UD 6 - Apache Spark - Spark API

Ya hemos visto que uno de los mayores atractivos de Apache Spark para los desarrolladores ha sido sus APIs fáciles de usar para operar en grandes conjuntos de datos, en diferentes lenguajes de programación tales como: Scala, Java, Python y R.

1. Spark APIs

Vamos a explicar los tres conjuntos de APIs disponibles en **Apache Spark** (**RDDs**, **DataFrames**, and **Datasets**) además de cuándo y por qué debemos usar cada conjunto de API; descubriremos rendimiento y beneficios de optimización y enumeraremos escenarios sobre cuándo usar DataFrames y Datasets en lugar de RDD. Principalmente, nos centraremos en DataFrames y Datasets, porque desde Apache Spark 2.0, estas dos API están unificadas.

1.1 RDD (Resilient Distributed Dataset)

RDD fue la principal API orientada al usuario en Spark desde su inicio. En el núcleo, un RDD es una colección distribuida **inmutable** de elementos de sus datos, **particionados a través de nodos en su clúster** que se pueden operar en **paralelo** con una API de bajo nivel que ofrece *transformaciones* y acciones.

¿Cuándo usar RDD?

- Necesitamos una transformación de **bajo nivel** y acciones y control en su conjunto de datos.
- Para datos **no estructurados**, como flujos de medios o flujos de texto.
- desea manipular sus datos con construcciones de programación funcional que expresiones específicas de dominio.
- No necesitamos imponer un esquema, como formato de columna, mientras procesamos o accedemos a los atributos de los datos por nombre o columna.
- Podemos renunciar a algunos beneficios de optimización y rendimiento disponibles con DataFrames y Datasets para datos estructurados y semiestructurados.

1.2 Dataset

Un **Dataset** es una **colección distribuida de datos**. Dataset es una nueva interfaz agregada en Spark 1.6 que proporciona los beneficios de los RDD (fuertemente tipado, capacidad de utilizar potentes funciones lambda) con los beneficios del motor de ejecución optimizado de Spark

SQL. Se puede construir un Dataset a partir de objetos JVM y luego manipularlo mediante transformaciones funcionales (map, flatMap, filter, etc.).

La API del Dataset está disponible en Scala y Java. Python no es compatible con la API del Dataset. Pero debido a la naturaleza dinámica de Python, muchos de los beneficios de la API del Dataset ya están disponibles (es decir, puede acceder al campo de una fila por su nombre row.columnName). El caso de R es similar.

1.3 Dataframe

Un **DataFrame** es un **Dataset organizado en columnas con nombre**. Es conceptualmente equivalente a una tabla en una base de datos relacional o un marco de datos en R/Python, pero con optimizaciones más ricas bajo el capó. Los DataFrames se pueden construir a partir de una amplia gama de fuentes, como: archivos de datos estructurados, tablas en Hive, bases de datos externas o RDDs existentes.

La API DataFrame está disponible en Scala, Java, Python y R. En Scala y Java, un DataFrame está representado por un Dataset de filas. En la API de Scala, DataFrame es simplemente un alias de tipo de Dataset[Row]. Mientras que, en la API de Java, los usuarios deben usar Dataset para representar un DataFrame.

1.4 Spark SQL

Spark SQL es un módulo Spark para el procesamiento de datos estructurados. A diferencia de la API básica de Spark RDD, las interfaces proporcionadas por Spark SQL brindan a Spark más información sobre la estructura de los datos y el cálculo que se realiza. Internamente, Spark SQL utiliza esta información adicional para realizar optimizaciones adicionales. Hay varias formas de interactuar con Spark SQL, incluidas SQL y la API del conjunto de datos. Al calcular un resultado, se utiliza el mismo motor de ejecución, independientemente de qué API/lenguaje esté utilizando para expresar el cálculo. Esta unificación significa que los desarrolladores pueden alternar fácilmente entre diferentes API según cuál proporcione la forma más natural de expresar una transformación determinada.

1.5 Beneficios Dataframe/Dataset APIs

1. **Tipado estático y tipado seguro en tiempo de ejecución**: Dadas las diferencias de tipado entre las 3 APIs de Spark nos encontramos con las siguientes diferencias

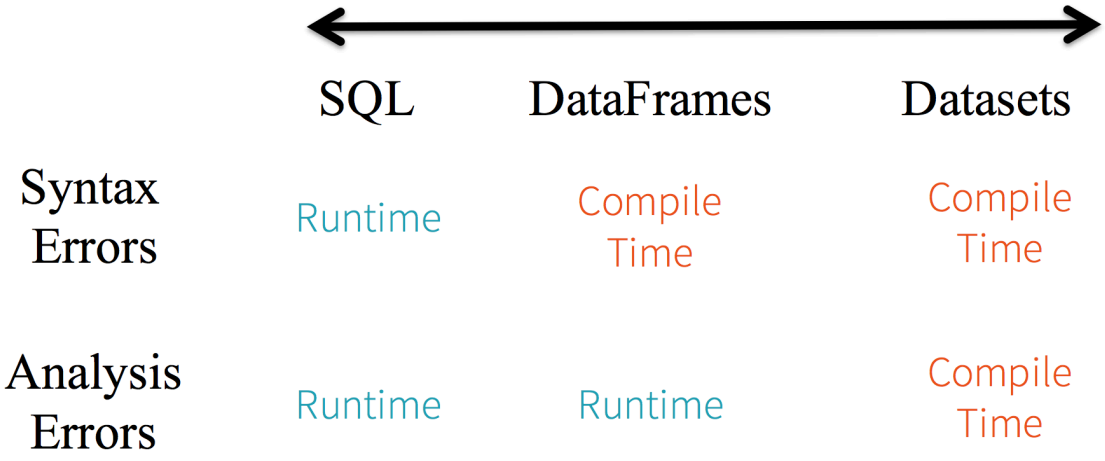

Figura 6.1_Spark API: Mejoras en tiempo de ejecución. (Fuente: databricks.com)

2. **Alto nivel de Abstracción y vista personalizada de datos estructurados y semiestructurados**.
3. **Facilidad de uso de API con estructura**: Introduce una semántica rica y un conjunto sencillo de operaciones específicas de dominio que pueden expresarse como construcciones de alto nivel. Es mucho más sencillo realizar operaciones `agg`, `select`, `sum`, `avg`, `map`, `filter`, `groupBy`, `filter`, `map`
4. **Rendimiento y optimización**: Mejoras de eficiencia en el uso de almacenamiento y ganancias de rendimiento.

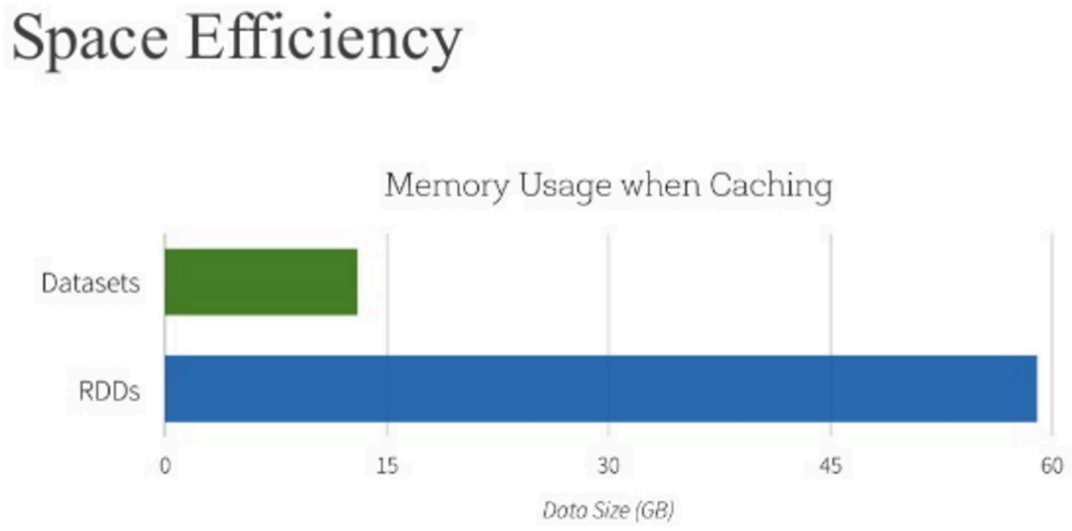

Figura 6.2_Spark API: Mejoras de eficiencia y rendimiento. (Fuente: databricks.com)

2. RDDs. Getting Started

Hay dos formas de crear RDD: paralelizar una colección existente en su driver program o hacer referencia a un conjunto de datos en un sistema de almacenamiento externo, como un sistema de archivos compartido, HDFS, HBase o cualquier fuente de datos que ofrezca un formato de entrada Hadoop.

2.1. Parallelized Collections

Parallelized Collections se crean llamando al método de parallelize de `SparkContext`. Los elementos de la colección se copian para formar un conjunto de datos distribuido que se puede operar en paralelo en nuestro cluster. Por ejemplo, Creamos una colección paralela que contenga los números del 1 al 8:

Python

```python
data = [1, 2, 3, 4, 5, 6 ,7, 8]
distData = sc.parallelize(data)
```

Scala

```scala
val data = Array(1, 2, 3, 4, 5)
val distData = sc.parallelize(data)
```

Java

```java
//JavaSparkContext en lugar de SparkContext
List<Integer> data = Arrays.asList(1, 2, 3, 4, 5);
JavaRDD<Integer> distData = sc.parallelize(data);
```

Una vez creado, el conjunto de datos distribuido (distData) se puede operar en paralelo. Por ejemplo, podemos llamar a `distData.reduce(lambda a, b: a + b)` para sumar los elementos de la lista.

2.2 Conjunto de datos externo

Spark puede crear conjuntos de datos distribuidos desde cualquier fuente de almacenamiento compatible con Hadoop, incluido su sistema de archivos local, HDFS, Cassandra, HBase, Amazon S3, etc. Spark admite archivos de texto, SequenceFiles y cualquier otro formato de entrada de Hadoop.

Los RDD de archivos de texto se pueden crear utilizando el método `textFile` de `SparkContext`. Este método toma un URI para el archivo (ya sea una ruta local en la máquina

o un URI hdfs://, s3a://, etc.) y lo lee como una colección de líneas. Aquí hay un ejemplo de invocación:

Python

```
>>> distFile = sc.textFile("data.txt")
```

Scala

```
scala> val distFile = sc.textFile("data.txt")
distFile: org.apache.spark.rdd.RDD[String] = data.txt MapPartitionsRDD[10] at textFile at <console>:26
```

Java

```
JavaRDD<String> distFile = sc.textFile("data.txt");
```

3. Spark SQL, DataFrames and Datasets. Getting started

Puedes consultar toda la información detallada en la documentación oficial

3.1 Starting Point: SparkSession

Python

El punto de entrada a todas las funciones de Spark es la clase SparkSession. Para crear una `SparkSession` básica, simplemente use `SparkSession.builder`:

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()
```

Encuentra los códigos completos en `examples/src/main/python/sql/basic.py` del repositorio Spark

Scala

El punto de entrada a todas las funciones de Spark es la clase SparkSession. Para crear una `SparkSession` básica, simplemente use `SparkSession.builder()`:

```
import org.apache.spark.sql.SparkSession
```

```
2
3  val spark = SparkSession
4    .builder()
5    .appName("Spark SQL basic example")
6    .config("spark.some.config.option", "some-value")
7    .getOrCreate()
```

Encuentra los códigos completos en `examples/src/main/scala/org/apache/spark/examples/sql/SparkSQLExample.scala` del repositorio Spark

Java

El punto de entrada a todas las funciones de Spark es la clase SparkSession. Para crear una `SparkSession` básica, simplemente use SparkSession.builder:

```
1  import org.apache.spark.sql.SparkSession;
2
3  SparkSession spark = SparkSession
4    .builder()
5    .appName("Java Spark SQL basic example")
6    .config("spark.some.config.option", "some-value")
7    .getOrCreate();
```

Encuentra los códigos completos en `examples/src/main/java/org/apache/spark/examples/sql/JavaSparkSQLExample.java` del repositorio Spark

R

El punto de entrada a todas las funciones de Spark es la clase SparkSession `. Para crear una` SparkSession `básica, simplemente use` sparkR.session():

```
1  sparkR.session(appName = "R Spark SQL basic example", sparkConfig =
list(spark.some.config.option = "some-value"))
```

Encuentra los códigos completos en `examples/src/main/r/RSparkSQLExample.R` del repositorio Spark

3.2 Creando Dataframes

Con `SparkSession`, las aplicaciones pueden crear **DataFrames** a partir de un RDD existente, de una tabla de Hive o de fuentes de datos de Spark.

Como ejemplo, lo siguiente crea un DataFrame basado en el contenido de un archivo JSON

Python

```python
# spark is an existing SparkSession
df = spark.read.json("examples/src/main/resources/people.json")
# Displays the content of the DataFrame to stdout
df.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

Scala

```scala
val df = spark.read.json("examples/src/main/resources/people.json")

// Displays the content of the DataFrame to stdout
df.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Java

```java
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

Dataset<Row> df =
spark.read().json("examples/src/main/resources/people.json");

// Displays the content of the DataFrame to stdout
df.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

R

```r
df <- read.json("examples/src/main/resources/people.json")

# Displays the content of the DataFrame
head(df)
##   age    name
## 1  NA Michael
```

```
## 2  30    Andy
## 3  19  Justin

# Another method to print the first few rows and optionally truncate the printing of long values
showDF(df)
## +----+-------+
## | age|   name|
## +----+-------+
## |null|Michael|
## |  30|   Andy|
## |  19| Justin|
## +----+-------+
```

3.3 DataFrame Operations

Los DataFrames proporcionan un lenguaje específico de dominio para la manipulación de datos estructurados en Scala, Java, Python y R.

Estas operaciones también se denominan "transformaciones no tipadas" en contraste con las "transformaciones fuertemente tipadas" que vienen con conjuntos de datos Scala/Java.

Aquí incluimos algunos ejemplos básicos de procesamiento de datos estructurados utilizando Datasets:

Python

```python
# spark, df are from the previous example
# Print the schema in a tree format
df.printSchema()
# root
# |-- age: long (nullable = true)
# |-- name: string (nullable = true)

# Select only the "name" column
df.select("name").show()
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+

# Select everybody, but increment the age by 1
df.select(df['name'], df['age'] + 1).show()
# +-------+---------+
# |   name|(age + 1)|
# +-------+---------+
# |Michael|     null|
# |   Andy|       31|
# | Justin|       20|
# +-------+---------+
```

```
27
28  # Select people older than 21
29  df.filter(df['age'] > 21).show()
30  # +---+----+
31  # |age|name|
32  # +---+----+
33  # | 30|Andy|
34  # +---+----+
35
36  # Count people by age
37  df.groupBy("age").count().show()
38  # +----+-----+
39  # | age|count|
40  # +----+-----+
41  # |  19|    1|
42  # |null|    1|
43  # |  30|    1|
44  # +----+-----+
```

Scala

```
1   // This import is needed to use the $-notation
2   import spark.implicits._
3   // Print the schema in a tree format
4   df.printSchema()
5   // root
6   // |-- age: long (nullable = true)
7   // |-- name: string (nullable = true)
8
9   // Select only the "name" column
10  df.select("name").show()
11  // +-------+
12  // |   name|
13  // +-------+
14  // |Michael|
15  // |   Andy|
16  // | Justin|
17  // +-------+
18
19  // Select everybody, but increment the age by 1
20  df.select($"name", $"age" + 1).show()
21  // +-------+---------+
22  // |   name|(age + 1)|
23  // +-------+---------+
24  // |Michael|     null|
25  // |   Andy|       31|
26  // | Justin|       20|
27  // +-------+---------+
28
29  // Select people older than 21
30  df.filter($"age" > 21).show()
31  // +---+----+
32  // |age|name|
33  // +---+----+
34  // | 30|Andy|
35  // +---+----+
36
```

```
// Count people by age
df.groupBy("age").count().show()
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
// +----+-----+
```

Java

```
// col("...") is preferable to df.col("...")
import static org.apache.spark.sql.functions.col;

// Print the schema in a tree format
df.printSchema();
// root
// |-- age: long (nullable = true)
// |-- name: string (nullable = true)

// Select only the "name" column
df.select("name").show();
// +-------+
// |   name|
// +-------+
// |Michael|
// |   Andy|
// | Justin|
// +-------+

// Select everybody, but increment the age by 1
df.select(col("name"), col("age").plus(1)).show();
// +-------+---------+
// |   name|(age + 1)|
// +-------+---------+
// |Michael|     null|
// |   Andy|       31|
// | Justin|       20|
// +-------+---------+

// Select people older than 21
df.filter(col("age").gt(21)).show();
// +---+----+
// |age|name|
// +---+----+
// | 30|Andy|
// +---+----+

// Count people by age
df.groupBy("age").count().show();
// +----+-----+
// | age|count|
// +----+-----+
// |  19|    1|
// |null|    1|
// |  30|    1|
```

```
// +----+-----+
```

R

```r
# Create the DataFrame
df <- read.json("examples/src/main/resources/people.json")

# Show the content of the DataFrame
head(df)
##   age    name
## 1  NA Michael
## 2  30    Andy
## 3  19  Justin


# Print the schema in a tree format
printSchema(df)
## root
## |-- age: long (nullable = true)
## |-- name: string (nullable = true)

# Select only the "name" column
head(select(df, "name"))
##      name
## 1 Michael
## 2    Andy
## 3  Justin

# Select everybody, but increment the age by 1
head(select(df, df$name, df$age + 1))
##      name (age + 1.0)
## 1 Michael          NA
## 2    Andy          31
## 3  Justin          20

# Select people older than 21
head(where(df, df$age > 21))
##   age name
## 1  30 Andy

# Count people by age
head(count(groupBy(df, "age")))
##   age count
## 1  19     1
## 2  NA     1
## 3  30     1
```

3.4 Ejecutando sentencias SQL

La función SQL en `SparkSession` permite que las aplicaciones ejecuten consultas SQL mediante programación y devuelve el resultado como un DataFrame.

Python

```python
# Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

Scala

```scala
// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people")

val sqlDF = spark.sql("SELECT * FROM people")
sqlDF.show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Java

```java
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

// Register the DataFrame as a SQL temporary view
df.createOrReplaceTempView("people");

Dataset<Row> sqlDF = spark.sql("SELECT * FROM people");
sqlDF.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

R

```r
df <- sql("SELECT * FROM table")
```

3.5 Vista temporal global

Las vistas temporales en Spark SQL tienen un ámbito de sesión y desaparecerán si finaliza la sesión que las crea. Si deseamos tener una vista temporal que se comparta entre todas las sesiones y mantenerla activa hasta que finalice la aplicación Spark, podemos crear una **vista temporal global**. La vista temporal global está vinculada a una base de datos `global_temp` preservada por el sistema, y debemos usar el nombre calificado para referirla, por ejemplo: `SELECT * FROM global_temp.view1`.

Python

```python
# Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

# Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+

# Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
# +----+-------+
# | age|   name|
# +----+-------+
# |null|Michael|
# |  30|   Andy|
# |  19| Justin|
# +----+-------+
```

Scala

```scala
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people")

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show()
// +----+-------+
// | age|   name|
// +----+-------+
```

```
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

Java

```
// Register the DataFrame as a global temporary view
df.createGlobalTempView("people");

// Global temporary view is tied to a system preserved database `global_temp`
spark.sql("SELECT * FROM global_temp.people").show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+

// Global temporary view is cross-session
spark.newSession().sql("SELECT * FROM global_temp.people").show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

3.6 Creando Dataset

Los Dataset son similares a los RDD, sin embargo, en lugar de utilizar la serialización Java o Kryo, utilizan un codificador `Encoder` para serializar los objetos para procesarlos o transmitirlos a través de la red. Si bien tanto los codificadores como la serialización estándar son responsables de convertir un objeto en bytes, los codificadores son código generado dinámicamente y utilizan un formato que permite a Spark realizar muchas operaciones como filtrado, clasificación y hash sin deserializar los bytes nuevamente en un objeto.

Scala

```
case class Person(name: String, age: Long)

// Encoders are created for case classes
val caseClassDS = Seq(Person("Andy", 32)).toDS()
caseClassDS.show()
// +----+---+
// |name|age|
// +----+---+
// |Andy| 32|
// +----+---+
```

```
11
12  // Encoders for most common types are automatically provided by importing
spark.implicits._
13  val primitiveDS = Seq(1, 2, 3).toDS()
14  primitiveDS.map(_ + 1).collect() // Returns: Array(2, 3, 4)
15
16  // DataFrames can be converted to a Dataset by providing a class. Mapping
will be done by name
17  val path = "examples/src/main/resources/people.json"
18  val peopleDS = spark.read.json(path).as[Person]
19  peopleDS.show()
20  // +----+-------+
21  // | age|   name|
22  // +----+-------+
23  // |null|Michael|
24  // |  30|   Andy|
25  // |  19| Justin|
26  // +----+-------+
```

Java

```
1   import java.util.Arrays;
2   import java.util.Collections;
3   import java.io.Serializable;
4
5   import org.apache.spark.api.java.function.MapFunction;
6   import org.apache.spark.sql.Dataset;
7   import org.apache.spark.sql.Row;
8   import org.apache.spark.sql.Encoder;
9   import org.apache.spark.sql.Encoders;
10
11  public static class Person implements Serializable {
12  private String name;
13  private long age;
14
15  public String getName() {
16      return name;
17  }
18
19  public void setName(String name) {
20      this.name = name;
21  }
22
23  public long getAge() {
24      return age;
25  }
26
27  public void setAge(long age) {
28      this.age = age;
29  }
30  }
31
32  // Create an instance of a Bean class
33  Person person = new Person();
34  person.setName("Andy");
35  person.setAge(32);
36
```

```java
// Encoders are created for Java beans
Encoder<Person> personEncoder = Encoders.bean(Person.class);
Dataset<Person> javaBeanDS = spark.createDataset(
Collections.singletonList(person),
personEncoder
);
javaBeanDS.show();
// +---+----+
// |age|name|
// +---+----+
// | 32|Andy|
// +---+----+

// Encoders for most common types are provided in class Encoders
Encoder<Long> longEncoder = Encoders.LONG();
Dataset<Long> primitiveDS = spark.createDataset(Arrays.asList(1L, 2L,
3L), longEncoder);
Dataset<Long> transformedDS = primitiveDS.map(
    (MapFunction<Long, Long>) value -> value + 1L,
    longEncoder);
transformedDS.collect(); // Returns [2, 3, 4]

// DataFrames can be converted to a Dataset by providing a class. Mapping
based on name
String path = "examples/src/main/resources/people.json";
Dataset<Person> peopleDS = spark.read().json(path).as(personEncoder);
peopleDS.show();
// +----+-------+
// | age|   name|
// +----+-------+
// |null|Michael|
// |  30|   Andy|
// |  19| Justin|
// +----+-------+
```

3.7 Interoperar con RDD

Spark SQL admite dos métodos diferentes para convertir RDD existentes en Dataset. El primer método utiliza la **reflexión** para inferir el esquema de un RDD que contiene tipos específicos de objetos. Este enfoque basado en la reflexión genera un código más conciso y funciona bien cuando ya conoce el esquema mientras escribe su aplicación Spark.

El segundo método para crear Dataset es a través de una **interfaz programática** que le permite construir un esquema y luego aplicarlo a un RDD existente. Si bien este método es más detallado, le permite construir Dataset cuando las columnas y sus tipos no se conocen hasta el tiempo de ejecución.

1. Usando Reflexión

Python

```python
from pyspark.sql import Row
```

```python

sc = spark.sparkContext

# Load a text file and convert each line to a Row.
lines = sc.textFile("examples/src/main/resources/people.txt")
parts = lines.map(lambda l: l.split(","))
people = parts.map(lambda p: Row(name=p[0], age=int(p[1])))

# Infer the schema, and register the DataFrame as a table.
schemaPeople = spark.createDataFrame(people)
schemaPeople.createOrReplaceTempView("people")

# SQL can be run over DataFrames that have been registered as a table.
teenagers = spark.sql("SELECT name FROM people WHERE age >= 13 AND age <=
19")

# The results of SQL queries are Dataframe objects.
# rdd returns the content as an :class:`pyspark.RDD` of :class:`Row`.
teenNames = teenagers.rdd.map(lambda p: "Name: " + p.name).collect()
for name in teenNames:
    print(name)
# Name: Justin
```

Scala

```scala
// For implicit conversions from RDDs to DataFrames
import spark.implicits._

// Create an RDD of Person objects from a text file, convert it to a
Dataframe
val peopleDF = spark.sparkContext
.textFile("examples/src/main/resources/people.txt")
.map(_.split(","))
.map(attributes => Person(attributes(0), attributes(1).trim.toInt))
.toDF()
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people")

// SQL statements can be run by using the sql methods provided by Spark
val teenagersDF = spark.sql("SELECT name, age FROM people WHERE age
BETWEEN 13 AND 19")

// The columns of a row in the result can be accessed by field index
teenagersDF.map(teenager => "Name: " + teenager(0)).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+

// or by field name
teenagersDF.map(teenager => "Name: " + teenager.getAs[String]
("name")).show()
// +------------+
// |       value|
// +------------+
// |Name: Justin|
```

```
// +------------+

// No pre-defined encoders for Dataset[Map[K,V]], define explicitly
implicit val mapEncoder = org.apache.spark.sql.Encoders.kryo[Map[String, Any]]
// Primitive types and case classes can be also defined as
// implicit val stringIntMapEncoder: Encoder[Map[String, Any]] = ExpressionEncoder()

// row.getValuesMap[T] retrieves multiple columns at once into a Map[String, T]
teenagersDF.map(teenager => teenager.getValuesMap[Any](List("name", "age"))).collect()
// Array(Map("name" -> "Justin", "age" -> 19))
```

Java

```
import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;
import org.apache.spark.api.java.function.MapFunction;
import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;
import org.apache.spark.sql.Encoder;
import org.apache.spark.sql.Encoders;

// Create an RDD of Person objects from a text file
JavaRDD<Person> peopleRDD = spark.read()
.textFile("examples/src/main/resources/people.txt")
.javaRDD()
.map(line -> {
    String[] parts = line.split(",");
    Person person = new Person();
    person.setName(parts[0]);
    person.setAge(Integer.parseInt(parts[1].trim()));
    return person;
});

// Apply a schema to an RDD of JavaBeans to get a DataFrame
Dataset<Row> peopleDF = spark.createDataFrame(peopleRDD, Person.class);
// Register the DataFrame as a temporary view
peopleDF.createOrReplaceTempView("people");

// SQL statements can be run by using the sql methods provided by spark
Dataset<Row> teenagersDF = spark.sql("SELECT name FROM people WHERE age BETWEEN 13 AND 19");

// The columns of a row in the result can be accessed by field index
Encoder<String> stringEncoder = Encoders.STRING();
Dataset<String> teenagerNamesByIndexDF = teenagersDF.map(
    (MapFunction<Row, String>) row -> "Name: " + row.getString(0),
    stringEncoder);
teenagerNamesByIndexDF.show();
// +------------+
// |       value|
// +------------+
// |Name: Justin|
// +------------+
```

```
40
41  // or by field name
42  Dataset<String> teenagerNamesByFieldDF = teenagersDF.map(
43      (MapFunction<Row, String>) row -> "Name: " + row.
<String>getAs("name"),
44      stringEncoder);
45  teenagerNamesByFieldDF.show();
46  // +------------+
47  // |       value|
48  // +------------+
49  // |Name: Justin|
50  // +------------+
```

2. Interfaz Programática

Se puede crear un DataFrame mediante programación con tres pasos:

- Creamos un RDD de tuplas/filas o listas a partir del RDD original;
- Creamos el esquema representado por un `StructType` que coincida con la estructura de tuplas/filas o listas en el RDD creado en el paso 1.
- Aplicamos el esquema al RDD mediante el método `createDataFrame` proporcionado por `SparkSession`

Python

```
1   # Import data types
2   from pyspark.sql.types import StringType, StructType, StructField
3
4   sc = spark.sparkContext
5
6   # Load a text file and convert each line to a Row.
7   lines = sc.textFile("examples/src/main/resources/people.txt")
8   parts = lines.map(lambda l: l.split(","))
9   # Each line is converted to a tuple.
10  people = parts.map(lambda p: (p[0], p[1].strip()))
11
12  # The schema is encoded in a string.
13  schemaString = "name age"
14
15  fields = [StructField(field_name, StringType(), True) for field_name in
schemaString.split()]
16  schema = StructType(fields)
17
18  # Apply the schema to the RDD.
19  schemaPeople = spark.createDataFrame(people, schema)
20
21  # Creates a temporary view using the DataFrame
22  schemaPeople.createOrReplaceTempView("people")
23
24  # SQL can be run over DataFrames that have been registered as a table.
25  results = spark.sql("SELECT name FROM people")
26
27  results.show()
```

```
# +-------+
# |   name|
# +-------+
# |Michael|
# |   Andy|
# | Justin|
# +-------+
```

Scala

```
import org.apache.spark.sql.Row

import org.apache.spark.sql.types._

// Create an RDD
val peopleRDD =
spark.sparkContext.textFile("examples/src/main/resources/people.txt")

// The schema is encoded in a string
val schemaString = "name age"

// Generate the schema based on the string of schema
val fields = schemaString.split(" ")
.map(fieldName => StructField(fieldName, StringType, nullable = true))
val schema = StructType(fields)

// Convert records of the RDD (people) to Rows
val rowRDD = peopleRDD
.map(_.split(","))
.map(attributes => Row(attributes(0), attributes(1).trim))

// Apply the schema to the RDD
val peopleDF = spark.createDataFrame(rowRDD, schema)

// Creates a temporary view using the DataFrame
peopleDF.createOrReplaceTempView("people")

// SQL can be run over a temporary view created using DataFrames
val results = spark.sql("SELECT name FROM people")

// The results of SQL queries are DataFrames and support all the normal
RDD operations
// The columns of a row in the result can be accessed by field index or
by field name
results.map(attributes => "Name: " + attributes(0)).show()
// +-------------+
// |        value|
// +-------------+
// |Name: Michael|
// |   Name: Andy|
// | Name: Justin|
// +-------------+
```

Java

```
import java.util.ArrayList;
```

```java
import java.util.List;

import org.apache.spark.api.java.JavaRDD;
import org.apache.spark.api.java.function.Function;

import org.apache.spark.sql.Dataset;
import org.apache.spark.sql.Row;

import org.apache.spark.sql.types.DataTypes;
import org.apache.spark.sql.types.StructField;
import org.apache.spark.sql.types.StructType;

// Create an RDD
JavaRDD<String> peopleRDD = spark.sparkContext()
.textFile("examples/src/main/resources/people.txt", 1)
.toJavaRDD();

// The schema is encoded in a string
String schemaString = "name age";

// Generate the schema based on the string of schema
List<StructField> fields = new ArrayList<>();
for (String fieldName : schemaString.split(" ")) {
StructField field = DataTypes.createStructField(fieldName,
DataTypes.StringType, true);
fields.add(field);
}
StructType schema = DataTypes.createStructType(fields);

// Convert records of the RDD (people) to Rows
JavaRDD<Row> rowRDD = peopleRDD.map((Function<String, Row>) record -> {
String[] attributes = record.split(",");
return RowFactory.create(attributes[0], attributes[1].trim());
});

// Apply the schema to the RDD
Dataset<Row> peopleDataFrame = spark.createDataFrame(rowRDD, schema);

// Creates a temporary view using the DataFrame
peopleDataFrame.createOrReplaceTempView("people");

// SQL can be run over a temporary view created using DataFrames
Dataset<Row> results = spark.sql("SELECT name FROM people");

// The results of SQL queries are DataFrames and support all the normal
RDD operations
// The columns of a row in the result can be accessed by field index or
by field name
Dataset<String> namesDS = results.map(
    (MapFunction<Row, String>) row -> "Name: " + row.getString(0),
    Encoders.STRING());
namesDS.show();
// +-------------+
// |        value|
// +-------------+
// |Name: Michael|
// |   Name: Andy|
```

```
// | Name: Justin|
// +-------------+
```

Esta es sólo una pequeña muestra de la Spark API. Puedes consultar la documentación completa en la página oficial

Ejemplo

```
import sys
from pyspark.sql import SparkSession

spark = SparkSession.builder.getOrCreate()

df = spark.read.options(header='True', inferSchema='True').csv(
sys.argv[1] )

def myFunc(s):
  if s["brand"]=="riche" and s["event_type"]=="cart":
    return [ ( s["product_id"], 1) ]
  return []

lines=df.rdd.flatMap(myFunc).reduceByKey(lambda a, b: a + b)

lines.saveAsTextFile( sys.argv[2] )
```

4. Ejemplo

Siguiendo con los ejemplos anteriores, vamos a lanzar la aplicación `wordcount` de python que nos facilitan los propios ejemplos de Spark , para contar las palabras de `El_quijote.txt` ya usado en otras ocasiones (Así, también nos podría servir para establecer una comparativa entre la ejecución en Yarn on MR y en Spark)

1. Iniciamos HDFS y Spark

```
start-dfs.sh
./sbin/start-master.sh
./sbin/start-workers.sh
```

2. Copiamos el fichero `El_quijote.txt` en HDFS

```
wget
https://gist.githubusercontent.com/jaimerabasco/cb528c32b4c4092e6a0763d8b6bc25c0/
hdfs dfs -mkdir -p /bda/spark/ejemplos
hdfs dfs -copyFromLocal El_Quijote.txt /bda/spark/ejemplos
```

3. Ejecutamos `wordcount` en Spark

```

```

```
spark-submit --master spark://192.168.11.10:7077
examples/src/main/python/wordcount.py /bda/spark/ejemplos/El_Quijote.txt
```

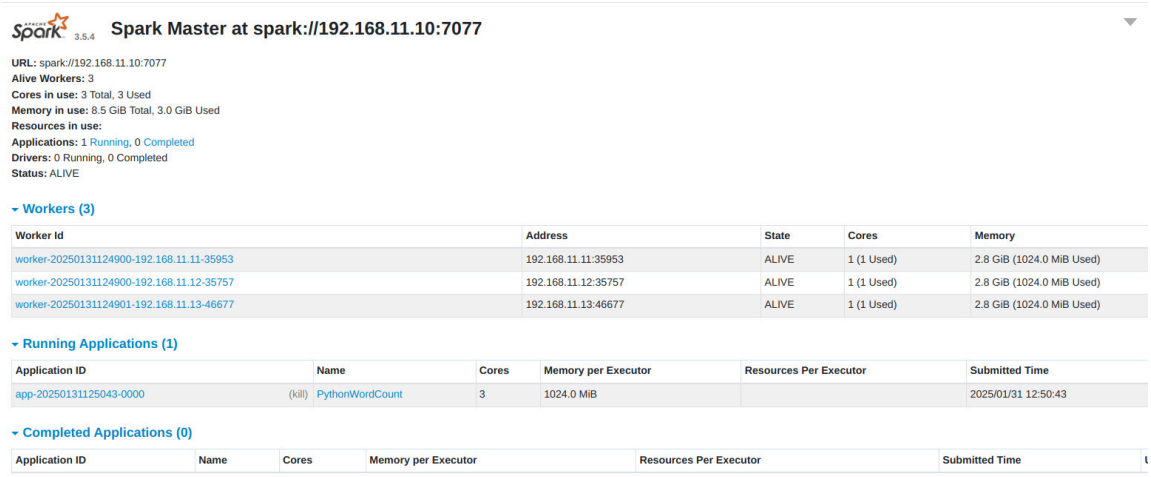

Figura 6.3_Spark API: Ejecución app wordcount

5. Pandas on PySpark

Figura 6.1_Spark API Pandas: Logo Pandas (Fuente: pandas.pydata.org)

5.1 ¿Qué es Pandas?

Pandas es una biblioteca de software de código abierto diseñada específicamente para la manipulación y el análisis de datos en el lenguaje Python. Es potente, flexible y fácil de usar.

Gracias a Pandas, podemos utilizar el lenguaje Python para cargar, alinear, manipular o incluso fusionar datos. Tiene un alto rendimiento.

El nombre "Pandas" es en realidad una contracción del término «Panel Data» para series de datos que incluyen observaciones a lo largo de varios periodos de tiempo. La biblioteca se creó como herramienta de alto nivel para el análisis en Python. Los creadores de Pandas pretenden que esta biblioteca evolucione hasta convertirse en la herramienta de análisis y manipulación de datos de código abierto más potente y flexible en cualquier lenguaje de programación.

Además del análisis de datos, Pandas se utiliza mucho para la «Data Wrangling«. Este término engloba los métodos de transformación de datos no estructurados para hacerlos procesables.

Puedes consultar toda la documentación en la página oficial de Pandas

5.2 Funcionamiento

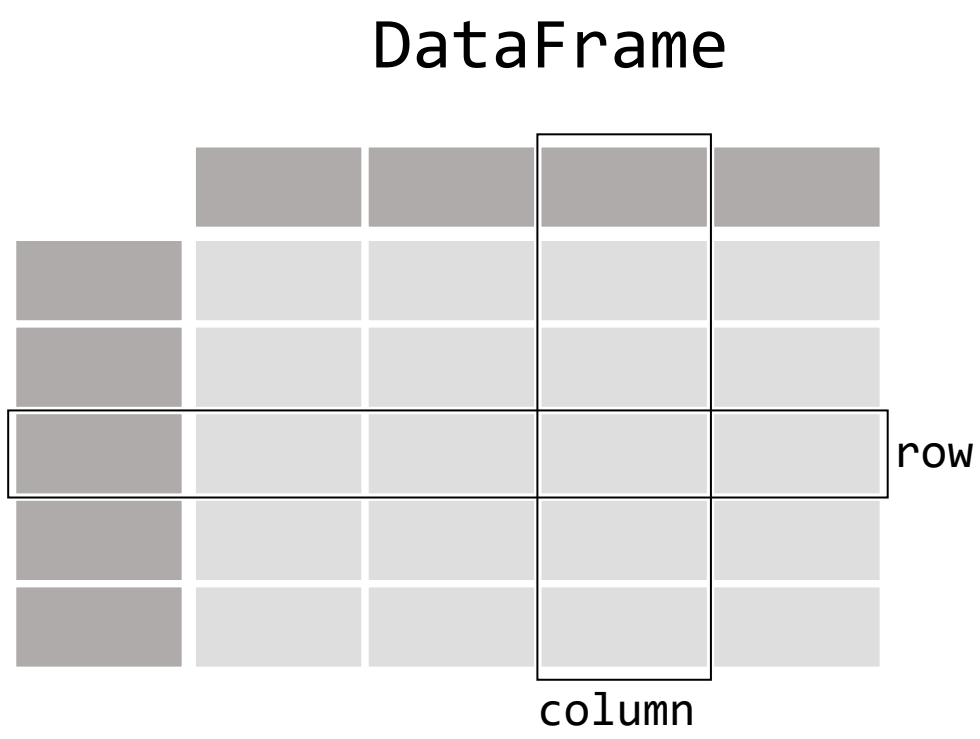

Figura 6.2_Spark API Pandas: Dataframe (Fuente: pandas.pydata.org)

Pandas esta construida con el paquete **Numpy** que es una librería de python para la manipulación de matrices y vectores n-dimensional. Entonces un **Dataframe** en pandas es la estructura de datos clave que va a permitir la manipulación de datos tabulados en filas y columnas.

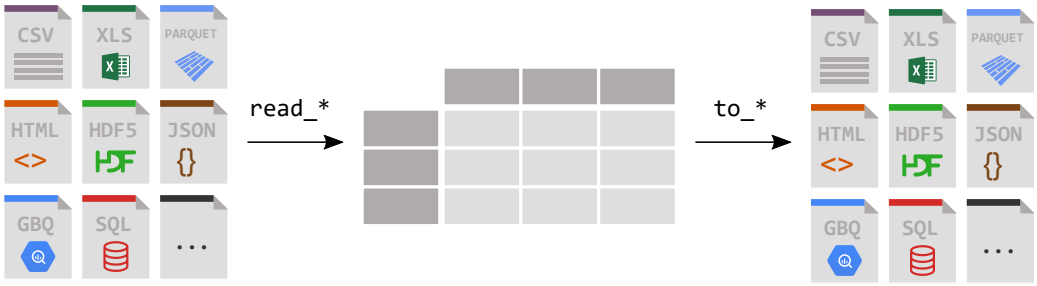

Figura 6.3_Spark API Pandas: Formatos (Fuente: pandas.pydata.org)

Admite la integración con muchos formatos de archivos o fuentes de datos listas para usar (csv, excel, sql, json, parquet,...). La importación de datos de cada una de estas fuentes de datos se realiza mediante una función con el prefijo `read_*`. De manera similar, los métodos `to_*` se utilizan para almacenar datos.

Además, te permite realizar numerosas operaciones sobre los **DataFrame muy eficientemente**, tales como:

- **Selección de un subconjunto**

Permite seleccionar y filtrar datos según cualquier condición y extraer los datos que queramos.

Figura 6.4_Spark API Pandas: Subconjunto (Fuente: pandas.pydata.org)

- **Crear tramas (plot)**

Pandas permite "dibujar" sus datos de forma inmediata gracias a *Matplotlib*. Podemos elegir el tipo de gráfico (dispersión, barras, diagrama de caja,...) correspondiente a los datos.

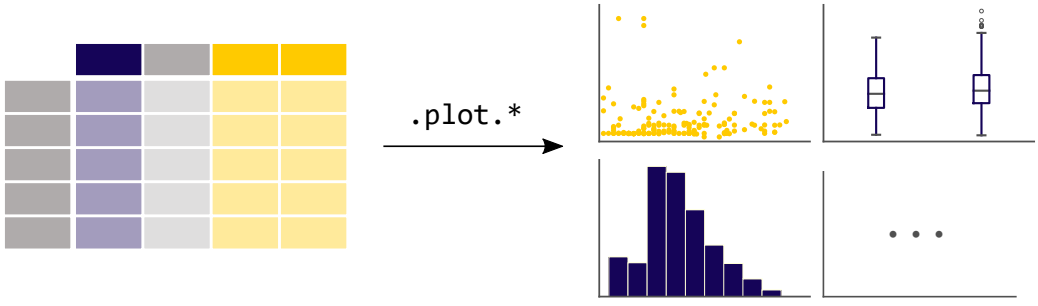

Figura 6.5_Spark API Pandas: Tramas (Fuente: pandas.pydata.org)

- **Crear nuevas columnas derivadas de columnas existentes**

No es necesario recorrer todas las filas de su tabla de datos para realizar cálculos. Las manipulaciones de datos en una columna funcionan por elementos. Así que podemos agregar una columna a un DataFrame en función de los datos existentes de forma sencilla y eficiente.

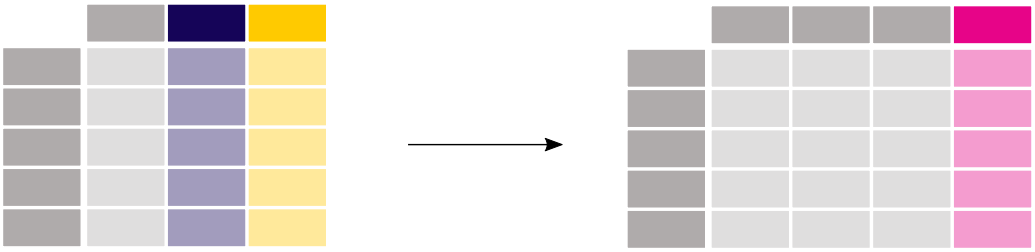

Figura 6.3_Spark API Pandas: Columnas nuevas (Fuente: pandas.pydata.org)

- **Calcular estadísticas**

Podemos realizar estadísticas básicas de forma sencilla (media, mediana, mínimo, máximo, recuentos...). Estas agregaciones personalizadas pueden aplicarse a todo el conjunto de datos, a un subconjunto de datos o agruparse por categorías. Este último también se conoce como enfoque de división, aplicación y combinación.

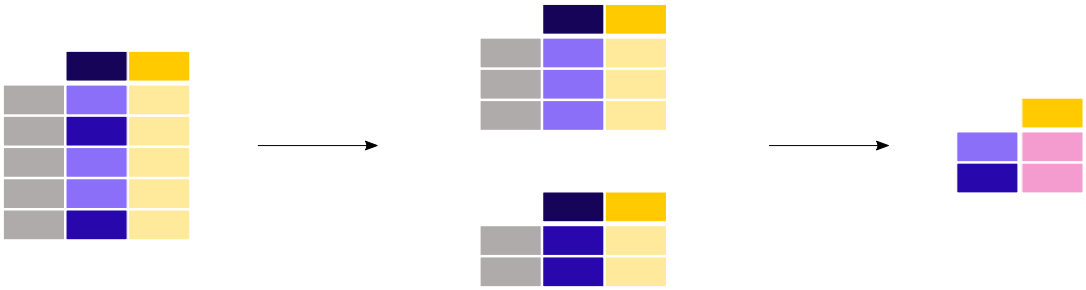

Figura 6.7_Spark API Pandas: Formatos (Fuente: pandas.pydata.org)

- **Remodelar la estructura de los datos**

Podemos cambiar la estructura de la tabla de datos de varias maneras. Podemos fundir (*melt()*) o pivotar (*pivot()*) el formato de la tabla de datos. Con agregaciones integradas, podemos crear una tabla dinámica con un solo comando.

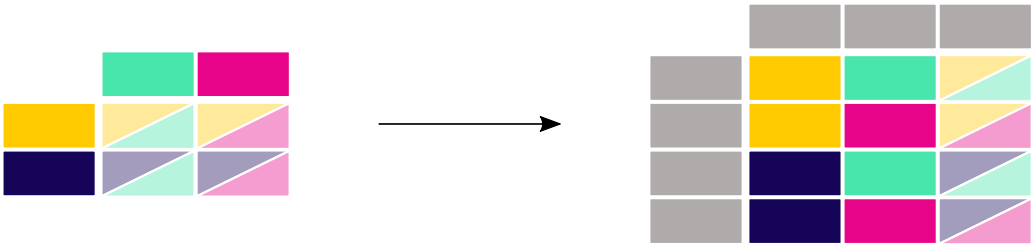

Figura 6.8_Spark API Pandas: Melt (Fuente: pandas.pydata.org)

- **Combinar datos de varias tablas**

Podemos concatenar varias tablas tanto por columnas como por filas, ya que pandas proporciona operaciones de unión/fusión similares a las de una base de datos para combinar varias tablas.

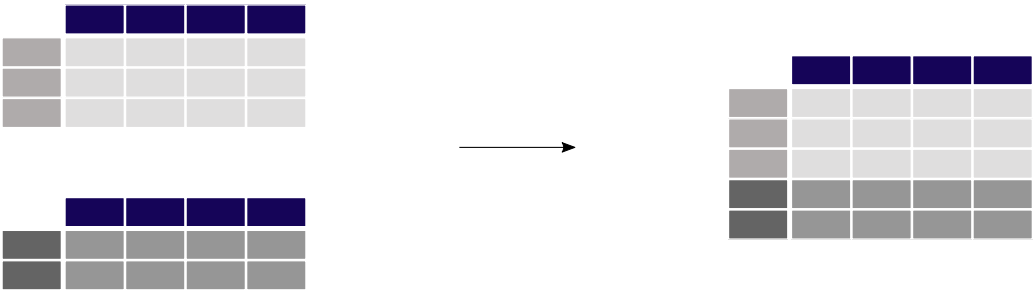

Figura 6.9_Spark API Pandas: Concatenar (Fuente: pandas.pydata.org)

- **Manejar datos de series de tiempo**

Pandas tiene un gran soporte para series temporales y un amplio conjunto de herramientas para trabajar con fechas, horas y datos indexados en el tiempo.

- **Manipular datos de texto**

Pandas también proporciona una amplia gama de funciones para limpiar datos textuales y extraer información útil de ellos.

> ✓ **Success**
>
> En temas de rendimiento **la máxima cantidad de registros que soporta pandas es de alrededor de 2 millones**. Si tienes un archivo con numero de registros mayor a esta cantidad una buena opción seria trabajarlo con **Spark**.

5.3 Quick Started Pandas on Pyspark

Vamos a ver una introducción a la API de pandas en Spark, dirigida principalmente a nuevos usuarios. Mostraremos algunas diferencias clave entre pandas y la API de pandas en Spark.

1. Importamos la API de pandas en Spark de la siguiente manera:

```
import pandas as pd
import numpy as np
import pyspark.pandas as ps
from pyspark.sql import SparkSession
```

Creación de objetos

2. Creamos una serie de pandas-on-Spark pasando una lista de valores, permitiendo que la API de pandas en Spark cree un índice entero predeterminado:

```
s = ps.Series([1, 3, 5, np.nan, 6, 8])
s
......
0    1.0
1    3.0
2    5.0
3    NaN
4    6.0
5    8.0
dtype: float64
```

3. Creamos un DataFrame de pandas-on-Spark pasando un dictado de objetos que se pueden convertir en series.

```
psdf = ps.DataFrame(
    {'a': [1, 2, 3, 4, 5, 6],
     'b': [100, 200, 300, 400, 500, 600],
     'c': ["one", "two", "three", "four", "five", "six"]},
    index=[10, 20, 30, 40, 50, 60])
psdf
......
    a   b   c
10  1   100 one
20  2   200 two
30  3   300 three
40  4   400 four
50  5   500 five
60  6   600 six
```

4. Creamos un DataFrame de pandas pasando una matriz numpy, con un índice de fecha y hora y columnas etiquetadas:

```
dates = pd.date_range('20130101', periods=6)
dates
......
DatetimeIndex(['2013-01-01', '2013-01-02', '2013-01-03', '2013-01-04',
```

```
               '2013-01-05', '2013-01-06'],
              dtype='datetime64[ns]', freq='D')
```

```
pdf = pd.DataFrame(np.random.randn(6, 4), index=dates,
columns=list('ABCD'))
pdf
A   B   C   D
2013-01-01  0.912558    -0.795645   -0.289115   0.187606
2013-01-02  -0.059703   -1.233897   0.316625    -1.226828
2013-01-03  0.332871    -1.262010   -0.434844   -0.579920
2013-01-04  0.924016    -1.022019   -0.405249   -1.036021
2013-01-05  -0.772209   -1.228099   0.068901    0.896679
2013-01-06  1.485582    -0.709306   -0.202637   -0.248766
```

5. Ahora, este DataFrame de pandas puede convertirse en un DataFrame de pandas on Spark

```
psdf = ps.from_pandas(pdf)
type(psdf)
....
pyspark.pandas.frame.DataFrame
```

6. Se ve y se comporta igual que un DataFrame de pandas.

```
psdf
A   B   C   D
2013-01-01  0.912558    -0.795645   -0.289115   0.187606
2013-01-02  -0.059703   -1.233897   0.316625    -1.226828
2013-01-03  0.332871    -1.262010   -0.434844   -0.579920
2013-01-04  0.924016    -1.022019   -0.405249   -1.036021
2013-01-05  -0.772209   -1.228099   0.068901    0.896679
2013-01-06  1.485582    -0.709306   -0.202637   -0.248766
```

7. Además, es posible crear fácilmente un DataFrame de pandas en Spark desde Spark DataFrame.

8. Creando un Spark DataFrame a partir de pandas DataFrame

```
spark = SparkSession.builder.getOrCreate()
sdf = spark.createDataFrame(pdf)
sdf.show()
........
+--------------------+-------------------+--------------------+----------
----------+
|                   A|                  B|                   C|
D|
+--------------------+-------------------+--------------------+----------
----------+
|    0.91255803205208|-0.7956452608556638|-0.28911463069772175|
0.18760566615081622|
|-0.05970271470242...| -1.233896949308984|  0.3166246451758431|
-1.2268284000402265|
```

```
  | 0.33287106947536615|-1.2620100816441786| -0.4348444277082644|
-0.5799199651437185|
  |  0.9240158461589916|-1.0220190956326003| -0.4052488880650239|
-1.0360212104348547|
  | -0.7722090016558953|-1.2280986385313222|  0.0689011451939635|
0.8966790729426755|
  |  1.4855822995785612|-0.7093056426018517|
-0.2026366848847041|-0.24876619876451092|
  +--------------------+-------------------+--------------------+----------
----------+
```

9. Creando pandas-on-Spark DataFrame desde Spark DataFrame.

```
psdf = sdf.pandas_api()
psdf
.........
A   B   C   D
0   0.912558    -0.795645   -0.289115   0.187606
1   -0.059703   -1.233897   0.316625    -1.226828
2   0.332871    -1.262010   -0.434844   -0.579920
3   0.924016    -1.022019   -0.405249   -1.036021
4   -0.772209   -1.228099   0.068901    0.896679
5   1.485582    -0.709306   -0.202637   -0.248766
```

10. Tipos específicos dtypes. Actualmente se admiten tipos que son comunes tanto para Spark como para pandas.

```
psdf.dtypes
.......
A    float64
B    float64
C    float64
D    float64
dtype: object
```

11. A continuación explicamos cómo mostrar las filas superiores del cuadro siguiente. Hay que tener en cuenta que los DataFrame de Spark no conservan el orden natural de forma predeterminada. El orden natural se puede preservar configurando la opción `compute.ordered_head`, pero provoca una sobrecarga de rendimiento con la clasificación interna.

```
psdf.head()
..................
A   B   C   D
0   0.912558    -0.795645   -0.289115   0.187606
1   -0.059703   -1.233897   0.316625    -1.226828
2   0.332871    -1.262010   -0.434844   -0.579920
3   0.924016    -1.022019   -0.405249   -1.036021
4   -0.772209   -1.228099   0.068901    0.896679
```

12. Mostrando el índice, las columnas y los numerosos datos subyacentes.

```
psdf.index
.....
Int64Index([0, 1, 2, 3, 4, 5], dtype='int64')
```

```
psdf.columns
.....
Index(['A', 'B', 'C', 'D'], dtype='object')
```

```
psdf.to_numpy()
.....
array([[ 0.91255803, -0.79564526, -0.28911463,  0.18760567],
       [-0.05970271, -1.23389695,  0.31662465, -1.2268284 ],
       [ 0.33287107, -1.26201008, -0.43484443, -0.57991997],
       [ 0.92401585, -1.0220191 , -0.40524889, -1.03602121],
       [-0.772209  , -1.22809864,  0.06890115,  0.89667907],
       [ 1.4855823 , -0.70930564, -0.20263668, -0.2487662 ]])
```

12. Mostrando un resumen estadístico rápido de sus datos

```
psdf.describe()
.....
A   B   C   D
count   6.000000    6.000000    6.000000    6.000000
mean    0.470519    -1.041829   -0.157720   -0.334542
std 0.809428    0.241511    0.294520    0.793014
min -0.772209   -1.262010   -0.434844   -1.226828
25% -0.059703   -1.233897   -0.405249   -1.036021
50% 0.332871    -1.228099   -0.289115   -0.579920
75% 0.924016    -0.795645   0.068901    0.187606
max 1.485582    -0.709306   0.316625    0.896679
```

13. Transponiendo los datos

```
psdf.T
.............
0   1   2   3   4   5
A   0.912558    -0.059703   0.332871    0.924016    -0.772209   1.485582
B   -0.795645   -1.233897   -1.262010   -1.022019   -1.228099   -0.709306
C   -0.289115   0.316625    -0.434844   -0.405249   0.068901    -0.202637
D   0.187606    -1.226828   -0.579920   -1.036021   0.896679    -0.248766
```

14. Ordenando por índice

```
psdf.sort_index(ascending=False)
.............
A   B   C   D
5   1.485582    -0.709306   -0.202637   -0.248766
4   -0.772209   -1.228099   0.068901    0.896679
3   0.924016    -1.022019   -0.405249   -1.036021
2   0.332871    -1.262010   -0.434844   -0.579920
1   -0.059703   -1.233897   0.316625    -1.226828
```

```
9  0   0.912558    -0.795645   -0.289115   0.187606
```

15. Ordenando por valor

```
psdf.sort_values(by='B')
..................
A   B   C   D
2   0.332871    -1.262010   -0.434844   -0.579920
1   -0.059703   -1.233897   0.316625    -1.226828
4   -0.772209   -1.228099   0.068901    0.896679
3   0.924016    -1.022019   -0.405249   -1.036021
0   0.912558    -0.795645   -0.289115   0.187606
5   1.485582    -0.709306   -0.202637   -0.248766
```

Datos perdidos (Missing Data)

16. La API de Pandas en Spark utiliza principalmente el valor `np.nan` para representar los datos perdidos. Por defecto no se incluye en los cálculos.

```
pdf1 = pdf.reindex(index=dates[0:4], columns=list(pdf.columns) + ['E'])
pdf1.loc[dates[0]:dates[1], 'E'] = 1
psdf1 = ps.from_pandas(pdf1)
psdf1
..................
A   B   C   D   E
2013-01-01  0.912558    -0.795645   -0.289115   0.187606    1.0
2013-01-02  -0.059703   -1.233897   0.316625    -1.226828   1.0
2013-01-03  0.332871    -1.262010   -0.434844   -0.579920   NaN
2013-01-04  0.924016    -1.022019   -0.405249   -1.036021   NaN
```

17. Eliminar cualquier fila a la que le falten datos.

```
psdf1.dropna(how='any')
..............
A   B   C   D   E
2013-01-01  0.912558    -0.795645   -0.289115   0.187606    1.0
2013-01-02  -0.059703   -1.233897   0.316625    -1.226828   1.0
```

18. Llenando datos perdidos.

```
psdf1.fillna(value=5)
......................
A   B   C   D   E
2013-01-01  0.912558    -0.795645   -0.289115   0.187606    1.0
2013-01-02  -0.059703   -1.233897   0.316625    -1.226828   1.0
2013-01-03  0.332871    -1.262010   -0.434844   -0.579920   5.0
2013-01-04  0.924016    -1.022019   -0.405249   -1.036021   5.0
```

Operaciones

19. **Estadísticas**. Realizando una estadística descriptiva:

```
psdf.mean()
.......
A    0.470519
B   -1.041829
C   -0.157720
D   -0.334542
dtype: float64
```

20. **Configuraciones Spark**. Se podrían aplicar varias configuraciones en PySpark internamente en la API de pandas en Spark. Por ejemplo, podemos habilitar la optimización de Arrow para acelerar enormemente la conversión interna de pandas. Consulte también la Guía de uso de PySpark para Pandas con Apache Arrow en la documentación de PySpark.

```
rev = spark.conf.get("spark.sql.execution.arrow.pyspark.enabled")  # Keep
its default value.
ps.set_option("compute.default_index_type", "distributed")  # Use default
index prevent overhead.
import warnings
warnings.filterwarnings("ignore")  # Ignore warnings coming from Arrow
optimizations.
```

```
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", True)
%timeit ps.range(300000).to_pandas()
..........
900 ms ± 186 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", False)
%timeit ps.range(300000).to_pandas()
...............
3.08 s ± 227 ms per loop (mean ± std. dev. of 7 runs, 1 loop each)
```

```
ps.reset_option("compute.default_index_type")
spark.conf.set("spark.sql.execution.arrow.pyspark.enabled", prev)  # Set
its default value back.
```

Agrupamiento (Grouping)

21. Por “agrupar por” nos referimos a un proceso que involucra uno o más de los siguientes pasos:

- Dividir los datos en grupos según algunos criterios.
- Aplicar una función a cada grupo de forma independiente
- Combinar los resultados en una estructura de datos.

```
psdf = ps.DataFrame({'A': ['foo', 'bar', 'foo', 'bar',
                           'foo', 'bar', 'foo', 'foo'],
                     'B': ['one', 'one', 'two', 'three',
```

```
                           'two', 'two', 'one', 'three'],
                   'C': np.random.randn(8),
                   'D': np.random.randn(8)})
psdf
.....................
A   B   C   D
0   foo one 1.039632    -0.571950
1   bar one 0.972089    1.085353
2   foo two -1.931621   -2.579164
3   bar three   -0.654371   -0.340704
4   foo two -0.157080   0.893736
5   bar two 0.882795    0.024978
6   foo one -0.149384   0.201667
7   foo three   -1.355136   0.693883
```

22. Agrupar y luego aplicar la función sum() a los grupos resultantes.

```
psdf.groupby('A').sum()
.............
    C   D
A
bar 1.200513    0.769627
foo -2.553589   -1.361828
```

23. La agrupación por varias columnas forma un índice jerárquico y nuevamente podemos aplicar la función de suma.

```
psdf.groupby(['A', 'B']).sum()
        C   D
A   B
foo one 0.890248    -0.370283
two -2.088701   -1.685428
bar three   -0.654371   -0.340704
foo three   -1.355136   0.693883
bar two 0.882795    0.024978
one 0.972089    1.085353
```

Plotting

Estos métodos deben ejecutarse en un notebook tipo Jupyter, por ejemplo: databricks o jupyter en pyspark(lo vemos en el siguiente punto). También en google colab, entre otros.

```
pser = pd.Series(np.random.randn(1000),
                 index=pd.date_range('1/1/2000', periods=1000))
psser = ps.Series(pser)
psser = psser.cummax()
psser.plot()
```

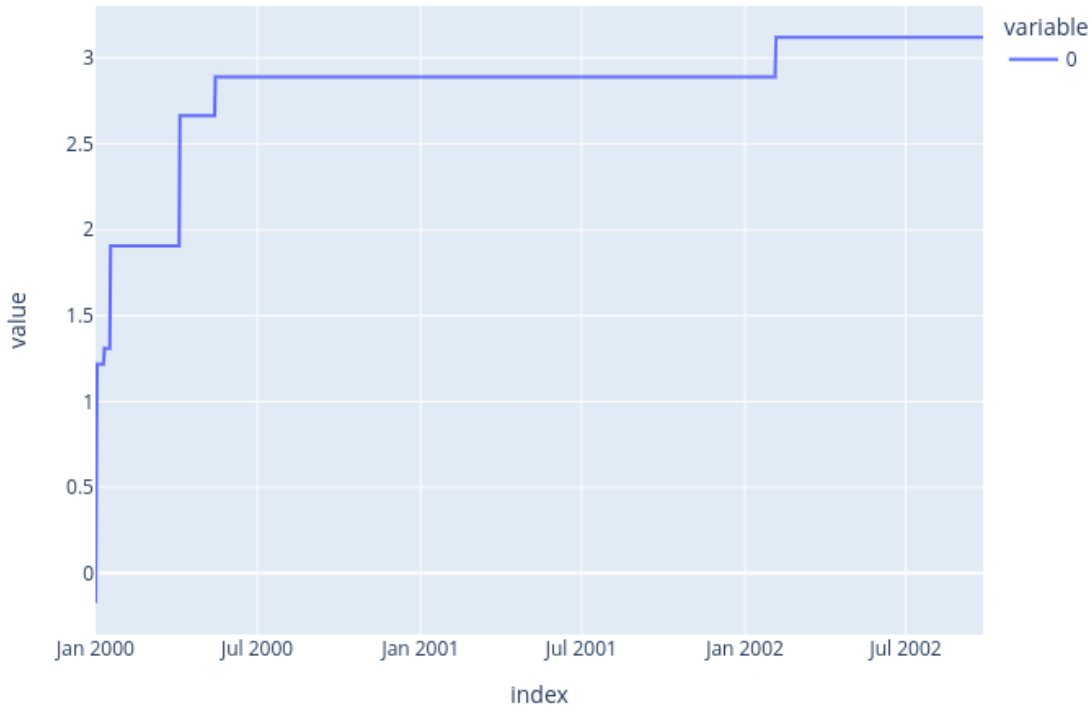

Figura 6.9_Spark API Pandas: Plot 1

1. En un DataFrame, es conveniente usar el método `plot()` para trazar todas las columnas con etiquetas:

```
pdf = pd.DataFrame(np.random.randn(1000, 4), index=pser.index,
                   columns=['A', 'B', 'C', 'D'])
psdf = ps.from_pandas(pdf)
psdf = psdf.cummax()
psdf.plot()
```

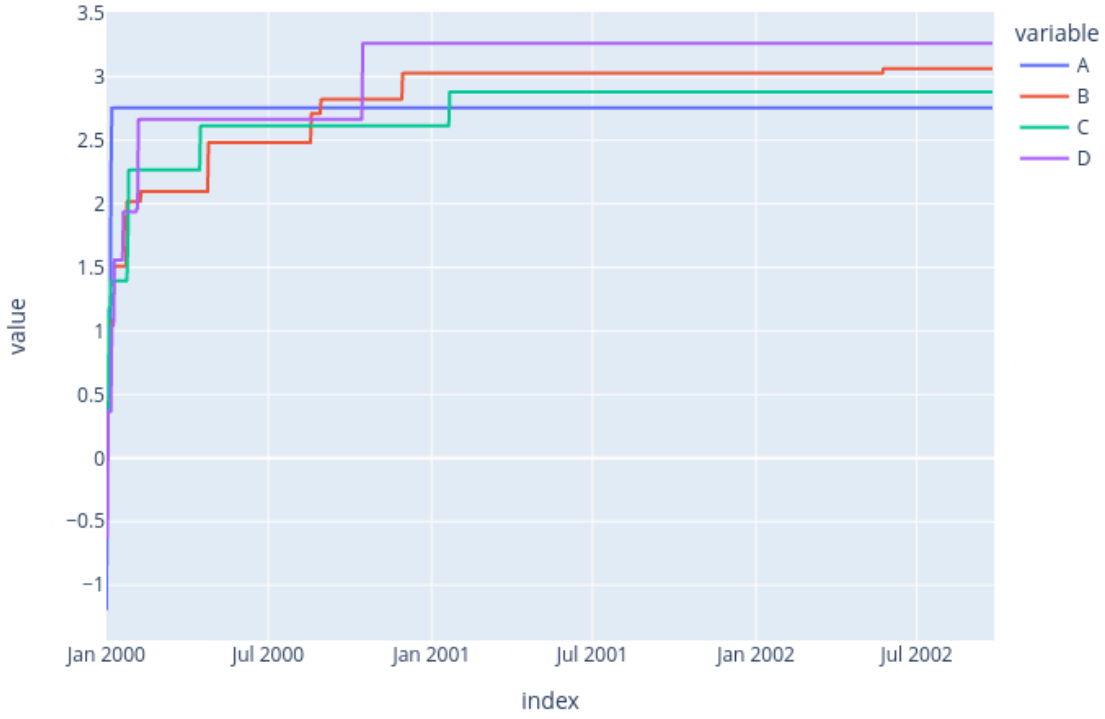

Figura 6.10_Spark API Pandas: Plot 2

Entrada/Salida de datos

25. **CSV**

```
psdf.to_csv('foo.csv')
ps.read_csv('foo.csv').head(10)
......................
A   B   C   D
0   -1.187097   -0.134645   0.377094    -0.627217
1   0.331741    0.166218    0.377094    -0.627217
2   0.331741    0.439450    0.377094    0.365970
3   0.621620    0.439450    1.190180    0.365970
4   0.621620    0.439450    1.190180    0.365970
5   2.169198    1.069183    1.395642    0.365970
6   2.755738    1.069183    1.395642    1.045868
7   2.755738    1.069183    1.395642    1.045868
8   2.755738    1.069183    1.395642    1.045868
9   2.755738    1.508732    1.395642    1.556933
```

26. **Parquet**. Parquet es un formato de archivo eficiente y compacto para leer y escribir más rápido.

```
df.to_parquet('bar.parquet')
ps.read_parquet('bar.parquet').head(10)
...............
```

```
   A   B   C   D
0   -1.187097   -0.134645   0.377094    -0.627217
1   0.331741    0.166218    0.377094    -0.627217
2   0.331741    0.439450    0.377094    0.365970
3   0.621620    0.439450    1.190180    0.365970
4   0.621620    0.439450    1.190180    0.365970
5   2.169198    1.069183    1.395642    0.365970
6   2.755738    1.069183    1.395642    1.045868
7   2.755738    1.069183    1.395642    1.045868
8   2.755738    1.069183    1.395642    1.045868
9   2.755738    1.508732    1.395642    1.556933
```

27. **Spark IO**. Además, la API de pandas en Spark es totalmente compatible con las diversas fuentes de datos de Spark, como ORC y una fuente de datos externa.

```
psdf.to_spark_io('zoo.orc', format="orc")
ps.read_spark_io('zoo.orc', format="orc").head(10)
..................
   A   B   C   D
0   -1.187097   -0.134645   0.377094    -0.627217
1   0.331741    0.166218    0.377094    -0.627217
2   0.331741    0.439450    0.377094    0.365970
3   0.621620    0.439450    1.190180    0.365970
4   0.621620    0.439450    1.190180    0.365970
5   2.169198    1.069183    1.395642    0.365970
6   2.755738    1.069183    1.395642    1.045868
7   2.755738    1.069183    1.395642    1.045868
8   2.755738    1.069183    1.395642    1.045868
9   2.755738    1.508732    1.395642    1.556933
```