Una chispa para dominarlos a todos

Daniel Sánchez Castelló (a.k.a. Dani Sancas)
Twitter: @SancasDev / GitHub: DaniSancas

Big Data Engineer @ Everis Bilbao
PEUMConf 2020

Índice de contenidos

Introducción

¿Qué es Big Data?

Ilustrémoslo con un poema...

Poema del framework único

Tres lenguajes de programación para los developers en sus portátiles con Windows Vista,

Siete servidores para las sysadminas en el CPD de la planta -2,

Nueve terabytes de datos para los DBA's condenados a repetir SQL's.

Un framework para (do)minarlos a todos. Una chispa para iluminar los datos y consumir recursos como si no hubiera mañana, en la Tierra de Big Data donde proliferan los vendehumos.

Introducción - Big Data

¿Qué es?

Término utilizado para referirse tanto a grandes volúmenes de datos como a su procesamiento masivo.

¿Para qué?

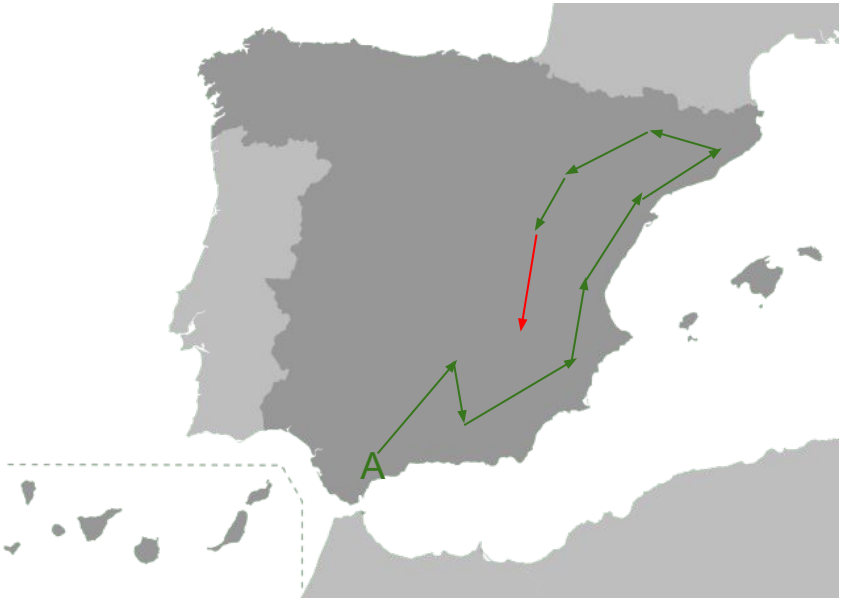
- Para superar las barreras derivadas del gran tamaño de los datos.
- Con computación típica (1 servidor y a menudo 1 core destinado al procesado) el procesamiento se haría eterno, o fallaría por falta de memoria.
- Con Big Data (múltiples servidores y múltiples cores por servidor) exprimimos al máximo los recursos, disminuyendo enormemente el tiempo de procesamiento.

Introducción - Big Data

¿Cómo?

- Computación distribuida y paralelizada.
- Tolerancia a fallos y resiliencia.
- A menudo se utiliza un stack de diferentes herramientas y servicios (cada uno con un rol) desplegados en múltiples servidores

Introducción - Ejemplo de computación single core



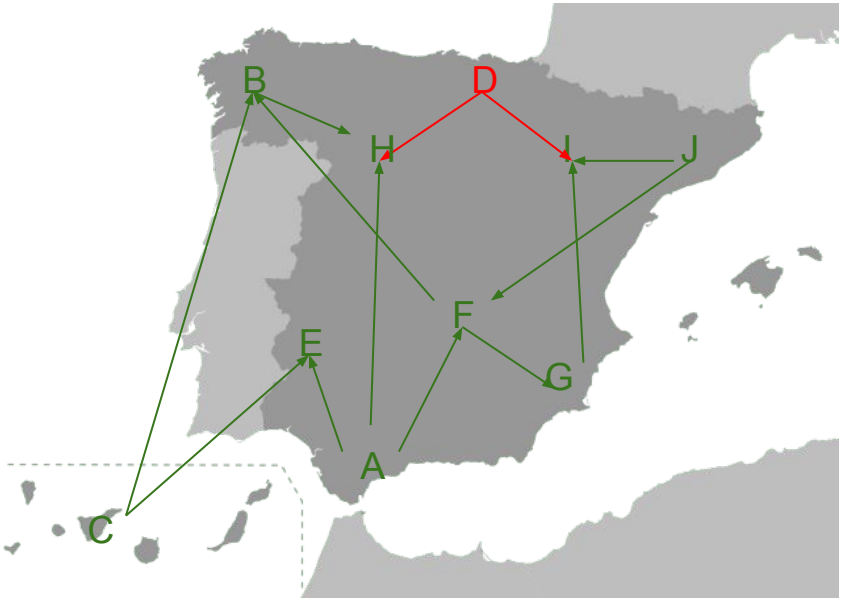
Queremos realizar el censo de España.

Disponemos únicamente de 1 ÚNICA persona, que se recorrerá los distritos de TODOS los municipios del país.

Es una actividad lenta, ardua y tediosa.

Si le pasara algo (flecha roja) a nuestro delegado, tendríamos que empezar de cero (no hay tolerancia a fallos ni resiliencia)

Introducción - Ejemplo de computación distribuida



Queremos realizar el censo de España.

Disponemos de 1 persona en CADA distrito de cada municipio del país.

Es una actividad rápida y paralelizable, los diferentes delegados se comunican entre ellos.

Si le pasara algo (flechas rojas) a un delegado, bastaría con poner a otro en su lugar. No perderíamos más que una pequeña parte del trabajo total, que sería fácilmente recuperable (tolerancia a fallos y resiliencia)

Introducción - Ejemplo de stack clásico (Hadoop)

Motor de ejecución

Procesamiento en batch

Almacenamiento distribuido

Negociación de recursos y ejecución contenerizada

Data warehouse y metastore

Procesamiento en tiempo real

Analítica y machine learning

Y… ¿qué pinta Spark en todo esto?

Introducción - Qué es Spark

¿Qué es Spark?

- Framework open source de computación distribuida
- Nacido en 2012, versión 1.0 en 2014 y actualmente mantenido por Apache
- Escrito para la JVM, con APIs en Scala, Java, Python (y R)
- Arquitectura Maestro-Esclavo (Driver-Executor en la jerga de Spark)
- A menudo forma parte de stacks de Big Data en entornos Hadoop (Hortonworks/Cloudera, proveedores cloud…) o en modo standalone
- Usos habituales:
  - Ingesta de datos
  - Preprocesado de datos (limpieza, agregados, etc…)
  - Minado y analítica de datos

Introducción - Ventajas

Ventajas

- Computación en clúster (distribuida), resiliente, tolerante a fallos
- Sintaxis sencilla
- Soporta ejecuciones batch y real-time
- Posee un módulo exclusivo para machine learning
- Gran abstracción de los problemas de la computación distribuida
- Fácilmente escalable (vertical y horizontalmente)
- Soporta volumetrías enormes (gigabytes, terabytes, petabytes…)
- Fácilmente integrable con otras herramientas/servicios Big Data (multitud de conectores)

Introducción - Desventajas

Desventajas

- (Potencialmente) gran consumo de recursos
- Fine-tunning costoso (conocimiento, tiempo, esfuerzo)
- Los errores por “memory overhead” serán vuestros amigos inseparables

Introducción - Ejemplo de stack clásico (Hadoop)

Motor de ejecución

Procesamiento en batch

Almacenamiento distribuido

Negociación de recursos y ejecución contenerizada

Data warehouse y metastore

Procesamiento en tiempo real

Analítica y machine learning

Introducción - Spark, el framework único

Motor de ejecución

Procesamiento en batch

Almacenamiento distribuido

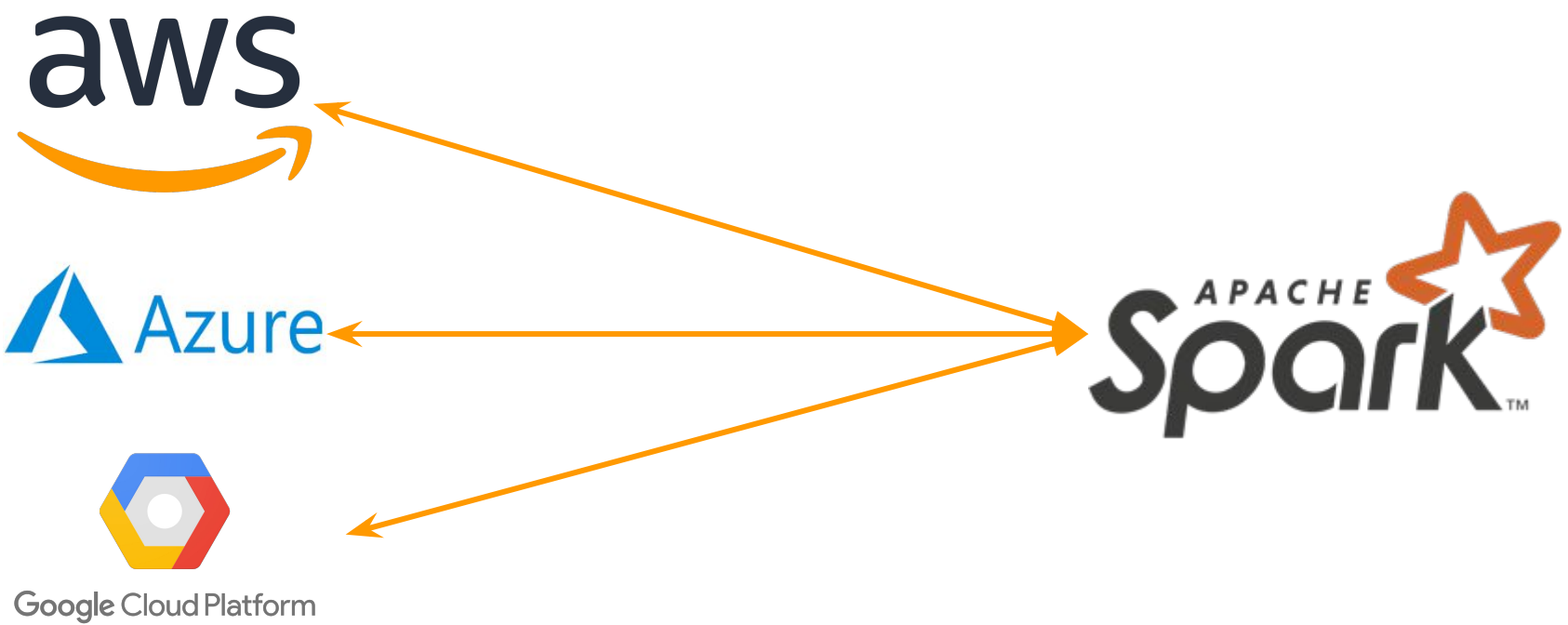
Negociación de recursos y ejecución contenerizada

Data warehouse y metastore

Procesamiento en tiempo real

Analítica y machine learning

Introducción - Almacenamiento y servicios cloud



Sintaxis y ejemplos

(en Scala, porque mola)

Sintaxis y ejemplos - Dependencias e inicialización

```
// Añadimos dependencias a nuestro build.sbt

scalaVersion := "2.12.10"

libraryDependencies +=
  "org.apache.spark" %% "spark-sql" % "3.0.1"


// ...o a nuestro pom.xml

<dependency>
  <groupId>org.apache.spark</groupId>
  <artifactId>spark-sql_2.12</artifactId>
  <version>3.0.1</version>
  <scope>provided</scope>
</dependency>


// ...o mediante pip

pip install pyspark
```

```
// Una vez en nuestro código Scala...

// Iniciamos la sesión de Spark

val spark = SparkSession
    .builder()
    .appName("PeumConf2020")
    .enableHiveSupport()
    .getOrCreate()
```

Sintaxis y ejemplos - Lectura de fuentes

```
// Lectura de fuentes (csv, texto plano, json, avro parquet, tablas Hive / SQL...)

val clients: DataFrame = spark
    .read
    .format("csv")
    .option("sep", "|")
    .option("inferSchema", "true")
    .option("header", "true")
    .load("/path/to/clients.csv")
    .filter($country == "ES")
    .select(
        $"client_id",
        $"name",
        $"zip_code",
        $"country")

val orders: DataFrame = spark
    .sql("SELECT order_id, client_id, order_date FROM peum.orders WHERE order_date < '2020-10-17 00:00:00'")
```

Sintaxis y ejemplos - Operaciones y escritura

```
// Operamos con los DataFrames

val clientOrders: DataFrame = clients
    .join(orders, Seq("client_id"))

val aggClientOrders: DataFrame = clientOrders
    .groupBy("client_id")
    .agg(count("order_id").alias("order_count"))


// Finalmente guardamos el resultado

aggClientOrders
    .write
    .parquet("/path/to/aggdata.parquet")
```

Vale, sí, muy bonito el “Hola mundo”...

Pero… ¿qué hace realmente por dentro?

Funcionamiento interno

Funcionamiento interno - Ejecución paso a paso

```
val clients: DataFrame = spark
    .read.(...).load(...)
    .filter(...)
    .select(...)

val orders: DataFrame = spark
    .sql(...)

val clientOrders: DataFrame = clients
    .join(orders, ...)

val aggClientOrders: DataFrame = clientOrders
    .groupBy(...).agg(...)

aggClientOrders
    .write.parquet(...)
```

Funcionamiento interno - Ejecución paso a paso

```
val clients: DataFrame = spark
    .read.(...).load(...)
    .filter(...)
    .select(...)

val orders: DataFrame = spark
    .sql(...)

val clientOrders: DataFrame = clients
    .join(orders, ...)

val aggClientOrders: DataFrame = clientOrders
    .groupBy(...).agg(...)

aggClientOrders
    .write.parquet(...)
```

Funcionamiento interno - Ejecución paso a paso

```
val clients: DataFrame = spark
    .read.(...).load(...)
    .filter(...)
    .select(...)

val orders: DataFrame = spark
    .sql(...)

val clientOrders: DataFrame = clients
    .join(orders, ...)

val aggClientOrders: DataFrame = clientOrders
    .groupBy(...).agg(...)

aggClientOrders
    .write.parquet(...)
```

Funcionamiento interno - Ejecución paso a paso

```
val clients: DataFrame = spark
    .read.(...).load(...)
    .filter(...)
    .select(...)

val orders: DataFrame = spark
    .sql(...)

val clientOrders: DataFrame = clients
    .join(orders, ...)

val aggClientOrders: DataFrame = clientOrders
    .groupBy(...).agg(...)

aggClientOrders
    .write.parquet(...)
```

Funcionamiento interno - Ejecución paso a paso

```
val clients: DataFrame = spark
    .read.(...).load(...)
    .filter(...)
    .select(...)

val orders: DataFrame = spark
    .sql(...)

val clientOrders: DataFrame = clients
    .join(orders, ...)

val aggClientOrders: DataFrame = clientOrders
    .groupBy(...).agg(...)

aggClientOrders
    .write.parquet(...)
```

Funcionamiento interno - Ejecución paso a paso

```
val clients: DataFrame = spark
    .read.(...).load(...)
    .filter(...)
    .select(...)

val orders: DataFrame = spark
    .sql(...)

val clientOrders: DataFrame = clients
    .join(orders, ...)

val aggClientOrders: DataFrame = clientOrders
    .groupBy(...).agg(...)

aggClientOrders
    .write.parquet(...)
```

Funcionamiento interno - Ejecución paso a paso

```
val clients: DataFrame = spark
    .read.(...).load(...)
    .filter(...)
    .select(...)

val orders: DataFrame = spark
    .sql(...)

val clientOrders: DataFrame = clients
    .join(orders, ...)

val aggClientOrders: DataFrame = clientOrders
    .groupBy(...).agg(...)

aggClientOrders
    .write.parquet(...)
```

Funcionamiento interno - Ejecución paso a paso

```
val clients: DataFrame = spark
    .read.(...).load(...)
    .filter(...)
    .select(...)

val orders: DataFrame = spark
    .sql(...)

val clientOrders: DataFrame = clients
    .join(orders, ...)

val aggClientOrders: DataFrame = clientOrders
    .groupBy(...).agg(...)

aggClientOrders
    .write.parquet(...)
```

Funcionamiento interno - Ejecución paso a paso

```
val clients: DataFrame = spark
    .read.(...).load(...)
    .filter(...)
    .select(...)

val orders: DataFrame = spark
    .sql(...)

val clientOrders: DataFrame = clients
    .join(orders, ...)

val aggClientOrders: DataFrame = clientOrders
    .groupBy(...).agg(...)

aggClientOrders
    .write.parquet(...)
```

Spark implementa “lazy evaluation”

Azul = Transformaciones

- Dan una estructura de Spark en memoria como resultado (p.e: DataFrame)
- Son “lazy”, anotan próximos pasos a realizar, retrasándolo lo máximo posible

Verde = Acciones

- NO dan una estructura de Spark en memoria como resultado (p.e: Array, Unit)
- Son “eager”, disparan todo el proceso de ejecución

Entonces… ahora sí empieza a consultar datos, ¿no?

...Pues no. Todavía faltan preparativos.

Funcionamiento interno - DAG

Construcción del grafo acíclico dirigido (DAG) de operaciones:

- Parsed logical plan
- Analyzed logical plan
- Optimized logical plan
- Physical plan

En otras palabras:

- Analiza la secuencia de operaciones que le pedimos
- Busca la mejor manera (en rendimiento) de organizarlas
- Genera el bytecode resultante
- El Driver transfiere ese bytecode a los Executors para que comiencen a trabajar

Funcionamiento interno - DAG

Ahora sí, los Executors se pondrán a trabajar.

Comenzarán consultando los datos de las fuentes, transformar los datos y, finalmente, escribirlos.

Cada Executor asume un porcentaje de la lectura, un porcentaje de la transformación de datos y, finalmente, un porcentaje de la escritura.

Mientras tanto, el Driver se encarga de coordinar: “Que trabajen otros, ¡que para eso soy el jefe!” - se jacta orgullosamente.

Funcionamiento interno - Cuando algo falla...

Pero… ¿y si un Executor falla? (cortes de red, memory overhead, apagado, etc...)

El Driver se encargará de coordinar el exceso de trabajo. Los Executors restantes asumirán una carga de trabajo extra, para suplir la ausencia del Executor caído.

De esta manera la aplicación conseguirá ejecutarse de principio a fin a pesar de encontrar piedras en el camino, llevando a buen puerto los datos.

Nota: Todas estas características (generación del DAG de operaciones, tolerancia a fallos, re-lanzamiento de tareas ante fallos, etc) son características propias y automáticas de Spark. ¡El/la desarrollador/a no necesita hacer nada al respecto!

Conclusiones

- Spark nos permite procesar datos a gran escala y de manera distribuida
- Simplifica enormemente la complejidad de cara al desarrollador/a
- Código en pocas líneas y con una sintaxis amigable (incluso cotidiana)
- Framework sólido y productivo desde hace años
- Considerado muchas veces la navaja suiza del Big Data

“Programar es un poco menos mierda cuando dispones de herramientas como Spark”

- Alguien con el hype muy subido

¡Muchas gracias
por asistir!

¿Alguna pregunta?

Bibliografía y recursos de interés para empezar

- Documentación oficial:
  - Quick Start: https://spark.apache.org/docs/latest/quick-start.html
  - Getting started Spark 3.0.1: https://spark.apache.org/docs/latest/sql-getting-started.html
- Databricks:
  - Notebooks con Spark en la nube, ideales para probar si tu portátil va a pedales: https://docs.databricks.com/getting-started/try-databricks.html