IMPROVING EFFICIENCY OF APACHE SPARK BY TUNING ITS INTERNAL FEATURES

Project report submitted in partial fulfillment of the requirement for the degree of Bachelor of Technology

in

Computer Science and Engineering/Information Technology

By

Shivank Prajapati 191424

Arnav Saraswat 191544

Under the supervision of

Dr. Hari Singh

to

Department of Computer Science & Engineering and Information Technology

Jaypee University of Information Technology, Waknaghat, Solan-173234, Himachal Pradesh

CERTIFICATE

This is to certify that the work which is being presented in the project report titled “IMPROVING EFFICIENCY OF APACHE SPARK BY TUNING ITS INTERNAL FEATURES” in partial fulfilment of the requirements for the award of the degree of B.Tech in Computer Science And Engineering and submitted to the Department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology, Waknaghat is an authentic record of work carried out by SHIVANK PRAJAPATI, 191424 and ARNAV SARASWAT, 191544 during the period from January 2023 to May 2023 under the supervision of Dr. HARI SINGH, Department of Computer Science & Engineering and Information Technology, Jaypee University of Information Technology, Waknaghat.

SHIVANK PRAJAPATI,191424

ARNAV SARASWAT,191544

The above statement made is correct to the best of my knowledge.

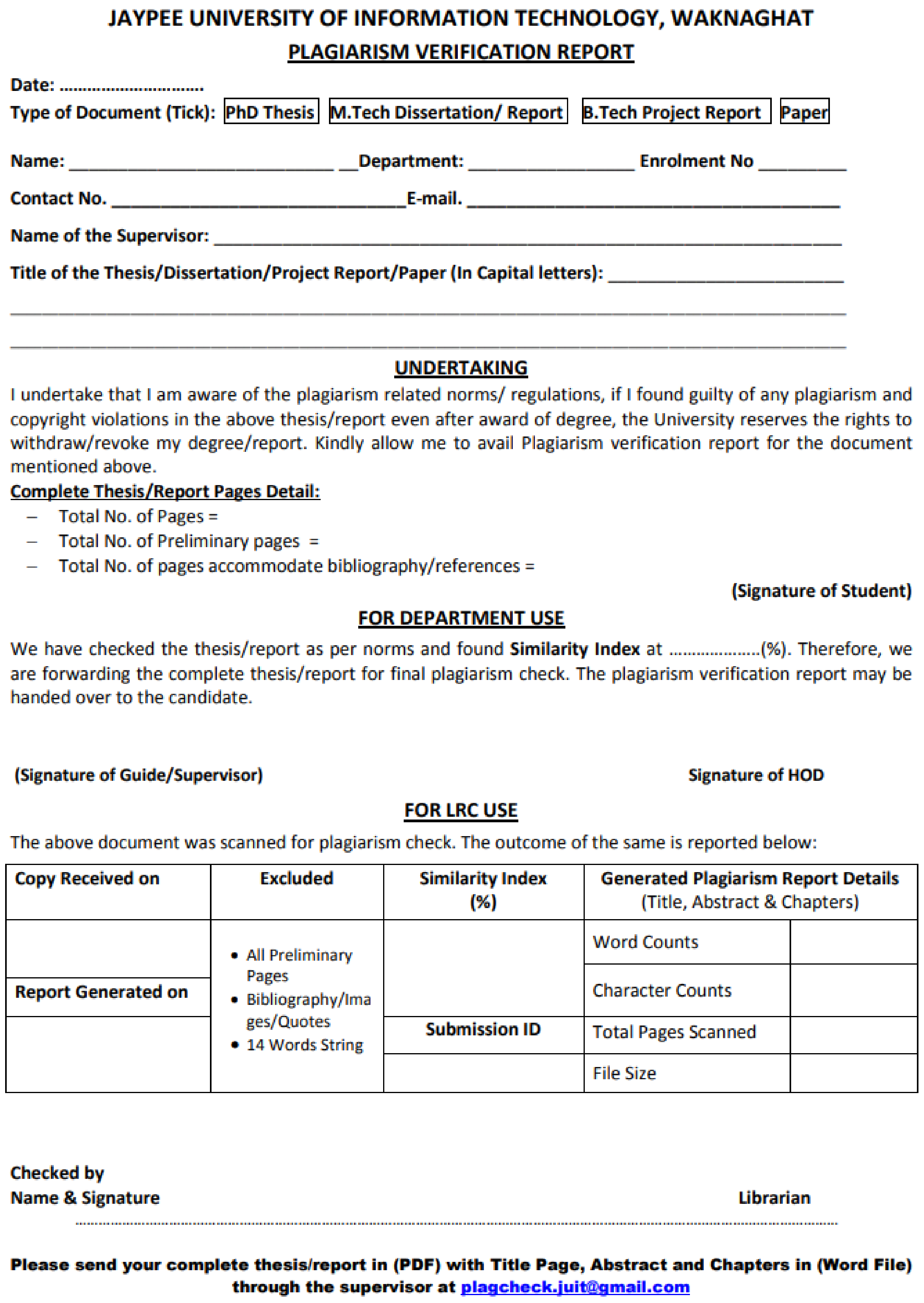
Dr. HARI SINGH

ASSISTANT PROFESSOR (SG)

Computer Science & Engineering and Information Technology

Jaypee University of Information Technology, Waknaghat.

PLAGIARISM CERTIFICATE



JAYPEE UNIVERSITY OF INFORMATION TECHNOLOGY, WAKNAGHAT

PLAGIARISM VERIFICATION REPORT

Date: ....

Type of Document (Tick): PhD Thesis M.Tech Dissertation/ Report B.Tech Project Report Paper

Name: ______________________________ __Department: ________________ Enrolment No _________

Contact No. _________________________________E-mail. ____

Name of the Supervisor: ____

Title of the Thesis/Dissertation/Project Report/Paper (In Capital letters): _______________________

____

____

UNDERTAKING

I undertake that I am aware of the plagiarism related norms/ regulations, if I found guilty of any plagiarism and copyright violations in the above thesis/report even after award of degree, the University reserves the rights to withdraw/revoke my degree/report. Kindly allow me to avail Plagiarism verification report for the document mentioned above.

Complete Thesis/Report Pages Detail:

– Total No. of Pages =
– Total No. of Preliminary pages =
– Total No. of pages accommodate bibliography/references =

(Signature of Student)

FOR DEPARTMENT USE

We have checked the thesis/report as per norms and found **Similarity Index** at ....(%). Therefore, we are forwarding the complete thesis/report for final plagiarism check. The plagiarism verification report may be handed over to the candidate.

(Signature of Guide/Supervisor) **Signature of HOD**

FOR LRC USE

The above document was scanned for plagiarism check. The outcome of the same is reported below:

Copy Received on	**Excluded**	**Similarity Index (%)**	**Generated Plagiarism Report Details** (Title, Abstract & Chapters)	
	• All Preliminary Pages • Bibliography/Images/Quotes • 14 Words String		Word Counts	
Report Generated on			Character Counts	
		Submission ID	Total Pages Scanned	
			File Size	

Checked by
Name & Signature **Librarian**

....

Please send your complete thesis/report in (PDF) with Title Page, Abstract and Chapters in (Word File) through the supervisor at plagcheck.juit@gmail.com

Candidate's Declaration

I hereby declare that the work presented in this report entitled **"Improving efficiency of Apache spark by tuning its internal features"** in partial fulfillment of the requirements for the award of the degree of **Bachelor of Technology** in **Computer Science and Engineering/Information Technology** submitted in the department of Computer Science & Engineering and Information Technology**,** Jaypee University of Information Technology Waknaghat is an authentic record of my own work carried out over a period from January 2023 to May 2023 under the supervision of **Dr. Hari Singh** (Assistant Professor (SG), Computer Science & Engineering and Information Technology).

I also authenticate that I have carried out the above-mentioned project work under the proficiency stream **Cloud Computing**.

The matter embodied in the report has not been submitted for the award of any other degree or diploma.

Shivank Prajapati, 191424.   Arnav Saraswat, 191544.

This is to certify that the above statement made by the candidate is true to the best of my knowledge.

Dr. Hari Singh
Assistant Professor (SG)
Computer Science & Engineering and Information Technology
Dated:

ACKNOWLEDGEMENT

Firstly, we express our heartiest thanks and gratefulness to almighty God for His divine blessing makes us possible to complete the project work successfully.

We are really grateful and wish my profound my indebtedness to Supervisor Dr. Hari Singh, Assistant Professor (SG), Department of CSE & IT Jaypee University of Information Technology, Wakhnaghat. Deep Knowledge and keen interest of my supervisor in the field of “Data Science, Data Processing and Cloud Computing” to carry out this project. His endless patience, scholarly guidance, continual encouragement, constant and energetic supervision, constructive criticism, valuable advice, reading many inferior drafts and correcting them at all stage have made it possible to complete this project.

We would like to express our heartiest gratitude to Dr.Pradeep Kumar Gupta, Associate Professor, Department of CSE, for his kind help to finish this project.

We would also generously welcome each one of those individuals who have helped us straight forwardly or in a roundabout way in making this project a win. In this unique situation, we might want to thank the various staff individuals, both educating and non-instructing, which have developed their convenient help and facilitated our undertaking.

Finally, we must acknowledge with due respect the constant support and patients of our parents.

Shivank Prajapati, 191424. Arnav Saraswat, 191544.

TABLE OF CONTENT

LIST OF FIGURES

LIST OF TABLES

ABSTRACT

We are always enhancing Spark's speed and usefulness. To improve Spark's usability, we and other community members are adding a substantial number of standard libraries that provide scaled variations of popular data analysis methods. For instance, in the previous year, the size of Spark's MLlib machine learning library increased by a factor of 4. Additionally, utilising DataFrames or SQL, it is simple to access external data sources using our pluggable data source API. These APIs make up one of the most integrated standard libraries for "big data" and will surely prompt creative design choices that will make the building of workflows more effective.

Big data is used to refer to data of the order of terabytes and beyond. This data is often difficult to process due to its sheer size. This is where a solution in the form of big data processing and handling platforms comes in. Apache spark is one such open-source platform. Spark has many configurational parameters that can affect the execution time to various degrees depending upon the nature of the job and manually changing these configurations to achieve the best configuration for the job can be very challenging. After assessing various works and studies, we have decided on using Grid search with a finer tuning, Controlled Random Search and ANN algorithms to find the best configurations for achieving a better efficiency. Ultimately, we find the fastest algorithm that can compute the best configurations.

Additionally, Spark is being utilized more and more in research initiatives, such as large-scale neuroscience, graph processing, online aggregation, and genomic data processing. We anticipate that Spark's sizable amount of built-in functionality and small amount of code will make it useful for both system- and application-oriented projects.

This article's functionality is all open source and accessible at spark.apache.org.

CHAPTER 1: INTRODUCTION

1.1 Introduction

There are tremendous potential and formidable computational hurdles associated with the expanding volume of data in business and research. Users required a new mechanism for spreading computing across numerous nodes when the volume of data exceeded the capacity of a single computer. As a result, a range of innovative cluster programming paradigms aimed at different workloads have proliferated. New models were created for new workloads since the original versions of these models were rather specialised. For instance, Google created Pregel for iterative graph algorithms and Dremel for interactive SQL queries, while MapReduce offered batch processing. The open-source Apache Hadoop platform is also the focus of systems like Storm and Impala. One-size-fits-all approaches are becoming less popular, even in the realm of relational databases. Unfortunately, the majority of big data applications call for a combination of diverse processing techniques. Diversity and chaos are at the heart of "big data." A typical pipeline needs SQL-style queries, iterative machine learning, and code similar to MapReduce to load data. Therefore, a separate engine may result in complexity as well as inefficiency. Some applications can't be represented in any engine well, forcing users to patch together many solutions.

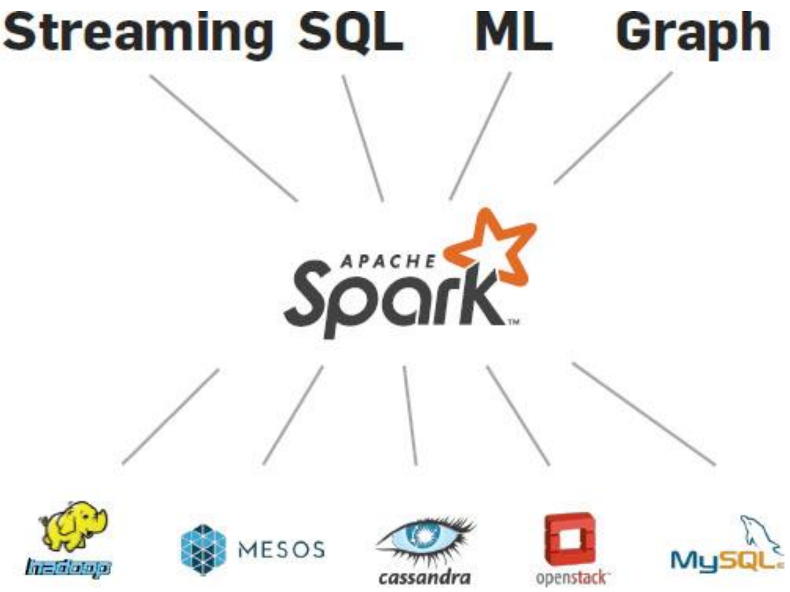


Figure 1. The implementation of customised processing libraries over the basic engine of the Apache Spark software stack.

A group from the University of California, Berkeley began work on the Apache Spark project in 2009 with the goal of creating an uniform engine for distributed data processing. Spark expands MapReduce's programming style with a "Stable Distributed Dataset," or RDD, which is an abstraction for data-sharing. With the help of this straightforward addition, Spark is now able to execute a variety of processing workloads that previously needed different engines, such as SQL, streaming, machine learning, and graph processing (see Figure 1). These implementations achieve comparable performance by employing the same optimizations as the specialised tools (such as incremental updates and column-oriented processing), but they run like libraries on a single engine, making them straightforward and effective. We claim that the results are more general than particular to certain workloads. When combined with data sharing, MapReduce may imitate any distributed computation and can thus handle a wide range of workloads.

Spark's generality has numerous vital blessings. First, apps are less difficult to expand due to the fact they use a unified API. second, combining processing responsibilities is extra green; whereas earlier systems required writing records to storage for transmission to every other engine, Spark can perform many extraordinary functions at the identical statistics, typically in reminiscence. subsequently, Spark allows new programs (together with graph-interactive queries and non-stop system learning) that had been not viable with preceding structures.

A powerful analogy on the fee of unity is to compare smartphones with disparate cellular gadgets that preceded them (consisting of cameras, mobile phones, and GPS gadgets). with the aid of unifying the features of these devices, smartphones have enabled new packages that combine their features (together with video messaging and Waze) that have been no longer viable on a single device. bag. for the reason that its launch in 2010, Spark has end up the maximum lively open source or large facts processing undertaking, with over 1,000 individuals. The assignment is utilized by greater than 1,000 companies, from generation organizations to banking, retail, biotechnology and astronomy.

The maximum extensively publicized implementation is over eight thousand nodes. As Spark advanced, the group sought to maintain to leverage its energy as a unified engine. The crew hold to construct an included well-known library on pinnacle of Spark, with capability from statistics ingestion to gadget getting to know. customers locate this potential powerful; in surveys we discover that most of the people of customers contain more than one Spark libraries in their utility. As parallel statistics processing will become mainstream, the composability of processing functions could be one of the maximum critical concerns for both usability and overall performance. an awful lot of the records evaluation is exploratory, with customers trying to fast comprise library capabilities right into a single workflow. however, for huge data particularly, duplicating facts between extraordinary systems is obvious for overall performance. consequently, users want generalizable and composable summaries.

1.2 Problem Statement

“IMPROVING EFFICIENCY OF APACHE SPARK BY TUNING ITS INTERNAL FEATURES.”

We all know that when developing a program, it is important to pay attention to performance as it helps in computing in-memory data. Spark jobs can be optimized in many ways, so let us take a closer look at each one.

Spark and Its Capabilities

It is a free and open-source cloud (cluster) computing framework for actual-time records processing. One of the key features is in-memory (main memory) cluster computing, which in turn speeds up software processing. Spark presents a programming interface across clusters with facts parallelism that is implicit and is also fault tolerant. It covers a wide variety of workloads, like batch packages, iterative algorithms, interactive queries and streaming.

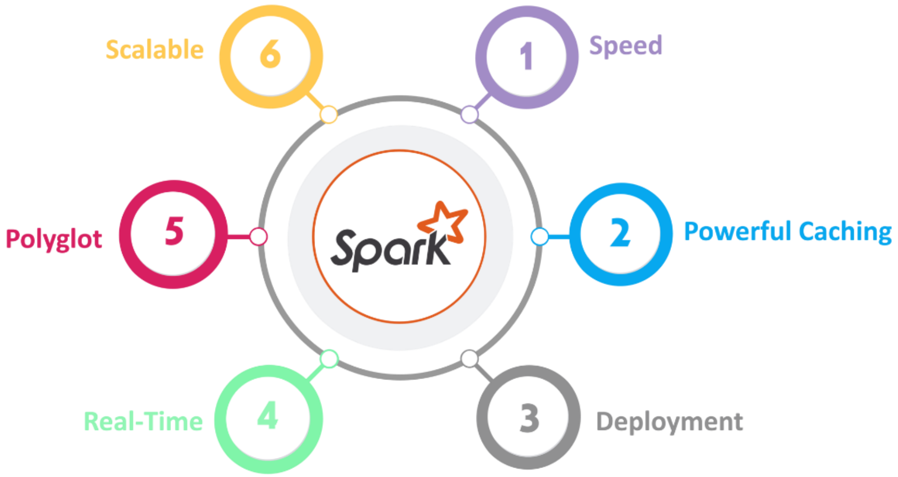


Figure 2. Features of Apache Spark.

- **SPEED**

For processing big amounts of data, Spark about hundred times quicker than Hadoop's MapReduce. it is able to also get this speed through properly managed partitioning.

- **STRONG CACHING**

Powerful cache and disc built-in abilties are supplied via a easy coding layer.

- **DEPLOYMENT**

It may be utilized by integrating Mesos, YARN for Hadoop, or Spark's cluster management.

- **REAL-TIME**

Because of in-memory (main memory) calculation, it gives decreased latency and actual-time computation.

- **POLYGLOT**

Excessive-stage APIs for Java, Scala, R, and Python are offered built-in Polyglot Spark. those four languages are all capable of generating Spark code. additionally, it gives Python and Scala shells.

Summary Spark’s architecture

Spark's layers and components are loosely connected and feature a properly built-in tiered layout. extra extensions and libraries are applied integratedto this layout.

Figure 3 describes the basic architecture of Spark. The strength of Spark is constituted of two abstractions:

- Directed acyclic Graph (DAG)
- Resilient distributed Dataset (RDD)

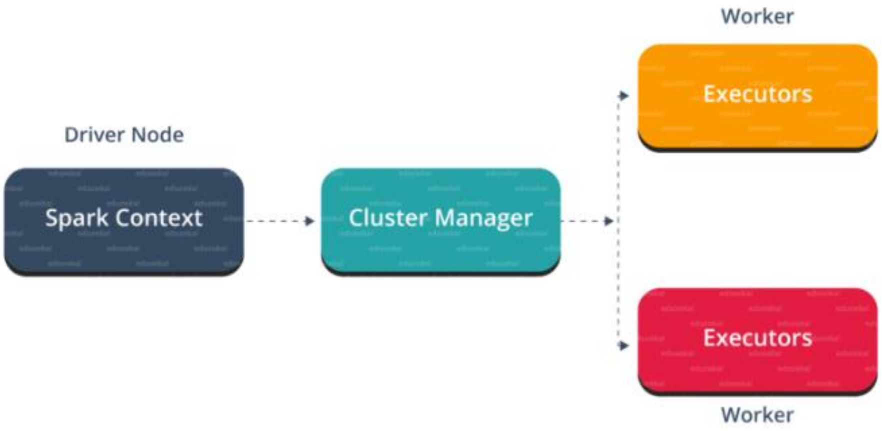


Figure 3. Architecture of Spark

As evident from *Figure 4*, the Spark system has components like Spark Streaming, SQL, Mlib, etc.

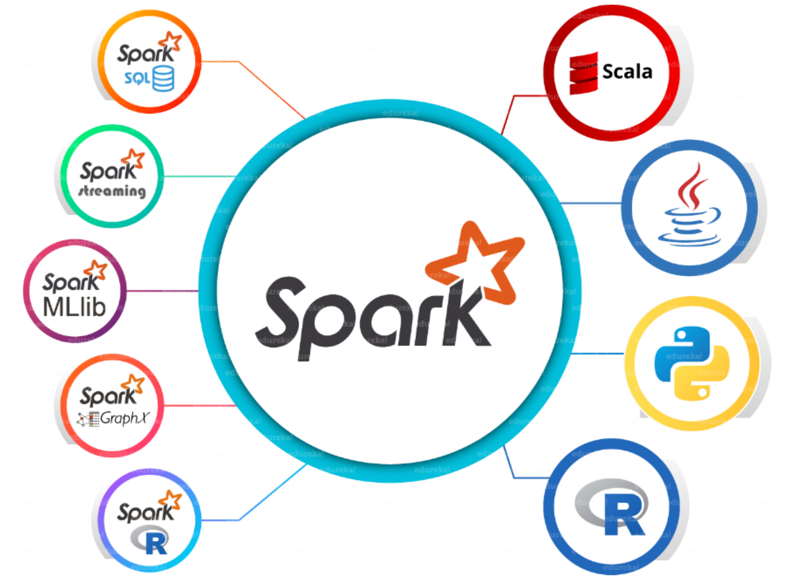


Figure 4. Ecosystem of Spark

- **SPARK CORE**

Spark center is the base engine for massively parallel and allotted computing. moreover, additional libraries constructed on pinnacle of the center allow a ramification of streaming, sq., and gadget mastering workloads. it is chargeable for managing and troubleshooting garage, scheduling jobs inside a cluster, distributing and monitoring, and interacting with storage structures.

- **SPARK STREAMING**

Spark Streaming, part of Spark used to method real-time data streaming. So it's a beneficial addition to the core of Spark's API. This permits excessive-throughput and stream processing of live records streams which is fault tolerant.

- **Spark SQL**

Spark SQL, a newer module for Spark that merges relational computation with Spark API's functional programming. helps querying statistics through SQL or Hive query language. For those acquainted with RDBMS, Spark SQL can ease migration from previous equipment and push the limits of conventional relational computing.

- **GraphX**

GraphX, a Spark API for Graphs and Graph-parallel computing. therefore, we make bigger Spark RDDs with resilient dispensed assets graphs. At a excessive degree, GraphX extends Spark's 1RDD abstraction by way of introducing a resilient disbursed property Graph - a Directed multigraph with properties associated with each Edge and Vertex.

- **MLlib (Machine Learning)**

MLlib is machine learning Library. Spark MLlib is used to run machine learning on Apache Spark.

- **SparkR**

This is an R package. Provides an implementation of disbursed dataframes.

Spark is packed with high-level libraries together with guide for R, square, Python, Scala, Java, and more. those general libraries power seamless integration into complex workflows. moreover, you can expand its abilties by means of integrating numerous services which include MLlib, GraphX, sql data frames, and streaming services.

Resilient Distributed Dataset (RDD)

RDDs are the essential for any Spark application. RDD stands for:

- Resilience: it's far fault tolerant and may recover statistics in case of failure.
- distributed: facts dispensed across multiple nodes inside the cluster.
- Datasets: collections of cut up information with values.

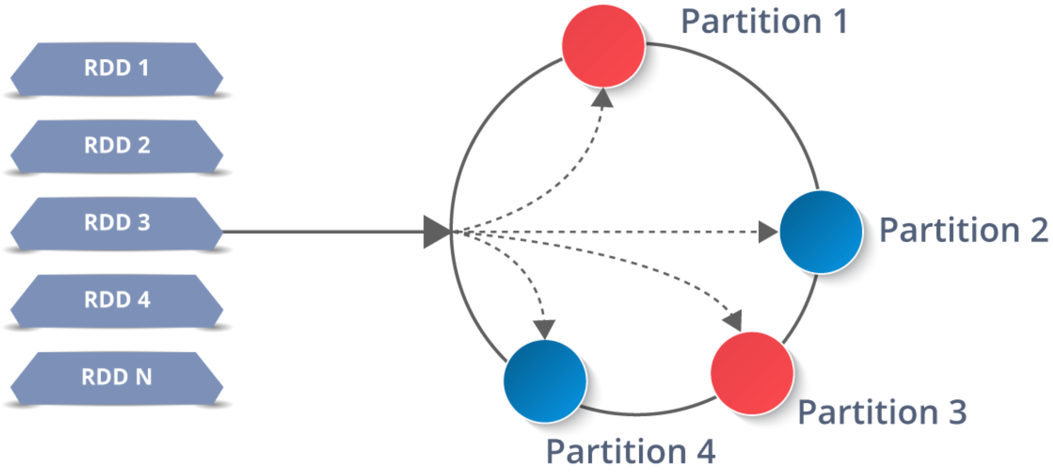


Figure 5. RDD system in Spark

This is a layer of abstracted facts on pinnacle of disbursed collections. it's far inherently immutable and obeys lazy ameliorations.

Information in an RDD is split into blocks based totally on keys. RDDs are very resilient. The same statistics block is replicated to multiple executor node, so you can fast get over troubles. So if one Executor nodes fails, every other node will preserve to process the facts. This lets in you to leverage the abilties of a couple of nodes to carry out characteristic computations in no time for your dataset.

It's an abstraction of the disbursed collection's records. it is unchanging by nature and lazy modifications.

An RDD divides its data into sections consistent with a key. RDDs are very strong, meaning that that they can speedy get better from any problems since the identical facts chunks are duplicated over several executor nodes. hence, information processing will continue despite the fact that 1 Executor nodes fails. using the speed of several nodes, we may also swiftly behavior our purposeful computations towards our dataset in this manner.

Moreover, as soon as created, RDDs are immutable. Immutable way an item whose country cannot be modified after advent, but which may be reliably transformed.

Speaking of distributed environments, records in an RDD are split into logical walls that may be computed on specific nodes of the cluster. This permits us to carry out adjustments or movements at the whole facts in parallel. don't worry about the distribution either. due to the fact Spark will deal with that. *Figure 6* describes working of Spark RDDs. *Figure 5* showcases the RDD system.

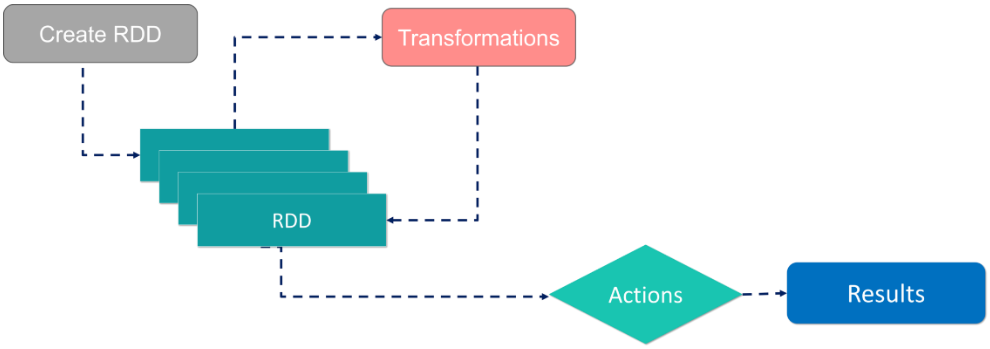


Figure 6. RDD workflow

RDDs may be created by the use of methods, parallelize current collections in driver software, and to reference records units in external storage systems together with: Shared report structures, HDFS, HBase, and so on.

RDDs can help you perform following type of operations:

- Transformation: Operations applied to create new RDDs.
- Action: implemented to an RDD, tells Apache Spark to apply a computation and return the end result to the driver.

Working of Spark's architecture

The master node has a driver application that drives the utility (see *Figure 7*). The code we write behaves like a driver software. With an interactive shell, the shell acts as a driver software. in the driver code, first create a Spark context. think the Spark context is the gateway to all Spark capabilities. this is similar to database connections. All instructions that run at the database undergo the database connection. in addition, the entirety we do in Spark goes through the Spark context.

The Spark context functions with the cluster supervisor now, controlling the tasks/jobs. The driver software and Spark context manage activity execution in the cluster. A task is cut up into a couple of tasks which are dispensed across worker nodes. each time an RDD is created in Spark context, it is able to be dispensed to one-of-a-kind nodes and cached there.

A worker node is essentially a slave node with a job to execute an assignment. these obligations are accomplished on a partitioned RDD on the worker node, so the results are back to his Spark context.

The Spark context picks up the process, splits the job into duties and dispatches them to worker nodes. those responsibilities operate on a partitioned RDD, carry out operations, gather consequences, and go back to the primary Spark context.

Increasing the range of employees permits jobs to be cut up throughout a couple of partitions and run in parallel on multiple systems. it will likely be plenty faster.

extra workers means greater memory size, which permits jobs to be cached and run quicker.

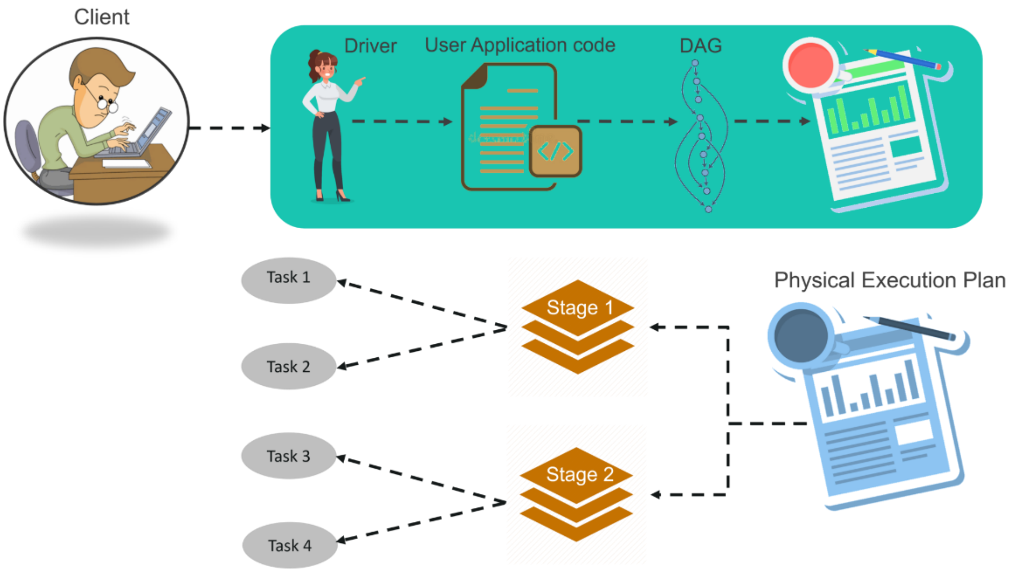


Figure 7. Infographic of Spark architecture

- **Step 1:** Client sends Spark user software code. when software code is sent, the driver

  transforms user code, along with variations and movements, right into a logical graph (DAG). This section also performs optimizations consisting of pipeline variations.

- **Step 2:** The logical graph, DAG, is then transformed into a physical execution hierarchy with many tiers. Then, every segment creates an execution unit referred to as a project. Then the tasks bundled and despatched to the cluster.

- **Step 3:** Here the driver communicates with the cluster supervisor/manager and demands resources. The cluster manager/supervisor begins executors on workers nodes on behalf of driver. Then driver submits the task to the executor based on data alignment. whilst the performer starts, it registers with the driver. therefore, the driving force has a complete overview of the performers performing the mission.

- **Step 4:** All through task execution, the driver software monitors running executors. The drivers node also configure future duties based totally on data placement/position.

1.3 Objective and Methodology

Spark overall performance tuning means manner of adjusting the settings for recording the cores, memory, and times used for your system. This manner ensures greatest Spark overall performance and forestalls Spark resource hunger. Areas of performance tuning in Spark are described in *Figure 8*.

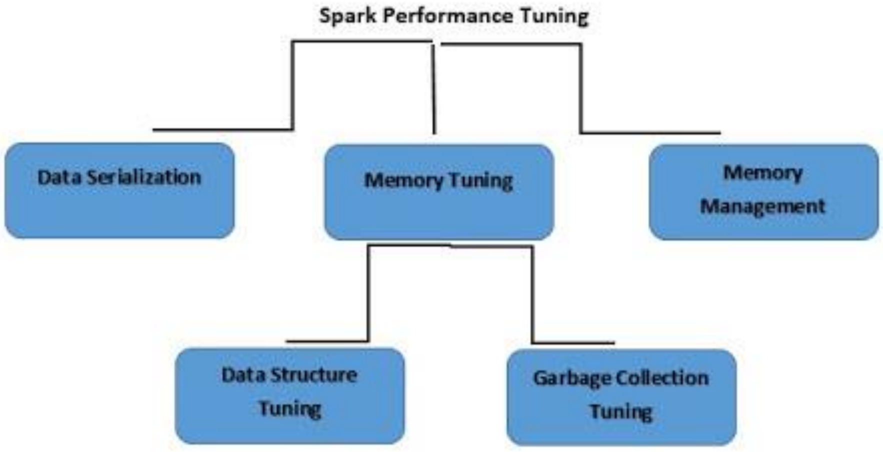


Figure 8. Performance tuning in Spark

Tuning is the system of modifying the machine's reminiscence, cores, and instance recording parameters. With the assist of this procedure, Spark performs at its excellent and aid constraints are prevented. based on system-specific parameters, all attributes and settings are efficiently changed to assure highest quality useful resource consumption. there is an in-memory computing issue to Apache Spark. As a end result, cluster resources just like the CPU and important memory might also turn out to be restricted.

To keep memory, RDDs are from time to time saved in a serialised manner. facts serialisation aids in memory intake discount, storage optimization, and desirable network overall performance.

Powerful tuning allows:

- Assures efficient and really apt usage of assets.
- cast off exhausting tasks.
- increase the system's speed.
- ensure the process is on the suitable execution engine.

Data Serialization

Converts an in-memory object to another format that can be saved to a file or sent over a network. It plays an important role in the performance of distributed applications. Slow computation due to formats that serialize slowly or consume large files. Apache Spark provides his two serialization libraries:

- Java Serialization
- Kryo Serialization

Java Serialization - Objects are serialized in Spark using the *ObjectOutputStream* framework and can be run in any class that implements *java.io.Serializable*. Serialization performance can be controlled with the *java.io.Externalizable* extension. It's flexible but slow, resulting in a large serialized format for many classes.

Kryo serialisation - Spark may utilise the Kryo library to serialise items (Version 2). It doesn't support every Serializable type, while being much tighter compared to Java serialisation. We must pre-register for the classes to get higher results. By using SparkConf to initialise our job and using *conf.set*("*spark.serializer*", "*org.apache.spark.serializer.KyroSerializer*"), we may change to Kryo.

For registering our class in Kryo, we employ the *registerKryoClasses* function. We are required to raise *spark.kryoserializer.buffer* configuration if our objects are huge. The value need to be substantial enough to accommodate the biggest item that we intend to serialise.

Memory Tuning

Keep in mind 3 things while optimizing memory usage:

Java items may be accessed, but devour two to five instances greater memory than raw information in fields. The reason for such behavior is:

- every unique Java object has an "object header". the size of this header is sixteen bytes. occasionally the object includes less data, so in such cases it is able to be large than the data.
- Java String raw string data has approximately 40 bytes of overhead. String uses UTF-16 encoding internally, so it stores each character as 2 bytes. a 10-character string can easily consume 60 bytes.
- common collection classes including HashMap and LinkedList use linked data structures. There you have got a "wrapper" object for each access. This object has each a header and a pointer (8 bytes each) to the following object inside the list.
- Collections of primitive types as "boxed objects". example: java.lang.Integer.

Data Structure Tuning

We can lessen memory intake by using warding off Java features that add overhead. There are numerous approaches to do this:

- avoid nested structures containing many small items and guidelines.
- Use numeric IDs or enum objects rather than strings for keys.
- in case your RAM size is less than 32 GB, set the JVM flag

–xx:+UseCompressedOops to create pointers to four bytes as opposed to eight bytes.

Garbage Collection Tuning

JVM's process of garbage collection becomes an issue when huge churn RDDs get stored programmatically. Java deletes old objects to make new ones. Track all obsolete objects and locate unused ones. But the point is garbage collection's price in Spark is similar to the amount of java objects. So for smaller objects he recommends using Spark's data structures. Another way to achieve this is storing the object in serialized format. So there is only 1 object/RDD partition.

Memory Management Tuning

Spark's memory management is segregated into 2 categories: storage and execution. "Execution memory" as the name suggests is utilised for computing in joins, shuffles, and aggregates. Storage is utilised for internal data propagation and in-cluster caching. A single area M is shared by storage and execution. The storage can utilise all of the memory while the execution memory is not in use. For storage memory, the same is true. If required, execution can exhaust the store. This is only done up until a particular threshold R is reached for storage memory consumption.

With this design, we may obtain numerous qualities. First off, if caching is not used, the programme can ustilise the whole available storage for execution. Data blocks in applications that employ caching will cause it to reserve a limited storage space, i.e. R, that is impervious to eviction.

Although we have two pertinent configurations, consumers don't need to change them. Since default values apply to the majority of workloads:

- M's size is described by *memory.fraction* as a percentage of (JVM's heap space-300MB) (default 0.6). The remaining 40% is kept in user's data structure, Spark internal metadata, and OOM error protection in the event of small and huge records.
- R is displayed as a part/fraction of M via *memory.storageFraction* (default 0.5).

Garbage Collection tuning in Spark

Step one in tweaking Apache Spark's garbage collection is to bring together data on how often trash collection takes place. additionally, it tracks the length of waste pickup time. that allows you to accomplish this, use the Java option -verbose:gc -XX:+PrintGCDetails -XX:+PrintGCTimeStamps. whilst the Spark task runs again, a notification will seem within the people log whenever trash collection takes place. not in the drivers programme, however inside the worker node, are these logs.

The Java's heap space is cut up into areas. old and young. younger generations own transient items, whereas older generations own long lasting items. The elder generation's lengthy-lived RDDs are the target of the trash collection adjustment. It additionally intends to be huge enough to preserve the assets of a youthful generation. This permits us to collect temporary objects produced in the course of task execution while not having to do a complete trash collection. the following actions ought to facilitate attaining this:

- There might not be sufficient memory to complete the work if full trash collection is called repeatedly earlier than it's far finished.
- we can decrease the amount of RAM wanted for caching if OldGen is nearly full in line with garbage collection information. this can be accomplished via decreasing spark.memory.fraction; although, caching fewer objects is ideal than delaying task completion. Alternately, we will

reduce -Xmn by way of lowering the dimensions of the younger generation.

Relying on our utility and the quantity of RAM consumed, Apache Spark garbage collection settings may also have exclusive outcomes.

Factors apart form the ones mentioned that can be used to improve performance:

a. Parallelism Level

Each application's stage of parallelism desires to be excessive enough to utilise the whole cluster. Spark determines the quantity of "Map" tasks to execute on every document based totally on the size of the document. A 2nd reason can be the amount of parallelism. To alter the default, we might also adjust the configuration setting *spark.default.parallelism*.

b. Memory Usage of Spark's Reduce Task

Despite the fact that RDDs match in our RAM, we often run into the *OutOfMemoryError* problem. this is because of our project *groupByKey*'s running set being very huge. this may be fixed with the aid of growing parallelism such that the input set for each system is condensed. due to the fact that Spark employs a unmarried executor JVM for a selection of activities and has a reasonably-priced task release cost, we can also enlarge the wide variety of cores in our cluster.

c. Using large variables in broadcasts

Using the broadcast capability in *SparkContext* decreases the size of every serialised job. flip a big item from the motive force programme utilized by a task into a printed variable. It commonly evaluates activities which are 20 Kb or much less for optimization.

d. Locality of Data in Apache Spark

Data locality has a considerable effect on how well Spark Jobs execute. The computation is quicker while the facts and the code that manipulates the records are gift. however, if the two are awesome, both the code or the records must be transferred. Because serialised code is smaller than a block of information, it can be sent from one place to every other extra fast.

There are numerous ranges of locality based totally on facts current area. From closest to furthest, on this order:

- The process local setting need to be in the same JVM as the executing code for the best locality.
- in this, NODE nearby is positioned at the equal node. this is therefore due to the fact method nearby is a whole lot faster at transferring records throughout processes.
- No pref facts does not have a geographical choice and is to be had international.
- Rack local records is positioned at the server's identical rack. facts is sent over the network over a unmarried transfer because it's far located at the equal rack however on a separate server.
- Any data that isn't in the very same rack is stored elsewhere inside the network.

1.4 Organization

- AWS
- Azure
- Apache Spark
- Python
- Oracle Java

CHAPTER 2: LITERATURE SURVEY

[1] Zahria, Reynolds, Xin and others

The expansion of data quantities in business and research creates both enormous potential and computing difficulties. Users required new technologies to scale out calculations to several nodes when data volumes grew beyond the capacity of a single computer. As a result, the amount of novel cluster programming models addressing various computing workloads has skyrocketed. For example, MapReduce allowed batch processing, but Google also created Dremel for interactive SQL queries and Pregel for iterative graph algorithms. At initially, these models were somewhat specialised, with new models produced for different workloads. Systems like Storm and Impala are additionally specialised inside the open-source Apache Hadoop stack. The tendency has been away from "one-size-fits-all" systems, even in relational databases.

Unfortunately, the majority of large-data applications necessitate combining many processing modalities. A general pipeline includes MapReduce-like code for data loading, SQL-like queries, and iterative machine learning since "big data" by its very nature is varied and chaotic. Therefore, specialised engines can lead to complexity and inefficiency since users must integrate many systems, and some applications can never be described effectively in any engine.

The next gen of computer applications will require data processing (scalable), yet this often entails a complicated series of processing processes using several computing technologies. The Spark project offered a unified programming paradigm and engine for large data applications to make this process easier. Our experience demonstrates that a strategy like this can accommodate current workloads well and provide consumers with significant advantages.

Apache Spark, we believe, emphasises the value of composability in large data programming libraries and encourages the creation of more readily interoperable libraries.

[2] Basha and Ramachandra

Actually, it's a highly significant approach and a very demanding task to analyse and process the SPARK in a fast and economical manner. And as is customary, SPARK helps people quickly and easily grasp huge quantities of data through quick and easy visualisation. There are numerous programmes on the market for creating such things studied, although one of the better tools is "SPARK," which creates the data to be adopted for a repository and maintaining "BIGDATA" items. Numerous techniques were created and proposed for evaluating and enhancing 'SPARK' success (performance). The study focuses mostly on fine-tuning configuration parameter technique.

Despite the fact that 'SPARK' is affected by a series of parameters, a map that minimised the relevant parameters had an effective influence.

The major goal of this effort is to improve "SPARK's" overall performance by speeding up job execution. Time savings are achieved by adjusting a few of the factors related to map reduction. Understanding these parameters is crucial since there are several sorts of parameters with awkward (incorrect) values that have a detrimental effect on performance as a whole. In this research, we offer a strategy that reduces task execution time and accurately and effectively optimises disc utilisation. In a heterogeneous environment, it precisely increases the overall functioning of Spark by 38.53% over the base system.

For this project, the base system configuration parameter of Apache SPARK was used to analyse and examine Twitter data. The study demonstrates changing the MapReduce job's parameter settings. Configuration parameters must be adjusted according to particular application until the waiting resource is completely utilised to achieve better results. Administrators of SPARK should use caution while choosing and changing the parameter's values. Since carelessness causes performance to decline. As a result, the paper-research suggests the "tuning approach," which improves overall performance by 38.5% over SPARK framework's default setup. At this time, SPARK has hundreds of different parameters.

[3] Dunner, Parnell, Atasu, Sifalakis and Pozidis

This paper investigates Apache Spark's performance bounds for ML applications. To start, examine the features of a cutting-edge disbursed ML algorithms that are built inside Spark and contrast it with a reference of implementation using MPI, a high performance/efficiency computing environment, that is similar. Paper pinpoints the Spark framework's most important bottlenecks and closely examine how they affect the algorithm's performance. Paper then suggests a variety of doable methods to reduce some of Spark's overheads to enhance performance.

It is demonstrated that, to achieve the greatest performance from any implementation, thorough algorithm tuning is required to account for the trade-off between calculation communication and time delay. Performance is not just dependent on maximising computational effectiveness and framework-related overheads. The ideal trade-off depends on the characteristics of the distributed algorithm as well as the infrastructure and framework. Finally, we use these technological and algorithmic advancements to 3 distinct disbursed linear ML algorithms that are built into Spark. We discuss our findings and demonstrate how adopting the proposed improvements may lower the performance gap between Spark and MPI from 20 to 2 times with the aid of five significant datasets.

In this study, it is shown that compared to similar MPI implementations, vanilla Spark implementations of distributed ML can exhibit performance losses of more than an order of magnitude. Language-dependent overheads are responsible for a significant portion of this loss.

It is demonstrated a reduction in this gap with MPI to only 2 after removing these overheads by offloading crucial calculations into C++, combining this with a number of useful enhancements to Spark, and effectively tweaking the method. We come to the conclusion that improving the computational efficiency of the implementation is insufficient for creating high-performance, disbursed ML applications in Spark and other distributed computing frameworks.

The algorithm must be carefully modified to take into account the tendency of the particular system on which such an application will be used. Algorithms with a tuning parameter that the user may utilise to adjust to changes in system-level conditions are therefore quite interesting from a research standpoint.

[4] Ahmed, Barczak, Susnjak and Rashid

Massive-scale dataset garage, processing, and analysis the usage of massive data analytics has come to be a crucial tool for the sector. New allotted computing frameworks like Spark and Hadoop provide powerful methods to take a look at large volumes of information. Spark profits a number of popularity because of the availability of its application programming interface (API) and its overall performance, surpassing the MapReduce framework in reputation. The combination of the extra than a hundred and fifty parameters in every of those frameworks has a sizable impact on cluster overall performance. The system administrator might also easily set up their system applications thanks to the preset machine settings, and they can use factory-set parameters to gauge the overall performance in their particular cluster.

The use of a cluster that has been set up in our lab, this research compares the performance of Spark and Hadoop with the aid of inspecting the maximum vital input splits, resource use, and shuffle settings. tweaking those settings through a huge number of tests the usage of a trial-and-errors method. WordCount and TeraSort were selected as the 2 workloads to be evaluated for you to examine the comparative analysis frameworks. Execution time, throughput, and speedup are the 3 elements used to calculate performance measures. Our experimental findings confirmed that the proper parameter selection and enter data size had a big effect on both system performances.

Whilst default parameter values are modified, the examination of the effects reveals that Spark plays higher than Hadoop for small statistics units, accelerating WordCount workloads by way of up to 2 instances and TeraSort workloads by using up to 14 instances.

[5] Aziz, Zaidouni and Bellafkih

One of the famous open-source big-data processing frameworks is Spark, which enables the concurrent processing of widespread datasets utilising a tremendous quantity of machines. applications of this framework frequently employ resource control tools like YARN, which allocate tasks a certain wide variety of resources for execution. The data that the framework will look at is likewise saved in a allotted report machine like HDFS. by way of executing jobs on a single-node cluster or multi-node cluster architecture, this technique permits green sharing of cluster resources. therefore, one tough task is to implement efficient resource management of these large cluster infrastructures in an effort to execute disbursed data analytics in a manner this is both sensible and less expensive.

In this research, we develop several ML algorithms the usage of the MLlib, after which we manage the sources (CPU, memory, and disc) to assess Apache Spark's performance. The assessment of severa studies that target useful resource management and records processing in huge facts platforms is offered in this study. moreover, we use Spark to do a scalability look at. Paper examines processing times and speedups. it's far concluded that after the cluster reaches a selected size, including greater nodes is now not required to growth overall performance and processing time.

The study then looks at Spark's resource allocation adjustments. it's been tested that enhancing performance depends on a way to optimise resource allocation instead of simply assigning all of the to be had assets. The paper suggests additional controlled parameters and demonstrates that they offer quicker ordinary processing times than Spark's default parameters. sooner or later, use system mastering techniques to analyze the patience of resilient dispensed datasets (RDDs) in Spark. One storage stage stands proud many of the others that have been tested for execution pace.

[6] Salloum, Dautov, Chen, Peng and Huang

With its cutting edge in-memory programming structure and upper-stage libraries for scalable gadget getting to know, graph analysis, streaming, and dependent records processing, Spark has come to be the de facto framework for huge statistics analytics. it's miles a widespread-purpose cluster computing framework featuring Scala, Java, Python, and R language-included APIs. it is able to be tough for teachers, particularly people who are new to this discipline, to recognize the whole body of work and research behind Spark due to the fact it is a speedy growing open supply project with a rising variety of contributors from each academia and business. this text offers a technical evaluation of Spark-primarily based massive information analytics. the primary features, abstractions, Sparks's elements are the subject of this text.

In further element, it demonstrates the talents of Spark for growing and deploying big data pipelines and algorithms for machine learning, graph evaluation, and flow processing. The file additionally discusses potential future regions for Spark research and improvement for big records analytics.

[7] Gupta, Sharma and Jindal

The standard processing framework for big data analytics is Spark. The key difficulties with big data analytics include managing a wide range of types, storing enormous amounts of data, and processing data quickly. Because Spark processes data in-memory, it has lots of benefits over MapReduce. The default settings for running Spark applications are made on commodity hardware, therefore they might not offer a solution that works for every setup and environment. To obtain the best performance, resource allocation for Spark applications must be tuned. In order to improve the speed of Spark applications, this article addresses a variety of settings and choices, including caching, broadcast variables, repartitioning, and the number of executors.

[8] Nguyen, Khan and Wang

Many businesses have chosen Apache Spark, a recently popularised data analytics platform.

Spark offers a wide array of configuration options that may be modified to enhance the performance of a particular application since the features of various Spark applications frequently differ greatly in terms of resource requirements and execution flow. Although some recent initiatives examined the issue of configuration tuning in the context of Apache Spark, it is challenging to adjust them automatically because to the vast number of options (which is typical for large-scale cloud systems). Additionally, tuning efforts must take into account combinations of settings since they are frequently connected to performance and may conflict. This is done to prevent inefficient configuration and/or potential configuration errors.

The paper examines machine learning-based algorithms that may automatically search and discover the set of recommended changes that may considerably increase performance compared to the default settings in order to automate the configuration tweaking process. Specifically, the paper employs Latin hypercube design technique to first select a set of configurations that are used to benchmark the system and gather training data for a specified number of parameters that may impact performance (which are recognised a priori). Then train several machine learning models and then choose the best one based on prediction accuracy. In the study, paper takes into account three distinct machine learning techniques—Artificial Neural Networks, Support Vector Regression, and Decision Trees—to build performance models for each application.

The most efficient ML model found in the previous stage is then used to fine-tune the configuration parameters for each application using the Recursive Random Search technique.

The study evaluated nine distinct apps, representing three different application categories, to test the framework because the same parameter may effect the performance of various applications differently. In particular, we used PageRank, Triangle Count, and Connected Components as representative of graph processing algorithms, as well as Word Count and Tera Sort as representative of batch processing applications, KMeans, Support Vector Machines, Matrix Factorization, and Decision Trees as representative of machine learning algorithms. In each instance, the article assesses the construct models' correctness and the performance enhancement brought on by configuration adjustment. According to the evaluation, our framework may greatly boost performance, with the improvement varying depending on the application from 22.8% to 40.0%.

[9] Schiavio, Bonetta and Binder

The adoption of big-data platforms has significantly increased, and Apache Spark is quickly becoming as the industry standard for contemporary data analytics. To improve the execution efficiency of analytical tasks on a range of data sources, Spark depends on SQL query compilation. Spark's SQL code generation has severe runtime overheads due to data access and de-serialization in spite of its scalable design. When applications use human-readable data formats like CSV or JSON, such a performance cost might be severe.

This paper gives a novel query compilation method that relies on run-time profiling and dynamic code creation to get around these restrictions. With textual-form data formats like JSON or CSV, Spark's new SQL compiler creates very efficient machine code, resulting in high speeds of up to 4.4 times on the TPC-H benchmark.

[10] Essertel, Tahboub, Decker, Brown, Olukotun and Rompf

Spark has recently taken over because the industry trendy for massive statistics processing. due to its adaptability and ease, Spark has allowed a huge number of customers to procedure petabyte-scale workloads: users can combine relational queries within the fashion of square with Scala or Python code, and the resulting programmes can be allotted across an entire cluster without the want to paintings with low-stage parallelization or community primitives.

but, a whole lot of workloads with real-international importance are not massive sufficient to warrant dispensed, scale-out execution because the information may be thoroughly contained in a unmarried, effective server's foremost memory. Spark remains favored by users due to its well known equipment and consumer interface.

because of Spark's preference for dealing with facts size over improving the computations on that information, its overall performance is subpar in positive scale-up situations. overall performance might also nonetheless be crucial for such medium-sized workloads if responsibilities want quite a few processing, must be repeated frequently on changing records, or have interaction with outside libraries and systems (e.g., TensorFlow for machine learning).

The paper introduces Flare, an accelerator Spark module that significantly hastens a huge variety of applications on scale-up systems. Flare carries a code creation method created to in shape the one of a kind functions of Spark and the characteristics of scale-up architectures, mainly processing data without delay from optimised record codecs and mixing square-style relational processing with outside facts assets. Flare became stimulated via query compilation methods from main-memory database systems.

CHAPTER 3: SYSTEM DEVELOPMENT

3.1 Analytical

Wide-spread use of MapReduce and huge-scale computing has caused the emergence of numerous cluster computing systems. these systems use various new APIs, often based totally on useful programming, to support both relational queries and greater complicated sorts of processing along with extraction, transformation, loading operations or machine getting to know.

Of these systems, Spark has come to be the maximum used.

This is because of our knowledge of over 500 deployments and the most lively contributor community on Apache (over 400 participants in 2014). not like preceding committed structures, Spark provides a fashionable-motive engine primarily based on undertaking DAGs and data sharing, able to going for walks workloads like batch jobs, streaming, square and chart analytics. There are APIs for Scala, Python, Java, and R. As Spark moved from early adopters to a broader audience, the opportunity to peer in which the practical API truly worked, wherein it is able to be progressed, and what new customers want.

In general, assisting not unusual analytics workloads has been a fulfillment for Spark. moreover, it employs ml, sql, streaming, graph processing, and streaming libraries, regularly with overall performance on par with specialized engines. due to the fact maximum customers blend more than one of those types of processing of their workloads, the Spark engine's adaptability is essential.

Nevertheless, given the diversity of supported data types and calculations, Spark's functional API posed several difficulties for both users and systems. The most typical difficulties are:

- **Functional semantics of an API.** The foundation of the Spark API is a set of Java/Python object collections, on which users may call any Python or Java function using operators like map or groupBy. We discovered that users frequently struggled to choose the ideal functional operators for a particular calculation. Using Spark's groupByKey operator, which produces a distributed collection of (key, list of value) pairs, as an example, and then aggregating each list is a typical issue (e.g., a sum). The reduceByKey operator in Spark may execute partial aggregation on each node, which would make this calculation significantly quicker. The groupByKey operator must transmit each list of records to one machine since that is its return signature.

It is also challenging for the engine to automatically detect and replace operators since the functions provided to Spark are random pieces of Python or Java code. Static analysis of UDFs has been suggested in some research, although this analysis can be fragile for sophisticated object-oriented applications.

- Despite Spark's facet-impact-loose, functional API, disbursed programmes are inherently difficult to debug due to the fact that customers ought to consider undertaking distribution and skew. according to our studies, performance debugging provides the most tough troubles considering that customers often are unaware that their activity is concentrated on a small wide variety of computers or that some of their records systems are reminiscence-inefficient.

- **Memory Control.** because "huge facts" can take many distinct styles and sizes, the engine must cautiously manage its memory. We found other resources of excessive memory usage notwithstanding the reality that external tactics for aggregation and joins are nicely regarded. as an example, sure programmes records information (such those used for photograph processing) is probably masses of gigabytes each, necessitating meticulous tracking as each file is read. any other example is that Spark first of all notion each block of the record, in HDFS is normally 128 MB, should shop all of its records in memory without delay. but, for some closely compressed datasets, each block might also decompress into 3–4 GB.

- **A significant IO.** With biggest clusters now having over eight thousand nodes and distinct tasks controlling over 1 PB, Spark workloads multiplied dramatically. The networking and that i/O layers of Spark have received high-quality engineering funding so that it will characteristic nicely at this size.

- **Non-experts' access.** Earlier cluster computing answers, like MapReduce, have been created with software developers in thoughts, however maximum agencies require "big information" to be to be had to a huge variety of people, together with non-developers with domain understanding (together with statisticians or information scientists). Better-level APIs are crucial for all customers as well on account that a whole lot of facts evaluation is exploratory and clients lack the time to create absolutely optimised distributed programmes. We have made a giant effort to offer excessive-level data-technological know-how API that mirror unmarried-node equipment/utilities, like R's records frames throughout Spark, so that it will solve those problems.

3.1.1 Constraints and Assumptions

3.1.1.1 Constraints

A large cluster couldn't be used cause of the lack of funding for heavy processing of data as such local based machines and virtual machines over which the workers were distributed were used

- Heavy techniques can't be used considering the processing power of client machines and the page load time of the website.
- Constrained ourself on the size of the data so that we don't have to wait for the long runtime of days for data preprocessing.

3.1.1.2 Assumptions

- There are enough resources needed to run spark and it's virtual environment over long period of time.
- The user has a basic knowledge about SQL, Python /R/Java/Scala.

3.1.1.3 Use case diagram

A diagram of a person's ability interactions with a device is called a use-case diagram. A use-case diagram, that's often complemented through other kinds of diagrams, presentations the several use instances and consumer kinds the device has.

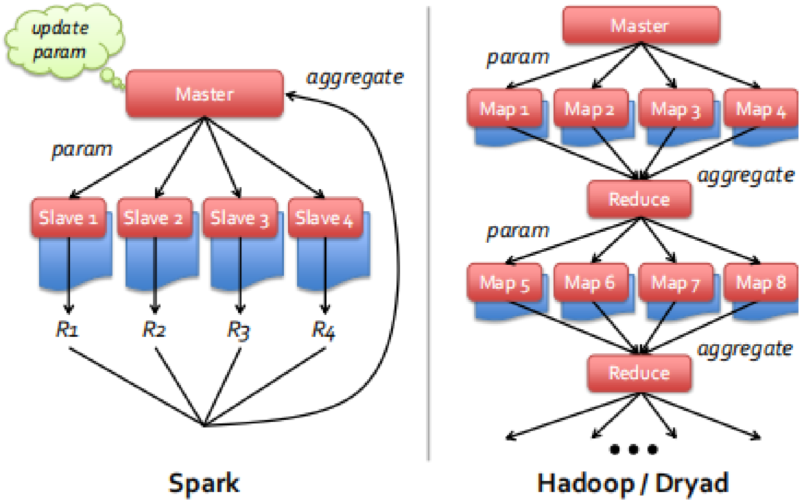


Figure 9. Spark vs Hadoop

Figure 9 shows the wroker heirarchy of spark versus hadoop and *Figure 10 and 11* detail the architectures of hadoop and spark respectively.

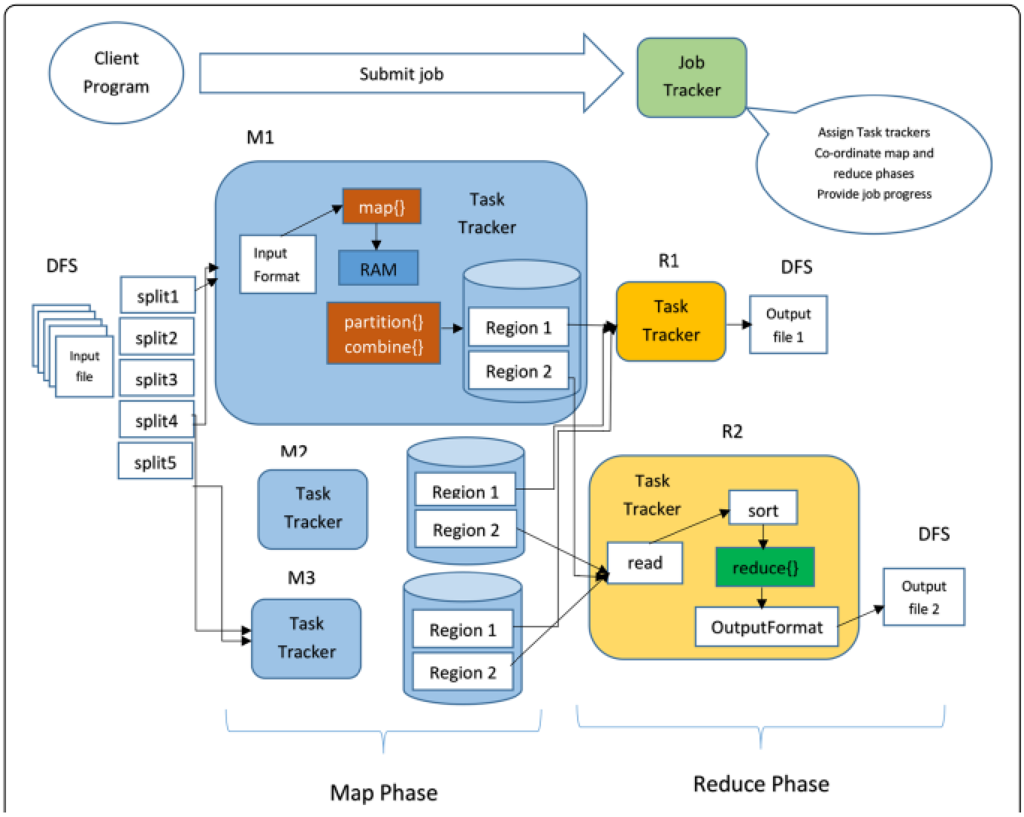


Figure 10. Hadoop based architecture

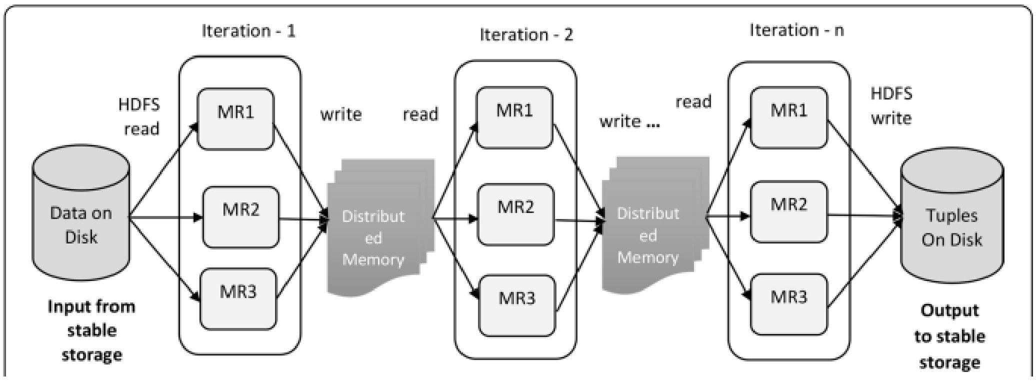


Figure 11. Spark based work architecture

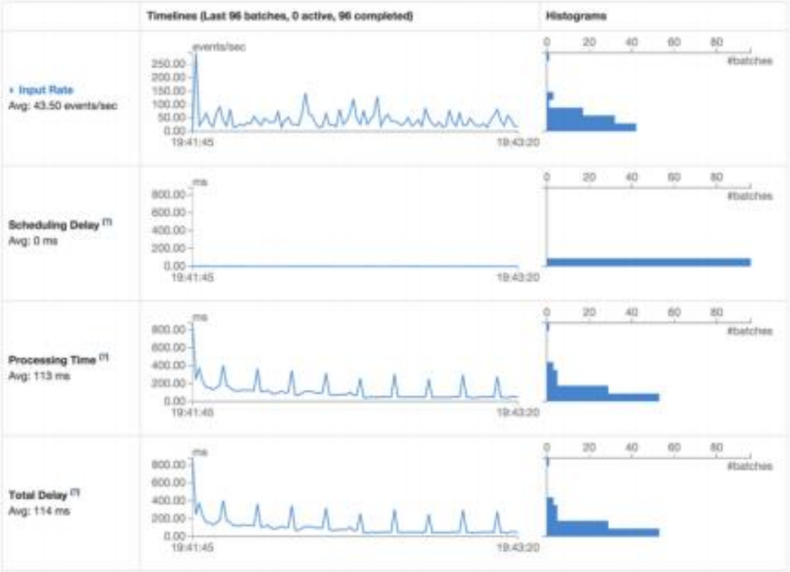


Figure 12. Metrics dashboard for Spark streaming

3.2 Improvements to The Execution Engine

Memory management and the networking layer make up the bulk of our efforts in this area. Both emphasise improving the engine's robustness and performance under heavy workloads.

3.2.1 Memory Management

We investigated the root causes of memory issues based on user feedback in order to enhance memory management, and we created a per-node allocator that controls all sources of memory consumption inside each node. When a limit was reached, the memory management for Spark would evict any remaining old data blocks, keeping track of how much "cached" data the user had chosen to materialise in memory. The first manager didn't specifically monitor how much RAM was being consumed for data processing. As a result, processing big joins or aggregations was a significant contributor to the memory fatigue issues. We introduced a second cap to monitor hash tables for joins and aggregates to handle this.

As the threads doing these operations increase their tables, this cap is dynamically distributed among them, and threads that are not permitted to consume additional RAM spill to disc. To determine whether the uncompressed data is still tiny enough to store, a third area was set aside for "unrolling" blocks that are read from disc. To accommodate skewed record sizes in each of these scenarios, we monitor memory use every 16 records. The engine functions reliably under these conditions under a variety of workloads.

3.2.2 Network Layer

Networking layer's largest assignment became helping shuffle operations on many nodes. Shuffle operations need to transport output information from map obligations to lessen tasks across the whole network, in order that each node is sending some statistics to each different node. they're a undertaking to enforce because every node might be fetching facts from diverse disks, multiple connections are typically required to saturate community bandwidth, and care ought to be taken to stability load. We formerly wrote a custom community module that was primarily based on Java's NIO. The module used the low-stage Java NIO networking API without delay and had to maintain complicated nation machines internally. further to this, it creates a higher memory strain from JVM's garbage series and higher CPU usage than wanted because of useless copies of community buffers.

We created a newer implementation of network module for Apache Spark [1] based at the excessive-overall performance networking framework Netty [2]. Netty gives a better degree asynchronous event-pushed abstraction that makes networking programming simpler. On top of Netty, we delivered a selection of features to enhance performance and scalability, consisting of:

- **No-copy I/O:** Tell the kernel to bypass user-space memory and transfer data from on-disk files directly to the socket. This lessens the demand on the JVM heap's memory as well as the amount of CPU time used for context transitions between kernel and user space.

- **Off-heap network buffer management:** Netty directly manages a pool of memory pages outside the Java heap, removing network buffers' negative effects on the JVM garbage collector.
- **Multiple connections:** To maximise the throughput of data fetches and distribute load evenly across the nodes providing data, each Spark worker node has multiple concurrently active connections (by default, 5). 200 machines connected by 10 Gbps lines may fully utilise a network with a bisectional bandwidth. We utilised it to beat the previous Hadoop-based records in the Daytona-GraySort competition [3] by sorting 100 TBs of on-disk data with 10% fewer workstations (Figure 10).

3.3 Mathematical

The amount of efficiency used here is time per node per broadcast used for a singular job.

In the events of the overlaying structure it was found efficiency(ε)

Higher number of nodes in a cluster greater the performance of cumulative jobs.

HDFS [4] divides documents into small chunks of blocks and stores them on different nodes. There are two sorts of nodes in HDFS: information nodes (employees) and call nodes (grasp nodes). All operations along with delete, study and write are based totally on those two kinds of nodes. The HDFS workflow is:

First, the namenode requests permission. If generic, convert the document name to list HDFS block IDs.

This includes files and facts nodes that shop blocks related to this document. list of IDs is then back to the customer on which the consumer can carry out similarly operations.

MapReduce [5] is a computing framework containing her operations of mappers and reducers. The mapper methods the documents primarily based upon the map's function and maps them to new key-value pairs.

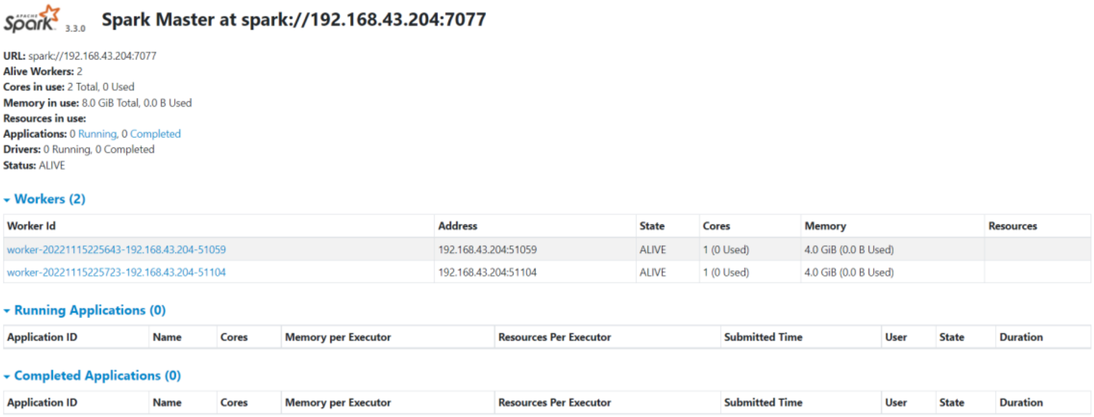
New key value pairs are then assigned to special partitions and sorted based on those keys. The combiner 's elective and can be visible as a nearby cut back operation, in order that key can pre-matter values to lessen I/O pressure. sooner or later, the partition splits the in-between key-price pair into various portions and sends them to reducer.

MapReduce takes to use the shuffle operation. Shufen means forwarding the mapper output records to correct reducer. Once the shuffle completes, the reducer starts some reproduction- threads (fetchers) to retrieve the output files of the map's challenge thru HTTP. The subsequent step is to merge the outputs into diverse documents. These documents are treated as reducer input facts.

CHAPTER 4: PERFORMANCE ANALYSIS

Distributing Spark jobs over a set of machines can significantly affect its execution times. *Figure 13* shows the base master view of a example spark job run on 2 workers (see *figure 16, 17, 18*).

Note: These executions were trial runs, just to check, determine and prove that optimization can and will improve the execution times and execution time can be a great predictor for efficiency of a spark job.



Spark Master at spark://192.168.43.204:7077

URL: spark://192.168.43.204:7077
Alive Workers: 2
Cores in use: 2 Total, 0 Used
Memory in use: 8.0 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 0 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20221115225643-192.168.43.204-51059	192.168.43.204:51059	ALIVE	1 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20221115225723-192.168.43.204-51104	192.168.43.204:51104	ALIVE	1 (0 Used)	4.0 GiB (0.0 B Used)	

Running Applications (0)

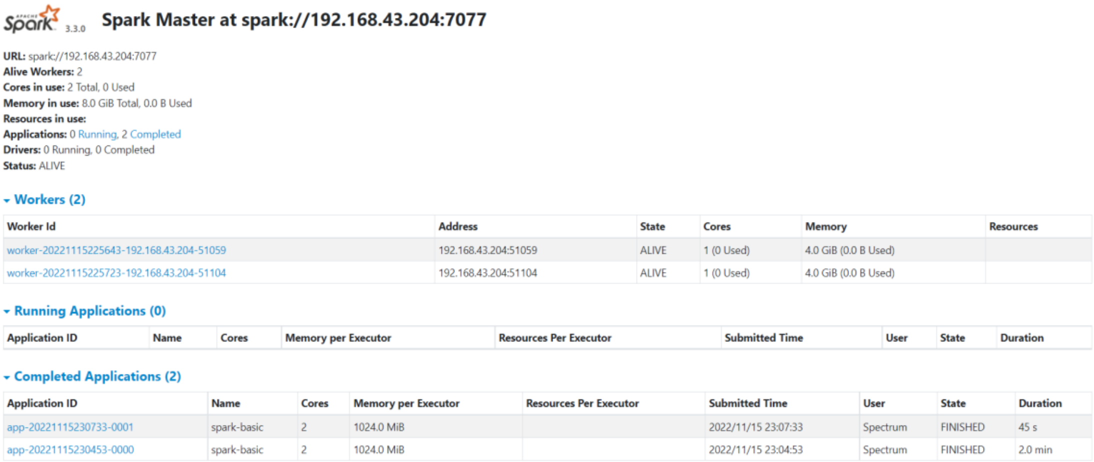
Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Figure 13. Base Master View

Figure 14 shows the same job running on same workers but with cache turned on, reducing the execution time from 2 minutes to just 45 seconds.



Spark Master at spark://192.168.43.204:7077

URL: spark://192.168.43.204:7077
Alive Workers: 2
Cores in use: 2 Total, 0 Used
Memory in use: 8.0 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 2 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20221115225643-192.168.43.204-51059	192.168.43.204:51059	ALIVE	1 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20221115225723-192.168.43.204-51104	192.168.43.204:51104	ALIVE	1 (0 Used)	4.0 GiB (0.0 B Used)	

Running Applications (0)

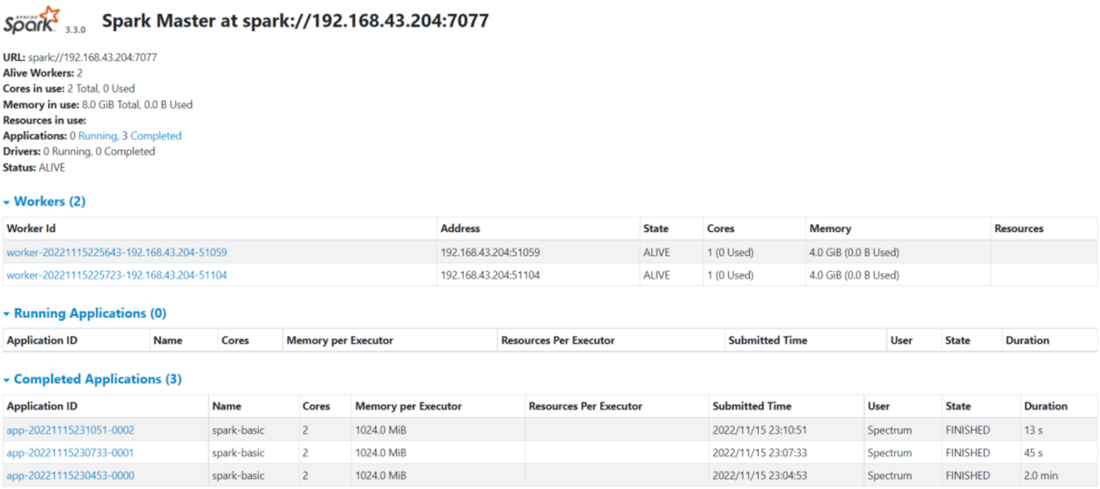
Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (2)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20221115230733-0001	spark-basic	2	1024.0 MiB		2022/11/15 23:07:33	Spectrum	FINISHED	45 s
app-20221115230453-0000	spark-basic	2	1024.0 MiB		2022/11/15 23:04:53	Spectrum	FINISHED	2.0 min

Figure 14. Job running with cache

Figure 15 shows the same job running but with persist turned on. The execution time is 13 seconds.



3.3.0 Spark Master at spark://192.168.43.204:7077

URL: spark://192.168.43.204:7077
Alive Workers: 2
Cores in use: 2 Total, 0 Used
Memory in use: 8.0 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 3 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20221115225643-192.168.43.204-51059	192.168.43.204:51059	ALIVE	1 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20221115225723-192.168.43.204-51104	192.168.43.204:51104	ALIVE	1 (0 Used)	4.0 GiB (0.0 B Used)	

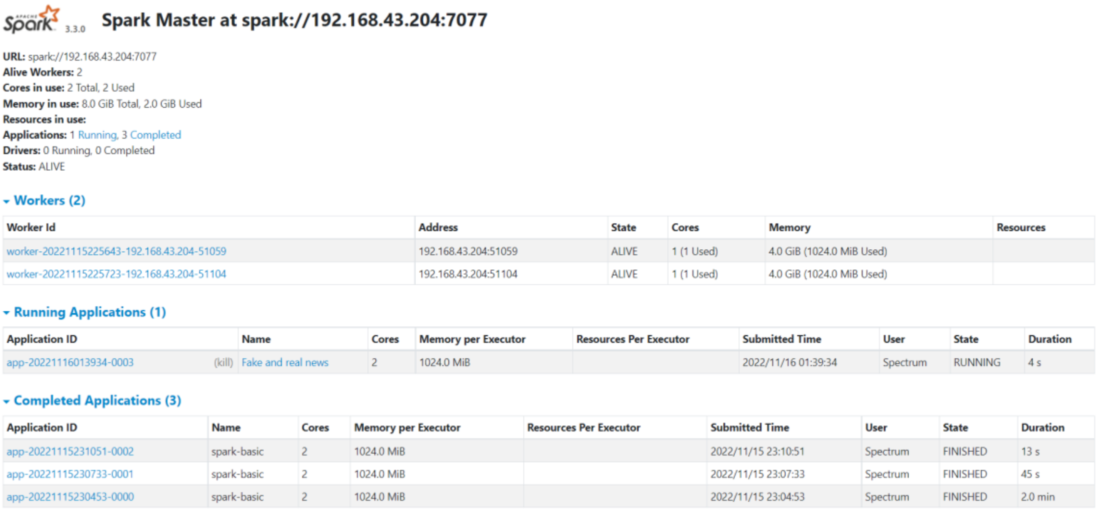
Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (3)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20221115231051-0002	spark-basic	2	1024.0 MiB		2022/11/15 23:10:51	Spectrum	FINISHED	13 s
app-20221115230733-0001	spark-basic	2	1024.0 MiB		2022/11/15 23:07:33	Spectrum	FINISHED	45 s
app-20221115230453-0000	spark-basic	2	1024.0 MiB		2022/11/15 23:04:53	Spectrum	FINISHED	2.0 min

Figure 15. Job running with persist



3.3.0 Spark Master at spark://192.168.43.204:7077

URL: spark://192.168.43.204:7077
Alive Workers: 2
Cores in use: 2 Total, 2 Used
Memory in use: 8.0 GiB Total, 2.0 GiB Used
Resources in use:
Applications: 1 Running, 3 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20221115225643-192.168.43.204-51059	192.168.43.204:51059	ALIVE	1 (1 Used)	4.0 GiB (1024.0 MiB Used)	
worker-20221115225723-192.168.43.204-51104	192.168.43.204:51104	ALIVE	1 (1 Used)	4.0 GiB (1024.0 MiB Used)	

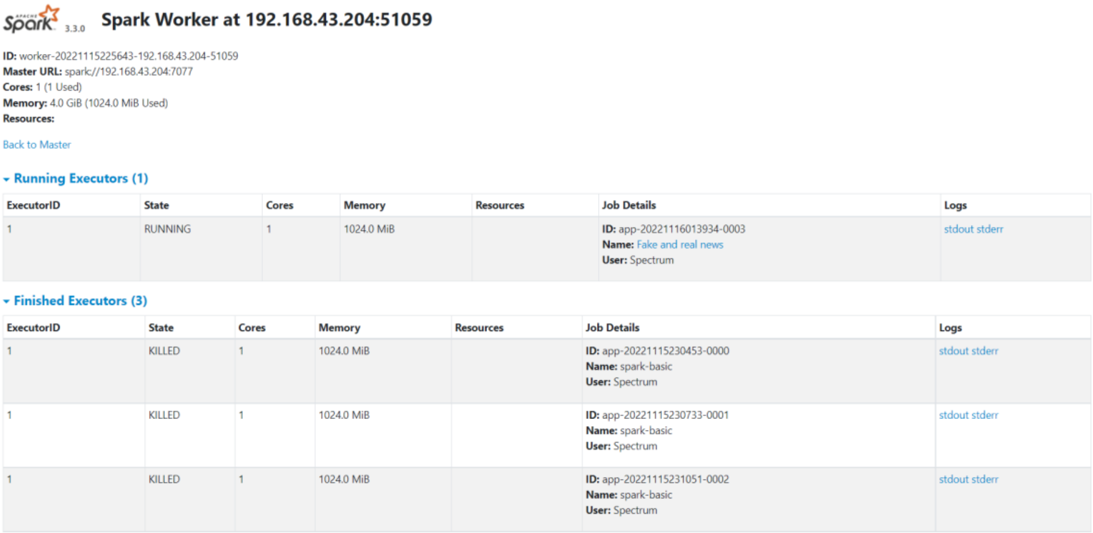
Running Applications (1)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20221116013934-0003 (kill)	Fake and real news	2	1024.0 MiB		2022/11/16 01:39:34	Spectrum	RUNNING	4 s

Completed Applications (3)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20221115231051-0002	spark-basic	2	1024.0 MiB		2022/11/15 23:10:51	Spectrum	FINISHED	13 s
app-20221115230733-0001	spark-basic	2	1024.0 MiB		2022/11/15 23:07:33	Spectrum	FINISHED	45 s
app-20221115230453-0000	spark-basic	2	1024.0 MiB		2022/11/15 23:04:53	Spectrum	FINISHED	2.0 min

Figure 16. Distributing job across workers



3.3.0 Spark Worker at 192.168.43.204:51059

ID: worker-20221115225643-192.168.43.204-51059
Master URL: spark://192.168.43.204:7077
Cores: 1 (1 Used)
Memory: 4.0 GiB (1024.0 MiB Used)
Resources:

Back to Master

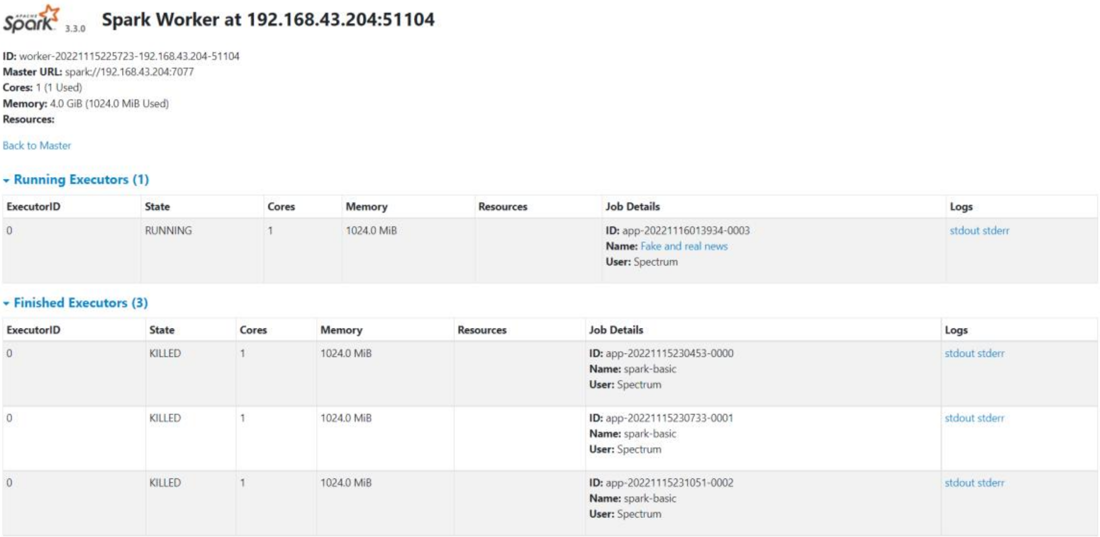
▾ Running Executors (1)

ExecutorID	State	Cores	Memory	Resources	Job Details	Logs
1	RUNNING	1	1024.0 MiB		**ID:** app-20221116013934-0003 **Name:** Fake and real news **User:** Spectrum	stdout stderr

▾ Finished Executors (3)

ExecutorID	State	Cores	Memory	Resources	Job Details	Logs
1	KILLED	1	1024.0 MiB		**ID:** app-20221115230453-0000 **Name:** spark-basic **User:** Spectrum	stdout stderr
1	KILLED	1	1024.0 MiB		**ID:** app-20221115230733-0001 **Name:** spark-basic **User:** Spectrum	stdout stderr
1	KILLED	1	1024.0 MiB		**ID:** app-20221115231051-0002 **Name:** spark-basic **User:** Spectrum	stdout stderr

Figure 17. 1st Slave View



3.3.0 Spark Worker at 192.168.43.204:51104

ID: worker-20221115225723-192.168.43.204-51104
Master URL: spark://192.168.43.204:7077
Cores: 1 (1 Used)
Memory: 4.0 GiB (1024.0 MiB Used)
Resources:

Back to Master

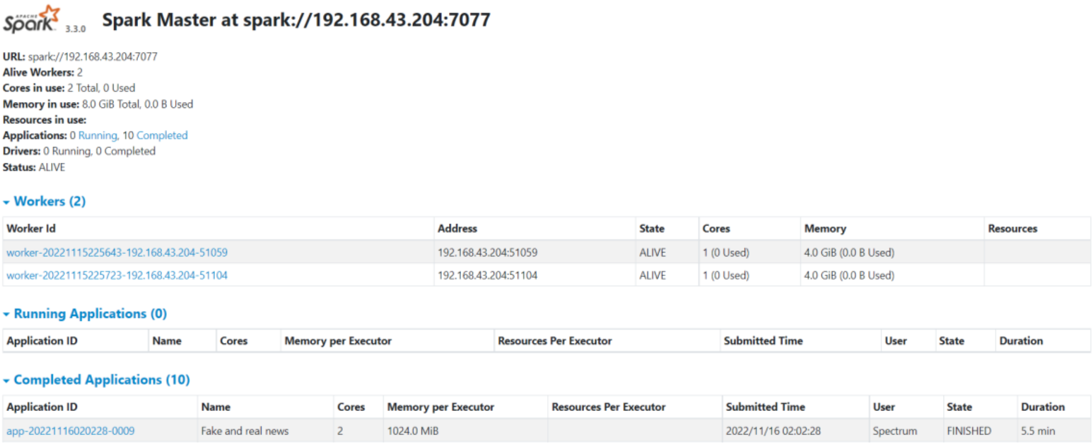
▾ Running Executors (1)

ExecutorID	State	Cores	Memory	Resources	Job Details	Logs
0	RUNNING	1	1024.0 MiB		**ID:** app-20221116013934-0003 **Name:** Fake and real news **User:** Spectrum	stdout stderr

▾ Finished Executors (3)

ExecutorID	State	Cores	Memory	Resources	Job Details	Logs
0	KILLED	1	1024.0 MiB		**ID:** app-20221115230453-0000 **Name:** spark-basic **User:** Spectrum	stdout stderr
0	KILLED	1	1024.0 MiB		**ID:** app-20221115230733-0001 **Name:** spark-basic **User:** Spectrum	stdout stderr
0	KILLED	1	1024.0 MiB		**ID:** app-20221115231051-0002 **Name:** spark-basic **User:** Spectrum	stdout stderr

Figure 18. 2nd Slave View



Spark 3.3.0 **Spark Master at spark://192.168.43.204:7077**

URL: spark://192.168.43.204:7077
Alive Workers: 2
Cores in use: 2 Total, 0 Used
Memory in use: 8.0 GiB Total, 0.0 B Used
Resources in use:
Applications: 0 Running, 10 Completed
Drivers: 0 Running, 0 Completed
Status: ALIVE

Workers (2)

Worker Id	Address	State	Cores	Memory	Resources
worker-20221115225643-192.168.43.204-51059	192.168.43.204:51059	ALIVE	1 (0 Used)	4.0 GiB (0.0 B Used)	
worker-20221115225723-192.168.43.204-51104	192.168.43.204:51104	ALIVE	1 (0 Used)	4.0 GiB (0.0 B Used)	

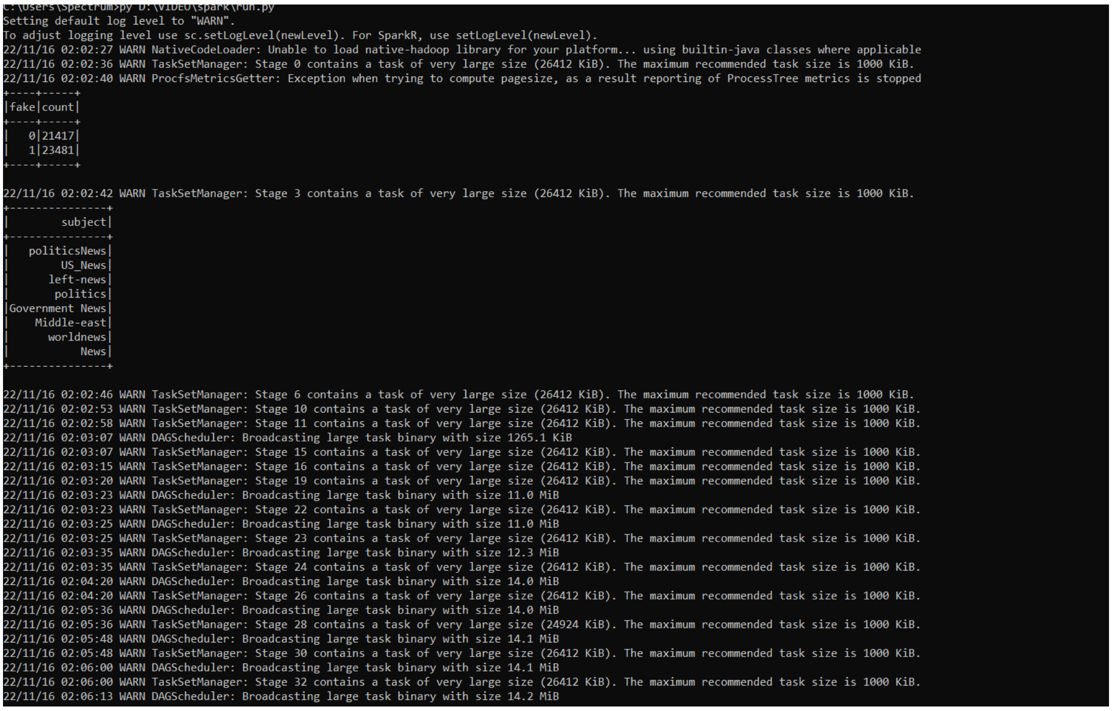
Running Applications (0)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration

Completed Applications (10)

Application ID	Name	Cores	Memory per Executor	Resources Per Executor	Submitted Time	User	State	Duration
app-20221116020228-0009	Fake and real news	2	1024.0 MiB		2022/11/16 02:02:28	Spectrum	FINISHED	5.5 min

Figure 19. Job ran over broadcasted over workers

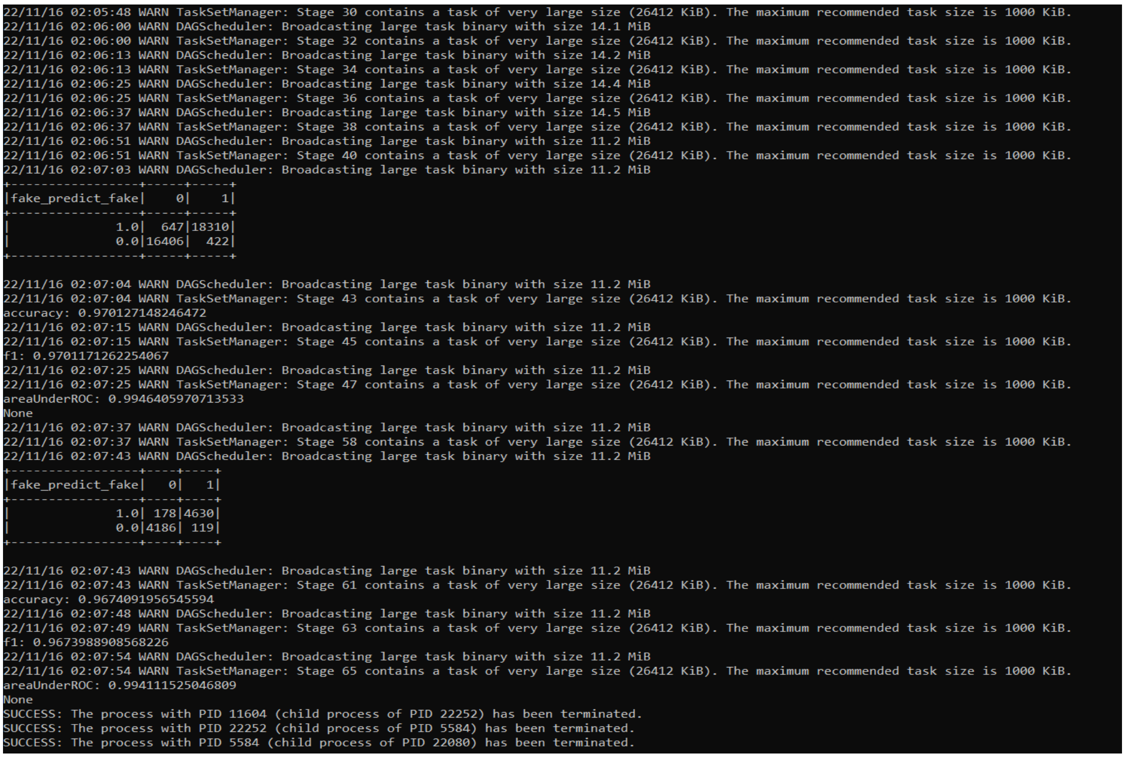


```
C:\Users\Spectrum>py D:\VIDEO\spark\run.py
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
22/11/16 02:02:27 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
22/11/16 02:02:36 WARN TaskSetManager: Stage 0 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:02:40 WARN ProcfsMetricsGetter: Exception when trying to compute pagesize, as a result reporting of ProcessTree metrics is stopped
+----+-----+
|fake|count|
+----+-----+
|   0|21417|
|   1|23481|
+----+-----+

22/11/16 02:02:42 WARN TaskSetManager: Stage 3 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
+---------------+
|        subject|
+---------------+
|   politicsNews|
|        US_News|
|      left-news|
|       politics|
|Government News|
|    Middle-east|
|      worldnews|
|           News|
+---------------+

22/11/16 02:02:46 WARN TaskSetManager: Stage 6 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:02:53 WARN TaskSetManager: Stage 10 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:02:58 WARN TaskSetManager: Stage 11 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:03:07 WARN DAGScheduler: Broadcasting large task binary with size 1265.1 KiB
22/11/16 02:03:07 WARN TaskSetManager: Stage 15 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:03:15 WARN TaskSetManager: Stage 16 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:03:20 WARN TaskSetManager: Stage 19 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:03:23 WARN DAGScheduler: Broadcasting large task binary with size 11.0 MiB
22/11/16 02:03:23 WARN TaskSetManager: Stage 22 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:03:25 WARN DAGScheduler: Broadcasting large task binary with size 11.0 MiB
22/11/16 02:03:25 WARN TaskSetManager: Stage 23 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:03:35 WARN DAGScheduler: Broadcasting large task binary with size 12.3 MiB
22/11/16 02:03:35 WARN TaskSetManager: Stage 24 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:04:20 WARN DAGScheduler: Broadcasting large task binary with size 14.0 MiB
22/11/16 02:04:20 WARN TaskSetManager: Stage 26 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:05:36 WARN DAGScheduler: Broadcasting large task binary with size 14.0 MiB
22/11/16 02:05:36 WARN TaskSetManager: Stage 28 contains a task of very large size (24924 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:05:48 WARN DAGScheduler: Broadcasting large task binary with size 14.1 MiB
22/11/16 02:05:48 WARN TaskSetManager: Stage 30 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:06:00 WARN DAGScheduler: Broadcasting large task binary with size 14.1 MiB
22/11/16 02:06:00 WARN TaskSetManager: Stage 32 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:06:13 WARN DAGScheduler: Broadcasting large task binary with size 14.2 MiB
```

Figure 20. Log view- Intial state



```
22/11/16 02:05:48 WARN TaskSetManager: Stage 30 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:06:00 WARN DAGScheduler: Broadcasting large task binary with size 14.1 MiB
22/11/16 02:06:00 WARN TaskSetManager: Stage 32 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:06:13 WARN DAGScheduler: Broadcasting large task binary with size 14.2 MiB
22/11/16 02:06:13 WARN TaskSetManager: Stage 34 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:06:25 WARN DAGScheduler: Broadcasting large task binary with size 14.4 MiB
22/11/16 02:06:25 WARN TaskSetManager: Stage 36 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:06:37 WARN DAGScheduler: Broadcasting large task binary with size 14.5 MiB
22/11/16 02:06:37 WARN TaskSetManager: Stage 38 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:06:51 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
22/11/16 02:06:51 WARN TaskSetManager: Stage 40 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:07:03 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
+-----------------+-----+-----+
|fake_predict_fake|    0|    1|
+-----------------+-----+-----+
|              1.0|  647|18310|
|              0.0|16406|  422|
+-----------------+-----+-----+

22/11/16 02:07:04 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
22/11/16 02:07:04 WARN TaskSetManager: Stage 43 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
accuracy: 0.970127148246472
22/11/16 02:07:15 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
22/11/16 02:07:15 WARN TaskSetManager: Stage 45 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
f1: 0.9701171262254067
22/11/16 02:07:25 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
22/11/16 02:07:25 WARN TaskSetManager: Stage 47 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
areaUnderROC: 0.9946405970713533
None
22/11/16 02:07:37 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
22/11/16 02:07:37 WARN TaskSetManager: Stage 58 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
22/11/16 02:07:43 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
+-----------------+----+----+
|fake_predict_fake|   0|   1|
+-----------------+----+----+
|              1.0| 178|4630|
|              0.0|4186| 119|
+-----------------+----+----+

22/11/16 02:07:43 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
22/11/16 02:07:43 WARN TaskSetManager: Stage 61 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
accuracy: 0.9674091956545594
22/11/16 02:07:48 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
22/11/16 02:07:49 WARN TaskSetManager: Stage 63 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
f1: 0.9673988908568226
22/11/16 02:07:54 WARN DAGScheduler: Broadcasting large task binary with size 11.2 MiB
22/11/16 02:07:54 WARN TaskSetManager: Stage 65 contains a task of very large size (26412 KiB). The maximum recommended task size is 1000 KiB.
areaUnderROC: 0.994111525046809
None
SUCCESS: The process with PID 11604 (child process of PID 22252) has been terminated.
SUCCESS: The process with PID 22252 (child process of PID 5584) has been terminated.
SUCCESS: The process with PID 5584 (child process of PID 22080) has been terminated.
```

Figure 21. Log View-Terminated stage

4.1 Process Overview

The entire code is in three scripts, one for each parameter optimization algorithm. All the three scripts compute the execution time of each workload and the performance evaluator.

Three workloads i.e., Memory-intensive, CPU-intensive, Iterative-intensive are first run on Spark's default configurations execution time is stored. Then, parameter optimization algorithms are run one by one. The performance evaluator compares the execution times of every iteration (altered value of parameter(s)) and stores the minimum execution time for all algorithms. The values of parameters are varied in a set range defined in the *Table 1*. The performance evaluator sets the Spark parameters as provided by the optimization algorithms, and then job runner runs the job as shown in *Figure 23*.

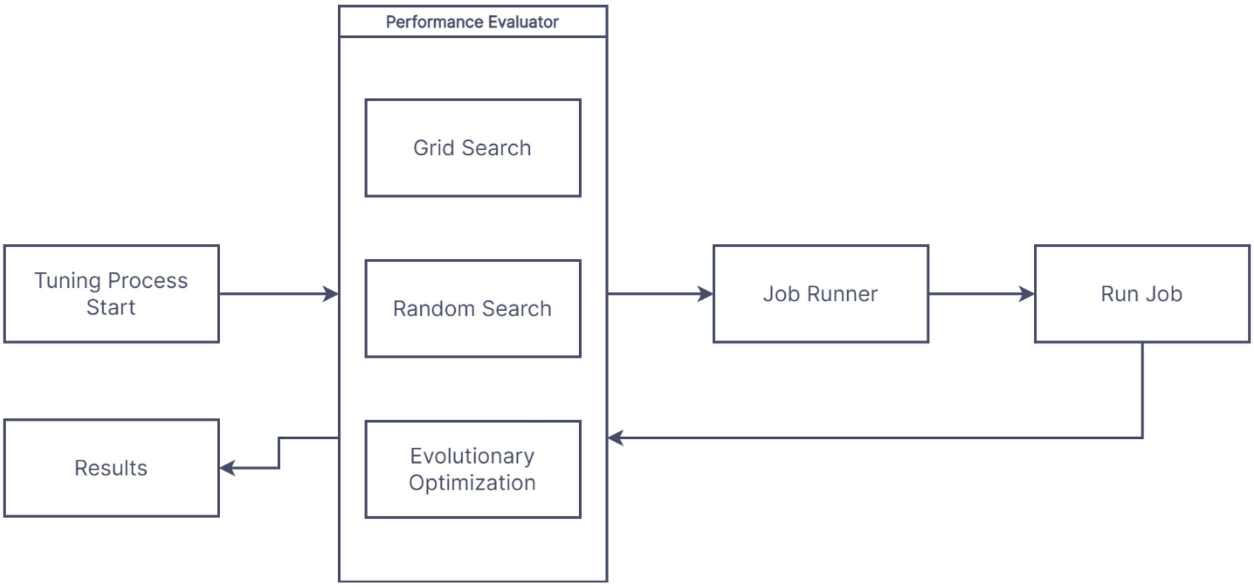


Figure 22. Process Overview

4.2 Experimental Setup

The above approach is implemented on a Spark cluster that consists of one master and two slave nodes. One slave and the master node reside on a single machine, configured with 8GB memory and 4 cores. The other machine (slave) is configured with 16GB memory and 6 cores.

1) ALGORITHM- GRID SEARCH

Grid search is a technique for finding the best combination of hyperparameters for a machine learning model. However, in the context of optimizing Spark configuration, grid search refers to a technique for finding the optimal combination of Spark configuration parameters.

To perform grid search for optimizing Spark configuration, you can follow these steps:

1. Define the range of values for each configuration parameter to optimize. For example, we may want to optimize the `spark.executor.memory` parameter, and we may define the range of values to be [1g, 2g, 3g, 4g].
2. Create a list of all possible combinations of the configuration parameters and their values. For example, if we have two configuration parameters with ranges [1g, 2g] and [2, 4], respectively, we would have four possible combinations: (1g, 2), (1g, 4), (2g, 2), and (2g, 4).
3. For each combination of configuration parameters and their values, set the Spark configuration accordingly and run Spark application or job.
4. Measure the performance of Spark application or job, for example, by measuring its execution time or resource utilization.
5. Repeat steps 3-4 for all combinations of configuration parameters and their values.
6. Select the combination of configuration parameters that resulted in the best performance, and use it for the Spark application.

Grid search can be computationally expensive, as it requires running the Spark application or job multiple times with different configurations. To reduce the computational cost, we consider using a randomized search, which randomly samples from the parameter space instead of exhaustively searching it.

2) ALGORTIHM- RANDOM SEARCH

Random search is an alternative technique to grid search for finding the optimal combination of Spark configuration parameters. In contrast to grid search, which exhaustively searches a predefined set of parameter values, random search randomly samples from the parameter space.

Here are the steps to perform random search for optimizing Spark configuration:

1. Define the parameter space for each of the configuration parameters to optimize. For example, we may define the parameter space for `spark.executor.memory` to be the range [1g, 4g].

2. Set the number of iterations for the random search. This determines how many random combinations of configuration parameters and values will be tried.
3. For each iteration, randomly sample a combination of configuration parameters and values from their respective parameter spaces. For example, if we have two configuration parameters with parameter spaces [1g, 2g, 3g, 4g] and [2, 4], respectively, we could randomly sample (2g, 4) or (1g, 2).
4. Set the Spark configuration to the sampled combination of parameters and values, and run Spark application or job.
5. Measure the performance of Spark application or job, for example, by measuring its execution time or resource utilization.
6. Repeat steps 3-5 for the specified number of iterations.
7. Select the combination of configuration parameters that resulted in the best performance, and use it for Spark application or job.

Random search can be a more efficient method than grid search because it samples a smaller number of combinations of configuration parameters and values. However, it may not guarantee that the optimal combination is found, as it is based on random sampling. Therefore, it is recommended to perform multiple random searches and select the best combination of parameters from the results.

3) ALGORITHM- EVOLUTIONARY OPTIMIZATION

Evolutionary optimization is a metaheuristic optimization technique inspired by biological evolution, where a population of candidate solutions evolves over time through processes such as mutation, selection, and crossover. This technique can also be applied to optimize Spark configuration parameters.

Here are the steps to perform evolutionary optimization for optimizing Spark configuration:

1. Define the parameter space for each configuration parameter to optimize, as well as the population size and the number of generations.
2. Generate an initial population of candidate solutions, where each solution represents a combination of configuration parameters and values. The solutions can be randomly generated or based on expert knowledge.
3. Evaluate the fitness of each solution by running Spark application or job with the corresponding configuration parameters and measuring its performance.
4. Select a subset of the population based on their fitness values, using techniques such as tournament selection or roulette wheel selection.
5. Apply genetic operators such as mutation and crossover to the selected solutions to generate new offspring solutions. For example, mutation could involve randomly changing the value of a configuration parameter, while crossover could involve combining the values of two parent solutions.
6. Evaluate the fitness of the new offspring solutions.
7. Replace some of the least fit solutions in the population with the new offspring solutions.
8. Repeat steps 4-7 for the specified number of generations.
9. Select the solution with the best fitness value from the final population and use it for your Spark application.

Evolutionary optimization can be a powerful technique for optimizing Spark configuration parameters, especially when the parameter space is large and complex.

However, it can also be computationally expensive, as it requires running Spark application or job multiple times to evaluate the fitness of each solution. Additionally, the quality of the solution obtained can depend on the initialization of the population and the choice of genetic operators.

4.3 Workload

The workloads used are designed to simulate archetypal workload behaviors of Spark. For simulating CPU-Intensive behavior we calculate the value of pi to 10^6, for Memory-Intensive behavior we have serialized RDD's of random numbers 0-1000 and a total size of 10000, for Iterative-Intensive behavior we have written a nested loop, with the outer loop running till 10^6 and inner loop running till 10^2.

4.4 Parameters

Following the examples of the previous works we have read and done, we find what factors affect the actual efficiency of Apache spark. Certain factors such as implementation of certain spark functions such as cache and persist or different data models which provide a better arrangement of collected data depending on how frequent u access or transform them, their choice validates on how well a job runs. In this paper we distributed the job into three factors namely Iterative intensive, CPU intensive and memory intensive. We carry out this job with distinct configuration of 9 parameters provided in *TABLE 1*.

Parameters	**Range**	**Default**
spark.task.cpus	[1, 5]	1
spark.memory.storageFraction	[0.25, 0.9]	0.5
spark.memory.fraction	[0.25, 0.8]	0.6
spark.shuffle.file.buffer	[16000, 512100]	32000
spark.scheduler.listenerbus.eventqueue.capacity	[2500, 25000]	10000
spark.storage.memoryMapThreshold	[1000000, 5000000]	2000000
spark.default.parallelism	[4, 24]	24

spark.shuffle.spill.compress	[FALSE, TRUE]	FALSE
spark.executor.cores	[1,32]	2

Table 1. List of parameters, range, and default values

CHAPTER 5: CONCLUSIONS

5.1 Results

The workloads run on default configurations is used as benchmark (see *Table 2*).

Workload	**Execution Time (seconds)**
Iterative- intensive	14.21
Memory- intensive	13.80
CPU- intensive	12.01

Table 2. Benchmark figures

Grid search computed the following (see *Table 3*).

Workload	**Execution Time (seconds)**
Iterative- intensive	9.93
Memory- intensive	10.91
CPU- intensive	9.87

Table 3. Grid search results

Random search computed the following (see *Table 4*).

Workload	**Execution Time (seconds)**
Iterative- intensive	10.21
Memory- intensive	11.65
CPU- intensive	10.06

Table 4. Random search results

Evolutionary optimization computed the following (see *Table 5*).

Workload	**Execution Time (seconds)**
Iterative- intensive	10.02
Memory- intensive	10.78
CPU- intensive	9.99

Table 5. Evolutionary optimization results

Grid search tunes the parameters and reduces the overall execution time by 23.26% in 10000 seconds.

Random search tunes the parameters and reduces the overall execution time by 20.23% in 6000 seconds.

Evolutionary optimization tunes the parameters and reduces the overall execution time by 23.06% in 7000 seconds.

5.2 Conclusion and Future Scope

We are continuing to improve Spark for both usability and performance. On the usability side, we and other members of the community are augmenting Spark with a large set of standard libraries containing scalable versions of common data analysis algorithms. For example, Spark's machine learning library, MLlib, grew by a factor of 4 in the past year. We have also designed a pluggable data source API that makes it easy to access external data sources in a uniform way using DataFrames or SQL [6]. Together, these APIs form one of the largest integrated standard libraries for "big data," and will undoubtedly lead to interesting design decisions to enable efficient composition of workflows.

We have also increasingly seen Spark used in research projects, including online aggregation [7], graph processing, genomic data processing, and large-scale neuroscience. We hope that Spark's relatively small code size and wide array of built-in functions make it amenable to both systems and application-oriented projects.

All the functionality described in this work is open source and available at spark.apache.org

It can be concluded that the Random search is the most efficient algorithm as its percentage reduction in execution time to overall execution time of algorithm is maximum.

Optimizing the Spark parameters used in this project might affect different workloads in different manners. So, selecting the correct parameters is a very important task. Here, in this project, better results might be achieved by selecting other parameters, a machine learning and prediction model can be created for the same.

REFERENCES

[1] Matei Zaharia, Reynold S. Xin, Patrick Wendell, Tathagata Das, Michael Armbrust, Ankur Dave, Xiangrui Meng, Josh Rosen, Shivaram Venkataraman, Michael J. Franklin, Ali Ghodsi, Joseph Gonzalez, Scott Shenker, Ion Stoica, "Apache Spark: A Unified Engine For Big Data Processing," in Comms. of the ACM, Vol. 59, No. 11, pp. 56-65, Nov. 2016, doi: 10.1145/2934664

[2] Shaik Hussain Bhasha, Dr. G.A. Ramachandra, "Analyzing & Optimizing Spark Performance," in Intl. Journal of Innovative Research in Science, Eng., and Tech., Vol. 7, Issue 10, October 2018, ISSN (Online): 2319-8753, ISSN (Print): 2347-6710

[3] Celestine Dunner, Thomas Parnell, Kubilay Atasu, Manolis Sifalakis, Haralampos Pozidis, "Understanding and Optimizing the Performance of Distributed Machine Learning Applications on Apache Spark," in 2017 IEEE International Conference on Big Data (BIGDATA), 11-14 December 2017, doi**:** 10.1109/BigData.2017.8257942

[4] Ahmed N., Barczak A.L.C., Susnjak T., "A comprehensive performance analysis of Apache Hadoop and Apache Spark for large scale data sets using HiBench," in J Big Data **7**, 110 (2020), December 2020, https://doi.org/10.1186/s40537-020-00388-5

[5] Aziz K., Zaidouni D. & Bellafkih M., "Leveraging resource management for efficient performance of Apache Spark," in *J Big Data* 6, 78 (2019), Aug 2019, https://doi.org/10.1186/s40537-019-0240-1

[6] Salloum S., Dautov R., Chen X., "Big data analytics on Apache Spark," in *Int J Data Sci Anal* 1, 145–164 (2016), Oct 2016, https://doi.org/10.1007/s41060-016-0027-9

[7] Gupta P., Sharma A., & Jindal R., "An Approach for Optimizing the Performance for Apache Spark Applications," in 2018 4th International Conference on Computing Communication and Automation (ICCCA), Dec 2018, doi:10.1109/ccaa.2018.8777541

[8] N. Nguyen, M. Maifi Hasan Khan and K. Wang, "Towards Automatic Tuning of Apache Spark Configuration," 2018 IEEE 11th International

Conference on Cloud Computing (CLOUD), 2018, pp. 417-425, doi: 10.1109/CLOUD.2018.00059

[9] Filippo Schiavio, Daniele Bonetta, Walter Binder,” Dynamic speculative optimizations for SQL compilation in Apache Spark,” in Proceedings of the VLDB Endowment, Vol. 13, Issue 5, pp 754–767, Feb 2020, doi: 0.14778/3377369.3377382

[10] Gregory Essertel, Ruby Tahboub, James Decker, Kevin Brown, Kunle Olukotun, Tiark Rompf, “ Flare: Optimizing Apache Spark with Native Compilation for Scale-Up Architectures and Medium-Size Data,” in Proceedings of the 13th USENIX Symposium on Operating Systems Design and Implementation (OSDI ’18), October 8–10, 2018, ISBN 978-1-939133-08-3

APPENDICES

[1] Apache spark v1.2

[2] www.netty.io

[3] https://spark.apache.org/news/spark-wins-daytona-gray-sort-100tb-benchmark.html

[4] https://hadoop.apache.org/docs/r1.2.1/hdfs_design.html

[5] https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html

[6] https://spark.apache.org/sql/

[7] https://freecontent.manning.com/aggregating-your-data-2/