Fundamentos de Apache Spark

Competencias

- Conocer los casos de uso de Spark, así como las diferencias entre `MapReduce` y `Spark`.
- Conocer la arquitectura y componentes de `Spark`.
- Conocer los principales modos de trabajo con `Spark`.
- Habilitar notebooks de Jupyter desde la instancia AWS EMR.
- Implementar consultas en `Spark`.
- Utilizar transformaciones y acciones en RDD para extraer resultados.

Motivación

Caveat: *La motivación sigue los puntos expuestos por Ryza, Laserson, Owens y Wills (2015), Advanced Analytics with Spark: Patterns for learning from Data at Scale.*

Una de las maneras más fáciles de entender las ventajas producidas por Spark es cuando lo comparamos con el paradigma MapReduce. Este último ofrece un modelo simple para escribir programas que pudieran ejecutar tareas en paralelo en cientos o miles de máquinas.

El motor de MapReduce tiene un desempeño que se puede asumir de manera lineal. En la medida que tenemos más datos, simplemente podemos agregar más máquinas (escalamiento horizontal) para aumentar el desempeño. Este busca dividir el trabajo en partes pequeñas de manera tal de evitar las fallas en tareas pequeñas sin la necesidad de comprometer la estabilidad en la ejecución de trabajo.

Spark trabaja sobre el principio de escalabilidad lineal y tolerancia a los fallos del ecosistema Hadoop, pero agrega tres mejoras:

1. Reemplaza el formato MapReduce de ejecución a un Grafo Acíclico Dirigido (GAD). Estos buscan resolver el punto que por defecto MapReduce necesita generar resultados intermedios de todo paso de procesamiento. Mediante los GAD, los resultados son pasados directamente al siguiente elemento del pipeline de trabajo.

2. `Spark` está construido de manera nativa en `scala`, un lenguaje que combina los paradigmas de programación orientada a objetos y programación funcional. En base a estos modos imperativos, `Spark` aumenta la expresividad del código por parte del usuario mediante una serie de transformaciones de datos y código que resultan ser más intuitivas.

3. Sobrepasa la capacidad de análisis del paradigma `MapReduce` mediante el procesamiento **en memoria** de los datos. Mediante la abstracción de Resilient Distributed Datasets (RDD), un usuario puede materializar cualquier paso dentro de un pipeline de trabajo hacia memoria. Esto reduce la cantidad de operaciones realizadas para una tarea específica.

Una de las principales características de Spark es la habilidad de mantener grandes conjuntos de datos en memoria **entre trabajos**. Esta característica otorga un desempeño superior a tareas similares, pero implementadas en MapReduce clásico. Por ejemplo, los algoritmos iterativos (donde una función se aplica a un conjunto de datos hasta la satisfacción de una condición de salida) y análisis interactivos (donde un usuario genera consultas ad-hoc en el conjunto de datos, se benefician de buena manera de Spark.

Actualmente Spark se está posicionando como una buena plataforma donde se pueden implementar herramientas analíticas. Presenta herramientas como módulos de Machine Learning (`MLlib`), procesamiento de grafos (`GraphX`, por ahora sólo en `scala`), procesamiento streaming (`Spark Streaming`) y SQL (con `Spark SQL`).

El modelo de programación de Spark

Para entender cómo funciona `Spark`, debemos tener en consideración que es la intersección entre dos abstracciones propuestas por el framework: **Almacenamiento** y **Ejecución**. `Spark` permite que cualquier paso intermedio en el procesamiento de datos sea almacenado en memoria para posterior uso. Un script de `Spark` generalmente contará de los siguientes pasos:

1. Definir un conjunto de transformaciones en datasets.

2. Invocar acciones que transforman los datos a almacenaje persistente o retorno a la memoria del local.

3. Correr computaciones locales que operan en los resultados en una manera distribuida
   .

Nuestra primera incursión en Spark

Ahora generaremos nuestra primera interacción con Spark. Cabe destacar que este primer ejemplo se implementa en nuestro local. Para efectos prácticos del curso, trabajaremos mediante la API `pyspark` que permite comunicar nuestro código en Python con `Spark`, reduciendo el leverage de tener que aprender Scala (aún cuando esto puede ser un buen elemento en su desarrollo profesional).

Las rutinas y flujos de trabajo en Spark se conocen como **Aplicaciones**, las cuales ejecutan una serie de operaciones paralelas en un cluster. Cada aplicación se puede entender como una serie de componentes orientados a la ejecución eficiente del trabajo. Estos componentes son agnósticos a la máquina con la cual estamos trabajando, por lo que no habrá diferencia entre nuestro trabajo en local o en instancias AWS EMR como lo haremos posteriormente.

Toda aplicación necesita declarar objeto de configuración y un objeto de contexto. Para generarlos desde `pyspark`, debemos importar `from pyspark import SparkConf, SparkContext`. Con la primera clase importada, `SparkConf` vamos a configurar los elementos de conexión. Para este caso estaremos trabajando con un `master` que tendrá 4 `workers` en local, definido por `setMaster("local[4]")`. Posteriormente trabajaremos con Spark en el contexto de clusters, para lo cual deberemos cambiar la línea `setMaster` a `setMaster("yarn")`. Para identificar las tareas, es bueno asignar de manera explícita el nombre de la aplicación, en este caso, `setAppName('spark-101-adl')`.

```
from pyspark import SparkConf, SparkContext

spark_conf = SparkConf()\
              .setMaster("local[4]")\
              .setAppName('spark-101-adl')
spark_conf
```

```
<pyspark.conf.SparkConf at 0x10c2f1978>
```

Este objeto que contiene la configuración específica lo vamos a integrar con la clase `SparkContext`, la cual nos generará un objeto que establece la comunicación con Spark. La convención indica que el nombre del objeto `SparkContext` va a ser `sc`. Si imprimimos su contenido, veremos que reporta los parámetros configurados en nuestro objeto `spark_conf`.

```
sc = SparkContext(conf=spark_conf)
sc
```

```
  <div>
      <p><b>SparkContext</b></p>

      <p><a href="http://192.168.2.37:4040">Spark UI</a></p>

      <dl>
        <dt>Version</dt>
          <dd><code>v2.4.3</code></dd>
        <dt>Master</dt>
          <dd><code>local[4]</code></dd>
        <dt>AppName</dt>
          <dd><code>spark-101-adl</code></dd>
      </dl>
  </div>
```

Un aspecto importante es considerar que debe existir **solo una instancia de la clase** `SparkContext`. Esto se debe a que Spark está diseñado con la lógica que un usuario implementa su flujo de trabajo en una conexión exclusiva. En caso de que intentemos levantar una segunda instancia en el mismo kernel, `Spark` arrojará un error del tipo `ValueError: Cannot run multiple SparkContexts at once`. Si es necesario, se puede finalizar la conexión con `Spark` utilizando `stop()` en el objeto instanciado con `SparkContext`.

El objeto `sc` tendrá una serie de métodos para ingresar datos en múltiples formatos. Algunos de ellos permiten ingresar datos desde `HDFS`, `AWS S3` y desde el local. También permite realizar la lectura batch de múltiples elementos contenidos en una carpeta, archivos de carácter secuencial y binarios.

Para reducir la complejidad de nuestro primer ejemplo, vamos a ingresar una lista de 10000 números generados con el método `sc.parallelize`. Este método generará un objeto del tipo `PythonRDD`, que representa la principal abstracción generada en Spark: **Resilient Distributed Dataset (RDD)**. Un RDD se genera a partir de un conjunto de datos ingresados.

```
lista_parallel = sc.parallelize(range(1000000))
lista_parallel
```

```
PythonRDD[1] at RDD at PythonRDD.scala:53
```

Todo objeto del tipo RDD tendrá dos comportamientos:

- **Transformaciones:** Comportamiento que permite generar un nuevo conjunto de datos a partir de uno ya existente.

- **Acciones:** Comportamiento donde se devuelve el valor de un conjunto de datos al usuario después de ejecutar una acción.

Posteriormente profundizaremos sobre estos comportamientos, pero por ahora mantengámonos en lo instrumental para esta exposición. Resulta que ya tenemos conocimiento sobre las principales operaciones existentes en un `RDD`: `map` y `filter`. Estas funcionan de la misma manera a como lo hace en Python nativo: necesitamos pasar una función anónima declarada en un `lambda`. Por ahora generemos tres transformaciones:

- `transform_1`: Elevará al cuadrado cada elemento dentro de nuestra lista.
- `transform_2`: Filtrará cada elemento divisible por 4 dentro del RDD `transform_1`.
- `transform_3`: Filtrará cada elemento menor a 50 dentro del RDD `transform_2`.

```
transform_1 = lista_parallel.map(lambda x: x ** 2)
transform_2 = transform_1.filter(lambda x: x % 4 == 0)
transform_3 = transform_2.filter(lambda x: x < 50)
```

```
transform_3.collect()
```

```
[0, 4, 16, 36]
```

Ya tenemos nuestras primeras transformaciones realizadas en pyspark. Probablemente nos parecerá extraño que se demoren tan poco en ejecutarse. Este comportamiento se conoce como **lazy evaluation** (evaluación perezosa), dado que no se computan los resultados de manera automática.

Entonces, el comportamiento que tiene es que sólo se ejecutarán cuando una **acción** requiera de su resultado. Algunos de los ejemplos de acciones que estudiaremos posteriormente son:

- Extraer las primeras tres observaciones de `transform_3` con el método `take`.

```
transform_3.take(3)
```

```
[0, 4, 16]
```

- Tomar una muestra aleatoria de los resultados contenidos en `transform_1` con `takeSample`.

```
transform_1.takeSample(withReplacement=False, num=3)
```

```
[186757079716, 185227805161, 9612822025]
```

- Contar la cantidad de elementos alojados en `transform_2` con el método `count`.

```
transform_2.count()
```

```
500000
```

¿Qué fue lo que pasó? Una visión más acabada del funcionamiento de Spark

Caveat: *Esta sección es un resumen orientado al funcionamiento grosso-modo. Aquellos que deseen profundizar sobre el comportamiento de Spark, pueden referirse a Zaharia, M; Chowdhury, M; Franklin, M; Shenker, S; Stoica, I. 2011. Spark: Cluster Computing with Working Sets.*

Spark hace uso de una arquitectura basada en la dinámica **Master/Worker** con un coordinador central y varios workers distribuidos. El coordinador central se denomina driver y es el encargado de comunicarse con un número determinado de **workers**. Los workers funcionarán como una entidad con una asignación limitada de recursos en cuanto a CPU, Memoria. Dentro de los workers encontramos **ejecutores**, que son instancias específicas encargadas de generar los cálculos para un conjunto de datos, los cuales también preservan los resultados de las operaciones en memoria.

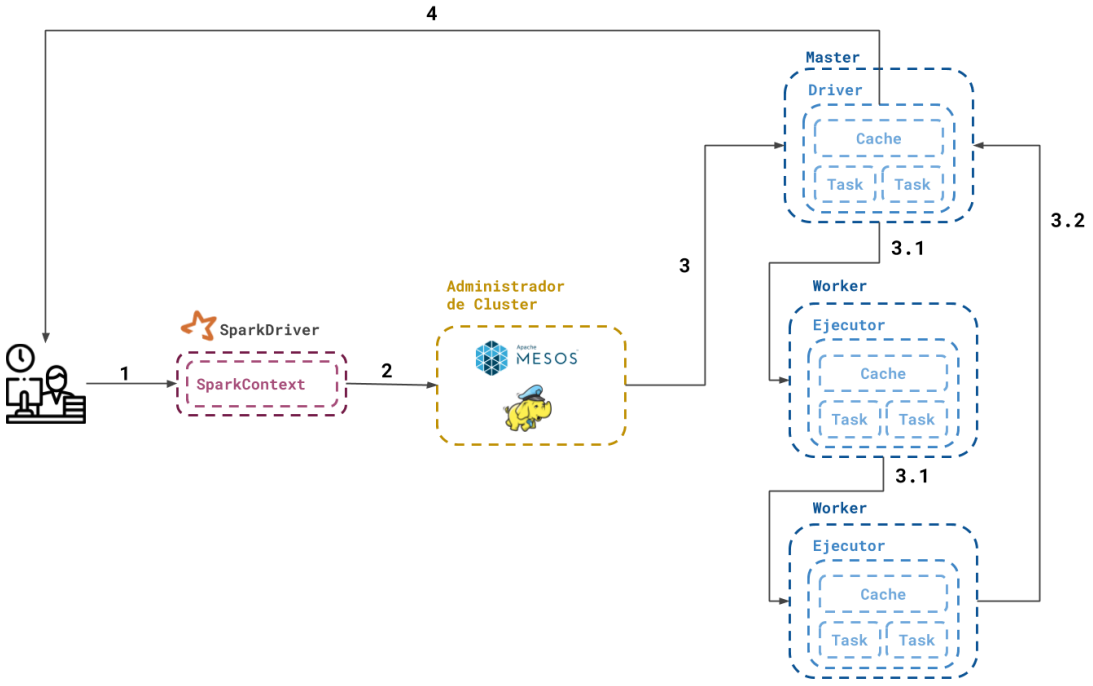

Imagen 1. Operaciones en memoria.

1. El cliente envía una aplicación de Spark al driver, donde se generan la configuración y contexto para su posterior aplicación. El objeto `SparkContext` es la abstracción que encapsula el cluster para el Driver, y por el cual se comunica toda la infraestructura de Spark.

2. Con la configuración establecida, se implementa una agenda de trabajo en algún administrador de recursos como `YARN` o `mesos`. Por defecto trabajaremos con `YARN` cuando estemos operando desde AWS EMR.

3. El ResourceManager asigna un ApplicationMaster para coordinar las tareas a nivel de workers.
    1. Los nodos workers se comunican con el master para indicar su capacidad de absorber nuevas tareas. Entre los workers se comparte la memoria caché de las representaciones de los datos en las RDD.
    2. El proceso es informado hacia el master.

4. El proceso se notifica al usuario

Por lo general, si implementamos transformaciones y/o acciones dentro de nuestra aplicación `Spark`, éste nos retornará un RDD. En la siguiente sección definiremos más detalles sobre este objeto.

Resilient Distributed Dataset

Un Dataset Resiliente Distribuido (Resilient Distributed Dataset o RDD de aquí en adelante) es el objeto fundamental utilizado en `Spark`. En un nivel alto, cada aplicación generada en `Spark` consiste en un programa controlador que ejecuta una función principal del usuario y ejecuta varias operaciones en paralelas dentro de un cluster. El RDD es una colección de elementos particionados en los nodos del cluster que se pueden operar en paralelo.

La mayoría de las operaciones de `Spark` cargan un RDD con datos externos y posteriormente se generan nuevos RDD a partir del original. Los RDD se crean comenzando con un archivo en HDFS, AWS S3, archivos de texto plano con diversas extensión o mediante otras colecciones de objetos generadas en `scala`. Si bien existen variadas opciones para generar persistencia de los RDD en el disco físico, el principal modo de operación es mediante el almacenamiento en memoria.

El término *Resilient Distributed Dataset* se puede entender como:

- **Resilientes:** Si un nodo ejecutando una acción se pierde por falla, se puede reconstruir. Esto se debe a que `spark` conoce el linaje del RDD perdido, así como su construcción mediante el GAD.

- **Distribuido:** Los datos dentro de un RDD se pueden dividir en varias particiones y distribuir como colecciones dentro de la memoria de los **workers**.

- **Dataset:** Los RDDs son conjuntos de registros que se pueden identificar dentro del dataset. Cada partición de registros distribuidos contiene un conjunto único de registros que se pueden operar de forma independiente.

Transformaciones, y Acciones dentro de un RDD

Recordemos de la introducción que las **transformaciones** en Spark son funciones que operan en un RDD y devuelven otro RDD, mientras que las **acciones** operan sobre un RDD y retornan un valor o un output.

Existen tres formas de crear un RDD:

1. Desde una colección de objetos **dentro** de la memoria (conocido como paralelización de una colección de datos). Esta forma es útil cuando estamos desarrollando prototipos de soluciones en conjuntos pequeños de datos.

2. Mediante el consumo de datos en un conjunto externo (como HDFS, AWS S3, etc...).

3. Mediante la transformación de una RDD existente.

Posteriormente veremos la amplia gama de RDDs existentes, así como sus transformaciones y acciones correspondientes. Antes de abordar este punto, es necesario generar una serie de definiciones sobre las características de las RDD.

Fact 1: Las transformaciones de un RDD se pueden considerar Gruesas (Coarse-Grained) o Finas (Fine-Grained)

Las operaciones en un RDD se consideran **Gruesas** cuando se implementa una función sobre **todos los elementos** en un RDD y retornan un nuevo RDD.

Las operaciones se pueden considerar como **Finas** cuando se manipulan un registro específico, como actualizaciones en un registro de una tabla SQL.

Fact 2: Lazy evaluation

Spark utiliza lazy evaluation para procesar expresiones. Una evaluación lazy evita el procesamiento **hasta** el momento en que la acción se llama. La acción se llama sólo cuando el output es necesario. Si concatenamos muchos RDDs sin procesar, cada declaración se concatena a la sintaxis y referencia de objetos. Cuando se implementa una acción, un GAD se crea en conjunto con los planes lógicos y físicos de ejecución. El Driver orquesta la ejecución.

La evaluación lazy permite a Spark combinar operaciones en medida de lo posible, reduciendo etapas de procesamiento y minimizando la cantidad de datos transferidas entre ejecutores en la fase de shuffle.

Fact 3: Los RDD pueden ser Persistentes y reutilizables

Los RDD se crean y existen predominantemente en memoria de los Ejecutores. Por defecto son objetos transientes que existen sólo cuando son llamados. Posterior a que se les implementa una operación del tipo transform en un nuevo RDD, y no se necesitan para otra operación, son eliminados de forma permanente.

Esto es problemático si es que un RDD se utiliza para más de una acción dado que deberá ser reevaluada cada vez. Una forma de solucionar esto es mediante la persistencia en memoria cache con `persists`.

Fact 4: Los RDD presentan un linaje

En un alto nivel, Spark lleva un registro del **linaje** de cada RDD: la secuencia de transformaciones que resulta en un RDD específico. Por defecto cada RDD recalcula el linaje, salvo que se sugiera persistencia de éste. Cada RDD tiene un RDD originario (en la literatura se conoce como **parent**) y dependiendo de su fase de ejecución, puede tener un RDD descendiente (en la literatura se conoce como **child**). Para ejemplificar el concepto de linaje, consideremos la figura que representa nuestro primer ejemplo de la lectura.

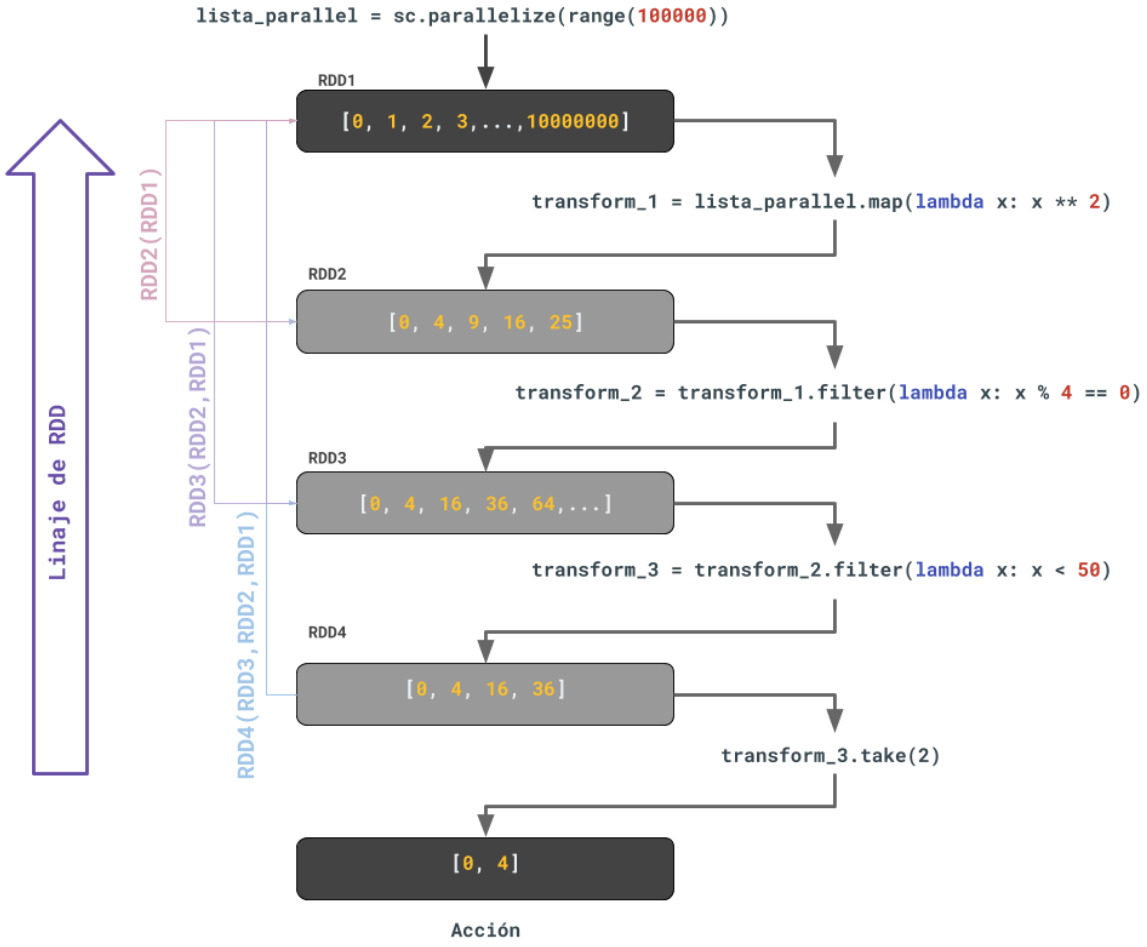

Imagen 2. Linaje de RDD..

Si leemos la imágen de manera lineal, el objetivo del procedimiento es elevar al cuadrado cada elemento dentro de nuestro RDD original (`lista_parallel`). Con el resultado de `transform_1`, procedemos a preservar todos los valores que sean divisibles por 4 en `transform_2`. La última transformación (`transform_3`) conlleva seleccionar sólo los elementos menores a 50. Finalmente, **materializamos** nuestro código mediante la acción `take`.

Dado que por defecto la transformación no se implementa hasta ser invocada mediante una acción, la implementación de las acciones respondía solo a la actualización de nuestro Grafo Acíclico Dirigido. Cuando ejecutamos la acción `transform_3.take(3)`, informamos a nuestro RDD que deberá ejecutar todas las transformaciones que le preceden.

Fact 5: Los RDD son tolerantes a fallas

Dado que se registra el linaje de cada RDD, cualquier RDD con sus particiones pertinentes puede ser reconstituido al estado antes de fallo, el cual pudo haber resultado de node failure o eliminación del `DataNode` en HDFS por ejemplo.

Preliminar: Configuración una instancia AWS EMR con Spark

Antes de comenzar con el ejercicio práctico, sigan estos pasos para poder habilitar PySpark dentro de un Jupyter Notebook que estará alojado en su instancia de trabajo. Para generar una instancia de trabajo en AWS EMR con Spark, debemos generar una configuración avanzada. Para ello, debemos ir a la sección Advanced configuration de la creación de clusters en AWS Management Services y seleccionar los siguientes componentes:

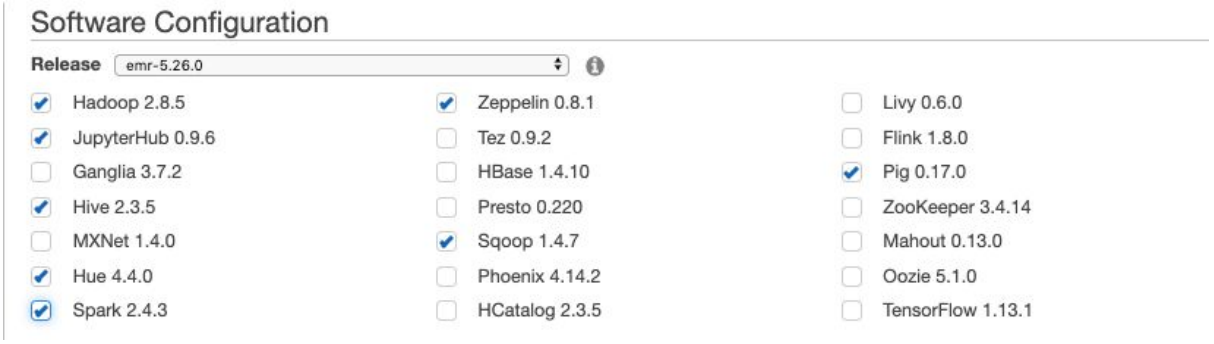

Imagen 3.Configuración de Software.

Después de crear la instancia de trabajo específica, seguimos con los pasos usuales de configuración de la instancia.

Con nuestra instancia creada y corriendo, generamos las conexiones para interactuar con la instancia y la conexión para habilitar un puerto dinámico y poder utilizar las interfaces gráficas. Con las conexiones habilitadas, tendremos acceso a todos los links de la instancia.

Imagen 4. Habilitando conexiones.

Para esta ocasión, utilizaremos `JupyterHub` que opera de la misma manera que un servidor de Jupyter de manera local. Si ingresamos al link tendremos un aviso que nuestra conexión no es privada. Para poder utilizar `JupyterHub` deberemos ir a **opciones avanzadas** y hacer click en Proceed to *<ip de la instancia>*.

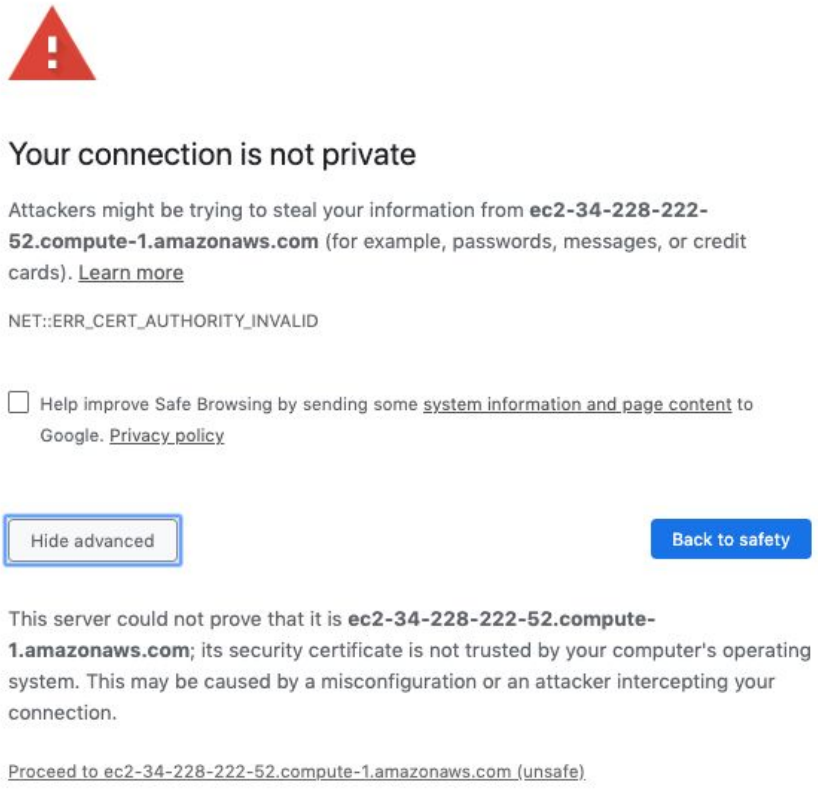

Imagen 5. Aviso de conexión no privada.

Posterior a este paso, deberemos ingresar con el usuario por defecto `jovyan` y su contraseña `jupyter`. Si este endpoint va a ser utilizado más de una vez, se recomienda cambiar su contraseña en el panel de administración de JupyterHub.

Una vez dentro de nuestro tree de trabajo en JupyterHub, crearemos un nuevo notebook con el kernel `PySpark3`, el cual vendrá con un objeto `SparkContext` ya creado. Una de las virtudes generadas por este Notebook creado es que la configuración creada con `SparkConf` incluirá de manera automática `YARN` como master, haciendo uso distribuido de los clusters existentes.

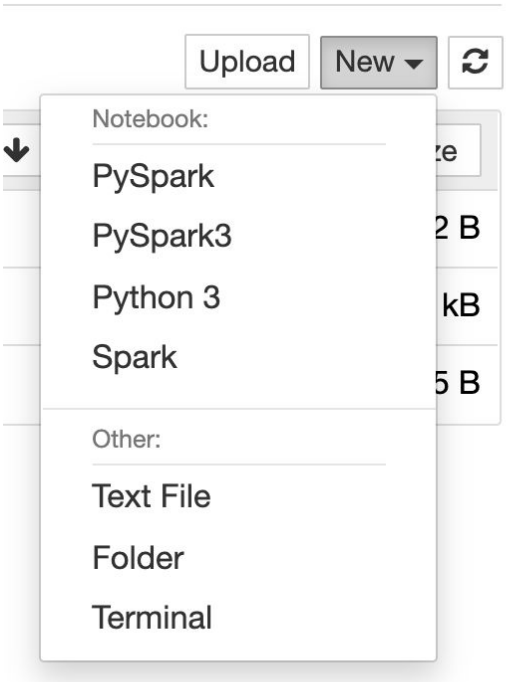

Imagen 6. Lista de Notebook.

Práctico: Evaluando el uso de bicicletas públicas en el Bay Area

Para este ejercicio trabajaremos con datos del sistema de bicicletas públicas del Bay Area (conurbación compuesta por San Francisco, Mountain View, Redwood City, San José y Palo Alto). Pueden encontrar más información sobre el sistema aquí. Para efectos prácticos de la lectura, los datos se encuentran en el bucket S3 del curso en la dirección `s3://bigdata-desafio/lecture-replication-data/bike-share/`. Dentro de esta carpeta, haremos uso de los `csv` **Stations** y **Status**. A continuación se entrega una breve referencia sobre los campos:

Bike station data

Cada registro del `csv` corresponde a una estación donde un usuario puede retirar o estacionar una bicicleta del programa. Los campos de cada registro se especifican a continuación:

Posición en Lista	Columna	Descripción	Tipo de Dato
0	station_id	Id de la estación	int
1	name	Nombre de la estación	str
2	lat	latitud	float
3	long	longitud	float
4	dockstation	Cantidad de docks disponibles	int
5	landmark	Ciudad	int
6	Installation	Fecha de inauguración	str

Tabla 1. Registro estación de bicicletas.

Bike status data

Cada registro del csv corresponde a la cantidad de bicicletas y docks disponibles en una estación en una hora específica. Los campos de cada registro se especifican a continuación:

Posición en Lista	Columna	Descripción	Tipo de Dato
0	station_id	Número de id	int
1	bikes_available	Cantidad de bicicletas disponibles	int
2	docks_available	Cantidad de docks disponibles	int
3	time	Fecha y Hora	str

Tabla 2. Registro de bicicletas.

Desde el San Francisco Metropolitan Transit Authority (SFMTA) le solicitan en su capacidad de analista de datos que responda las siguientes preguntas:

- ¿Cuál es la cantidad de estaciones existentes en el Bay Area?
- Acorde al conjunto de datos entregados, ¿Cuál es la cantidad de viajes registrados por ciudad?
- ¿A qué hora y dónde es más probable encontrar bicicletas disponibles?

Nos aseguramos que el kernel esté corriendo con `sc`.

Paso 1: Carga de archivos y generación de RDD

El primer objeto RDD que crearemos siempre será la carga de un archivo para tener su representación en RDD. El objeto `SparkContext` creado permitirá la carga de archivos de distinto tipo. Para efectos prácticos, nos concentramos en la versión más simple, que es la carga de un archivo de texto plano. Para ello utilizaremos el método `textFile`. La sintaxis es:

```
data = sc.textFile('archivo')
```

Donde `'archivo'` puede incluir distintos esquemas de sistemas de archivo y URI. A continuación se detallan las formas de conexión.

Sistema de Archivos	Estructura URI
Sistema de Archivo Local	`file:///ruta/absoluta/o/relativa/`
Hadoop Distributed File System (HDFS)	`hdfs://ruta/absoluta/hdfs`
Amazon S3	`s3://path/del/bucket/`

Tabla 3. Formas de conexión.

Para este ejemplo cargaremos `stations.csv` y `status.csv` que están alojados en el bucket. Si nos encontramos en una situación donde tenemos múltiples archivos con la misma estructura alojados dentro de un bucket, podremos realizar la carga con `sc.wholeTextFiles`.

Cabe destacar que este ejemplo lo estamos realizando dentro de un Notebook de JupyterHub habilitado en nuestra instancia AWS EMR. Partamos por habilitar nuestro contexto de Spark con `sc` el cual se encuentra instanciado en el kernel `PySpark3`.

```
sc
```

```
<SparkContext master=yarn appName=livy-session-2>
```

Ahora ingresaremos los `csv` especificados anteriormente:

```
bike_station_data =
sc.textFile('s3://bigdata-desafio/lecture-replication-data/bike-share/stations.csv')
bike_status_data =
sc.textFile('s3://bigdata-desafio/lecture-replication-data/bike-share/status.csv')
```

Los objetos `bike_station_data` y `bike_status_data` resultan ser RDD. Si ejecutamos la acción `take(1)` en estos, obtendremos la representación cruda de los datos. Para ambos casos, resultan ser un registro que está en formato string.

```
bike_station_data.take(1)
```

```
[u'2,San Jose Diridon Caltrain Station,37.329732,-121.901782,27,San Jose,8/6/2013']
```

```
bike_status_data.take(1)
```

```
[u'10,9,6,"2015-02-28 23:59:01"']
```

Paso 2: Preparación de los datos a partir del RDD

Posterior a la ingesta de los datos, nuestro siguiente elemento es transformar este string en un formato trabajable para Spark. Lo que debemos lograr es una lista de tuplas, donde las tuplas representarán a un registro, y los elementos dentro de la tupla corresponderá a los datos señalados en la descripción de las tablas. Partamos por ordenar la información de estaciones:

```
bike_station_proc = bike_station_data\
                        .map(lambda x: x.split(','))\
                        .map(lambda x: (int(x[0]), str(x[1]),
float(x[2]),
                                        float(x[3]), float(x[4]),
str(x[5]), str(x[6])))
```

El retorno de nuestra expresión será un `PipelinedRDD`, que representa una sucesión de transformaciones en un RDD. Si extraemos el primer elemento procesado, obtendremos nuestra lista de tuplas.

```
type(bike_station_proc)
```

```
<class 'pyspark.rdd.PipelinedRDD'>
```

```
bike_station_proc.take(1)
```

```
[(2, 'San Jose Diridon Caltrain Station', 37.329732, -121.901782, 27.0,
'San Jose', '8/6/2013')]
```

Para sintetizar el código descrito arriba, se presenta la siguiente imagen:

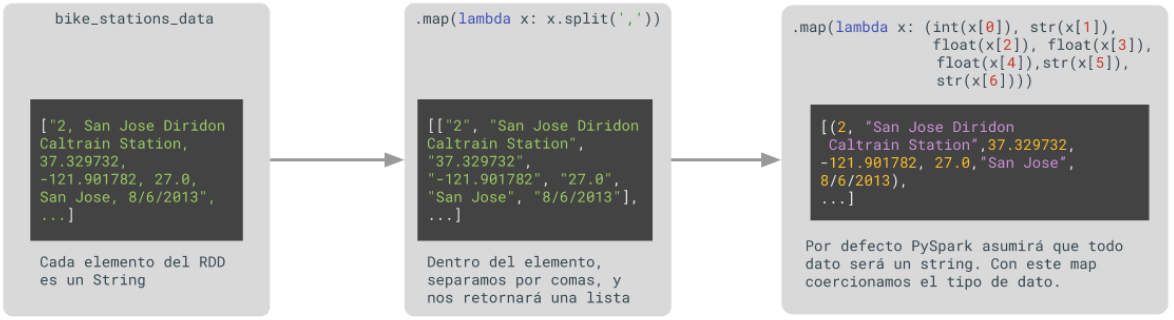

Imagen 7. Preparación de los datos.

Ahora vamos a implementar un procedimiento similar para el objeto `bike_status_data`. Uno de nuestros objetivos va ser dividir la fecha y la hora registrada, eliminando sus separadores. Debemos partir por separar el string, y posterior a esto, vamos a normalizar las comillas dobles ocurrentes en el texto por comillas simples.

El último registro tiene la estructura `"2015-02-28 23:59:01"`, por lo que debemos generar los siguientes pasos. Partimos por separar la fecha de la hora, la cual está indicada por un espacio en blanco. Después podremos separar los elementos de fecha y hora con sus separadores respectivos (`-` para fechas y `:` para hora).

```
bike_status_proc = bike_status_data\
                    .map(lambda x: x.split(','))\
                    .map(lambda x: (x[0], x[1], x[2], x[3].replace('"',
''')))\
                    .map(lambda x: (x[0], x[1], x[2], x[3].split(' ')))\
                    .map(lambda x: (x[0], x[1], x[2],
x[3][0].split('-'), x[3][1].split(':')))\
                    .map(lambda x: (int(x[0]), int(x[1]), int(x[2]),
int(x[3][0]), int(x[3][1]), int(x[3][2]), int(x[4][0])))
```

```
bike_status_proc.take(1)
```

```
[(10, 9, 6, 2015, 2, 28, 23)]
```

Consulta 1: Cantidad de estaciones por ciudad dentro del Bay Area

Nuestra primera consulta es contar la cantidad de registros asociados a cada ciudad en la tabla `bike_station_proc`. Si nos fijamos bien en el requerimiento, observaremos que es el problema clásico de MapReduce. Resulta que la implementación en Spark de este problema es bastante simple y mesurado en código. Para efectos prácticos, el código se representa en la siguiente imágen:

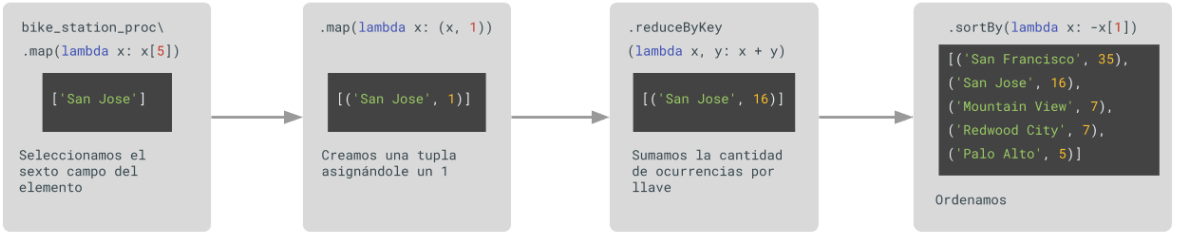

Imagen 8. Implementación en `Spark`.

```
get_stations_by_city = bike_station_proc\
                       .map(lambda x: x[5])\
                       .map(lambda x: (x, 1))\
                       .reduceByKey(lambda x, y: x + y)\
                       .sortBy(ascending=False)
get_stations_num = get_stations_by_city.collect()
```

Así, observamos que la ciudad de San Francisco tiene 35 estaciones habilitadas, seguidas por San José, Mountain View, Redwood City y Palo Alto.

```
print(get_stations_num)
```

```
[('San Francisco', 35), ('San Jose', 16), ('Mountain View', 7),
('Redwood City', 7), ('Palo Alto', 5)]
```

Consulta 2: Cantidad de viajes registrados por ciudad

La siguiente consulta encargada es ver la cantidad de viajes realizados por ciudad. Para ello vamos a tener que identificar cuales son las id asociadas a cada región. Partamos por solucionar cómo obtendríamos las ID para San Francisco.

```
# utilizando el RDD de estaciones
get_frisco_stations_id = bike_station_proc\
                         # vamos a filtrar sólo aquellas
correspondientes a San Fco.
                         .filter(lambda x: x[5] == 'San Francisco')\
                         # y vamos a extraer su nombre y Id
                         .map(lambda x: (x[5], x[0]))\
                         # con collect vamos a retornar los
resultados a una lista.
                         .collect()
print(get_frisco_stations_id)
```

```
[('San Francisco', 41), ('San Francisco', 42), ('San Francisco', 45),
('San Francisco', 46), ('San Francisco', 47), ('San Francisco', 48),
('San Francisco', 49), ('San Francisco', 50), ('San Francisco', 51),
('San Francisco', 39), ('San Francisco', 54), ('San Francisco', 55),
('San Francisco', 56), ('San Francisco', 57), ('San Francisco', 58),
('San Francisco', 59), ('San Francisco', 60), ('San Francisco', 61),
('San Francisco', 62), ('San Francisco', 63), ('San Francisco', 64),
('San Francisco', 65), ('San Francisco', 66), ('San Francisco', 67),
('San Francisco', 68), ('San Francisco', 69), ('San Francisco', 70),
('San Francisco', 71), ('San Francisco', 72), ('San Francisco', 73),
('San Francisco', 74), ('San Francisco', 75), ('San Francisco', 76),
('San Francisco', 77), ('San Francisco', 82)]
```

Ya sabemos que el rango de valores va entre el 41 y 82. Para filtrarlos de una mejor manera, vamos a crear la función `get_frisco_id` que operará a nivel de elemento y retornará el elemento si es que está en el rango.

```
# generamos la función
def get_frisco_id(x):
    if x >= 41 and x <= 82:
        return x

# vamos a preservar todos los datos que satisfagan la condición descrita
get_frisco_statuses = bike_status_proc\
                        .filter(lambda x: get_frisco_id(x[0]))
```

Utilizaremos la acción `count` para contar todos los elementos resultantes. Observamos que en los datos disponibles hay 453600 viajes realizados.

```
num = get_frisco_statuses.count()
print("Cantidad de viajes registrados en San Francisco: {}".format(num))
```

```
Cantidad de viajes registrados en San Francisco: 453600
```

Podríamos replicar el código para cada una de las ciudades restantes, pero optaremos por una solución más escalable. El código buscará separar la ciudad y el ID, agrupar los datos por ID y devolver un arreglo.

```
# utilizando el RDD de las estaciones
get_stations_range = bike_station_proc\
                        # separamos el nombre de la ciudad y el id
                        .map(lambda x: (x[5], x[0]))\
                        # agrupamos por el nombre de la ciudad (la llave
del map anterior)
                        .groupByKey()\
                        # vamos a retornar el nombre y la lista de todas
las ID
                        .map(lambda x: (x[0], list(x[1])))
```

Ahora nos aseguraremos que los datos devueltos hagan sentido con el siguiente loop.

```
print("Station Name | N stations | Station Id")
# para cada elemento registrado
for station in get_stations_range.collect():
    # imprimir el nombre de la estación, la cantidad de estaciones y la
lista de estaciones.
    print(station[0], len(station[1]), station[1])
```

```
Station Name | N stations | Station Id
('Palo Alto', 5, [34, 35, 36, 37, 38])
('Mountain View', 7, [27, 28, 29, 30, 31, 32, 33])
('San Jose', 16, [2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 16, 80,
84])
('Redwood City', 7, [21, 22, 23, 24, 25, 26, 83])
('San Francisco', 35, [41, 42, 45, 46, 47, 48, 49, 50, 51, 39, 54, 55,
56, 57, 58, 59, 60, 61, 62, 63, 64, 65, 66, 67, 68, 69, 70, 71, 72, 73,
74, 75, 76, 77, 82])
```

Ahora generaremos un loop para filtrar por ID de cada ciudad y contar la cantidad de viajes en el RDD de status.

```
for station in get_stations_range.collect():
    tmp_spark_query = bike_status_proc\
                        .filter(lambda x: x[0] in station[1])
    print("Cantidad de viajes registrados en {}: {}".format(station[0],
tmp_spark_query.count()))
```

```
Cantidad de viajes registrados en Palo Alto: 64800
Cantidad de viajes registrados en Mountain View: 90720
Cantidad de viajes registrados en San Jose: 207360
Cantidad de viajes registrados en Redwood City: 90720
Cantidad de viajes registrados en San Francisco: 453600
```

Consulta 3: Cantidad de bicicletas disponibles por hora y estación

Para la última consulta realizada, vamos a generar un cruce entre los elementos contenidos en los RDD `bike_station_proc` y `bike_status_proc`. Para ello debemos reorganizarlos por algún elemento en común. Resulta que éste elemento específico es el `id` de la estación. Utilizaremos la transformación `keyBy` donde indicaremos la posición del ID.

Posterior a eso, vamos a asociar cada registro del RDD `bike_status_proc` con la información asociada a su ID proviniente de `bike_station_proc`. Posterior a la creación de este join, vamos a extraer la ID, la cantidad de bicicletas disponibles, la hora de medición y el nombre de la estación.

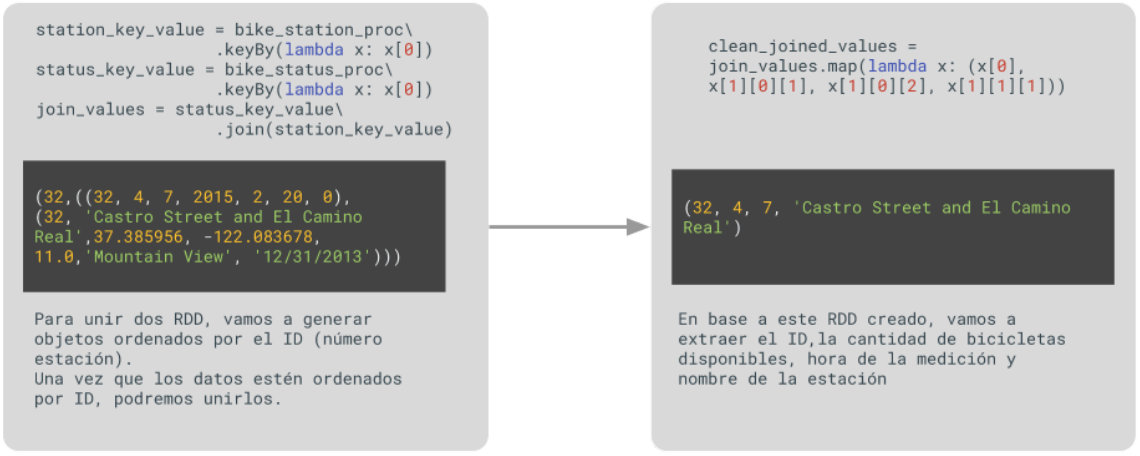

Imagen 9. Consulta, cantidad de bicicletas por hora y estación.

```
station_key_value = bike_station_proc.keyBy(lambda x: x[0])
status_key_value = bike_status_proc.keyBy(lambda x: x[0])
```

```
join_values = status_key_value.join(station_key_value)
```

```
clean_joined_values = join_values.map(lambda x: (x[0], x[1][0][1],
x[1][0][2], x[1][1][1]))
```

```
clean_joined_values.take(2)
```

```
[(32, 4, 7, 'Castro Street and El Camino Real'), (32, 4, 7, 'Castro
Street and El Camino Real')]
```

Con los datos necesarios ya listos para procesar, ahora procederemos a implementar nuestra consulta en Spark. A grandes rasgos, el proceso se puede resumir en los siguientes pasos:

1. Agrupamos los datos por estación y hora con la instrucción `keyBy(lambda x: (x[3], x[2])`.

2. Para cada registro ordenado, vamos a asignar un 1 como identificador.

3. Vamos a seleccionar la cantidad de bicicletas y la cantidad de veces que se repite la llave.

4. Vamos a sumar la cantidad de bicicletas y cantidad de veces de repetición para cada llave. Dividiremos la cantidad de bicicletas por la cantidad de ocurrencias para obtener un promedio aproximado.

5. Vamos a ordenar los datos por orden descendente y extraemos los primeros diez.

El procedimiento se puede representar en la siguiente imagen:

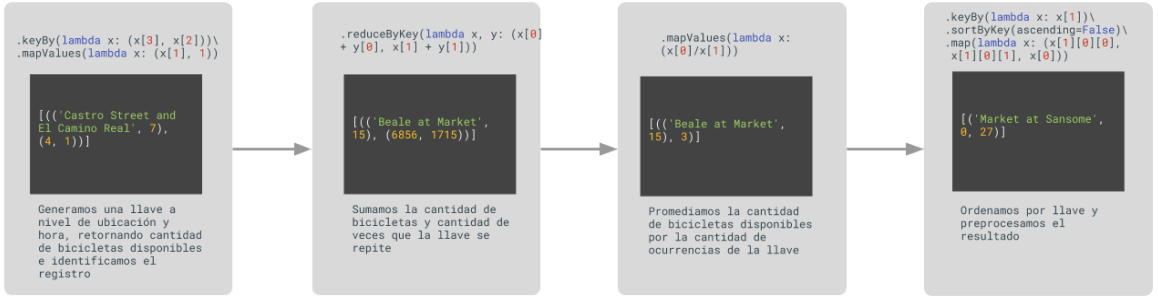

Imagen 10. Procedimiento para consulta Spark.

```
average_by_hour = clean_joined_values\
                    .keyBy(lambda x: (x[3], x[2]))\
                    .mapValues(lambda x: (x[1], 1))\
                    .reduceByKey(lambda x, y: (x[0] + y[0], x[1] +
y[1]))\
                    .mapValues(lambda x: (x[0]/x[1]))
```

```
average_by_hour\
    .keyBy(lambda x: x[1])\
    .sortByKey(ascending=False)\
    .map(lambda x: (x[1][0][0], x[1][0][1], x[0]))\
    .take(10)
```

```
[('Market at Sansome', 0, 27), ('Market at Sansome', 1, 26), ('2nd at
Townsend', 0, 26), ('2nd at Townsend', 1, 25), ('Market at 10th', 2,
25), ('Market at 10th', 3, 24), ('2nd at Townsend', 2, 24), ('Harry
Bridges Plaza (Ferry Building)', 0, 23), ('Market at Sansome', 3, 23),
('2nd at Townsend', 3, 23)]
```

Observamos que en promedio las principales estaciones con un alto número de bicicletas disponibles corresponde al siguiente escenario:

1. Estaciones ubicadas cerca de la Avenida Market en San Francisco.

2. La mayor disponibilidad se sitúa a la media noche.

Cierre: Diagramas sobre las formas de operar en RDDs

A lo largo de la lectura aprendimos sobre cómo funciona a grandes rasgos Spark, y conocimos el funcionamiento de su principal abstracción: el **Resilient Distributed Dataset**. El principio rector de los RDD es la noción de poder vectorizar toda instrucción a operaciones funcionales como `map` y `filter`. A continuación se presenta una serie de esquemáticas para facilitar el uso de las acciones y transformaciones en el contexto de RDD.

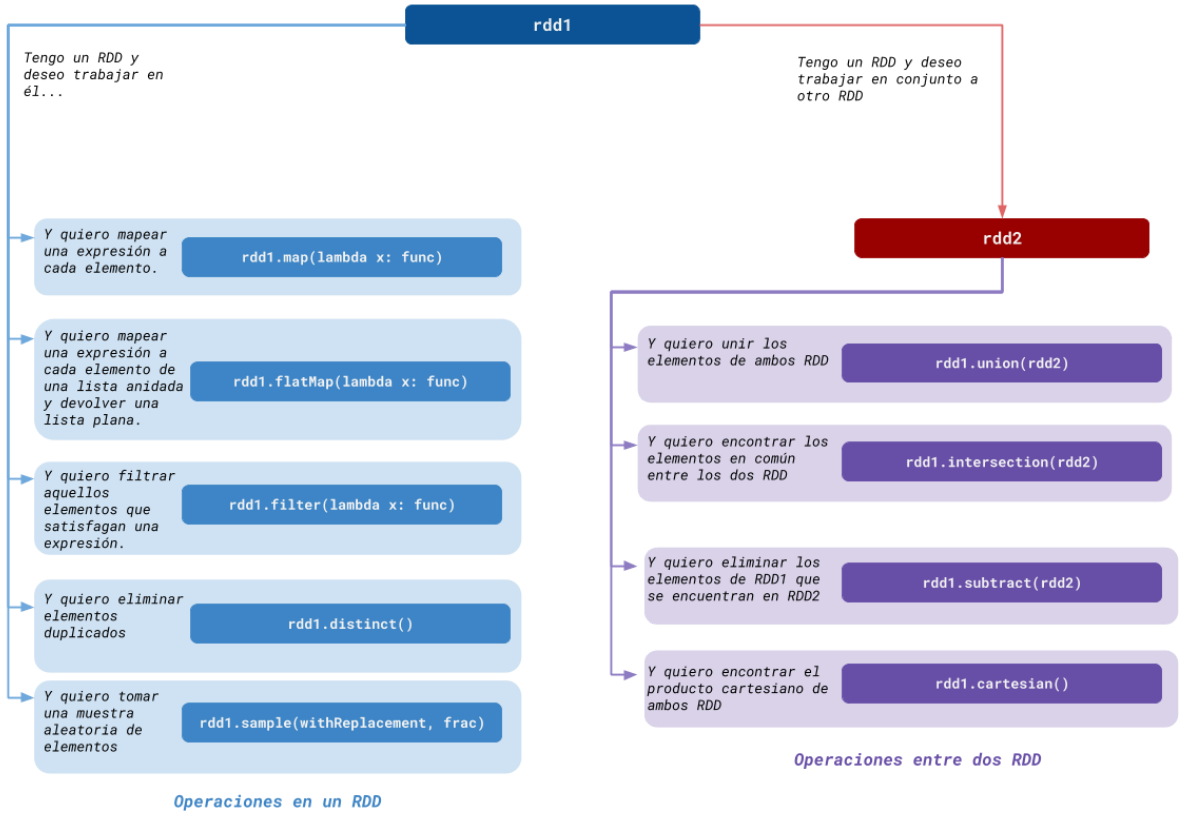

Imagen 11. Esquemáticas rdd.

Referencias

- Karau, H; Konwinski, A; Wendell, P; Zaharia, M. 2015. Learning Spark: Lighting-fast Data Analysis. Sebastopol, CA: O'Reilly Media Inc.

- Ryza, S; Laserson, U; Owen, S; Wills, J. 2015. Advanced Analytics with Spark: Patterns for Learning from Data at Scale. Sebastopol, CA: O'Reilly Media Inc.

- White, T. 2014. Hadoop The Definitive Guide: Storage and Analysis at Internet Scale. Sebastopol, CA: O'Reilly Media Inc.