Get Spark from the downloads page of the project website. This documentation is for Spark version 3.5.5. Spark uses Hadoops client libraries for HDFS and YARN. Downloads are pre-packaged for a handful of popular Hadoop versions.Users can also download a Hadoop free binary and run Spark with any Hadoop versionby augmenting Sparks classpath.Scala and Java users can include Spark in their projects using its Maven coordinates and Python users can install Spark from PyPI. If youd like to build Spark fromsource, visit Building Spark. Spark runs on both Windows and UNIX-like systems (e.g. Linux, Mac OS), and it should run on any platform that runs a supported version of Java. This should include JVMs on x86_64 and ARM64. Its easy to run locally on one machine all you need is to have java installed on your system PATH, or the JAVA_HOME environment variable pointing to a Java installation. Spark runs on Java 8/11/17, Scala 2.12/2.13, Python 3.8+, and R 3.5+.Java 8 prior to version 8u371 support is deprecated as of Spark 3.5.0.When using the Scala API, it is necessary for applications to use the same version of Scala that Spark was compiled for.For example, when using Scala 2.13, use Spark compiled for 2.13, and compile code/applications for Scala 2.13 as well. For Java 11, setting -Dio.netty.tryReflectionSetAccessible=true is required for the Apache Arrow library. This prevents the java.lang.UnsupportedOperationException: sun.misc.Unsafe or java.nio.DirectByteBuffer.(long, int) not available error when Apache Arrow uses Netty internally. Running the Examples and Shell Spark comes with several sample programs. Python, Scala, Java, and R examples are in theexamples/src/main directory. To run Spark interactively in a Python interpreter, usebin/pyspark: ./bin/pyspark --master "local[2]" Sample applications are provided in Python. For example: ./bin/spark-submit examples/src/main/python/pi.py 10 To run one of the Scala or Java sample programs, usebin/run-example [params] in the top-level Spark directory. (Behind the scenes, thisinvokes the more generalspark-submit script forlaunching applications). For example, ./bin/run-example SparkPi 10 You can also run Spark interactively through a modified version of the Scala shell. This is agreat way to learn the framework. ./bin/spark-shell --master "local[2]" The --master option specifies themaster URL for a distributed cluster, or local to runlocally with one thread, or local[N] to run locally with N threads. You should start by usinglocal for testing. For a full list of options, run the Spark shell with the --help option. Since version 1.4, Spark has provided an R API (only the DataFrame APIs are included).To run Spark interactively in an R interpreter, use bin/sparkR: ./bin/sparkR --master "local[2]" Example applications are also provided in R. For example: ./bin/spark-submit examples/src/main/r/dataframe.R Running Spark Client Applications Anywhere with Spark Connect Spark Connect is a new client-server architecture introduced in Spark 3.4 that decouples Sparkclient applications and allows remote connectivity to Spark clusters. The separation betweenclient and server allows Spark and its open ecosystem to be leveraged from anywhere, embeddedin any application. In Spark 3.4, Spark Connect provides DataFrame API coverage for PySpark andDataFrame/Dataset API support in Scala. To learn more about Spark Connect and how to use it, see Spark Connect Overview. Launching on a Cluster The Spark cluster mode overview explains the key concepts in running on a cluster.Spark can run both by itself, or over several existing cluster managers. It currently provides severaloptions for deployment: Where to Go from Here Programming Guides: Quick Start: a quick introduction to the Spark API; start here! RDD Programming Guide: overview of Spark basics - RDDs (core but old API), accumulators, and broadcast variables Spark SQL, Datasets, and DataFrames: processing structured data with relational queries (newer API than RDDs) Structured Streaming: processing structured data streams with relation queries (using Datasets and DataFrames, newer API than DStreams) Spark Streaming: processing data streams using DStreams (old API) MLlib: applying machine learning algorithms GraphX: processing graphs SparkR: processing data with Spark in R PySpark: processing data with Spark in Python Spark SQL CLI: processing data with SQL on the command line API Docs: Deployment Guides: Other Documents: External Resources: Note that Spark 4 is pre-built with Scala 2.13, and support for Scala 2.12 has been officially dropped. Spark 3 is pre-built with Scala 2.12 in general and Spark 3.2+ provides additional pre-built distribution with Scala 2.13. Spark artifacts are hosted in Maven Central. You can add a Maven dependency with the following coordinates: groupId: org.apache.sparkartifactId: spark-core_2.13version: 4.0.0 Installing with PyPiPySpark is now available in pypi. To install just run pip install pyspark. Installing with Docker Spark docker images are available from Dockerhub under the accounts of both The Apache Software Foundation and Official Images. Note that, these images contain non-ASF software and may be subject to different license terms. Please check their Dockerfiles to verify whether they are compatible with your deployment. Release notes for stable releases Archived releases As new Spark releases come out for each development stream, previous ones will be archived,but they are still available at Spark release archives. NOTE: Previous releases of Spark may be affected by security issues. Please consult theSecurity page for a list of known issues that may affect the version you downloadbefore deciding to use it. Setup instructions, programming guides, and other documentation are available for each stable version of Spark below: Documentation for preview releases: The documentation linked to above covers getting started with Spark, as well the built-in components MLlib,Spark Streaming, and GraphX. In addition, this page lists other resources for learning Spark. VideosSee the Apache Spark YouTube Channel for videos from Spark events. There are separate playlists for videos of different topics. Besides browsing through playlists, you can also find direct links to videos below. Screencast Tutorial Videos Spark Summit Videos Meetup Talk VideosIn addition to the videos listed below, you can also view all slides from Bay Area meetups here. Training Materials Training materials and exercises from Spark Summit 2014 are available online. These include videos and slides of talks as well as exercises you can run on your laptop. Topics include Spark core, tuning and debugging, Spark SQL, Spark Streaming, GraphX and MLlib. Spark Summit 2013 included a training session, with slides and videos available on the training day agenda. The session also included exercises that you can walk through on Amazon EC2. The UC Berkeley AMPLab regularly hosts training camps on Spark and related projects.Slides, videos and EC2-based exercises from each of these are available online: Hands-On Exercises External Tutorials, Blog Posts, and Talks Books Learning Spark, by Holden Karau, Andy Konwinski, Patrick Wendell and Matei Zaharia (O'Reilly Media) Spark in Action, by Marko Bonaci and Petar Zecevic (Manning) Advanced Analytics with Spark, by Juliet Hougland, Uri Laserson, Sean Owen, Sandy Ryza and Josh Wills (O'Reilly Media) Spark GraphX in Action, by Michael Malak (Manning) Fast Data Processing with Spark, by Krishna Sankar and Holden Karau (Packt Publishing) Machine Learning with Spark, by Nick Pentreath (Packt Publishing) Spark Cookbook, by Rishi Yadav (Packt Publishing) Apache Spark Graph Processing, by Rindra Ramamonjison (Packt Publishing) Mastering Apache Spark, by Mike Frampton (Packt Publishing) Big Data Analytics with Spark: A Practitioner's Guide to Using Spark for Large Scale Data Analysis, by Mohammed Guller (Apress) Large Scale Machine Learning with Spark, by Md. Rezaul Karim, Md. Mahedi Kaysar (Packt Publishing) Big Data Analytics with Spark and Hadoop, by Venkat Ankam (Packt Publishing) Examples The Spark examples page shows the basic API in Scala, Java and Python. Research Papers Spark was initially developed as a UC Berkeley research project, and much of the design is documented in papers.The research page lists some of the original motivation and direction. At a high level, every Spark application consists of a driver program that runs the users main function and executes various parallel operations on a cluster. The main abstraction Spark provides is a resilient distributed dataset (RDD), which is a collection of elements partitioned across the nodes of the cluster that can be operated on in parallel. RDDs are created by starting with a file in the Hadoop file system (or any other Hadoop-supported file system), or an existing Scala collection in the driver program, and transforming it. Users may also ask Spark to persist an RDD in memory, allowing it to be reused efficiently across parallel operations. Finally, RDDs automatically recover from node failures. A second abstraction in Spark is shared variables that can be used in parallel operations. By default, when Spark runs a function in parallel as a set of tasks on different nodes, it ships a copy of each variable used in the function to each task. Sometimes, a variable needs to be shared across tasks, or between tasks and the driver program. Spark supports two types of shared variables: broadcast variables, which can be used to cache a value in memory on all nodes, and accumulators, which are variables that are only added to, such as counters and sums. This guide shows each of these features in each of Sparks supported languages. It is easiest to followalong with if you launch Sparks interactive shell either bin/spark-shell for the Scala shell orbin/pyspark for the Python one. Spark 3.5.5 works with Python 3.8+. It can use the standard CPython interpreter,so C libraries like NumPy can be used. It also works with PyPy 7.3.6+. Spark applications in Python can either be run with the bin/spark-submit script which includes Spark at runtime, or by including it in your setup.py as: install_requires=['pyspark==3.5.5'] To run Spark applications in Python without pip installing PySpark, use the bin/spark-submit script located in the Spark directory.This script will load Sparks Java/Scala libraries and allow you to submit applications to a cluster.You can also use bin/pyspark to launch an interactive Python shell. If you wish to access HDFS data, you need to use a build of PySpark linkingto your version of HDFS.Prebuilt packages are also available on the Spark homepagefor common HDFS versions. Finally, you need to import some Spark classes into your program. Add the following line: from pyspark import SparkContext, SparkConf PySpark requires the same minor version of Python in both driver and workers. It uses the default python version in PATH,you can specify which version of Python you want to use by PYSPARK_PYTHON, for example: $ PYSPARK_PYTHON=python3.8 bin/pyspark$ PYSPARK_PYTHON=/path-to-your-pypy/pypy bin/spark-submit examples/src/main/python/pi.py Spark 3.5.5 is built and distributed to work with Scala 2.12by default. (Spark can be built to work with other versions of Scala, too.) To writeapplications in Scala, you will need to use a compatible Scala version (e.g. 2.12.X). To write a Spark application, you need to add a Maven dependency on Spark. Spark is available through Maven Central at: groupId = org.apache.sparkartifactId = spark-core_2.12version = 3.5.5 In addition, if you wish to access an HDFS cluster, you need to add a dependency onhadoop-client for your version of HDFS. groupId = org.apache.hadoopartifactId = hadoop-clientversion = Finally, you need to import some Spark classes into your program. Add the following lines: import org.apache.spark.SparkContextimport org.apache.spark.SparkConf (Before Spark 1.3.0, you need to explicitly import org.apache.spark.SparkContext._ to enable essential implicit conversions.) Spark 3.5.5 supportslambda expressionsfor concisely writing functions, otherwise you can use the classes in theorg.apache.spark.api.java.function package. Note that support for Java 7 was removed in Spark 2.2.0. To write a Spark application in Java, you need to add a dependency on Spark. Spark is available through Maven Central at: groupId = org.apache.sparkartifactId = spark-core_2.12version = 3.5.5 In addition, if you wish to access an HDFS cluster, you need to add a dependency onhadoop-client for your version of HDFS. groupId = org.apache.hadoopartifactId = hadoop-clientversion = Finally, you need to import some Spark classes into your program. Add the following lines: import org.apache.spark.api.java.JavaSparkContext;import org.apache.spark.api.java.JavaRDD;import org.apache.spark.SparkConf; Initializing Spark The first thing a Spark program must do is to create a SparkContext object, which tells Sparkhow to access a cluster. To create a SparkContext you first need to build a SparkConf objectthat contains information about your application. conf = SparkConf().setAppName(appName).setMaster(master)sc = SparkContext(conf=conf) The first thing a Spark program must do is to create a SparkContext object, which tells Sparkhow to access a cluster. To create a SparkContext you first need to build a SparkConf objectthat contains information about your application. Only one SparkContext should be active per JVM. You must stop() the active SparkContext before creating a new one. val conf = new SparkConf().setAppName(appName).setMaster(master)new SparkContext(conf) The first thing a Spark program must do is to create a JavaSparkContext object, which tells Sparkhow to access a cluster. To create a SparkContext you first need to build a SparkConf objectthat contains information about your application. SparkConf conf = new SparkConf().setAppName(appName).setMaster(master);JavaSparkContext sc = new JavaSparkContext(conf); The appName parameter is a name for your application to show on the cluster UI.master is a Spark, Mesos or YARN cluster URL,or a special local string to run in local mode.In practice, when running on a cluster, you will not want to hardcode master in the program,but rather launch the application with spark-submit andreceive it there. However, for local testing and unit tests, you can pass local to run Sparkin-process. Using the Shell In the PySpark shell, a special interpreter-aware SparkContext is already created for you, in thevariable called sc. Making your own SparkContext will not work. You can set which master thecontext connects to using the --master argument, and you can add Python .zip, .egg or .py filesto the runtime path by passing a comma-separated list to --py-files. For third-party Python dependencies,see Python Package Management. You can also add dependencies(e.g. Spark Packages) to your shell session by supplying a comma-separated list of Maven coordinatesto the --packages argument. Any additional repositories where dependencies might exist (e.g. Sonatype)can be passed to the --repositories argument. For example, to runbin/pyspark on exactly four cores, use: $ ./bin/pyspark --master local[4] Or, to also add code.py to the search path (in order to later be able to import code), use: $ ./bin/pyspark --master local[4] --py-files code.py For a complete list of options, run pyspark --help. Behind the scenes,pyspark invokes the more general spark-submit script. It is also possible to launch the PySpark shell in IPython, theenhanced Python interpreter. PySpark works with IPython 1.0.0 and later. Touse IPython, set the PYSPARK_DRIVER_PYTHON variable to ipython when running bin/pyspark: $ PYSPARK_DRIVER_PYTHON=ipython ./bin/pyspark To use the Jupyter notebook (previously known as the IPython notebook), $ PYSPARK_DRIVER_PYTHON=jupyter PYSPARK_DRIVER_PYTHON_OPTS=notebook ./bin/pyspark You can customize the ipython or jupyter commands by setting PYSPARK_DRIVER_PYTHON_OPTS. After the Jupyter Notebook server is launched, you can create a new notebook fromthe Files tab. Inside the notebook, you can input the command %pylab inline as part ofyour notebook before you start to try Spark from the Jupyter notebook. In the Spark shell, a special interpreter-aware SparkContext is already created for you, in thevariable called sc. Making your own SparkContext will not work. You can set which master thecontext connects to using the --master argument, and you can add JARs to the classpathby passing a comma-separated list to the --jars argument. You can also add dependencies(e.g. Spark Packages) to your shell session by supplying a comma-separated list of Maven coordinatesto the --packages argument. Any additional repositories where dependencies might exist (e.g. Sonatype)can be passed to the --repositories argument. For example, to run bin/spark-shell on exactlyfour cores, use: $ ./bin/spark-shell --master local[4] Or, to also add code.jar to its classpath, use: $ ./bin/spark-shell --master local[4] --jars code.jar To include a dependency using Maven coordinates: $ ./bin/spark-shell --master local[4] --packages "org.example:example:0.1" For a complete list of options, run spark-shell --help. Behind the scenes,spark-shell invokes the more general spark-submit script. Resilient Distributed Datasets (RDDs) Spark revolves around the concept of a resilient distributed dataset (RDD), which is a fault-tolerant collection of elements that can be operated on in parallel. There are two ways to create RDDs: parallelizingan existing collection in your driver program, or referencing a dataset in an external storage system, such as ashared filesystem, HDFS, HBase, or any data source offering a Hadoop InputFormat. Parallelized Collections Parallelized collections are created by calling SparkContexts parallelize method on an existing iterable or collection in your driver program. The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5: data = [1, 2, 3, 4, 5]distData = sc.parallelize(data) Once created, the distributed dataset (distData) can be operated on in parallel. For example, we can call distData.reduce(lambda a, b: a + b) to add up the elements of the list.We describe operations on distributed datasets later on. Parallelized collections are created by calling SparkContexts parallelize method on an existing collection in your driver program (a Scala Seq). The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5: val data = Array(1, 2, 3, 4, 5)val distData = sc.parallelize(data) Once created, the distributed dataset (distData) can be operated on in parallel. For example, we might call distData.reduce((a, b) => a + b) to add up the elements of the array. We describe operations on distributed datasets later on. Parallelized collections are created by calling JavaSparkContexts parallelize method on an existing Collection in your driver program. The elements of the collection are copied to form a distributed dataset that can be operated on in parallel. For example, here is how to create a parallelized collection holding the numbers 1 to 5: List data = Arrays.asList(1, 2, 3, 4, 5);JavaRDD distData = sc.parallelize(data); Once created, the distributed dataset (distData) can be operated on in parallel. For example, we might call distData.reduce((a, b) -> a + b) to add up the elements of the list.We describe operations on distributed datasets later on. One important parameter for parallel collections is the number of partitions to cut the dataset into. Spark will run one task for each partition of the cluster. Typically you want 2-4 partitions for each CPU in your cluster. Normally, Spark tries to set the number of partitions automatically based on your cluster. However, you can also set it manually by passing it as a second parameter to parallelize (e.g. sc.parallelize(data, 10)). Note: some places in the code use the term slices (a synonym for partitions) to maintain backward compatibility. External Datasets PySpark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc. Spark supports text files, SequenceFiles, and any other Hadoop InputFormat. Text file RDDs can be created using SparkContexts textFile method. This method takes a URI for the file (either a local path on the machine, or a hdfs://, s3a://, etc URI) and reads it as a collection of lines. Here is an example invocation: >>> distFile = sc.textFile("data.txt") Once created, distFile can be acted on by dataset operations. For example, we can add up the sizes of all the lines using the map and reduce operations as follows: distFile.map(lambda s: len(s)).reduce(lambda a, b: a + b). Some notes on reading files with Spark: If using a path on the local filesystem, the file must also be accessible at the same path on worker nodes. Either copy the file to all workers or use a network-mounted shared file system. All of Sparks file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well. For example, you can use textFile("/my/directory"), textFile("/my/directory/*.txt"), and textFile("/my/directory/*.gz"). The textFile method also takes an optional second argument for controlling the number of partitions of the file. By default, Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value. Note that you cannot have fewer partitions than blocks. Apart from text files, Sparks Python API also supports several other data formats: SparkContext.wholeTextFiles lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file. RDD.saveAsPickleFile and SparkContext.pickleFile support saving an RDD in a simple format consisting of pickled Python objects. Batching is used on pickle serialization, with default batch size 10. SequenceFile and Hadoop Input/Output Formats Note this feature is currently marked Experimental and is intended for advanced users. It may be replaced in future with read/write support based on Spark SQL, in which case Spark SQL is the preferred approach. Writable Support PySpark SequenceFile support loads an RDD of key-value pairs within Java, converts Writables to base Java types, and pickles theresulting Java objects using pickle. When saving an RDD of key-value pairs to SequenceFile,PySpark does the reverse. It unpickles Python objects into Java objects and then converts them to Writables. The followingWritables are automatically converted: Writable TypePython TypeTextstrIntWritableintFloatWritablefloatDoubleWritablefloatBooleanWritableboolBytesWritablebytearrayNullWritableNoneMapWritabledict Arrays are not handled out-of-the-box. Users need to specify custom ArrayWritable subtypes when reading or writing. When writing,users also need to specify custom converters that convert arrays to custom ArrayWritable subtypes. When reading, the defaultconverter will convert custom ArrayWritable subtypes to Java Object[], which then get pickled to Python tuples. To getPython array.array for arrays of primitive types, users need to specify custom converters. Saving and Loading SequenceFiles Similarly to text files, SequenceFiles can be saved and loaded by specifying the path. The key and valueclasses can be specified, but for standard Writables this is not required. >>> rdd = sc.parallelize(range(1, 4)).map(lambda x: (x, "a" * x))>>> rdd.saveAsSequenceFile("path/to/file")>>> sorted(sc.sequenceFile("path/to/file").collect())[(1, u'a'), (2, u'aa'), (3, u'aaa')] Saving and Loading Other Hadoop Input/Output Formats PySpark can also read any Hadoop InputFormat or write any Hadoop OutputFormat, for both new and old Hadoop MapReduce APIs.If required, a Hadoop configuration can be passed in as a Python dict. Here is an example using theElasticsearch ESInputFormat: $ ./bin/pyspark --jars /path/to/elasticsearch-hadoop.jar>>> conf = {"es.resource" : "index/type"} # assume Elasticsearch is running on localhost defaults>>> rdd = sc.newAPIHadoopRDD("org.elasticsearch.hadoop.mr.EsInputFormat", "org.apache.hadoop.io.NullWritable", "org.elasticsearch.hadoop.mr.LinkedMapWritable", conf=conf)>>> rdd.first() # the result is a MapWritable that is converted to a Python dict(u'Elasticsearch ID', {u'field1': True, u'field2': u'Some Text', u'field3': 12345}) Note that, if the InputFormat simply depends on a Hadoop configuration and/or input path, andthe key and value classes can easily be converted according to the above table,then this approach should work well for such cases. If you have custom serialized binary data (such as loading data from Cassandra / HBase), then you will first need totransform that data on the Scala/Java side to something which can be handled by pickles pickler.A Converter trait is providedfor this. Simply extend this trait and implement your transformation code in the convertmethod. Remember to ensure that this class, along with any dependencies required to access your InputFormat, are packaged into your Spark job jar and included on the PySparkclasspath. See the Python examples andthe Converter examplesfor examples of using Cassandra / HBase InputFormat and OutputFormat with custom converters. Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc. Spark supports text files, SequenceFiles, and any other Hadoop InputFormat. Text file RDDs can be created using SparkContexts textFile method. This method takes a URI for the file (either a local path on the machine, or a hdfs://, s3a://, etc URI) and reads it as a collection of lines. Here is an example invocation: scala> val distFile = sc.textFile("data.txt")distFile: org.apache.spark.rdd.RDD[String] = data.txt MapPartitionsRDD[10] at textFile at :26 Once created, distFile can be acted on by dataset operations. For example, we can add up the sizes of all the lines using the map and reduce operations as follows: distFile.map(s => s.length).reduce((a, b) => a + b). Some notes on reading files with Spark: If using a path on the local filesystem, the file must also be accessible at the same path on worker nodes. Either copy the file to all workers or use a network-mounted shared file system. All of Sparks file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well. For example, you can use textFile("/my/directory"), textFile("/my/directory/*.txt"), and textFile("/my/directory/*.gz"). When multiple files are read, the order of the partitions depends on the order the files are returned from the filesystem. It may or may not, for example, follow the lexicographic ordering of the files by path. Within a partition, elements are ordered according to their order in the underlying file. The textFile method also takes an optional second argument for controlling the number of partitions of the file. By default, Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value. Note that you cannot have fewer partitions than blocks. Apart from text files, Sparks Scala API also supports several other data formats: SparkContext.wholeTextFiles lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file. Partitioning is determined by data locality which, in some cases, may result in too few partitions. For those cases, wholeTextFiles provides an optional second argument for controlling the minimal number of partitions. For SequenceFiles, use SparkContexts sequenceFile[K, V] method where K and V are the types of key and values in the file. These should be subclasses of Hadoops Writable interface, like IntWritable and Text. In addition, Spark allows you to specify native types for a few common Writables; for example, sequenceFile[Int, String] will automatically read IntWritables and Texts. For other Hadoop InputFormats, you can use the SparkContext.hadoopRDD method, which takes an arbitrary JobConf and input format class, key class and value class. Set these the same way you would for a Hadoop job with your input source. You can also use SparkContext.newAPIHadoopRDD for InputFormats based on the new MapReduce API (org.apache.hadoop.mapreduce). RDD.saveAsObjectFile and SparkContext.objectFile support saving an RDD in a simple format consisting of serialized Java objects. While this is not as efficient as specialized formats like Avro, it offers an easy way to save any RDD. Spark can create distributed datasets from any storage source supported by Hadoop, including your local file system, HDFS, Cassandra, HBase, Amazon S3, etc. Spark supports text files, SequenceFiles, and any other Hadoop InputFormat. Text file RDDs can be created using SparkContexts textFile method. This method takes a URI for the file (either a local path on the machine, or a hdfs://, s3a://, etc URI) and reads it as a collection of lines. Here is an example invocation: JavaRDD distFile = sc.textFile("data.txt"); Once created, distFile can be acted on by dataset operations. For example, we can add up the sizes of all the lines using the map and reduce operations as follows: distFile.map(s -> s.length()).reduce((a, b) -> a + b). Some notes on reading files with Spark: If using a path on the local filesystem, the file must also be accessible at the same path on worker nodes. Either copy the file to all workers or use a network-mounted shared file system. All of Sparks file-based input methods, including textFile, support running on directories, compressed files, and wildcards as well. For example, you can use textFile("/my/directory"), textFile("/my/directory/*.txt"), and textFile("/my/directory/*.gz"). The textFile method also takes an optional second argument for controlling the number of partitions of the file. By default, Spark creates one partition for each block of the file (blocks being 128MB by default in HDFS), but you can also ask for a higher number of partitions by passing a larger value. Note that you cannot have fewer partitions than blocks. Apart from text files, Sparks Java API also supports several other data formats: JavaSparkContext.wholeTextFiles lets you read a directory containing multiple small text files, and returns each of them as (filename, content) pairs. This is in contrast with textFile, which would return one record per line in each file. For SequenceFiles, use SparkContexts sequenceFile[K, V] method where K and V are the types of key and values in the file. These should be subclasses of Hadoops Writable interface, like IntWritable and Text. For other Hadoop InputFormats, you can use the JavaSparkContext.hadoopRDD method, which takes an arbitrary JobConf and input format class, key class and value class. Set these the same way you would for a Hadoop job with your input source. You can also use JavaSparkContext.newAPIHadoopRDD for InputFormats based on the new MapReduce API (org.apache.hadoop.mapreduce). JavaRDD.saveAsObjectFile and JavaSparkContext.objectFile support saving an RDD in a simple format consisting of serialized Java objects. While this is not as efficient as specialized formats like Avro, it offers an easy way to save any RDD. RDD Operations RDDs support two types of operations: transformations, which create a new dataset from an existing one, and actions, which return a value to the driver program after running a computation on the dataset. For example, map is a transformation that passes each dataset element through a function and returns a new RDD representing the results. On the other hand, reduce is an action that aggregates all the elements of the RDD using some function and returns the final result to the driver program (although there is also a parallel reduceByKey that returns a distributed dataset). All transformations in Spark are lazy, in that they do not compute their results right away. Instead, they just remember the transformations applied to some base dataset (e.g. a file). The transformations are only computed when an action requires a result to be returned to the driver program. This design enables Spark to run more efficiently. For example, we can realize that a dataset created through map will be used in a reduce and return only the result of the reduce to the driver, rather than the larger mapped dataset. By default, each transformed RDD may be recomputed each time you run an action on it. However, you may also persist an RDD in memory using the persist (or cache) method, in which case Spark will keep the elements around on the cluster for much faster access the next time you query it. There is also support for persisting RDDs on disk, or replicated across multiple nodes. Basics To illustrate RDD basics, consider the simple program below: lines = sc.textFile("data.txt")lineLengths = lines.map(lambda s: len(s))totalLength = lineLengths.reduce(lambda a, b: a + b) The first line defines a base RDD from an external file. This dataset is not loaded in memory orotherwise acted on: lines is merely a pointer to the file.The second line defines lineLengths as the result of a map transformation. Again, lineLengthsis not immediately computed, due to laziness.Finally, we run reduce, which is an action. At this point Spark breaks the computation into tasksto run on separate machines, and each machine runs both its part of the map and a local reduction,returning only its answer to the driver program. If we also wanted to use lineLengths again later, we could add: lineLengths.persist() before the reduce, which would cause lineLengths to be saved in memory after the first time it is computed. To illustrate RDD basics, consider the simple program below: val lines = sc.textFile("data.txt")val lineLengths = lines.map(s => s.length)val totalLength = lineLengths.reduce((a, b) => a + b) The first line defines a base RDD from an external file. This dataset is not loaded in memory orotherwise acted on: lines is merely a pointer to the file.The second line defines lineLengths as the result of a map transformation. Again, lineLengthsis not immediately computed, due to laziness.Finally, we run reduce, which is an action. At this point Spark breaks the computation into tasksto run on separate machines, and each machine runs both its part of the map and a local reduction,returning only its answer to the driver program. If we also wanted to use lineLengths again later, we could add: lineLengths.persist() before the reduce, which would cause lineLengths to be saved in memory after the first time it is computed. To illustrate RDD basics, consider the simple program below: JavaRDD lines = sc.textFile("data.txt");JavaRDD lineLengths = lines.map(s -> s.length());int totalLength = lineLengths.reduce((a, b) -> a + b); The first line defines a base RDD from an external file. This dataset is not loaded in memory orotherwise acted on: lines is merely a pointer to the file.The second line defines lineLengths as the result of a map transformation. Again, lineLengthsis not immediately computed, due to laziness.Finally, we run reduce, which is an action. At this point Spark breaks the computation into tasksto run on separate machines, and each machine runs both its part of the map and a local reduction,returning only its answer to the driver program. If we also wanted to use lineLengths again later, we could add: lineLengths.persist(StorageLevel.MEMORY_ONLY()); before the reduce, which would cause lineLengths to be saved in memory after the first time it is computed. Passing Functions to Spark Sparks API relies heavily on passing functions in the driver program to run on the cluster.There are three recommended ways to do this: Lambda expressions,for simple functions that can be written as an expression. (Lambdas do not support multi-statementfunctions or statements that do not return a value.) Local defs inside the function calling into Spark, for longer code. Top-level functions in a module. For example, to pass a longer function than can be supported using a lambda, considerthe code below: """MyScript.py"""if __name__ == "__main__": def myFunc(s): words = s.split(" ") return len(words) sc = SparkContext(...) sc.textFile("file.txt").map(myFunc) Note that while it is also possible to pass a reference to a method in a class instance (as opposed toa singleton object), this requires sending the object that contains that class along with the method.For example, consider: class MyClass(object): def func(self, s): return s def doStuff(self, rdd): return rdd.map(self.func) Here, if we create a new MyClass and call doStuff on it, the map inside there references thefunc method of that MyClass instance, so the whole object needs to be sent to the cluster. In a similar way, accessing fields of the outer object will reference the whole object: class MyClass(object): def __init__(self): self.field = "Hello" def doStuff(self, rdd): return rdd.map(lambda s: self.field + s) To avoid this issue, the simplest way is to copy field into a local variable insteadof accessing it externally: def doStuff(self, rdd): field = self.field return rdd.map(lambda s: field + s) Sparks API relies heavily on passing functions in the driver program to run on the cluster.There are two recommended ways to do this: Anonymous function syntax,which can be used for short pieces of code. Static methods in a global singleton object. For example, you can define object MyFunctions and thenpass MyFunctions.func1, as follows: object MyFunctions { def func1(s: String): String = { ... }} myRdd.map(MyFunctions.func1) Note that while it is also possible to pass a reference to a method in a class instance (as opposed toa singleton object), this requires sending the object that contains that class along with the method.For example, consider: class MyClass { def func1(s: String): String = { ... } def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(func1) }} Here, if we create a new MyClass instance and call doStuff on it, the map inside there references thefunc1 method of that MyClass instance, so the whole object needs to be sent to the cluster. It issimilar to writing rdd.map(x => this.func1(x)). In a similar way, accessing fields of the outer object will reference the whole object: class MyClass { val field = "Hello" def doStuff(rdd: RDD[String]): RDD[String] = { rdd.map(x => field + x) }} is equivalent to writing rdd.map(x => this.field + x), which references all of this. To avoid thisissue, the simplest way is to copy field into a local variable instead of accessing it externally: def doStuff(rdd: RDD[String]): RDD[String] = { val field_ = this.field rdd.map(x => field_ + x)} Sparks API relies heavily on passing functions in the driver program to run on the cluster.In Java, functions are represented by classes implementing the interfaces in theorg.apache.spark.api.java.function package.There are two ways to create such functions: Implement the Function interfaces in your own class, either as an anonymous inner class or a named one,and pass an instance of it to Spark. Use lambda expressionsto concisely define an implementation. While much of this guide uses lambda syntax for conciseness, it is easy to use all the same APIsin long-form. For example, we could have written our code above as follows: JavaRDD lines = sc.textFile("data.txt");JavaRDD lineLengths = lines.map(new Function() { public Integer call(String s) { return s.length(); }});int totalLength = lineLengths.reduce(new Function2() { public Integer call(Integer a, Integer b) { return a + b; }}); Or, if writing the functions inline is unwieldy: class GetLength implements Function { public Integer call(String s) { return s.length(); }}class Sum implements Function2 { public Integer call(Integer a, Integer b) { return a + b; }} JavaRDD lines = sc.textFile("data.txt");JavaRDD lineLengths = lines.map(new GetLength());int totalLength = lineLengths.reduce(new Sum()); Note that anonymous inner classes in Java can also access variables in the enclosing scope as longas they are marked final. Spark will ship copies of these variables to each worker node as it doesfor other languages. Understanding closures One of the harder things about Spark is understanding the scope and life cycle of variables and methods when executing code across a cluster. RDD operations that modify variables outside of their scope can be a frequent source of confusion. In the example below well look at code that uses foreach() to increment a counter, but similar issues can occur for other operations as well. Example Consider the naive RDD element sum below, which may behave differently depending on whether execution is happening within the same JVM. A common example of this is when running Spark in local mode (--master = local[n]) versus deploying a Spark application to a cluster (e.g. via spark-submit to YARN): counter = 0rdd = sc.parallelize(data) # Wrong: Don't do this!!def increment_counter(x): global counter counter += xrdd.foreach(increment_counter) print("Counter value: ", counter) var counter = 0var rdd = sc.parallelize(data) // Wrong: Don't do this!!rdd.foreach(x => counter += x) println("Counter value: " + counter) int counter = 0;JavaRDD rdd = sc.parallelize(data); // Wrong: Don't do this!!rdd.foreach(x -> counter += x); println("Counter value: " + counter); Local vs. cluster modes The behavior of the above code is undefined, and may not work as intended. To execute jobs, Spark breaks up the processing of RDD operations into tasks, each of which is executed by an executor. Prior to execution, Spark computes the tasks closure. The closure is those variables and methods which must be visible for the executor to perform its computations on the RDD (in this case foreach()). This closure is serialized and sent to each executor. The variables within the closure sent to each executor are now copies and thus, when counter is referenced within the foreach function, its no longer the counter on the driver node. There is still a counter in the memory of the driver node but this is no longer visible to the executors! The executors only see the copy from the serialized closure. Thus, the final value of counter will still be zero since all operations on counter were referencing the value within the serialized closure. In local mode, in some circumstances, the foreach function will actually execute within the same JVM as the driver and will reference the same original counter, and may actually update it. To ensure well-defined behavior in these sorts of scenarios one should use an Accumulator. Accumulators in Spark are used specifically to provide a mechanism for safely updating a variable when execution is split up across worker nodes in a cluster. The Accumulators section of this guide discusses these in more detail. In general, closures - constructs like loops or locally defined methods, should not be used to mutate some global state. Spark does not define or guarantee the behavior of mutations to objects referenced from outside of closures. Some code that does this may work in local mode, but thats just by accident and such code will not behave as expected in distributed mode. Use an Accumulator instead if some global aggregation is needed. Printing elements of an RDDAnother common idiom is attempting to print out the elements of an RDD using rdd.foreach(println) or rdd.map(println). On a single machine, this will generate the expected output and print all the RDDs elements. However, in cluster mode, the output to stdout being called by the executors is now writing to the executors stdout instead, not the one on the driver, so stdout on the driver wont show these! To print all elements on the driver, one can use the collect() method to first bring the RDD to the driver node thus: rdd.collect().foreach(println). This can cause the driver to run out of memory, though, because collect() fetches the entire RDD to a single machine; if you only need to print a few elements of the RDD, a safer approach is to use the take(): rdd.take(100).foreach(println). Working with Key-Value Pairs While most Spark operations work on RDDs containing any type of objects, a few special operations areonly available on RDDs of key-value pairs.The most common ones are distributed shuffle operations, such as grouping or aggregating the elementsby a key. In Python, these operations work on RDDs containing built-in Python tuples such as (1, 2).Simply create such tuples and then call your desired operation. For example, the following code uses the reduceByKey operation on key-value pairs to count howmany times each line of text occurs in a file: lines = sc.textFile("data.txt")pairs = lines.map(lambda s: (s, 1))counts = pairs.reduceByKey(lambda a, b: a + b) We could also use counts.sortByKey(), for example, to sort the pairs alphabetically, and finallycounts.collect() to bring them back to the driver program as a list of objects. While most Spark operations work on RDDs containing any type of objects, a few special operations areonly available on RDDs of key-value pairs.The most common ones are distributed shuffle operations, such as grouping or aggregating the elementsby a key. In Scala, these operations are automatically available on RDDs containingTuple2 objects(the built-in tuples in the language, created by simply writing (a, b)). The key-value pair operations are available in thePairRDDFunctions class,which automatically wraps around an RDD of tuples. For example, the following code uses the reduceByKey operation on key-value pairs to count howmany times each line of text occurs in a file: val lines = sc.textFile("data.txt")val pairs = lines.map(s => (s, 1))val counts = pairs.reduceByKey((a, b) => a + b) We could also use counts.sortByKey(), for example, to sort the pairs alphabetically, and finallycounts.collect() to bring them back to the driver program as an array of objects. Note: when using custom objects as the key in key-value pair operations, you must be sure that acustom equals() method is accompanied with a matching hashCode() method. For full details, seethe contract outlined in the Object.hashCode()documentation. While most Spark operations work on RDDs containing any type of objects, a few special operations areonly available on RDDs of key-value pairs.The most common ones are distributed shuffle operations, such as grouping or aggregating the elementsby a key. In Java, key-value pairs are represented using thescala.Tuple2 classfrom the Scala standard library. You can simply call new Tuple2(a, b) to create a tuple, and accessits fields later with tuple._1() and tuple._2(). RDDs of key-value pairs are represented by theJavaPairRDD class. You can constructJavaPairRDDs from JavaRDDs using special versions of the map operations, likemapToPair and flatMapToPair. The JavaPairRDD will have both standard RDD functions and specialkey-value ones. For example, the following code uses the reduceByKey operation on key-value pairs to count howmany times each line of text occurs in a file: JavaRDD lines = sc.textFile("data.txt");JavaPairRDD pairs = lines.mapToPair(s -> new Tuple2(s, 1));JavaPairRDD counts = pairs.reduceByKey((a, b) -> a + b); We could also use counts.sortByKey(), for example, to sort the pairs alphabetically, and finallycounts.collect() to bring them back to the driver program as an array of objects. Note: when using custom objects as the key in key-value pair operations, you must be sure that acustom equals() method is accompanied with a matching hashCode() method. For full details, seethe contract outlined in the Object.hashCode()documentation. Transformations The following table lists some of the common transformations supported by Spark. Refer to theRDD API doc(Scala, Java, Python, R)and pair RDD functions doc(Scala, Java)for details. TransformationMeaning map(func) Return a new distributed dataset formed by passing each element of the source through a function func. filter(func) Return a new dataset formed by selecting those elements of the source on which func returns true. flatMap(func) Similar to map, but each input item can be mapped to 0 or more output items (so func should return a Seq rather than a single item). mapPartitions(func) Similar to map, but runs separately on each partition (block) of the RDD, so func must be of type Iterator => Iterator when running on an RDD of type T. mapPartitionsWithIndex(func) Similar to mapPartitions, but also provides func with an integer value representing the index of the partition, so func must be of type (Int, Iterator) => Iterator when running on an RDD of type T. sample(withReplacement, fraction, seed) Sample a fraction fraction of the data, with or without replacement, using a given random number generator seed. union(otherDataset) Return a new dataset that contains the union of the elements in the source dataset and the argument. intersection(otherDataset) Return a new RDD that contains the intersection of elements in the source dataset and the argument. distinct([numPartitions])) Return a new dataset that contains the distinct elements of the source dataset. groupByKey([numPartitions]) When called on a dataset of (K, V) pairs, returns a dataset of (K, Iterable) pairs. Note: If you are grouping in order to perform an aggregation (such as a sum or average) over each key, using reduceByKey or aggregateByKey will yield much better performance. Note: By default, the level of parallelism in the output depends on the number of partitions of the parent RDD. You can pass an optional numPartitions argument to set a different number of tasks. reduceByKey(func, [numPartitions]) When called on a dataset of (K, V) pairs, returns a dataset of (K, V) pairs where the values for each key are aggregated using the given reduce function func, which must be of type (V,V) => V. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. aggregateByKey(zeroValue)(seqOp, combOp, [numPartitions]) When called on a dataset of (K, V) pairs, returns a dataset of (K, U) pairs where the values for each key are aggregated using the given combine functions and a neutral "zero" value. Allows an aggregated value type that is different than the input value type, while avoiding unnecessary allocations. Like in groupByKey, the number of reduce tasks is configurable through an optional second argument. sortByKey([ascending], [numPartitions]) When called on a dataset of (K, V) pairs where K implements Ordered, returns a dataset of (K, V) pairs sorted by keys in ascending or descending order, as specified in the boolean ascending argument. join(otherDataset, [numPartitions]) When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (V, W)) pairs with all pairs of elements for each key. Outer joins are supported through leftOuterJoin, rightOuterJoin, and fullOuterJoin. cogroup(otherDataset, [numPartitions]) When called on datasets of type (K, V) and (K, W), returns a dataset of (K, (Iterable, Iterable)) tuples. This operation is also called groupWith. cartesian(otherDataset) When called on datasets of types T and U, returns a dataset of (T, U) pairs (all pairs of elements). pipe(command, [envVars]) Pipe each partition of the RDD through a shell command, e.g. a Perl or bash script. RDD elements are written to the process's stdin and lines output to its stdout are returned as an RDD of strings. coalesce(numPartitions) Decrease the number of partitions in the RDD to numPartitions. Useful for running operations more efficiently after filtering down a large dataset. repartition(numPartitions) Reshuffle the data in the RDD randomly to create either more or fewer partitions and balance it across them. This always shuffles all data over the network. repartitionAndSortWithinPartitions(partitioner) Repartition the RDD according to the given partitioner and, within each resulting partition, sort records by their keys. This is more efficient than calling repartition and then sorting within each partition because it can push the sorting down into the shuffle machinery. Actions The following table lists some of the common actions supported by Spark. Refer to theRDD API doc(Scala, Java, Python, R) and pair RDD functions doc(Scala, Java)for details. ActionMeaning reduce(func) Aggregate the elements of the dataset using a function func (which takes two arguments and returns one). The function should be commutative and associative so that it can be computed correctly in parallel. collect() Return all the elements of the dataset as an array at the driver program. This is usually useful after a filter or other operation that returns a sufficiently small subset of the data. count() Return the number of elements in the dataset. first() Return the first element of the dataset (similar to take(1)). take(n) Return an array with the first n elements of the dataset. takeSample(withReplacement, num, [seed]) Return an array with a random sample of num elements of the dataset, with or without replacement, optionally pre-specifying a random number generator seed. takeOrdered(n, [ordering]) Return the first n elements of the RDD using either their natural order or a custom comparator. saveAsTextFile(path) Write the elements of the dataset as a text file (or set of text files) in a given directory in the local filesystem, HDFS or any other Hadoop-supported file system. Spark will call toString on each element to convert it to a line of text in the file. saveAsSequenceFile(path) (Java and Scala) Write the elements of the dataset as a Hadoop SequenceFile in a given path in the local filesystem, HDFS or any other Hadoop-supported file system. This is available on RDDs of key-value pairs that implement Hadoop's Writable interface. In Scala, it is also available on types that are implicitly convertible to Writable (Spark includes conversions for basic types like Int, Double, String, etc). saveAsObjectFile(path) (Java and Scala) Write the elements of the dataset in a simple format using Java serialization, which can then be loaded using SparkContext.objectFile(). countByKey() Only available on RDDs of type (K, V). Returns a hashmap of (K, Int) pairs with the count of each key. foreach(func) Run a function func on each element of the dataset. This is usually done for side effects such as updating an Accumulator or interacting with external storage systems. Note: modifying variables other than Accumulators outside of the foreach() may result in undefined behavior. See Understanding closures for more details. The Spark RDD API also exposes asynchronous versions of some actions, like foreachAsync for foreach, which immediately return a FutureAction to the caller instead of blocking on completion of the action. This can be used to manage or wait for the asynchronous execution of the action. Shuffle operations Certain operations within Spark trigger an event known as the shuffle. The shuffle is Sparksmechanism for re-distributing data so that its grouped differently across partitions. This typicallyinvolves copying data across executors and machines, making the shuffle a complex andcostly operation. Background To understand what happens during the shuffle, we can consider the example of thereduceByKey operation. The reduceByKey operation generates a new RDD where allvalues for a single key are combined into a tuple - the key and the result of executing a reducefunction against all values associated with that key. The challenge is that not all values for asingle key necessarily reside on the same partition, or even the same machine, but they must beco-located to compute the result. In Spark, data is generally not distributed across partitions to be in the necessary place for aspecific operation. During computations, a single task will operate on a single partition - thus, toorganize all the data for a single reduceByKey reduce task to execute, Spark needs to perform anall-to-all operation. It must read from all partitions to find all the values for all keys,and then bring together values across partitions to compute the final result for each key -this is called the shuffle. Although the set of elements in each partition of newly shuffled data will be deterministic, and sois the ordering of partitions themselves, the ordering of these elements is not. If one desires predictablyordered data following shuffle then its possible to use: mapPartitions to sort each partition using, for example, .sorted repartitionAndSortWithinPartitions to efficiently sort partitions while simultaneously repartitioning sortBy to make a globally ordered RDD Operations which can cause a shuffle include repartition operations likerepartition and coalesce, ByKey operations(except for counting) like groupByKey and reduceByKey, andjoin operations like cogroup and join. Performance ImpactThe Shuffle is an expensive operation since it involves disk I/O, data serialization, andnetwork I/O. To organize data for the shuffle, Spark generates sets of tasks - map tasks toorganize the data, and a set of reduce tasks to aggregate it. This nomenclature comes fromMapReduce and does not directly relate to Sparks map and reduce operations. Internally, results from individual map tasks are kept in memory until they cant fit. Then, theseare sorted based on the target partition and written to a single file. On the reduce side, tasksread the relevant sorted blocks. Certain shuffle operations can consume significant amounts of heap memory since they employin-memory data structures to organize records before or after transferring them. Specifically,reduceByKey and aggregateByKey create these structures on the map side, and 'ByKey operationsgenerate these on the reduce side. When data does not fit in memory Spark will spill these tablesto disk, incurring the additional overhead of disk I/O and increased garbage collection. Shuffle also generates a large number of intermediate files on disk. As of Spark 1.3, these filesare preserved until the corresponding RDDs are no longer used and are garbage collected.This is done so the shuffle files dont need to be re-created if the lineage is re-computed.Garbage collection may happen only after a long period of time, if the application retains referencesto these RDDs or if GC does not kick in frequently. This means that long-running Spark jobs mayconsume a large amount of disk space. The temporary storage directory is specified by thespark.local.dir configuration parameter when configuring the Spark context. Shuffle behavior can be tuned by adjusting a variety of configuration parameters. See theShuffle Behavior section within the Spark Configuration Guide. RDD Persistence One of the most important capabilities in Spark is persisting (or caching) a dataset in memoryacross operations. When you persist an RDD, each node stores any partitions of it that it computes inmemory and reuses them in other actions on that dataset (or datasets derived from it). This allowsfuture actions to be much faster (often by more than 10x). Caching is a key tool foriterative algorithms and fast interactive use. You can mark an RDD to be persisted using the persist() or cache() methods on it. The first timeit is computed in an action, it will be kept in memory on the nodes. Sparks cache is fault-tolerant if any partition of an RDD is lost, it will automatically be recomputed using the transformationsthat originally created it. In addition, each persisted RDD can be stored using a different storage level, allowing you, for example,to persist the dataset on disk, persist it in memory but as serialized Java objects (to save space),replicate it across nodes.These levels are set by passing aStorageLevel object (Scala,Java,Python)to persist(). The cache() method is a shorthand for using the default storage level,which is StorageLevel.MEMORY_ONLY (store deserialized objects in memory). The full set ofstorage levels is: Storage LevelMeaning MEMORY_ONLY Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, some partitions will not be cached and will be recomputed on the fly each time they're needed. This is the default level. MEMORY_AND_DISK Store RDD as deserialized Java objects in the JVM. If the RDD does not fit in memory, store the partitions that don't fit on disk, and read them from there when they're needed. MEMORY_ONLY_SER (Java and Scala) Store RDD as serialized Java objects (one byte array per partition). This is generally more space-efficient than deserialized objects, especially when using a fast serializer, but more CPU-intensive to read. MEMORY_AND_DISK_SER (Java and Scala) Similar to MEMORY_ONLY_SER, but spill partitions that don't fit in memory to disk instead of recomputing them on the fly each time they're needed. DISK_ONLY Store the RDD partitions only on disk. MEMORY_ONLY_2, MEMORY_AND_DISK_2, etc. Same as the levels above, but replicate each partition on two cluster nodes. OFF_HEAP (experimental) Similar to MEMORY_ONLY_SER, but store the data in off-heap memory. This requires off-heap memory to be enabled. Note: In Python, stored objects will always be serialized with the Pickle library,so it does not matter whether you choose a serialized level. The available storage levels in Python include MEMORY_ONLY, MEMORY_ONLY_2,MEMORY_AND_DISK, MEMORY_AND_DISK_2, DISK_ONLY, DISK_ONLY_2, and DISK_ONLY_3. Spark also automatically persists some intermediate data in shuffle operations (e.g. reduceByKey), even without users calling persist. This is done to avoid recomputing the entire input if a node fails during the shuffle. We still recommend users call persist on the resulting RDD if they plan to reuse it. Which Storage Level to Choose? Sparks storage levels are meant to provide different trade-offs between memory usage and CPUefficiency. We recommend going through the following process to select one: If your RDDs fit comfortably with the default storage level (MEMORY_ONLY), leave them that way.This is the most CPU-efficient option, allowing operations on the RDDs to run as fast as possible. If not, try using MEMORY_ONLY_SER and selecting a fast serialization library tomake the objects much more space-efficient, but still reasonably fast to access. (Java and Scala) Dont spill to disk unless the functions that computed your datasets are expensive, or they filtera large amount of the data. Otherwise, recomputing a partition may be as fast as reading it fromdisk. Use the replicated storage levels if you want fast fault recovery (e.g. if using Spark to serverequests from a web application). All the storage levels provide full fault tolerance byrecomputing lost data, but the replicated ones let you continue running tasks on the RDD withoutwaiting to recompute a lost partition. Removing Data Spark automatically monitors cache usage on each node and drops out old data partitions in aleast-recently-used (LRU) fashion. If you would like to manually remove an RDD instead of waiting forit to fall out of the cache, use the RDD.unpersist() method. Note that this method does notblock by default. To block until resources are freed, specify blocking=true when calling this method. Normally, when a function passed to a Spark operation (such as map or reduce) is executed on aremote cluster node, it works on separate copies of all the variables used in the function. Thesevariables are copied to each machine, and no updates to the variables on the remote machine arepropagated back to the driver program. Supporting general, read-write shared variables across taskswould be inefficient. However, Spark does provide two limited types of shared variables for twocommon usage patterns: broadcast variables and accumulators. Broadcast Variables Broadcast variables allow the programmer to keep a read-only variable cached on each machine ratherthan shipping a copy of it with tasks. They can be used, for example, to give every node a copy of alarge input dataset in an efficient manner. Spark also attempts to distribute broadcast variablesusing efficient broadcast algorithms to reduce communication cost. Spark actions are executed through a set of stages, separated by distributed shuffle operations.Spark automatically broadcasts the common data needed by tasks within each stage. The databroadcasted this way is cached in serialized form and deserialized before running each task. Thismeans that explicitly creating broadcast variables is only useful when tasks across multiple stagesneed the same data or when caching the data in deserialized form is important. Broadcast variables are created from a variable v by calling SparkContext.broadcast(v). Thebroadcast variable is a wrapper around v, and its value can be accessed by calling the valuemethod. The code below shows this: >>> broadcastVar = sc.broadcast([1, 2, 3]) >>> broadcastVar.value[1, 2, 3] scala> val broadcastVar = sc.broadcast(Array(1, 2, 3))broadcastVar: org.apache.spark.broadcast.Broadcast[Array[Int]] = Broadcast(0) scala> broadcastVar.valueres0: Array[Int] = Array(1, 2, 3) Broadcast broadcastVar = sc.broadcast(new int[] {1, 2, 3}); broadcastVar.value();// returns [1, 2, 3] After the broadcast variable is created, it should be used instead of the value v in any functionsrun on the cluster so that v is not shipped to the nodes more than once. In addition, the objectv should not be modified after it is broadcast in order to ensure that all nodes get the samevalue of the broadcast variable (e.g. if the variable is shipped to a new node later). To release the resources that the broadcast variable copied onto executors, call .unpersist().If the broadcast is used again afterwards, it will be re-broadcast. To permanently release allresources used by the broadcast variable, call .destroy(). The broadcast variable cant be usedafter that. Note that these methods do not block by default. To block until resources are freed,specify blocking=true when calling them. Accumulators Accumulators are variables that are only added to through an associative and commutative operation and cantherefore be efficiently supported in parallel. They can be used to implement counters (as inMapReduce) or sums. Spark natively supports accumulators of numeric types, and programmerscan add support for new types. As a user, you can create named or unnamed accumulators. As seen in the image below, a named accumulator (in this instance counter) will display in the web UI for the stage that modifies that accumulator. Spark displays the value for each accumulator modified by a task in the Tasks table. Tracking accumulators in the UI can be useful for understanding the progress ofrunning stages (NOTE: this is not yet supported in Python). An accumulator is created from an initial value v by calling SparkContext.accumulator(v). Tasksrunning on a cluster can then add to it using the add method or the += operator. However, they cannot read its value.Only the driver program can read the accumulators value, using its value method. The code below shows an accumulator being used to add up the elements of an array: >>> accum = sc.accumulator(0)>>> accumAccumulator >>> sc.parallelize([1, 2, 3, 4]).foreach(lambda x: accum.add(x))...10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s >>> accum.value10 While this code used the built-in support for accumulators of type Int, programmers can alsocreate their own types by subclassing AccumulatorParam.The AccumulatorParam interface has two methods: zero for providing a zero value for your datatype, and addInPlace for adding two values together. For example, supposing we had a Vector classrepresenting mathematical vectors, we could write: class VectorAccumulatorParam(AccumulatorParam): def zero(self, initialValue): return Vector.zeros(initialValue.size) def addInPlace(self, v1, v2): v1 += v2 return v1 # Then, create an Accumulator of this type:vecAccum = sc.accumulator(Vector(...), VectorAccumulatorParam()) A numeric accumulator can be created by calling SparkContext.longAccumulator() or SparkContext.doubleAccumulator()to accumulate values of type Long or Double, respectively. Tasks running on a cluster can then add to it usingthe add method. However, they cannot read its value. Only the driver program can read the accumulators value,using its value method. The code below shows an accumulator being used to add up the elements of an array: scala> val accum = sc.longAccumulator("My Accumulator")accum: org.apache.spark.util.LongAccumulator =

LongAccumulator(id: 0, name: Some(My Accumulator), value: 0) scala> sc.parallelize(Array(1, 2, 3, 4)).foreach(x => accum.add(x))...10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s scala> accum.valueres2: Long = 10 While this code used the built-in support for accumulators of type Long, programmers can alsocreate their own types by subclassing AccumulatorV2.The AccumulatorV2 abstract class has several methods which one has to override: reset for resettingthe accumulator to zero, add for adding another value into the accumulator,merge for merging another same-type accumulator into this one. Other methods that must be overriddenare contained in the API documentation. For example, supposing we had a MyVector classrepresenting mathematical vectors, we could write: class VectorAccumulatorV2 extends AccumulatorV2[MyVector, MyVector] { private val myVector: MyVector = MyVector.createZeroVector def reset(): Unit = { myVector.reset() } def add(v: MyVector): Unit = { myVector.add(v) } ...} // Then, create an Accumulator of this type:val myVectorAcc = new VectorAccumulatorV2// Then, register it into spark context:sc.register(myVectorAcc, "MyVectorAcc1") Note that, when programmers define their own type of AccumulatorV2, the resulting type can be different than that of the elements added. A numeric accumulator can be created by calling SparkContext.longAccumulator() or SparkContext.doubleAccumulator()to accumulate values of type Long or Double, respectively. Tasks running on a cluster can then add to it usingthe add method. However, they cannot read its value. Only the driver program can read the accumulators value,using its value method. The code below shows an accumulator being used to add up the elements of an array: LongAccumulator accum = jsc.sc().longAccumulator(); sc.parallelize(Arrays.asList(1, 2, 3, 4)).foreach(x -> accum.add(x));// ...// 10/09/29 18:41:08 INFO SparkContext: Tasks finished in 0.317106 s accum.value();// returns 10 While this code used the built-in support for accumulators of type Long, programmers can alsocreate their own types by subclassing AccumulatorV2.The AccumulatorV2 abstract class has several methods which one has to override: reset for resettingthe accumulator to zero, add for adding another value into the accumulator,merge for merging another same-type accumulator into this one. Other methods that must be overriddenare contained in the API documentation. For example, supposing we had a MyVector classrepresenting mathematical vectors, we could write: class VectorAccumulatorV2 implements AccumulatorV2 { private MyVector myVector = MyVector.createZeroVector(); public void reset() { myVector.reset(); } public void add(MyVector v) { myVector.add(v); } ...} // Then, create an Accumulator of this type:VectorAccumulatorV2 myVectorAcc = new VectorAccumulatorV2();// Then, register it into spark context:jsc.sc().register(myVectorAcc, "MyVectorAcc1"); Note that, when programmers define their own type of AccumulatorV2, the resulting type can be different than that of the elements added. Warning: When a Spark task finishes, Spark will try to merge the accumulated updates in this task to an accumulator.If it fails, Spark will ignore the failure and still mark the task successful and continue to run other tasks. Hence,a buggy accumulator will not impact a Spark job, but it may not get updated correctly although a Spark job is successful. For accumulator updates performed inside actions only, Spark guarantees that each tasks update to the accumulatorwill only be applied once, i.e. restarted tasks will not update the value. In transformations, users should be awareof that each tasks update may be applied more than once if tasks or job stages are re-executed. Accumulators do not change the lazy evaluation model of Spark. If they are being updated within an operation on an RDD, their value is only updated once that RDD is computed as part of an action. Consequently, accumulator updates are not guaranteed to be executed when made within a lazy transformation like map(). The below code fragment demonstrates this property: accum = sc.accumulator(0)def g(x): accum.add(x) return f(x)data.map(g)# Here, accum is still 0 because no actions have caused the `map` to be computed. val accum = sc.longAccumulatordata.map { x => accum.add(x); x }// Here, accum is still 0 because no actions have caused the map operation to be computed. LongAccumulator accum = jsc.sc().longAccumulator();data.map(x -> { accum.add(x); return f(x); });// Here, accum is still 0 because no actions have caused the `map` to be computed. Deploying to a Cluster The application submission guide describes how to submit applications to a cluster.In short, once you package your application into a JAR (for Java/Scala) or a set of .py or .zip files (for Python),the bin/spark-submit script lets you submit it to any supported cluster manager. Launching Spark jobs from Java / Scala The org.apache.spark.launcherpackage provides classes for launching Spark jobs as child processes using a simple Java API. Unit Testing Spark is friendly to unit testing with any popular unit test framework.Simply create a SparkContext in your test with the master URL set to local, run your operations,and then call SparkContext.stop() to tear it down.Make sure you stop the context within a finally block or the test frameworks tearDown method,as Spark does not support two contexts running concurrently in the same program. Where to Go from Here You can see some example Spark programs on the Spark website.In addition, Spark includes several samples in the examples directory(Scala, Java, Python, R).You can run Java and Scala examples by passing the class name to Sparks bin/run-example script; for instance: ./bin/run-example SparkPi For Python examples, use spark-submit instead: ./bin/spark-submit examples/src/main/python/pi.py For R examples, use spark-submit instead: ./bin/spark-submit examples/src/main/r/dataframe.R For help on optimizing your programs, the configuration andtuning guides provide information on best practices. They are especially important formaking sure that your data is stored in memory in an efficient format.For help on deploying, the cluster mode overview describes the components involvedin distributed operation and supported cluster managers. Finally, full API documentation is available inScala, Java, Python and R. Page 2 Get Spark from the downloads page of the project website. This documentation is for Spark version 3.5.5. Spark uses Hadoops client libraries for HDFS and YARN. Downloads are pre-packaged for a handful of popular Hadoop versions.Users can also download a Hadoop free binary and run Spark with any Hadoop versionby augmenting Sparks classpath.Scala and Java users can include Spark in their projects using its Maven coordinates and Python users can install Spark from PyPI. If youd like to build Spark fromsource, visit Building Spark. Spark runs on both Windows and UNIX-like systems (e.g. Linux, Mac OS), and it should run on any platform that runs a supported version of Java. This should include JVMs on x86_64 and ARM64. Its easy to run locally on one machine all you need is to have java installed on your system PATH, or the JAVA_HOME environment variable pointing to a Java installation. Spark runs on Java 8/11/17, Scala 2.12/2.13, Python 3.8+, and R 3.5+.Java 8 prior to version 8u371 support is deprecated as of Spark 3.5.0.When using the Scala API, it is necessary for applications to use the same version of Scala that Spark was compiled for.For example, when using Scala 2.13, use Spark compiled for 2.13, and compile code/applications for Scala 2.13 as well. For Java 11, setting -Dio.netty.tryReflectionSetAccessible=true is required for the Apache Arrow library. This prevents the java.lang.UnsupportedOperationException: sun.misc.Unsafe or java.nio.DirectByteBuffer.(long, int) not available error when Apache Arrow uses Netty internally. Running the Examples and Shell Spark comes with several sample programs. Python, Scala, Java, and R examples are in theexamples/src/main directory. To run Spark interactively in a Python interpreter, usebin/pyspark: ./bin/pyspark --master "local[2]" Sample applications are provided in Python. For example: ./bin/spark-submit examples/src/main/python/pi.py 10 To run one of the Scala or Java sample programs, usebin/run-example [params] in the top-level Spark directory. (Behind the scenes, thisinvokes the more generalspark-submit script forlaunching applications). For example, ./bin/run-example SparkPi 10 You can also run Spark interactively through a modified version of the Scala shell. This is agreat way to learn the framework. ./bin/spark-shell --master "local[2]" The --master option specifies themaster URL for a distributed cluster, or local to runlocally with one thread, or local[N] to run locally with N threads. You should start by usinglocal for testing. For a full list of options, run the Spark shell with the --help option. Since version 1.4, Spark has provided an R API (only the DataFrame APIs are included).To run Spark interactively in an R interpreter, use bin/sparkR: ./bin/sparkR --master "local[2]" Example applications are also provided in R. For example: ./bin/spark-submit examples/src/main/r/dataframe.R Running Spark Client Applications Anywhere with Spark Connect Spark Connect is a new client-server architecture introduced in Spark 3.4 that decouples Sparkclient applications and allows remote connectivity to Spark clusters. The separation betweenclient and server allows Spark and its open ecosystem to be leveraged from anywhere, embeddedin any application. In Spark 3.4, Spark Connect provides DataFrame API coverage for PySpark andDataFrame/Dataset API support in Scala. To learn more about Spark Connect and how to use it, see Spark Connect Overview. Launching on a Cluster The Spark cluster mode overview explains the key concepts in running on a cluster.Spark can run both by itself, or over several existing cluster managers. It currently provides severaloptions for deployment: Where to Go from Here Programming Guides: Quick Start: a quick introduction to the Spark API; start here! RDD Programming Guide: overview of Spark basics - RDDs (core but old API), accumulators, and broadcast variables Spark SQL, Datasets, and DataFrames: processing structured data with relational queries (newer API than RDDs) Structured Streaming: processing structured data streams with relation queries (using Datasets and DataFrames, newer API than DStreams) Spark Streaming: processing data streams using DStreams (old API) MLlib: applying machine learning algorithms GraphX: processing graphs SparkR: processing data with Spark in R PySpark: processing data with Spark in Python Spark SQL CLI: processing data with SQL on the command line API Docs: Deployment Guides: Other Documents: External Resources: Join the community Spark has a thriving open source community, with contributors from around the globe building features, documentation and assisting other users. This page shows you how to use different Apache Spark APIs with simple examples. Spark is a great engine for small and large datasets. It can be used with single-node/localhost environments, or distributed clusters. Sparks expansive API, excellent performance, and flexibility make it a good option for many analyses. This guide shows examples with the following Spark APIs: DataFrames SQL Structured Streaming RDDs The examples use small datasets so the they are easy to follow. Spark DataFrame example This section shows you how to create a Spark DataFrame and run simple operations. The examples are on a small DataFrame, so you can easily see the functionality. Lets start by creating a Spark Session: from pyspark.sql import SparkSessionspark = SparkSession.builder.appName("demo").getOrCreate() Some Spark runtime environments come with pre-instantiated Spark Sessions. The getOrCreate() method will use an existing Spark Session or create a new Spark Session if one does not already exist. Create a Spark DataFrame Start by creating a DataFrame with first_name and age columns and four rows of data: df = spark.createDataFrame([("sue", 32), ("li", 3), ("bob", 75), ("heo", 13),], ["first_name", "age"],) Use the show() method to view the contents of the DataFrame: df.show()+----------+---+|first_name|age|+----------+---+| sue| 32|| li| 3|| bob| 75|| heo| 13|+----------+---+ Now, lets perform some data processing operations on the DataFrame. Add a column to a Spark DataFrame Lets add a life_stage column to the DataFrame that returns child if the age is 12 or under, teenager if the age is between 13 and 19, and adult if the age is 20 or older. from pyspark.sql.functions import col, whendf1 = df.withColumn("life_stage", when(col("age") < 13, "child") .when(col("age").between(13, 19), "teenager") .otherwise("adult"),) Its easy to add columns to a Spark DataFrame. Lets view the contents of df1. df1.show()+----------+---+----------+|first_name|age|life_stage|+----------+---+----------+| sue| 32| adult|| li| 3| child|| bob| 75| adult|| heo| 13| teenager|+----------+---+----------+ Notice how the original DataFrame is unchanged: df.show()+----------+---+|first_name|age|+----------+---+| sue| 32|| li| 3|| bob| 75|| heo| 13|+----------+---+ Spark operations dont mutate the DataFrame. You must assign the result to a new variable to access the DataFrame changes for subsequent operations. Filter a Spark DataFrame Now, filter the DataFrame so it only includes teenagers and adults. df1.where(col("life_stage").isin(["teenager", "adult"])).show()+----------+---+----------+|first_name|age|life_stage|+----------+---+----------+| sue| 32| adult|| bob| 75| adult|| heo| 13| teenager|+----------+---+----------+ Group by aggregation on Spark DataFrame Now, lets compute the average age for everyone in the dataset: from pyspark.sql.functions import avgdf1.select(avg("age")).show()+--------+|avg(age)|+--------+| 30.75|+--------+ You can also compute the average age for each life_stage: df1.groupBy("life_stage").avg().show()+----------+--------+|life_stage|avg(age)|+----------+--------+| adult| 53.5|| child| 3.0|| teenager| 13.0|+----------+--------+ Spark lets you run queries on DataFrames with SQL if you dont want to use the programmatic APIs. Query the DataFrame with SQL Heres how you can compute the average age for everyone with SQL: spark.sql("select avg(age) from {df1}", df1=df1).show()+--------+|avg(age)|+--------+| 30.75|+--------+ And heres how to compute the average age by life_stage with SQL: spark.sql("select life_stage, avg(age) from {df1} group by life_stage", df1=df1).show()+----------+--------+|life_stage|avg(age)|+----------+--------+| adult| 53.5|| child| 3.0|| teenager| 13.0|+----------+--------+ Spark lets you use the programmatic API, the SQL API, or a combination of both. This flexibility makes Spark accessible to a variety of users and powerfully expressive. Spark SQL Example Lets persist the DataFrame in a named Parquet table that is easily accessible via the SQL API. df1.write.saveAsTable("some_people") Make sure that the table is accessible via the table name: spark.sql("select * from some_people").show()+----------+---+----------+|first_name|age|life_stage|+----------+---+----------+| heo| 13| teenager|| sue| 32| adult|| bob| 75| adult|| li| 3| child|+----------+---+----------+ Now, lets use SQL to insert a few more rows of data into the table: spark.sql("INSERT INTO some_people VALUES ('frank', 4, 'child')") Inspect the table contents to confirm the row was inserted: spark.sql("select * from some_people").show()+----------+---+----------+|first_name|age|life_stage|+----------+---+----------+| heo| 13| teenager|| sue| 32| adult|| bob| 75| adult|| li| 3| child|| frank| 4| child|+----------+---+----------+ Run a query that returns the teenagers: spark.sql("select * from some_people where life_stage='teenager'").show()+----------+---+----------+|first_name|age|life_stage|+----------+---+----------+| heo| 13| teenager|+----------+---+----------+ Spark makes it easy to register tables and query them with pure SQL. Spark Structured Streaming Example Spark also has Structured Streaming APIs that allow you to create batch or real-time streaming applications. Lets see how to use Spark Structured Streaming to read data from Kafka and write it to a Parquet table hourly. Suppose you have a Kafka stream thats continuously populated with the following data: {"student_name":"someXXperson", "graduation_year":"2023", "major":"math"}{"student_name":"liXXyao", "graduation_year":"2025", "major":"physics"} Heres how to read the Kafka source into a Spark DataFrame: df = (spark.readStream.format("kafka") .option("kafka.bootstrap.servers", "host1:port1,host2:port2") .option("subscribe", subscribeTopic) .load()) Create a function that cleans the input data. schema = StructType([StructField("student_name", StringType()), StructField("graduation_year", StringType()), StructField("major", StringType()),])def with_normalized_names(df, schema): parsed_df = (df.withColumn("json_data", from_json(col("value").cast("string"), schema)) .withColumn("student_name", col("json_data.student_name")) .withColumn("graduation_year", col("json_data.graduation_year")) .withColumn("major", col("json_data.major")) .drop(col("json_data")) .drop(col("value"))) split_col = split(parsed_df["student_name"], "XX") return (parsed_df.withColumn("first_name", split_col.getItem(0)) .withColumn("last_name", split_col.getItem(1)) .drop("student_name")) Now, create a function that will read all of the new data in Kafka whenever its run. def perform_available_now_update(): checkpointPath = "data/tmp_students_checkpoint/" path = "data/tmp_students" return df.transform(lambda df: with_normalized_names(df)).writeStream.trigger(availableNow=True).format("parquet").option("checkpointLocation", checkpointPath).start(path) Invoke the perform_available_now_update() function and see the contents of the Parquet table. You can set up a cron job to run the perform_available_now_update() function every hour so your Parquet table is regularly updated. Spark RDD Example The Spark RDD APIs are suitable for unstructured data. The Spark DataFrame API is easier and more performant for structured data. Suppose you have a text file called some_text.txt with the following three lines of data: these are wordsthese are more wordswords in english You would like to compute the count of each word in the text file. Here is how to perform this computation with Spark RDDs: text_file = spark.sparkContext.textFile("some_words.txt")counts = (text_file.flatMap(lambda line: line.split(" ")) .map(lambda word: (word, 1)) .reduceByKey(lambda a, b: a + b)) Lets take a look at the result: counts.collect()[('these', 2), ('are', 2), ('more', 1), ('in', 1), ('words', 3), ('english', 1)] Spark allows for efficient execution of the query because it parallelizes this computation. Many other query engines arent capable of parallelizing computations. Conclusion These examples have shown how Spark provides nice user APIs for computations on small datasets. Spark can scale these same code examples to large datasets on distributed clusters. Its fantastic how Spark can handle both large and small datasets. Spark also has an expansive API compared with other query engines. Spark allows you to perform DataFrame operations with programmatic APIs, write SQL, perform streaming analyses, and do machine learning. Spark saves you from learning multiple frameworks and patching together various libraries to perform an analysis. Additional examples Many additional examples are distributed with Spark: Date: May 19, 2025 Version: 4.0.0Useful links:Live Notebook | GitHub | Issues | Examples | Community | Stack Overflow | Dev Mailing List | User Mailing ListPySpark is the Python API for Apache Spark. It enables you to perform real-time,large-scale data processing in a distributed environment using Python. It also provides a PySparkshell for interactively analyzing your data.PySpark combines Pythons learnability and ease of use with the power of Apache Sparkto enable processing and analysis of data at any size for everyone familiar with Python.PySpark supports all of Sparks features such as Spark SQL,DataFrames, Structured Streaming, Machine Learning (MLlib) and Spark Core.Python Spark Connect ClientSpark Connect is a client-server architecture within Apache Spark thatenables remote connectivity to Spark clusters from any application.PySpark provides the client for the Spark Connect server, allowingSpark to be used as a service. Spark SQL and DataFramesSpark SQL is Apache Sparks module for working with structured data.It allows you to seamlessly mix SQL queries with Spark programs.With PySpark DataFrames you can efficiently read, write, transform,and analyze data using Python and SQL.Whether you use Python or SQL, the same underlying executionengine is used so you will always leverage the full power of Spark.Quickstart: DataFrameLive Notebook: DataFrameSpark SQL API ReferencePandas API on SparkPandas API on Spark allows you to scale your pandas workload to any sizeby running it distributed across multiple nodes. If you are already familiarwith pandas and want to leverage Spark for big data, pandas API on Spark makesyou immediately productive and lets you migrate your applications without modifying the code.You can have a single codebase that works both with pandas (tests, smaller datasets)and with Spark (production, distributed datasets) and you can switch between thepandas API and the Pandas API on Spark easily and without overhead.Pandas API on Spark aims to make the transition from pandas to Spark easy butif you are new to Spark or deciding which API to use, we recommend using PySpark(see Spark SQL and DataFrames). Structured StreamingStructured Streaming is a scalable and fault-tolerant stream processing engine built on the Spark SQL engine.You can express your streaming computation the same way you would express a batch computation on static data.The Spark SQL engine will take care of running it incrementally and continuously and updating the final resultas streaming data continues to arrive.Structured Streaming Programming GuideStructured Streaming API ReferenceMachine Learning (MLlib)Built on top of Spark, MLlib is a scalable machine learning library that providesa uniform set of high-level APIs that help users create and tune practical machinelearning pipelines. Spark Core and RDDsSpark Core is the underlying general execution engine for the Spark platform that allother functionality is built on top of. It provides RDDs (Resilient Distributed Datasets)and in-memory computing capabilities.Note that the RDD API is a low-level API which can be difficult to use and you do not getthe benefit of Sparks automatic query optimization capabilities.We recommend using DataFrames (see Spark SQL and DataFrames above)instead of RDDs as it allows you to express what you want more easily and lets Spark automaticallyconstruct the most efficient query for you. Spark Streaming (Legacy)Spark Streaming is an extension of the core Spark API that enables scalable,high-throughput, fault-tolerant stream processing of live data streams.Note that Spark Streaming is the previous generation of Sparks streaming engine.It is a legacy project and it is no longer being updated.There is a newer and easier to use streaming engine in Spark calledStructured Streaming which youshould use for your streaming applications and pipelines. Spark SQL is a Spark module for structured data processing. Unlike the basic Spark RDD API, the interfaces providedby Spark SQL provide Spark with more information about the structure of both the data and the computation being performed. Internally,Spark SQL uses this extra information to perform extra optimizations. There are several ways tointeract with Spark SQL including SQL and the Dataset API. When computing a result,the same execution engine is used, independent of which API/language you are using to express the computation. This unification means that developers can easily switch back and forth between different APIs based on which provides the most natural way to express a given transformation. All of the examples on this page use sample data included in the Spark distribution and can be run in the spark-shell, pyspark shell, or sparkR shell. SQL One use of Spark SQL is to execute SQL queries.Spark SQL can also be used to read data from an existing Hive installation. For more on how toconfigure this feature, please refer to the Hive Tables section. When runningSQL from within another programming language the results will be returned as a Dataset/DataFrame.You can also interact with the SQL interface using the command-line or over JDBC/ODBC. Datasets and DataFrames A Dataset is a distributed collection of data.Dataset is a new interface added in Spark 1.6 that provides the benefits of RDDs (strongtyping, ability to use powerful lambda functions) with the benefits of Spark SQLs optimizedexecution engine. A Dataset can be constructed from JVM objects and thenmanipulated using functional transformations (map, flatMap, filter, etc.).The Dataset API is available in Scala andJava. Python does not have the support for the Dataset API. But due to Pythons dynamic nature,many of the benefits of the Dataset API are already available (i.e. you can access the field of a row by name naturallyrow.columnName). The case for R is similar. A DataFrame is a Dataset organized into named columns. It is conceptuallyequivalent to a table in a relational database or a data frame in R/Python, but with richeroptimizations under the hood. DataFrames can be constructed from a wide array of sources suchas: structured data files, tables in Hive, external databases, or existing RDDs.The DataFrame API is available inPython, Scala,Java and R.In Scala and Java, a DataFrame is represented by a Dataset of Rows.In the Scala API, DataFrame is simply a type alias of Dataset[Row].While, in Java API, users need to use Dataset to represent a DataFrame. Throughout this document, we will often refer to Scala/Java Datasets of Rows as DataFrames. This tutorial provides a quick introduction to using Spark. We will first introduce the API through Sparksinteractive shell (in Python or Scala), then show how to write applications in Java, Scala, and Python. To follow along with this guide, first, download a packaged release of Spark from theSpark website. Since we wont be using HDFS,you can download a package for any version of Hadoop. Note that, before Spark 2.0, the main programming interface of Spark was the Resilient Distributed Dataset (RDD). After Spark 2.0, RDDs are replaced by Dataset, which is strongly-typed like an RDD, but with richer optimizations under the hood. The RDD interface is still supported, and you can get a more detailed reference at the RDD programming guide. However, we highly recommend you to switch to use Dataset, which has better performance than RDD. See the SQL programming guide to get more information about Dataset. Interactive Analysis with the Spark Shell Sparks shell provides a simple way to learn the API, as well as a powerful tool to analyze data interactively.It is available in either Scala (which runs on the Java VM and is thus a good way to use existing Java libraries)or Python. Start it by running the following in the Spark directory: Or if PySpark is installed with pip in your current environment: Sparks primary abstraction is a distributed collection of items called a Dataset. Datasets can be created from Hadoop InputFormats (such as HDFS files) or by transforming other Datasets. Due to Pythons dynamic nature, we dont need the Dataset to be strongly-typed in Python. As a result, all Datasets in Python are Dataset[Row], and we call it DataFrame to be consistent with the data frame concept in Pandas and R. Lets make a new DataFrame from the text of the README file in the Spark source directory: >>> textFile = spark.read.text("README.md") You can get values from DataFrame directly, by calling some actions, or transform the DataFrame to get a new one. For more details, please read the API doc. >>> textFile.count() # Number of rows in this DataFrame126 >>> textFile.first() # First row in this DataFrameRow(value=u'# Apache Spark') Now lets transform this DataFrame to a new one. We call filter to return a new DataFrame with a subset of the lines in the file. >>> linesWithSpark = textFile.filter(textFile.value.contains("Spark")) We can chain together transformations and actions: >>> textFile.filter(textFile.value.contains("Spark")).count() # How many lines contain "Spark"?15 Sparks primary abstraction is a distributed collection of items called a Dataset. Datasets can be created from Hadoop InputFormats (such as HDFS files) or by transforming other Datasets. Lets make a new Dataset from the text of the README file in the Spark source directory: scala> val textFile = spark.read.textFile("README.md")textFile: org.apache.spark.sql.Dataset[String] = [value: string] You can get values from Dataset directly, by calling some actions, or transform the Dataset to get a new one. For more details, please read the API doc. scala> textFile.count() // Number of items in this Datasetres0: Long = 126 // May be different from yours as README.md will change over time, similar to other outputs scala> textFile.first() // First item in this Datasetres1: String = # Apache Spark Now lets transform this Dataset into a new one. We call filter to return a new Dataset with a subset of the items in the file. scala> val linesWithSpark = textFile.filter(line => line.contains("Spark"))linesWithSpark: org.apache.spark.sql.Dataset[String] = [value: string] We can chain together transformations and actions: scala> textFile.filter(line => line.contains("Spark")).count() // How many lines contain "Spark"?res3: Long = 15 More on Dataset Operations Dataset actions and transformations can be used for more complex computations. Lets say we want to find the line with the most words: >>> from pyspark.sql import functions as sf>>> textFile.select(sf.size(sf.split(textFile.value, "\s+")).name("numWords")).agg(sf.max(sf.col("numWords"))).collect()[Row(max(numWords)=15)] This first maps a line to an integer value and aliases it as numWords, creating a new DataFrame. agg is called on that DataFrame to find the largest word count. The arguments to select and agg are both Column, we can use df.colName to get a column from a DataFrame. We can also import pyspark.sql.functions, which provides a lot of convenient functions to build a new Column from an old one. One common data flow pattern is MapReduce, as popularized by Hadoop. Spark can implement MapReduce flows easily: >>> wordCounts = textFile.select(sf.explode(sf.split(textFile.value, "\s+")).alias("word")).groupBy("word").count() Here, we use the explode function in select, to transform a Dataset of lines to a Dataset of words, and then combine groupBy and count to compute the per-word counts in the file as a DataFrame of 2 columns: word and count. To collect the word counts in our shell, we can call collect: >>> wordCounts.collect()[Row(word=u'online', count=1), Row(word=u'graphs', count=1), ...] scala> textFile.map(line => line.split(" ").size).reduce((a, b) => if (a > b) a else b)res4: Int = 15 This first maps a line to an integer value, creating a new Dataset. reduce is called on that Dataset to find the largest word count. The arguments to map and reduce are Scala function literals (closures), and can use any language feature or Scala/Java library. For example, we can easily call functions declared elsewhere. Well use Math.max() function to make this code easier to understand: scala> import java.lang.Mathimport java.lang.Math scala> textFile.map(line => line.split(" ").size).reduce((a, b) => Math.max(a, b))res5: Int = 15 One common data flow pattern is MapReduce, as popularized by Hadoop. Spark can implement MapReduce flows easily: scala> val wordCounts = textFile.flatMap(line => line.split(" ")).groupByKey(identity).count()wordCounts: org.apache.spark.sql.Dataset[(String, Long)] = [value: string, count(1): bigint] Here, we call flatMap to transform a Dataset of lines to a Dataset of words, and then combine groupByKey and count to compute the per-word counts in the file as a Dataset of (String, Long) pairs. To collect the word counts in our shell, we can call collect: scala> wordCounts.collect()res6: Array[(String, Int)] = Array((means,1), (under,2), (this,3), (Because,1), (Python,2), (agree,1), (cluster.,1), ...) CachingSpark also supports pulling data sets into a cluster-wide in-memory cache. This is very useful when data is accessed repeatedly, such as when querying a small hot dataset or when running an iterative algorithm like PageRank. As a simple example, lets mark our linesWithSpark dataset to be cached: >>> linesWithSpark.cache() >>> linesWithSpark.count()15 >>> linesWithSpark.count()15 It may seem silly to use Spark to explore and cache a 100-line text file. The interesting part isthat these same functions can be used on very large data sets, even when they are striped acrosstens or hundreds of nodes. You can also do this interactively by connecting bin/pyspark toa cluster, as described in the RDD programming guide. scala> linesWithSpark.cache()res7: linesWithSpark.type = [value: string] scala> linesWithSpark.count()res8: Long = 15 scala> linesWithSpark.count()res9: Long = 15 It may seem silly to use Spark to explore and cache a 100-line text file. The interesting part isthat these same functions can be used on very large data sets, even when they are striped acrosstens or hundreds of nodes. You can also do this interactively by connecting bin/spark-shell toa cluster, as described in the RDD programming guide. Self-Contained ApplicationsSuppose we wish to write a self-contained application using the Spark API. We will walk through asimple application in Scala (with sbt), Java (with Maven), and Python (pip). Now we will show how to write an application using the Python API (PySpark). If you are building a packaged PySpark application or library you can add it to your setup.py file as: install_requires=['pyspark==4.0.0'] As an example, well create a simple Spark application, SimpleApp.py: """SimpleApp.py"""from pyspark.sql import SparkSession logFile = "YOUR_SPARK_HOME/README.md" # Should be some file on your systemspark = SparkSession.builder.appName("SimpleApp").getOrCreate()logData = spark.read.text(logFile).cache() numAs = logData.filter(logData.value.contains('a')).count()numBs = logData.filter(logData.value.contains('b')).count() print("Lines with a: %i, lines with b: %i" % (numAs, numBs)) spark.stop() This program just counts the number of lines containing a and the number containing b in atext file.Note that youll need to replace YOUR_SPARK_HOME with the location where Spark is installed.As with the Scala and Java examples, we use a SparkSession to create Datasets.For applications that use custom classes or third-party libraries, we can also add codedependencies to spark-submit through its --py-files argument by packaging them into a .zip file (see spark-submit --help for details).SimpleApp is simple enough that we do not need to specify any code dependencies. We can run this application using the bin/spark-submit script: # Use spark-submit to run your application$ YOUR_SPARK_HOME/bin/spark-submit \ --master "local[4]" \ SimpleApp.py...Lines with a: 46, Lines with b: 23 If you have PySpark pip installed into your environment (e.g., pip install pyspark), you can run your application with the regular Python interpreter or use the provided spark-submit as you prefer. # Use the Python interpreter to run your application$ python SimpleApp.py...Lines with a: 46, Lines with b: 23 Well create a very simple Spark application in Scalaso simple, in fact, that itsnamed SimpleApp.scala: /* SimpleApp.scala */import org.apache.spark.sql.SparkSession object SimpleApp { def main(args: Array[String]): Unit = { val logFile = "YOUR_SPARK_HOME/README.md" // Should be some file on your system val spark = SparkSession.builder.appName("Simple Application").getOrCreate() val logData = spark.read.textFile(logFile).cache() val numAs = logData.filter(line => line.contains("a")).count() val numBs = logData.filter(line => line.contains("b")).count() println(s"Lines with a: $numAs, Lines with b: $numBs") spark.stop() } } Note that applications should define a main() method instead of extending scala.App.Subclasses of scala.App may not work correctly. This program just counts the number of lines containing a and the number containing b in theSpark README. Note that youll need to replace YOUR_SPARK_HOME with the location where Spark isinstalled. Unlike the earlier examples with the Spark shell, which initializes its own SparkSession,we initialize a SparkSession as part of the program. We call SparkSession.builder to construct a SparkSession, then set the application name, and finally call getOrCreate to get the SparkSession instance. Our application depends on the Spark API, so well also include an sbt configuration file,build.sbt, which explains that Spark is a dependency. This file also adds a repository thatSpark depends on: name := "Simple Project" version := "1.0" scalaVersion := "2.13.16" libraryDependencies += "org.apache.spark" %% "spark-sql" % "4.0.0" For sbt to work correctly, well need to layout SimpleApp.scala and build.sbtaccording to the typical directory structure. Once that is in place, we can create a JAR packagecontaining the applications code, then use the spark-submit script to run our program. # Your directory layout should look like this$ find .../build.sbt./src./src/main./src/main/scala./src/main/scala/SimpleApp.scala # Package a jar containing your application$ sbt package...[info] Packaging {..}/{..}/target/scala-2.13/simple-project_2.13-1.0.jar # Use spark-submit to run your application$ YOUR_SPARK_HOME/bin/spark-submit \ --class "SimpleApp" \ --master "local[4]" \ target/scala-2.13/simple-project_2.13-1.0.jar...Lines with a: 46, Lines with b: 23 This example will use Maven to compile an application JAR, but any similar build system will work. Well create a very simple Spark application, SimpleApp.java: /* SimpleApp.java */import org.apache.spark.sql.SparkSession;import org.apache.spark.sql.Dataset; public class SimpleApp { public static void main(String[] args) { String logFile = "YOUR_SPARK_HOME/README.md"; // Should be some file on your system SparkSession spark = SparkSession.builder().appName("Simple Application").getOrCreate(); Dataset logData = spark.read().textFile(logFile).cache(); long numAs = logData.filter(s -> s.contains("a")).count(); long numBs = logData.filter(s -> s.contains("b")).count(); System.out.println("Lines with a: " + numAs + ", lines with b: " + numBs); spark.stop(); } } This program just counts the number of lines containing a and the number containing b in theSpark README. Note that youll need to replace YOUR_SPARK_HOME with the location where Spark isinstalled. Unlike the earlier examples with the Spark shell, which initializes its own SparkSession,we initialize a SparkSession as part of the program. To build the program, we also write a Maven pom.xml file that lists Spark as a dependency.Note that Spark artifacts are tagged with a Scala version. edu.berkeley simple-project 4.0.0 Simple Project jar 1.0 org.apache.spark spark-sql_2.13 4.0.0 provided We lay out these files according to the canonical Maven directory structure: $ find ../pom.xml./src./src/main./src/main/java./src/main/java/SimpleApp.java Now, we can package the application using Maven and execute it with ./bin/spark-submit. # Package a JAR containing your application$ mvn package...[INFO] Building jar: {..}/{..}/target/simple-project-1.0.jar # Use spark-submit to run your application$ YOUR_SPARK_HOME/bin/spark-submit \ --class "SimpleApp" \ --master "local[4]" \ target/simple-project-1.0.jar...Lines with a: 46, Lines with b: 23 Other dependency management tools such as Conda and pip can be also used for custom classes or third-party libraries. See also Python Package Management. Where to Go from HereCongratulations on running your first Spark application! # For Python examples, use spark-submit directly:./bin/spark-submit examples/src/main/python/pi.py # For Scala and Java, use run-example:./bin/run-example SparkPi # For R examples, use spark-submit directly:./bin/spark-submit examples/src/main/r/dataframe.R Apache Software Foundation Apache Homepage License Sponsorship Thanks Event

Spark-sql examples. Spark sql query.