable application should be able to process larger volume of dataset, if needed, by just adding more resources, such as machines into the cluster. The algorithms themselves do not have to be enhanced (for handling larger dataset).

Based on what have been discussed in Section 2.2, the following are our main ideas for achieving both criteria in our proposed technique specifically for the Spark system.

The following methods are employed to achieve scalability:
(1) The use of distributed storage and tasks that run parallel, each task run on a worker node is designed to access local data (stored in the machine disk). Thus, each RDD (storing big data input) partition is created by reading local block of HDFS.
(2) The use of distributed memory to process big data. Large size of RDD objects are not collected into the Driver machine as this will limit the size of RDD objects (up to at most the size of allocated memory in that Driver machine). The design:
(a) Each dendrogram tree is built by reading RDD partitions in the worker, and the trees themselves are stored as local RDD objects in the worker.
(b) The patterns computation ("cutting" the trees at certain distance and then compute the pattern components from the clusters formed by the cut) are performed in the worker nodes by accessing the local RDD partition of dendrograms. Then, the patterns results are directly written into HDFS without being collected first.

The following methods are employed to achieve an efficient technique:
(1) To minimize I/O cost, the input big data, which is stored as blocks of HDFS, is read once.
(2) To reduce computation complexity, memory management that employs deserialized RDD objects is adopted. Deserialized RDD objects that are accessed repeatedly are persisted in the worker node memory.
(3) For limiting the network cost, the application stages are designed to use one wide transformation (involving shuffling process) only, which is described as follows:
(a) The wide transformation is employed for splitting the big data records randomly into bags of dataset, each bag is then stored as one or more RDD partitions.
(b) To reduce traffic cost during the shuffling process, a variable input that is used to define the number of bags is provided. The variable can be set with a number that is sufficiently random the objects (such as less than 3000). Each bag is stored in the worker node memory (as RDD partitions). Then, a dendrogram tree is created locally from this RDD partition, by processing up to *maxObject* objects for a single dendrogram. This process is performed without involving (or exchanging data) with other RDD partitions stored in other worker nodes.
(4) To avoid exponential time respond, the algorithm for tasks that are run parallelly is design with time complexity less than $O(n^3)$.

The detailed design that materialize the previously discussed principles is discussed as follows.

3.2. Job Stages

We design a Spark application with one job. The results needed are cluster patterns computed from big data stored as HDFS blocks stored across slave nodes. There is no intermediate result other than RDDs (computed from transformation operations) that need to be computed between jobs. So, there no need to chain jobs.

As discussed in Section 2.2, a job may have more than one stage, and each stage consists of tasks run parallelly. The stage design with its the description of each task in each stage is depicted in Figure 10. There is only one wide transformation (Stage-2). The RDD groupByKey transformation operation is employed in Stage-2. More discussion of each stage is as follows.

Stage-1

Tasks in this stage read the big data from HDFS blocks line by line. Each line is then transformed into (key, value) pair. The key is generated randomly, having a value between 1 to the number of data bags/splits, namely *nSplits*. The value is an object of Node (see Figure 11), where the line or record of input data is stored in *data* attribute. All of the (key, value) pairs are stored in an RDD, namely *mapData*. One worker node may have more than one of *mapData* partition.

The random generator that generate key will create count of each value almost equally that will result in almost equal size of bags containing (key, value) pairs.

The tasks performed parallelly, each task in a worker node read block of HDFS in the machine node.

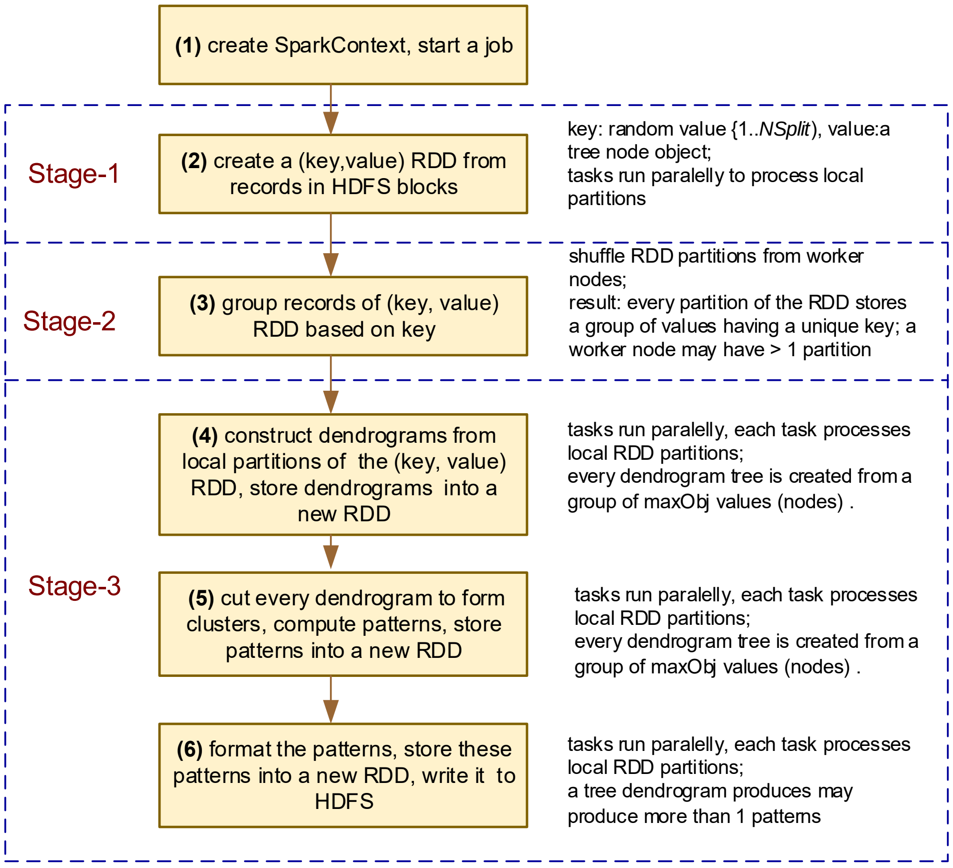
(1) create SparkContext, start a job
Stage-1
(2) create a (key,value) RDD from records in HDFS blocks
key: random value {1..NSplit), value:a tree node object; tasks run paralelly to process local partitions
Stage-2
(3) group records of (key, value) RDD based on key
shuffle RDD partitions from worker nodes; result: every partition of the RDD stores a group of values having a unique key; a worker node may have > 1 partition
Stage-3
(4) construct dendrograms from local partitions of the (key, value) RDD, store dendrograms into a new RDD
tasks run paralelly, each task processes local RDD partitions; every dendrogram tree is created from a group of maxObj values (nodes) .
(5) cut every dendrogram to form clusters, compute patterns, store patterns into a new RDD
tasks run paralelly, each task processes local RDD partitions; every dendrogram tree is created from a group of maxObj values (nodes) .
(6) format the patterns, store these patterns into a new RDD, write it to HDFS
tasks run paralelly, each task processes local RDD partitions; a tree dendrogram produces may produce more than 1 patterns

Figure 10. The High-Level Design Of The Application.

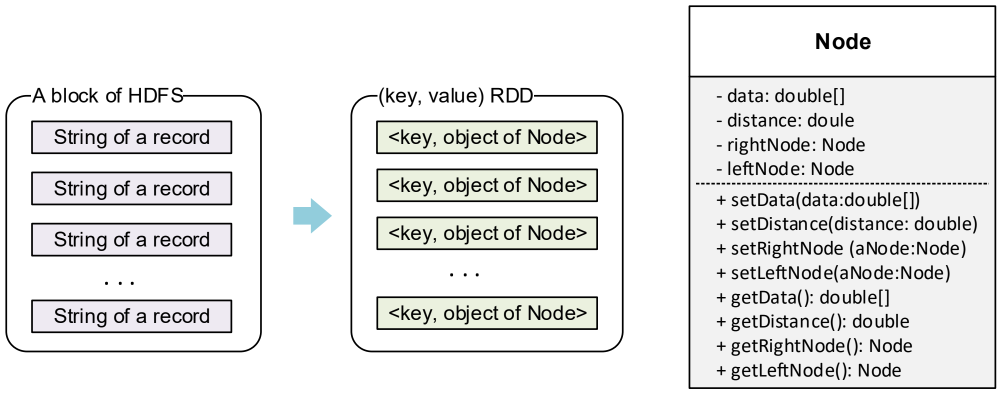

Figure 11. Records Are Read From HDFS Blocks And Stored As (Key, Value) RDD.

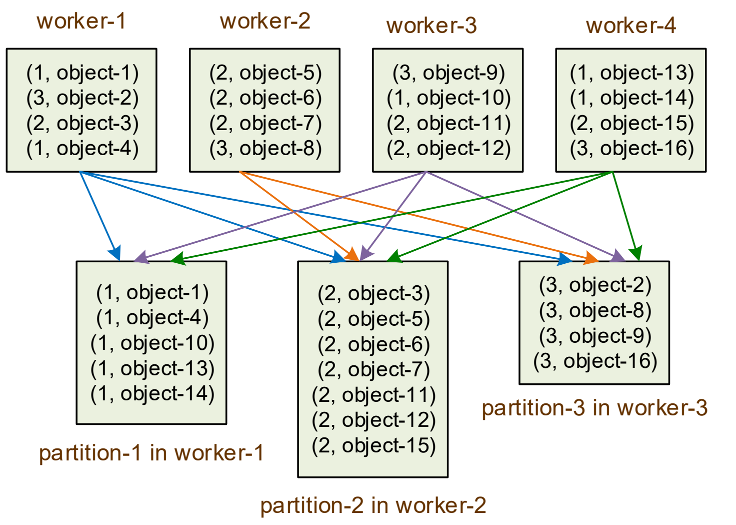

Figure 12. Example Of Grouping (Key, Value) Records That Involve Shuffling.

Stage-2

This stage is materialized by calling RDD *groupByKey* transformation operation, which causes pair of (key, value) shuffling across worker nodes (see Figure 12).

The value of *nSplits,* defined in Stage-1, will affect the network cost. The larger the *nSplits*, the more random the record in each data split (bag), but with the more the network cost that could slow down the splitting process. Splitting the input data with a larger *nSplits* will give results of smaller size of RDD partitions in the worker nodes. A bag (key, value) pairs with the same key value may be stored in one or more RDD partitions, depending the size of pairs.

To reduce the network cost, *nSplits* can be set relatively small (for instance, 50 to 100), consequently, each bag of (key, value) pairs will be large. To avoid getting to general patterns, then the maximum number of objects in a dendrogram can bet set small (for example, 30 to 100).

Stage-3

The tasks for constructing dendrogram trees performed parallelly. Each task in a worker node processes an RDD partition in this node. A tree is constructed from no more than the defined maximum of objects in a tree, *maxObj* (see Figure 13, for an example). Each task stores the dendrogram trees into one or more new RDD partitions in the local memory. After a dendrogram tree is created, the task function will cut the tree by certain distance (*co*) to form clusters, and then compute patterns from each cluster. The patterns are stored a new RDD, namely *patterns*, hence each task store those computed patterns in a local *patterns* partition.

The value of *maxObj* along with the cut-off distance will determine the percentage of the reduction. The larger value of *maxObj* and cut-off distance will result in larger percentage of reduction, as fewer patterns will be created.

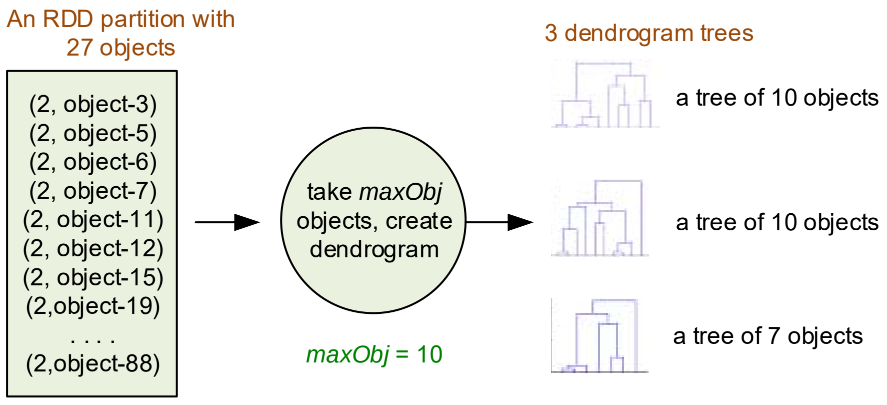

Figure 13. Constructing Dendrogram Trees From An RDD Partition Representing Bag Of (2, Value) Pairs.

3.2. Algorithms Design

In this subsection, the overall algorithm of tasks in executed in Stage-3 (namely, *reduce*) is firstly discussed discussed. Then, the main algorithms for constructing dendrogram trees and computing patterns from the trees are discussed.

Algorithm-1: **reduce**

Input: list of <key, Node objects> with unique value of key, *dataBag*; maximum objects in a dendrogram, *maxObj*; distance type for constructing dendrograms, *distType* ε {*single*,*complete*,*centroid*}, and cut-offdistance, *co*. The variable of *maxObj, distType, and co* are broadcasted by Driver.

Description**:** Build dendrograms from RDD partitions, where each dendrogram contains no more that maxObj nodes. Create clusters from dendrogram using *distType*. Cut dendrogram using *co* and compute cluster patterns, then write these patterns to HDFS.

Steps:

```
objectList ← Node[] // initialize array of Node (RDD)
patterns ← Pattern[] // initialize for collecting cluster
patterns (RDD)
//loop-1:
foreach element in dataBag.values do
    objectList ← objectList join element
    isProcessed ← false
    if count of objectList == maxObj then
        dendrogram ← generate_dendrogram(objectList,
         distType)
          pt ← compute_patterns(dendrogram, co)
          patterns ← patterns join pt
          isProcessed ← true
          delete element of objectList
      end
      if isProcessed == false  then  // objectList
       contains < maxObj elements
          dendrogram ← generate_dendrogram
           (objectList, distType)
          pt ← compute_patterns(dendrogram, co)
           patterns ← pattern join pt
       end
   end

   //loop-2: write patterns to distributed storage
   foreach pattern in patterns do
          write pattern to HDFS
    end
```

Algorithm of constructing dendrogram:
Algorithm-2**: generate_dendrogram**

Input**:** list of objects, *objectList*; type of distance for constructing dendrogram, *distType*
Output**:** *dendrogram*, distance between clusters, *distanceMatrix*

Description**:** Construct a *dendrogram* from *objectList* using *distType*
Steps:

```
distanceMatrix ← [][] // array of double representing
distance between clusters (RDD)
nodeListCluster ← [] //array of List<Node> representing
clusters (RDD)
dendrogram ← [] // array of Node objects representing
dendrogram tree (RDD)
i ← 0
    //loop-1:
    foreach node in objectList do   //initialize
       nodeListCluster and dendrogram
   nodeListCluster[i].add(node)
   dendrogram ← dendrogram join node
   distanceMatrix ← distanceMatrix add
ArrayBuffer[Double]
   i ← i + 1
 end
//loop-2:
i ← 1; j ← 0
for i < distanceMatrix.length do // fill distaceMatrix
  using the distance of distType
   for j < i do
      distanceMatrix[i][j] ←
findMinDist(nodeListCluster[i], nodeListCluster[j],
distType)
       j ← j + 1
    end
    i ← i + 1
end
//loop-3:
while dendrogram.length  != 1  do // construct a
dendrogram tree
    x ← 1; y ← 0; temp ← 0; coordX ← 0;
    coordY ← 0
    result ← Double.MaxValue
    for x < distanceMatrix.length do
        for y <= x do
            temp ← distanceMatrix[x][y]
            if temp < result then result ← temp;
               coordX ← x; coordY ← y
            j ← j + 1
        end
        i ← i + 1
    end
dendrogram ←
  formClustersBetweenNearestNeighbours(coordX,
   coordY) //update dendrogram
distanceMatrix ← recalculateMatrix(coordX,  coordY)
//update distanceMatrix
end
```

Algorithm-3**: compute_patterns**

Input: a *dendrogram*; cut-off distance (having value between 0 to 1), *co*
Output: patterns computed from all clusters generated from dendrogram

Description: Cutting a dendrogram tree to form clusters, compute patterns from every formed cluster. The pattern components in each cluster are the average, minimum, maximum, standard deviation of every attribute and count of Node objects.
Steps:

```
clusters ← [] // initialize array of Node objects to store clusters (RDD)
bfs ← dendrogram
dist ← co x dendrogram.distance
//loop-1: visit each node top-down and from left to right (BFS)
while bfs not null do
    node ← bfs.remove(0)  //move the first node of bfs to node
    if node.distance <= dist then clusters.add(node)
    else
        left ← node.left
        right ← node.right
        if left != null then bfs.add(left)
        if right != null then bfs.add(right)
end
patterns[] ← [] // initialize array of Pattern objects
//loop-2:
foreach cluster in clusters do
    p ← computeAPattern(cluster) // compute average, minimum, maximum, deviation of every attribute, count of Node objects from cluster
    patterns.add(p)
end
return patterns
```

The *reduce* algorithm, which is translated into tasks run parallelly in every executor, the time complexity can be derived from Algorithm-1, -2 and -3 as follows:
(1) Time complexity of Algorithm-1: in loop-1: $O(m)$, where $m <= maxObj$; in loop-2: $O(n)$, where n = number of patterns, but usually $n << m$, so the complexity can be simplified to $O(m)$.
(2) Time complexity of Algorithm-2: in loop-1: $O(s)$ where $s <= maxObj$, in loop-2: $O(pq)$, $p <= maxObj$ and $q <= maxObj-1$, in loop-3: $O(abc)$, $a <= maxObj$, $b <= maxObj$, $c <= maxObj-1$, so the highest complexity is $O(n^3)$.
(3) Time complexity of Algorithm-3: in loop-1: $O(x)$ where $x <= maxObj$, the larger the value of co, the smaller the value of x; in loop-2: $O(y)$, y = number of clusters in a dendrogram, so time complexity is $O(n)$.

Based on the above derivation, the overall complexity of *reduce* algorithm is approximately $O(n^3)$. The part of the algorithm with the highest complexity is constructing dendrograms. Since *reduce* is executed parallelly across executors, the approximate time complexity is $O(n^3)/nExec$, where *nExec* is the number of executors.

4. EXPERIMENTS

4.1. Experiment Environment

The experiments were performed on a Hadoop cluster running YARN consisting one master and 10 slave nodes. The specification of each machine: The processor is Intel core i5 8500 running at @3.00 GHz with 6 cores, the master memory is 24 Gb and slave memory is 8 Gb, the disk space is 500 Gb, the operating system is Ubuntu 18.0.4. The HDFS block size is set to 32 Mb.

During the experiments, we used 6 Gb of memory in every worker node, and 1 core to 5 cores on each machine (one core is left for the machine operation). As there are 10 machines, we could use up to 50 cores, each core runs an executor (Figure 14). The function in Stage-1 and *reduce* algorithm in Stage-3 were run as parallel tasks on 10 to 50 cores.

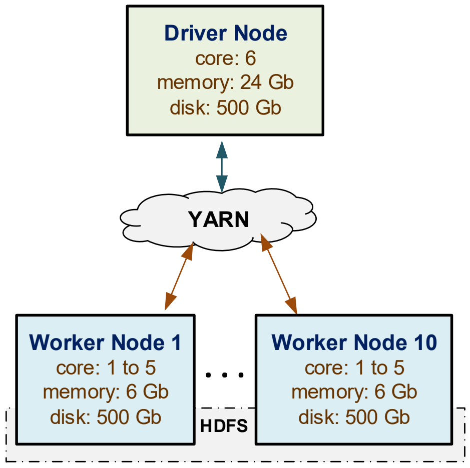

Figure 14. The Cluster Running Hadoop YARN And Spark With 10 Machine Workers, Which Can Run 10 To 50 Spark Executors.

4.2. Experiments Goals and Methods

We conducted a series of experiment with the intention to evaluate the added capability (standard deviation as a pattern component computation), scalability and efficiency of the proposed technique. The experiments were grouped as follows:
(1) Comparing the Spark BDRT versus Hadoop MapReduce BDRT-ParAgglo.
(2) Evaluating whether Spark BDRT can handle standard-deviation computation and did not significantly degrade its performance.

(3) Evaluating the data reduction percentage based on the number of objects (*maxObj*) in a dendrogram.
(4) Evaluating the effect of number of bags that cause shuffling towards Spark BDRT performance.
(5) Evaluating the effect of maximum number of objects towards Spark BDRT performance.
(6) Evaluating the effect of adding machine cores (executors) towards Spark BDRT performance.
The performance measures were the execution time (efficiency) and scalability (able to process large data).

With the above goals, we ran Hadoop MapReduce BDRT-ParAgglo and specifically Spark BDRT many times using a variety of *nSplit* and *maxObj* input parameters (*co* is set to 0.8), as well as number of cores used in every worker machine. Each ran using the data input depicted on Table 1, then the execution time was recorded.

4.3. Big Data Used

In these experiments, we use the same big data used in [5]. It is the dataset of household energy consumption, which is obtained from https://archive.ics.uci.edu/ml/datasets/ with the size of approximately 132 Mb. This archive contains 2075259 measurements (records/objects) gathered between December 2006 and November 2010. The sample of the dataset are as follows:
9/6/2007; 17:31:00 ; 0.486 ; 0.066; 241.810; 2.000; 0.000 ; 0.000 ; 0.000
9/6/2007; 17:32:00 ; 0.484 ; 0.066; 241.220; 2.000 ; 0.000 ; 0.000; 0.000
9/6/2007; 17:33:00 ; 0.484 ; 0.066 ; 241.510; 2.000; 0.000 ; 0.000 ; 0.000
Each line presents a record with 9 attributes, the excerpts are: (1) Date; (2) Time; (3, 4, 5, 6) some results of metrics; (7) sub_metering_1: energy sub-metering (watt-hour) that corresponds to the kitchen, (8) sub_metering_2: energy sub-metering that corresponds to the laundry room; (9) sub_metering_3: energy sub-metering that corresponds to a water-heater and an air-conditioner.

The data preprocessing performed in Map function is as follows:
(a) Number of day (1, 2, …7) is extracted from Date and stored as attribute-1.
(b) Hour (1, 2,…24) is extracted from Time and stored as attribute-2.
(c) The value of sub_metering_1, _2 and _3 are taken as is and stored as attribute-3, -4, -5.

Thus, the preprocessed dataset has 5 attributes, which are day number, hour and 3 sub-metering measures.

Then we produced the synthetic big data of 5, 10, 15 and 20 Gb by multiplying the preprocessed data. All of these datasets are stored in the distributed HDFS in the Hadoop cluster that are partitioned into blocks (see Table 1).

Table 1. The Number Of Objects And HDFS Blocks Of Big Data Input.

Size (Gb)	#Objects	#HDFS blocks
5	144,000,000	157
10	256,000,000	279
15	400,000,000	435
20	529,000,000	576

4.4. Experiment Results dan Discussion

(1) Comparing the performance of Hadoop MapReduce BDRT-ParAgglo versus Spark BDRT

This was performed using 10 machines as master and slave nodes, 10 cores/executors and 30 objects/tree (*maxObj* = 30).

The results are depicted on Figure 15 with the discussion as follows:
Hadoop MapReduce BDRT-ParAgglo: It does not handle splitting the big data using large value of *nSplit* or splitting the big data into larger number of bags. Time response increased sharply (exponentially) with the number of splits for the same big data size. Thus, it is not scalable, since larger size of big data should be split into more bags. Our analysis: During the shuffling process, mapper write the output to (local) disk, then each reducer read and scan the (key, object) and take the pair having the same key from all of the mappers run in slave nodes. Thus, this shuffling process cause high cost of IO and network.
Spark BDRT: It handles big data splitting using large value of *nSplit* well. By splitting the data into larger bags, the time respond decreased. The shuffling process for splitting the big data into 3000 bags can be handled well.
These experiments show that by employing one shuffling, BDRT-AggloSpark is scalable.

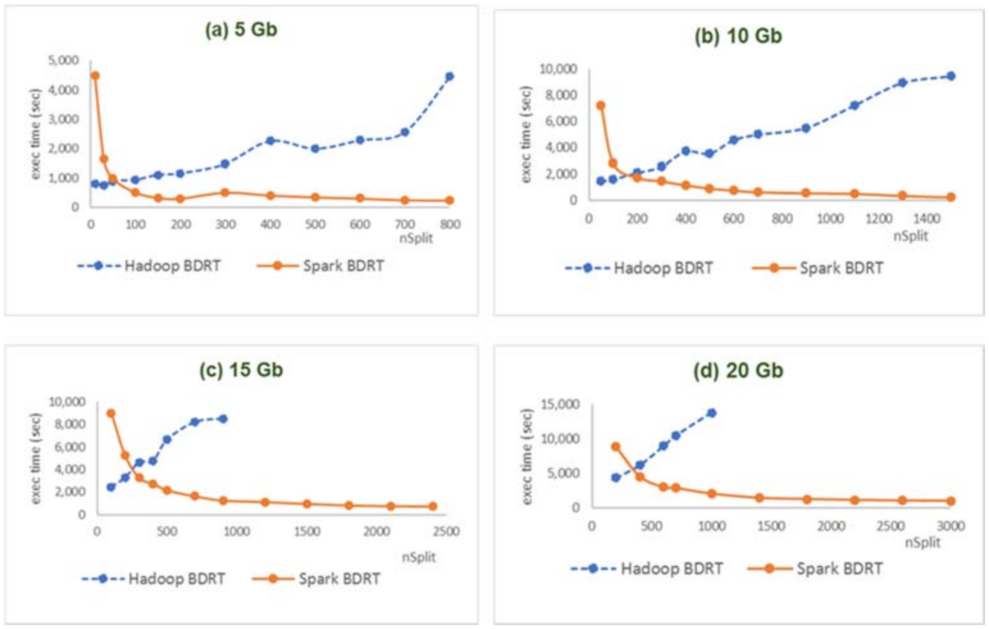

Figure 15. Execution Time Of Hadoop Vs Spark BDRT Using 10 Slaves, 10 Cores And 30 Objects/Tree, With The Input Data Size Of: (A) 5 Gb, (B) 10 Gb, (C) 15 Gb And (D) 20 Gb.

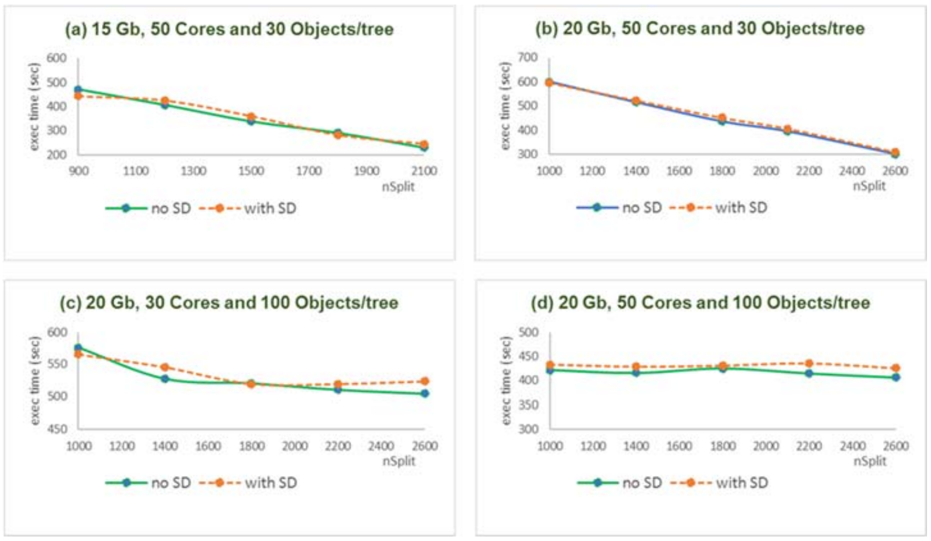

Figure 16. Execution Time Of Spark BDRT On 30-50 Cores/Executors That Include Standard Deviation (With SD) Vs Without SD (No SD) For 15-20 Gb Data Size, Configured With 30-100 Objects/Dendrogram Tree.

(2) Evaluating the performance of Spark BDRT in computing standard-deviation (included in pattern)

As discussed in Section 3.2 (Algorithm-3), Spark BDRT adds one component in the cluster patterns, which is standard deviation of each attribute value among objects in every cluster. Standard deviation computation involves two iteration: First, compute the average attribute values. Second, using that value to subtract every attribute value for computing the standard deviation. In these experiments, Spark BDRT processed 15 to 20 Gb of data and 3-5 cores in 10 worker machines are used.

As shown on Figure 16, the data reduction that involves standard deviation computation only increase the execution time slightly. The tasks executed parallelly on 30 and 50 cores do not degrade the efficiency significantly. This can be achieved because, in Algorithm-3, the computation is performed by accessing RDD (of clusters) partitions stored in local memory (in the worker machine) without involving additional network and I/O cost.

(3) Evaluating data reduction percentage

In these experiments, Spark BDRT processed 10 to 20 Gb of data and ran using 30 cores in 10 worker machines. The cut-off distance (*co*) is set to 0.8 in all executions.

As depicted on Figure 17, the reduced percentage decreases as number of objects (*maxObj*) in a dendrogram tree increases. The percentage is computed by dividing the size of patterns by the size of data input.

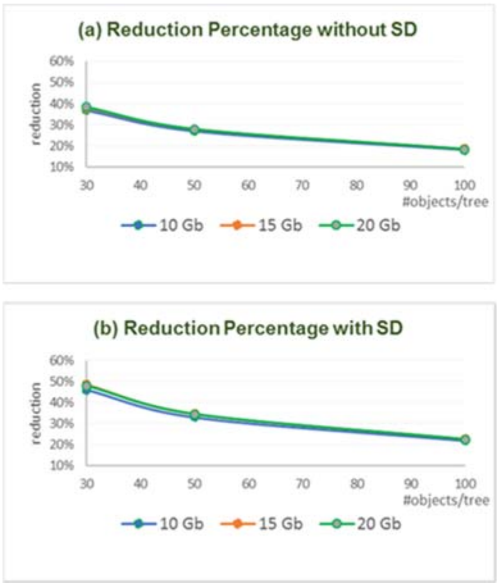

Figure 17. Data Reduction Percentage: (A) Without Including Standard Deviation (SD); (B) Including Standard Deviation As A Component In The Cluster Patterns.

The decreasing of the reduction (as the results of more objects/tree) will vary from one to another big data as it depends on the dendrogram trees constructed from the big data. This experiment proves that by configuring Spark BDRT with more objects in each dendrogram tree, the reduction percentage will be smaller. This is consistent with the concept discussed in Section 2.3.

(4) Evaluating the effect of *nSplit* towards the execution time

It is expected that the larger the value of *nSplit*, the more cost of shuffling process (among RDD partitions across worker machines). We performed two groups of experiments: The first group was processing 3 Gb to 20 Gb of data input with *nSplit* = 100..600 executed on 10 cores; the second groups was processing 20 Gb data with larger values of *nSplit*, which are = 1000, 1500, 2600, and executed on 10, 30 and 50 cores. The results are presented on Figure 18 and 19.

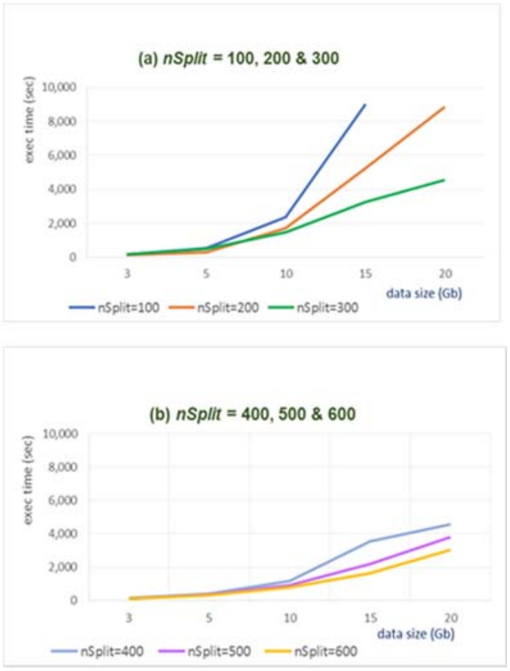

Figure 18. Execution Time Of Spark BDRT Run On 10 Cores, Maxobj = 30 And Nsplit = 100, …, 600.

Figure 18 shows that the larger value of *nSPlit*, the faster process. Analysis: Larger value of *nSPlit* means smaller bags stored as RDD partitions as the input of *reduce* algorithm that runs parallelly as worker tasks. Hence, parallel tasks executed faster when they process smaller input data. The results also show that the shuffling process and network cost is less significant compared to the complexity of the reduce algorithm. The execution time is mostly determined by *reduce* algorithm.

Figure 19 shows that using more cores or executors, up to a point, reduces the execution time. The 20 Gb of data is stored as 576 blocks (see Table 1). This means that each machine approximately stores 5-6 HDFS blocks. When all blocks are processed locally by the local parallel worker tasks, the computation will be faster. When only 10 cores are used, there are only one task (representing *reduce* algorithm) running on each worker machine. This means, the blocks are processed sequentially, one by one, which results in long execution time. Contrarily, when 5 cores on each machine are used, each of the 5 parallel tasks (in a single machine) processes a single block. This significantly reduces the computation time.

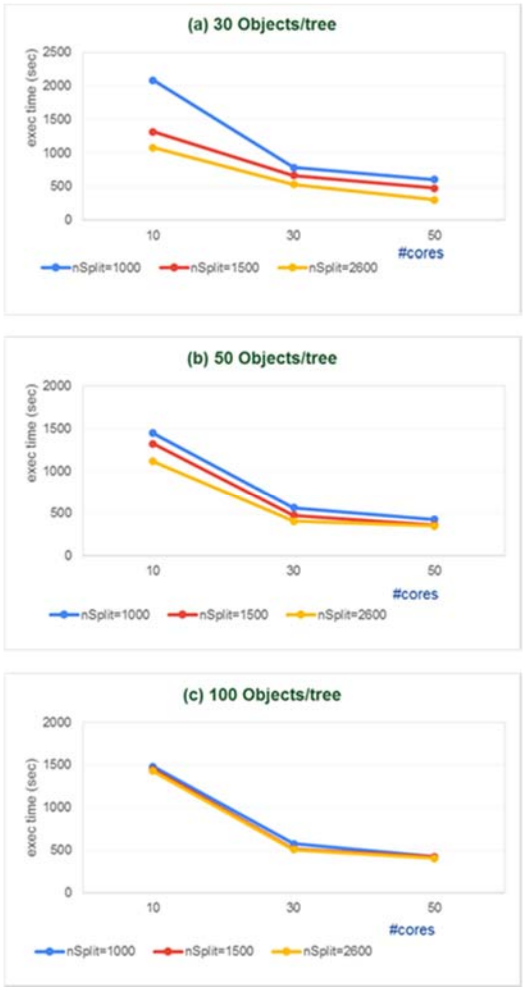

Figure 19. Execution Time Of Spark BDRT With 20 Gb Of Data Size, Run On 10, 30 And 50 Cores.

In line with the previous results (Figure 18), in this experiment, we find that the larger value of *nSPlit* and smaller value of *maxObj*, the faster the computation.

(5) Evaluating the effect of *maxObj, nSplit* and number of cores towards the execution time

To observe more clearly of the effect of *maxObj, nSplit* and number of cores towards the BDRT-AggloSpark execution time, more experiments were conducted. The results are presented as follows.

The experiment results using 10 cores, *maxObj* = 30, 50 and 100, nSplit = 300…2500 and data input of 5, 10, 15 and 20 Gb are presented on Figure 20.

Figure 20 shows that when *maxObj* is relatively small, which is 30 and 50, larger value of *nSplit* reduces the execution time significantly up to a point. However, when *maxObj* is 100, which means a dendrogram tree is constructed from 100 objects (except the "left-over" objects in the bag, which can be less), larger value of *nSplit* only slightly reduces the execution time.

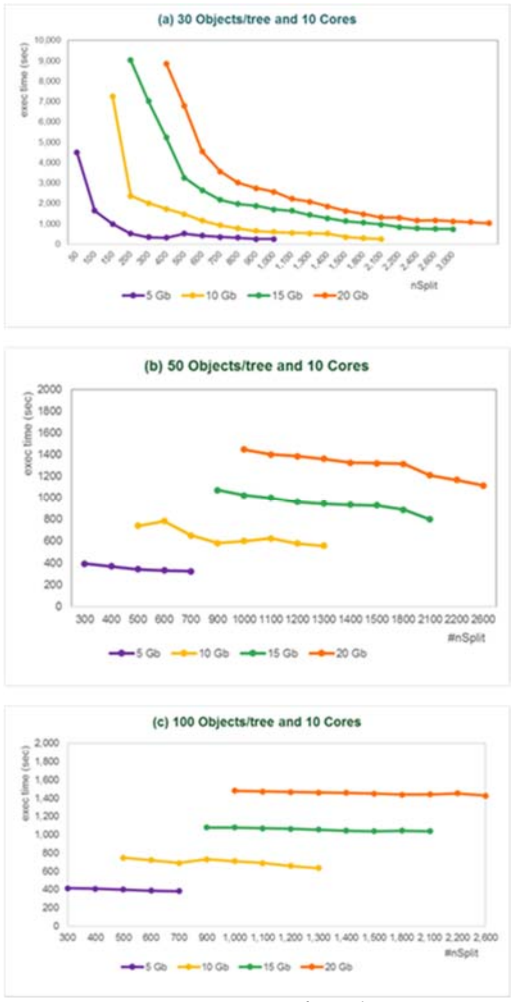

Figure 20. Time Execution Of Spark BDRT Using Maxobj = 30, 50 And 100 On 10 Cores.

Analysis: As discussed previously, the complexity of Algorithm-1, -2, and -3, is approximately $O(n^3)$. Here, the *n* is determined by (mostly) *maxObj*. Increasing the *maxObj* will significantly increase the time for constructing dendrogram trees, computing clusters and patterns. As the complexity for a single task is $O(n^3)$, other costs (I/O, splitting data into bags and shuffling) is less significant in contributing to the overall execution time.

Specifically using *maxObj* = 30, more experiments were performed using 30 and 50 cores. The results, which are presented on Figure 21, show the consistent results. The execution time decreases as *nSplit* increases.

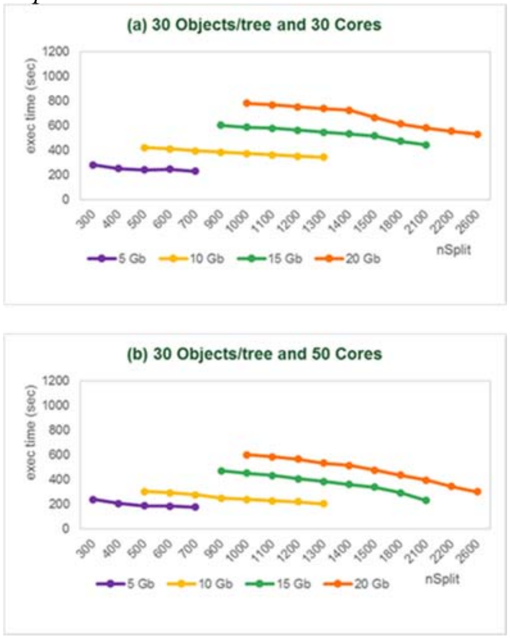

Figure 21. Time Execution Of Spark BDRT Using Maxobj = 30 On 10 Machines And 3-5 Cores.

Analysis: With 30 objects/tree, the time gained from parallelly processing smaller bags (by worker tasks) outweighs the overhead for splitting the dataset (represented as string RDD) into more bags (represented as *Node* RDD in Figure 11). As processing a smaller Node RDD partition will be done faster, a task can further process another RDD partition after it is done with one partition. For parallel tasks, this leads to faster execution in processing the whole bags (stored as RDD partitions in a machine).

Specifically using *maxObj* = 50, more experiments were performed with 10, 30 and 50 cores. The results shown on Figure 22 are analogous with Figure 21, however, the reduction of the execution time (over increasing *nSplit* value) is less.

Analysis: The complexity for a single task is $O(n^3)$ for constructing trees and patterns, where n is 50, becomes more significant compared to other costs (I/O, splitting data into bags and shuffling). Hence, although larger value of *nSplits* lead to having smaller bags (*Node* RDD partitions), the execution time of *reduce* algorithm (as parallel tasks) does not decrease significantly.

More experiments were performed using *maxObj* = 100 and 10, 30 and 50 cores. The results are presented on Figure 23. It can be interpreted that constructing a dendrogram tree from 100 objects will not give benefit in terms of reducing the execution time for larger value of *nSplit* (smaller size of bags or Node RDD partitions).

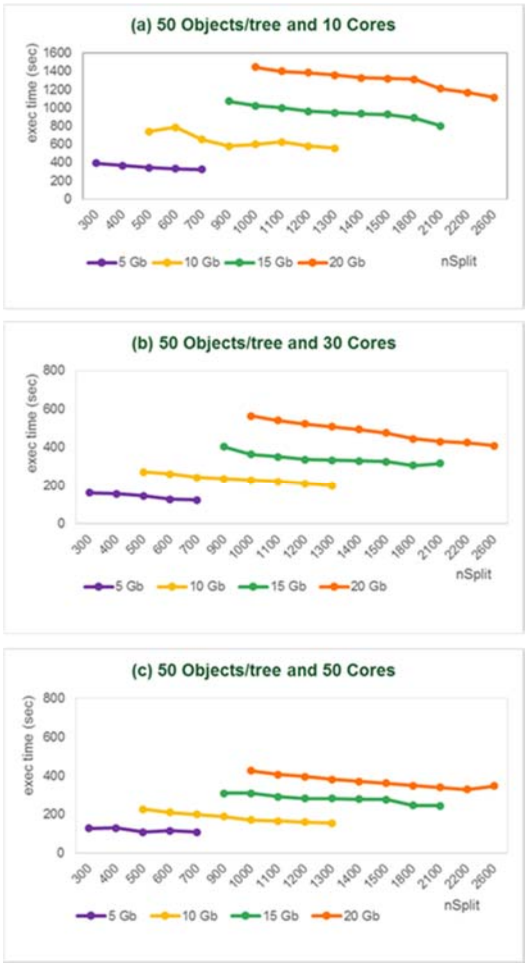

Figure 22. Time Execution Of Spark BDRT Using Maxobj = 50 On 10 Machines And 3-5 Cores.

Analysis: The complexity for a single task that is $O(n^3)$, where n is 100, becomes very significant compared to other costs (I/O, splitting data into bags and shuffling). More time is required to construct dendrogram trees and computing patterns, such that when added with other costs, the benefit of splitting the data into smaller bags is not beneficial.

The comparison of execution times using 30, 50 and 100 objects/dendrogram tree for input of 20 Gb data and a variety of *nSPlit* values are also presented on Figure 24. The figure also presents that when *maxObj* = 30, increasing the *nSplit* value will lead in decreasing the execution time significantly, whether Spark BDRT is executed with 10, 30 or 50 cores. However, when *maxObj* is set to 50 and 100, the decreasing (of the execution time) is less significant.

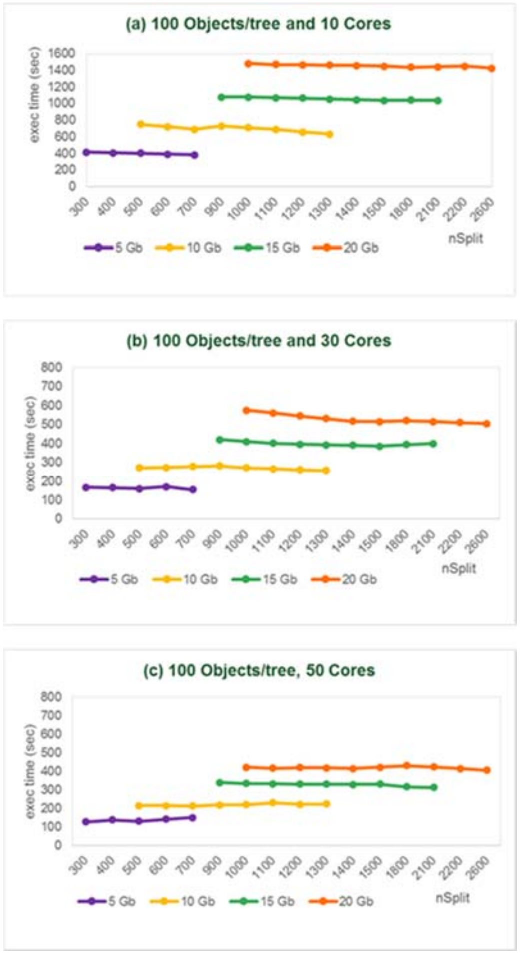

Figure 23. Time execution of Spark BDRT using maxObj = 100 on 10 machines and 3-5 cores.

(6) Conclusion of the Experiments

The over all findings of the experiments can be concluded as follows:
(a) Spark BDRT has resolved the scalability issue of Hadoop BDRT.
(b) The standard-deviation computation performed parallelly and does not significantly degrade Spark BDRT performance.
(c) The data reduction percentage decreases along with setting more objects in a dendrogram tree (so is by giving larger value of cut-off distance as discussed in [5].
(d) The shuffling process (in Stage-2) does not contribute significantly towards Spark BDRT execution time. The parallel tasks with complexity of $O(n^3)$ dominates the execution time.
(e) In line with the above findings, as the n in $O(n^3)$ is the *maxObj*, setting larger value of *maxObj* increases the execution time significantly.
(f) Adding machine cores, which leads to more executors assigned in executing Spark BDRT, will speed up the execution time up to a condition. When adding more cores means that each executor can have more tasks that will process RDD partitions parallelly, Spark BDRT will be executed faster.

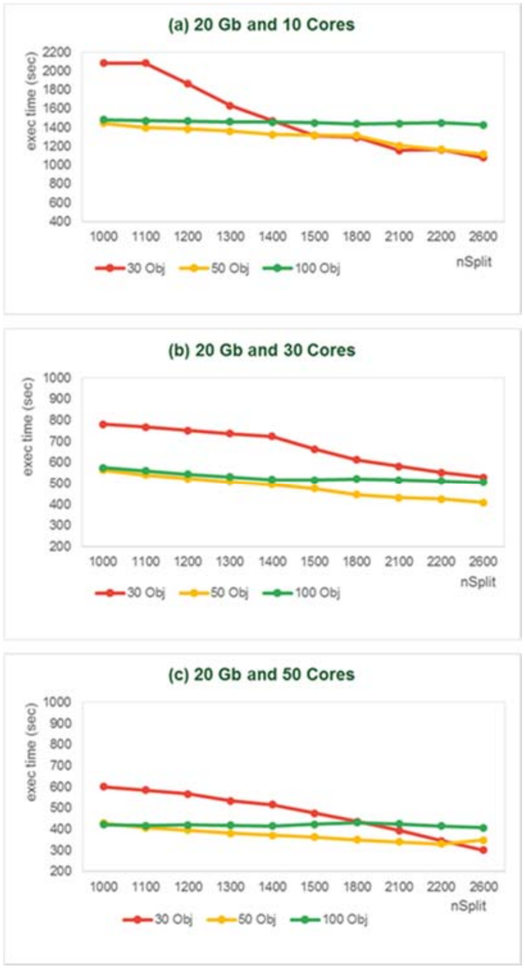

Figure 24. Time execution of Spark BDRT 20 Gb data on 10 machines and 3-5 cores.

5. COMPARISON OF HADOOP AND SPARK BDRT

Based on the literature study and the experiment results discussed in Section 4, Table 2 presents the comparison between Hadoop and Spark BDRT in terms of its data processing speed. The comparison is given based on the computation stages of the BDRT.

Table 2. Comparison between Hadoop and Spark BDRT

Stage and Task	Hadoop BDRT	Spark BDRT
Stage-1: Create (key, object) pairs from blocks of HDFS locally	slower (map task involve partitioning, sorting and writing workers' local disk)	fast (RDD of the pairs are stored as partitions in workers' local memory)
Stage-2: Shuffle the object across worker nodes to avoid biased in constructing clusters	very slow as every reduce task performs reading objects from workers' disk, transferring object to the network, and writing to local disk (where the reduce task runs), and sorting/grouping objects	moderate (every reduce task reads/writes RDD objects from/to memory, objects being grouped are transferred across the network)
Stage-3: Constructing dendrograms	moderate (every tree is constructed from *maxObj* objects with $O(n^3)$ complexity)	moderate (every tree is constructed from *maxObj* objects with $O(n^3)$ complexity)
Stage-3: Cut dendrograms and compute patterns	without deviation standard computation, fast (traversing a small size of tree)	fast (with deviation standard computation)
Stage-3: Format and write patterns to HDFS	fast (every reduce task write patterns to HDFS)	fast (every reduce task write patterns to HDFS)

From Table 2, it can be seen that Spark BDRT addresses the problems in Stage 1 and 2 of Hadoop BDRT, such that it is now scalable.

6. CONCLUSION AND FURTHER WORKS

Spark BDRT has resolved the scalability issue of Hadoop BDRT for reducing big data. From the experiments of reducing data using large value of *nSplit* (up to 2600), which used to randomly split the big data records into *nSplit* bags of dataset (RDD partitions) before computing clusters and patterns, it is shown that Spark BDRT performs well. It has also been proved that by adopting only one shuffling process in the Spark BDRT, the cost of shuffling is less significant compared to the cost of time complexity of the parallel tasks.
Although the complexity of the proposed algorithm in the function run parallelly is $O(n^3)$, where n is the maximum objects in every dendrogram tree, better efficiency of Spark BDRT can be achieved by setting lower number of objects in a dendrogram and/or adding more cores (when executing the application).

The experiment results also show that computing a standard deviation with complex computation can be handled by Spark BDRT. For further works, the pattern components for representing each cluster can be modified, such as used in BIRCH algorithm. To achieve efficient computation, the function for computing patterns should be designed as executor tasks, such that can can be run parallelly.

Spark BDRT can be viewed as a model of big data computation that needs random records splitting than performing some specific computation on the resulted bags of dataset. For further research, the model can be adopted and enhanced to handle big data with non numerical records or objects. In this case, we intend to continue our research for summarizing Indonesian data reviews [14] and big graph [15]. One option of future works is reducing big data that can be fed into parallel classification algorithms, such as parallel Naïve Bayes, Random Forest that have been implemented in MLLib and ML Spark [2, 7], as well as for incremental parallel classifier (our research progress is discussed in [16]) .

ACKNOWLEDGMENT

This research is funded by *Direktorat Riset dan Pengabdian Masyarakat, Direktorat Jenderal Penguatan Riset dan Pengembangan*, through *Penelitian Dasar Unggulan Pergurutan Tinggi* scheme in 2020, contract number: III/LPPM/2020-04/107-PE-S. We would like to thank for the supports.

REFERENCES

[1] X. Su, *Introduction to Big Data*, Institutt for informatikk og e-læring ved NTNU, Norway, 2015.

[2] Holden Karau, Andy Konwinski, Patrick Wendell, and Matei Zaharia, *Learning Spark*, O'Reilly Media, Inc., 2015

[3] J. Han, M. Kamber and J. Pei, *Data Mining Concepts and Techniques 3rd Ed.*, The Morgan Kaufmann Publ., USA, 2012.

[4] M. H. U. Rehman, C. S. Liew, A. Abbas, P. P. Jayaraman, T. Y. Wah, S. U. Khan, "Big data reduction methods: a survey", *Data Science and Engineering*, pp. 1-20, December 2016.

[5] V. S. Moertini, G. W. Suarjana, L. Venica and G. Karya, "Big Data Reduction Technique using Parallel Hierarchical Agglomerative Clustering", *IAENG International Journal of Computer Science*, Vol. 45, No. 1, 2018.

[6] B. Chambers and M. Zaharia, *Spark: The Definitive Guide, Big Data Processing Made Simple*, O'Reilly Media, Inc., USA, 2018.

[7] Holden Karau and Rachel Warren, *High Performance Spark*, O'Reilly Media, Inc., USA, 2017.

[8] A. Holmes, *Hadoop in Practice*, USA: Manning Publications Co., 2012.

[9] T. White, *Hadoop: The Definitive Guide 3rd Ed.*, O'Reilly Media, Inc., 2012.

[10] K. Tsiptsis and A. Chorianopoulos, *Data Mining Techniques in CRM: Inside Customer Segmentation*, John Wiley and Sons, L., UK, 2009.

[11] Zhang, T., Ramakrishnan, R., dan Livny, M. "BIRCH: An Efficient Data Clustering Databases Method for Very Large", *ACM SIGMOD International Conference on Management of Data*, 1, 103–114, 1996.

[12] Xu, D., and Tian, Y. A, "Comprehensive Survey of Clustering Algorithms", *Annals of Data Science*, 2(2), 165–193, 2015.

[13] V. S. Moertini, L. Venica, "Enhancing parallel k-means using map reduce for discovering knowledge from big data", *Proc. of. 2016 IEEE Intl. Conf. on Cloud Computing and Big Data Analysis* (ICCCBDA 2016), Chengdu China, 4-7 July 2016, pp. 81-87.

[14] V. S. Moertini, V. Kevin, J. Satyadi, "Mining Opinions from Big Data of Indonesian Hotel Reviews", *Journal of Theoretical and Applied Information Technology*, Vol.95. No 14, 2017.

[15] Atastina, I., Sitohang, B., Saptawati, G. P., and Moertini, V. S., "A review of Big Graph Mining Research", *IOP Conference Series: Materials Science and Engineering*, IOP Publishing, 180, 012065, 2017. https://doi.org/10.1088/1757-899X/180/1/012065.

[16] Moertini, V.S., Septrianto, M. and Venica, L., "Incremental Parallel Classifier for Big Data with Case Study: Naïve Bayes using Mapreduce Patterns", *Journal of Theoretical and Applied Information Technology*, Vol.97. No 11, pp. 3077- 3097, 2019.