# Introduction to the Spark API

olsen
software ltd

# Contents

1. Essential concepts
2. Creating an RDD
3. Working with RDDs

# 1. Essential Concepts

- Overview of the Spark library
- Important Spark abstractions
- Overview of SparkContext
- Accessing a SparkContext in the Spark shell
- Overview of Resilient Distributed Datasets (RDD)
- Key features of RDD

# Overview of the Spark Library

- Spark has a rich library of cluster computing capabilities
  - The Spark library is written in Scala
- Spark provides APIs for numerous languages, including:
  - Scala
  - Java
  - Python
  - R
- We'll use the Scala API
  - We'll use the Spark shell initially, because it's an extremely productive way to learn Spark
  - Then we'll see how to write full Scala applications later

# Important Spark Abstractions

- The Spark API comprises two important abstractions
  - SparkContext
  - Resilient Distributed Datasets (RDD)

- Applications use these abstractions to connect to a Spark cluster, and to use the cluster resources
  - See following slides for details

# Overview of SparkContext

- `SparkContext` is the entry-point class in the Spark API
  - A Spark app must create an instance of this class, to represent a connection to a Spark cluster

- `SparkContext` has various constructors
  - Default ctor gets config settings from system properties

```
val sc = new SparkContext()
```

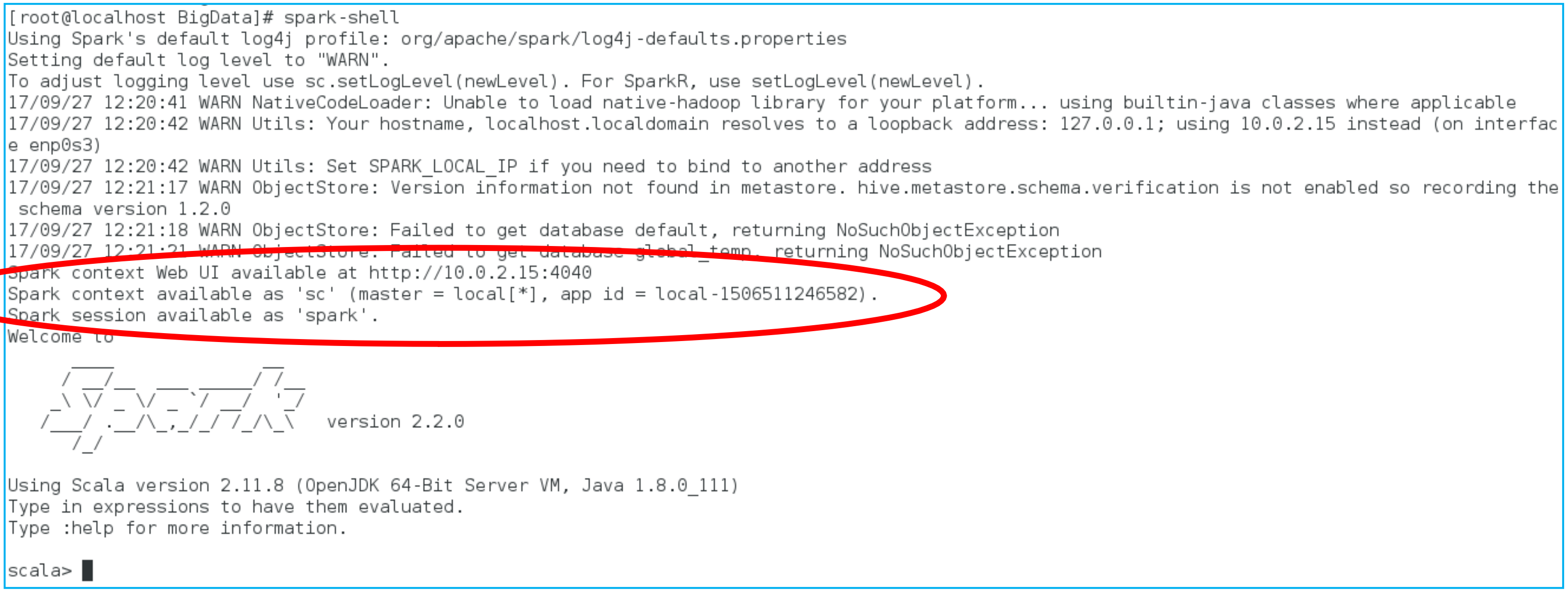
  - [illegible]

```
val config = new SparkConf()
                 .setMaster("spark://somehost:port")
                 .setAppName("my big app")

val sc = new SparkContext(config)
```

# Accessing a SparkContext in the Spark Shell

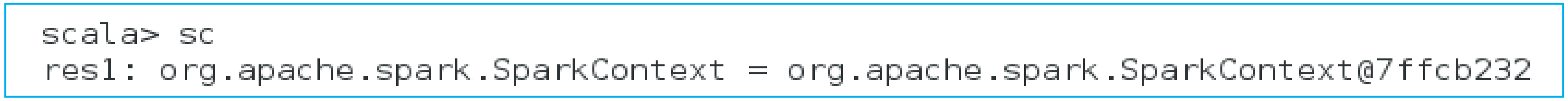
- The Spark shell already provides a `SparkContext` object
  - Available as `sc`



```
[root@localhost BigData]# spark-shell
Using Spark's default log4j profile: org/apache/spark/log4j-defaults.properties
Setting default log level to "WARN".
To adjust logging level use sc.setLogLevel(newLevel). For SparkR, use setLogLevel(newLevel).
17/09/27 12:20:41 WARN NativeCodeLoader: Unable to load native-hadoop library for your platform... using builtin-java classes where applicable
17/09/27 12:20:42 WARN Utils: Your hostname, localhost.localdomain resolves to a loopback address: 127.0.0.1; using 10.0.2.15 instead (on interfac
e enp0s3)
17/09/27 12:20:42 WARN Utils: Set SPARK_LOCAL_IP if you need to bind to another address
17/09/27 12:21:17 WARN ObjectStore: Version information not found in metastore. hive.metastore.schema.verification is not enabled so recording the
 schema version 1.2.0
17/09/27 12:21:18 WARN ObjectStore: Failed to get database default, returning NoSuchObjectException
17/09/27 12:21:21 WARN ObjectStore: Failed to get database global_temp, returning NoSuchObjectException
Spark context Web UI available at http://10.0.2.15:4040
Spark context available as 'sc' (master = local[*], app id = local-1506511246582).
Spark session available as 'spark'.
Welcome to
      ____              __
     / __/__  ___ _____/ /__
    _\ \/ _ \/ _ `/ __/  '_/
   /___/ .__/\_,_/_/ /_/\_\   version 2.2.0
      /_/

Using Scala version 2.11.8 (OpenJDK 64-Bit Server VM, Java 1.8.0_111)
Type in expressions to have them evaluated.
Type :help for more information.

scala> 
```



```
scala> sc
res1: org.apache.spark.SparkContext = org.apache.spark.SparkContext@7ffcb232
```

  - Here's the proof ☺

# Overview of Resilient Distributed Datasets (RDD)

- RDD is the primary data abstraction mechanism in Spark
  - It's an abstract class in the Spark API
  - Represents a collection of partitioned data elements that can be operated on in parallel

- Conceptually an RDD is similar to a Python list except…
  - RDD represents a distributed dataset
  - RDD supports lazy operations (see later)

- The following slide describes the key features of an RDD

# Key features of RDD (1 of 2)

- Immutable
  - An RDD is an immutable data structure
  - Once created, it can't be modified
  - Operations that seem to modify an RDD actually return a new RDD

- Partitioned
  - Data represented by an RDD is split into partitions
  - These partitions are generally distributed across a cluster of nodes

# Key features of RDD (2 of 2)

- Fault tolerant
  - RDD is designed to be fault tolerant, to cope with the fact nodes in a cluster are liable to failure
  - RDD automatically handles node failures - when a node fails, Spark reconstructs the lost RDD partitions on another node
- Uniform API
  - RDD is an abstract class
  - Provides a uniform API for various data sources
  - E.g. HadoopRDD, ParallelCollectionRDD, JdbcRDD, CassandraRDD
- Fast
  - Spark allows RDDs to be cached or persisted in memory
  - Magnitudes of faster than operating on non-cached RDDs

# 2. Creating an RDD

- Overview
- Creating an RDD from a text file
- Creating an RDD from all text files
- Creating an RDD from a sequence file

# Overview

- RDD is an abstract class
    - You can't instantiate directly
    - Instead you use factory methods in the `SparkContext` class

- In this section we'll show various `SparkContext` methods for creating RDDs

- You can also create an RDD by transforming an existing RDD - see later

# Creating an RDD from a Text File

- You can create an RDD from an existing text file
  - Call `textFile()` and specify a file or directory
  - The directory could on a local file system, HDFS, or any other Hadoop-supported storage system
  - Returns RDD of strings, each element represents 1 line in the file

- Examples
  - Create an RDD from a file or directory on HDFS

```
val rdd = sc.textFile("hdfs://namenode:9000/some-file-or-directory")
```

  - Read all compressed files in a directory

```
val rdd = sc.textFile("hdfs://namenode:9000/some-directory/*.gz")
```

  - You can pass a 2nd arg, specifying the number of partitions (default is 1, you can specify a higher number to increase parallelization)

# Creating an RDD from All Text Files

- You can create an RDD from all text files in a directory
  - Call `wholeTextFile()` and specify a directory
  - The directory could be on any file system, as discussed previously
  - Returns key-value pairs (keys are file paths, values are file contents)

- Example
  - Create an RDD from all .txt files in a directory

```
val rdd = sc.wholeTextFiles("hdfs://namenode:9000/some-directory/*.txt")
```

# Creating an RDD from a Sequence File

- You can create an RDD from a sequence file
  - Call `sequenceFile()` and specify a sequence file that contains key-value pairs
  - You must also specify the data types of the keys and values
  - The file could be on any file system, as discussed previously
  - Returns key-value pairs from the sequence file

- Example
  - Create an RDD from a sequence file, where the keys and values are strings

```
val rdd = sc.sequenceFile[String,String]("some-file")
```

# 3. Working with RDDs

- Overview
- Types of RDD operations
- Example scenario
- Example code
- Viewing the output

# Overview

- Spark applications process data by using methods defined in the RDD class and subclasses
  - These methods are known as "RDD operations"

- You can use these operations on a wide range of data
  - From a few bytes to several petabytes in size
  - On the local file system or on a distributed storage system

# Types of RDD Operations

- RDD operations are categorized into two types…

- Transformations
  - Create new RDD by performing a computation on source RDD
  - E.g. `union()`, `map()`, `filter()`
  - Can operate on data distributed across a cluster of nodes
  - Note: RDD creation and transformation operations are lazy - we'll discuss this later

- Actions
  - Cause a job to be executed
  - Return a value to the driver program
  - E.g. `collect()`, `min()`, `max()`, `saveAsTextFile()`

# Example Scenario

- To illustrate the end-to-end process of working with RDDs, we'll consider the following simple scenario
  - Create an RDD from a text file
  - Determine the lengths of all lines
  - Save the lengths to a file
  - View the output

- For this example we will use the sample text file:
  - `Macbeth.txt`

# Example Code

- Create an RDD from a text file

```
val inputFile = sc.textFile("Macbeth.txt")
```

- Determine the lengths of all lines
  - Via the `map()` transformation operation

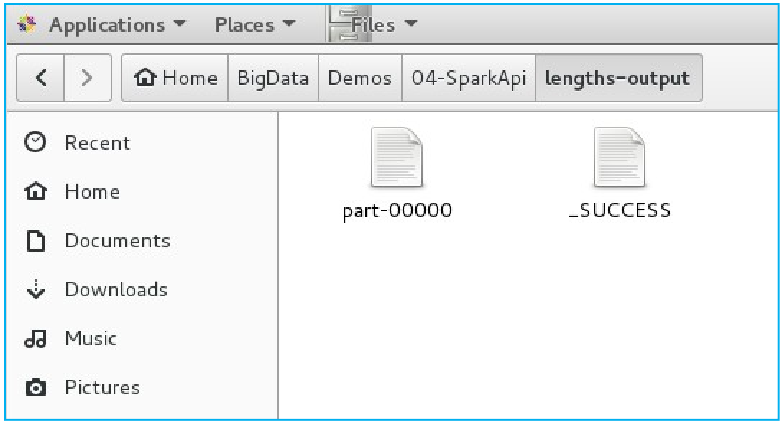
```
val lengths = inputFile map { line => line.length }
```

- Save the lengths to a file
  - Via the `saveAsTextFile()` action operation
  - Specify the name of the output folder

```
lengths.saveAsTextFile("lengths-output")
```

# Viewing the Output

- The code on the previous slide created a folder as follows:



- Here's a listing of `part-00000` (first few lines)

```
5
24
0
43
12
30
34
0
13
27
31
0
12
32
```

# Any Questions?