Available online at www.sciencedirect.com

ScienceDirect

Procedia Computer Science 218 (2023) 594–603

Procedia Computer Science

www.elsevier.com/locate/procedia

International Conference on Machine Learning and Data Engineering

Performance Analysis and Deployment of Partitioning Strategies in Apache Spark

Tinku Singh[a], Shivam Gupta[b], Satakshi[c], Manish Kumar[a]

[a]*Indian Institute of Information Technology Allahabad, Prayagraj, India*
[b]*Indian Institute of Engineering Science and Technology, Shibpur, India*
[c]*SHUATS Prayagraj, U.P. India*

Abstract

Data is flourishing day by day to a large extent, the data that need to be analyzed are not just large, but it may be high dimensional, heterogeneous, complex, unstructured, incomplete and noisy as well. Apache Spark framework is used for high-performance computing of Big Data. The proper division of the dataset can impact the degree of parallelism achieved. In Apache Spark, partitioning techniques help to manage the dataset in a distributed fashion. The optimal number of partitions and associated data with these partitions ensures the appropriate storage and quick access to data. Inbuilt libraries and flexibility utilizing the available techniques help Apache Spark to divide the dataset as a convenience. In this study, partitioning techniques available in Spark have been discussed and implemented. The pros and cons of the partitioning techniques have been highlighted. Furthermore, the performance of partitioning techniques has been compared in terms of execution time which help to select the suitable partitioning strategy for the different jobs.

Keywords: Partitioning; MapReduce; Load Balancing; Big data; Hashing;

1. Introduction

The recent years witnessed exceptional growth in the amount of digital data being generated through various measures like computers, mobile devices, sensors, satellites, etc. The massive amount of data can be employed to produce high–valued information for decision-support, forecasting, business intelligence, research, and other fields or applications. However, the traditional data processing software and analysis algorithms cannot satisfy the need to process such a large amount of data efficiently. The scalable distributed computing architectures like MapReduce, Apache Spark, etc., can be employed to process and analyze diverse and large-scale datasets. Apache Spark is an efficient distributed computing framework for processing large-scale data. It has been demonstrated worthwhile, being 100 times quicker for In-Memory computing operations and around 10 times quicker for on-disk operations in comparison with Hadoop [1].

In Apache Spark, Resilient Distributed Datasets (RDD) is used to store the data. RDD is a distributed and immutable collection of objects divided into logical partitions, which might be processed on distinct nodes of the Spark cluster in parallel. It utilizes the MapReduce framework for processing the data. The MapReduce job typically runs in three phases namely Map, Shuffle, and Reduce. Mapping is done based on a feature of the data known as key by the mapper. The intermediate data is treated as key/value tuples. The shuffle operation is optional that moves the data related to a task in similar partitions. The quantity of data to be processed by each of the reducers is decided by the partitioner.

In the cluster processing environment, the main challenge is to limit the network traffic. At the point when the data is in <key, value> pair format, an individual operation can be performed on one partition at a time, therefore similar keys or range of keys needs to be put in a single partition so that for an action or transformation command the data need not travel over the network or it should be minimum. Partitioning plays an important role in data localization that helps to scale down the data shuffling across the processing nodes and reduces the network latency, which is a significant segment of transformation operation consequently reducing the completion time. The shuffle write operation is performed for redistribution of the data among partition/bucket. The single partitioning technique cannot process the faster results for all types of jobs. The selection of a suitable partitioning strategy for a task leads to the effective utilization of cluster resources that results in fast execution of the task. There are multiple partitioning strategies in Apache Spark like hash (default), range, coalesce and repartition, etc. The hash partition is a good choice for the dataset having a uniform key distribution. It distributes the workload effectively among the worker nodes but for the skewed dataset, it creates unbalanced partitions that causes the uneven load distribution among the workers and slow down the overall execution of a task. The selection of the right partitioning strategy will help in balancing the execution load among the nodes and limit the network traffic caused by data shuffling.

The outline of this work is as follows: the overview of related work and highlights of the findings are described in 2. The detailed methodology including cluster configuration, experimental setup, and existing partitioning techniques has been discussed in section 3. The outcomes of extensive experiments with different partitioning strategies for different use cases along with the execution time have been discussed in section 4. Finally conclusive remarks and future directions are included in section 5.

2. Literature Survey

For the effective load balancing on different nodes in Spark cluster and time efficient processing several techniques have been tried and introduced in big data analytics. Kwon et al [2] proposed a Skew Tuning method for dynamic repartitioning the unprocessed data of a task based on remaining time and idle node availability present on the cluster while range partitioning is utilized to distribute the unprocessed data. Rahimian et al. [3] proposed an algorithm JA-BE-JA for balanced graph partitioning that aims at resolving the balanced k-way problem. The algorithm implies heuristics to drive the system into lower energy state (energy of system defined as number of edges between nodes with different colors). With the colour distribution being maintained throughout the final result delivers balanced partitions. Utilizing JA-BE-JA and means of the Bulk Synchronous Parallel abstraction, a Balanced Graph Partitioning for Apache Spark was proposed in the study by Carlini et Al. [4]. The algorithm accesses the neighbours in the graphs and uses a small random sample of the graph to improve the access of neighbouring nodes. The relaxation of the assumption for original JA-BE-JA algorithm resulted in certain degree of approximations but after 10 runs of the algorithm it provided better results compared to original one.

Huang et al.[5] proposed an approach utilizing Apriori, branch and bound algorithms for database partitioning. Apriori algorithm along with cosine similarity is used for determining the frequent patterns in a transactional database. Branch and bound algorithm is utilized for optimal database partitioning. The method shows significant improvement in cost optimization and brings down the number of iterations significantly for accessing the leaf nodes. Huang et al. [6], proposed Smart Partitioning Mechanism (SPM) algorithm. The method is based on continuous monitoring of the data at reducers end. SPM reduces the skewness by data related to the similar keys at different reducers. Initially the data is distributed among the reducers and as the data reduces to a threshold, SPM again distributes the data among the reducers. Chen et Al. [7] proposed Mitigating Reducer Skew in MapReduce (MRSIM) consisting of MRSIM-LB and MRSIM-LBF algorithms. The algorithms calculates the load at each reducer and record the size at reducer load monitor. If a reducer during shuffle phase shows the load above the threshold value then the reducer with largest load stops and the extra data assigned to it is divided among the two reducers. The algorithm is capable of handling the heterogenous environment and performs load sharing among reducers.

Tang et Al. [8] proposed an algorithm skewed intermediate data (SCID) for handling the data skewness using splitting and combination. SCID uses the reservoir sampling algorithm and evaluates the statistical properties of the dataset including skewness and variance of the input data. The sampling algorithm runs before the MapReduce jobs for predicting the partition size in advance. SCID requires about **20%** of the input data of map phase to function properly. Tang et Al. [9] proposed another technique – key reassigning and splitting partition algorithm (SKRSP). It uses a step-based key rejection algorithm to approximate the distribution of keys in the data. The data sampling after map phase improves the overall execution efficiency and accuracy of key distribution measure. A hash based key reassigning partition algorithm is utilized to divide intermediate data in the balanced way to the reducer tasks. SKRSP works well on all applications except sort requiring operations. One such technique named Spark load balancing mechanism based on Linear Regression partition (SP-LRP) was proposed by Huang et Al. [10]. It analyse the operation statistics such as the amount of data processed by the map tasks and the partitions formed. Once the analysis is over then the partition size is predicted by the algorithm and the largest partition is fixed. The skewed data is identified and the shuffle operation fetch and merge the data from different mapper nodes and dynamically adjust the skewed data.

Li et Al. [11] introduced MBR to solve load balancing problems in Apache Spark utilizing the pre-processing scheduling and the self-adaption scheduling. The pre-processing scheduling determines the distribution of reducers. If the load assigned on a reducer is found greater than the sum of assigned load on other reducers then the data is distributed among all reducers. The self-adaption scheduling algorithm is used when the output of the mappers is sent directly to the reducers and the execution time of a reducer is greater than the sum of others then the load is balanced by data redistribution. Geetha J et Al. [12] proposed an algorithm based on dynamic partitioning of graphs in Apache Spark. The custom dynamic partitioning algorithm exploits the relationship between the nodes to decide the partitions. The input file is converted to RDDs initially and then converted to Graph RDDs. The algorithm applies mathematical methods such as Laplacian Matrix to identify the Eigen Value and Eigen Vectors. These values are forwarded to define a new graph RDD which in turn help to predict the optimal partitions. Ahmad et Al. [13] proposed a technique focused on improving the Hash partitioning function in Apache Spark. The technique aims to find an offset parameter **’c’** that can be appended at the end of the Hash function of Apache Spark, hence decreases the overall makespan time of the process. The different bucketing methods viz. modulo-bucketing and consecutive binning separate algorithms were also introduced. In the algorithm for modulo labeling, the labels for each of the bins are rotated from right to left by the same distance instead of rotating the contents of the bins which assists in finding optimal partitions faster. The second algorithm for consecutive binning considers from a subset of **’c’** values in increasing order such that for each value of c at least one of the items shifts to the next bin and crosses a threshold value of boundaries in front of it.

3. Methodology

Partitions are the fundamental unit to achieve parallelism in Apache Spark. It is a recognized division of information put away on the nodes. The logical collection of partitions is termed an RDD. The lazy evaluations are performed on RDDs that stores the transformations as Directed Acyclic Graph (DAG). When an action command like join, is called on RDDs then DAG is computed. Apache Spark supports in-memory computation, consequently, the frequent actions on RDDs are cached that saves the re-computation time in case there is a repetition of actions. Apache Spark uses the partitions equivalent to the number of cores present in the cluster to store the data. This is done to minimize the computation time of data in the partitions. In the processing of a task, partitioning plays an important role since the same task is applied to a single partition at a time. Apache spark provides various partition strategies discussed in section 3.3 that aim at the data localization and balanced partitioning of the dataset on the cluster. To understand the role of partitioning strategies in different applications, the optimal configuration of the Spark cluster that includes the number of CPU cores along with memory per executors needs to be fixed. Henceforth, partitioning strategies will be evaluated for different datasets.

3.1. Cluster Configuration

In Spark cluster two types of nodes: master node and worker node are used for executing the tasks, client submit the job at the master node and master node distributes this job using the resource manager to the worker nodes. Executor is responsible to handle the task on the worker node. While performing the parallel execution of the tasks, the optimal configuration of the Spark cluster will help to achieve the better parallelism. Single java virtual machine (JVM) can run more than one task in parallel, assigning one executor per core will lead to underutilization of JVM. Also, there will be an overhead of sharing broadcast variables and accumulators with each core. On the other hand, the assignment of one executor per node leads the higher number of CPU cores and memory into superfluous garbage results. There is a trade-off between the number of executors and CPU cores usages. We have designed an optimal executors algorithm 1 that will return the optimal number of executors for a heterogeneous Spark cluster based on the configuration provided.

Algorithm 1 Optimal Executors

```
Require: Number of Worker Nodes, CORES [], Cores Per Executor
Ensure: Optimal No. of Executors
1: $min \leftarrow \sum_{i=1}^{n} Minimum(CORES[i])$
2: if $Cores\ Per\ Executor \leq min$ then
3:     Cores per executor ← min                    ▷ Apply the differencing method
4: end if
5: $sum \leftarrow \sum_{i=1}^{n} CORES[i]$
6: available executor ← $\lfloor \frac{sum}{Cores\ per\ executor} \rfloor$
7: optimal executor ← available executor -1
8: return optimal executor
```

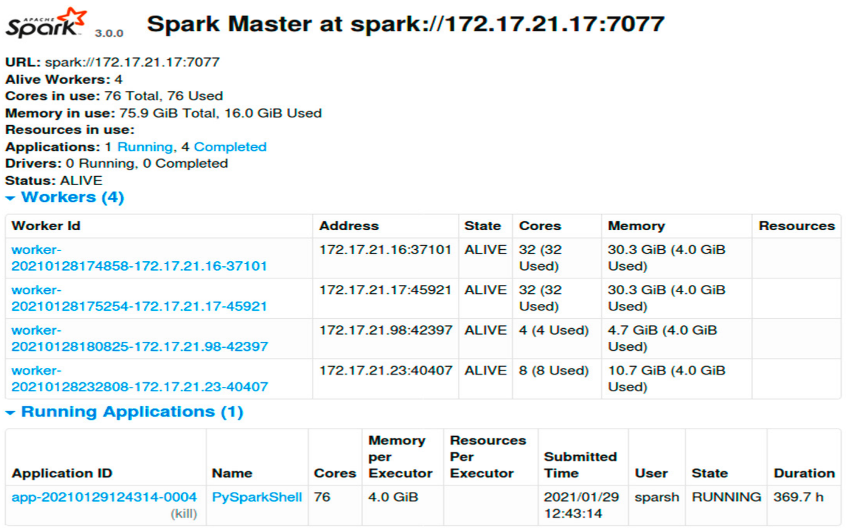
In algorithm 1, Number of worker nodes represents the worker nodes in a cluster, *CORES* is an array of available CPU cores per worker and Cores Per Executor defines the number of cores assigned per executor. The unanimity of Spark tuning guides advice to assign 5 cores to an executor for optimal parallel processing [14].

We will calculate the optimal number of executors for our cluster, utilizing the algorithm 1,

Number of Worker Nodes = 4
$CORES = [32, 32, 8, 4]$
$min = \sum_{i=1}^{n} Minimum(CORES[i]) = 4$
$Cores\ per\ executor = min = 4$
$sum = \sum_{i=1}^{n} CORES[i] = 76$
$available\ executor = \lfloor \frac{sum}{Cores\ per\ executor} \rfloor = \lfloor \frac{76}{4} \rfloor = 19$
One executor is left as application manager
$optimal\ executor = available\ executor$ -1 = 19 - 1 = 18

3.2. Experimental Setup

The experiments for this study has been performed on the heterogeneous Apache Spark Cluster (ASC) that includes two systems with Intel® Xeon(R) CPU E5-2630 v3 @ 2.40GHz processor with 32 cores, 32 GB RAM, 2 system with Intel(R) Core(TM) i7-3770 CPU @3.40GHZ with 4 Cores and 4 GB RAM and ubuntu 16.04 LTS operating system. The softwares used in the study are Python 3.7.3, Apache Spark 3.0.0 and Hadoop 2.7, the snapshot of running ASC can be seen in Fig. 1.


3.0.0 **Spark Master at spark://172.17.21.17:7077**

URL: spark://172.17.21.17:7077
Alive Workers: 4
Cores in use: 76 Total, 76 Used
Memory in use: 75.9 GiB Total, 16.0 GiB Used
Resources in use:
Applications: 1 Running, 4 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (4)

Worker Id	Address	State	Cores	Memory	Resources
worker-20210128174858-172.17.21.16-37101	172.17.21.16:37101	ALIVE	32 (32 Used)	30.3 GiB (4.0 GiB Used)	
worker-20210128175254-172.17.21.17-45921	172.17.21.17:45921	ALIVE	32 (32 Used)	30.3 GiB (4.0 GiB Used)	
worker-20210128180825-172.17.21.98-42397	172.17.21.98:42397	ALIVE	4 (4 Used)	4.7 GiB (4.0 GiB Used)	
worker-20210128232808-172.17.21.23-40407	172.17.21.23:40407	ALIVE	8 (8 Used)	10.7 GiB (4.0 GiB Used)	

Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20210129124314-0004 (kill)	PySparkShell	76	4.0 GiB		2021/01/29 12:43:14	sparsh	RUNNING	369.7 h

Fig. 1. Running Spark Cluster with 4 Worker nodes

3.3. *Partitioning Strategies*

3.3.1. *Hash Partitioning*

In hash partitioning, the data is spread among the partitions based on the hash function of the key value. Consider $C = \{C_1, C_2, C_3, ..., C_n\}$, total n number of cores available in the spark cluster then $P = \{P_1, P_2, P_3, ..., P_n\}$, n number of partitions will be created by default. A key K will move to partition i based on the formula

$$P_i = Hash\ Code\ of\ K_i \% N \quad (1)$$

Where n indicates the number of partitions.The main objective of the hash partitioner is to equally distribute the data among the available partitions. However, it depends on the distribution of keys, how the data will be mapped to the available partitions. Uneven distribution of keys may results in data skewness.

3.3.2. *Range Partitioning*

The range partitioning is considered an efficient partitioning technique for RDDs having the keys that follow a specific ordering. In this technique, tuples including keys inside a similar range will show up on a similar machine. It divides the data into partitions based on the range. The range partition provides more balanced partitions as compared to Hash Partitioning. Consider $K = \{K_1, K_2, K_3, ..., K_n\}$ a set of keys belonging to a dataset. Let $R_1 = \{K_1, K_2, ...K_5\}$,$R_2 = \{K_6, K_7, ...K_x\}$,$R_3 = \{K_{x+1}, K_{x+2}, ...K_n\}$ be set of ranges for partitioning the keys into. All the ranges defined will have an upper bound and the except the first range all the ranges will have an inclusive lower-bound.

$$P_i = K_i \in R_i \quad (2)$$

Although there still can be skewness in key distribution as there can be key distribution more weighted towards a certain range hence producing an imbalance in load distribution among the nodes.

3.3.3. *Repartition*

Repartition method is utilized to customize the number of partitions for a Spark data frame. It creates specified number of partitions by performing a complete shuffle of data among the existing partitions also keeping the size of new partitions created almost equal. Repartition can be employed both ways, to reduce the number of partitions or to increase the number of partitions. The function specified in equation 3 is used for repartition.

$$df = repartition(x, numPartitions = NULL, col = NULL) \quad (3)$$

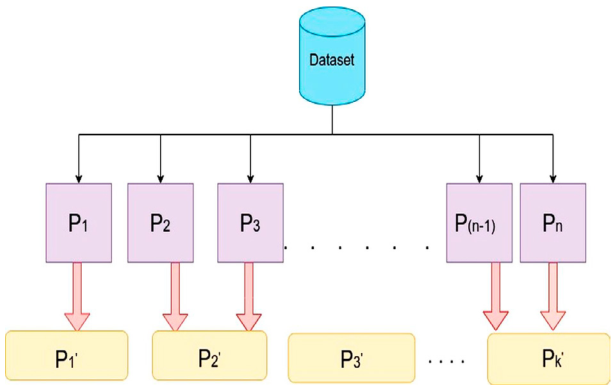


Fig. 2. Functioning of Coalesce Partitioning Strategy

where *x* is a spark data frame, *numPartitions* specifies the number of partitions required, *col* specifies the column that will be used for the partitioning. The repartition function is available in Apache Spark 1.4.0 and higher versions. Also if we specify a single column with repartition method then hash paritioning is done and partitions equal to specified number of partitions are created but when the more than one column is specified then in that case also hash partitioning is done but number of partitions equate to **spark.sql.shuffle.partitions = 200(default)** which is default number of partitions when data is shuffled for joins and aggregations[15].

3.3.4. Coalesce

Coalesce partitioning is used to obtain the desired number of partitions from the available partitions. It returns the Spark Data Frame that has precisely *numPart* partitions, where *numPart* is the user defined variable. This method leads to a narrow dependency, for instance, the conversion from 100 to 10 number of partitions, instead of shuffle, every new partition from 10 partitions will demand 1 of the current partitions. Unlike repartition, the coalesce method reduces the shuffle operation to a very large extent. Also, it is used only when the number of partitions have to be reduced, as can be seen in Fig. 2.

$$Y = coalesce(x, numPartitions) \tag{4}$$

where, x denotes a Spark Data Frame or a Column and *numPartitions* denotes the number of partitions to be done. Although the partition numbers obtained are as required and none of them has zero size, but the partition sizes are not equal as compared to repartition. Sometimes the difference between the partition sizes is very large. The coalesce method is generally faster as compared to the repartition but it is generally a slower process to work with the partitions that have considerable differences in their sizes. The coalesce function is available in Apache Spark 2.1.1 and higher versions

3.3.5. Repartitioning vs Coalesce

The repartition method can be used when we want to increase or decrease the total number of partitions contrary to the fact that we can only use coalesce to decrease the number of partitions. What repartition does is basically it performs a full shuffle of data and forms the specified number of partitions that are roughly equi-sized completely from scratch without taking into account any aspect of the existing partitions while we can say coalesce is a bit intelligent in this case as it takes into account the already existing partitions and shuffles data across them in such a way to create the specified number of partitions. This makes the coalesce much faster but as we know nothing is perfect and coalesce is also no exception. On collating the partitions created by the two methods it is clearly visible that a considerable difference exists in the size of partitions created by coalesce.

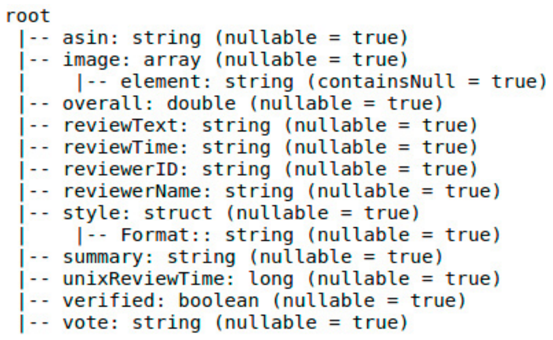
4. Result and Discussion

The selection of partitioning strategy for a task depends on the operations that are being performed on the data. To test the effectiveness of the partition strategies for the different jobs, two case studies have been discussed.

4.1. *Word Count job using different partition strategies*

4.1.1. *Dataset Used*

In the first case study, the word count job was executed on the Amazon Product Reviews dataset [16]. The data was in JavaScript Object Notation (JSON) format acquiring 12 features, the schema for the same generated by python script in pyspark environment can be seen in the Fig. 3. The customers review are stored in the reviewText attribute that has been utilized for the Word Count purpose.



```
root
 |-- asin: string (nullable = true)
 |-- image: array (nullable = true)
 |    |-- element: string (containsNull = true)
 |-- overall: double (nullable = true)
 |-- reviewText: string (nullable = true)
 |-- reviewTime: string (nullable = true)
 |-- reviewerID: string (nullable = true)
 |-- reviewerName: string (nullable = true)
 |-- style: struct (nullable = true)
 |    |-- Format:: string (nullable = true)
 |-- summary: string (nullable = true)
 |-- unixReviewTime: long (nullable = true)
 |-- verified: boolean (nullable = true)
 |-- vote: string (nullable = true)
```

Fig. 3. Amazon Product dataset Schema (Features with data types) generated using Apache Spark

4.1.2. *Execution*

There are various product categories with different volumes for the Amazon Product reviews dataset. The word count job has been executed on the reviewText attribute utilizing the Apache Spark cluster setup discussed in section 3.2, the reviewText attribute is in string format. The RDD has been partitioned utilizing the Hash, Coalesce, Repartition, and Range partitioning techniques. The resultant RDD has been utilized to execute the word count job.

Table 1. Running time for Word Count using different Partition Strategies

Input	**Running Time (Seconds)**			
Size (MB)	***Hash***	***Coalesce***	***Repartition***	***Range***
166.14	4.98	4.76	7.54	10.87
373.89	8.14	8.04	13.64	18.07
767.76	11.86	10.67	21.07	27.87
2370	37.2	36.5	85.6	102.47
7500	102.48	100.19	339.96	365.24

The RDD build after applying the Coalesce partition technique takes the minimum execution time for the word count job. The results have been tested against the different input capacity. The input size and execution time utilizing the different partition strategies can be seen in the table 1. RDD with Hash partitioning also have the execution time very close to Coalesce partitioning. The execution time taken by the RDDs partitioned by repartition and range technique is very poor compared to the Hash and Coalesce partitioning. The comparision for the execution time for the different partitioning strategies can be seen in the Fig. 4(a). The maximum time is taken by the range partitioning among all.

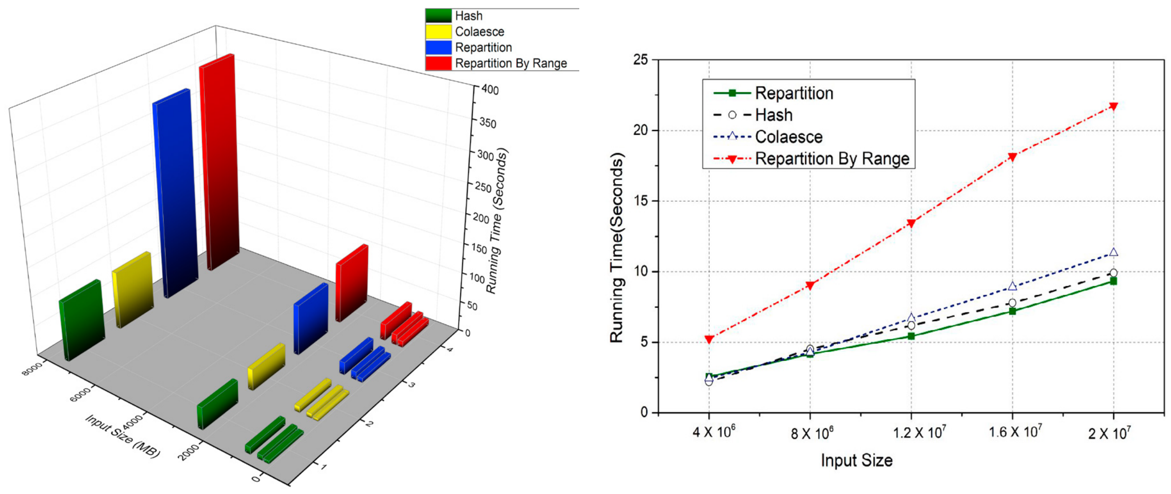
4.2. *Sorting Job*

The second experiment was performed for sorting the numeric values. An RDD was formed against the randomly generated numbers utilizing the NumPy library. RDD was re-partitioned utilizing the various partitioning schemes to perform the experiments. The different size of RDDs containing $\{4X10^6, 8X10^6, 1.2X10^7, 1.6X10^7, 2.0X10^7\}$ randomly generated numbers were executed for the sorting job and the running time for these jobs have been stored in Table 2.

Table 2. Running time for sorting using different Partition Strategies

Input	Running Time (Seconds)			
Size (MB)	*Hash*	*Coalesce*	*Repartition*	*Range*
$4 * 10^6$	2.55	2.18	2.44	5.24
$8 * 10^6$	4.15	4.51	4.27	9.05
$1.2 * 10^7$	5.42	6.18	6.68	13.46
$1.6 * 10^7$	7.19	7.77	8.9	18.17
$2 * 10^7$	9.31	9.89	11.3	21.75

RDD created by applying the Repartition and Coalesce partitioning schemes take the minimum time for the small size of inputs i.e. up to $8 * 10^6$ number of inputs, after that Hash partition performed better while the range partitioning takes the maximum time among all. The comparative performance for different size of inputs can be seen in the Fig. 4(b).



(a) Word Count Task (b) Sorting Task

Fig. 4. Comparison for Partitioning strategies for (a) Word Count (b) Sorting

4.3. *Join Operation*

4.3.1. *Dataset Used*

The third experiment was performed for joining two dataframes.The first was formed by combining three files in Comma Separated Variable (csv) format. The total number of records in the file were 6015746. The study was performed on the Crimes in Chicago dataset obtained from Kaggle[17]. The dataset consisted of 23 attributes out of which 7 were selected with the schema generated by python script in pyspark environment shown in Fig. 5. The Ward column is used for performing the join operation.

4.3.2. *Execution*

There are five small dataframes that have been formed by using one file of the downloaded dataset, each consisting of 120000,240000,360000,480000,600000 records respectively. The larger dataframe was converted into an rdd for partitioning using different strategies and then converted back to dataframes for performing the join sequnetially to the smaller dataframes.The recorded execution times are quoted in the table 3

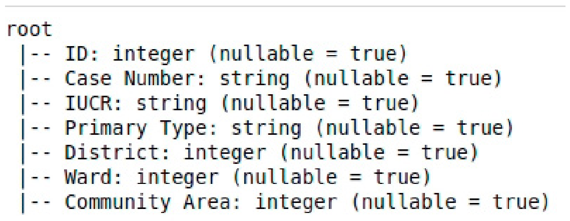

```
root
 |-- ID: integer (nullable = true)
 |-- Case Number: string (nullable = true)
 |-- IUCR: string (nullable = true)
 |-- Primary Type: string (nullable = true)
 |-- District: integer (nullable = true)
 |-- Ward: integer (nullable = true)
 |-- Community Area: integer (nullable = true)
```

Fig. 5. Crimes in Chicago dataset Schema (Features with data types) generated using Apache Spark

Table 3. Running time for Join Operation using different Partition Strategies

Input	Running Time (Seconds)			
Size (MB)	*Hash*	*Coalesce*	*Repartition*	*Range*
$1.2 * 10^5$	5378	4185	1058	744
$2.4 * 10^5$	5213	4077	1020	727
$3.6 * 10^5$	5205	3988	954	709
$4.8 * 10^5$	5196	3930	835	686
$6.0 * 10^5$	5043	3835	751	647

Note: The number of input corresponds to the smaller dataframe sizes which are being joined serially to the larger dataframe of size 6015746.

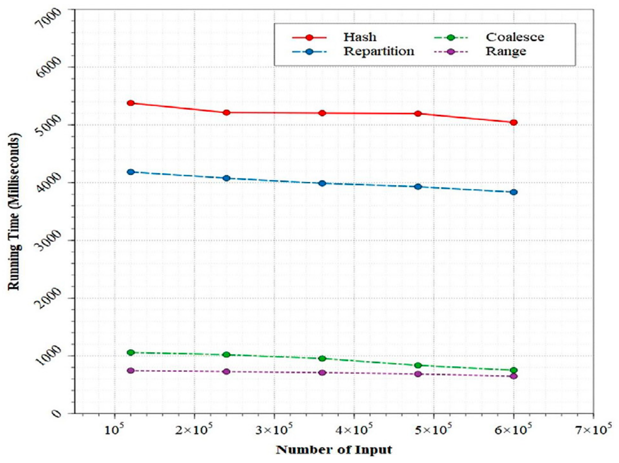


Fig. 6. Comparison of Join Operation

The comparative performance between the partitioning techniques can be seen in the Fig. 6 below. The best performance is shown by Range Partitioning which was expected due to the fact that it sorts the data within ranges. Next up is Coalesce which takes less time due to reduced shuffling of data between partitions. The Hash partitioning shows the worst performance.

5. Conclusions and Future Work

Apache Spark utilizes partitioning techniques for parallelizing the execution of tasks, thereby accelerating their processing. Choosing appropriate partitioning techniques for RDDs can speed up the execution of a task. This study highlights the implementation and usefulness of the partitioning strategies for different problem domains. The efficiency of partitioning techniques is compared based on the execution time. When solving a task where the order of occurrence for the key element is not important, such as a word count task, the Hash and Coalesce partitioning techniques are recommended. The sorting task is better served by partitioning and hashing since the order of occurrence for key elements matters. Though Repartition takes the least time for partitioning, the overall processing time may be more as the data locality vanishes. Hash partitioning is effective yet for certain applications, it may result in data skewness. A suitable partition strategy can be selected considering the data type, type of processing, and level of load balancing required. In future work, it is important to carry out the techniques which consider all kinds of boundaries and increase the execution efficiency of the Big Data.

References

[1] Yassir Samadi, Mostapha Zbakh, and Claude Tadonki. "Comparative study between Hadoop and Spark based on Hibench benchmarks". In: *2016 2nd International Conference on Cloud Computing Technologies and Applications (CloudTech)*. IEEE. 2016, pp. 267–275.

[2] YongChul Kwon et al. "Skewtune: mitigating skew in mapreduce applications". In: *Proceedings of the 2012 ACM SIGMOD International Conference on Management of Data*. 2012, pp. 25–36.

[3] Fatemeh Rahimian et al. "Ja-be-ja: A distributed algorithm for balanced graph partitioning". In: *2013 IEEE 7th International Conference on Self-Adaptive and Self-Organizing Systems*. IEEE. 2013, pp. 51–60.

[4] Emanuele Carlini et al. "Balanced graph partitioning with apache spark". In: *European Conference on Parallel Processing*. Springer. 2014, pp. 129–140.

[5] Yin-Fu Huang and Chen-Ju Lai. "Integrating frequent pattern clustering and branch-and-bound approaches for data partitioning". In: *Information Sciences* 328 (2016), pp. 288–301.

[6] Tzu-Chi Huang et al. "Smart partitioning mechanism for dealing with intermediate data skew in reduce task on cloud computing". In: *2017 IEEE 31st International Conference on Advanced Information Networking and Applications (AINA)*. IEEE. 2017, pp. 819–826.

[7] Lei Chen et al. "MRSIM: mitigating reducer skew In MapReduce". In: *2017 31st International Conference on Advanced Information Networking and Applications Workshops (WAINA)*. IEEE. 2017, pp. 379–384.

[8] Zhuo Tang et al. "An intermediate data placement algorithm for load balancing in spark computing environment". In: *Future Generation Computer Systems* 78 (2018), pp. 287–301.

[9] Zhuo Tang et al. "An intermediate data partition algorithm for skew mitigation in spark computing environment". In: *IEEE Transactions on Cloud Computing* (2018).

[10] Zichun Huang, Wenguo Wei, and Guiyuan Xie. "Load Balancing Mechanism Based on Linear Regression Partition Prediction in Spark". In: *Journal of Physics: Conference Series*. Vol. 1575. 1. IOP Publishing. 2020, p. 012109.

[11] Jianjiang Li et al. "Map-balance-reduce: an improved parallel programming model for load balancing of MapReduce". In: *Future Generation Computer Systems* 105 (2020), pp. 993–1001.

[12] J Geetha, DS Jayalakshmi, and NG Harshit. "Implementation and performance analysis of dynamic partitioning of graphs in Apache Spark". In: *International Journal of Advanced Computer Research* 10.48 (2020), p. 116.

[13] Zafar Ahmad et al. "Improved MapReduce Load Balancing through Distribution-Dependent Hash Function Optimization". In: *2020 IEEE 26th International Conference on Parallel and Distributed Systems (ICPADS)*. IEEE. 2020, pp. 9–18.

[14] Ken Withee Jason Howell. *Cluster configuration optimization for Apache Spark*. 2020. URL: `https://docs.microsoft.com/en-us/azure/hdinsight/spark/optimize-cluster-configuration`.

[15] *Apache Spark Documentation*. `http://spark.apache.org/docs/latest/sql-performance-tuning.html`.

[16] UCSD Julian McAuley. *Amazon review data*. (Accessed on 11/10/2020). 2018. URL: `http://jmcauley.ucsd.edu/data/amazon/`.

[17] *Crimes in Chicago*. (Accessed on 06/07/2021). 2017. URL: `https://www.kaggle.com/currie32/crimes-in-chicago`.