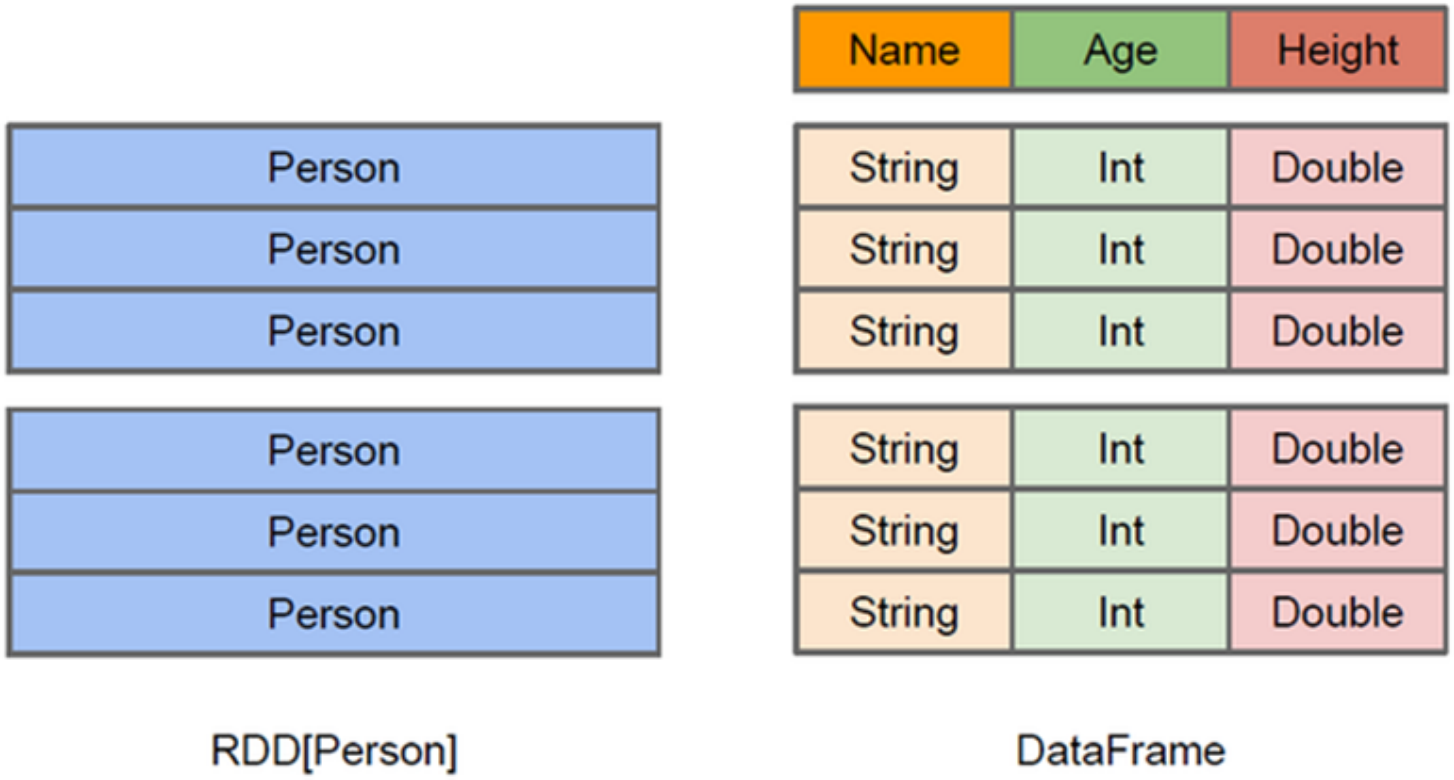
Spark SQL and DataFrame

DataFrames and SQL

The latest versions of Spark include a new kind of object called a **DataFrame**, which is an RDD with some extra structure some extra functionality.

A DataFrame is still a collection of items but those items are special **Row objects**, which have more structure. This makes working a DataFrame a lot like working with a table.

DataFrame is more like a traditional database of two-dimensional form, in addition to data, but also to grasp the structural information of the data, that is, the schema. Let's use an example to see the difference between DataFrame and RDD:



- RDD[Person] although with Person for type parameters, but the Spark framework itself does not understand internal structure of the Person class.
- DataFrame has provided a detailed structural information, making Spark SQL can clearly know what columns are included in the dataset, and what is the name and type of each column. Thus, Spark SQL query optimizer can target optimization.

In fact, Spark allows us to use SQL to query the data in a DataFrame. We'll see that is done in the next slide.

DataFrames in a Python shell

Let's see how to work with DataFrames in Spark. We'll start by working in a Python shell. And we'll work with a CSV file.

If you click on the panel on the right you will get a terminal connection to a server with Hadoop and YARN installed and running.

In your home directory on the server there is a CSV file called "employees.csv". To work with this file using Spark we need to put it into HDFS:

```
$ hdfs dfs -mkdir -p /user/user
$ hdfs dfs -put employees.csv
```

Now let's start a Python shell, with Spark loaded into Python for us:

```
$ pyspark
```

Creating a DataFrame

In addition to the `sc` object which we have used to create and work with RDDs, your Python shell also has an object `spark` which we will use to create and work with DataFrames.

First, we can create a DataFrame by loading the data in employees.csv, as follows:

```
>>> employees = spark.read.format("csv").option("header", "true").load("employees.csv")
```

Here we are using the `read` property of the `spark` object. We specify that the format of the file we want to read is csv, and that it has a header row, and that it is called "employees.csv". Spark assumes that it is our default working directory in HDFS, as it is.

You now have an DataFrame called "employees" which contains the contents of the file "employees.csv". To see the number of items in the DataFrame you can use the `count()` action on the DataFrame:

```
>>> employees.count()
```

To see the first item of the DataFrame you can use the `first()` action:

```
>>> employees.first()
```

So far this is working just like an RDD. But now let's doing something new - use SQL statements to query the data.

Using SQL

We first need to create a temporary table. Let's call it "employees". We do so using the following command:

```
>>> employees.createOrReplaceTempView("employees")
```

Now we can run some queries to answer some questions. To do so we use the `sql()` method of the `spark` object.

What is the maximum salary of employees?

```
>>> spark.sql("SELECT MAX(int(salary)) FROM employees").show()
```

Note that we have used the show() method at the end. This tells Spark to show us the results.

What are the distinct salaries, and which employees have those salaries?

```
>>> spark.sql("SELECT salary, COLLECT_LIST(last_name) FROM employees GROUP BY salary").show()
```

Which employee have the highest salary?

```
>>> spark.sql("SELECT first_name, last_name, salary FROM employees WHERE salary = (SELECT MAX(int(s
```

Which employees were hired in 2007 and have a salary of more than $6,000?

```
>>> spark.sql("SELECT first_name, last_name, hire_date, salary FROM employees WHERE hire_date LIKE
```

DataFrames in a Python program

Now let's see how to use DataFrames in a Python program.

If you click on the panel on the right you will get a terminal connection to a server with Hadoop and YARN installed and running. In your home directory on the server there is the same CSV file called "employees.csv". And there is also a Python program called "program.py".

To work with this file using Spark we need to put it into HDFS:

```
$ hdfs dfs -mkdir -p /user/user
$ hdfs dfs -put employees.csv
```

Now we can submit the Python program to Spark:

```
$ spark-submit program.py
```

You should see Spark begin to execute and produce the output.

Saving output to a file

At the moment the Python program asks Spark to show the results in the terminal window. You can also get Spark to save the results to a file. You can do that by replacing `results.show()` with the following line:

```
results.write.format("csv").save('output')
```

This tells Spark to save the results in CSV format, in a folder called "output" in your default directory in HDFS (it will create this for you).

Now run the program again, as per above. This time the results will be saved to HDFS.

After Spark completes running the program, you can see the results by listing the contents of the "output" folder:

```
$ hdfs dfs -ls output
```

You should see two files: "_SUCCESS" and one that starts with "part-". The first file shows the status of the program, the second file contains the result. You can see the results by showing the contents of the second file, using the following command:

```
$ hdfs dfs -cat output/part-*
```

You might find it convenient to move this file to your home directory on the server's local file system. You might like to rename it in the process - let's call it "result". Here's the command:

```
$ hdfs dfs -get output/part-* result.csv
```

Now you can view the contents of the file using the much simpler command:

```
$ cat result.csv
```

Create DataFrames from an existing RDD

Now let's see how to create DataFrames from an existing RDD.

If you click on the panel on the right you will get a terminal connection to a server with Hadoop and YARN installed and running. In your home directory on the server, there is a text file called "employees.txt". And there is also a Python program called "program.py".

The task is to find **the employees who were hired in 2007 and have a salary of more than $6,000.**

We first load the text file and convert each line to a **Row** object.

```
sc = SparkContext('local', 'employeesTxt')
text = sc.textFile("employees.txt")

parts = text.map(lambda l: l.split(","))

employees_txt = parts.map(lambda p: Row(employee_id=p[0],first_name=p[1],last_name=p[2],hire_date=p
```

Next, we use `createDataFrame` and `createOrReplaceTempView` to infer the schema, and register the DataFrame as a table.

```
spark = SparkSession.builder.master("local").appName("employees").getOrCreate()

schemaEmployees = spark.createDataFrame(employees_txt)
schemaEmployees.createOrReplaceTempView("employees")
```

At last, SQL can be run over DataFrames that have been registered as a table.

```
results = spark.sql("SELECT first_name, last_name, hire_date, salary FROM employees WHERE hire_date
```

To work with the text file using Spark we need to put it into HDFS:

```
$ hdfs dfs -mkdir -p /user/user
$ hdfs dfs -put employees.txt
```

Now we can submit the Python program to Spark:

```
$ spark-submit program.py
```

You should see Spark begin to execute and produce the output.

Try it: salary holders

Now try writing your own Python script using DataFrames and SQL.

When you click the panel on the right you'll get a connection to a server with Hadoop and YARN installed and running. In your home directory on the server there is the same CSV file called "employees.csv". A Python script called "script.py" has been created for you.

Try writing using DataFrames and SQL to **produce a list of salaries, and for each salary the last names of the people with that salary**. You can use COLLECT_LIST.

Remember that you will need to copy employees.csv into HDFS:

```
$ hdfs dfs -mkdir -p /user/user
$ hdfs dfs -put employees.csv
```

You can run your Python script using the following command:

```
$ spark-submit script.py
```