

**MATLAB® Compiler™**

Spark™ Integration



**MATLAB®**

R2020a



# How to Contact MathWorks



Latest news: [www.mathworks.com](http://www.mathworks.com)  
Sales and services: [www.mathworks.com/sales\\_and\\_services](http://www.mathworks.com/sales_and_services)  
User community: [www.mathworks.com/matlabcentral](http://www.mathworks.com/matlabcentral)  
Technical support: [www.mathworks.com/support/contact\\_us](http://www.mathworks.com/support/contact_us)



Phone: 508-647-7000



The MathWorks, Inc.  
1 Apple Hill Drive  
Natick, MA 01760-2098

*MATLAB® Compiler™ Spark™ Integration*

© COPYRIGHT 2016–2020 by The MathWorks, Inc.

The software described in this document is furnished under a license agreement. The software may be used or copied only under the terms of the license agreement. No part of this manual may be photocopied or reproduced in any form without prior written consent from The MathWorks, Inc.

FEDERAL ACQUISITION: This provision applies to all acquisitions of the Program and Documentation by, for, or through the federal government of the United States. By accepting delivery of the Program or Documentation, the government hereby agrees that this software or documentation qualifies as commercial computer software or commercial computer software documentation as such terms are used or defined in FAR 12.212, DFARS Part 227.72, and DFARS 252.227-7014. Accordingly, the terms and conditions of this Agreement and only those rights specified in this Agreement, shall pertain to and govern the use, modification, reproduction, release, performance, display, and disclosure of the Program and Documentation by the federal government (or other entity acquiring for or through the federal government) and shall supersede any conflicting contractual terms or conditions. If this License fails to meet the government's needs or is inconsistent in any respect with federal procurement law, the government agrees to return the Program and Documentation, unused, to The MathWorks, Inc.

## Trademarks

MATLAB and Simulink are registered trademarks of The MathWorks, Inc. See [www.mathworks.com/trademarks](http://www.mathworks.com/trademarks) for a list of additional trademarks. Other product or brand names may be trademarks or registered trademarks of their respective holders.

## Patents

MathWorks products are protected by one or more U.S. patents. Please see [www.mathworks.com/patents](http://www.mathworks.com/patents) for more information.

## Revision History

September 2016	Online only	New for Version 6.3 (Release 2016b)
March 2017	Online only	Revised for Version 6.4 (Release R2017a)
September 2017	Online only	Revised for Version 6.5 (Release R2017b)
March 2018	Online only	Revised for Version 6.6 (Release R2018a)
September 2018	Online only	Revised for Version 7.0 (Release R2018b)
March 2019	Online only	Revised for Version 7.0.1 (Release R2019a)
September 2019	Online only	Revised for Version 7.1 (Release R2019b)
March 2020	Online only	Revised for Version 8.0 (Release R2020a)

<b>1</b>	<b>Classes— MATLAB API for Spark</b>	
<b>2</b>	<b>Methods – RDD</b>	
<b>3</b>	<b>Methods – SparkContext</b>	
<b>4</b>	<b>Apache Spark Basics</b>	
	<b>Apache Spark Basics</b> .....	<b>4-2</b>
	Running against Spark .....	<b>4-2</b>
	Cluster Managers Supported by Spark .....	<b>4-3</b>
	Relationship Between Spark and Hadoop .....	<b>4-4</b>
	Driver .....	<b>4-4</b>
	Executor .....	<b>4-5</b>
	RDD .....	<b>4-5</b>
	Transformations .....	<b>4-5</b>
	Actions .....	<b>4-5</b>
	Distinguishing Between Transformations and Actions .....	<b>4-5</b>
	SparkConf .....	<b>4-5</b>
	SparkContext .....	<b>4-6</b>
<b>5</b>	<b>Configure MATLAB Environment for Spark Deployment</b>	
	<b>Configure Environment for Interactive Debugging</b> .....	<b>5-2</b>

## Deploy Tall Arrays to a Spark enabled Hadoop Cluster

### 6

<b>Example on Deploying Tall Arrays to a Spark Enabled Hadoop Cluster</b>	<b>6-2</b>
.....	
<b>Deploying Tall Arrays to a Cloudera Spark Enabled Hadoop Cluster</b> . . . .	<b>6-8</b>
Prerequisites . . . . .	<b>6-9</b>
Deploy Tall Arrays . . . . .	<b>6-9</b>

## Deploy MATLAB Applications to Spark using the MATLAB API for Spark

### 7

<b>Example on Deploying Applications to Spark Using the MATLAB API for Spark</b>	<b>7-2</b>
Local . . . . .	<b>7-2</b>
Hadoop YARN . . . . .	<b>7-6</b>
<b>Deploying Applications to Cloudera Spark Using the MATLAB API for Spark</b>	<b>7-11</b>
Prerequisites . . . . .	<b>7-11</b>
Deploy Applications to Cloudera Spark . . . . .	<b>7-12</b>

# Classes— MATLAB API for Spark

---

## matlab.compiler.mlspark.RDD class

**Package:** matlab.compiler.mlspark

Interface class to represent a Spark Resilient Distributed Dataset (RDD)

### Description

A Resilient Distributed Dataset or RDD is a programming abstraction in Spark. It represents a collection of elements distributed across many nodes that can be operated in parallel. All work in Spark is expressed as either creating new RDDs, transforming existing RDDs, or calling operations on RDDs to compute a result. You can create RDDs in two ways:

- By loading an external dataset
- By parallelizing a collection of objects in the *driver* on page 4-4 program

Once created, two types of operations can be performed using RDDs: *transformations* on page 1-3 and *actions* on page 1-4.

### Construction

An RDD object can only be created using the methods of the `SparkContext` class. A collection of `SparkContext` methods used to create RDDs is listed below for convenience. See the documentation of the `SparkContext` on page 1-13 class for more information.

SparkContext Method Name	Purpose
parallelize on page 3-9	Create an RDD from local MATLAB values
datastoreToRDD on page 3-6	Convert MATLAB datastore to a Spark RDD
textFile on page 3-14	Create an RDD from a text file

Once an RDD has been created using a method from the `SparkContext` class, you can use any of the methods in the RDD class to manipulate your RDD.

### Properties

The properties of this class are hidden.

### Methods

## Transformations

<code>aggregateByKey</code>	Aggregate the values of each key, using given combine functions and a neutral “zero value”
<code>cartesian</code>	Create an RDD that is the Cartesian product of two RDDs
<code>coalesce</code>	Reduce the number of partitions in an RDD
<code>cogroup</code>	Group data from RDDs sharing the same key
<code>combineByKey</code>	Combine the elements for each key using a custom set of aggregation functions
<code>distinct</code>	Return a new RDD containing the distinct elements of an existing RDD
<code>filter</code>	Return a new RDD containing only the elements that satisfy a predicate function
<code>flatMap</code>	Return a new RDD by first applying a function to all elements of an existing RDD, and then flattening the results
<code>flatMapValues</code>	Pass each value in the key-value pair RDD through a <code>flatMap</code> method without changing the keys
<code>foldByKey</code>	Merge the values for each key using an associative function and a neutral “zero value”
<code>fullOuterJoin</code>	Perform a full outer join between two key-value pair RDDs
<code>glom</code>	Coalesce all elements within each partition of an RDD
<code>groupBy</code>	Return an RDD of grouped items
<code>groupByKey</code>	Group the values for each key in the RDD into a single sequence
<code>intersection</code>	Return the set intersection of one RDD with another
<code>join</code>	Return an RDD containing all pairs of elements with matching keys
<code>keyBy</code>	Create tuples of the elements in an RDD by applying a function
<code>keys</code>	Return an RDD with the keys of each tuple
<code>leftOuterJoin</code>	Perform a left outer join
<code>map</code>	Return a new RDD by applying a function to each element of an input RDD
<code>mapValues</code>	Pass each value in a key-value pair RDD through a map function without modifying the keys
<code>reduceByKey</code>	Merge the values for each key using an associative reduce function
<code>repartition</code>	Return a new RDD that has exactly <code>numPartitions</code> partitions
<code>rightOuterJoin</code>	Perform a right outer join
<code>sortBy</code>	Sort an RDD by a given function
<code>sortByKey</code>	Sort RDD consisting of key-value pairs by key
<code>subtract</code>	Return the values resulting from the set difference between two RDDs
<code>subtractByKey</code>	Return key-value pairs resulting from the set difference of keys between two RDDs
<code>union</code>	Return the set union of one RDD with another
<code>values</code>	Return an RDD with the values of each tuple
<code>zip</code>	Zip one RDD with another
<code>zipWithIndex</code>	Zip an RDD with its element indices
<code>zipWithUniqueId</code>	Zip an RDD with generated unique Long IDs

## Actions

aggregate	Aggregate the elements of each partition and subsequently the results for all partitions into a single value
collect	Return a MATLAB cell array that contains all of the elements in an RDD
collectAsMap	Return the key-value pairs in an RDD as a MATLAB <code>containers.Map</code> object
count	Count number of elements in an RDD
fold	Aggregate elements of each partition and the subsequent results for all partitions
reduce	Reduce elements of an RDD using the specified commutative and associative function
reduceByKeyLocally	Merge the values for each key using an associative reduce function, but return the results immediately to the driver
saveAsTallDatastore	Save RDD as a MATLAB tall array to a binary file that can be read back using the <code>datastore</code> function
saveAsKeyValueDatastore	Save key-value RDD as a binary file that can be read back using the <code>datastore</code> function
saveAsTextFile	Save RDD as a text file

## Operations

cache	Store an RDD in memory
checkpoint	Mark an RDD for checkpointing
getCheckpointFile	Get the name of the file to which an RDD is checkpointed
getDefaultReducePartitions	Get the number of default reduce partitions in an RDD
getNumPartitions	Return the number of partitions in an RDD
isEmpty	Determine if an RDD contains any elements
keyLimit	Return threshold of unique keys that can be stored before spilling to disk
persist	Set the value of an RDD's storage level to persist across operations after it is computed
toDebugString	Obtain a description of an RDD and its recursive dependencies for debugging
unpersist	Mark an RDD as nonpersistent, remove all blocks for it from memory and disk

## More About

### Resilient Distributed Dataset

A Resilient Distributed Dataset or RDD is a programming abstraction in Spark. It represents a collection of elements distributed across many nodes that can be operated in parallel. RDDs tend to be fault-tolerant. You can create RDDs in two ways:

- By loading an external dataset.



- By parallelizing a collection of objects in the *driver* on page 4-4 program.

After creation, you can perform two types of operations using RDDs: transformations and actions.

### **Transformations**

Transformations are operations on an existing RDD that return a new RDD. Many, but not all, transformations are element-wise operations.

### **Actions**

Actions compute a final result based on an RDD and either return that result to the driver program or save it to an external storage system such as HDFS™.

### **References**

See the latest Spark documentation for more information.

### **See Also**

#### **Classes**

`matlab.compiler.mlspark.SparkConf` | `matlab.compiler.mlspark.SparkContext`

#### **Topics**

“Example on Deploying Applications to Spark Using the MATLAB API for Spark” on page 7-2  
“Apache Spark Basics” on page 4-2

### **Introduced in R2016b**

## matlab.compiler.mlspark.SparkConf class

**Package:** matlab.compiler.mlspark

Interface class to configure an application with Spark parameters as key-value pairs

### Description

A SparkConf object stores the configuration parameters of the application being deployed to Spark. Every application must be configured prior to deployment on a Spark cluster. The configuration parameters are passed onto a Spark cluster through a SparkContext.

### Construction

`conf = matlab.compiler.mlspark.SparkConf('AppName', name, 'Master', url, 'SparkProperties', prop)` creates a SparkConf object with the specified configuration parameters.

`conf = matlab.compiler.mlspark.SparkConf( ____, Name, Value)` creates a SparkConf object with additional configuration parameters specified by one or more Name, Value pair arguments. Name is a property name of the class and Value is the corresponding value. Name must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as Name1, Value1, . . . , NameN, ValueN.

### Input Arguments

**name — Name of the MATLAB application deployed to Spark**

character vector | string

Name of application specified as a character vector inside single quotes ( ' ').

Example: 'AppName', 'myApp'

Data Types: char | string

**url — Master URL to connect to**

character vector | string

Name of the master URL specified as a character vector inside single quotes ( ' ').

URL	Description
local	Run Spark locally with one worker thread. There is no parallelism by selecting this option.
local[K]	Run Spark locally with K worker threads. Set K to the number of cores on your machine.
local[*]	Run Spark locally with as many worker threads as logical cores on your machine.
yarn-client	Connect to a Hadoop® YARN cluster in client mode. The cluster location is found based on the HADOOP_CONF_DIR or YARN_CONF_DIR variable.

Example: 'Master', 'yarn-client'

Data Types: char | string

**prop — Map of key-value pairs that specify Spark configuration properties**

containers.Map object

A containers.Map object containing Spark configuration properties as key-value pairs.

---

**Note** When deploying to a local cluster using the MATLAB API for Spark, the 'SparkProperties' property name can be ignored during the construction of a SparkConf object, thereby requiring no value for prop. Or you can set prop to an empty containers.Map object as follows:

```
'SparkProperties',containers.Map({' ',' '})
```

The key and value of the containers.Map object are empty char vectors.

---

When deploying to a Hadoop YARN cluster, set the value for prop with the appropriate Spark configuration properties as key-value pairs. The precise set of Spark configuration properties vary from one deployment scenario to another, based on the deployment cluster environment. Users must verify the Spark setup with a system administrator to use the appropriate configuration properties. See the table for commonly used Spark properties. For a full set of properties, see the latest Spark documentation.

### Running Spark on YARN

Property Name (Key)	Default (Value)	Description
<code>spark.executor.cores</code>	1	<p>The number of cores to use on each executor.</p> <p>For YARN and Spark standalone mode only. In Spark standalone mode, setting this parameter allows an application to run multiple executors on the same worker, provided that there are enough cores on that worker. Otherwise, only one executor per application runs on each worker.</p>
<code>spark.executor.instances</code>	2	<p>The number of executors.</p> <hr/> <p><b>Note</b> This property is incompatible with <code>spark.dynamicAllocation.enabled</code>. If both <code>spark.dynamicAllocation.enabled</code> and <code>spark.executor.instances</code> are specified, dynamic allocation is turned off and the specified number of <code>spark.executor.instances</code> is used.</p>
<code>spark.driver.memory</code>	<ul style="list-style-type: none"> <li>1g</li> <li>2048m (<i>recommended</i>)</li> </ul>	<p>Amount of memory to use for the driver process.</p> <p>If you get any out of memory errors while using <code>take/gather</code>, consider increasing this value.</p>
<code>spark.executor.memory</code>	<ul style="list-style-type: none"> <li>1g</li> <li>2048m (<i>recommended</i>)</li> </ul>	<p>Amount of memory to use per executor process.</p> <p>If you get any out of memory errors while using <code>take/gather</code>, consider increasing this value.</p>

Property Name (Key)	Default (Value)	Description
<code>spark.yarn.executor.memoryOverhead</code>	<ul style="list-style-type: none"> <li><code>executorMemory * 0.10</code>, with minimum of 384.</li> <li>4096m (<i>recommended</i>)</li> </ul>	<p>The amount of off-heap memory (in MBs) to be allocated per executor.</p> <p>If you get any out of memory errors while using <code>tall/gather</code>, consider increasing this value.</p>
<code>spark.dynamicAllocation.enabled</code>	false	<p>This option integrates Spark with the YARN resource management. Spark initiates as many executors as possible given the executor memory requirement and number of cores. This property requires that the cluster be set up.</p> <p>Setting this property to <code>true</code> specifies whether to use dynamic resource allocation, which scales the number of executors registered with this application up and down based on the workload.</p> <p>This property requires <code>spark.shuffle.service.enabled</code> to be set. The following configurations are also relevant: <code>spark.dynamicAllocation.minExecutors</code>, <code>spark.dynamicAllocation.maxExecutors</code>, and <code>spark.dynamicAllocation.initialExecutors</code></p>
<code>spark.shuffle.service.enabled</code>	false	<p>Enables the external shuffle service. This service preserves the shuffle files written by executors so the executors can be safely removed. This must be enabled if <code>spark.dynamicAllocation.enabled</code> is set to <code>true</code>. The external shuffle service must be set up in order to enable it.</p>

**MATLAB Specific Properties**

Property Name (Key)	Default (Value)	Description
spark.matlab.worker.debug	false	For use in standalone/ interactive mode only. If set to true, a Spark deployable MATLAB application executed within the MATLAB desktop environment, starts another MATLAB session as worker, and will enter the debugger. Logging information is directed to log_<nbr>.txt.
spark.matlab.worker.reuse	true	When set to true, a Spark executor pools workers and reuses them from one stage to the next. Workers terminate when the executor under which the workers are running terminates.
spark.matlab.worker.profile	false	Only valid when using a session of MATLAB as a worker. When set to true, it turns on the MATLAB Profiler and generates a Profile report that is saved to the file profworker_<split_index>_<socket>_<worker pass>.mat.
spark.matlab.worker.numberOfKeys	10000	Number of unique keys that can be held in a containers.Map object while performing *ByKey operations before map data is spilled to a file.
spark.matlab.executor.timeout	600000	Spark executor timeout in milliseconds. Not applicable when deploying tall arrays.

## Monitoring and Logging

Property Name (Key)	Default (Value)	Description
spark.history.fs.logDirectory	file:/tmp/spark-events	Directory that contains application event logs to be loaded by the history server.
spark.eventLog.dir	file:///tmp/spark-events	Base directory in which Spark events are logged, if <code>spark.eventLog.enabled</code> is <code>true</code> . Within this base directory, Spark creates a sub directory for each application, and logs the events specific to the application in this directory. You can set this to a unified location like an HDFS directory so history files can be read by the history server.
spark.eventLog.enabled	false	Whether to log Spark events. This is useful for reconstructing the web UI after the application has finished.

Data Types: char

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### ExecutorEnv — Map of key-value pairs that will be used to establish the executor environment

containers.Map object

Map of key-value pairs specified as a `containers.Map` object.

Example: 'ExecutorEnv', `containers.Map({'SPARK_JAVA_OPTS'}, {'-Djava.library.path=/my/custom/path'})`

### MCRRoot — Path to MATLAB Runtime that is used to execute driver application

character vector | string

A character vector specifying the path to MATLAB Runtime within single quotes `' '`.

Example: 'MCRRoot',  `'/share/MATLAB/MATLAB_Runtime/v91'`

Data Types: char | string

## Properties

The properties of this class are hidden.

## Methods

There are no user executable methods for this class.

## Examples

### Configure an Application With Spark Parameters

The `SparkConf` class allows you to configure an application with Spark parameters as key-value pairs.

```
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});  
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...  
    'Master','local[1]', 'SparkProperties',sparkProp);
```

## More About

### SparkConf

`SparkConf` stores the configuration parameters of the application being deployed to Spark. Every application must be configured prior to being deployed on a Spark cluster. Some of the configuration parameters define properties of the application and some are used by Spark to allocate resources on the cluster. The configuration parameters are passed onto a Spark cluster through a `SparkContext`.

## References

See the latest Spark documentation for more information.

## See Also

### Classes

`matlab.compiler.mlspark.RDD` | `matlab.compiler.mlspark.SparkContext`

### Topics

“Example on Deploying Applications to Spark Using the MATLAB API for Spark” on page 7-2

“Apache Spark Basics” on page 4-2

### Introduced in R2016b



# matlab.compiler.mlspark.SparkContext class

**Package:** matlab.compiler.mlspark

Interface class to initialize a connection to a Spark enabled cluster

## Description

A `SparkContext` object serves as an entry point to Spark by initializing a connection to a Spark cluster. It accepts a `SparkConf` object as an input argument and uses the parameters specified in that object to set up the internal services necessary to establish a connection to the Spark execution environment.

## Construction

`sc = matlab.compiler.mlspark.SparkContext(conf)` creates a `SparkContext` object initializes a connection to a Spark cluster.

## Input Arguments

**conf** — Variable name representing a `SparkConf` object

`SparkConf` object

Pass the `SparkConf` object as input to the `SparkContext`.

Example: `sc = matlab.compiler.mlspark.SparkContext(conf);`

See `matlab.compiler.mlspark.SparkConf` for information on how to create a `SparkConf` object.

## Properties

The properties of this class are hidden.

## Methods

<code>addJar</code>	Add JAR file dependency for all tasks that need to be executed in a <code>SparkContext</code>
<code>broadcast</code>	Broadcast a read-only variable to the cluster
<code>delete</code>	Shutdown connection to Spark enabled cluster
<code>datastoreToRDD</code>	Convert MATLAB <code>datastore</code> to a Spark RDD
<code>getSparkConf</code>	Get <code>SparkConf</code> configuration parameters
<code>parallelize</code>	Create an RDD from a collection of local MATLAB values
<code>setCheckpointDir</code>	Set the directory under which RDDs are to be checkpointed
<code>setLogLevel</code>	Set log level
<code>textFile</code>	Create an RDD from a text file

## Examples

### Initialize a Connection to a Spark Enabled Cluster

The `SparkContext` class initializes a connection to a Spark enabled cluster using Spark properties.

```
% Setup Spark Properties as a containers.Map object
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});

% Create SparkConf object
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);

% Create a SparkContext
sc = matlab.compiler.mlspark.SparkContext(conf);
```

## More About

### SparkContext

A `SparkContext` represents a connection to a Spark cluster. It is the entry point to Spark and sets up the internal services necessary to establish a connection to the Spark execution environment.

## References

See the latest Spark documentation for more information.

## See Also

### Classes

`matlab.compiler.mlspark.RDD` | `matlab.compiler.mlspark.SparkConf`

### Topics

“Example on Deploying Applications to Spark Using the MATLAB API for Spark” on page 7-2  
“Apache Spark Basics” on page 4-2

### Introduced in R2016b

# matlab.mapreduce.DeploySparkMapReducer class

**Package:** matlab.mapreduce

Configure a MATLAB tall array application with Spark parameters as key-value pairs

## Description

A `DeploySparkMapReducer` object stores the configuration parameters of the tall array application being deployed to Spark. Every tall array application must be configured prior to being deployed on a Spark cluster. Some of the configuration parameters define properties of the application and some are used by Spark to allocate resources on the cluster. The configuration parameters are passed onto a Spark cluster through a `mapreducer` function.

## Construction

`conf = matlab.mapreduce.DeploySparkMapReducer('AppName', name, 'Master', url, 'SparkProperties', prop)` creates a `DeploySparkMapReducer` object with the specified configuration parameters.

`conf = matlab.mapreduce.DeploySparkMapReducer('AppName', name, 'Master', url, 'SparkProperties', prop, Name, Value)` creates a `DeploySparkMapReducer` object with additional configuration parameters specified by one or more `Name, Value` pair arguments. `Name` is a property name of the class and `Value` is the corresponding value. `Name` must appear inside single quotes ( ' '). You can specify several name-value pair arguments in any order as `Name1, Value1, ..., NameN, ValueN`.

## Input Arguments

**name** — Name of the MATLAB application deployed to Spark

character vector | string

Name of application specified as a character vector inside single quotes ( ' ' ).

Example: 'AppName', 'myApp'

Data Types: char | string

**url** — Master URL to connect to

character vector | string

Name of the master URL specified as a character vector inside single quotes ( ' ' ).

URL	Description
yarn-client	Connect to a Hadoop YARN cluster in client mode. The cluster location is found based on the HADOOP_CONF_DIR or YARN_CONF_DIR variable.

Example: 'Master', 'yarn-client'

Data Types: char | string

**prop — Map of key-value pairs that specify Spark configuration properties**

`containers.Map` object

A `containers.Map` object containing Spark configuration properties as key-value pairs.

When deploying to a Hadoop YARN cluster, set the value for `prop` with the appropriate Spark configuration properties as key-value pairs. The precise set of Spark configuration properties vary from one deployment scenario to another, based on the deployment cluster environment. Users must verify the Spark setup with a system administrator to use the appropriate configuration properties. See the table for commonly used Spark properties. For a full set of properties, see the latest Spark documentation.

**Running Spark on YARN**

Property Name (Key)	Default (Value)	Description
spark.executor.cores	1	<p>The number of cores to use on each executor.</p> <p>For YARN and Spark standalone mode only. In Spark standalone mode, setting this parameter allows an application to run multiple executors on the same worker, provided that there are enough cores on that worker. Otherwise, only one executor per application runs on each worker.</p>
spark.executor.instances	2	<p>The number of executors.</p> <hr/> <p><b>Note</b> This property is incompatible with spark.dynamicAllocation.enabled. If both spark.dynamicAllocation.enabled and spark.executor.instances are specified, dynamic allocation is turned off and the specified number of spark.executor.instances is used.</p>
spark.driver.memory	<ul style="list-style-type: none"> <li>1g</li> <li>2048m (<i>recommended</i>)</li> </ul>	<p>Amount of memory to use for the driver process.</p> <p>If you get any out of memory errors while using <code>tail/gather</code>, consider increasing this value.</p>
spark.executor.memory	<ul style="list-style-type: none"> <li>1g</li> <li>2048m (<i>recommended</i>)</li> </ul>	<p>Amount of memory to use per executor process.</p> <p>If you get any out of memory errors while using <code>tail/gather</code>, consider increasing this value.</p>

Property Name (Key)	Default (Value)	Description
<code>spark.yarn.executor.memoryOverhead</code>	<ul style="list-style-type: none"> <li><code>executorMemory * 0.10</code>, with minimum of 384.</li> <li>4096m (<i>recommended</i>)</li> </ul>	<p>The amount of off-heap memory (in MBs) to be allocated per executor.</p> <p>If you get any out of memory errors while using <code>tall/gather</code>, consider increasing this value.</p>
<code>spark.dynamicAllocation.enabled</code>	false	<p>This option integrates Spark with the YARN resource management. Spark initiates as many executors as possible given the executor memory requirement and number of cores. This property requires that the cluster be set up.</p> <p>Setting this property to <code>true</code> specifies whether to use dynamic resource allocation, which scales the number of executors registered with this application up and down based on the workload.</p> <p>This property requires <code>spark.shuffle.service.enabled</code> to be set. The following configurations are also relevant: <code>spark.dynamicAllocation.minExecutors</code>, <code>spark.dynamicAllocation.maxExecutors</code>, and <code>spark.dynamicAllocation.initialExecutors</code></p>
<code>spark.shuffle.service.enabled</code>	false	<p>Enables the external shuffle service. This service preserves the shuffle files written by executors so the executors can be safely removed. This must be enabled if <code>spark.dynamicAllocation.enabled</code> is set to <code>true</code>. The external shuffle service must be set up in order to enable it.</p>

**MATLAB Specific Properties**

Property Name (Key)	Default (Value)	Description
spark.matlab.worker.debug	false	For use in standalone/ interactive mode only. If set to true, a Spark deployable MATLAB application executed within the MATLAB desktop environment, starts another MATLAB session as worker, and will enter the debugger. Logging information is directed to log_<nbr>.txt.
spark.matlab.worker.reuse	true	When set to true, a Spark executor pools workers and reuses them from one stage to the next. Workers terminate when the executor under which the workers are running terminates.
spark.matlab.worker.profile	false	Only valid when using a session of MATLAB as a worker. When set to true, it turns on the MATLAB Profiler and generates a Profile report that is saved to the file profworker_<split_index>_<socket>_<worker pass>.mat.
spark.matlab.worker.numberOfKeys	10000	Number of unique keys that can be held in a containers.Map object while performing *ByKey operations before map data is spilled to a file.
spark.matlab.executor.timeout	600000	Spark executor timeout in milliseconds. Not applicable when deploying tall arrays.

## Monitoring and Logging

Property Name (Key)	Default (Value)	Description
<code>spark.history.fs.logDirectory</code>	<code>file:/tmp/spark-events</code>	Directory that contains application event logs to be loaded by the history server.
<code>spark.eventLog.dir</code>	<code>file:///tmp/spark-events</code>	Base directory in which Spark events are logged, if <code>spark.eventLog.enabled</code> is <code>true</code> . Within this base directory, Spark creates a sub directory for each application, and logs the events specific to the application in this directory. You can set this to a unified location like an HDFS directory so history files can be read by the history server.
<code>spark.eventLog.enabled</code>	<code>false</code>	Whether to log Spark events. This is useful for reconstructing the web UI after the application has finished.

### Name-Value Pair Arguments

Specify optional comma-separated pairs of `Name`, `Value` arguments. `Name` is the argument name and `Value` is the corresponding value. `Name` must appear inside quotes. You can specify several name and value pair arguments in any order as `Name1, Value1, . . . , NameN, ValueN`.

### MCRRoot — Path to MATLAB Runtime that is used to execute driver application

character vector | string

A character vector specifying the path to MATLAB Runtime within single quotes `' '`.

Example: `'MCRRoot', '/share/MATLAB/MATLAB_Runtime/v91'`

Data Types: `char` | `string`

### SparkLogLevel — Set the Spark log level

`'ALL'` | `'DEBUG'` | `'ERROR'` | `'FATAL'` | `'INFO'` | `'OFF'` | `'TRACE'` | `'WARN'`

Specify the log level to set as a character vector with log level enclosed in `' '`.

Data Types: `char` | `string`

## Properties

The properties of this class are hidden.

## Methods

There are no user executable methods for this class.



## Examples

### Create DeploySparkMapReducer Object

Define Spark properties and create a DeploySparkMapReducer object.

```
sparkProperties = containers.Map( ...
    {'spark.executor.cores', ...
     'spark.executor.memory', ...
     'spark.yarn.executor.memoryOverhead', ...
     'spark.dynamicAllocation.enabled', ...
     'spark.shuffle.service.enabled', ...
     'spark.eventLog.enabled', ...
     'spark.eventLog.dir'}, ...
    {'1', ...
     '2g', ...
     '1024', ...
     'true', ...
     'true', ...
     'true', ...
     'hdfs://hadoopfs:54310/user/<username>/sparkdeploy'});

conf = matlab.mapreduce.DeploySparkMapReducer( ...
    'AppName','myTallApp', ...
    'Master','yarn-client', ...
    'SparkProperties',sparkProperties);

mapreducer(conf);
```

## See Also

### Topics

“Example on Deploying Tall Arrays to a Spark Enabled Hadoop Cluster” on page 6-2  
“Apache Spark Basics” on page 4-2

### Introduced in R2016b



## Methods — RDD

---

## aggregate

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Aggregate the elements of each partition and subsequently the results for all partitions into a single value

### Syntax

```
result = aggregate(obj, zeroValue, seqOp, combOp)
```

### Description

`result = aggregate(obj, zeroValue, seqOp, combOp)` aggregates the elements into a single value using given combine functions specified by `seqOp` and `combOp`, and a neutral “zero value” specified by `zeroValue`.

### Input Arguments

#### **obj — Input RDD**

RDD object

An input RDD, specified as a RDD object.

#### **zeroValue — Neutral “zero value”**

cell array of numbers

A neutral “zero value”, specified as a cell array of numbers.

Data Types: `cell`

#### **seqOp — Function to aggregate the values of each key**

function handle

A function to aggregate the values of each key, specified as a function handle.

Data Types: `function_handle`

#### **combOp — Function to aggregate results of seqOp**

function handle

A function to aggregate results of `seqOp`, specified as a function handle.

Data Types: `function_handle`

### Output Arguments

#### **result — RDD containing aggregated elements**

RDD object

An RDD containing aggregated elements, returned as an RDD object.

## Examples

### Aggregate Elements of an RDD into a Single Value

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% Aggregate
seqOp = @(x,y)({x{1} + y, x{2} + 1});
combOp = @(x,y)({x{1} + y{1}, x{2} + y{2}});
x = sc.parallelize({1, 2, 3, 4});
y = x.aggregate({0, 0}, seqOp, combOp) % {10,4}
```

### See Also

[aggregateByKey](#) | [parallelize](#)

**Introduced in R2016b**

## aggregateByKey

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Aggregate the values of each key, using given combine functions and a neutral “zero value”

### Syntax

```
result = aggregateByKey(obj, zeroValue, seqFunc, combFunc, numPartitions)
```

### Description

`result = aggregateByKey(obj, zeroValue, seqFunc, combFunc, numPartitions)` aggregates the values of each key, using given combine functions specified by `seqFunc` and `combFunc`, and a neutral “zero value” specified by `zeroValue`. The input argument `numPartitions` is optional.

### Input Arguments

#### **obj — Input RDD**

RDD object

An input RDD, specified as a RDD object.

#### **zeroValue — Neutral “zero value”**

cell array of numbers

A neutral “zero value”, specified as a cell array of numbers.

Data Types: `cell`

#### **seqFunc — Function to aggregate the values of each key**

function handle

Function that aggregates the values of each key, specified as a function handle.

Data Types: `function_handle`

#### **combFunc — Function to aggregate results of seqFunc**

function handle

Function to aggregate results of `seqFunc`, specified as a function handle.

Data Types: `function_handle`

#### **numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value. This argument is optional.

Data Types: `double`

## Output Arguments

**result** — RDD containing elements aggregated by key

RDD object

An RDD containing elements aggregated by key, returned as a RDD object.

## Examples

### Aggregate the Values of Each Key

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% aggregateByKey
x = sc.parallelize({'a', 'b', 'c', 'd'},4);
y = x.map(@(x)({x,1}));
z = y.aggregateByKey(10,@(x,y)(x+y),@(x,y)(x+y));
viewRes = z.collect() % { {'d',11},{ 'a',11},{ 'b',11},{ 'c',11}}
```

### See Also

[aggregate](#) | [collect](#) | [combineByKey](#) | [foldByKey](#) | [groupByKey](#) | [map](#) | [parallelize](#) | [reduceByKey](#) | [sortByKey](#) | [subtractByKey](#)

**Introduced in R2016b**

## cache

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Store an RDD in memory

### Syntax

```
cache(obj)
```

### Description

cache(obj) stores an RDD object specified by obj in the memory of the executors across a cluster.

### Input Arguments

**obj** — RDD to be cached in memory

RDD object

An RDD to be cached in memory, specified as an RDD object.

### Examples

#### Cache an RDD in Memory

Store an RDD in the memory of the executors across the cluster.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% cache
myFile = sc.textFile('airlinesmall.csv');
myFile.cache();
```

### See Also

textFile

**Introduced in R2016b**



# cartesian

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Create an RDD that is the Cartesian product of two RDDs

## Syntax

```
result = cartesian(obj1,obj2)
```

## Description

`result = cartesian(obj1,obj2)` creates a new RDD that is the Cartesian product between two RDDs, `obj1` and `obj2`.

## Input Arguments

### **obj1 – First input RDD**

RDD object

The first input RDD, specified as a RDD object.

### **obj2 – Second input RDD**

RDD object

The second input RDD, specified as a RDD object.

## Output Arguments

### **result – RDD representing the Cartesian product of two RDDs**

RDD object

An RDD representing the Cartesian product of two RDDs, returned as a RDD object.

## Examples

### **Compute Cartesian Product Between Two RDDs**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% cartesian
x = sc.parallelize({'A', 2, 'C'});
```

```
y = sc.parallelize({'D', 1});  
out = x.cartesian(y).collect(); % {'A','D'},{'A',1},{2,'D'},{2,1},{'C','D'},{'C',1}}
```

### See Also

[collect](#) | [parallelize](#)

**Introduced in R2016b**

# checkpoint

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Mark an RDD for checkpointing

## Syntax

```
checkpoint(obj)
```

## Description

checkpoint(obj) marks an RDD for checkpointing.

## Input Arguments

### obj — Input RDD

RDD object

An input RDD that is to be marked for checkpointing, specified as an RDD object.

## Examples

### Mark an RDD for Checkpointing

Use the checkpoint method to save an RDD to a file inside the checkpoint directory.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% checkpoint
sc.setCheckpointDir('myDir')
myFile = sc.parallelize({1,2,3});
mapRDD = myFile.map(@(x){x,1});
mapRDD.checkpoint();
```

## See Also

getCheckpointFile | map | parallelize | setCheckpointDir

**Introduced in R2016b**

## coalesce

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Reduce the number of partitions in an RDD

### Syntax

```
result = coalesce(obj,numPartitions,doShuffle)
```

### Description

`result = coalesce(obj,numPartitions,doShuffle)` reduces the number of partitions in an RDD to a number specified by `numPartitions`.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

#### **numPartitions** — Number of partitions to create

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: `double`

#### **doShuffle** — Perform shuffle

`false` (default) | `true`|`false`

Specify whether shuffle must be performed or not. By default `doShuffle` is set to `false`.

Data Types: `logical`

### Output Arguments

#### **result** — RDD with reduced number of partitions

RDD object

An RDD with reduced number of partitions, returned as a RDD object.

### Examples

#### Reduce the Number of Partitions in an RDD

```
% Connect to Spark  
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...
    'Master', 'local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% coalesce
inputRDD = sc.parallelize({'A', 'B', 'C', 'A', 'B'}, 2);
redRDD= inputRDD.map(@(x){x,1}).reduceByKey(@(x,y)(x+y),3);
coaRDD = redRDD.checkpoint(2);
viewRes = coaRDD.glom.collect() % {{{'B',2}},{{'C',1}},{'A',2}}}
```

## See Also

[collect](#) | [glom](#) | [map](#) | [parallelize](#) | [reduceByKey](#)

**Introduced in R2016b**

## cogroup

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Group data from RDDs sharing the same key

### Syntax

```
result = cogroup(obj1,obj2,numPartitions)
```

### Description

`result = cogroup(obj1,obj2,numPartitions)` groups the data from `obj1` and `obj2` that share the same key.

### Input Arguments

#### **obj1 — First input RDD**

RDD object

The first input RDD, specified as a RDD object.

#### **obj2 — Second input RDD**

RDD object

The second input RDD, specified as a RDD object.

#### **numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: `double`

### Output Arguments

#### **result — RDD containing grouped data**

RDD object

An RDD containing grouped data, returned as a RDD object.

### Examples

#### **Group Data from Key-Value Pair RDDs Sharing the Same Key**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
```

```
sc = matlab.compiler.mlspark.SparkContext(conf);  
  
%% cogroup  
x = sc.parallelize({ {'a'}, 1}, {'b'}, 4 } );  
y = sc.parallelize({ {'a'}, 2 } );  
z=x.cogroup(y);
```

## See Also

coalesce | parallelize

**Introduced in R2016b**

## collect

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return a MATLAB cell array that contains all of the elements in an RDD

### Syntax

```
result = collect(obj)
```

### Description

`result = collect(obj)` returns a MATLAB cell array that contains all of the elements in `obj`.

### Input Arguments

**obj** — Input RDD

RDD object

An input RDD, specified as an RDD object.

### Output Arguments

**result** — Elements of an RDD

cell array

Elements of an RDD, returned as a cell array.

### Examples

#### Return Contents of an RDD

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]','SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% Collect
x = sc.parallelize({'A', 2, 'C'});
y = sc.parallelize({'D', 1});
z = x.cartesian(y);
out = z.collect()
```

### See Also

`cartesian` | `collectAsMap` | `glom` | `parallelize`



**Introduced in R2016b**

## collectAsMap

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return the key-value pairs in an RDD as a MATLAB `containers.Map` object

### Syntax

```
result = collectAsMap(obj)
```

### Description

`result = collectAsMap(obj)` returns the key-value pairs in `obj` as a MATLAB `containers.Map` object.

### Input Arguments

**obj** — RDD object

RDD object

An RDD object, specified as a RDD object.

### Output Arguments

**result** — Key-value pairs in an RDD

`containers.Map` object

The key-value pairs in an RDD, returned as a MATLAB `containers.Map` object.

### Examples

#### Return the Key-Value Pairs in an RDD

```
% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

% Collect as Map
x = sc.parallelize({ {'a',1} ,{'b',2} });
c = x.collectAsMap() % c is a MAP with 'a','b' as keys and 1,2 as values
```

### See Also

`cartesian` | `collect` | `glom` | `parallelize`

**Introduced in R2016b**

# combineByKey

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Combine the elements for each key using a custom set of aggregation functions

## Syntax

```
result = combineByKey(obj,createCombiner,mergeValue,mergeCombiners,  
numPartitions)
```

## Description

`result = combineByKey(obj,createCombiner,mergeValue,mergeCombiners, numPartitions)` combines the elements for each key using a custom set of aggregation functions: `createCombiner` and `mergeValue`. The input argument `numPartitions` specifies the number of partitions to create in the resulting RDD.

## Input Arguments

### **obj — Input RDD to combine**

RDD object

An input RDD to combine, specified as a RDD object.

### **createCombiner — Combiner function (C), given a value (V)**

function handle

Combiner function (C), given a value (V), specified as a function handle.

Data Types: `function_handle`

### **mergeValue — Function representing a merging of the given value (V) with an existing combiner (C)**

function handle

Function representing a merging of the given value (V) with an existing combiner (C), specified as a function handle.

Data Types: `function_handle`

### **mergeCombiners — Function representing the merging of two combiners to return a new combiner**

function handle

Function representing the merging of two combiners to return a new combiner, specified as a function handle.

Example:

Data Types: `function_handle`

**numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Example:

Data Types: double

**Output Arguments****result — Cell array containing the elements of an RDD**

cell array

A MATLAB cell array containing the elements of an RDD.

**Examples****Combine the Elements for Each Key**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% combineByKey
inputRdd = sc.parallelize({'a',1}, {'b',1}, {'a',1});
resRdd = inputRdd.combineByKey(@(value) num2str(value), ...
    @(acc,value) strcat(acc, value), ...
    @(rdd1value, rdd2Value) strcat(rdd1Value, rdd2Value));
viewRes = resRdd.collect()
```

**See Also**

aggregateByKey | collect | foldByKey | glom | groupByKey | parallelize | reduceByKey | sortByKey | subtractByKey

**Introduced in R2016b**

# count

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Count number of elements in an RDD

## Syntax

```
result = count(obj)
```

## Description

`result = count(obj)` counts the number of elements in `obj`.

## Input Arguments

**obj** — Input RDD to count

RDD object

An input RDD to count, specified as a RDD object.

## Output Arguments

**result** — Number of elements in the input RDD

scalar

The number of elements in an input RDD, returned as a scalar.

## Examples

### Count the Number of Elements in an RDD

```
% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

% count
countVal = sc.parallelize({1, 2, 3, 4, 5}).count();
disp(countVal);
```

## See Also

parallelize

**Introduced in R2016b**

## distinct

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return a new RDD containing the distinct elements of an existing RDD

### Syntax

```
result = distinct(obj,numPartitions)
```

### Description

`result = distinct(obj,numPartitions)` returns a new RDD `result` containing the distinct elements of `obj` by eliminating duplicate values.

### Input Arguments

**obj — Input RDD to remove duplicates from**  
RDD object

An input RDD, specified as a RDD object.

**numPartitions — Number of partitions to create**  
scalar value

Number of partitions to create, specified as a scalar value.

Data Types: `double`

### Output Arguments

**result — RDD containing distinct elements**  
RDD object

A pipelined RDD containing distinct elements of the input RDD, returned as a RDD object.

## Examples

### Get the Distinct Elements of an RDD

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% distinct
inputRDD = sc.parallelize({1,2,1,2});
```

```
dRDD = inputRDD.distinct();  
viewRes = dRDD.glom().collect()  %{1,2}
```

**See Also**

[collect](#) | [glom](#) | [parallelize](#)

**Introduced in R2016b**

## filter

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return a new RDD containing only the elements that satisfy a predicate function

### Syntax

```
result = filter(obj, func, varargin)
```

### Description

`result = filter(obj, func, varargin)` applies a predicate function `func` to every element in `obj`. Elements that satisfy the criteria set by the predicate function are retained, others are discarded. A predicate function is one that returns `true` in a given logical function.

### Input Arguments

**obj — Input RDD to be filtered**

RDD object

An input RDD to be filtered, specified as a RDD object.

**func — Predicate function**

function handle

Predicate function, specified as a function handle. A predicate function returns `true` in a given logical function.

Data Types: `function_handle`

**varargin — Variable-length input argument list**

valid inputs to the predicate function

A variable-length input argument list representing inputs to the predicate function.

### Output Arguments

**result — Output RDD**

RDD object

An output RDD, returned as a RDD object.

### Examples

**Apply a Predicate Function to an RDD**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```



```
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...
    'Master', 'local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% filter
rdd = sc.parallelize({1, 2, 3, 4, 5});
res = rdd.filter(@(x) mod(x,2) == 0).collect();
```

## See Also

[collect](#) | [flatMap](#) | [map](#) | [parallelize](#)

**Introduced in R2016b**

## flatMap

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return a new RDD by first applying a function to all elements of an existing RDD, and then flattening the results

### Syntax

```
result = flatMap(obj, func, varargin)
```

### Description

`result = flatMap(obj, func, varargin)` returns a new RDD `result` by first applying a function `func` to all elements of `obj`, and then flattening the results.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD on which a function is applied, specified as a RDD object.

#### **func** — Function to apply to each element

function handle

Function to be applied to each element in the input RDD, specified as a function handle.

Example:

Data Types: `function_handle`

#### **varargin** — Variable-length input argument list

valid inputs to the transformation function

A variable-length input argument list representing inputs to the function that is being applied.

### Output Arguments

#### **result** — Output RDD

RDD object

An output RDD, returned as a RDD object.

### Examples

#### **Apply a Function to an RDD and Flatten the Results**

```
% Connect to Spark  
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...  
    'Master', 'local[1]', 'SparkProperties', sparkProp);  
sc = matlab.compiler.mlspark.SparkContext(conf);  
  
%% flatMap  
inRDD = sc.parallelize({'A', 'B'});  
flatRDD = inRDD.flatMap(@(x)({x,1}));  
viewRes = flatRDD.collect();
```

## See Also

[collect](#) | [filter](#) | [flatMapValues](#) | [map](#) | [parallelize](#)

**Introduced in R2016b**

## flatMapValues

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Pass each value in the key-value pair RDD through a `flatMap` method without changing the keys

### Syntax

```
result = flatMapValues(obj, func)
```

### Description

`result = flatMapValues(obj, func)` passes each value in a key-value pair RDD `obj` through the `flatMap` method without changing the keys. `func` represents the function to be applied by the `flatMap` method.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD on which a transformation function is applied, specified as a RDD object.

#### **func** — Function to apply to each element

function handle

Function to be applied to each element in the input RDD, specified as a function handle.

Example:

Data Types: `function_handle`

### Output Arguments

#### **result** — Output RDD

RDD object

An output pipelined RDD, returned as a RDD object.

### Examples

#### **Tokenize an RDD**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...
    'Master', 'local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);
```

```
%% flatMapValues
inRDD = sc.parallelize({ {'AA', {1,2,3}}, {'BB',3}, {'CC', {'cc',4}} });
f = @(x)(x);
out = inRDD.flatMapValues(f).collect();
% out : {'AA',1},{'AA',2},{'AA',3},{'BB',3},{'CC','cc'},{'CC',4}
```

## See Also

collect | flatMap | map | parallelize

**Introduced in R2016b**

## fold

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Aggregate elements of each partition and the subsequent results for all partitions

### Syntax

```
result = fold(obj, zeroValue, func)
```

### Description

`result = fold(obj, zeroValue, func)` aggregates the elements of each partition in `obj` and the subsequent results for all the partitions, using an associative function `func` and a neutral “zero value” represented by `zeroValue`.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as a RDD object.

**zeroValue — Neutral “zero value”**

cell array of numbers

A neutral “zero value”, specified as a cell array of numbers.

Data Types: `cell`

**func — Function for the folding action**

function handle

A function for the folding action, specified as a function handle.

Data Types: `function_handle`

### Output Arguments

**result — Result of the folding action**

scalar

Result of the folding action, returned as a scalar.

### Examples

**Accumulate the Elements of Each Partition and Subsequent Partitions**

```
% Connect to Spark  
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...  
    'Master', 'local[1]', 'SparkProperties', sparkProp);  
sc = matlab.compiler.mlspark.SparkContext(conf);  
  
%% fold  
foldVal = sc.parallelize({1, 2, 3, 4, 5}, 1).fold(10, @(x,y)(x+y));  
disp(foldVal);
```

## See Also

[collect](#) | [foldByKey](#) | [groupBy](#) | [parallelize](#)

**Introduced in R2016b**

## foldByKey

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Merge the values for each key using an associative function and a neutral “zero value”

### Syntax

```
result = foldByKey(obj, zeroValue, func, numPartitions)
```

### Description

`result = foldByKey(obj, zeroValue, func, numPartitions)` merges the values for each key in `obj` using an associative function `func` and a neutral “zero value” represented by `zeroValue`. The input argument `numPartitions` specifies the number of partitions to create in the resulting RDD.

### Input Arguments

#### **obj — Input RDD**

RDD object

An input RDD, specified as a RDD object.

#### **zeroValue — Neutral “zero value”**

cell array of numbers

A neutral “zero value”, specified as a cell array of numbers.

Data Types: `cell`

#### **func — Function for folding the values of each key**

function handle

Function for folding the values of each key, specified as a function handle.

Data Types: `function_handle`

#### **numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: `double`

### Output Arguments

#### **result — RDD containing the aggregation result**

RDD object

A pipelined RDD containing the aggregation result, returned as a RDD object.



## Examples

### Merge Values for Each Key

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% foldByKey
x = sc.parallelize({ {'a',1}, {'b',1}, {'a',1} });
y = x.foldByKey(5, @(x,y)(x+y));
viewRes = y.collect() % {'a',7},{'b',6}}
```

### See Also

aggregateByKey | collect | combineByKey | fold | groupByKey | map | parallelize |  
reduceByKey | sortByKey | subtractByKey

### Introduced in R2016b

## fullOuterJoin

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Perform a full outer join between two key-value pair RDDs

### Syntax

```
result = fullOuterJoin(obj1,obj2,numPartitions)
```

### Description

`result = fullOuterJoin(obj1,obj2,numPartitions)` performs a full outer join between two key-value pair RDDs, `obj1` and `obj2`. `numPartitions` specifies the number of partitions to create in the resulting RDD.

### Input Arguments

**obj1 — First input RDD to be joined**

RDD object

Input RDD to be joined, specified as a RDD object. RDD must be a key-value pair RDD.

**obj2 — Second input RDD to be joined**

RDD object

Input RDD to be joined, , specified as a RDD object. RDD must be a key-value pair RDD.

**numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: `double`

### Output Arguments

**result — RDD containing all pairs of elements with matching keys in the two input RDDs**

RDD object

A pipelined RDD containing all pairs of elements with matching keys in the two input RDDs, returned as a RDD object.

### Examples

**Perform a Full Outer Join**

```
% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]','SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% fullOuterJoin
x = sc.parallelize({ {'a',1}, {'b', 4} });
y = sc.parallelize({ {'a',2}, {'c', 8} });
z = x.fullOuterJoin(y);
viewRes = z.collect() % {'a',{1,2}},{'b',{4,[1]}},{'c',{[1],8}}
```

## See Also

[collect](#) | [join](#) | [leftOuterJoin](#) | [parallelize](#) | [rightOuterJoin](#)

**Introduced in R2016b**

## getCheckpointFile

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Get the name of the file to which an RDD is checkpointed

### Syntax

```
file = getCheckpointFile(obj)
```

### Description

`file = getCheckpointFile(obj)` gets the name of the file to which the RDD `obj` is checkpointed.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as an RDD object.

### Output Arguments

**file — File name to which an RDD was checkpointed**

character vector

The name of the file to which an RDD was checkpointed, returned as a character vector.

### Examples

#### Get Checkpointed File Name

Get the name of the file to which an RDD was checkpointed.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% getCheckpointFile
sc.setCheckpointDir('myDir') % set the checkpoint directory
x = sc.parallelize({1,2,3})
y = x.map(@(x)({x,1}));
y.checkpoint() % tell spark to checkpoint the RDD
y.collect()
% need to call collect, so that spark actually materializes the RDD
```

```
% and checkpoints to the myDir directory  
y.getCheckpointFile()
```

**See Also**

checkpoint | map | parallelize | setCheckpointDir

**Introduced in R2016b**

## getDefaultReducePartitions

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Get the number of default reduce partitions in an RDD

### Syntax

```
numPartitions = getDefaultReducePartitions(obj)
```

### Description

`numPartitions = getDefaultReducePartitions(obj)` gets the number of default reduce partitions in `obj`.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as an RDD object

### Output Arguments

**numPartitions — Number of default reduce partitions in the input RDD**

scalar value

The number of default reduce partitions in the input RDD, returned as a scalar value.

### Examples

#### Get Default Reduce Partitions

Get the number of default reduce partitions in an RDD.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% getDefaultReducePartitions
x = sc.parallelize({1,2,3});
y = x.map(@(x)({x,1}));
z1 = y.reduceByKey(@(a,b)(a+b));
z2 = y.reduceByKey(@(a,b)(a+b), 3);
```

```
z1.getDefaultReducePartitions() % ans is 1  
z2.getDefaultReducePartitions() % ans is 3, as the 2nd argument to reduceByKey is the number of
```

## See Also

`getNumPartitions` | `map` | `parallelize`

**Introduced in R2016b**

## getNumPartitions

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Return the number of partitions in an RDD

### Syntax

```
numPartitions = getNumPartitions(obj)
```

### Description

`numPartitions = getNumPartitions(obj)` returns the number of partitions in `obj`.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD, specified as an RDD object.

### Output Arguments

#### **numPartitions** — Number of partitions

scalar value

Number of partitions in the input RDD, returned as a scalar value.

### Examples

#### **Number of Partitions in an RDD**

Use the `getNumPartitions` method to return the number of partitions in an RDD.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% getNumPartitions
inputRDD = sc.parallelize({'A','B','C','A','B'},2);
redRDD= inputRDD.map(@(x){x,1}).reduceByKey(@(x,y)(x+y),3);
coaRDD = redRDD.coalesce(2); % {{{'B',2}},{{'C',1}},{'A',2}}}
disp(['Number of Partitions: ' num2str(coaRDD.getNumPartitions())]);
```

### See Also

`coalesce` | `getDefaultReducePartitions` | `map` | `parallelize`



**Introduced in R2016b**

## glom

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Coalesce all elements within each partition of an RDD

### Syntax

```
result = glom(obj)
```

### Description

`result = glom(obj)` returns an RDD `result` created by coalescing all elements within each partition of `obj`.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

### Output Arguments

#### **result** — RDD containing coalesced elements within each partition of the input RDD

RDD object

A pipelined RDD containing coalesced elements within each partition of the input RDD, returned as a RDD object.

### Examples

#### Coalesce All Elements Within Each Partition

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% glom
inputRDD = sc.parallelize({'A','B','C','D','E','C','B'});
mapRDD = inputRDD.map(@(x){x,1});
redRDD = mapRDD.reduceByKey(@(x,y)(x+y),3);
```

---

```
out = redRDD.glom().collect() % { {'C',2}}, {'A',1},{'D',1}}, {'B',2},{'E',1}} }  
% 3 cell arrays as 3 partitions were created by reduceByKey
```

**See Also**

[collect](#) | [collectAsMap](#) | [parallelize](#)

**Introduced in R2016b**

## groupBy

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return an RDD of grouped items

### Syntax

```
result = groupBy(obj,func,numPartitions)
```

### Description

`result = groupBy(obj,func,numPartitions)` groups the elements of `obj` according a user-specified criteria denoted by `func`. `numPartitions` specifies the number of partitions to create in the resulting RDD.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

#### **func** — Function to group by

function handle

Function performing grouping, specified as a function handle.

Data Types: `function_handle`

#### **numPartitions** — Number of partitions to create

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: `double`

### Output Arguments

#### **result** — RDD containing grouped elements

RDD object

A pipelined RDD containing grouped elements of the input RDD, returned as a RDD object.

### Examples

#### **RDD of Grouped Items**

Groups the elements of an RDD according a user-specified criteria.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% groupBy
inRDD = sc.parallelize({1,2,3,4,5});
outRDD = inRDD.groupBy(@(x)(mod(x,2))).collect(); % {{0},{2,4}},{1},{1,3,5}}
```

## See Also

groupByKey | parallelize

**Introduced in R2016b**

## groupByKey

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Group the values for each key in the RDD into a single sequence

### Syntax

```
result = groupByKey(obj,numPartitions)
```

### Description

`result = groupByKey(obj,numPartitions)` groups the values for each key in `obj` into a single sequence. `numPartitions` specifies the number of partitions to create in the resulting RDD.

### Input Arguments

#### **obj** — Input RDD to group

RDD object

An input RDD to group, specified as a RDD object.

#### **numPartitions** — Number of partitions to create

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: `double`

### Output Arguments

#### **result** — RDD containing elements grouped by key

RDD object

A pipelined RDD containing elements of the input RDD grouped by key, returned as a RDD object.

## Examples

### Group into a Single Sequence RDD Using a Key

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% groupByKey
```

```
inRDD = sc.parallelize({{'a',1},{'b',2}, {'a','3'}, {'b',4}});  
outRDD = inRDD.groupByKey().collect(); % {'a',{1,'3'}},{'b',{2,4}}'
```

## See Also

groupBy | parallelize

**Introduced in R2016b**

## intersection

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return the set intersection of one RDD with another

### Syntax

```
result = intersection(obj1,obj2)
```

### Description

`result = intersection(obj1,obj2)` returns elements that are the set intersection of `obj1` and `obj2`.

### Input Arguments

#### **obj1 — First input RDD**

RDD object

An input RDD, , specified as a RDD object.

#### **obj2 — Second input RDD**

RDD object

An input RDD, specified as a RDD object.

### Output Arguments

#### **result — RDD containing the set intersection of the two input RDDs**

RDD object

A pipelined RDD containing the set intersection of the two input RDDs, returned as a RDD object.

### Examples

#### **Set Intersection of RDDs**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% intersection
x = sc.parallelize({'A','B','C'});
y = sc.parallelize({'B','D'});
```



```
z = sc.parallelize({'E', 'F'});  
out1 = x.intersection(y).collect(); % {'B'}
```

**See Also**

collect | parallelize | subtract | union

**Introduced in R2016b**

## isEmpty

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Determine if an RDD contains any elements

### Syntax

```
tf = isEmpty(obj)
```

### Description

`tf = isEmpty(obj)` returns a logical 1 (true) if the input RDD `obj` contains no elements, and a logical 0 (false) otherwise.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as an RDD object.

### Output Arguments

**tf — Input RDD is empty or not**

logical 1 (true)|0 (false)

Indicates whether the input RDD is empty or not, returned as a logical value.

### Examples

#### Check If an RDD Contains Elements

Check if an RDD contains any elements.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]','SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% isEmpty
x = sc.parallelize({});
tf = x.map(@(x){x,1}).isEmpty(); % tf=1
```

### See Also

`collect` | `parallelize`

**Introduced in R2016b**

## join

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return an RDD containing all pairs of elements with matching keys

### Syntax

```
result = join(obj1,obj2,numPartitions)
```

### Description

`result = join(obj1,obj2,numPartitions)` performs an inner join on `obj1` and `obj2` and returns an RDD `result` of key-value pairs containing all pairs of elements with matching keys in the input RDDs. `obj1` and `obj2` must be key-value pair RDDs. `numPartitions` specifies the number of partitions to create in the resulting RDD.

### Input Arguments

#### **obj1 — First input RDD to be joined**

RDD object

The first input RDD to be joined, specified as a RDD object. RDD must be a key-value pair RDD.

#### **obj2 — Second input RDD to be joined**

RDD object

The second input RDD to be joined, specified as a RDD object. RDD must be a key-value pair RDD.

#### **numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: `double`

### Output Arguments

#### **result — RDD containing all pairs of elements with matching keys in the two input RDDs**

RDD object

A pipelined RDD containing all pairs of elements with matching keys in the two input RDDs, returned as a RDD object.

### Examples

#### **Join Two RDDs**

```
% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...
    'Master', 'local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% join
x = sc.parallelize({ {'a',11}, {'b', {1,2,3}}, {'b', '22'} ,{'c', 'dd'} });
y = sc.parallelize({ {'a',33}, {'b',44} , {'a',55}, {'d', 5 }});
z = x.join(y, 2);
viewRes = z.collect() % {'b',{1,2,3},44},{b','22',44},{a',{11,33}},{'a',{11,55}}}
```

## See Also

[collect](#) | [fullOuterJoin](#) | [leftOuterJoin](#) | [parallelize](#) | [rightOuterJoin](#)

**Introduced in R2016b**

## keyBy

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Create tuples of the elements in an RDD by applying a function

### Syntax

```
result = keyBy(obj, func)
```

### Description

`result = keyBy(obj, func)` takes a function `func` that returns a key for any given element in `obj`. The `keyBy` method applies this function to all the elements in `obj` and returns an output RDD `result` of key-value pairs.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

#### **func** — Function to be applied

function handle

Function to be applied, specified as a function handle.

Data Types: `function_handle`

### Output Arguments

#### **result** — RDD containing tuples of the elements in the input RDD

RDD object

A pipelined RDD containing tuples of the elements in the input RDD, returned as a RDD object.

### Examples

#### **Create a Tuple of Keys and Values**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% keyBy
```

```
x = sc.parallelize({1,2,3});  
c = x.keyBy(@(x)(x*x)).collect(); % {{1,1},{4,2},{9,3}}
```

## See Also

keys | parallelize

**Introduced in R2016b**

## keys

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return an RDD with the keys of each tuple

### Syntax

```
result = keys(obj)
```

### Description

`result = keys(obj)` returns an RDD `result` with the keys of each tuple in `obj`.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as a RDD object.

### Output Arguments

**result — RDD containing the keys of each tuple in the input RDD**

RDD object

A pipelined RDD containing the keys of each tuple in the input RDD, returned as a RDD object.

## Examples

### Get Keys From a Key-Value RDD

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% keys
m = sc.parallelize({'AA', {5,15} }, {'BB', 200});
out = m.keys().collect();
```

### See Also

`keyBy` | `parallelize`

**Introduced in R2016b**



# leftOuterJoin

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Perform a left outer join

## Syntax

```
result = leftOuterJoin(obj1,obj2,numPartitions)
```

## Description

`result = leftOuterJoin(obj1,obj2,numPartitions)` performs a left outer join on `obj1` and `obj2`. `numPartitions` specifies the number of partitions to create in the resulting RDD.

## Input Arguments

**obj1 — First input RDD to be joined**

RDD object

Input RDD to be joined, specified as a RDD object. RDD must be a key-value pair RDD.

**obj2 — Second input RDD to be joined**

RDD object

Input RDD to be joined, specified as a RDD object. RDD must be a key-value pair RDD.

**numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: double

## Output Arguments

**result — RDD containing all pairs of elements with matching keys in the two input RDDs**

RDD object

A pipelined RDD containing all pairs of elements with matching keys in the two input RDDs, returned as a RDD object.

## Examples

### Perform a Left Outer Join

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
```

```
sc = matlab.compiler.mlspark.SparkContext(conf, 'Master', 'local[1]', 'SparkProperties', sparkProp);  
  
%% leftOuterJoin  
x = sc.parallelize({ {'a', 1}, {'b', 4} });  
y = sc.parallelize({ {'a', 2} });  
z = x.leftOuterJoin(y);  
viewRes = z.collect()
```

**See Also**

[collect](#) | [fullOuterJoin](#) | [join](#) | [parallelize](#) | [rightOuterJoin](#)

**Introduced in R2016b**

# map

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Return a new RDD by applying a function to each element of an input RDD

## Syntax

```
result = map(obj, func, varargin)
```

## Description

`result = map(obj, func, varargin)` returns a new RDD `result` by first applying a function `func` to all elements of `obj`. `varargin` represents a variable-length input argument list for the inputs to the function being applied.

## Input Arguments

### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

### **func** — Function to be applied to each element

function handle

Function to be applied to each element in the input RDD, specified as a function handle.

Data Types: `function_handle`

### **varargin** — Variable-length input argument list

valid inputs to the function that is being applied

A variable-length input argument list, specifying inputs to the function that is being applied.

## Output Arguments

### **result** — RDD containing mapped elements

RDD object

A pipelined RDD containing mapped elements of the input RDD, returned as a RDD object.

## Examples

### Apply a Function to Each Element of an RDD

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...
```

```
sc = matlab.compiler.mlspark.SparkContext(conf, 'Master', 'local[1]', 'SparkProperties', sparkProp);  
sc = matlab.compiler.mlspark.SparkContext(conf);  
  
%% map  
inputRDD = sc.parallelize({'A', 'B', 'C', 'A', 'B'}, 2);  
redRDD = inputRDD.map(@(x)({x, 1})).collect();
```

### See Also

[collect](#) | [flatMap](#) | [parallelize](#) | [reduce](#)

**Introduced in R2016b**

# mapValues

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Pass each value in a key-value pair RDD through a map function without modifying the keys

## Syntax

```
result = mapValues(obj, func)
```

## Description

`result = mapValues(obj, func)` passes each value in a key-value pair RDD `obj` through a map function `func` without modifying the keys.

## Input Arguments

### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

### **func** — Function to be applied to each element

function handle

Function to be applied to each element, specified as a function handle.

Data Types: `function_handle`

## Output Arguments

### **result** — RDD containing mapped elements

RDD object

A pipelined RDD containing mapped elements of the input RDD, returned as a RDD object.

## Examples

### Apply Function to Each Value in an RDD

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...
    'Master', 'local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% mapValues
x = sc.parallelize({'A', 'B', 'A'});
```

```
% reduce by key the above keys and square each of the values,  
% so {'A',1} -> {'A',1*1}, {'B', 2} -> {'B', 2*2}  
  
y = x.map(@(x)({x,1})).reduceByKey(@(x,y)(x+y)).mapValues(@(x)(x*x))  
z = y.collect() % {'A',4},{'B',1}}
```

### See Also

[collect](#) | [flatMap](#) | [map](#) | [parallelize](#)

**Introduced in R2016b**

# keyLimit

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Return threshold of unique keys that can be stored before spilling to disk

## Syntax

```
result = keyLimit(obj)
```

## Description

`result = keyLimit(obj)` returns the threshold of unique keys in `obj` that can be stored in memory before spilling to disk.

## Input Arguments

### **obj** — Input RDD

RDD object

An input RDD, specified as an RDD object.

## Output Arguments

### **result** — Threshold of unique keys

scalar value

Threshold of unique keys that can be stored before spilling to disk, returned as a scalar value.

## Examples

### Get Threshold of Unique Keys

Use the `keyLimit` method to return the threshold of unique keys that can be stored in a `containers.Map` object that specifies Spark properties. Keys that breach the threshold are spilled to disk.

```
%% Connect to Spark
% Change number of keys from a default threshold of 10,000 to 500
sparkProp = containers.Map( ...
    {'spark.executor.cores', ...
     'spark.executor.memory', ...
     'spark.executor.instances', ...
     'spark.matlab.worker.numOfKeys', ...
    }, ...
    {'1', ...
     '2g', ...
     '1', ...
    }
```

```
'500'});  
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...  
    'Master','local[1]','SparkProperties',sparkProp);  
sc = matlab.compiler.mlspark.SparkContext(conf);  
  
%% keyLimit  
x = sc.parallelize({1,2,3});  
x.keyLimit % ans: 500
```

## See Also

parallelize

**Introduced in R2016b**



## persist

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Set the value of an RDD's storage level to persist across operations after it is computed

### Syntax

```
persist(obj, storageLevel)
```

### Description

`persist(obj, storageLevel)` sets a persistent storage level specified by `storageLevel` in RDD object `obj`. The default storage level is `MEMORY_ONLY`. Use the `persist` method to assign a new storage level if `obj` does not have a storage level set. You can also use it to set a persistent storage level in memory across operations.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD, specified as an RDD object.

#### **storageLevel** — New storage level to be assigned

`MEMORY_ONLY` (default) | `DISK_ONLY` | `MEMORY_AND_DISK` | `MEMORY_ONLY_2` | `DISK_ONLY_2` | `MEMORY_AND_DISK_2` | `OFF_HEAP`

New storage level to be assigned, specified as a character vector enclosed in `' '`. Use `storageLevel` to assign a new storage level if the RDD does not have a storage level set. The default storage level is `MEMORY_ONLY`.

Storage Level	Description
<code>MEMORY_ONLY</code>	Store the RDD in memory. If the RDD does not fit in memory, some partitions are not cached, and are recomputed each time they are needed.
<code>DISK_ONLY</code>	Store the RDD partitions on disk.
<code>MEMORY_AND_DISK</code>	Store the RDD in memory. If it does not fit in memory, then spill to disk.
<code>MEMORY_ONLY_2</code>	Store the RDD in memory, but replicate each partition in two cluster nodes.
<code>DISK_ONLY_2</code>	Store the RDD partitions on disk, but replicate each partition in two cluster nodes.
<code>MEMORY_AND_DISK_2</code>	Store the RDD in memory. If it does not fit in memory, then spill to disk. Replicate each partition in two cluster nodes.

Storage Level	Description
OFF_HEAP	Store RDD in serialized format. For more information, see the programming guide at <a href="https://spark.apache.org/">https://spark.apache.org/</a>

Data Types: char

## Examples

### Persist an RDD

Use the `persist` method without any parameter to store an RDD in the memory of the executors across a cluster.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% persist
myFile = sc.textFile('airlinesmall.csv');
myFile.persist();
myFile.unpersist();
```

### See Also

`cache` | `parallelize` | `textFile` | `unpersist`

**Introduced in R2016b**

## reduce

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Reduce elements of an RDD using the specified commutative and associative function

### Syntax

```
result = reduce(obj,func)
```

### Description

`result = reduce(obj,func)` reduces the elements of `obj` using the specified commutative and associative function `func`.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as a RDD object.

**func — Commutative and associative function to apply on elements of the RDD, specified as a function handle**

function handle

Commutative and associative function that will be applied to elements of the RDD.

Data Types: `function_handle`

### Output Arguments

**result — Reduced elements of the input RDD**

scalar value

Reduced elements of the input RDD, returned as a scalar.

### Examples

#### Aggregate the Elements of an RDD Using a Binary Operator

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% Reduce
```

```
reduceVal = sc.parallelize({1, 2, 3, 4, 5}).reduce(@(x,y)(x+y));  
disp(reduceVal);
```

### **See Also**

[collect](#) | [fold](#) | [map](#) | [parallelize](#) | [reduceByKey](#) | [reduceByKeyLocally](#)

**Introduced in R2016b**

## reduceByKey

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Merge the values for each key using an associative reduce function

### Syntax

```
result = reduceByKey(obj,func,numPartitions)
```

### Description

`result = reduceByKey(obj,func,numPartitions)` merges the values for each key in `obj` using an associative reduce function `func`. `numPartitions` specifies the number of partitions to create in the resulting RDD.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as a RDD object.

**func — Associative function to be applied**

function handle

Associative function to be applied to the elements of the input RDD, specified as a function handle.

Data Types: `function_handle`

**numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: `double`

### Output Arguments

**result — RDD containing values reduced by key**

RDD object

A pipelined RDD containing values reduced by key, returned as a RDD object.

### Examples

**Reduce Values By Key**

```
% Connect to Spark  
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...  
    'Master', 'local[1]', 'SparkProperties', sparkProp);  
sc = matlab.compiler.mlspark.SparkContext(conf);  
  
%% reduceByKey  
inputRDD = sc.parallelize({'A', 'B', 'C', 'A', 'B'}, 2);  
redRDD = inputRDD.map(@(x)({x, 1})).reduceByKey(@(x,y)(x+y), 3);
```

## See Also

[collect](#) | [map](#) | [parallelize](#) | [reduce](#) | [reduceByKeyLocally](#)

**Introduced in R2016b**

# reduceByKeyLocally

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Merge the values for each key using an associative reduce function, but return the results immediately to the driver

## Syntax

```
result = reduceByKeyLocally(obj,func)
```

## Description

`result = reduceByKeyLocally(obj,func)` merges the values for each key using an associative reduce function `func`, and returns the results immediately to the driver.

## Input Arguments

### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

### **func** — Associative function to be applied

function handle

Associative function to be applied to the elements of the input RDD, specified as a function handle.

Data Types: `function_handle`

## Output Arguments

### **result** — List of key-value pairs

`containers.Map` object

A list of key-value pairs, returned in a `containers.Map` object.

## Examples

### Reduce Values for Each Key Using an Associative Reduce Function

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% reduceByKeyLocally
```

```
x = sc.parallelize({ {'a',10}, {'b',20}, {'a',30}, {'b',30 } }, 2);  
m = x.reduceByKeyLocally(@(x,y)(x+y));
```

### **See Also**

[collect](#) | [map](#) | [parallelize](#) | [reduce](#) | [reduceByKey](#)

**Introduced in R2016b**



# repartition

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Return a new RDD that has exactly numPartitions partitions

## Syntax

```
result = repartition(obj,numPartitions)
```

## Description

result = repartition(obj,numPartitions) returns a new RDD result that has exactly numPartitions partitions.

## Input Arguments

### obj — Input RDD

RDD object

An input RDD, specified as a RDD object.

### numPartitions — Number of partitions to create

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: double

## Output Arguments

### result — Repartitioned RDD

RDD object

A repartitioned RDD with partitions specified by numPartitions, returned as a RDD object.

## Examples

### Repartition an RDD

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% repartition
inputRDD = sc.parallelize({1,2,3,2,1},4);
```

```
outRDD1 = inputRDD.repartition(1); % {1,2,3,2,1}
viewRes = outRDD1.collect()
```

### **See Also**

[collect](#) | [getDefaultReducePartitions](#) | [getNumPartitions](#) | [parallelize](#)

**Introduced in R2016b**

# rightOuterJoin

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Perform a right outer join

## Syntax

```
result = rightOuterJoin(obj1,obj2,numPartitions)
```

## Description

`result = rightOuterJoin(obj1,obj2,numPartitions)` performs a right outer join between two key-value pair RDDs, `obj1` and `obj2`. `numPartitions` specifies the number of partitions to create in the resulting RDD.

## Input Arguments

**obj1 — First input RDD to be joined**

RDD object

Input RDD to be joined, specified as a RDD object. RDD must be a key-value pair RDD.

**obj2 — Second input RDD to be joined**

RDD object

Input RDD to be joined, specified as a RDD object. RDD must be a key-value pair RDD.

**numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: double

## Output Arguments

**result — RDD containing all pairs of elements with matching keys in the two input RDDs**

RDD object

A pipelined RDD containing all pairs of elements with matching keys in the two input RDDs, returned as a RDD object.

## Examples

### Right Outer Join an RDD

```
% Connect to Spark  
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]','SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% rightOuterJoin
x = sc.parallelize({ {'a',1}, {'b', 4} });
y = sc.parallelize({ {'a',2} });
z = y.rightOuterJoin(x);
viewRes = z.collect() % {'a',{2,1}},{'b',{[],4}}}
```

## See Also

[collect](#) | [fullOuterJoin](#) | [join](#) | [leftOuterJoin](#) | [parallelize](#)

**Introduced in R2016b**

# saveAsTallDatastore

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Save RDD as a MATLAB tall array to a binary file that can be read back using the `datastore` function

## Syntax

```
saveAsMatlabBinaryFile(obj,path)
```

## Description

`saveAsMatlabBinaryFile(obj,path)` saves `obj` as a MATLAB tall array in a binary file that can be read back using the `datastore` function. `path` specifies the directory location in which to save the binary file.

## Input Arguments

### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

### **path** — Directory location

character vector

Directory location in which to save the binary file, specified as a character vector enclosed in `' '`.

Data Types: `char`

## Examples

### Save RDD as a Tall Array

Save an RDD as a MATLAB tall array to a binary file that can be read back using the `datastore` function.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% saveAsTallDatastore

% May require setting HADOOP_PREFIX or HADOOP_HOME environment variables to a
% valid Hadoop installation folder even if running locally.
% For example:
% setenv('HADOOP_PREFIX','/share/hadoop/hadoop-2.5.2')
```

```
inRDD = sc.parallelize({1,2,3,4,5});  
% Store RDD in a file as a tall array that can be read back into MATLAB using datastore  
inRDD.saveAsTallDatastore('myDir');  
ds = datastore(['myDir' '/part*'], 'Type', 'tall');  
ds.readall()
```

### See Also

[datastore](#) | [parallelize](#) | [saveAsKeyValueDatastore](#) | [saveAsTextFile](#)

**Introduced in R2016b**

# saveAsKeyValueDatastore

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Save key-value RDD as a binary file that can be read back using the `datastore` function

## Syntax

```
saveAsKeyValueDatastore(obj,path)
```

## Description

`saveAsKeyValueDatastore(obj,path)` saves a key-value input RDD `obj` as a binary file that can be read back using the `datastore` function.

## Input Arguments

### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

### **path** — Directory location

character vector

Directory location where binary file must be saved, specified as a character vector enclosed in `' '`.

Data Types: `char`

## Examples

### Save RDD as a Key-Value Binary File

Save a key-value RDD as a binary file that can be read back using the `datastore` function.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% saveAsKeyValueDatastore

% May require setting HADOOP_PREFIX or HADOOP_HOME environment variables to a
% valid Hadoop installation folder even if running locally.
% For example:
% setenv('HADOOP_PREFIX','/share/hadoop/hadoop-2.5.2')

inRDD = sc.parallelize({1,2,3,4,5});
redRDD= inputRDD.map(@(x)({x,1})).reduceByKey(@(x,y)(x+y))
```

```
% Store RDD in a key-value binary file that can be read back into MATLAB using datastore  
redRdd.saveAsKeyValueDatastore('myKVdir')
```

### **See Also**

[datastore](#) | [parallelize](#) | [saveAsTallDatastore](#) | [saveAsTextFile](#)

**Introduced in R2016b**



# saveAsTextFile

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Save RDD as a text file

## Syntax

```
saveAsTextFile(obj,path)
```

## Description

saveAsTextFile(obj,path) saves obj as a text file in a location specified by path.

## Input Arguments

### obj — Input RDD

RDD object

An input RDD, specified as a RDD object.

### path — Directory location

character vector

Directory location in which to save the text file, specified as a character vector enclosed in ' '.

Data Types: char

## Examples

### Save an RDD as a Text File

Save an RDD as a text file by converting each RDD element to its string representation and storing it as a line of text.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]','SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% saveAsTextFile

% May require setting HADOOP_PREFIX or HADOOP_HOME environment variables to a
% valid Hadoop installation folder even if running locally.
% For example:
% setenv('HADOOP_PREFIX','/mathworks/AH/hub/apps_PCT/LS_Hadoop_hadoop01glnxa64/hadoop-2.5.2')
```

```
inRDD = sc.parallelize({1,2,3,4,5});  
inRDD.saveAsTextFile('myFile.txt');
```

### **See Also**

[datastore](#) | [parallelize](#) | [saveAsKeyValueDatastore](#) | [saveAsTextFile](#)

**Introduced in R2016b**

## sortBy

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Sort an RDD by a given function

### Syntax

```
result = sortBy(obj,func,numPartitions)
```

### Description

`result = sortBy(obj,func,numPartitions)` sorts `obj` using a given `func`. `numPartitions` specifies the number of partitions to create in the resulting RDD.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD specified as a RDD object.

#### **func** — Function to compute the sort key for each element

function handle

Function that computes the sort key for each element in the input RDD, specified as a function handle.

Data Types: `function_handle`

#### **numPartitions** — Number of partitions to create

scalar value

A scalar value specifying the number of partitions to create, returned as a RDD object.

Data Types: `double`

### Output Arguments

#### **result** — Output RDD

RDD object

An output pipelined RDD.

### Examples

#### Sort an RDD

```
% Connect to Spark  
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]','SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% sortBy
x = sc.parallelize({ {'a', 50}, {'b', 20}, {'f', 40}, {'d', 30}, {'2',5} });
% sort by 2nd element in each key-value pair
z = x.sortBy(@(x)(x{2}));
viewRes = z.collect() % {'2',5},{'b', 20},{'d', 30},{'f', 40}, {'a', 50}}
```

## See Also

[parallelize](#) | [sortByKey](#)

**Introduced in R2016b**

## sortByKey

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Sort RDD consisting of key-value pairs by key

### Syntax

```
result = sortByKey(obj,numPartitions)
```

### Description

`result = sortByKey(obj,numPartitions)` sorts a key-value `obj` by key. `numPartitions` specifies the number of partitions to create in the resulting RDD.

### Input Arguments

#### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

#### **numPartitions** — Number of partitions to create

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: double

### Output Arguments

#### **result** — RDD containing elements sorted by key

RDD object

A pipelined RDD containing elements sorted by key, returned as a RDD object.

### Examples

#### Sort an RDD by Key

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% sortByKey
x = sc.parallelize({'a',1}, {'d',3}, {'f', 3} , {'b', 2}, {'e', 9}}, 3);
```

```
z = x.sortByKey();  
viewRes = z.collect() % {'a',1},{'b',2},{'d',3},{'e',9},{'f',3}}
```

### See Also

`parallelize` | `sortBy`

**Introduced in R2016b**

# subtract

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Return the values resulting from the set difference between two RDDs

## Syntax

```
result = subtract(obj1,obj2,numPartitions)
```

## Description

`result = subtract(obj1,obj2,numPartitions)` returns elements that are the set difference of `obj1` and `obj2`. `numPartitions` specifies the number of partitions to create in the resulting RDD.

## Input Arguments

### **obj1 — First input RDD**

RDD object

An input RDD, specified as a RDD object.

### **obj2 — Second input RDD**

RDD object

An input RDD, specified as a RDD object.

### **numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: double

## Output Arguments

### **result — RDD containing the set subtraction of the two input RDDs**

RDD object

A pipelined RDD containing the set subtraction of the two input RDDs, returned as a RDD object.

## Examples

### **Set Difference of Two RDDs**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
```

```
sc = matlab.compiler.mlspark.SparkContext(conf);  
  
%% subtract  
x = sc.parallelize({ 1,2,3 });  
y = sc.parallelize({ 2,4,5 });  
c = x.subtract(y,2).collect(); % {1,3}
```

### See Also

[intersection](#) | [parallelize](#) | [subtractByKey](#) | [union](#)

**Introduced in R2016b**



# subtractByKey

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Return key-value pairs resulting from the set difference of keys between two RDDs

## Syntax

```
result = subtractByKey(obj1,obj2,numPartitions)
```

## Description

`result = subtractByKey(obj1,obj2,numPartitions)` returns a key-value pair RDD `result` resulting from the set difference of keys between `obj1` and `obj2`. `numPartitions` specifies the number of partitions to create in the resulting RDD.

## Input Arguments

### **obj1 — First input RDD**

RDD object

An input RDD, specified as a RDD object.

### **obj2 — Second input RDD**

RDD object

An input RDD, specified as a RDD object.

### **numPartitions — Number of partitions to create**

scalar value

Number of partitions to create, specified as a scalar value.

Data Types: double

## Output Arguments

### **result — RDD containing the set difference of keys between two RDDs**

RDD object

A pipelined RDD containing the set difference of keys between two RDDs, returned as a RDD object.

## Examples

### **Set Difference of Two RDDs by Key**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
```

```
sc = matlab.compiler.mlspark.SparkContext(conf, 'Master', 'local[1]', 'SparkProperties', sparkProp);  
  
%% subtractByKey  
x = sc.parallelize({ {'a',1}, {'b',4}, {'b',5} , {'a',2} });  
y = sc.parallelize({ {'a',3}, {'c',4} });  
z = sc.parallelize({ {'a',2}, {'c',4} });  
a = x.subtractByKey(y).collect(); % {'b',4},{b',5}}
```

## See Also

[parallelize](#) | [subtract](#)

**Introduced in R2016b**

## toDebugString

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Obtain a description of an RDD and its recursive dependencies for debugging

### Syntax

```
str = toDebugString(obj)
```

### Description

`str = toDebugString(obj)` gets a description of input RDD and its recursive dependencies for debugging purposes.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as an RDD object.

### Output Arguments

**str — Character vector with a description of an RDD**

character vector

Description of an RDD and its recursive dependencies for debugging purposes, returned as a character vector.

Data Types: char

### Examples

#### Obtain Description of an RDD

Use the `toDebugString` method to get a description of an RDD and its recursive dependencies.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% toDebugString
myFile = sc.textFile('airlinesmall.csv');
```

```
myFile.persist();  
myFile.toDebugString()
```

**See Also**

parallelize | persist

**Introduced in R2016b**

## union

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Return the set union of one RDD with another

### Syntax

```
result = union(obj1,obj2)
```

### Description

`result = union(obj1,obj2)` returns elements that are the set union of `obj1` and `obj2`.

### Input Arguments

#### **obj1 — First input RDD**

RDD object

An input RDD, specified as a RDD object.

#### **obj2 — Second input RDD**

RDD object

An input RDD, specified as a RDD object.

### Output Arguments

#### **result — RDD containing the set union of the two input RDDs**

RDD object

A pipelined RDD containing the set union of the two input RDDs, returned as a RDD object.

### Examples

#### **Set Union of Two RDDs**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% union
inRDD = sc.parallelize({'A','B'});
```

```
newRDD = inRDD.union(inRDD); %{'A','B', 'A','B'}  
viewRes = newRDD.collect()
```

**See Also**

[intersection](#) | [parallelize](#) | [subtract](#) | [subtractByKey](#)

**Introduced in R2016b**

# unpersist

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Mark an RDD as nonpersistent, remove all blocks for it from memory and disk

## Syntax

```
unpersist(obj)
```

## Description

`unpersist(obj)` marks input RDD object as nonpersistent and removes all blocks for it from memory and disk.

## Input Arguments

### **obj** — Input RDD

RDD object

An input RDD, specified as an RDD object.

## Examples

### **Mark an RDD as Non-persistent**

Use the `unpersist` method to mark an RDD as nonpersistent.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% unpersist
myFile = sc.textFile('airlinesmall.csv');
myFile.persist();
myFile.unpersist();
```

### **See Also**

`parallelize` | `persist` | `textFile`

**Introduced in R2016b**

## values

**Class:** `matlab.compiler.mlspark.RDD`

**Package:** `matlab.compiler.mlspark`

Return an RDD with the values of each tuple

### Syntax

```
result = values(obj)
```

### Description

`result = values(obj)` returns an RDD `result` with the values of each tuple in `obj`.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as a RDD object.

### Output Arguments

**result — RDD containing values of each tuple**

RDD object

A pipelined RDD containing values of each tuple, returned as a RDD object.

### Examples

#### Return an RDD with the Values of Each Tuple

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% values
m = sc.parallelize({ {'AA', {5,15} }, {'BB', 200}});
out = m.values().collect(); % { {5,15}, 200 }
```

### See Also

`keys` | `parallelize`

**Introduced in R2016b**



# zip

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Zip one RDD with another

## Syntax

```
result = zip(obj1,obj2)
```

## Description

`result = zip(obj1,obj2)` returns a key-value pair RDD `result`, where the first element in the pair is from `obj1` and second element is from `obj2`. The output RDD `result` has the same number of elements as `obj1`. Both the `obj1` and the `obj2` must have the same length.

## Input Arguments

### **obj1 — First input RDD**

RDD object

An input RDD, specified as a RDD object.

### **obj2 — Second input RDD**

RDD object

An input RDD, specified as a RDD object.

## Output Arguments

### **result — Output RDD zipped from two input RDDs**

RDD object

An output RDD zipped from two input RDDs, returned as a RDD object.

## Examples

### **Zip One RDD With Another**

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]','SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% zip
x = sc.parallelize({'A', 'B', 3});
```

```
y = sc.parallelize({1, 2, 'C'});  
out = x.zip(y).collect(); % {{'A',1},{'B',2},{3,'C'}}
```

### See Also

[cartesian](#) | [parallelize](#) | [zipWithIndex](#) | [zipWithUniqueId](#)

**Introduced in R2016b**

# zipWithIndex

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Zip an RDD with its element indices

## Syntax

```
result = zipWithIndex(obj)
```

## Description

`result = zipWithIndex(obj)` zips an `obj` with its element indices.

## Input Arguments

### **obj** — Input RDD

RDD object

An input RDD, specified as a RDD object.

## Output Arguments

### **result** — Output RDD

RDD object

An output pipelined RDD, returned as a RDD object.

## Examples

### Zip an RDD With its Element Indices

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% zipWithIndex
x = sc.parallelize({'a','b','c','d'},3);
out = x.zipWithIndex().collect(); % {'a',0},{'b',1},{'c',2},{'d',3}
```

## See Also

[parallelize](#) | [zip](#) | [zipWithUniqueId](#)

**Introduced in R2016b**

## zipWithUniqueId

**Class:** matlab.compiler.mlspark.RDD

**Package:** matlab.compiler.mlspark

Zip an RDD with generated unique Long IDs

### Syntax

```
result = zipWithUniqueId(obj)
```

### Description

`result = zipWithUniqueId(obj)` zips `obj` with generated unique Long IDs.

### Input Arguments

**obj — Input RDD**

RDD object

An input RDD, specified as a RDD object.

### Output Arguments

**result — Output RDD**

RDD object

An output pipelined RDD, returned as a RDD object.

### Examples

#### Zip an RDD With Unique Long IDs

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% zipWithUniqueId
x = sc.parallelize({'a','b','c','d','e'},3);
out = x.zipWithUniqueId().collect(); % {'a',0},{'b',1},{'c',4},{'d',2},{'e',5}
```

### See Also

[parallelize](#) | [zip](#) | [zipWithIndex](#)

**Introduced in R2016b**

## Methods — SparkContext

---

## addJar

**Class:** matlab.compiler.mlspark.SparkContext

**Package:** matlab.compiler.mlspark

Add JAR file dependency for all tasks that need to be executed in a SparkContext

### Syntax

```
addJar(sc, filePath)
```

### Description

`addJar(sc, filePath)` adds a JAR file from a location specified by `filePath` to all tasks that need to be executed in SparkContext `sc`.

### Input Arguments

**sc — SparkContext to use**

SparkContext object

The SparkContext to use, specified as a SparkContext object.

**filePath — Location of JAR file**

character vector | string

Location of JAR file, specified as a character vector enclosed in `' '`.

Data Types: char | string

### Examples

**Add JAR File**

Add a JAR file dependency for all tasks that need to be executed in a SparkContext.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...
    'Master', 'local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);
%% Add JAR file
sc.addJar('/share/myArchive.jar')
```

### See Also

**Introduced in R2016b**

# broadcast

**Class:** matlab.compiler.mlspark.SparkContext

**Package:** matlab.compiler.mlspark

Broadcast a read-only variable to the cluster

## Syntax

```
result = broadcast(sc,value)
```

## Description

`result = broadcast(sc,value)` broadcasts a read-only variable `value` to the cluster initialized by SparkContext `sc`.

## Input Arguments

**sc — SparkContext to use**

SparkContext object

The SparkContext to use, specified as a SparkContext object.

**value — Value to be broadcast**

any supported data type

The value to be broadcast, specified as any supported data type.

Data Types: `single` | `double` | `int8` | `int16` | `int32` | `int64` | `uint8` | `uint16` | `uint32` | `uint64` | `logical` | `char` | `struct` | `table` | `cell` | `function_handle` | `categorical` | `string`

## Output Arguments

**result — Broadcast variable**

Broadcast object

Broadcast variable, returned as a broadcast object with the following properties:

- `id`
- `value`
- `path`

## Examples

### Broadcast a Variable

Broadcast a read-only variable to the cluster.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...
    'Master', 'local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% broadcast
myRdd = sc.parallelize({1,2,3,4,5});
myBroadcast = sc.broadcast('Hello, World');
newRdd = myRdd.map(@(x) myBroadcast.value);
countdata = newRdd.collect()
```

### See Also

**Introduced in R2016b**



# delete

**Class:** matlab.compiler.mlspark.SparkContext

**Package:** matlab.compiler.mlspark

Shutdown connection to Spark enabled cluster

## Syntax

```
delete(sc)
```

## Description

`delete(sc)` deletes a `SparkContext` `sc` and shuts down the connection to Spark enabled cluster.

## Input Arguments

**sc — SparkContext to use**

`SparkContext` object

The `SparkContext` to use, specified as a `SparkContext` object.

## Examples

### Delete a SparkContext

Delete a `SparkContext` object.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% delete
delete(sc)
```

## See Also

**Introduced in R2016b**

## datastoreToRDD

**Class:** matlab.compiler.mlspark.SparkContext

**Package:** matlab.compiler.mlspark

Convert MATLAB datastore to a Spark RDD

### Syntax

```
rdd = datastoreToRDD(sc, ds)
```

### Description

`rdd = datastoreToRDD(sc, ds)` converts a MATLAB datastore object `ds` to a Spark RDD.

### Input Arguments

**sc — SparkContext to use**

SparkContext object

The SparkContext to use, specified as a SparkContext object.

**ds — Datastore to be converted**

MATLAB datastore

Datastore to be converted to an RDD, specified as a MATLAB datastore object.

### Output Arguments

**rdd — Output RDD**

RDD object

Output RDD representing the converted datastore object, returned as a RDD object.

### Examples

#### Convert MATLAB Datastore to Spark RDD

Convert a MATLAB datastore object to a Spark RDD.

```
% Setup Spark Properties as a MATLAB Map object using a containers.Map class  
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
```

```
% Create SparkConf object  
conf = matlab.compiler.mlspark.SparkConf(...  
    'AppName'           , 'myApp', ...  
    'Master'            , 'local[1]', ...  
    'SparkProperties', sparkProp );
```

```
% Create a SparkContext
```

```
sc = matlab.compiler.mlspark.SparkContext(conf);  
  
% Create a MATLAB datastore  
ds = datastore('airlinesmall.csv','TreatAsMissing','NA');  
  
% Convert MATLAB datastore to Spark RDD  
rdd = datastoreToRDD(sc,ds);  
  
% Alternate object usage:  
rdd = sc.datastoreToRDD(ds);
```

## See Also

**Introduced in R2016b**

## getSparkConf

**Class:** matlab.compiler.mlspark.SparkContext

**Package:** matlab.compiler.mlspark

Get SparkConf configuration parameters

### Syntax

```
conf = getSparkConf(sc)
```

### Description

`conf = getSparkConf(sc)` retrieves the SparkConf configuration parameters.

### Input Arguments

**sc — SparkContext to use**

SparkContext object

The SparkContext to use, specified as a SparkContext object.

### Output Arguments

**conf — Spark configuration parameters**

SparkConf object

Spark configuration parameters, returned as a SparkConf object with properties.

### Examples

#### Get SparkConf Configuration Parameters

Retrieve the SparkConf configuration parameters.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% getSparkConf
getSparkConf(sc) % Alternate Usage: >> sc.getSparkConf();
```

### See Also

**Introduced in R2016b**

# parallelize

**Class:** matlab.compiler.mlspark.SparkContext

**Package:** matlab.compiler.mlspark

Create an RDD from a collection of local MATLAB values

## Syntax

```
rdd = parallelize(sc,cellArray)
rdd = parallelize(sc,cellArray,numSlices)
```

## Description

`rdd = parallelize(sc,cellArray)` creates an RDD from a collection of local MATLAB values grouped as a cell array.

`rdd = parallelize(sc,cellArray,numSlices)` creates an RDD with the number of partitions specified by `numSlices`.

## Input Arguments

### **sc** — SparkContext to use

SparkContext object

The SparkContext to use, specified as a SparkContext object.

### **cellArray** — Collection of values

cell array

A collection of values, specified as a MATLAB cell array.

Data Types: `cell`

### **numSlices** — Number of partitions to create

scalar

Number of partitions to create, specified as a scalar.

Data Types: `double`

## Output Arguments

### **rdd** — Output RDD created from the collection of values

RDD object

An output RDD, created from the collection of values, and returned as an RDD object.

## Examples

### Create an RDD From MATLAB Values

Create an RDD from local MATLAB values.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% parallelize
x = sc.parallelize({1, 2, 3, 4, 5});
y = x.count()
```

### See Also

**Introduced in R2016b**

# setCheckpointDir

**Class:** matlab.compiler.mlspark.SparkContext

**Package:** matlab.compiler.mlspark

Set the directory under which RDDs are to be checkpointed

## Syntax

```
setCheckpointDir(sc,dirName)
```

## Description

`setCheckpointDir(sc,dirName)` set the directory `dirName` under which RDDs are to be checkpointed.

## Input Arguments

**sc — SparkContext to use**

SparkContext object

The SparkContext to use, specified as a SparkContext object.

**dirName — Directory where RDDs are to be checkpointed**

character vector | string

Directory where the RDDs are to be checkpointed, specified as a character vector enclosed in ' '.

Data Types: char | string

## Examples

### Set Checkpoint Directory

Set the directory under which RDDs are to be checkpointed.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties', sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

%% setCheckpointDir
sc.setCheckpointDir('myDir')
```

## See Also

**Introduced in R2016b**

## setLogLevel

**Class:** matlab.compiler.mlspark.SparkContext

**Package:** matlab.compiler.mlspark

Set log level

### Syntax

```
setLogLevel(sc, logLevel)
```

### Description

`setLogLevel(sc, logLevel)` sets the log level to one of eight possible options.

### Input Arguments

**sc** — SparkContext to use

SparkContext object

The SparkContext to use, specified as a SparkContext object.

**logLevel** — Log level to set

'ALL' | 'DEBUG' | 'ERROR' | 'FATAL' | 'INFO' | 'OFF' | 'TRACE' | 'WARN'

Log level, specified as one of the following values:

- 'ALL'
- 'DEBUG'
- 'ERROR'
- 'FATAL'
- 'INFO'
- 'OFF'
- 'TRACE'
- 'WARN'

Data Types: char | string

### Examples

#### Set Log Level

Set the log level for execution against Spark.

```
%% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName', 'myApp', ...
```



```
sc = matlab.compiler.mlspark.SparkContext('Master','local[1]','SparkProperties',sparkProp);  
%% Set log level  
sc.setLogLevel('OFF');
```

## See Also

**Introduced in R2016b**

## textFile

**Class:** matlab.compiler.mlspark.SparkContext

**Package:** matlab.compiler.mlspark

Create an RDD from a text file

### Syntax

```
rdd = textFile(sc,pathToFile)
rdd = textFile(sc,pathToFile,minPartitions)
```

### Description

`rdd = textFile(sc,pathToFile)` creates an RDD from a text file located in `pathToFile`.

`rdd = textFile(sc,pathToFile,minPartitions)` creates an RDD with minimum partitions specified by `minPartitions`.

### Input Arguments

#### **sc — SparkContext to use**

SparkContext object

The SparkContext to use, specified as a SparkContext object.

#### **pathToFile — File path to text file**

character vector | string

File path to text file, specified as a character vector enclosed in ' '.

Data Types: char | string

#### **minPartitions — Minimum number of partitions to be created**

scalar

Minimum number of partitions to be created, specified as a scalar.

Data Types: double

### Output Arguments

#### **result — Output RDD created from text file**

RDD object

An output RDD created from reading in a text file, returned as an RDD object.

### Examples

## Create RDD from Text File

```
% Connect to Spark
sparkProp = containers.Map({'spark.executor.cores'}, {'1'});
conf = matlab.compiler.mlspark.SparkConf('AppName','myApp', ...
    'Master','local[1]', 'SparkProperties',sparkProp);
sc = matlab.compiler.mlspark.SparkContext(conf);

% textFile
rdd = sc.textFile('/<matlabroot>/toolbox/matlab/demos/airlinesmall.csv')
```

## See Also

**Introduced in R2016b**



# Apache Spark Basics

---

## Apache Spark Basics

### In this section...

“Running against Spark” on page 4-2

“Cluster Managers Supported by Spark” on page 4-3

“Relationship Between Spark and Hadoop” on page 4-4

“Driver” on page 4-4

“Executor” on page 4-5

“RDD” on page 4-5

“Transformations” on page 4-5

“Actions” on page 4-5

“Distinguishing Between Transformations and Actions” on page 4-5

“SparkConf” on page 4-5

“SparkContext” on page 4-6

*Apache Spark* is a fast, general-purpose engine for large-scale data processing.

Every Spark application consists of a driver program that manages the execution of your application on a cluster. The workers on a Spark enabled cluster are referred to as executors. The driver process runs the user code on these executors.

In a typical Spark application, your code will establish a SparkContext, create a Resilient Distributed Dataset (RDD) from external data, and then execute methods known as transformations and actions on that RDD to arrive at the outcome of an analysis.

An RDD is the main programming abstraction in Spark and represents an immutable collection of elements partitioned across the nodes of a cluster that can be operated on in parallel. A Spark application can run locally on a single machine or on a cluster.

Spark is mainly written in Scala and has APIs in other programming languages, including MATLAB. The MATLAB API for Spark exposes the Spark programming model to MATLAB and enables MATLAB implementations of numerous Spark functions. Many of these MATLAB implementations of Spark functions accept function handles or anonymous functions as inputs to perform various types of analyses.

### Running against Spark

To run against Spark means executing an application against a Spark enabled cluster using a supported cluster manager. A cluster can be local or on a network. You can run against Spark in two ways:

- Execute commands in an *interactive shell* that is connected to Spark.
- Create and execute a *standalone application* against a Spark cluster.

When using an interactive shell, Spark allows you to interact with data that is distributed on disk or in memory across many machines and perform ad-hoc analysis. Spark takes care of the underlying distribution of work across various machines. Interactive shells are only available in Python® and Scala.

The MATLAB API for Spark in MATLAB Compiler provides an interactive shell similar to a Spark shell that allows you to debug your application prior to deploying it. The interactive shell only runs against a local cluster.

When creating and executing standalone applications against Spark, applications are first packaged or compiled as standalone applications before being executed against a Spark enabled cluster. You can author standalone applications in Scala, Java®, Python, and MATLAB.

The MATLAB API for Spark in MATLAB Compiler lets you create standalone applications that can run against Spark.

## Cluster Managers Supported by Spark

### Local

A local cluster manager represents a pseudo-cluster and works in a nondistributed mode on a single machine. You can configure it to use one worker thread, or on a multicore machine, multiple worker threads. In applications, it is denoted by the word `local`.

---

**Note** The MATLAB API for Spark, which allows you to interactively debug your applications, works only with a local cluster manager.

---

### Standalone

A Standalone cluster manager ships with Spark. It consists of a master and multiple workers. To use a Standalone cluster manager, place a compiled version of Spark on each cluster node. A Standalone cluster manager can be started using scripts provided by Spark. In applications, it is denoted as: `spark://host:port`. The default port number is `7077`.

---

**Note** The Standalone cluster manager that ships with Spark is not to be confused with the *standalone application* that can run against Spark. MATLAB Compiler does not support the Standalone cluster manager.

---

### YARN

A YARN cluster manager was introduced in Hadoop 2.0. It is typically installed on the same nodes as HDFS. Therefore, running Spark on YARN lets Spark access HDFS data easily. In applications, it is denoted using the term `yarn`. There are two modes that are available when starting applications on YARN:

- In `yarn-client` mode, the driver runs in the client process, and the application master is used only for requesting resources from YARN.
- In `yarn-cluster` mode, the Spark driver runs inside an application master process that is managed by YARN on the cluster, and the client can retire after initiating the application.

---

**Note** MATLAB Compiler supports the YARN cluster manager only in `yarn-client` mode.

---

## Mesos

A Mesos cluster manager is an open-source cluster manager developed by Apache™. In applications, it is usually denoted as: `mesos://host:port`. The default port number is 5050.

---

**Note** MATLAB Compiler does not support a Mesos cluster manager.

---

You can use the following table to see which MATLAB Compiler deployment option is supported by each cluster manager.

Deploy Against Spark Option	Local Cluster (local)	Hadoop Cluster (yarn-client)
Deploy standalone applications containing tall arrays on page 6-2.	Not supported.	Supported.
Deploy standalone applications created using the MATLAB API for Spark on page 7-2.	Supported.	Supported.
Interactively debug your applications using the MATLAB API for Spark on page 7-2.	Supported.	Not supported.

## Relationship Between Spark and Hadoop

The relationship between Spark and Hadoop comes into play only if you want to run Spark on a cluster that has Hadoop installed. Otherwise, you do not need Hadoop to run Spark.

To run Spark on a cluster you need a shared file system. A Hadoop cluster provides access to a distributed file-system via HDFS and a cluster manager in the form of YARN. Spark can use YARN as a cluster manager for distributing work and use HDFS to access data. Also, some Spark applications can use Hadoop's MapReduce programming model, but MapReduce does not constitute the core programming model in Spark.

Hadoop is not required to run Spark on cluster. You can also use other options such as Mesos.

---

**Note** The deployment options in MATLAB Compiler currently support deploying only against a Spark enabled Hadoop cluster.

---

## Driver

Every Spark application consists of a *driver* program that initiates various operations on a cluster. The driver is a process in which the `main()` method of a program runs. The driver process runs user code that creates a `SparkContext`, creates RDDs, and performs transformations and actions. When a Spark driver executes, it performs two duties:

- Convert a user program into tasks.

The Spark driver application is responsible for converting a user program into units of physical execution called tasks. Tasks are the smallest unit of work in Spark.



- Schedule tasks on executors.

The Spark driver tries to schedule each task in an appropriate location, based on data placement. It also tracks the location of cached data, and uses it to schedule future tasks that access that data.

Once the driver terminates, the application is finished.

---

**Note** When using the MATLAB API for Spark in MATLAB Compiler, MATLAB application code becomes the Spark driver program.

---

## Executor

A Spark executor is a worker process responsible for running the individual tasks in a given Spark job. Executors are started at the beginning of a Spark application and persist for the entire lifetime of an application. Executors perform two roles:

- Run the tasks that make up the application, and return the results to the driver.
- Provide in-memory storage for RDDs that are cached by user programs.

## RDD

A Resilient Distributed Dataset or RDD is a programming abstraction in Spark. It represents a collection of elements distributed across many nodes that can be operated in parallel. RDDs tend to be fault-tolerant. You can create RDDs in two ways:

- By loading an external dataset.
- By parallelizing a collection of objects in the *driver* on page 4-4 program.

After creation, you can perform two types of operations using RDDs: transformations and actions.

## Transformations

Transformations are operations on an existing RDD that return a new RDD. Many, but not all, transformations are element-wise operations.

## Actions

Actions compute a final result based on an RDD and either return that result to the driver program or save it to an external storage system such as HDFS.

## Distinguishing Between Transformations and Actions

Check the return data type. Transformations return RDDs, whereas actions return other data types.

## SparkConf

SparkConf stores the configuration parameters of the application being deployed to Spark. Every application must be configured prior to being deployed on a Spark cluster. Some of the configuration

parameters define properties of the application and some are used by Spark to allocate resources on the cluster. The configuration parameters are passed onto a Spark cluster through a `SparkContext`.

### **SparkContext**

A `SparkContext` represents a connection to a Spark cluster. It is the entry point to Spark and sets up the internal services necessary to establish a connection to the Spark execution environment.

# Configure MATLAB Environment for Spark Deployment

---

## Configure Environment for Interactive Debugging

**Supported Platform:** Linux<sup>®</sup> only.

Before you can interactively debug your applications using the MATLAB API for Spark, you must configure your MATLAB environment by adding the location of the Spark assembly jar to the front of MATLAB's static Java class path.

A Spark assembly JAR file includes all the Spark dependencies, including some internal Hadoop dependencies. This JAR file is created during Spark installation and is specific to a particular combination of Spark and Hadoop versions. For example, using Spark version 1.4.0 and Hadoop version 2.4.0 creates the file `spark-assembly-1.4.0-hadoop2.4.0.jar`.

To add the location of the Spark assembly jar to the front of MATLAB's static Java class path, use the following procedure:

- 1 Open a Linux terminal.
- 2 Navigate to the directory `<matlabroot>/bin`, where `<matlabroot>` is the location where MATLAB is installed.
- 3 Set the environment variable `SPARK_HOME` to point to the Spark installation directory. For example:

Shell	Command
<code>csh / tcsh</code>	<code>% setenv SPARK_HOME /share/spark</code>
<code>bash</code>	<code>\$ export SPARK_HOME=/share/spark</code>

- 4 From the directory `<matlabroot>/bin` execute the shell script `generatesparkjcpath`. For example:

```
$ ./generatesparkjcpath
```

This will create a file called `javaclasspath.txt` in your `<matlabroot>/bin` directory. The contents of a sample `javaclasspath.txt` file look like this:

```
<before>
/share/spark/conf/
/share/spark/lib/spark-assembly-1.4.0-hadoop2.4.0.jar
/share/spark/lib/datanucleus-rdbms-3.2.9.jar
/share/spark/lib/datanucleus-api-jdo-3.2.6.jar
/share/spark/lib/datanucleus-core-3.2.10.jar
```

The `<before>` tag ensures that `spark-assembly-1.4.0-hadoop2.4.0.jar` is added to the front of the Java class path.

- 5 Start MATLAB from the `<matlabroot>/bin` directory.

**Note** You can create the `javaclasspath.txt` file in a directory other than `<matlabroot>/bin` by specifying the `-o` option. The file has to be named `javaclasspath.txt`. For example:

```
$ ./generatesparkjcpath -o /home/user/javaclasspath.txt
```

Start MATLAB from the directory `/home/user`. It is important that you start MATLAB from the directory containing the `javaclasspath.txt` file.

You can verify whether `spark-assembly-1.4.0-hadoop2.4.0.jar` was added to the front of the Java class path by typing the following at the MATLAB command prompt:

```
>> spath = javaclasspath('-static') ;  
>> spath(1:5)
```

You should see `/share/spark/lib/spark-assembly-1.4.0-hadoop2.4.0.jar` on the top of the list.

You need to complete this configuration just once. Subsequent usage only requires that you start MATLAB from the folder containing the `javaclasspath.txt` file.

## See Also

### Related Examples

- “Example on Deploying Applications to Spark Using the MATLAB API for Spark” on page 7-2



# Deploy Tall Arrays to a Spark enabled Hadoop Cluster

---

- “Example on Deploying Tall Arrays to a Spark Enabled Hadoop Cluster” on page 6-2
- “Deploying Tall Arrays to a Cloudera Spark Enabled Hadoop Cluster” on page 6-8

## Example on Deploying Tall Arrays to a Spark Enabled Hadoop Cluster

**Supported Platform:** Linux only.

This example shows how to deploy a MATLAB application containing tall arrays to a Spark enabled Hadoop cluster.

**Goal:** Compute the mean arrival delay and the biggest arrival delays of airlines from the given dataset.

<b>Dataset:</b>	airlinesmall.csv
<b>Description:</b>	Airline departure and arrival information from 1987-2008.
<b>Location:</b>	/usr/local/MATLAB/R2020a/toolbox/matlab/demos

**Note** You can follow the same instructions to deploy tall array Spark applications to Cloudera® CDH. To see an example on MATLAB Answers™, click here.

To use Cloudera CDH encryption zones, add the JAR file `commons-codec-1.9.jar` to the static classpath of MATLAB Runtime. Location of the file: `$HADOOP_PREFIX/lib/commons-codec-1.9.jar`, where `$HADOOP_PREFIX` is the location where Hadoop is installed.

**Note** If you are using Spark version 1.6 or higher, you will need to increase the Java heap size in MATLAB to at least 512MB. For information on how to increase the Java heap size in MATLAB, see “Java Heap Memory Preferences” (MATLAB).

### Prerequisites

- 1 Start this example by creating a new work folder that is visible to the MATLAB search path.
- 2 Install the MATLAB Runtime in a folder that is accessible by every worker node in the Hadoop cluster. This example uses `/usr/local/MATLAB/MATLAB_Runtime/v##` as the location of the MATLAB Runtime folder.

If you don't have the MATLAB Runtime, you can download it from the website at: <https://www.mathworks.com/products/compiler/mcr>.

**Note** Replace all references to the MATLAB Runtime version `v##` in this example with the MATLAB Runtime version number corresponding to your MATLAB release. For example, MATLAB R2017b has MATLAB Runtime version number `v92`. For information about MATLAB Runtime version numbers corresponding MATLAB releases, see this list.

- 3 Copy the file `airlinesmall.csv` into Hadoop Distributed File System (HDFS) folder `/user/<username>/datasets`. Here `<username>` refers to your user name in HDFS.

```
$ ./hadoop fs -copyFromLocal airlinesmall.csv hdfs://host:54310/user/<username>/datasets
```

### Procedure

- 1 Set up the environment variable, `HADOOP_PREFIX` to point at your Hadoop install folder. These properties are necessary for submitting jobs to your Hadoop cluster.



```
setenv('HADOOP_PREFIX', '/usr/lib/hadoop')
```

The HADOOP\_PREFIX environment variable must be set when using the MATLAB `datastore` function to point to data on HDFS. Setting this environment variable has nothing to do with Spark. See “Relationship Between Spark and Hadoop” on page 4-4 for more information.

If you plan on using a dataset that’s on your local machine as opposed to one on HDFS, then you can skip this step.

---

**Note** This example uses `/usr/lib/hadoop` as directory where Hadoop is installed. Your Hadoop installation directory maybe different.

---

## 2 Specify Spark properties.

Use a `containers.Map` object to specify Spark properties.

```
sparkProperties = containers.Map( ...
    {'spark.executor.cores', ...
     'spark.executor.memory', ...
     'spark.yarn.executor.memoryOverhead', ...
     'spark.dynamicAllocation.enabled', ...
     'spark.shuffle.service.enabled', ...
     'spark.eventLog.enabled', ...
     'spark.eventLog.dir'}, ...
    {'1', ...
     '2g', ...
     '1024', ...
     'true', ...
     'true', ...
     'true', ...
     'hdfs://host:54310/user/<username>/log'});
```

For more information on Spark properties, expand the `prop` value of the `'SparkProperties'` name-value pair in the “Input Arguments” on page 1-6 section of the `SparkConf` class. The `SparkConf` class is part of the MATLAB API for Spark, which provides an alternate way to deploy MATLAB applications to Spark. For more information, see “Deploy Applications Using the MATLAB API for Spark”.

## 3 Configure your MATLAB application containing tall arrays with Spark parameters.

Use the class `matlab.mapreduce.DeploySparkMapReducer` to configure your MATLAB application containing tall arrays with Spark parameters as key-value pairs.

```
conf = matlab.mapreduce.DeploySparkMapReducer( ...
    'AppName', 'myTallApp', ...
    'Master', 'yarn-client', ...
    'SparkProperties', sparkProperties);
```

For more information, see `matlab.mapreduce.DeploySparkMapReducer`.

## 4 Define the Spark execution environment.

Use the `mapreducer` function to define the Spark execution environment.

```
mapreducer(conf)
```

For more information, see `mapreducer`.

## 5 Include your MATLAB application code containing tall arrays.

Use the MATLAB function `datastore` to create a datastore object pointing to the file `airlinesmall.csv` in HDFS. Pass the datastore object as an input argument to the `tall` function. This will create a tall array. You can perform operations on the tall array to compute the mean arrival delay and the biggest arrival delays.

```
% Create a |datastore| for a collection of tabular text files representing airline data.
% Select the variables of interest, specify a categorical data type for the
% |Origin| and |Dest| variables.
% ds = datastore('airlinesmall.csv') % if using a dataset on your local machine
ds = datastore('hdfs:///<username>/datasets/airlinesmall.csv');
ds.TreatAsMissing = 'NA';
ds.SelectedVariableNames = {'Year', 'Month', 'ArrDelay', 'DepDelay', 'Origin', 'Dest'};
ds.SelectedFormats(5:6) = {'%C', '%C'};

% Create Tall Array
% Tall arrays are like normal MATLAB arrays, except that they can have any
% number of rows. When a |tall| array is backed by a |datastore|, the underlying class of
% the tall array is based on the type of datastore.
tt = tall(ds);

% Remove Rows with Missing Data or NaN Values
idx = any(ismissing(tt),2);
tt(idx,:) = [];

% Compute Mean Delay
meanArrivalDelay = mean(tt.DepDelay, 'omitnan');
biggestDelays = topkrows(tt,10, 'ArrDelay');

% Gather Results
% The |gather| function forces evaluation of all queued operations and
% brings the resulting output back into memory.
[meanArrivalDelay, biggestDelays] = gather(meanArrivalDelay, biggestDelays)

% Delete mapreducer object
delete(conf);
```

## 6 Create a Spark application.

Use the `mcc` command with the `-vCW` options to create a Spark application.

```
>> mcc -vCW 'Spark:myTallApp' deployTallArrayToSpark.m
```

The following files are created.

Files	Description
<code>run_myTallApp.sh</code>	Shell script to run application. The script invokes <code>spark-submit</code> to launch the application on the cluster.
<code>myTallApp.jar</code>	Application JAR. The application JAR contains packaged MATLAB code and other dependencies.
<code>readme.txt</code>	Readme file containing details on how to run the application.
<code>requiredMCRProducts.txt</code>	
<code>mccExcludedFiles.log</code>	

For more information, see `mcc`.

- 7 Run the application from a Linux shell using the following command:

```
$ ./run_myTallApp.sh /usr/local/MATLAB/MATLAB_Runtime/v##
```

/usr/local/MATLAB/MATLAB\_Runtime/v## is an argument indicating the location of the MATLAB Runtime.

- 8 You will see the following output:

```
meanArrivalDelay =
    7.1201
```

```
biggestDelays =
```

```
10x5 table
```

Year	Month	ArrDelay	Origin	Dest
1995	11	1014	HNL	LAX
2007	4	914	JFK	DTW
2001	4	887	MCO	DTW
2008	7	845	CMH	ORD
1988	3	772	ORD	LEX
2008	4	710	EWR	RDU
1998	10	679	MCI	DFW
2006	6	603	ABQ	PHX
2008	6	586	PIT	LGA
2007	4	568	RNO	SLC

Optionally, if you want to analyze or view the results generated by your application in MATLAB, you need to write the results to a file on HDFS using the `write` function for tall arrays. You can then read the file using the `datastore` function.

To write the results to file on HDFS, add the following line of code to your MATLAB application just before the `delete(conf)` statement and then package your application:

```
write('hdfs:///user/<username>/results', tall(biggestDelays));
```

Replace `<username>` with your user name.

You can only save one variable to a file using the `write` function for tall arrays. Therefore, you will need to write to multiple files if you want to save multiple variables.

To view the results in MATLAB after executing the application against a Spark enabled cluster, use the `datastore` function as follows:

```
>> ds = datastore('hdfs:///user/<username>/results')
>> readall(ds)
```

You may need to set the environment variable `HADOOP_PREFIX` using the function `setenv` in case you are unable to view the results using the `datastore` function.

**Note** If the tall array application being deployed is a MATLAB function as opposed to a MATLAB script, use the following execution syntax:

```
$ ./run_<applicationName>.sh \
  <MATLAB_Runtime_Location> \
```

```
[Spark arguments] \  
[Application arguments]
```

For example:

```
$ ./run_myTallApp.sh \  
  /usr/local/MATLAB/MATLAB_Runtime/v92 \  
  yarn-client \  
  hdfs://host:54310/user/<username>/datasets/airlinesmall.csv \  
  hdfs://host:54310/user/<username>/result
```

---

### Code:

#### deployTallArrayToSpark.m

```
%% Set Hadoop environemnt variables  
setenv('HADOOP_PREFIX', '/usr/lib/hadoop')  
  
%% Setup Spark Properties  
% Use a containers.Map object to specify Spark properties.  
sparkProperties = containers.Map( ...  
    {'spark.executor.cores', ...  
     'spark.executor.memory', ...  
     'spark.yarn.executor.memoryOverhead', ...  
     'spark.dynamicAllocation.enabled', ...  
     'spark.shuffle.service.enabled', ...  
     'spark.eventLog.enabled', ...  
     'spark.eventLog.dir'}, ...  
    {'1', ...  
     '2g', ...  
     '1024', ...  
     'true', ...  
     'true', ...  
     'true', ...  
     'hdfs://host:54310/user/<username>/log'});  
  
%% Configure a MATLAB Tall Array Application  
% Use the class matlab.mapreduce.DeploySparkMapReducer to configure the  
% MATLAB tall array application with Spark parameters as key-value pairs.  
conf = matlab.mapreduce.DeploySparkMapReducer( ...  
    'AppName', 'myTallApp', ...  
    'Master', 'yarn-client', ...  
    'SparkProperties', sparkProperties);  
  
%% Define the Spark Execution Environment  
% Use the mapreducer function to define the Spark execution environment.  
mapreducer(conf);  
  
%% Create a |datastore| for a collection of tabular text files representing airline data.  
% Select the variables of interest, specify a categorical data type for the  
% |Origin| and |Dest| variables.  
ds = datastore('hdfs://host:54310/user/<username>/datasets/airlinesmall.csv');  
ds.TreatAsMissing = 'NA';  
ds.SelectedVariableNames = {'Year', 'Month', 'ArrDelay', 'Origin', 'Dest'};  
ds.SelectedFormats(4:5) = {'%C', '%C'};  
  
%% Create Tall Array
```

```
% Tall arrays are like normal MATLAB arrays, except that they can have any
% number of rows. When a |tall| array is backed by a |datastore|, the underlying class of
% the tall array is based on the type of datastore.
tt = tall(ds);

%% Remove Rows with Missing Data or NaN Values
idx = any(ismissing(tt),2);
tt(idx,:) = [];

%% Compute Mean Delay
meanArrivalDelay = mean(tt.ArrDelay,'omitnan');
biggestDelays = topkrows(tt,10,'ArrDelay');

%% Gather Results
% The |gather| function forces evaluation of all queued operations and
% brings the resulting output back into memory.
[meanArrivalDelay,biggestDelays] = gather(meanArrivalDelay,biggestDelays)

%% Write results to HDFS
write('hdfs:///user/<username>/results', tall(meanArrivalDelay));
write('hdfs:///user/<username>/results', tall(biggestDelays));

%% Delete mapreducer object
delete(conf);
```

## Deploying Tall Arrays to a Cloudera Spark Enabled Hadoop Cluster

### In this section...

“Prerequisites” on page 6-9

“Deploy Tall Arrays” on page 6-9

This example shows how to deploy a MATLAB application containing tall arrays to a Cloudera Spark enabled Hadoop cluster.

Deploying MATLAB applications against a Cloudera distribution of Spark requires a special wrapper type that you generate using the `mcc` command. This wrapper type generates a `jar` file as well as a shell script which calls `spark-submit`. The `spark-submit` script in the `Sparkbin` directory is used to start applications on a cluster. It supports both `yarn-client` mode and `yarn-cluster` mode.

The inputs to the application are:

- `master` — URL to the Spark cluster
- `inputFile` — the file containing the input data
- `outputFile` — the file containing the results of the computation

**Note** The complete code for this example is in the file `meanArrivalDemo.m`, as shown below.

### meanArrivalDemo.m

```
%
% Copyright 2016 MathWorks, Inc
%
function meanArrivalDemo(master,inputFile,outputFile)
%%
% Set up datastore for airlinesmall.csv
%
ds = datastore( ...
    inputFile, ...
    'datastoretype','tabulartext', ...
    'TreatAsMissing','NA', ...
    'SelectedVariableNames','ArrDelay' ...
);
%%
% Set up spark to use yarn resource manager
% 1 thread per executor task
% 1 executor task
sparkProperties = containers.Map( ...
    {'spark.executor.cores','spark.executor.instances'}, ...
    {'1','1'});
%%
% Indicate that environment is deployment
mr = matlab.mapreduce.DeploySparkMapReducer( ...
    'SparkProperties', sparkProperties, ...
    'AppName', 'meanArrivalDemo', ...
    'Master', master ...
);
```

```
mapreducer(mr)
%%
% Create tall table and compute mean on ArrDelay column
%
t = tall(ds);
meanresult=mean(t.ArrDelay, 1, 'omitnan');
write(outputFile,meanresult);
```

## Prerequisites

- 1 Install the MATLAB Runtime in the default location on the desktop. This example uses `/usr/local/MATLAB/MATLAB_Runtime/v91` as the default location for the MATLAB Runtime. If you don't have MATLAB Runtime, see "Install and Configure the MATLAB Runtime" for installation instructions.
- 2 Install the MATLAB Runtime on every worker node.
- 3 Copy the `airlinesmall.csv` from folder `toolbox/matlab/demos` of your MATLAB install area into Hadoop Distributed File System (HDFS) folder `/datasets/airlinemod`.

## Deploy Tall Arrays

- 1 At the MATLAB command prompt, use the `mcc` command to generate a jar file and shell script for the MATLAB application `meanArrivalDemo.m`.

```
>> mcc -vCW 'Spark:meanArrivalDemoApp' meanArrivalDemo.m
```

This action creates a jar file named `meanArrivalDempApp.jar` and a shell script named `run_meanArrivalDemoApp.sh`.

---

**Note** To use the shell script, set up the environment variables `HADOOP_PREIX`, `HADOOP_CONF_DIR` and `SPARK_HOME`.

---

- 2 Execute the shell script in either `yarn-client` mode or `yarn-cluster` mode. In `yarn-client` mode, the driver runs on the desktop. In `yarn-cluster` mode, the driver runs in the Application Master process in the cluster.

The general syntax to execute the shell script is:

```
./run_meanArrivalDemoApp.sh <runtime install root> [Spark arguments] [Application arguments]
```

### yarn-client mode

Run the following command from a Linux terminal:

```
$ ./run_meanArrivalDemoApp.sh \
  /usr/local/MATLAB/MATLAB_Runtime/v91 \
  yarn-client \
  hdfs://hadoop01glnxa64:54310/datasets/airlinemod/airlinesmall.csv \
  hdfs://hadoop01glnxa64:54310/user/someuser/meanArrivalResult
```

To examine the result, enter the following from the MATLAB command prompt:

```
>> ds = datastore('hdfs:///user/someuser/meanArrivalResult/*');
>> readall(ds)
```

### yarn-cluster mode

Run the following command from a Linux terminal:

```
$ ./run_meanArrivalDemoApp.sh \  
  /usr/local/MATLAB/MATLAB_Runtime/v91 \  
  --deploy-mode cluster --master yarn yarn-cluster \  
  hdfs://hadoop01glnxa64:54310/datasets/airlinemod/airlinesmall.csv \  
  hdfs://hadoop01glnxa64:54310/user/someuser/meanArrivalResult
```

In `yarn-cluster` mode, since the driver is running on a worker node in the cluster, any standard output from the MATLAB function is not displayed on your desktop. In addition, files can be saved anywhere. To prevent such behavior, this example uses the `write` function to explicitly save the results to a particular location in HDFS.



# Deploy MATLAB Applications to Spark using the MATLAB API for Spark

---

- “Example on Deploying Applications to Spark Using the MATLAB API for Spark” on page 7-2
- “Deploying Applications to Cloudera Spark Using the MATLAB API for Spark” on page 7-11

## Example on Deploying Applications to Spark Using the MATLAB API for Spark

**Supported Platform:** Linux only.

This example shows you how to deploy a standalone application to Spark using the MATLAB API for Spark. Your application can be deployed against Spark using one of two supported cluster managers: local and Hadoop YARN. This example shows you how to deploy your application using both cluster managers. For a discussion on cluster managers, see “Cluster Managers Supported by Spark” on page 4-3.

In this section...
“Local” on page 7-2
“Hadoop YARN” on page 7-6

**Goal:** Count the number of unique airlines in the given dataset.

<b>Dataset:</b>	airlinesmall.csv
<b>Description:</b>	Airline departure and arrival information from 1987-2008.
<b>Location:</b>	/usr/local/MATLAB/R2020a/toolbox/matlab/demos

### Helper Function

Create a MATLAB file named `carrierToCount.m` with the following code:

```
function results = carrierToCount(input)
    tbl = input{1};
    intermKeys = tbl.UniqueCarrier;
    [intermKeys, ~, idx] = unique(intermKeys);
    intermValues = num2cell(accumarray(idx, ones(size(idx))));
    results = cellfun( @(x,y) {x,y} , ...
        intermKeys, intermValues, ...
        'UniformOutput', false);
```

This helper function is passed in as a function handle to one of the methods in the example.

---

**Note** If you are using Spark version 1.6 or higher, you will need to increase the Java heap size in MATLAB to at least 512MB. For information on how to increase the Java heap size in MATLAB, see “Java Heap Memory Preferences” (MATLAB).

---

### Local

A local cluster manager represents a pseudo Spark enabled cluster and works in a non-distributed mode on a single machine. It can be configured to use one worker thread, or on a multi-core machine, multiple worker threads. In applications, it is denoted by the word `local`. A local cluster manager is handy for debugging your application prior to full blown deployment on a Spark enabled Hadoop cluster.

**Prerequisites**

- 1 Start this example by creating a new work folder that is visible to the MATLAB search path.
- 2 Create the helper function `carrierToCount.m` mentioned above.

**Procedure**

- 1 Specify Spark properties.

Use a `containers.Map` object to specify Spark properties.

```
sparkProp = containers.Map(...
    {'spark.executor.cores', ...
     'spark.matlab.worker.debug'}, ...
    {'1', ...
     'true'});
```

Spark properties indicate the Spark execution environment of the application that is being deployed. Every application must be configured with specific Spark properties in order for it to be deployed.

For more information on Spark properties, expand the `prop` value of the 'SparkProperties' name-value pair in the "Input Arguments" on page 1-6 section of the `SparkConf` class.

- 2 Create a `SparkConf` object.

Use the class `matlab.compiler.mlspark.SparkConf` to create a `SparkConf` object. A `SparkConf` object stores the configuration parameters of the application being deployed to Spark. The configuration parameters of an application are passed onto a Spark cluster through a `SparkContext`.

```
conf = matlab.compiler.mlspark.SparkConf(...
    'AppName', 'mySparkAppDepLocal', ...
    'Master', 'local[1]', ...
    'SparkProperties', sparkProp );
```

For more information on `SparkConf`, see `matlab.compiler.mlspark.SparkConf`.

- 3 Create a `SparkContext` object.

Use the class `matlab.compiler.mlspark.SparkContext` with the `SparkConf` object as an input to create a `SparkContext` object.

```
sc = matlab.compiler.mlspark.SparkContext(conf);
```

A `SparkContext` object serves as an entry point to Spark by initializing a connection to a Spark cluster. It accepts a `SparkConf` object as an input argument and uses the parameters specified in that object to set up the internal services necessary to establish a connection to the Spark execution environment.

For more information on `SparkContext`, see `matlab.compiler.mlspark.SparkContext`.

- 4 Create an RDD object from the data.

Use the MATLAB function `datastore` to create a `datastore` object pointing to the file `airlinesmall.csv`. Then use the `SparkContext` method `datastoreToRDD` to convert the `datastore` object to a Spark RDD object.

```
% Create a MATLAB datastore (LOCAL)
ds = datastore('airlinesmall.csv', ...
```

```

    'TreatAsMissing','NA', ...
    'SelectedVariableNames','UniqueCarrier');
% Convert MATLAB datastore to Spark RDD
rdd = sc.datastoreToRDD(ds);

```

In general, input RDDs can be created using the following methods of the `SparkContext` class: `parallelize` on page 3-9, `datastoreToRDD` on page 3-6, and `textFile` on page 3-14.

**5** Perform operations on the RDD object.

Use a Spark RDD method such as `flatMap` on page 2-24 to apply a function to all elements of the RDD object and flatten the results. The function `carrierToCount` that was created earlier serves as the function that is going to be applied to the elements of the RDD. A function handle to the function `carrierToCount` is passed as an input argument to the `flatMap` method.

```

maprdd = rdd.flatMap(@carrierToCount);
redrdd = maprdd.reduceByKey( @(acc,value) acc+value );
countdata = redrdd.collect();

% Count and display carrier occurrences
count = 0;
for i=1:numel(countdata)
    count = count + countdata{i}{2};
    fprintf('\nCarrier Name: %s, Count: %d', countdata{i}{1}, countdata{i}{2});
end
fprintf('\n Total count : %d\n', count);

% Delete Spark Context
delete(sc)

```

In general, you will provide MATLAB functions handles or anonymous functions as input arguments to Spark RDD methods known as transformations and actions. These function handles and anonymous functions are executed on the workers of the deployed application.

For a list of supported RDD transformations and actions, see “Transformations” on page 1-3 and “Actions” on page 1-4 in the Methods section of the RDD class.

For more information on transformations and actions, see “Apache Spark Basics” on page 4-2.

**6** Create a standalone application.

Use the `mcc` command with the `-m` flag to create a standalone application. The `-m` flag creates a standard executable that can be run from a command line. The `-a` flag includes the dependent dataset `airlinesmall.csv` from the folder `<matlabroot>/toolbox/matlab/demos`. The `mcc` command automatically picks up the dependent file `carrierToCount.m` as long as it is in the same work folder.

```
>> mcc -m deployToSparkMlApiLocal.m -a <matlabroot>/toolbox/matlab/demos/airlinesmall.csv
```

The `mcc` command creates a shell script `run_deployToSparkMlApiLocal.sh` to run the executable file `deployToSparkMlApiLocal`.

For more information, see `mcc`.

**7** Run the standalone application from a Linux shell using the following command:

```
$ ./run_deployToSparkMlApiLocal.sh /share/MATLAB/MATLAB_Runtime/v91
```

`/share/MATLAB/MATLAB_Runtime/v91` is an argument indicating the location of the MATLAB Runtime.

Prior to executing the above command, make sure the `javaclasspath.txt` file is in the same folder as the shell script and the executable.

Your application will fail to execute if it cannot find the file `javaclasspath.txt`.

Your application may also fail to execute if the optional line containing the folder location of the Hadoop configuration files is uncommented. To execute your application on the `local` cluster manager, this line must be commented. This line should only be uncommented if you plan on running your application using `yarn-client` as your cluster manager on a Spark enabled Hadoop cluster.

**8** You will see the following output:

```
Carrier Name: 9E, Count: 521
Carrier Name: AA, Count: 14930
Carrier Name: AQ, Count: 154
Carrier Name: AS, Count: 2910
Carrier Name: B6, Count: 806
Carrier Name: CO, Count: 8138
...
...
...
Carrier Name: US, Count: 13997
Carrier Name: WN, Count: 15931
Carrier Name: XE, Count: 2357
Carrier Name: YV, Count: 849
Total count : 123523
```

**Code:**

### **deployToSparkMIApiLocal.m**

```
% Setup Spark Properties as a MATLAB Map object using a containers.Map class
sparkProp = containers.Map(...
    {'spark.executor.cores',...
     'spark.matlab.worker.debug'},...
    {'1',...
     'true'});

% Create SparkConf object
conf = matlab.compiler.mlspark.SparkConf(...
    'AppName', 'mySparkAppDepLocal', ...
    'Master', 'local[1]', ...
    'SparkProperties', sparkProp );

% Create a SparkContext
sc = matlab.compiler.mlspark.SparkContext(conf);

% Create a MATLAB datastore (LOCAL)
ds = datastore('airlinesmall.csv',...
    'TreatAsMissing','NA', ...
    'SelectedVariableNames','UniqueCarrier');

% Convert MATLAB datastore to Spark RDD
rdd = sc.datastoreToRDD(ds);

% Count number of times each airline occurs in airlinesmall.csv
% File Dependency: carrierToCount.m
```

```
maprdd = rdd.flatMap(@carrierToCount);
redrdd = maprdd.reduceByKey( @(acc,value) acc+value );
countdata = redrdd.collect();

%% Count and display carrier occurrences
count = 0;
for i=1:numel(countdata)
    count = count + countdata{i}{2};
    fprintf('\nCarrier Name: %s, Count: %d', countdata{i}{1}, countdata{i}{2});
end
fprintf('\n Total count : %d\n', count);

%% Delete Spark Context
delete(sc)
```

## Hadoop YARN

A yarn-client cluster manager represents a Spark enabled Hadoop cluster. A YARN cluster manager was introduced in Hadoop 2.0. It is typically installed on the same nodes as HDFS. Therefore, running Spark on YARN lets Spark access HDFS data easily. In applications, it is denoted using the word `yarn-client`.

Since the steps for deploying your application using `yarn-client` as your cluster manager are similar to using the local cluster manager shown above, the steps are presented with minimal discussion. For a detailed discussion of each step, check the “Local” on page 7-2 case above.

---

**Note** You can follow the same instructions to deploy Spark applications created using the MATLAB API for Spark to Cloudera CDH. To see an example on MATLAB Answers, [click here](#).

To use Cloudera CDH encryption zones, add the JAR file `commons-codec-1.9.jar` to the static classpath of MATLAB Runtime. Location of the file: `$HADOOP_PREFIX/lib/commons-codec-1.9.jar`, where `$HADOOP_PREFIX` is the location where Hadoop is installed.

---

### Prerequisites

- 1 Start this example by creating a new work folder that is visible to the MATLAB search path.
- 2 Install the MATLAB Runtime in a folder that is accessible by every worker node in the Hadoop cluster. This example uses `/share/MATLAB/MATLAB_Runtime/v91` as the location of the MATLAB Runtime folder.

If you don't have the MATLAB Runtime, you can download it from the website at: <https://www.mathworks.com/products/compiler/mcr>.

- 3 Copy the `airlinesmall.csv` into Hadoop Distributed File System (HDFS) folder `/user/<username>/datasets`. Here `<username>` refers to your username in HDFS.

```
$ ./hadoop fs -copyFromLocal airlinesmall.csv hdfs://host:54310/user/<username>/datasets
```

### Procedure

- 1 Set up the environment variable, `HADOOP_PREFIX` to point at your Hadoop install folder. These properties are necessary for submitting jobs to your Hadoop cluster.

```
setenv('HADOOP_PREFIX', '/share/hadoop/hadoop-2.6.0')
```

The HADOOP\_PREFIX environment variable must be set when using the MATLAB `datastore` function to point to data on HDFS. Setting this environment variable has nothing to do with Spark. See “Relationship Between Spark and Hadoop” on page 4-4 for more information.

**2** Specify Spark properties.

Use a `containers.Map` object to specify Spark properties.

```
sparkProperties = containers.Map( ...
    {'spark.executor.cores', ...
     'spark.executor.memory', ...
     'spark.yarn.executor.memoryOverhead', ...
     'spark.dynamicAllocation.enabled', ...
     'spark.shuffle.service.enabled', ...
     'spark.eventLog.enabled', ...
     'spark.eventLog.dir'}, ...
    {'1', ...
     '2g', ...
     '1024', ...
     'true', ...
     'true', ...
     'true'});
```

For more information on Spark properties, expand the `prop` value of the `'SparkProperties'` name-value pair in the “Input Arguments” on page 1-6 section of the `SparkConf` class.

**3** Create a `SparkConf` object.

Use the class `matlab.compiler.mlspark.SparkConf` to create a `SparkConf` object.

```
conf = matlab.compiler.mlspark.SparkConf( ...
    'AppName', 'myApp', ...
    'Master', 'yarn-client', ...
    'SparkProperties', sparkProperties);
```

For more information on `SparkConf`, see `matlab.compiler.mlspark.SparkConf`.

**4** Create a `SparkContext` object.

Use the class `matlab.compiler.mlspark.SparkContext` with the `SparkConf` object as an input to create a `SparkContext` object.

```
sc = matlab.compiler.mlspark.SparkContext(conf);
```

For more information on `SparkContext`, see `matlab.compiler.mlspark.SparkContext`.

**5** Create an RDD object from the data.

Use the MATLAB function `datastore` to create a `datastore` object pointing to the file `airlinesmall.csv` in HDFS. Then use the `SparkContext` method `datastoreToRDD` to convert the `datastore` object to a Spark RDD object.

```
% Create a MATLAB datastore (HADOOP)
ds = datastore(...
    'hdfs:///user/<username>/datasets/airlinesmall.csv', ...
    'TreatAsMissing', 'NA', ...
    'SelectedVariableNames', 'UniqueCarrier');
```

```
% Convert MATLAB datastore to Spark RDD
rdd = sc.datastoreToRDD(ds);
```

In general, input RDDs can be created using the following methods of the `SparkContext` class: `parallelize` on page 3-9, `datastoreToRDD` on page 3-6, and `textFile` on page 3-14.

**6** Perform operations on the RDD object.

Use a Spark RDD method such as `flatMap` on page 2-24 to apply a function to all elements of the RDD object and flatten the results. The function `carrierToCount` that was created earlier serves as the function that is going to be applied to the elements of the RDD. A function handle to the function `carrierToCount` is passed as an input argument to the `flatMap` method.

```
maprdd = rdd.flatMap(@carrierToCount);
redrdd = maprdd.reduceByKey( @(acc,value) acc+value );
countdata = redrdd.collect();

% Count and display carrier occurrences
count = 0;
for i=1:numel(countdata)
    count = count + countdata{i}{2};
    fprintf('\nCarrier Code: %s, Count: %d', countdata{i}{1}, countdata{i}{2});
end
fprintf('\n Total count : %d\n', count);

% Save results to MAT file
save('countdata.mat','countdata');

% Delete Spark Context
delete(sc);
```

For a list of supported RDD transformations and actions, see “Transformations” on page 1-3 and “Actions” on page 1-4 in the Methods section of the RDD class.

For more information on transformations and actions, see “Apache Spark Basics” on page 4-2.

**7** Create a standalone application.

Use the `mcc` command with the `-m` flag to create a standalone application. The `-m` flag creates a standalone application that can be run from a command line. You do not need to attach the dataset `airlinesmall.csv` since it resides on HDFS. The `mcc` command automatically picks up the dependent file `carrierToCount.m` as long as it is in the same work folder.

```
>> mcc -m deployToSparkMlApiHadoop.m
```

The `mcc` command creates a shell script `run_deployToSparkMlApiHadoop.sh` to run the executable file `deployToSparkMlApiHadoop`.

For more information, see `mcc`.

**8** Run the standalone application from a Linux shell using the following command:

```
$ ./run_deployToSparkMlApiHadoop.sh /share/MATLAB/MATLAB_Runtime/v91
```

`/share/MATLAB/MATLAB_Runtime/v91` is an argument indicating the location of the MATLAB Runtime.

Prior to executing the above command, make sure the `javaclasspath.txt` file is in the same folder as the shell script and the executable.

Your application will fail to execute if it cannot find the file `javaclasspath.txt`.





```
'1024',...
'true',...
'true',...
'true',...
'hdfs://host:54310/user/<username>/sparkdeploy'});

%% Create SparkConf object
conf = matlab.compiler.mlspark.SparkConf( ...
    'AppName','myApp', ...
    'Master','yarn-client', ...
    'SparkProperties',sparkProperties);

%% Create a SparkContext
sc = matlab.compiler.mlspark.SparkContext(conf);

%% Create a MATLAB datastore (HADOOP)
ds = datastore(...
    'hdfs:///user/<username>/datasets/airlinesmall.csv',...
    'TreatAsMissing','NA',...
    'SelectedVariableNames','UniqueCarrier');

%% Convert MATLAB datastore to Spark RDD
rdd = sc.datastoreToRDD(ds);

%% Count number of times each airline occurs in airlinesmall.csv
% File Dependency: carrierToCount.m
maprdd = rdd.flatMap(@carrierToCount);
redrdd = maprdd.reduceByKey( @(acc,value) acc+value );
countdata = redrdd.collect();

%% Count and display carrier occurrences
count = 0;
for i=1:numel(countdata)
    count = count + countdata{i}{2};
    fprintf('\nCarrier Code: %s, Count: %d', countdata{i}{1}, countdata{i}{2});
end
fprintf('\n Total count : %d\n', count);

%% Save results to MAT file
save('countdata.mat','countdata');

%% Delete Spark Context
delete(sc);
```

## Deploying Applications to Cloudera Spark Using the MATLAB API for Spark

This example shows you how to deploy a MATLAB application developed using the MATLAB API for Spark against a Cloudera Spark enabled Hadoop cluster.

The application `flightsByCarrierDemo.m` computes the number of airline carrier types from airline data. The inputs to the application are:

- `master` — URL to the Spark cluster
- `inputFile` — the file containing the input data

**Note** The complete code for this example is in the file `flightsByCarrierDemo.m`, as shown below.

### `flightsByCarrierDemo.m`

```
%
% Copyright 2016 MathWorks, Inc
%
function flightsByCarrierDemo(master, inputFile)
import matlab.compiler.mlspark.*
%%
% Set up datastore for airlinesmall.csv
%
ds = datastore( ...
    inputFile, ...
    'datastoretype','tabulartext', ...
    'TreatAsMissing','NA', ...
    'SelectedVariableNames','UniqueCarrier');%%
%% Set up spark to use yarn resource manager...
% 1 thread per executor task
% 1 executor task
sparkProperties = containers.Map( ...
    {'spark.executor.cores','spark.executor.instances'}, ...
    {'1','1'});
conf = SparkConf( ...
    'AppName','flightsByCarrierDemo', ...
    'Master',master, ...
    'SparkProperties',sparkProperties );
sc = SparkContext(conf);
%%
% Create an rdd that contains the number of flights per airline.
countRDD = sc.datastoreToRDD(ds).flatMap(@map carriertocountusingds) ...
    .reduceByKey( @(acc,value) acc+value) ...
    .map( @(keyvalue) sprintf('%s,%i',keyvalue{1},keyvalue{2}));
countRDD.saveAsTextFile('flightsByCarrierResults');
```

### Prerequisites

- Install the MATLAB Runtime in the default location on the desktop. This example uses `/usr/local/MATLAB/MATLAB_Runtime/v91` as the default location for the MATLAB Runtime.

If you don't have MATLAB Runtime, see "Install and Configure the MATLAB Runtime" for installation instructions.

- Install the MATLAB Runtime on every worker node.
- Copy the `airlinesmall.csv` from folder `toolbox/matlab/demos` of your MATLAB install area into Hadoop Distributed File System (HDFS) folder `/datasets/airlinemod`.

### Deploy Applications to Cloudera Spark

- 1 At the MATLAB command prompt, use the `mcc` command to generate a jar file and a shell script for the MATLAB application `flightsByCarrierDemo.m`.

```
>> mcc -C -W 'Spark:flightsByCarrierDemoApp' flightsByCarrierDemo.m
```

This action creates a jar file named `flightsByCarrierDemoApp.jar` and a shell script named `run_flightsByCarrierDemoApp.sh`.

- 2 Execute the shell script in either `yarn-client` mode or `yarn-cluster` mode. In `yarn-client` mode, the driver runs on the desktop. In `yarn-cluster` mode, the driver runs in the Application Master process in the cluster. The results of the computation in both cases are saved to a text file on HDFS by calling the `saveAsTextFile` method on the RDD.

#### yarn-client mode

Run the following command from a Linux terminal:

```
$ ./run_flightsByCarrierDemoApp.sh \  
  /usr/local/MATLAB/MATLAB_Runtime/v91 \  
  yarn-client \  
  hdfs://hadoop01glnxa64:54310/datasets/airlinemod/airlinesmall.csv
```

To examine the results, enter the following from a Linux terminal:

```
$ hadoop fs -cat flightsByCarrierResults/*
```

#### yarn-cluster mode

Run the following command from a Linux terminal:

```
$ ./run_flightsByCarrierDemoApp.sh \  
  /usr/local/MATLAB/MATLAB_Runtime/v91 \  
  --deploy-mode cluster --master yarn yarn-cluster \  
  hdfs://hadoop01glnxa64:54310/datasets/airlinemod/airlinesmall.csv
```