

O'REILLY®

# Aprendizaje

## Chispa - chispear

Análisis de datos ultrarrápido



Saludos de



databricks

Jules S. Damji,  
Brooke Wenig,  
Tathagata Das y  
Denny Lee  
Prólogo de Matei Zaharia



## WEBINAR

# Making Apache Spark™ better with Delta Lake

[LEARN MORE](#)

Michael Armbrust,  
Principal Engineer

 **databricks**



## Elogio por Learning Spark, segunda edición

Este libro ofrece un enfoque estructurado para aprender Apache Spark, cubriendo nuevos desarrollos en el proyecto. Es una excelente manera para que los desarrolladores de Spark comiencen con Big Data.

—Reynold Xin, arquitecto jefe y cofundador de Databricks y miembro de Apache Spark PMC

Para los científicos e ingenieros de datos que buscan aprender Apache Spark y cómo crear aplicaciones de big data escalables y confiables, ¡este libro es una guía esencial!

—Ben Lorica, científico de datos jefe de Databricks, ex presidente del programa O'Reilly Strata Conferences, presidente del programa Spark + AI Summit



SEGUNDA EDICION

---

**chispa de aprendizaje  
Análisis de datos ultrarrápido**

**Jules S. Damji, Brooke  
Little, Tathagata Das y Denny Lee**

Pekín Boston Farnham Sebastopol Tokio

**O'REILLY®**

## Learning

**Spark** de Jules S. Damji, Brooke Wenig, Tathagata Das y Denny Lee

Copyright © 2020 Databricks, Inc. Todos los derechos reservados.

Impreso en los Estados Unidos de América.

Publicado por O'Reilly Media, Inc., 1005 Gravenstein Highway North, Sebastopol, CA 95472.

Los libros de O'Reilly se pueden comprar con fines educativos, comerciales o de promoción de ventas. Las ediciones en línea también están disponibles para la mayoría de los títulos (<http://oreilly.com>). Para obtener más información, comuníquese con nuestro departamento de ventas corporativo/institucional: 800-998-9938 o [corporate@oreilly.com](mailto:corporate@oreilly.com).

**Editor de adquisiciones:** Jonathan Hassell

**Indexador:** Potomac Indexing, LLC

**Editora de desarrollo:** Michele Cronin

**Diseñador de interiores:** David Futato

**Editora de producción:** Deborah Baker

**Diseño de portada:** Karen Montgomery

**Correctora:** Rachel Head

**Ilustrador:** Rebecca Demarest

**Correctora:** Penelope Perkins

Enero 2015: Primera edición

Julio 2020: Segunda edición

### Historial de revisiones de la segunda edición

2020-06-24: Primer lanzamiento

Ver <http://oreilly.com/catalog/errata.csp?isbn=9781492050049> para detalles de lanzamiento.

El logotipo de O'Reilly es una marca comercial registrada de O'Reilly Media, Inc. Learning Spark, la imagen de portada y la imagen comercial relacionada son marcas comerciales de O'Reilly Media, Inc.

Las opiniones expresadas en este trabajo son las de los autores y no representan las opiniones del editor.

Si bien el editor y los autores se han esforzado de buena fe para garantizar que la información y las instrucciones contenidas en este trabajo sean precisas, el editor y los autores renuncian a toda responsabilidad por errores u omisiones, incluida, entre otras, la responsabilidad por daños resultantes del uso o confianza en este trabajo. El uso de la información e instrucciones contenidas en este trabajo es bajo su propio riesgo. Si alguna muestra de código u otra tecnología que este trabajo contiene o describe está sujeta a licencias de código abierto o a los derechos de propiedad intelectual de otros, es su responsabilidad asegurarse de que su uso cumpla con dichas licencias y/o derechos.

Este trabajo es parte de una colaboración entre O'Reilly y Databricks. Consulte nuestra [declaración de independencia editorial](#).

978-1-492-05004-9

---

# Tabla de contenido

Prefacio.....	XIII
Prefacio.....	XV
<b>1. Introducción a Apache Spark: un motor de análisis unificado.....</b>	<b>1</b>
La génesis de la chispa	1
Big Data y computación distribuida en Google Hadoop en	1
Yahoo!	2
Los primeros años de Spark en	3
AMPLab ¿Qué es Apache Spark?	4
Velocidad	4
Facilidad de	5
uso Modularidad	5
Extensibilidad Análisis	6
unificado Componentes de Apache Spark como una pila	6
unificada Ejecución distribuida de Apache Spark La	10
experiencia del desarrollador ¿Quién usa Spark y para	14
qué?	14
Adopción y expansión de la comunidad	15
<b>2. Descarga de Apache Spark y primeros pasos.....</b>	<b>19</b>
Paso 1: Descarga de directorios y archivos	19
de Apache Spark	21
Paso 2: Uso de	21
Scala o PySpark Shell	22
Uso de la máquina local	22
Paso 3: Comprensión de los conceptos de la	23
aplicación Spark	25
Aplicación Spark y SparkSession	25
	26

Empleos de Spark	27
Etapas de chispa	28
Tareas de chispa	28
Transformaciones, acciones y evaluación perezosa	28
Transformaciones estrechas y anchas	30
La interfaz de usuario de Spark	31
Su primera aplicación independiente	34
Contando M&Ms para el Monstruo de las Galletas	35
Creación de aplicaciones independientes en Scala	40
Resumen	42
<b>3. API estructuradas de Apache Spark.....</b>	<b>43</b>
Spark: ¿Qué hay debajo de un RDD? 43	
Estructuración de Spark	44
Ventajas y ventajas clave La API	45
de DataFrame Tipos de datos	47
básicos de Spark Tipos de datos	48
estructurados y complejos de Spark Esquemas y creación	49
de DataFrames Columnas y expresiones Filas Operaciones	50
comunes de DataFrame Ejemplo de DataFrame de extremo	54
a extremo	57
58	
68	
La API del conjunto de datos	69
Objetos con tipo, objetos sin tipo y filas genéricas	69
Creación de conjuntos de datos	71
Operaciones de conjuntos de datos	72
Ejemplo de conjunto de datos de extremo a extremo	74
Marcos de datos frente a conjuntos de datos	74
Cuándo usar RDD	75
Spark SQL y el motor subyacente	76
El optimizador de catalizador	77
Resumen	82
<b>4. Spark SQL y DataFrames: Introducción a las fuentes de datos integradas.....</b>	<b>83</b>
Uso de Spark SQL en ejemplos de consultas básicas	84
de Spark Applications	85
Tablas y vistas de SQL	89
Tablas administradas versus no administradas	89
Creación de bases de datos y tablas SQL	90
Creación de vistas	91
Ver los metadatos	93

Almacenamiento en caché de tablas SQL	93
Lectura de tablas en marcos de datos	93
Orígenes de datos para marcos de datos y tablas SQL	94
Lector de tramas de datos	94
Escritor de tramas de datos	96
Parquet	97
JSON	100
CSV	102
Euro	104
ORCO	106
Imágenes	108
Archivos binarios	110
Resumen	111
<b>5. Spark SQL y DataFrames: interacción con fuentes de datos externas . . . . .</b>	<b>113</b>
Spark SQL y Apache Hive	113
Funciones definidas por el usuario	114
Consultas con Spark SQL Shell, Beeline y Tableau	119
Usando el shell SQL de Spark	119
Trabajando con Beeline	120
Trabajar con Tableau	122
Fuentes de datos externas	129
Bases de datos JDBC y SQL	129
postgresql	132
mysql	133
Azure Cosmos DB	134
Servidor MS SQL	136
Otras fuentes externas	137
Funciones de orden superior en DataFrames y Spark SQL	138
Opción 1: Explotar y recopilar	138
Opción 2: Función definida por el usuario	138
Funciones integradas para tipos de datos complejos	139
Funciones de orden superior	141
Operaciones comunes de DataFrames y Spark SQL	144
sindicatos	147
Uniones	148
ventanas	149
Modificaciones	151
Resumen	155
<b>6. Spark SQL y conjuntos de datos . . . . .</b>	<b>157</b>
API única para Java y Scala	157

Clases de casos de Scala y JavaBeans para conjuntos de datos	158
Trabajar con conjuntos de datos	160
Creación de datos de muestra	160
Transformación de datos de muestra	162
Gestión de memoria para conjuntos de datos y marcos de datos	167
Codificadores de conjuntos de datos	168
Formato interno de Spark versus formato de objeto Java	168
Serialización y Deserialización (SerDe)	169
Costos de usar conjuntos de datos	170
Estrategias para mitigar costos	170
Resumen	172
<b>7. Optimización y puesta a punto de aplicaciones Spark . . . . .</b>	<b>173</b>
Optimización y ajuste de Spark para la eficiencia	173
Visualización y establecimiento de configuraciones de Apache Spark	173
Escalamiento de Spark para grandes cargas de trabajo	177
Almacenamiento en caché y persistencia de	183
datos DataFrame.cache()	183
Marco de datos.persist()	184
Cuándo almacenar en caché y persistir	187
Cuándo no almacenar en caché y persistir	187
Una familia de Spark Joins Difundir hash Join	187
Shuffle Sort Merge Join Inspección de la	188
interfaz de usuario de Spark Viaje a través	189
de las pestañas de la interfaz de usuario de	197
Spark Resumen	197
	205
<b>8. Transmisión estructurada . . . . .</b>	<b>207</b>
Evolución del motor de procesamiento de Apache Spark Stream	207
El advenimiento del procesamiento de flujo de microlotes	208
Lecciones aprendidas de Spark Streaming (DStreams)	209
La filosofía de la transmisión estructurada	210
El modelo de programación de transmisión estructurada	211
Los fundamentos de una consulta de transmisión estructurada	213
Cinco pasos para definir una consulta de transmisión	213
Bajo el capó de una consulta de transmisión activa	219
Recuperación de fallas con garantías exactamente una vez	221
Supervisión de una consulta activa	223
Fuentes y sumideros de datos de transmisión	226
archivos	226
apache kafka	228

Fuentes y sumideros de transmisión personalizados	230
Transformaciones de datos	234
Ejecución incremental y estado de transmisión	234
Transformaciones sin estado	235
Transformaciones con estado	235
Agregaciones de transmisión con estado	238
Agregaciones no basadas en el tiempo	238
Agregaciones con ventanas de tiempo de eventos	239
Uniones de transmisión	246
Combinaciones estáticas de flujo	246
Uniones de flujo a flujo	248
Cálculos arbitrarios con estado	253
Modelado de operaciones arbitrarias con estado con mapGroupsWithState()	254
Uso de tiempos de espera para administrar grupos inactivos	257
Generalización con flatMapGroupsWithState()	261
La optimización del rendimiento	262
Resumen	264
<b>9. Creación de lagos de datos fiables con Apache Spark.....</b>	<b>265</b>
La importancia de una solución de almacenamiento óptima	265
bases de datos	266
Una breve introducción a las bases de datos	266
Leer y escribir en bases de datos usando Apache Spark	267
Limitaciones de las bases de datos	267
Lagos de datos	268
Una breve introducción a los lagos de datos	268
Lectura y escritura en lagos de datos con Apache Spark	269
Limitaciones de los lagos de datos	270
Lakehouses: el siguiente paso en la evolución de las soluciones de almacenamiento	271
apache hudi	272
iceberg apache	272
lago delta	273
Construcción de lagos con Apache Spark y Delta Lake	274
Configuración de Apache Spark con Delta Lake	274
Carga de datos en una tabla de Delta Lake	275
Carga de flujos de datos en una tabla de Delta Lake	277
Hacer cumplir el esquema en escritura para evitar la corrupción de datos	278
Esquemas en evolución para acomodar datos cambiantes	279
Transformar datos existentes	279
Auditoría de cambios de datos con historial de operaciones	282
Consulta de instantáneas anteriores de una tabla con viajes en el tiempo	283
Resumen	284

<b>10. Aprendizaje automático con MLlib. . . . .</b>	<b>285</b>
¿Qué es el aprendizaje automático?	286
Aprendizaje supervisado	286
Aprendizaje no supervisado	288
¿Por qué Spark para el aprendizaje automático?	289
Diseño de canalizaciones de aprendizaje automático	289
Ingestión y exploración de datos Creación de conjuntos de datos de prueba y entrenamiento	290
Preparación de características con transformadores	293
Comprensión de la regresión lineal Uso de estimadores para crear modelos Creación de una canalización Evaluación de modelos	294
Almacenamiento y carga de modelos Ajuste de hiperparámetros Modelos basados en árboles	295
Validación cruzada de k-fold Optimización de canalizaciones Resumen	296
Resumen	302
Resumen	306
Resumen	307
Resumen	316
Resumen	320
Resumen	321
<b>11. Gestión, implementación y escalado de canalizaciones de aprendizaje automático con Apache Spark. . . 323</b>	
Gestión de modelos	323
MLflow	324
Opciones de implementación de modelos con MLlib	330
Lote	332
Transmisión	333
Patrones de exportación de modelos para inferencia en tiempo real	334
Aprovechamiento de Spark para modelos que no son MLlib	336
Pandas UDF	336
Spark para el ajuste de hiperparámetros distribuidos	337
Resumen	341
<b>12. Epílogo: Apache Spark 3.0. . . . .</b>	<b>343</b>
Spark Core y Spark SQL	343
Poda de partición dinámica	343
Ejecución de consulta adaptativa	345
Sugerencias de combinación de SQL	348
API de complemento de catálogo y DataSourceV2	349
Programador compatible con aceleradores	351
Transmisión estructurada	352
PySpark, UDF de Pandas y API de funciones de Pandas	354
UDF de Pandas rediseñadas con sugerencias de tipo de Python	354

Compatibilidad con iteradores en Pandas UDF	355
Nuevas API de funciones de Pandas	356
Funcionalidad modificada	357
Idiomas admitidos y obsoletos	357
Cambios en las API de DataFrame y Dataset	357
Comandos de explicación de DataFrame y SQL	358
Resumen	360
<b>Índice.</b>	<b>361</b>



---

# Prefacio

Apache Spark ha evolucionado significativamente desde que comencé el proyecto en UC Berkeley en 2009. Después de mudarme a Apache Software Foundation, el proyecto de código abierto ha tenido más de 1400 colaboradores de cientos de empresas y el [grupo](#) de reunión global [Spark](#) ha crecido a más de medio millón de miembros. La base de usuarios de Spark también se ha vuelto muy diversa y abarca desarrolladores de Python, R, SQL y JVM, con casos de uso que van desde la ciencia de datos hasta la inteligencia empresarial y la ingeniería de datos. He estado trabajando en estrecha colaboración con la comunidad de Apache Spark para ayudar a continuar con su desarrollo y estoy encantado de ver el progreso hasta el momento.

El lanzamiento de Spark 3.0 marca un hito importante para el proyecto y ha despertado la necesidad de material de aprendizaje actualizado. La idea de una segunda edición de Learning Spark ha surgido muchas veces, y estaba atrasada. Aunque fui coautor de [Learning Spark](#) y [Spark: La guía definitiva](#) (ambos O'Reilly), era hora de que dejara que la próxima generación de colaboradores de Spark continuara con la narrativa. Estoy encantado de que cuatro profesionales y desarrolladores experimentados, que han estado trabajando de cerca con Apache Spark desde sus inicios, se hayan unido para escribir esta segunda edición del libro, incorporando las API más recientes y las mejores prácticas para desarrolladores de Spark de una manera clara, y guía informativa.

El enfoque de los autores de esta edición es muy propicio para el aprendizaje práctico. Los conceptos clave de Spark y el procesamiento distribuido de big data se han resumido en capítulos fáciles de seguir. A través de los ejemplos de código ilustrativos del libro, los desarrolladores pueden generar confianza usando Spark y obtener una mayor comprensión de sus API estructuradas y cómo aprovecharlas. Espero que esta segunda edición de Learning Spark lo guíe en su viaje de procesamiento de datos a gran escala, independientemente de los problemas que desee abordar con Spark.

— Matei Zaharia, tecnólogo jefe,  
cofundador de Databricks, asistente. Profesor de Stanford  
y creador original de Apache Spark



---

# Prefacio

Te damos la bienvenida a la segunda edición de Learning Spark. Han pasado cinco años desde que se publicó la primera edición en 2015, escrita originalmente por Holden Karau, Andy Konwinski, Patrick Wendell y Matei Zaharia. Esta nueva edición se ha actualizado para reflejar la evolución de Apache Spark a través de Spark 2.x y Spark 3.0, incluido su ecosistema ampliado de fuentes de datos integradas y externas, aprendizaje automático y tecnologías de transmisión con las que Spark está estrechamente integrado.

A lo largo de los años desde su primer lanzamiento 1.x, Spark se ha convertido de facto en el motor de procesamiento unificado de big data. En el camino, ha ampliado su alcance para incluir soporte para varias cargas de trabajo analíticas. Nuestra intención es capturar y seleccionar esta evolución para los lectores, mostrando no solo cómo puede usar Spark, sino también cómo encaja en la nueva era de big data y aprendizaje automático. Por lo tanto, hemos diseñado cada capítulo para construir progresivamente sobre los cimientos establecidos por los capítulos anteriores, asegurando que el contenido sea adecuado para nuestra audiencia prevista.

## Para quien es este libro

La mayoría de los desarrolladores que lidian con big data son ingenieros de datos, científicos de datos o ingenieros de aprendizaje automático. Este libro está dirigido a aquellos profesionales que buscan usar Spark para escalar sus aplicaciones para manejar cantidades masivas de datos.

En particular, los ingenieros de datos aprenderán cómo usar las API estructuradas de Spark para realizar una exploración y análisis de datos complejos en datos tanto por lotes como de transmisión; use Spark SQL para consultas interactivas; use las fuentes de datos internas y externas de Spark para leer, refinar y escribir datos en diferentes formatos de archivo como parte de sus tareas de extracción, transformación y carga (ETL); y cree lagos de datos confiables con Spark y el formato de tabla Delta Lake de código abierto.

Para los científicos de datos y los ingenieros de aprendizaje automático, la biblioteca MLlib de Spark ofrece muchos algoritmos comunes para crear modelos de aprendizaje automático distribuido. Cubriremos cómo construir canalizaciones con MLlib, las mejores prácticas para el aprendizaje automático distribuido,

cómo usar Spark para escalar modelos de un solo nodo y cómo administrar e implementar estos modelos usando la biblioteca de código abierto MLflow.

Si bien el libro se enfoca en aprender Spark como un motor analítico para diversas cargas de trabajo, no cubriremos todos los lenguajes que admite Spark. La mayoría de los ejemplos de los capítulos están escritos en Scala, Python y SQL. Donde fue necesario, hemos infundido un poco de Java. Para aquellos interesados en aprender Spark con R, recomendamos [Mastering Spark with R](#) de Javier Luraschi, Kevin Kuo y Edgar Ruiz. (O'Reilly).

Finalmente, debido a que Spark es un motor distribuido, es fundamental desarrollar una comprensión de los conceptos de la aplicación Spark. Lo guiaremos a través de cómo su aplicación Spark interactúa con los componentes distribuidos de Spark y cómo la ejecución se descompone en tareas paralelas en un clúster. También cubriremos qué modos de implementación son compatibles y en qué entornos.

Si bien hay muchos temas que hemos elegido cubrir, hay algunos en los que hemos optado por no centrarnos. Estos incluyen las antiguas API de conjuntos de datos distribuidos resistentes (RDD) de bajo nivel y GraphX, la API de Spark para gráficos y cómputo paralelo de gráficos.

Tampoco hemos cubierto temas avanzados como cómo extender el optimizador Catalyst de Spark para implementar sus propias operaciones, cómo implementar su propio catálogo o cómo escribir sus propios fuentes y sumideros de datos DataSource V2. Aunque forman parte de Spark, están más allá del alcance de su primer libro sobre el aprendizaje de Spark.

En cambio, hemos centrado y organizado el libro en torno a las API estructuradas de Spark, en todos sus componentes, y cómo puede usar Spark para procesar datos estructurados a escala para realizar sus tareas de ingeniería o ciencia de datos.

## Cómo está organizado el libro

Organizamos el libro de manera que lo lleve de un capítulo a otro al presentar conceptos, demostrar estos conceptos a través de fragmentos de código de ejemplo y proporcionar ejemplos de código completo o cuadernos en el repositorio de [GitHub del libro](#).

### [Capítulo 1, Introducción a Apache Spark: un motor de análisis unificado](#)

Le presenta la evolución de big data y proporciona una descripción general de alto nivel de Apache Spark y su aplicación a big data.

### [Capítulo 2, Descarga de Apache Spark y primeros pasos](#) Lo guía

a través de la descarga y configuración de Apache Spark en su máquina local.

### [Capítulo 3, API estructuradas de Apache Spark hasta el Capítulo 6, Spark SQL y conjuntos de datos](#)

Estos capítulos se centran en el uso de las API estructuradas de DataFrame y Dataset para ingerir datos de fuentes de datos integradas y externas, aplicar funciones integradas y personalizadas y utilizar Spark SQL . Estos capítulos constituyen la base para capítulos posteriores, incorporando todos los cambios más recientes de Spark 3.0 donde corresponde.

## Capítulo 7, Optimización y ajuste de aplicaciones Spark

Le brinda las mejores prácticas para ajustar, optimizar, depurar e inspeccionar sus aplicaciones Spark a través de la interfaz de usuario de Spark, así como detalles sobre las configuraciones que puede ajustar para aumentar el rendimiento.

## Capítulo 8, Transmisión estructurada

Lo guía a través de la evolución del motor Spark Streaming y el modelo de programación de transmisión estructurada. Examina la anatomía de una consulta de transmisión típica y analiza las diferentes formas de transformar los datos de transmisión (agregaciones con estado, uniones de transmisión y agregación con estado arbitraria), al mismo tiempo que brinda orientación sobre cómo diseñar consultas de transmisión eficaces.

## Capítulo 9, Creación de lagos de datos fiables con Apache Spark

Examina tres soluciones de almacenamiento en formato de tabla de código abierto, como parte del ecosistema Spark, que emplean Apache Spark para crear lagos de datos confiables con garantías transaccionales. Debido a la estrecha integración de Delta Lake con Spark tanto para cargas de trabajo por lotes como de transmisión, nos enfocamos en esa solución y exploramos cómo facilita un nuevo paradigma en la administración de datos, Lakehouse.

## Capítulo 10, Aprendizaje automático con MLlib

Presenta MLlib, la biblioteca de aprendizaje automático distribuida para Spark, y lo guía a través de un ejemplo integral de cómo crear una canalización de aprendizaje automático, que incluye temas como la ingeniería de características, el ajuste de hiperparámetros, las métricas de evaluación y el guardado y la carga de modelos.

## Capítulo 11, Administración, implementación y escalado de canalizaciones de aprendizaje automático con Apache Spark

Cubre cómo rastrear y administrar sus modelos MLlib con MLflow, compara y contrasta diferentes opciones de implementación de modelos y explora cómo aprovechar Spark para modelos que no son MLlib para la inferencia de modelos distribuidos, ingeniería de funciones y/o ajuste de hiperparámetros.

## Capítulo 12, Epílogo: Apache Spark 3.0

El epílogo destaca características y cambios notables en Spark 3.0. Si bien la gama completa de mejoras y características es demasiado extensa para caber en un solo capítulo, destacamos los principales cambios que debe tener en cuenta y le recomendamos que consulte las notas de la versión cuando se lance oficialmente Spark 3.0.

A lo largo de estos capítulos, hemos incorporado o anotado las características de Spark 3.0 donde fue necesario y probamos todos los ejemplos de código y cuadernos con Spark 3.0.0-preview2.

## Cómo usar los ejemplos de código

Los ejemplos de código en el libro van desde fragmentos breves hasta aplicaciones Spark completas y cuadernos de extremo a extremo, en Scala, Python, SQL y, cuando sea necesario, Java.

Mientras que algunos fragmentos de código cortos en un capítulo son autónomos y se pueden copiar y pegar para ejecutarlos en un shell de Spark (pyspark o spark-shell), otros son fragmentos de aplicaciones de Spark independientes o cuadernos de extremo a extremo. Para ejecutar aplicaciones Spark independientes en Scala, Python o Java, lea las instrucciones en los archivos LÉAME del capítulo correspondiente en el repositorio de [GitHub de este libro](#).

En cuanto a los cuadernos, para ejecutarlos deberá registrarse para obtener una edición gratuita de la [comunidad de Databricks](#). cuenta. Detallamos cómo importar los cuadernos y crear un clúster usando Spark 3.0 en el [LÉAME](#).

## Software y configuración utilizados

La mayor parte del código de este libro y los cuadernos que lo acompañan se escribieron y probaron con Apache Spark 3.0.0-preview2, que estaba disponible para nosotros cuando estábamos escribiendo los capítulos finales.

Para cuando se publique este libro, Apache Spark 3.0 se habrá lanzado y estará disponible para el uso general de la comunidad. Le recomendamos que [descargue](#) y usa la versión oficial con las siguientes configuraciones para tu sistema operativo:

- Apache Spark 3.0 (precompilado para Apache Hadoop 2.7) •  
Kit de desarrollo de Java (JDK) 1.8.0

Si tiene la intención de usar solo Python, simplemente puede ejecutar pip install pyspark.

## Las convenciones usadas en este libro

En este libro se utilizan las siguientes convenciones tipográficas:

### Cursiva

Indica nuevos términos, URL, direcciones de correo electrónico, nombres de archivo y extensiones de archivo.

### Ancho constante

Se utiliza para listas de programas, así como dentro de párrafos para referirse a elementos de programas como nombres de variables o funciones, bases de datos, tipos de datos, variables de entorno, declaraciones y palabras clave.

### Negrita de ancho

**constante** Muestra comandos u otro texto que el usuario debe escribir literalmente.

### *Cursiva de ancho constante*

Muestra texto que debe ser reemplazado con valores proporcionados por el usuario o por valores determinados por el contexto.



Este elemento significa una nota general.

## **Uso de ejemplos de código**

Si tiene una pregunta técnica o un problema al usar los ejemplos de código, envíe un correo electrónico a [bookquestions@oreilly.com](mailto:bookquestions@oreilly.com).

Este libro está aquí para ayudarle a hacer su trabajo. En general, si se ofrece un código de ejemplo con este libro, puede usarlo en sus programas y documentación. No necesita comunicarse con nosotros para obtener permiso a menos que esté reproduciendo una parte significativa del código. Por ejemplo, escribir un programa que use varios fragmentos de código de este libro no requiere permiso. Vender o distribuir ejemplos de libros de O'Reilly requiere permiso. Responder una pregunta citando este libro y citando código de ejemplo no requiere permiso. La incorporación de una cantidad significativa de código de ejemplo de este libro en la documentación de su producto requiere permiso.

Apreciamos, pero generalmente no requerimos, atribución. Una atribución suele incluir el título, el autor, el editor y el ISBN. Por ejemplo: "Learning Spark, 2.<sup>a</sup> edición, de Jules S. Damji, Brooke Wenig, Tathagata Das y Denny Lee. Copyright 2020 Databricks, Inc., 978-1-492-05004-9."

Si cree que su uso de los ejemplos de código está fuera del uso justo o del permiso otorgado anteriormente, no dude en contactarnos en [permisos@oreilly.com](mailto:permisos@oreilly.com).

## **Aprendizaje en línea de O'Reilly**



Durante más de 40 años, **O'Reilly Media** ha brindado capacitación, conocimientos y perspectivas en tecnología y negocios para ayudar a las empresas a tener éxito.

Nuestra red única de expertos e innovadores comparte su conocimiento y experiencia a través de libros, artículos y nuestra plataforma de aprendizaje en línea. La plataforma de aprendizaje en línea de O'Reilly le brinda acceso a pedido a cursos de capacitación en vivo, rutas de aprendizaje en profundidad, entornos de codificación interactivos y una amplia colección de texto y video de O'Reilly y más de 200 editores más. Para obtener más información, visite <http://oreilly.com>.

## Cómo contactarnos

Dirija sus comentarios y preguntas sobre este libro a la editorial:

O'Reilly Media, Inc.  
1005 Gravenstein Highway North  
Sebastopol, CA 95472 800-998-9938 (en  
los Estados Unidos o Canadá) 707-829-0515 (internacional  
o local) 707-829-0104 (fax)

Visite nuestra página web para este libro, donde enumeramos erratas, ejemplos y cualquier información adicional, en <https://oreil.ly/LearningSpark2>.

Envíe un correo electrónico [a bookquestions@oreilly.com](mailto:a.bookquestions@oreilly.com) para comentar o hacer preguntas técnicas sobre este libro.

Para noticias e información sobre nuestros libros y cursos, visite <http://oreilly.com>.

Encuéntrenos en Facebook: <http://facebook.com/oreilly>

Síganos en Twitter: <http://twitter.com/oreillymedia> Míranos en

YouTube: <http://www.youtube.com/oreillymedia>

## Expresiones de gratitud

Este proyecto fue realmente un esfuerzo de equipo que involucró a muchas personas, y sin su apoyo y comentarios no hubiéramos podido terminar este libro, especialmente en los tiempos sin precedentes de COVID-19 de hoy.

En primer lugar, queremos agradecer a nuestro empleador, Databricks, por apoyarnos y asignarnos tiempo dedicado como parte de nuestro trabajo para terminar este libro. En particular, queremos agradecer a Matei Zaharia, Reynold Xin, Ali Ghodsi, Ryan Boyd y Rick Schultz por animarnos a escribir la segunda edición.

En segundo lugar, nos gustaría agradecer a nuestros revisores técnicos: Adam Breindel, Amir Issaei, Jacek Laskowski, Sean Owen y Vishwanath Subramanian. Sus comentarios diligentes y constructivos, informados por su experiencia técnica en la comunidad y el punto de vista de la industria, hicieron de este libro lo que es: un recurso valioso para aprender Spark.

Además de los revisores de libros formales, recibimos comentarios invaluables de otros conocedores de temas específicos y secciones de los capítulos, y queremos reconocer sus contribuciones. Muchas gracias a: Conor Murphy, Hyukjin Kwon, Maryann Xue, Niall Turritt, Wenchen Fan, Xiao Li y Yuanjian Li.

Por último, nos gustaría agradecer a nuestros colegas de Databricks (por su tolerancia con respecto a que no cumplamos con los plazos de los proyectos o los descuidemos), a nuestras familias y seres queridos (por su paciencia y empatía cuando escribimos a primera hora del día o tarde en la noche en entre semana y fines de semana), y toda la comunidad Spark de código abierto. Sin sus continuas contribuciones, Spark no estaría donde está hoy, y los autores no habríamos tenido mucho sobre lo que escribir.

¡Gracias a todos!



## CAPÍTULO 1

# Introducción a Apache Spark: Un motor de análisis unificado

Este capítulo expone los orígenes de Apache Spark y su filosofía subyacente. También examina los componentes principales del proyecto y su arquitectura distribuida. Si está familiarizado con la historia de Spark y los conceptos de alto nivel, puede omitir este capítulo.

## La génesis de la chispa

En esta sección, trazaremos el curso de la breve evolución de Apache Spark: su génesis, inspiración y adopción en la comunidad como un motor de procesamiento unificado de big data de facto.

### **Big Data y computación distribuida en Google**

Cuando pensamos en la escala, no podemos evitar pensar en la capacidad del motor de búsqueda de Google para indexar y buscar los datos del mundo en Internet a la velocidad del rayo. El nombre Google es sinónimo de escala. De hecho, Google es un error ortográfico deliberado del término matemático googol: ¡eso es 1 más 100 ceros!

Ni los sistemas de almacenamiento tradicionales, como los sistemas de administración de bases de datos relacionales (RDBMS), ni las formas imperativas de programación pudieron manejar la escala a la que Google quería construir y buscar documentos indexados en Internet. La necesidad resultante de nuevos enfoques condujo a la creación del **sistema de archivos de Google (GFS)**, **MapReduce (MR)**, y **Bigtable**.

Si bien GFS proporcionó un sistema de archivos tolerante a fallas y distribuido en muchos servidores de hardware básicos en una granja de clústeres, Bigtable ofreció almacenamiento escalable de datos estructurados en GFS. MR introdujo un nuevo paradigma de programación paralela,

basado en programación funcional, para el procesamiento a gran escala de datos distribuidos en GFS y Bigtable.

En esencia, sus aplicaciones de MR interactúan con el [sistema MapReduce](#) que envía código de computación (funciones map y reduce) al lugar donde residen los datos, favoreciendo la localidad de datos y la afinidad de rack de clúster en lugar de traer datos a su aplicación.

Los trabajadores en el clúster agregan y reducen los cálculos intermedios y producen un resultado adjunto final de la función de reducción, que luego se escribe en un almacenamiento distribuido donde su aplicación puede acceder a él. Este enfoque reduce significativamente el tráfico de la red y mantiene la mayor parte de la entrada/salida (E/S) local en el disco en lugar de distribuirla por la red.

La mayor parte del trabajo que hizo Google era propietario, pero las ideas expresadas en los [tres documentos](#) mencionados anteriormente estimuló ideas innovadoras en otras partes de la comunidad de código abierto, especialmente en Yahoo!, que estaba lidiando con desafíos similares de escala de big data para su motor de búsqueda.

## Hadoop en Yahoo!

Los desafíos computacionales y las soluciones expresadas en el documento GFS de Google proporcionaron un modelo para el [Sistema de archivos Hadoop \(HDFS\)](#), incluida la implementación de MapReduce como marco para la computación distribuida. Donado a la [Apache Software Foundation \(ASF\)](#), una organización sin fines de lucro independiente del proveedor, en abril de 2006, se convirtió en parte de [Apache Hadoop](#) framework de módulos relacionados: Hadoop Common, MapReduce, HDFS y Apache Hadoop YARN.

Aunque Apache Hadoop obtuvo una adopción generalizada fuera de Yahoo!, lo que inspiró a una gran comunidad de colaboradores de código abierto y a dos empresas comerciales basadas en código abierto (Cloudera y Hortonworks, ahora fusionadas), el marco MapReduce en HDFS tenía algunas deficiencias.

Primero, era difícil de manejar y administrar, con una complejidad operativa engorrosa. En segundo lugar, su API MapReduce de procesamiento por lotes general era detallada y requería una gran cantidad de código de configuración repetitivo, con tolerancia a fallas frágil. Tercero, con grandes lotes de trabajos de datos con muchos pares de tareas de MR, el resultado calculado intermedio de cada par se escribe en el disco local para la etapa subsiguiente de su operación (consulte la [Figura 1-1](#)).

Este rendimiento repetido de E/S de disco pasó factura: los trabajos de MR grandes podían ejecutarse durante horas, o incluso días.

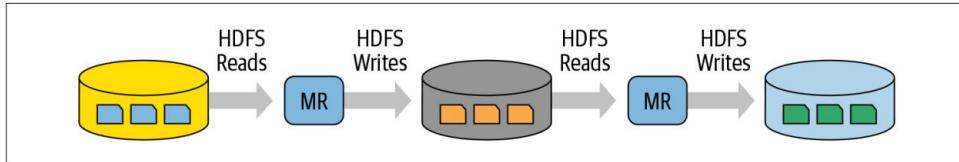


Figura 1-1. Iteración intermitente de lecturas y escrituras entre mapear y reducir cálculos

Y finalmente, a pesar de que Hadoop MR era propicio para trabajos a gran escala para el procesamiento por lotes general, se quedó corto para combinar otras cargas de trabajo, como aprendizaje automático, transmisión o consultas interactivas de tipo SQL.

Para manejar estas nuevas cargas de trabajo, los ingenieros desarrollaron sistemas a medida (Apache Hive, Apache Storm, Apache Impala, Apache Giraph, Apache Drill, Apache Mahout, etc.), cada uno con sus propias API y configuraciones de clúster, lo que aumenta aún más la complejidad operativa. de Hadoop y la empinada curva de aprendizaje de los desarrolladores.

La pregunta entonces fue (teniendo en cuenta el adagio de Alan Kay: "Las cosas simples deberían ser simples, las cosas complejas deberían ser posibles"), ¿había alguna manera de hacer que Hadoop y MR fueran más simples y rápidos?

### Los primeros años de Spark en

**AMPLab** Los investigadores de UC Berkeley que habían trabajado anteriormente en Hadoop MapReduce aceptaron este desafío con un proyecto al que llamaron Spark. Reconocieron que MR era ineficiente (o intratable) para trabajos de computación interactivos o iterativos y un marco complejo para aprender, por lo que desde el principio aceptaron la idea de hacer que Spark sea más simple, rápido y fácil. Este esfuerzo comenzó en 2009 en RAD Lab, que luego se convirtió en AMPLab (y ahora se conoce como RISELab).

**Primeros trabajos** publicado en Spark demostró que era de 10 a 20 veces más rápido que Hadoop MapReduce para ciertos trabajos. Hoy en día, es **muchos órdenes de magnitud más rápido**. El objetivo central del proyecto Spark era traer ideas prestadas de Hadoop Map-Reduce, pero para mejorar el sistema: hacerlo altamente tolerante a fallas y vergonzosamente paralelo, admitir el almacenamiento en memoria para resultados intermedios entre mapas iterativos e interactivos y reducir los cálculos , ofrecer API fáciles de componer en varios idiomas como modelo de programación y admita otras cargas de trabajo de manera unificada. Volveremos a esta idea de unificación en breve, ya que es un tema importante en Spark.

Para 2013, Spark había ganado un uso generalizado y algunos de sus creadores e investigadores originales (Matei Zaharia, Ali Ghodsi, Reynold Xin, Patrick Wendell, Ion Stoica y Andy Konwinski) donaron el proyecto Spark a la ASF y formaron una empresa llamada Databricks.

Databricks y la comunidad de desarrolladores de código abierto trabajaron para lanzar [Apache Spark 1.0](#) en mayo de 2014, bajo la dirección de la ASF. Este primer lanzamiento importante estableció el impulso para lanzamientos futuros frecuentes y contribuciones de características notables a Apache Spark de Databricks y más de 100 proveedores comerciales.

## ¿Qué es Apache Spark?

[chispa apache](#) es un motor unificado diseñado para el procesamiento de datos distribuidos a gran escala, en las instalaciones de los centros de datos o en la nube.

Spark proporciona almacenamiento en memoria para cálculos intermedios, lo que lo hace mucho más rápido que Hadoop MapReduce. Incorpora bibliotecas con API componibles para aprendizaje automático (MLlib), SQL para consultas interactivas (Spark SQL), procesamiento de secuencias (Structured Streaming) para interactuar con datos en tiempo real y procesamiento de gráficos (GraphX).

La filosofía de diseño de Spark se centra en cuatro características clave:

- Velocidad
- Facilidad de uso
- Modularidad •
- Extensibilidad

Echemos un vistazo a lo que esto significa para el marco.

### Speed

Spark ha perseguido el objetivo de la velocidad de varias maneras. En primer lugar, su implementación interna se beneficia enormemente de los grandes avances recientes de la industria del hardware para mejorar el precio y el rendimiento de las CPU y la memoria. Los servidores básicos de hoy en día son baratos, con cientos de gigabytes de memoria, múltiples núcleos y el sistema operativo subyacente basado en Unix que aprovecha el procesamiento paralelo y multiproceso eficiente. El marco está optimizado para aprovechar todos estos factores.

En segundo lugar, Spark construye sus cálculos de consulta como un gráfico acíclico dirigido (DAG); su programador DAG y su optimizador de consultas construyen un gráfico computacional eficiente que generalmente se puede descomponer en tareas que se ejecutan en paralelo entre los trabajadores del clúster. Y tercero, su motor de ejecución física, Tungsten, utiliza la generación de código de etapa completa para generar código compacto para la ejecución (cubriremos la optimización de SQL y la generación de código de etapa completa en el [Capítulo 3](#)).

Con todos los resultados intermedios retenidos en la memoria y su E/S de disco limitada, esto le da un gran impulso al rendimiento.

## **Facilidad de uso**

Spark logra la simplicidad al proporcionar una abstracción fundamental de una estructura de datos lógicos simple llamada Conjunto de datos distribuido resistente (RDD) sobre el cual se construyen todas las demás abstracciones de datos estructurados de nivel superior, como Marcos de datos y Conjuntos de datos. Al proporcionar un conjunto de transformaciones y acciones como operaciones, Spark ofrece un modelo de programación simple que puede usar para crear aplicaciones de big data en lenguajes familiares.

## **Modularidad**

Las operaciones de Spark se pueden aplicar en muchos tipos de cargas de trabajo y se pueden expresar en cualquiera de los lenguajes de programación admitidos: Scala, Java, Python, SQL y R. Spark ofrece bibliotecas unificadas con API bien documentadas que incluyen los siguientes módulos como núcleo componentes: Spark SQL, Spark Structured Streaming, Spark MLlib y GraphX, combinando todas las cargas de trabajo que se ejecutan en un solo motor. Echaremos un vistazo más de cerca a todos estos en la siguiente sección.

Puede escribir una sola aplicación Spark que pueda hacerlo todo, sin necesidad de motores distintos para cargas de trabajo dispares, sin necesidad de aprender API separadas. Con Spark, obtiene un motor de procesamiento unificado para sus cargas de trabajo.

## **Extensibilidad**

Spark se enfoca en su rápido motor de cómputo paralelo en lugar de en el almacenamiento. A diferencia de Apache Hadoop, que incluía tanto almacenamiento como cómputo, Spark los desacopla.

Eso significa que puede usar Spark para leer datos almacenados en innumerables fuentes (Apache Hadoop, Apache Cassandra, Apache HBase, MongoDB, Apache Hive, RDBMS y más) y procesarlo todo en la memoria. Los DataFrameReaders y DataFrame Writers de Spark también se pueden ampliar para leer datos de otras fuentes, como Apache Kafka, Kinesis, Azure Storage y Amazon S3, en su abstracción de datos lógicos, en los que puede operar.

La comunidad de desarrolladores de Spark mantiene una lista de [paquetes de Spark de terceros](#) como parte del ecosistema en crecimiento (ver [Figura 1-2](#)). Este rico ecosistema de paquetes incluye conectores Spark para una variedad de fuentes de datos externas, monitores de rendimiento y más.

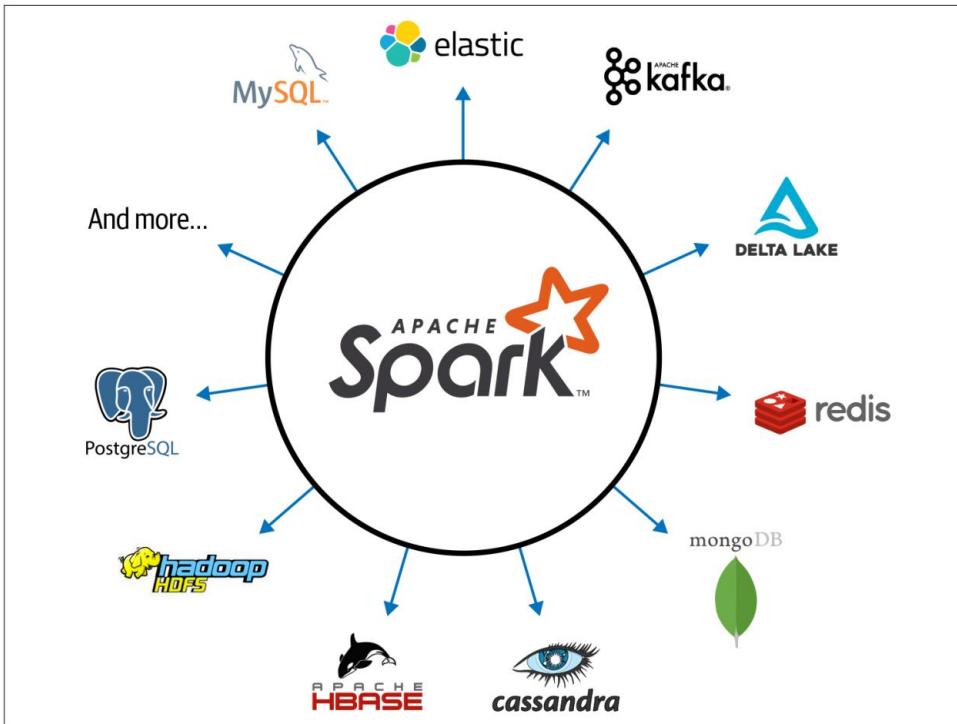


Figura 1-2. El ecosistema de conectores de Apache Spark

## Uni ed Analytics

Si bien la noción de unificación no es exclusiva de Spark, es un componente central de su filosofía de diseño y evolución. En noviembre de 2016, la Association for Computing Machinery (ACM) reconoció a Apache Spark y otorgó a sus creadores originales el prestigioso premio ACM por su trabajo . describiendo a Apache Spark como un "motor unificado para el procesamiento de Big Data". El artículo premiado señala que Spark reemplaza todos los motores independientes de procesamiento por lotes, gráficos, secuencias y consultas como Storm, Impala, Dremel, Pregel, etc. con una pila unificada de componentes que aborda diversas cargas de trabajo en un único motor rápido distribuido.

### Componentes de Apache Spark como una pila unificada

Como se muestra en la Figura 1-3, Spark ofrece cuatro componentes distintos como bibliotecas para diversas cargas de trabajo: Spark SQL, Spark MLlib, Spark Structured Streaming y GraphX. Cada uno de estos componentes es independiente del motor central tolerante a fallas de Spark, ya que usa API para escribir su aplicación Spark y Spark lo convierte en un DAG que ejecuta el motor central.

Entonces, ya sea que escriba su código Spark usando las API estructuradas provistas (que trataremos en el Capítulo 3) en Java, R, Scala, SQL o

Python, el código subyacente se descompone en un código de bytes muy compacto que se ejecuta en las JVM de los trabajadores en todo el clúster.

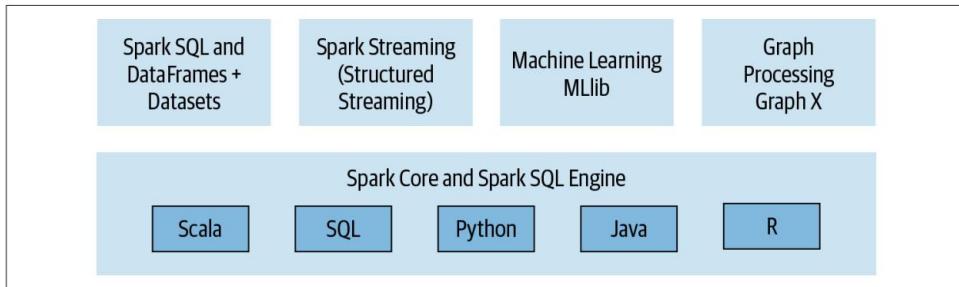


Figura 1-3. Pila de API y componentes de Apache Spark

Veamos cada uno de estos componentes con más detalle.

#### Chispa SQL

Este módulo funciona bien con datos estructurados. Puede leer datos almacenados en una tabla RDBMS o desde formatos de archivo con datos estructurados (CSV, texto, JSON, Avro, ORC, Parquet, etc.) y luego construir tablas permanentes o temporales en Spark. Además, al usar las API estructuradas de Spark en Java, Python, Scala o R, puede combinar consultas de tipo SQL para consultar los datos recién leídos en un Spark DataFrame. Hasta la fecha, Spark SQL es compatible con [ANSI SQL:2003](#) y también funciona como un motor SQL puro.

Por ejemplo, en este fragmento de código de Scala, puede leer desde un archivo JSON almacenado en Amazon S3, cree una tabla temporal y emita una consulta similar a SQL sobre los resultados leídos en la memoria como un Spark DataFrame:

```
// En Scala //
Leer datos del depósito de Amazon S3 en un Spark DataFrame
spark.read.json("s3://apache_spark/data/committers.json")
.createOrReplaceTempView("contratistas")
// Emite una consulta SQL y devuelva el resultado como Spark DataFrame val
resultados = chispa.sql("""SELECCIONE nombre, organización, módulo, versión,
número_compromisos DE confirmadores DONDE módulo = 'mllib' Y número_compromisos
> 10 ORDEN POR número_commits DESC """)
```

Puede escribir fragmentos de código similares en Python, R o Java, y el código de bytes generado será idéntico, lo que dará como resultado el mismo rendimiento.

#### Spark MLlib

Spark viene con una biblioteca que contiene algoritmos comunes de aprendizaje automático (ML) llamados MLlib. Desde el primer lanzamiento de Spark, el rendimiento de este componente de la biblioteca ha mejorado significativamente debido a las mejoras del motor subyacente de Spark 2.x.

MLlib proporciona muchos algoritmos de aprendizaje automático populares construidos sobre API basadas en DataFrame de alto nivel para construir modelos.



A partir de Apache Spark 1.6, el [proyecto MLlib](#) se divide en dos paquetes: spark.mllib y spark.ml. La API basada en DataFrame es la última, mientras que la primera contiene las API basadas en RDD, que ahora están en modo de mantenimiento. Todas las funciones nuevas van a spark.ml. Este libro se refiere a "MLlib" como la biblioteca general para el aprendizaje automático en Apache Spark.

Estas API le permiten extraer o transformar funciones, crear canalizaciones (para entrenamiento y evaluación) y conservar modelos (para guardarlos y volver a cargarlos) durante la implementación. Las utilidades adicionales incluyen el uso de operaciones y estadísticas comunes de álgebra lineal. MLlib incluye otras primitivas de ML de bajo nivel, incluida una optimización de descenso de gradiente genérica. El siguiente fragmento de código de Python resume las operaciones básicas que un científico de datos puede realizar al crear un modelo (se analizarán ejemplos más extensos en los Capítulos 10 y 11):

```
# En Python
desde pyspark.ml.classification import LogisticRegression
...
entrenamiento = chispa.read.csv("s3://...")
prueba = chispa.read.csv("s3://...")

# Cargar datos de
entrenamiento lr = LogisticRegression(maxIter=10, regParam=0.3, elasticNetParam=0.8)

# Ajustar el modelo
lrModel = lr.fit(entrenamiento)

# Predecir
lrModel.transform(prueba)
...
```

### Spark Structured Streaming

Apache Spark 2.0 introdujo un [modelo experimental de transmisión continua](#) y [API de transmisión estructurada](#), construido sobre el motor Spark SQL y las API basadas en DataFrame. En Spark 2.2, la transmisión estructurada estaba disponible en general, lo que significa que los desarrolladores podían usarla en sus entornos de producción.

Necesario para que los desarrolladores de big data combinen y reaccionen en tiempo real tanto a datos estáticos como a datos de transmisión de motores como Apache Kafka y otras fuentes de transmisión, el nuevo modelo ve una transmisión como una tabla en continuo crecimiento, con nuevas filas de datos adjuntas al final. . Los desarrolladores pueden simplemente tratar esto como una tabla estructurada y realizar consultas en ella como lo harían con una tabla estática.

Debajo del modelo de transmisión estructurada, el motor central Spark SQL maneja todos los aspectos de la tolerancia a fallas y la semántica de datos tardíos, lo que permite a los desarrolladores concentrarse en escribir aplicaciones de transmisión con relativa facilidad. Este nuevo modelo obvió el antiguo modelo DStreams en la serie 1.x de Spark, que analizaremos con más detalle en el [Capítulo 8](#). Además, Spark 2.x y Spark 3.0 ampliaron la gama de fuentes de datos de transmisión para incluir Apache Kafka, Kinesis y almacenamiento en la nube o basado en HDFS.

El siguiente fragmento de código muestra la anatomía típica de una aplicación de transmisión estructurada. Lee desde un socket localhost y escribe los resultados del recuento de palabras en un tema de Apache Kafka:

```
# En Python #
Leer un flujo desde un host local desde
pyspark.sql.functions importar explotar, dividir líneas =
(spark .readStream .format("socket") .option("host",
"localhost") .option("port ", 9999) .cargar())

# Realizar la transformación #
Dividir las líneas en palabras palabras
= líneas.seleccionar(explotar(dividir(líneas.valor, " ")).alias("palabra"))

# Generar conteo de palabras en
ejecución word_counts = palabras.groupBy("palabra").count()

# Escribir en la transmisión a la consulta de
Kafka =
(word_counts .writeStream .format("kafka") .option("tema", "salida"))
```

## GraphX

Como sugiere el nombre, GraphX es una biblioteca para manipular gráficos (p. ej., gráficos de redes sociales, rutas y puntos de conexión, o gráficos de topología de redes) y realizar cálculos de gráficos paralelos. Ofrece los algoritmos de gráficos estándar para análisis, conexiones y recorridos, aportados por usuarios de la comunidad: los algoritmos disponibles incluyen PageRank, componentes conectados y conteo de triángulos.<sup>1</sup> Este fragmento de código muestra un ejemplo simple de cómo unir dos gráficos usando el API de GraphX:

---

<sup>1</sup> Contribuido a la comunidad por Databricks como un proyecto de código abierto, [GraphFrames](#) es un gráfico general biblioteca de procesamiento que es similar a GraphX de Apache Spark pero usa API basadas en DataFrame.

```
// En Scala
val graph = Graph(vértices, aristas)
mensajes = spark.textFile("hdfs://...") val graph2
= graph.joinVertices(mensajes) { (id, vertex, msg) =>
...
}
```

## Ejecución distribuida de Apache Spark Si ha

leído hasta aquí, ya sabe que Spark es un motor de procesamiento de datos distribuido con sus componentes trabajando en colaboración en un grupo de máquinas. Antes de explorar la programación con Spark en los siguientes capítulos de este libro, debe comprender cómo funcionan juntos y se comunican todos los componentes de la arquitectura distribuida de Spark, y qué modos de implementación están disponibles.

Comencemos observando cada uno de los componentes individuales que se muestran en la **Figura 1-4** y cómo encajan en la arquitectura de Spark. En un alto nivel de la arquitectura de Spark, una aplicación de Spark consiste en un programa controlador que es responsable de orquestar operaciones paralelas en el clúster de Spark. El controlador accede a los componentes distribuidos en el clúster (los ejecutores de Spark y el administrador del clúster) a través de una `SparkSession`.

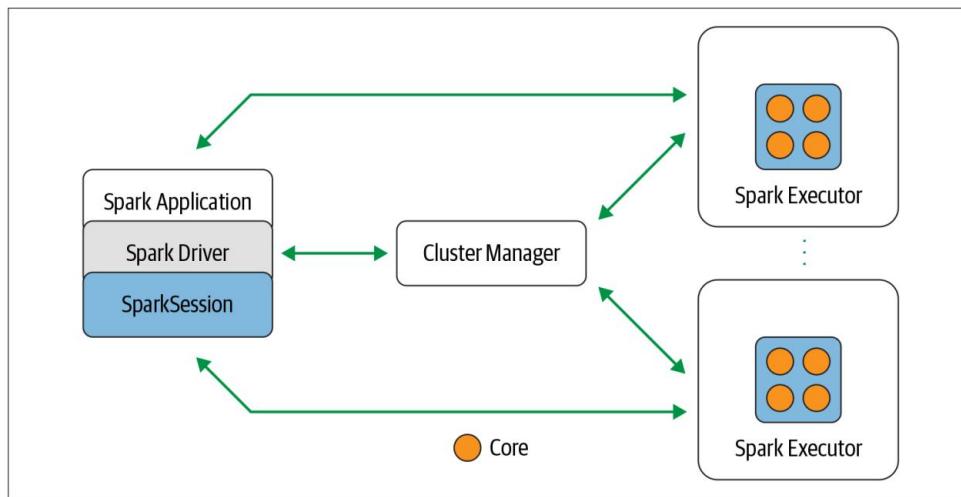


Figura 1-4. Arquitectura y componentes de Apache Spark

### Controlador

**Spark** Como parte de la aplicación Spark responsable de instanciar una `SparkSession`, el controlador Spark tiene varias funciones: se comunica con el administrador del clúster; solicita recursos (CPU, memoria, etc.) al administrador de clústeres para los ejecutores de Spark (JVM); y transforma todas las operaciones de Spark en cálculos DAG, programaciones

y distribuye su ejecución como tareas entre los ejecutores de Spark. Una vez asignados los recursos, se comunica directamente con los ejecutores.

### SparkSession

En Spark 2.0, SparkSession se convirtió en un conducto unificado para todas las operaciones y datos de Spark. No solo incluyó los puntos de entrada anteriores a Spark como SparkContext, SQLContext, HiveContext, SparkConf y StreamingContext, pero también simplificó y facilitó el trabajo con Spark.



Aunque en Spark 2.x SparkSession subsume todos los demás contextos, aún puede acceder a los contextos individuales y sus métodos respectivos. De esta forma, la comunidad mantuvo la retrocompatibilidad. Es decir, su antiguo código 1.x con SparkContext o SQLContext seguirá funcionando.

A través de este único conducto, puede crear parámetros de tiempo de ejecución de JVM, definir marcos de datos y conjuntos de datos, leer fuentes de datos, acceder a metadatos de catálogos y emitir consultas Spark SQL. SparkSession proporciona un único punto de entrada unificado a toda la funcionalidad de Spark.

En una aplicación Spark independiente, puede crear una SparkSession utilizando una de las API de alto nivel en el lenguaje de programación que elija. En el shell de Spark (más sobre esto en el próximo capítulo), la SparkSession se crea para usted y puede acceder a ella a través de una variable global llamada spark o sc.

Mientras que en Spark 1.x habría tenido que crear contextos individuales (para streaming, SQL, etc.), introduciendo código repetitivo adicional, en una aplicación Spark 2.x puede crear una SparkSession por JVM y usarla para realizar varias operaciones de Spark.

Echemos un vistazo a un ejemplo:

```
// En Scala
import org.apache.spark.sql.SparkSession

// Construir SparkSession
val chispa =
  SparkSession.builder

  .appName("LearnSpark").config("spark.sql.shuffle.partitions",
  6).getOrCreate()
  ...
  // Usar la sesión para leer JSON val
  people = spark.read.json(" ... ")
  ...
  // Usa la sesión para emitir una consulta SQL val
  resultsDF = spark.sql("SELECCIONA ciudad, población, estado, código postal FROM table_name")
```

## Administrador

**de clústeres** El administrador de clústeres es responsable de administrar y asignar recursos para el clúster de nodos en los que se ejecuta su aplicación Spark. Actualmente, Spark admite cuatro administradores de clústeres: el administrador de clústeres independiente integrado, Apache Hadoop YARN, Apache Mesos y Kubernetes.

## Ejecutor de

**Spark** Un ejecutor de Spark se ejecuta en cada nodo trabajador del clúster. Los ejecutores se comunican con el programa conductor y son los encargados de ejecutar tareas sobre los trabajadores. En la mayoría de los modos de implementación, solo se ejecuta un único ejecutor por nodo.

## Modos de

**implementación** Una característica atractiva de Spark es su compatibilidad con innumerables modos de implementación, lo que permite que Spark se ejecute en diferentes configuraciones y entornos. Debido a que el administrador de clústeres es independiente de dónde se ejecuta (siempre que pueda administrar los ejecutores de Spark y cumplir con las solicitudes de recursos), Spark se puede implementar en algunos de los entornos más populares, como Apache Hadoop YARN y Kubernetes, y puede operar en diferentes modos. La Tabla 1-1 resume los modos de implementación disponibles.

Tabla 1-1. Hoja de trucos para los modos de implementación de Spark

modo	conductor de chispa	ejecutor de chispas	Cluster Manager
Local	Se ejecuta en una sola JVM, como una computadora portátil o un solo nodo	Se ejecuta en la misma JVM que el controlador	Se ejecuta en el mismo host
Ser único	Puede ejecutarse en cualquier nodo del clúster	Cada nodo en el clúster lanzará su propio ejecutor JVM	Se puede asignar arbitrariamente a cualquier host en el clúster
YARN (cliente)	Se ejecuta en un cliente, no forma parte del clúster	El contenedor de NodeManager de YARN	El Administrador de recursos de YARN trabaja con el Maestro de aplicaciones de YARN para asignar los contenedores en NodeManagers para ejecutores
HILo (grupo)	Funciona con la aplicación YARN Maestría	Igual que el modo de cliente YARN	Igual que el modo de cliente YARN
Kubernetes	Se ejecuta en un pod de Kubernetes	Cada trabajador corre dentro de su propio pod	Maestro de Kubernetes

## Datos distribuidos y particiones

Los datos físicos reales se distribuyen a través del almacenamiento como particiones que residen en HDFS o almacenamiento en la nube (consulte la Figura 1-5). Si bien los datos se distribuyen como particiones en el clúster físico, Spark trata cada partición como una abstracción de datos lógicos de alto nivel, como un marco de datos en la memoria. Aunque esto no siempre es posible, a cada ejecutor de Spark se le asigna preferentemente una tarea que requiere que lea la partición más cercana a él en la red, observando la localidad de los datos.

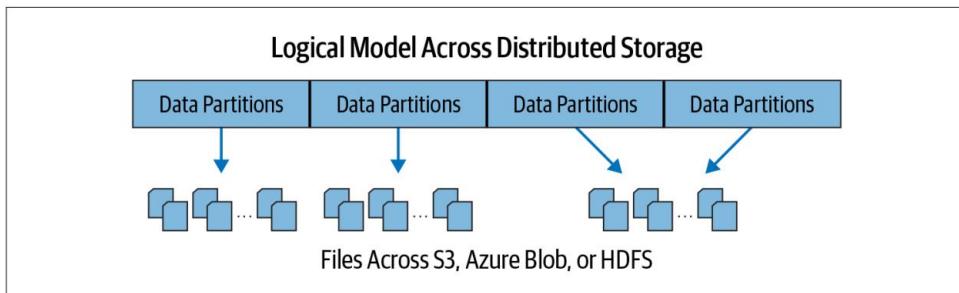


Figura 1-5. Los datos se distribuyen entre máquinas físicas.

La partición permite un paralelismo eficiente. Un esquema distribuido de dividir los datos en fragmentos o particiones permite a los ejecutores de Spark procesar solo los datos que están cerca de ellos, lo que minimiza el ancho de banda de la red. Es decir, al núcleo de cada ejecutor se le asigna su propia partición de datos para trabajar (vea la Figura 1-6).

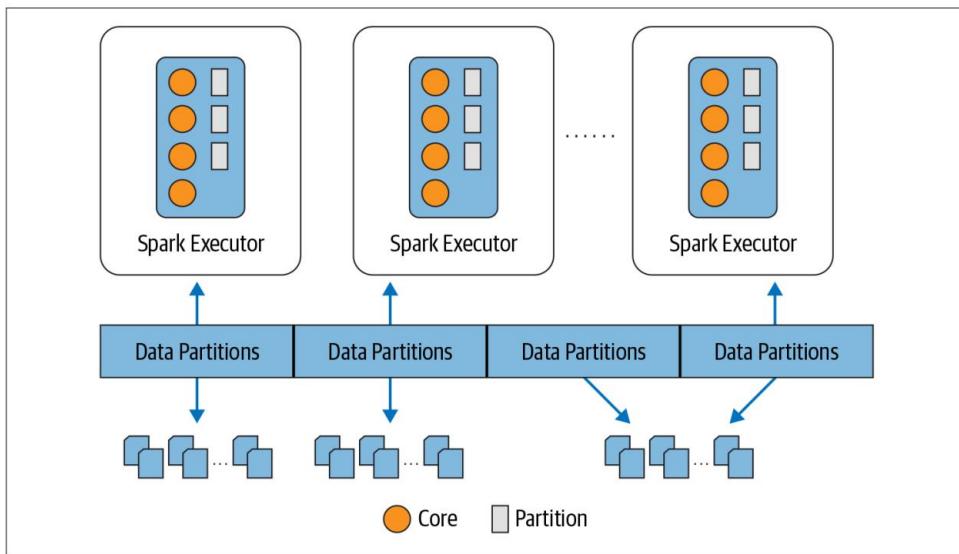


Figura 1-6. El núcleo de cada ejecutor obtiene una partición de datos para trabajar

Por ejemplo, este fragmento de código dividirá los datos físicos almacenados en los clústeres en ocho particiones, y cada ejecutor obtendrá una o más particiones para leer en su memoria:

```
# En Python
log_df = spark.read.text("path_to_large_text_file").repartition(8)
print(log_df.rdd.getNumPartitions())
```

Y este código creará un DataFrame de 10.000 enteros distribuidos en ocho particiones de memoria:

```
# En Python
df = spark.range(0, 10000, 1, 8)
print(df.rdd.getNumPartitions())
```

Ambos fragmentos de código se imprimirán 8.

En los Capítulos 3 y 7, discutiremos cómo ajustar y cambiar la configuración de partición para un paralelismo máximo en función de cuántos núcleos tiene en sus ejecutores.

## La experiencia del desarrollador

De todas las delicias de los desarrolladores, ninguna es más atractiva que un conjunto de API componibles que aumentan la productividad y son fáciles de usar, intuitivas y expresivas. Uno de los principales atractivos de Apache Spark para los desarrolladores han sido sus [API fáciles de usar](#), para operar en conjuntos de datos pequeños a grandes, a través de lenguajes: Scala, Java, Python, SQL y R.

Una motivación principal detrás de Spark 2.x fue unificar y simplificar el marco al limitar la cantidad de conceptos con los que los desarrolladores tienen que lidiar. Spark 2.x introdujo API de abstracción de nivel superior como construcciones de lenguaje específicas de dominio, lo que hizo que la programación de Spark fuera muy expresiva y una experiencia agradable para los desarrolladores. Usted expresa lo que quiere que calcule la tarea u operación, no cómo calcularlo, y deja que Spark determine cuál es la mejor manera de hacerlo por usted. Cubriremos estas API estructuradas en el [Capítulo 3](#), pero primero echemos un vistazo a quiénes son los desarrolladores de Spark.

son.

## ¿Quién usa Spark y para qué?

No es sorprendente que la mayoría de los desarrolladores que lidian con big data sean ingenieros de datos, científicos de datos o ingenieros de aprendizaje automático. Se sienten atraídos por Spark porque les permite crear una variedad de aplicaciones usando un solo motor, con lenguajes de programación familiares.

Por supuesto, los desarrolladores pueden usar muchos sombreros y, a veces, realizar tareas de ciencia de datos e ingeniería de datos, especialmente en empresas emergentes o grupos de ingeniería más pequeños. Sin embargo, entre todas estas tareas, los datos (cantidades masivas de datos) son la base.

### Tareas de ciencia de datos

Como disciplina que ha cobrado importancia en la era de los grandes datos, la ciencia de datos se trata de usar datos para contar historias. Pero antes de que puedan narrar las historias, los científicos de datos deben limpiar los datos, explorarlos para descubrir patrones y construir modelos para predecir o sugerir resultados. Algunas de estas tareas requieren conocimientos de estadística, matemáticas, informática y programación.

La mayoría de los científicos de datos dominan el uso de herramientas analíticas como SQL, se sienten cómodos con bibliotecas como NumPy y pandas, y conocen lenguajes de programación como R.

y Pitón. Pero también deben saber cómo manipular o transformar datos, y cómo utilizar algoritmos de clasificación, regresión o agrupamiento establecidos para construir modelos. A menudo, sus tareas son iterativas, interactivas o ad hoc, o experimentales para afirmar sus hipótesis.

Afortunadamente, Spark es compatible con estas diferentes herramientas. MLlib de Spark ofrece un conjunto común de algoritmos de aprendizaje automático para crear canalizaciones modelo, utilizando estimadores de alto nivel, transformadores y caracterizadores de datos. Spark SQL y Spark Shell facilitan la exploración de datos interactiva y ad hoc.

Además, Spark permite a los científicos de datos abordar grandes conjuntos de datos y escalar su entrenamiento y evaluación de modelos. Apache Spark 2.4 introdujo un nuevo programador de pandillas, como parte del [Proyecto Hidrógeno](#), para adaptarse a las necesidades tolerantes a fallas de capacitación y programación de modelos de aprendizaje profundo de manera distribuida, y Spark 3.0 ha introducido la capacidad de admitir la recopilación de recursos de GPU en los modos de implementación independiente, YARN y Kubernetes. Esto significa que los desarrolladores cuyas tareas exigen técnicas de aprendizaje profundo pueden usar Spark.

### Tareas de ingeniería de

**datos** Después de construir sus modelos, los científicos de datos a menudo necesitan trabajar con otros miembros del equipo, quienes pueden ser responsables de implementar los modelos. O pueden necesitar trabajar en estrecha colaboración con otros para crear y transformar datos sin procesar y sucios en datos limpios que otros científicos de datos puedan consumir o utilizar fácilmente. Por ejemplo, un modelo de clasificación o agrupación no existe de forma aislada; funciona junto con otros componentes como una aplicación web o un motor de transmisión como Apache Kafka, o como parte de una canalización de datos más grande. Esta tubería a menudo es construida por ingenieros de datos.

Los ingenieros de datos tienen una sólida comprensión de los principios y metodologías de la ingeniería de software, y poseen habilidades para construir canalizaciones de datos escalables para un caso de uso empresarial establecido. Las canalizaciones de datos permiten transformaciones de extremo a extremo de datos sin procesar provenientes de innumerables fuentes: los datos se limpian para que los desarrolladores puedan consumirlos posteriormente, almacenarlos en la nube o en NoSQL o RDBMS para la generación de informes, o ponerlos a disposición de los analistas de datos a través de herramientas de inteligencia de negocios.

Spark 2.x introdujo un modelo de transmisión evolutivo llamado [aplicaciones continuas](#) con Flujo Estructurado (discutido en detalle en el [Capítulo 8](#)). Con las API de transmisión estructurada, los ingenieros de datos pueden crear canalizaciones de datos complejas que les permitan obtener datos ETL de fuentes de datos estáticas y en tiempo real.

Los ingenieros de datos usan Spark porque proporciona una forma sencilla de parallelizar los cálculos y oculta toda la complejidad de la distribución y la tolerancia a fallas. Esto los deja libres para enfocarse en el uso de API basadas en DataFrame de alto nivel y consultas de lenguaje específico de dominio (DSL) para hacer ETL, leer y combinar datos de múltiples fuentes.

Las mejoras de rendimiento en Spark 2.x y Spark 3.0, debido al [optimizador Catalyst](#) para SQL y [Tungsten](#) para la generación de código compacto, han facilitado mucho la vida de los ingenieros de datos. Pueden optar por utilizar cualquiera de los tres [Spark](#) API (RDD, DataFrames o conjuntos de datos) que se adaptan a la tarea en cuestión y aprovechan los beneficios de Spark.

#### Casos de uso populares de Spark

Tanto si es ingeniero de datos, científico de datos o ingeniero de aprendizaje automático, Spark le resultará útil para los siguientes casos de uso:

- Procesamiento en paralelo de grandes conjuntos de datos distribuidos en un clúster.
- Realización de consultas ad hoc o interactivas para explorar y visualizar conjuntos de datos.
- Creación, entrenamiento y evaluación de modelos de aprendizaje automático mediante MLlib.
- Análisis de conjuntos de datos gráficos y redes sociales

#### Adopción y expansión de la comunidad

No es sorprendente que Apache Spark tocara una fibra sensible en la comunidad de código abierto, especialmente entre los ingenieros y científicos de datos. Su filosofía de diseño y su inclusión como proyecto de Apache Software Foundation han despertado un gran interés entre la comunidad de desarrolladores.

Hoy en día, hay más de 600 [grupos de Apache Spark Meetup](#) a nivel mundial con cerca de medio millón de miembros. Cada semana, alguien en el mundo da una charla en una reunión o conferencia o comparte una publicación de blog sobre cómo usar Spark para construir canalizaciones de datos. Cumbre [Spark + IA](#) es la conferencia más grande dedicada al uso de Spark para el aprendizaje automático, la ingeniería de datos y la ciencia de datos en muchas verticales.

Desde el primer lanzamiento de Spark 1.0 en 2014, ha habido muchos lanzamientos menores y mayores, y el lanzamiento principal más reciente de Spark 3.0 llegará en 2020. Este libro cubrirá aspectos de Spark 2.x y Spark 3.0. En el momento de su publicación, la comunidad habrá lanzado Spark 3.0 y la mayor parte del código de este libro se ha probado con Spark 3.0-preview2.

En el transcurso de sus lanzamientos, Spark ha seguido atrayendo colaboradores de todo el mundo y de numerosas organizaciones. En la actualidad, Spark tiene cerca de 1500 colaboradores, más de 100 lanzamientos, 21 000 bifurcaciones y unas 27 000 confirmaciones en GitHub, como muestra la [Figura 1-7](#). Y esperamos que cuando termine este libro, se sienta obligado a contribuir también.

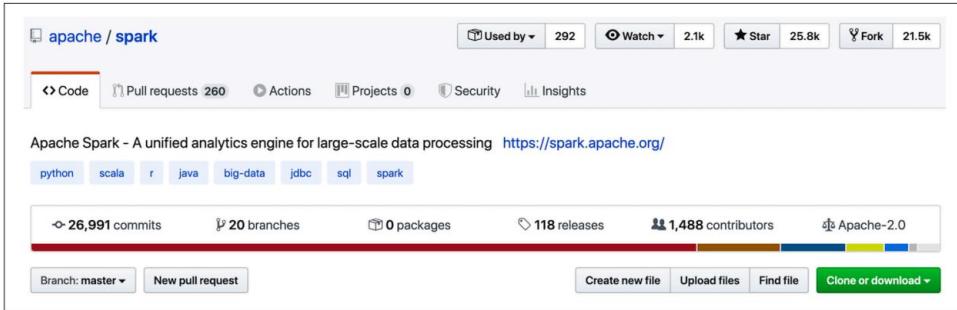


Figura 1-7. El estado de Apache Spark en GitHub (fuente: <https://github.com/apache/spark>)

Ahora podemos centrar nuestra atención en la diversión de aprender: dónde y cómo empezar a usar Spark. En el próximo capítulo, le mostraremos cómo comenzar a utilizar Spark en tres sencillos pasos.



## CAPITULO 2

# Descarga de Apache Spark y primeros pasos

En este capítulo, lo configuraremos con Spark y lo guaremos a través de tres pasos simples que puede seguir para comenzar a escribir su primera aplicación independiente.

Usaremos el modo local, donde todo el procesamiento se realiza en una sola máquina en un shell de Spark; esta es una manera fácil de aprender el marco, proporcionando un ciclo de retroalimentación rápido para realizar operaciones de Spark de forma iterativa. Con un shell de Spark, puede crear prototipos de operaciones de Spark con conjuntos de datos pequeños antes de escribir una aplicación Spark compleja, pero para conjuntos de datos grandes o trabajo real en el que desea obtener los beneficios de la ejecución distribuida, el modo local no es adecuado: querrá usar los modos de implementación YARN o Kubernetes en su lugar.

Si bien el shell de Spark solo es compatible con Scala, Python y R, puede escribir una aplicación de Spark en cualquiera de los lenguajes admitidos (incluido Java) y emitir consultas en Spark SQL. Esperamos que tenga cierta familiaridad con el idioma de su elección.

## Paso 1: Descargar Apache Spark

Para comenzar, vaya a la [página de descarga de Spark](#), seleccione "Preconstruido para Apache Hadoop 2.7" del menú desplegable en el paso 2 y haga clic en el enlace "Descargar Spark" en el paso 3 ([Figura 2-1](#)).

The screenshot shows the Apache Spark website's download section. It features the Apache Spark logo at the top left, followed by the tagline "Lightning-fast unified analytics engine". Below the logo is a navigation bar with links for Download, Libraries, Documentation, Examples, Community, Developers, and Apache Software Foundation. The main content area is titled "Download Apache Spark™" and contains a form for selecting a spark release (set to "3.0.0-preview2 (Dec 23 2019)"), a package type (set to "Pre-built for Apache Hadoop 2.7"), and a download link ("spark-3.0.0-preview2-bin-hadoop2.7.tgz"). Below the form, it says "Verify this release using the 3.0.0-preview2 signatures, checksums and project release KEYS." A note states that Spark is pre-built with Scala 2.11 except version 2.4.2, which is pre-built with Scala 2.12. To the right, a sidebar titled "Latest News" lists recent releases: "Preview release of Spark 3.0 (Dec 23, 2019)", "Preview release of Spark 3.0 (Nov 06, 2019)", "Spark 2.3.4 released (Sep 09, 2019)", and "Spark 2.4.4 released (Sep 01, 2019)".

Figura 2-1. La página de descarga de Apache Spark

Esto descargará el tarball `spark-3.0.0-preview2-bin-hadoop2.7.tgz`, que contiene todos los archivos binarios relacionados con Hadoop que necesitará para ejecutar Spark en modo local en su computadora portátil. Alternativamente, si va a instalarlo en una instalación existente de HDFS o Hadoop, puede seleccionar la versión de Hadoop correspondiente en el menú desplegable. Cómo compilar desde el código fuente está más allá del alcance de este libro, pero puede leer más sobre esto en la [documentación](#).



En el momento en que se imprimió este libro, Apache Spark 3.0 todavía estaba en modo de vista previa, pero puede descargar la versión más reciente de Spark 3.0 utilizando el mismo método de descarga e instrucciones.

Desde el lanzamiento de Apache Spark 2.2, los desarrolladores que solo se preocupan por aprender Spark en Python tienen la opción de instalar PySpark desde el [repositorio de PyPI](#). Si solo programa en Python, no tiene que instalar todas las demás bibliotecas necesarias para ejecutar Scala, Java o R; esto hace que el binario sea más pequeño. Para instalar PySpark desde PyPI, simplemente ejecute `pip install pyspark`.

Hay algunas dependencias adicionales que se pueden instalar para SQL, ML y MLlib, a través de `pip install pyspark[sql,ml,mllib]` (o `pip install pyspark[sql]` si solo desea las dependencias de SQL).



Deberá instalar Java 8 o superior en su máquina y configurar la variable de entorno `JAVA_HOME`. Ver la [documentación](#) para obtener instrucciones sobre cómo descargar e instalar Java.

Si desea ejecutar R en un modo de shell interpretativo, debe [instalar R](#) y luego corre chispaR. Para hacer computación distribuida con R, también puedes usar el proyecto de código abierto [brillante](#), creado por la comunidad R.

## Directorio y archivos de Spark

Suponemos que está ejecutando una versión del sistema operativo Linux o macOS en su computadora portátil o clúster, y todos los comandos e instrucciones en este libro serán en ese sabor. Una vez que haya terminado de descargar el tarball, cd al descargado directorio, extraiga el contenido del tarball con tar -xf spark-3.0.0-preview2-bin hadoop2.7.tgz, y cd en ese directorio y eche un vistazo a los contenidos:

```
$ cd chispa-3.0.0-preview2-bin-hadoop2.7
$ ls
LICENCIA R           LIBERAR conf      ejemplos kubernetes python yarn
DARSE CUENTA        README.md datos bin frascos licencias sbin
```

Resumamos brevemente la intención y el propósito de algunos de estos archivos y directorios. Se agregaron nuevos elementos en Spark 2.x y 3.0, y el contenido de algunos de los existentes. También se cambiaron los archivos y directorios:

### LÉAME.md

Este archivo contiene nuevas instrucciones detalladas sobre cómo usar Spark shells, construir Spark desde la fuente, ejecute ejemplos de Spark independientes, examine los enlaces a la documentación de Spark guías de mentación y configuración, y contribuye a Spark.

compartiendo

Este directorio, como sugiere el nombre, contiene la mayoría de los scripts que empleará para interactuar con Spark, incluidos los shells de Spark (spark-sql, pyspark, spark shell y sparkR).

Usaremos estos shells y ejecutables en este directorio más adelante en este capítulo para enviar una aplicación Spark independiente mediante el envío de chispa y escribir un script que cree y envíe imágenes de Docker cuando se ejecuta Spark con soporte para Kubernetes.

### sbin

La mayoría de los scripts en este directorio tienen un propósito administrativo, para iniciar y detener los componentes de Spark en el clúster en sus diversas implementaciones modos. Para obtener detalles sobre los modos de implementación, consulte la hoja de trucos en [la Tabla 1-1](#) en [Capítulo 1](#).

### kubernetes

Desde el lanzamiento de Spark 2.4, este directorio contiene Dockerfiles para crear imágenes de Docker para su distribución de Spark en un clúster de Kubernetes. También contiene un archivo que proporciona instrucciones sobre cómo construir la distribución de Spark antes construyendo sus imágenes de Docker.

## data

Este directorio se llena con archivos \*.txt que sirven como entrada para los componentes de Spark: MLlib, Structured Streaming y GraphX.

## ejemplos

Para cualquier desarrollador, dos imperativos que facilitan el viaje para aprender cualquier plataforma nueva son un montón de ejemplos de código de "cómo hacer" y una documentación completa. Spark proporciona ejemplos para Java, Python, R y Scala, y querrá emplearlos cuando aprenda el marco. Aludiremos a algunos de estos ejemplos en este capítulo y en los siguientes.

## Paso 2: Uso de Scala o PySpark Shell

Como se mencionó anteriormente, Spark viene con cuatro intérpretes ampliamente utilizados que actúan como "capas" interactivas y permiten el análisis de datos ad hoc: pyspark, spark-shell, spark sql y sparkR. En muchos sentidos, su interactividad imita shells con los que ya estará familiarizado si tiene experiencia con Python, Scala, R, SQL o shells de sistemas operativos Unix, como bash o el shell Bourne.

Estos shells se han aumentado para admitir la conexión al clúster y permitirle cargar datos distribuidos en la memoria de los trabajadores de Spark. Ya sea que esté tratando con gigabytes de datos o pequeños conjuntos de datos, los shells de Spark son propicios para aprender Spark rápidamente.

Para iniciar PySpark, haga cd en el directorio bin e inicie un shell escribiendo **pyspark**. Si ha instalado PySpark desde PyPI, bastará con escribir **pyspark**:

```
$ pyspark
Python 3.7.3 (predeterminado, 27 de marzo de 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] en darwin Escriba
"ayuda", "derechos de autor", "créditos" o "licencia" para obtener más información.
20/02/16 19:28:48 WARN NativeCodeLoader: No se puede cargar la biblioteca de hadoop nativo para su
plataforma... usando clases java integradas donde corresponda Bienvenido a
```

```
/_\_ _ _ _ /_\_ \V_ V
_`/_\`/_/_/._\_,/_/_/\_
versión 3.0.0-preview2
/_/
```

Usando Python versión 3.7.3 (predeterminado, 27 de marzo de 2019 09:23:15)

SparkSession disponible como 'chispa'. >>>

**chispa.versión** '3.0.0-preview2'

>>>

Para iniciar un shell de Spark similar con Scala, haga cd en el directorio bin y escriba **spark-shell**:

**\$ spark-shell**

```
20/05/07 19:30:26 WARN NativeCodeLoader: No se puede cargar la biblioteca de hadoop nativo
para su plataforma... usando clases java integradas donde corresponda Interfaz de usuario web del
contexto de Spark disponible en http://10.0.1.7 :4040 Contexto de Spark disponible como
'sc' (maestro = local[*], ID de aplicación = local-1581910231902)
Sesión de chispa disponible como 'chispa'.
Bienvenido a
```

```
/__/
 /_ \_ _ _ _ _ /_ \_ \_ \_
V _ \_ /' / / / . _ \_ , / / /
\_\ versión 3.0.0-preview2
 / /
```

Usando Scala versión 2.12.10 (Java HotSpot(TM) 64-Bit Server VM, Java 1.8.0\_241)

Escriba expresiones para que se evalúen.

Escriba: ayuda para obtener más

información. scala> **chispa.versión** res0:

Cadena = 3.0.0-preview2 scala>

## Uso de la máquina local

Ahora que descargó e instaló Spark en su máquina local, durante el resto de este capítulo utilizará los shells interpretativos de Spark localmente. Es decir, Spark se ejecutará en modo local.



Consulte la [Tabla 1-1](#) en el [Capítulo 1](#) para recordar qué componentes se ejecutan en modo local.

Como se señaló en el capítulo anterior, los cálculos de Spark se expresan como operaciones.

Estas operaciones luego se convierten en código de bytes basado en RDD de bajo nivel como tareas, que se distribuyen a los ejecutores de Spark para su ejecución.

Veamos un breve ejemplo en el que leemos un archivo de texto como un DataFrame, mostramos una muestra de las cadenas leídas y contamos el número total de líneas en el archivo. Este sencillo ejemplo ilustra el uso de las API estructuradas de alto nivel, que trataremos en el próximo capítulo. La operación `show(10, false)` en el DataFrame solo muestra las primeras 10 líneas sin truncar; por defecto, el indicador booleano `truncated` es verdadero. Así es como se ve esto en el shell de Scala:

```

scala> val strings = chispa.leer.text("../README.md")
cadenas: org.apache.spark.sql.DataFrame = [valor: cadena]

scala> cadenas.mostrar(10, falso)
+-----+
| valor
+-----+
|#Apache chispa |||
|Spark es un motor
|de análisis unificado para el procesamiento de datos a gran escala. Proporciona API de alto nivel
|en Scala, Java, Python y R, y un motor optimizado que admite gráficos de cálculo generales para
|el análisis de datos. También admite un amplio conjunto de herramientas de alto nivel, como Spark
|SQL para SQL y |DataFrames, MLlib para aprendizaje automático, GraphX para procesamiento de
|gráficos, | y transmisión estructurada para el procesamiento de transmisiones. | <https://
|spark.apache.org/>
+-----+
| solo se muestran las 10 primeras filas
+-----+

```

```

scala> cuerdas.contar()
res2: Largo = 109
scala>

```

Bastante sencillo. Veamos un ejemplo similar usando el shell interpretativo de Python, chispa:

```

$ chispa
Python 3.7.3 (predeterminado, 27 de marzo de 2019, 09:23:15)
[Clang 10.0.1 (clang-1001.0.46.3)] en darwin
Escriba "ayuda", "derechos de autor", "créditos" o "licencia" para obtener más información.
ADVERTENCIA: Se ha producido una operación de acceso reflexivo ilegal
ADVERTENCIA: Acceso reflexivo ilegal por org.apache.spark.unsafe.Platform ADVERTENCIA:
Use --illegal-access=warn para habilitar advertencias de más operaciones ilegales de acceso
reflexivo
ADVERTENCIA: todas las operaciones de acceso ilegal serán denegadas en una versión futura
20/01/10 11:28:29 WARN NativeCodeLoader: no se puede cargar la biblioteca de hadoop nativa para su
plataforma... usando clases java integradas donde corresponda
Uso del perfil log4j predeterminado de Spark: org/apache/spark/log4j-defaults.properties
Establecer el nivel de registro predeterminado en "ADVERTIR".
Para ajustar el nivel de registro, use sc.setLogLevel(newLevel). Para SparkR, use
setLogLevel(newLevel).
Bienvenido a

```

Usando Python versión 3.7.3 (predeterminado, 27 de marzo de 2019 09:23:15)  
 SparkSession disponible como 'chispa'.  
 >>> cadenas = chispa.leer.texto("../README.md")

```
>>> cadenas.mostrar(10, truncar=False)
+-----+
| valor
+-----+
| #Apache chispa ||
| Spark es un motor
| de análisis unificado para el procesamiento de datos a gran escala. Proporciona API de alto nivel
| en Scala, Java, Python y R, y un motor optimizado que admite gráficos de cálculo generales para
| el análisis de datos. También admite un amplio conjunto de herramientas de nivel superior, como
| Spark SQL para SQL y DataFrames, MLlib para aprendizaje automático, GraphX para procesamiento
| de gráficos y transmisión estructurada para procesamiento de secuencias. | |<https://spark.apache.org/
>

+-----+
| solo se muestran las 10 primeras filas

>>> cadenas.contar()
109
>>>
```

Para salir de cualquiera de los shells de Spark, presione Ctrl-D. Como puedes ver, esta rápida interactividad con Spark shells es propicio no solo para el aprendizaje rápido sino también para la creación rápida de prototipos, también.

En los ejemplos anteriores, observe la sintaxis de API y la paridad de firmas en ambos Scala y Phyton. A lo largo de la evolución de Spark desde 1.x, ese ha sido uno (entre muchos) de las mejoras duraderas.

También tenga en cuenta que usamos las API estructuradas de alto nivel para leer un archivo de texto en un Spark DataFrame en lugar de un RDD. A lo largo del libro, nos centraremos más en estos API estructuradas; desde Spark 2.x, los RDD ahora se asignan a API de bajo nivel.



Cada cálculo expresado en API estructuradas de alto nivel es descompuesto en operaciones RDD optimizadas y generadas de bajo nivel ciones y luego convertido en bytecode Scala para los ejecutores JVM. Este código de operación RDD generado no es accesible para usuarios, ni es lo mismo que las API de RDD orientadas al usuario.

## Paso 3: comprender los conceptos de la aplicación Spark

Ahora que ha descargado Spark, instálelo en su computadora portátil de forma independiente modo, lanzó un shell Spark y ejecutó algunos ejemplos de código corto de forma interactiva, estás listo para dar el último paso.

Para comprender lo que sucede debajo del capó con nuestro código de muestra, deberá estar familiarizado con algunos de los conceptos clave de una aplicación Spark y cómo se

transformado y ejecutado como tareas a través de los ejecutores de Spark. Comenzaremos definiendo algunos términos importantes:

#### Aplicación Un

programa de usuario creado en Spark usando sus API. Consiste en un programa controlador y ejecutores en el clúster.

#### SparkSession

Un objeto que proporciona un punto de entrada para interactuar con la funcionalidad subyacente de Spark y permite programar Spark con sus API. En un shell de Spark interactivo, el controlador de Spark crea una instancia de SparkSession para usted, mientras que en una aplicación de Spark, usted mismo crea un objeto de SparkSession .

#### Trabajo

Un cómputo paralelo que consta de varias tareas que se generan en respuesta a una acción de Spark (p. ej., guardar(), recopilar()).

#### Etapa

Cada trabajo se divide en conjuntos más pequeños de tareas llamadas etapas que dependen unas de otras.

#### Tarea

Una única unidad de trabajo o ejecución que se enviará a un ejecutor de Spark.

Profundicemos un poco más en estos conceptos.

#### Aplicación Spark y SparkSession

En el núcleo de cada aplicación Spark se encuentra el programa controlador Spark, que crea un objeto SparkSession . Cuando trabaja con un shell Spark, el controlador es parte del shell y el objeto SparkSession (accesible a través de la variable spark) se crea para usted, como vio en los ejemplos anteriores cuando inició los shells.

En esos ejemplos, debido a que lanzó Spark Shell localmente en su computadora portátil, todas las operaciones se ejecutaron localmente, en una sola JVM. Pero puede lanzar un Spark Shell con la misma facilidad para analizar datos en paralelo en un clúster como en modo local. Los comandos spark-shell --help o pyspark --help le mostrarán cómo conectarse al administrador de clústeres de Spark. [La figura 2-2](#) muestra cómo se ejecuta Spark en un clúster una vez que haya hecho esto.

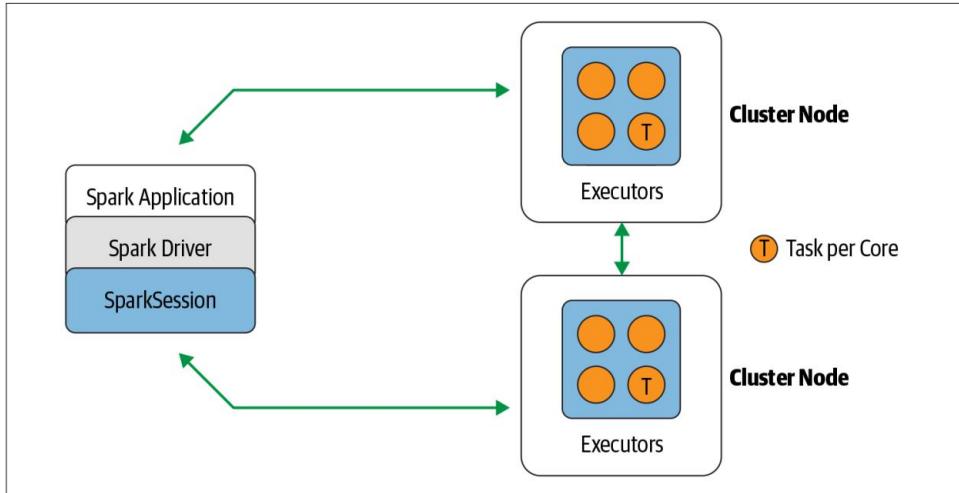


Figura 2-2. Los componentes de Spark se comunican a través del controlador Spark en la arquitectura distribuida de Spark

Una vez que tenga una **SparkSession**, puede [programar Spark usando las API](#) para realizar operaciones de Spark.

#### Trabajos de

**Spark** Durante las sesiones interactivas con shells de Spark, el controlador convierte su aplicación Spark en uno o más trabajos de Spark ([Figura 2-3](#)). Luego transforma cada trabajo en un DAG. Este, en esencia, es el plan de ejecución de Spark, donde cada nodo dentro de un DAG podría ser una o varias etapas de Spark.

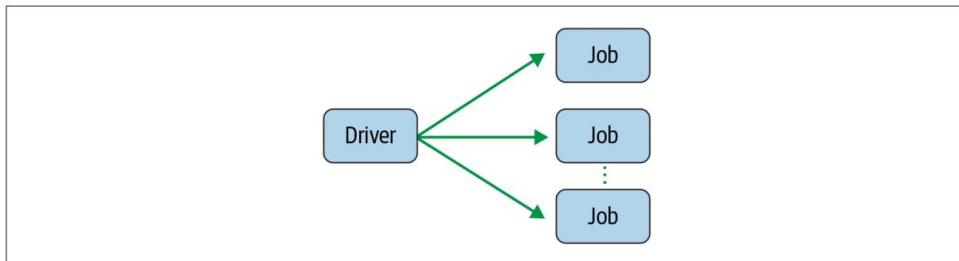


Figura 2-3. Controlador Spark creando uno o más trabajos Spark

## Spark Stages

Como parte de los nodos DAG, las etapas se crean en función de las operaciones que se pueden realizar en serie o en paralelo ([Figura 2-4](#)). No todas las operaciones de Spark pueden ocurrir en una sola etapa, por lo que pueden dividirse en varias etapas. A menudo, las etapas se delinean en los límites de cálculo del operador, donde dictan la transferencia de datos entre los ejecutores de Spark.

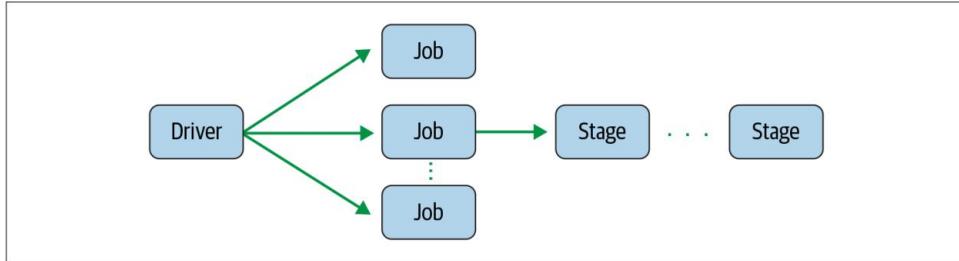


Figura 2-4. Spark job creando una o más etapas

## Tareas de

**Spark** Cada etapa se compone de tareas de Spark (una unidad de ejecución), que luego se federan en cada ejecutor de Spark; cada tarea se asigna a un solo núcleo y funciona en una sola partición de datos ([Figura 2-5](#)). Como tal, un ejecutor con 16 núcleos puede tener 16 o más tareas trabajando en 16 o más particiones en paralelo, ¡lo que hace que la ejecución de las tareas de Spark sea extremadamente paralela!

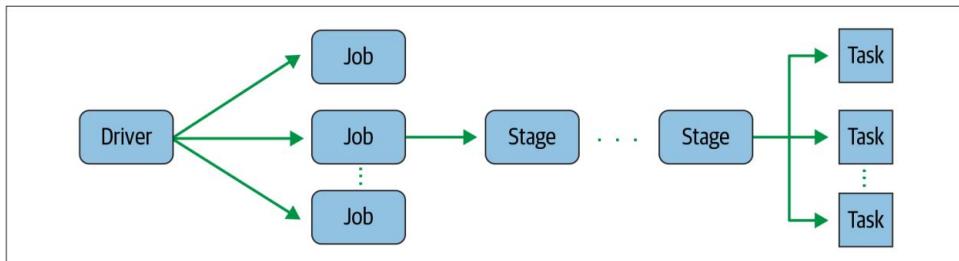


Figura 2-5. Etapa Spark creando una o más tareas para ser distribuidas a los ejecutores

## Transformaciones, acciones y evaluación perezosa

Las operaciones Spark en datos distribuidos se pueden clasificar en dos tipos: transformaciones y acciones. Las transformaciones, como su nombre indica, transforman un Spark DataFrame en un nuevo DataFrame sin alterar los datos originales, otorgándole la propiedad de inmutabilidad. Dicho de otra manera, una operación como `select()` o `filter()` no cambiará el DataFrame original; en su lugar, devolverá los resultados transformados de la operación como un nuevo DataFrame.

Todas las transformaciones se evalúan perezosamente. Es decir, sus resultados no se computan inmediatamente, sino que se registran o recuerdan como un linaje. Un linaje registrado permite a Spark, en un momento posterior de su plan de ejecución, reorganizar ciertas transformaciones, fusionarlas u optimizar las transformaciones en etapas para una ejecución más eficiente. La evaluación diferida es la estrategia de Spark para retrasar la ejecución hasta que se invoca una acción o se "tocan" los datos (se leen o se escriben en el disco).

Una acción desencadena la evaluación perezosa de todas las transformaciones registradas. En la [figura 2-6](#), todas las transformaciones T se registran hasta que se invoca la acción A. Cada transformación T produce un nuevo DataFrame.

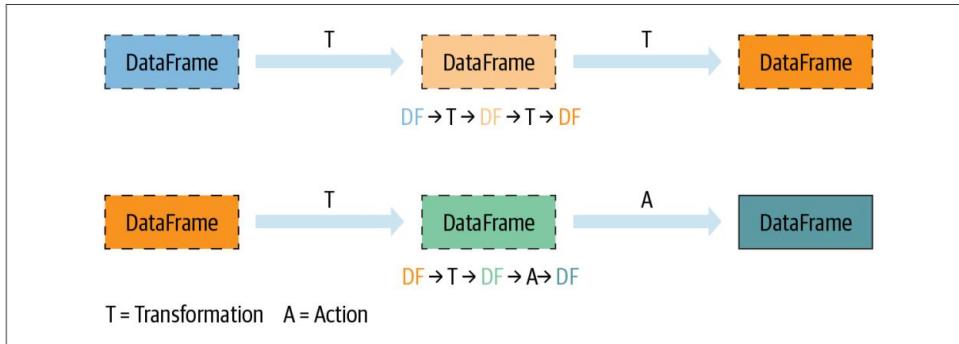


Figura 2-6. Transformaciones perezosas y acciones ansiosas

Si bien la evaluación perezosa permite que Spark optimice sus consultas al observar sus transformaciones encadenadas, el linaje y la inmutabilidad de los datos brindan tolerancia a fallas.

Debido a que Spark registra cada transformación en su linaje y los DataFrames son inmutables entre transformaciones, puede reproducir su estado original simplemente reproduciendo el linaje registrado, lo que le brinda resiliencia en caso de fallas.

[La Tabla 2-1](#) enumera algunos ejemplos de transformaciones y acciones.

Tabla 2-1. Transformaciones y acciones como operaciones de Spark

Acciones de Transformaciones	
ordenar por ()	show()
agrupar por()	tomar()
filtrar()	contar()
seleccionar	recoger ()
() unirse ()	guardar ()

Las acciones y transformaciones contribuyen a un plan de consulta de Spark, que trataremos en el próximo capítulo. Nada en un plan de consulta se ejecuta hasta que se invoca una acción. El siguiente ejemplo, que se muestra tanto en Python como en Scala, tiene dos transformaciones, `read()` y `filter()`, y una acción, `count()`. La acción es lo que

desencadena la ejecución de todas las transformaciones registradas como parte del plan de ejecución de la consulta. En este ejemplo, no sucede nada hasta que se ejecuta filtered.count() en el shell:

```
# En Python
>>> cadenas = spark.read.text("../README.md")
>>> filtrado = cadenas.filtro(cadenas.valor.contains("Spark")) >>>
filtrado.cuenta() 20

// En Scala
scala> import org.apache.spark.sql.functions._
scala> val strings = spark.read.text("../README.md") scala>
val filtered = strings.filter(col(" value").contains("Spark")) scala> filtered.count()
res5: Long = 20
```

## Transformaciones estrechas y anchas

Como se señaló, las transformaciones son operaciones que Spark evalúa con pereza. Una gran ventaja del esquema de evaluación perezoso es que Spark puede inspeccionar su consulta computacional y determinar cómo puede optimizarla. Esta optimización se puede realizar uniendo o canalizando algunas operaciones y asignándolas a una etapa, o dividiéndolas en etapas determinando qué operaciones requieren una mezcla o intercambio de datos entre clústeres.

Las transformaciones se pueden clasificar en dependencias estrechas o dependencias amplias. Cualquier transformación en la que se pueda calcular una única partición de salida a partir de una única partición de entrada es una transformación estrecha. Por ejemplo, en el fragmento de código anterior, filter() y contains() representan transformaciones limitadas porque pueden operar en una sola partición y producir la partición de salida resultante sin ningún intercambio de datos.

Sin embargo, groupBy() o orderBy() indican a Spark que realice transformaciones amplias, donde los datos de otras particiones se leen, combinan y escriben en el disco. Dado que cada partición tendrá su propio conteo de la palabra que contiene la palabra "Spark" en su fila de datos, un conteo (groupBy()) forzará una mezcla de datos de cada una de las particiones del ejecutor en el clúster. En esta transformación, orderBy() requiere la salida de otras particiones para calcular la agregación final.

La figura 2-7 ilustra los dos tipos de dependencias.

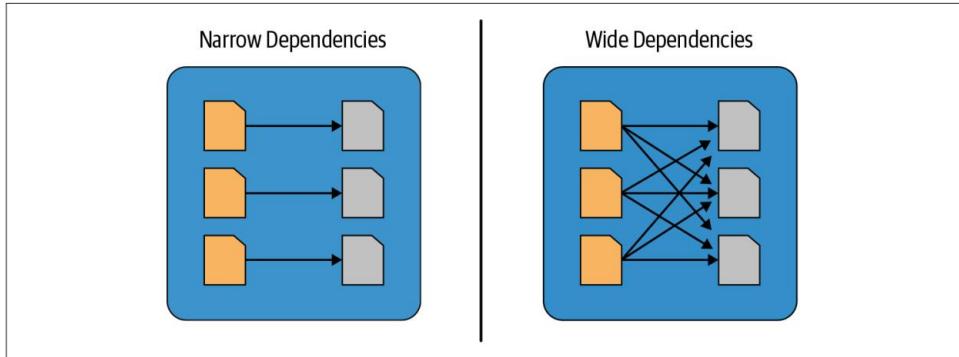


Figura 2-7. Transformaciones estrechas versus amplias

#### La interfaz de usuario de Spark

Spark incluye una [interfaz gráfica de usuario](#) que puede usar para inspeccionar o monitorear aplicaciones Spark en sus diversas etapas de descomposición, es decir, trabajos, etapas y tareas.

Según cómo se implemente Spark, el controlador inicia una interfaz de usuario web, que se ejecuta de manera predeterminada en el puerto 4040, donde puede ver métricas y detalles como:

- Una lista de etapas y tareas del programador
- Un resumen de los tamaños de RDD y el uso de la memoria
- Información sobre el medio ambiente
- Información sobre los ejecutores en ejecución • Todas las consultas Spark SQL

En modo local, puede acceder a esta interfaz en <http://<localhost>:4040> en un navegador web.



Cuando inicia Spark-Shell, parte del resultado muestra la URL del host local para acceder al puerto 4040.

Inspeccionemos cómo el ejemplo de Python de la sección anterior se traduce en trabajos, etapas y tareas. Para ver cómo se ve el DAG, haga clic en "Visualización de DAG" en la interfaz de usuario web. Como muestra la [Figura 2-8](#), el conductor creó un solo trabajo y una sola etapa.

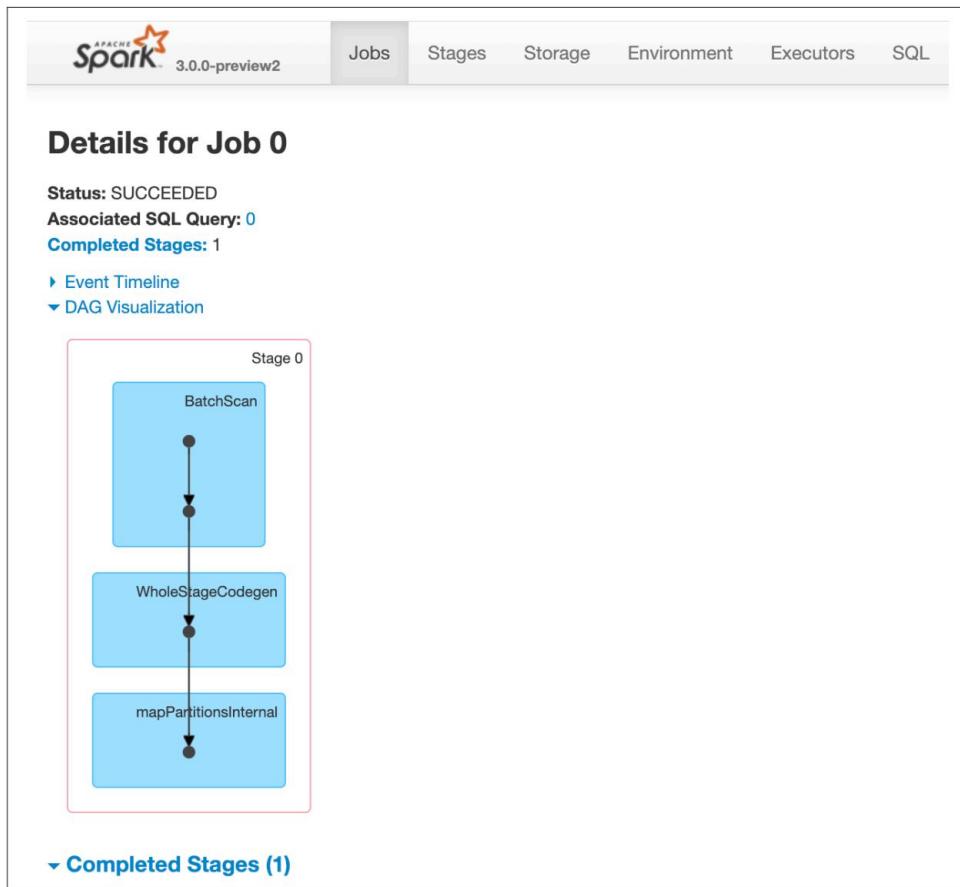


Figura 2-8. El DAG para nuestro ejemplo simple de Python

Tenga en cuenta que no hay intercambio, donde los datos se intercambian entre ejecutores, requerido porque solo hay una etapa. Las operaciones individuales de la etapa se muestran en cuadros azules.

La etapa 0 se compone de una tarea. Si tiene varias tareas, se ejecutarán en paralelo. Puede ver los detalles de cada etapa en la pestaña Etapas, como se muestra en la Figura 2-9.

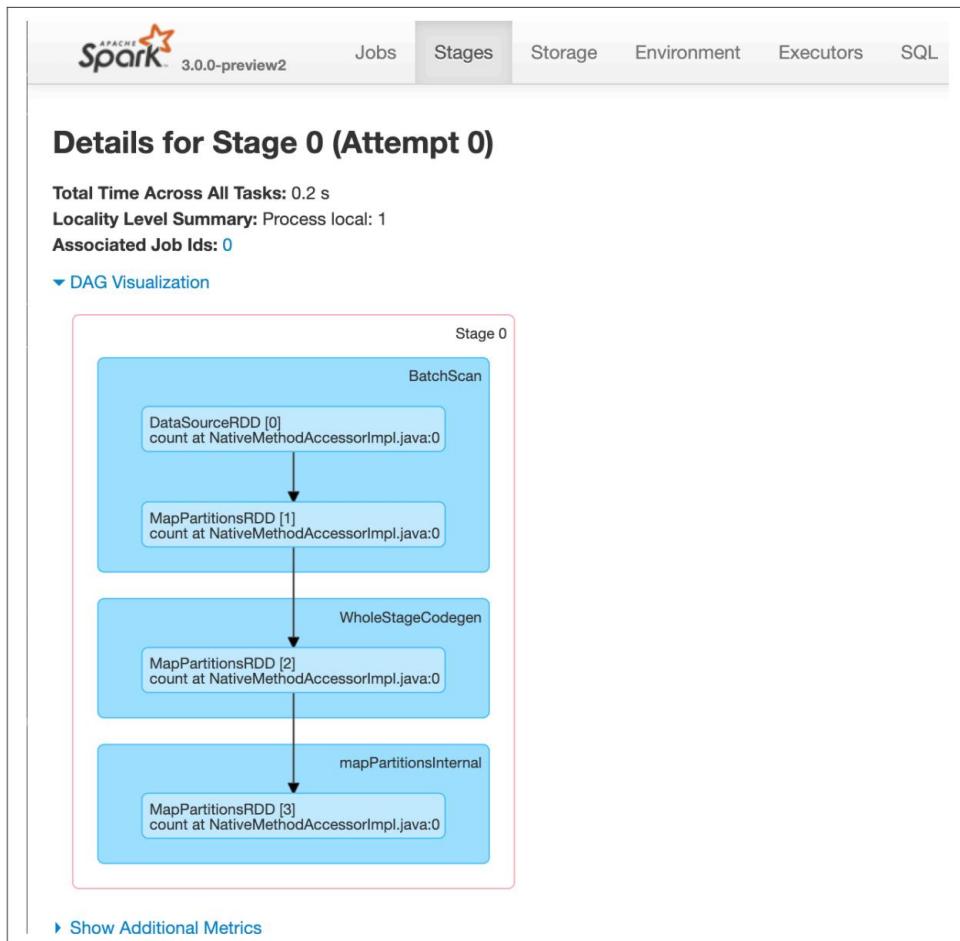


Figura 2-9. Detalles de la etapa 0

Cubriremos la interfaz de usuario de Spark con más detalle en el [Capítulo 7](#). Por ahora, solo tenga en cuenta que la interfaz de usuario proporciona una lente microscópica en el funcionamiento interno de Spark como una herramienta para depurar e inspeccionar.

## Databricks Community Edition

Databricks es una empresa que ofrece una plataforma Apache Spark administrada en la nube.

Además de usar su máquina local para ejecutar Spark en modo local, puede probar algunos de los ejemplos de este y otros capítulos con la edición gratuita de Databricks Community ([Figura 2-10](#)). Como herramienta de aprendizaje para Apache Spark, Community Edition tiene muchos tutoriales y ejemplos dignos de mención. Además de escribir sus propios cuadernos en Python, R, Scala o SQL, también puede importar otros cuadernos, incluidos los cuadernos de Jupyter.

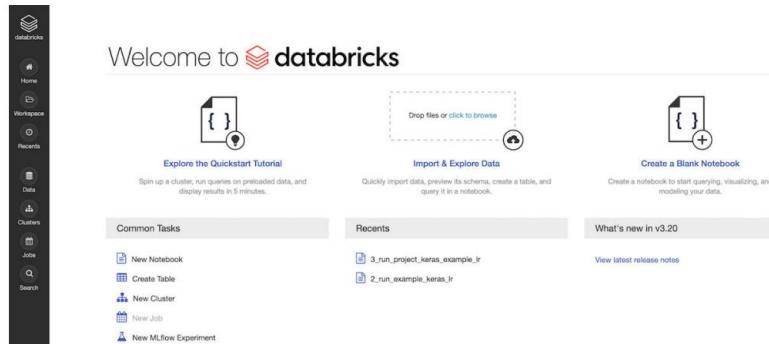


Figura 2-10. Edición de la comunidad de ladrillos de datos

Para obtener una cuenta, vaya a <https://databricks.com/try> y siga las instrucciones para probar la Community Edition de forma gratuita. Una vez registrado, puede importar los cuadernos para este libro desde su repositorio de [GitHub](#).

## Su primera aplicación independiente

Para facilitar el aprendizaje y la exploración, la distribución de Spark viene con un conjunto de aplicaciones de muestra para cada uno de los componentes de Spark. Le invitamos a examinar el directorio de ejemplos en su ubicación de instalación para tener una idea de lo que está disponible.

Desde el directorio de instalación de su máquina local, puede ejecutar uno de los varios programas de ejemplo de Java o Scala que se proporcionan mediante el comando bin/run example `<class> [params]`. Por ejemplo:

```
$ ./bin/ejecutar-ejemplo JavaWordCount README.md
```

Esto arrojará mensajes INFO en su consola junto con una lista de cada palabra en el archivo README.md y su conteo (contar palabras es el “Hola, mundo” de la computación distribuida).

## Contando M&Ms para el Monstruo de las Galletas

**Galletas** En el ejemplo anterior, contamos palabras en un archivo. Si el archivo fuera enorme, se distribuiría en un clúster dividido en pequeños fragmentos de datos, y nuestro programa Spark distribuiría la tarea de contar cada palabra en cada partición y nos devolvería el recuento final agregado. Pero ese ejemplo se ha convertido en un cliché.

Resolvamos un problema similar, pero con un conjunto de datos más grande y usando más funciones de distribución de Spark y las API de DataFrame. Cubriremos las API utilizadas en este programa en capítulos posteriores, pero por ahora tenga paciencia con nosotros.

Entre los autores de este libro se encuentra una científica de datos a la que le encanta hornear galletas con M&M, y recompensa a sus estudiantes en los estados de EE. UU., donde frecuentemente imparte cursos de aprendizaje automático y ciencia de datos con lotes de esas galletas. Pero ella está basada en datos, obviamente, y quiere asegurarse de obtener los colores correctos de M&M en las galletas para los estudiantes en los diferentes estados ([Figura 2-11](#)).



Figura 2-11. Distribución de M&Ms por color (fuente: <https://oreil.ly/mhWIT>)

Escribamos un programa Spark que lea un archivo con más de 100,000 entradas (donde cada fila o línea tiene un `<estado, mnm_color, conteo>`) y calcule y agregue los conteos para cada color y estado. Estos conteos agregados nos dicen los colores de M&M preferidos por los estudiantes en cada estado. La lista completa de Python se proporciona en el [Ejemplo 2-1](#).

## Ejemplo 2-1. Contar y agregar M&Ms (versión de Python)

```
# Importar las bibliotecas necesarias.
# Ya que estamos usando Python, importe SparkSession y las funciones relacionadas # desde el
módulo PySpark. sistema de importación

desde pyspark.sql import SparkSession desde
pyspark.sql.functions import count

if __name__ == "__main__":
    if len(sys.argv) != 2: print("Uso:
        mnmcount <archivo>", file=sys.stderr) sys.exit(-1)

    # Cree una SparkSession utilizando las API de SparkSession.
    # Si no existe, cree una instancia. # Solo puede haber una SparkSession
    por JVM. chispa =
    (SparkSession .builder .appName("PythonMnMCount") .getOrCreate())

    # Obtenga el nombre de archivo del conjunto de datos de M&M de los argumentos de la línea de
comandos mnm_file = sys.argv[1]
    # Lea el archivo en un Spark DataFrame utilizando el formato CSV #
    infiriendo el esquema y especificando que el archivo # contiene un
encabezado, que proporciona nombres de columna para campos separados por
comas #. mnm_df = (spark.read.format("csv") .option("header",
"true") .option("inferSchema", "true") .load(mnm_file))

    # Usamos las API de alto nivel de DataFrame. Tenga en
cuenta que no usamos RDD en absoluto. Debido a que algunas de las
funciones # de Spark devuelven el mismo objeto, podemos encadenar llamadas a funciones.
    # 1. Seleccione del DataFrame los campos "Estado", "Color" y "Recuento"
    # 2. Ya que queremos agrupar cada estado y su conteo de colores M&M, # usamos
groupBy()
    # 3. Recuentos agregados de todos los colores y groupBy() State y Color # 4 orderBy()
en orden descendente count_mnm_df = (mnm_df .select("State", "Color",
"Count") .groupBy("State", "Color") .agg(recuento("Recuento").alias("Total")) .orderBy("Total",
ascendente=False))

    # Muestra las agregaciones resultantes para todos los estados y colores; # un recuento
total de cada color por estado.
    # Tenga en cuenta que show() es una acción que activará la # consulta
anterior para que se ejecute. count_mnm_df.show(n=60, truncar=False)
print("Filas totales = %d" % (count_mnm_df.count()))
```

```
# Mientras que el código anterior agregó y contó para todos los #
estados, ¿qué pasa si solo queremos ver los datos para # un solo
estado, por ejemplo, CA?
# 1. Seleccione de todas las filas en el DataFrame #
2. Filtre solo el estado de CA # 3. groupBy() Estado y
color como hicimos anteriormente # 4. Agregue los
recuentos para cada color # 5. orderBy() en orden
descendente # Buscar el conteo agregado para California
filtrando ca_count_mnm_df = (mnm_df .select("State", "Color",
"Count") .where(mnm_df.State == "CA") .groupBy("State", "Color") .
agg(recuento("Recuento").alias("Total")) .orderBy("Total",
ascendente=False))
```

```
# Muestre la agregación resultante para California.
# Como arriba, show() es una acción que activará la ejecución de todo el cálculo.
ca_count_mnm_df.show(n=10, truncado=False)
```

```
# Detener la SparkSession
spark.stop()
```

Puede ingresar este código en un archivo de Python llamado mnmcount.py usando su editor favorito, descargar el archivo mnn\_dataset.csv del repositorio de [GitHub de este libro](#), y envíelo como un trabajo de Spark utilizando el script de envío de chispa en el directorio bin de la instalación. Establezca su variable de entorno SPARK\_HOME en el directorio de nivel raíz donde instaló Spark en su máquina local.



El código anterior usa la API de DataFrame, que se lee como consultas DSL de alto nivel. Cubriremos esto y las otras API en el próximo capítulo; por ahora, tenga en cuenta la claridad y la simplicidad con las que puede indicarle a Spark qué hacer, no cómo hacerlo, a diferencia de la API de RDD. ¡Cosas interesantes!

Para evitar que se impriman mensajes de INFORMACIÓN detallados en la consola, copie el archivo log4j.properties.template en log4j.properties y establezca log4j.rootCategory=WARN en el archivo conf/log4j.properties.

Envíemos nuestro primer trabajo de Spark usando las API de Python (para obtener una explicación de lo que hace el código, lea los comentarios en línea en el [Ejemplo 2-1](#)):

```
$SPARK_HOME/bin/spark-submit mnmcount.py data/mnm_dataset.csv
```

```
+-----+
|Estado|Color |Total|
+-----+
|CA |Amarillo|1807 |
| WA | Verde | 1779 |
|O |Naranja|1743 |
```

TX	Verde	1737
TX	Rojo	1725
CA	Verde	1723
CO	Amarillo	1721
CA	Marrón	1718
CO	Verde	1713
Nevada	Naranja	1712
TX	Amarillo	1703
NV	Verde	1698
AZ	Marrón	1698
CO	Azul	1695
WY	Verde	1695
NM	Rojo	1690
AZ	naranja	1689
NM	Amarillo	1688
NM	Marrón	1687
UT	Naranja	1684
NM	Verde	1682
UT	Rojo	1680
AZ	Verde	1676
NV	Amarillo	1675
NV	Azul	1673
Washington	Rojo	1671
wy	Rojo	1670
WA	Marrón	1669
NM	Naranja	1665
WY	Azul	1664
WA	Amarillo	1663
WA	Naranja	1658
NV	Marrón	1657
CA	Naranja	1657
CA	Rojo	1656
CO	Marrón	1656
UT	Azul	1655
AZ	Amarillo	1654
TX	Naranja	1652
Arizona	Rojo	1648
O	Azul	1646
UT	Amarillo	1645
O	Rojo	1645
CO	Naranja	1642
TX	Marrón	1641
NM	Azul	1638
AZ	Azul	1636
O	Verde	1634
UT	Marrón	1631
WY	Amarillo	1626
WA	Azul	1625
CO	Rojo	1624
O	Marrón	1621
TX	Azul	1614
O	Amarillo	1614

```
|NV    |Rojo |1610 |
|CA |Azul |1603 |
|WY |Naranja|1595 |
|UT |Verde |1591 |
|WY |Marrón |1532 |
+---+-----+
```

Filas totales = 60

```
+-----+
|Estado|Color |Total|
+-----+
|CA |Amarillo|1807 |
|CA |Verde |1723 |
|CA |Marrón |1718 |
| CA | Naranja | 1657 |
| CA    |Rojo |1656 |
|CA |Azul |1603 |
+-----+
```

Primero vemos todas las agregaciones para cada color de M&M para cada estado, seguidas por aquellas solo para CA (donde el color preferido es el amarillo).

¿Qué pasa si quieres usar una versión Scala de este mismo programa Spark? Las API son similares; en Spark, la paridad se conserva bien en todos los idiomas admitidos, con pequeñas diferencias de sintaxis. El [ejemplo 2-2](#) es la versión Scala del programa. Eche un vistazo y en la siguiente sección le mostraremos cómo compilar y ejecutar la aplicación.

Ejemplo 2-2. Contar y agregar M&Ms (versión Scala)

```
paquete main.scala.chapter2

importar org.apache.spark.sql.SparkSession
importar org.apache.spark.sql.functions._

/***
 * Uso: MnMcount <mnm_file_dataset> */
object MnMcount { def main(args:
Array[String]) { val spark =  
  
SparkSession .builder .appName("MnMCount") .getOrCreate()  
  
if (args.longitud < 1) {
  print("Uso: MnMcount <mnm_file_dataset>") sys.exit(1)
}
// Obtener el nombre de archivo del conjunto de
datos de M&M val mnmFile = args(0)
```

```

// Lee el archivo en un Spark DataFrame val
mnmDF = spark.read.format("csv") .option("header",
    "true") .option("inferSchema", "true") .load(mnmFile)

// Recuentos agregados de todos los colores y groupBy() State and Color //
// orderBy() en orden descendente val countMnMDF = mnmDF
    .select("Estado", "Color",
    "Recuento") .groupBy("Estado",
        "Color") .agg(recuento("Recuento").alias("Total")) .orderBy(desc("Total "))
// Mostrar las agregaciones resultantes para todos los estados y colores
countMnMDF.show(60) println(s" Filas totales = ${countMnMDF.count()}") println()

// Encuentre los conteos agregados para California filtrando val
caCountMnMDF = mnmDF
    .select("Estado", "Color",
    "Recuento") .where(col("Estado") ===
    "CA") .groupBy("Estado",
        "Color") .agg(recuento("Recuento" ).alias("Total")) .orderBy(desc("Total "))
// Mostrar las agregaciones resultantes para California
caCountMnMDF.show(10)
// Detener SparkSession
spark.stop() }

```

## Creación de aplicaciones independientes en Scala

Ahora le mostraremos cómo crear su primer programa Scala Spark, utilizando **Scala Build Tool (sbt)**.



Debido a que Python es un lenguaje interpretado y no existe el paso de compilar primero (aunque es posible compilar su código de Python en bytecode en .pyc), no entraremos en este paso aquí. Para obtener detalles sobre cómo usar Maven para crear programas Java Spark, lo remitimos a la [guía](#) en el sitio web de Apache Spark. Por brevedad en este libro, cubrimos ejemplos principalmente en Python y Scala.

build.sbt es el archivo de especificación que, como un archivo MAKE, describe e indica al compilador de Scala que cree sus tareas relacionadas con Scala, como archivos jar, paquetes, qué dependencias resolver y dónde buscarlas. En nuestro caso, tenemos un archivo sbt simple para nuestro código M&M ([Ejemplo 2-3](#)).

Ejemplo 2-3. archivo de compilación sbt

```
// Nombre del nombre del paquete := "main/scala/chapter2"
// Versión de nuestro paquete
version := "1.0"
// Versión de Scala
scalaVersion := "2.12.10"
// Dependencias de la biblioteca Spark
libraryDependencies ++=
  Seq( "org.apache.spark" %% "spark-core" % "3.0.0-preview2",
    "org.apache.spark" %% "spark-sql" % "3.0.0-vista previa2" )
```

Suponiendo que tiene el [kit de desarrollo de Java \(JDK\)](#) y sbt instalado y Conjunto JAVA\_HOME y SPARK\_HOME , con un solo comando, puede construir su aplicación Spark:

```
$ sbt clean package
[info] Archivo actualizado /Users/julesdamji/gits/LearningSparkV2/chapter2/scala/project/
build.properties: establecer sbt.version a 1.2.8 [info] Cargando definición de proyecto desde /
Users/julesdamji/gits/ LearningSparkV2/ chapter2/scala/project [info] Actualizando [info] Actualización
finalizada.

...
[información] Compilando 1 código fuente de Scala en /Users/julesdamji/gits/LearningSparkV2/
chapter2/scala/target/scala-2.12/classes ... [información] Compilación finalizada. [info]
Empaqueado /Users/julesdamji/gits/LearningSparkV2/chapter2/scala/target/ scala-2.12/main-
scala-chapter2_2.12-1.0.jar ... [info] Empaqueado terminado. [éxito] Tiempo total: 6 s, completado el
11 de enero de 2020, 4:11:02 p. m.
```

Después de una compilación exitosa, puede ejecutar la versión Scala del ejemplo de conteo de M&M de la siguiente manera:

```
$SPARK_HOME/bin/spark-submit --class main.scala.chapter2.MnMcount \ jars/
main-scala-chapter2_2.12-1.0.jar data/mnm_dataset.csv
...
...
20/01/11 16:00:48 INFO TaskSchedulerImpl: Eliminación de todas las tareas en ejecución en la
etapa 4: Etapa finalizada 20/01/11 16:00:48 INFO DAGScheduler: Trabajo 4 terminado: mostrar
en MnMcount.scala:49, tomó 0.264579 s

+---+---+---+
|Estado| Color|Total|
+---+---+---+
| CA|Amarillo| 1807| |
| CA| Verde| 1723| |
| CA|Marrón| 1718| |
| CA |Naranja | 1657|
```

```
CA | rojo | 1656 |
.. CA| azul| 1603|
+-----+-----+
```

El resultado es el mismo que para la ejecución de Python. ¡Intentalo!

Ahí lo tiene: nuestro autor científico de datos estará más que feliz de usar estos datos para decidir qué colores de M&M usar en las galletas que hornea para sus clases en cualquiera de los estados en los que enseña.

## Resumen

En este capítulo, cubrimos los tres pasos simples que debe seguir para comenzar con Apache Spark: descargar el marco, familiarizarse con el shell interactivo Scala o PySpark y familiarizarse con los conceptos y términos de la aplicación Spark de alto nivel. . Brindamos una descripción general rápida del proceso mediante el cual puede usar transformaciones y acciones para escribir una aplicación Spark, y presentamos brevemente el uso de la interfaz de usuario de Spark para examinar los trabajos, las etapas y las tareas creadas.

Finalmente, a través de un breve ejemplo, le mostramos cómo puede usar las API estructuradas de alto nivel para decirle a Spark qué hacer, lo que nos lleva al siguiente capítulo, donde examinamos esas API con más detalle.

## CAPÍTULO 3

# API estructuradas de Apache Spark

En este capítulo, exploraremos las principales motivaciones detrás de agregar estructura a Apache Spark, cómo esas motivaciones llevaron a la creación de API de alto nivel (marcos de datos y conjuntos de datos) y su unificación en Spark 2.x en todos sus componentes. También veremos el motor Spark SQL que sustenta estas API estructuradas de alto nivel.

Cuando **Spark SQL** se introdujo por primera vez en las primeras versiones de Spark 1.x, seguido de **DataFrames** como sucesor de **SchemaRDD** en Spark 1.3, vimos por primera vez la estructura en Spark. Spark SQL introdujo funciones operativas expresivas de alto nivel, imitando la sintaxis similar a SQL, y DataFrames, que sentó las bases para una mayor estructura en versiones posteriores, allanó el camino hacia operaciones de alto rendimiento en las consultas computacionales de Spark.

Pero antes de hablar sobre las API estructuradas más nuevas, echemos un vistazo breve a lo que es no tener estructura en Spark al echar un vistazo al modelo de API de programación RDD simple.

## Spark: ¿Qué hay debajo de un RDD?

El **RDD** es la abstracción más básica en Spark. Hay tres características vitales asociadas con un RDD:

- Dependencias •
- Particiones (con alguna información de localidad)
- Función de cálculo: Partición => Iterador[T]

Los tres son parte integral del modelo API de programación RDD simple sobre el cual se construye toda la funcionalidad de nivel superior. Primero, se requiere una lista de dependencias que indique a Spark cómo se construye un RDD con sus entradas. Cuando sea necesario para reproducir resultados, Spark puede recrear un RDD a partir de estas dependencias y replicar operaciones en él. Esta característica le da resiliencia a los RDD.

En segundo lugar, las particiones brindan a Spark la capacidad de dividir el trabajo para parallelizar el cálculo en las particiones entre los ejecutores. En algunos casos, por ejemplo, al leer de HDFS, Spark usará la información de la localidad para enviar el trabajo a los ejecutores cercanos a los datos. De esa manera se transmiten menos datos a través de la red.

Y finalmente, un RDD tiene una función de cómputo que produce un Iterator[T] para los datos que se almacenarán en el RDD.

¡Simple y elegante! Sin embargo, hay un par de problemas con este modelo original. Por un lado, la función de cómputo (o computación) es opaca para Spark. Es decir, Spark no sabe lo que está haciendo en la función de cómputo. Ya sea que esté realizando una unión, un filtro, una selección o una agregación, Spark solo lo ve como una expresión lambda. Otro problema es que el tipo de datos Iterator [T] también es opaco para Python RDD; Spark solo sabe que es un objeto genérico en Python.

Además, debido a que no puede inspeccionar el cálculo o la expresión en la función, Spark no tiene forma de optimizar la expresión: no comprende su intención. Y finalmente, Spark no tiene conocimiento del tipo de datos específico en T. Para Spark es un objeto opaco; no tiene idea si está accediendo a una columna de cierto tipo dentro de un objeto. Por lo tanto, todo lo que Spark puede hacer es serializar el objeto opaco como una serie de bytes, sin usar ninguna técnica de compresión de datos.

Esta opacidad claramente obstaculiza la capacidad de Spark para reorganizar su cálculo en un plan de consulta eficiente. Entonces, ¿cuál es la solución?

## Chispa Estructurante

Spark 2.x introdujo algunos esquemas clave para estructurar Spark. Una es expresar los cálculos mediante el uso de patrones comunes que se encuentran en el análisis de datos. Estos patrones se expresan como operaciones de alto nivel, como filtrar, seleccionar, contar, agregar, promediar y agrupar. Esto proporciona mayor claridad y simplicidad.

Esta especificidad se reduce aún más mediante el uso de un conjunto de operadores comunes en un DSL. A través de un conjunto de operaciones en DSL, disponibles como API en los lenguajes compatibles con Spark (Java, Python, Spark, R y SQL), estos operadores le permiten decirle a Spark qué desea calcular con sus datos y, como resultado, puede construir un plan de consulta eficiente para su ejecución.

Y el esquema final de orden y estructura es permitirle organizar sus datos en un formato tabular, como una tabla SQL o una hoja de cálculo, con tipos de datos estructurados admitidos (que trataremos en breve).

Pero, ¿para qué sirve toda esta estructura?

## Méritos y beneficios clave

La estructura ofrece una serie de beneficios, incluido un mejor rendimiento y eficiencia de espacio en todos los componentes de Spark. Exploraremos más a fondo estos beneficios cuando hablemos sobre el uso de las API DataFrame y Dataset en breve, pero por ahora nos concentraremos en las otras ventajas: expresividad, simplicidad, compatibilidad y uniformidad.

Demostremos primero la expresividad y la composición, con un simple fragmento de código. En el siguiente ejemplo, queremos agregar todas las edades de cada nombre, agrupar por nombre y luego promediar las edades, un patrón común en el análisis y descubrimiento de datos.

Si tuviéramos que usar la API RDD de bajo nivel para esto, el código se vería de la siguiente manera:

```
# En Python
# Crear un RDD de tuplas (nombre, edad)
dataRDD = sc.parallelize([('Brooke', 20), ('Denny', 31), ('Jules', 30),
    ('TD', 35), ('Brooke', 25)])
# Use las transformaciones map y reduceByKey con sus # expresiones
# lambda para agregar y luego calcular el promedio

edadesRDD =
    (dataRDD .map(lambda x: (x[0], (x[1],
        1))) .reduceByKey(lambda x, y: (x[0] + y[0], x[1] +
        y[1])) .map(lambda x: (x[0], x[1][0]/x[1][1])))
```

Nadie discutiría que este código, que le dice a Spark cómo agregar claves y calcular promedios con una serie de funciones lambda, es críptico y difícil de leer. En otras palabras, el código le indica a Spark cómo calcular la consulta. Es completamente opaco para Spark, porque no comunica la intención. Además, el código RDD equivalente en Scala se vería muy diferente del código de Python que se muestra aquí.

Por el contrario, ¿qué pasaría si expresáramos la misma consulta con operadores de DSL de alto nivel y la API de DataFrame, indicándole así a Spark qué hacer? Echar un vistazo:

```
# En Python
desde pyspark.sql import SparkSession
from pyspark.sql.functions import avg #
Crear un DataFrame usando SparkSession
spark =

    (SparkSession .builder .appName("AuthorsAges") .getOrCreate())
# Crear un DataFrame
data_df = spark.createDataFrame([('Brooke', 20), ('Denny', 31), ('Jules', 30),
```

```

("TD", 35), ("Brooke", 25)], ["nombre", "edad"])
# Agrupe los mismos nombres, agregue sus edades y calcule un promedio avg_df =
data_df.groupBy("name").agg(avg("age"))
# Mostrar los resultados de la ejecución final
avg_df.show()

+-----+
| nombre|promedio(edad)|
+-----+
|Brooke|      22.5|
| Julio|      30.0|
| DT|      35.0|
| Denny|      31.0|
+-----+

```

Esta versión del código es mucho más expresiva y más simple que la versión anterior, porque estamos utilizando operadores DSL de alto nivel y API para decirle a Spark qué hacer. En efecto, hemos empleado estos operadores para componer nuestra consulta. Y debido a que Spark puede inspeccionar o analizar esta consulta y comprender nuestra intención, puede optimizar u organizar las operaciones para una ejecución eficiente. Spark sabe exactamente lo que deseamos hacer: agrupar a las personas por sus nombres, agregar sus edades y luego calcular la edad promedio de todas las personas con el mismo nombre. Hemos compuesto un cálculo completo utilizando operadores de alto nivel como una sola consulta simple. ¿Qué tan expresivo es eso?

Algunos dirían que al usar solo operadores DSL expresivos de alto nivel asignados a patrones de análisis de datos comunes o recurrentes para introducir orden y estructura, estamos limitando el alcance de la capacidad de los desarrolladores para instruir al compilador o controlar cómo se deben calcular sus consultas. Tenga la seguridad de que no está confinado a estos patrones estructurados; puede volver en cualquier momento a la API RDD de bajo nivel no estructurada, aunque casi nunca encontramos la necesidad de hacerlo.

Además de ser más fácil de leer, la estructura de las API de alto nivel de Spark también presenta uniformidad entre sus componentes y lenguajes. Por ejemplo, el código de Scala que se muestra aquí hace lo mismo que el código anterior de Python, y la API se ve casi idéntica:

```

// En Scala
import org.apache.spark.sql.functions.avg import
org.apache.spark.sql.SparkSession // Crear un
DataFrame usando SparkSession val spark =

```

```

SparkSession .builder .appName("AuthorsAges") .getOrCreate()
// Crear un DataFrame de nombres y edades val
dataDF = spark.createDataFrame(Seq(("Brooke", 20), ("Brooke", 25), ("Denny", 31),
("Jules", 30), ("TD", 35))).toDF("nombre", "edad")
// Agrupe los mismos nombres, agregue sus edades y calcule un valor promedio avgDF =
dataDF.groupBy ("name").agg(avg("age"))
// Mostrar los resultados de la ejecución final

```

```
avgDF.mostrar()
```

nombre	promedio(edad)
Brooke	22.5
Julio	30.0
DT	35.0
Denny	31.0



Algunos de estos operadores de DSL realizan operaciones de tipo relacional con los que estará familiarizado si sabe SQL, como seleccionar, llenar clasificación, agrupación y agregación.

Toda esta sencillez y expresividad que apreciamos los desarrolladores es posible gracias a el motor Spark SQL sobre el que se construyen las API estructuradas de alto nivel. Está gracias a este motor, que sustenta todos los componentes del Spark, conseguimos unificar formulario API. Ya sea que exprese una consulta contra un DataFrame en Flujo estructurado: ing o MLlib, siempre está transformando y operando en DataFrames como datos estructurados. Echaremos un vistazo más de cerca al motor Spark SQL más adelante en este capítulo, pero por ahora exploremos esas API y DSL para operaciones comunes y cómo usar ellos para el análisis de datos.

## La API de marco de datos

Inspirado en [pandas DataFrames](#) en estructura, formato y algunas operaciones específicas, Spark DataFrames son como tablas distribuidas en memoria con columnas con nombre y esquemas, donde cada columna tiene un tipo de datos específico: entero, cadena, matriz, mapa, real, fecha, marca de tiempo, etc. Para el ojo humano, un Spark DataFrame es como una tabla. Un ejemplo se muestra en [la Tabla 3-1](#).

Tabla 3-1. El formato de tabla de un DataFrame

—	Primer	Último	URL	Publicado	Golpes	Campañas
(En t)	(Cuerda)	(Cuerda)	(Cuerda)	(Fecha)	(En t)	(Lista[Cadenas])
1	Julio	Damji	https:// tinyurl.1	4/1/2016	4535	[twitter, LinkedIn]
2	Brooke	Pequeño	https:// tinyurl.2	5/5/2018	8908	[twitter, LinkedIn]
3	Denny	Solavento	https:// diminutaurl.3	7/6/2019	7659	[web, twitter, Facebook, LinkedIn]
4	Tathagata Das		https:// tinyurl.4	5/12/2018	10568	[twitter, Facebook]

Identificador (En t)	Primeros (Cuerda)	Últimos (Cuerda)	URL (Cuerda)	Publicado (Fecha)	Golpes (En t)	Campañas (Lista[Cadenas])
5	mateo	Zaharia	https://diminutaurl.5	14/5/2014 40578 [web, twitter, Facebook, LinkedIn]		
6	reynold	Preguntas por	https://diminutaurl.6	3/2/2015	25568 [twitter, LinkedIn]	

Cuando los datos se visualizan como una tabla estructurada, no solo es fácil de digerir, sino que también es fácil trabajar con ellos cuando se trata de operaciones comunes que quizás desee ejecutar en filas y columnas. Recuerde también que, como aprendió en el [Capítulo 2](#), los DataFrames son inmutables y Spark mantiene un linaje de todas las transformaciones. Puede agregar o cambiar los nombres y tipos de datos de las columnas, creando nuevos DataFrames mientras se conservan las versiones anteriores. Una columna con nombre en un DataFrame y su tipo de datos Spark asociado se pueden declarar en el esquema.

Examinemos los tipos de datos estructurados y genéricos disponibles en Spark antes de usarlos para definir un esquema. Luego, ilustraremos cómo crear un DataFrame con un esquema, capturando los datos en la [Tabla 3-1](#).

### Tipos de datos básicos de

**Spark** Coinciendo con sus lenguajes de programación admitidos, Spark admite tipos de datos internos básicos. Estos tipos de datos pueden declararse en su aplicación Spark o definirse en su esquema. Por ejemplo, en Scala, puede definir o declarar un nombre de columna en particular para que sea de tipo String, Byte, Long o Map, etc. Aquí, definimos nombres de variables vinculados a un tipo de datos Spark:

```
$SPARK_HOME/bin/spark-shell scala>
import org.apache.spark.sql.types._ import
org.apache.spark.sql.types._ scala> val nameTypes =
StringType nameTypes: org.apache.spark.sql.
tipos.StringType.type = StringType scala> val firstName = nameTypes firstName:
org.apache.spark.sql.types.StringType.type = StringType scala> val lastName = nameTypes
lastName: org.apache.spark.sql.types.StringType. tipo = tipo de cadena
```

La [Tabla 3-2](#) enumera los tipos de datos básicos de Scala admitidos en Spark. Todos ellos son subtipos de la clase [DataTypes](#), excepto para [DecimalType](#).

Tabla 3-2. Tipos de datos básicos de Scala en Spark

Tipo de datos	Valor asignado en Scala	API para instanciar
tipo de byte	Byte	Tipos de datos.ByteType
Tipo corto	Corto	Tipos de datos. Tipo corto
IntegerType Int		DataTypes.IntegerType
Tipo largo	Largo	Tipos de datos. Tipo largo
tipo de flotador	Flotar	Tipos de datos. Tipo flotante
DobleTipo Doble		Tipos de datos.DoubleType
Tipo de cadena Cadena		Tipos de datos.StringType
Tipo booleano Booleano		Tipos de datos.BooleanType
DecimalType java.math.BigDecimal DecimalType		

Spark admite [tipos de datos de Python básicos similares](#), como se enumera en la Tabla 3-3.

Tabla 3-3. Tipos de datos básicos de Python en Spark

Tipo de datos	Valor asignado en la API de Python para instanciar	
tipo de byte	En t	Tipos de datos.ByteType
Tipo corto	En t	Tipos de datos. Tipo corto
IntegerType int		DataTypes.IntegerType
Tipo largo	En t	Tipos de datos. Tipo largo
tipo de flotador	flotar	Tipos de datos. Tipo flotante
Flotador de tipo doble		Tipos de datos.DoubleType
Tipo de cadena str		Tipos de datos.StringType
Tipo booleano bool		Tipos de datos.BooleanType
DecimalType decimal.Decimal		tipo decimal

## Tipos de datos complejos y estructurados de Spark

Para análisis de datos complejos, no tratará solo con tipos de datos simples o básicos. Su los datos serán complejos, a menudo estructurados o anidados, y necesitará Spark para manejar estos tipos de datos complejos. Vienen en muchas formas: mapas, arreglos, estructuras, fechas, marcas de tiempo, campos, etc. [La Tabla 3-4](#) enumera los tipos de datos estructurados de Scala que Spark apoya

Tabla 3-4. Tipos de datos estructurados de Scala en Spark

Tipo de datos	Valor asignado en Scala	API para instanciar
Matriz de tipo binario [Byte]		Tipos de datos.BinaryType
marca de tiempo	java.sql.Timestamp	Tipos de datos.TimestampType
Tipo		
Tipo de fecha	java.sql.Fecha	Tipos de datos. Tipo de fecha
tipo de matriz	scala.colección.Seq	TiposDeDatos.createArrayType(Elemento Tipo)
Tipo de mapa	scala.colección.Mapa	DataTypes.createMapType(keyType, valueType)
StructType org.apache.spark.sql.Row	StructField Un tipo de valor correspondiente al tipo de este campo StructField(name, dataType, [null se convirtió])	StructType(ArrayType[fieldTypes])

Se enumeran los tipos de datos estructurados equivalentes en Python que admite Spark en la [Tabla 3-5](#).

Tabla 3-5. Tipos de datos estructurados de Python en Spark

Tipo de datos	Valor asignado en Python	API para instanciar
BinaryType	bytearray	tipobinario()
TimestampType	datetime.datetime	Tipo de marca de tiempo ()
Tipo de fecha	fechahora.fecha	Tipo de fecha ()
tipo de matriz	Lista, tupla o matriz	ArrayType(tipo de datos, [anulable])
Tipo de mapa	dictar	MapType(keyType, valueType, [nul etiqueta])
Tipo de estructura	Lista o tupla	TipoEstructura([campos])
campo de estructura	Un tipo de valor correspondiente al tipo de este campo	StructField(nombre, tipo de datos, [nul etiqueta])

Si bien estas tablas muestran los innumerables tipos admitidos, es mucho más importante vea cómo estos tipos se combinan cuando define un esquema para sus datos.

## Esquemas y creación de marcos de datos

Un esquema en Spark define los nombres de las columnas y los tipos de datos asociados para un DataMarco. La mayoría de las veces, los esquemas entran en juego cuando lee datos estructurados de una fuente de datos externa (más sobre esto en el próximo capítulo). Definición de un esquema por adelantado en lugar de adoptar un enfoque de lectura de esquema ofrece tres beneficios:

- Libera a Spark de la responsabilidad de inferir tipos de datos. •

Evita que Spark cree un trabajo separado solo para leer una gran parte de su archivo para determinar el esquema, lo que para un archivo de datos grande puede ser costoso y llevar mucho tiempo.

- Puede detectar errores antes si los datos no coinciden con el esquema.

Por lo tanto, lo alentamos a que siempre defina su esquema por adelantado cada vez que desee leer un archivo grande de una fuente de datos. Para una breve ilustración, definamos un esquema para los datos en la [Tabla 3-1](#) y usemos ese esquema para crear un DataFrame.

## Dos formas de definir un

**esquema** Spark le permite definir un esquema de dos formas. Una es definirlo programáticamente y la otra es emplear una cadena de lenguaje de definición de datos (DDL), que es mucho más simple y fácil de leer.

Para definir un esquema mediante programación para un DataFrame con tres columnas con nombre, autor, título y páginas, puede usar la API de Spark DataFrame. Por ejemplo:

```
// En Scala
import org.apache.spark.sql.types._ val
schema = StructType(Array(StructField("autor", StringType, false),
  StructField("título", StringType, false),
  StructField("páginas", IntegerType, false)))
```

```
# En Python
desde pyspark.sql.types import *
esquema = StructType([StructField("autor", StringType(), False),
  StructField("título", StringType(), False),
  StructField("páginas", IntegerType(), False)])
```

Definir el mismo esquema usando DDL es mucho más simple:

```
// En Scala
val schema = "autor STRING, título STRING, páginas INT"
```

```
# En el
esquema de Python = "autor STRING, título STRING, páginas INT"
```

Puede elegir la forma que desee para definir un esquema. Para muchos ejemplos, usaremos ambos:

```
# En Python
desde pyspark.sql import SparkSession
```

```
# Defina el esquema para nuestros datos
usando DDL schema = "Id` INT, `First` STRING, `Last` STRING, `Url` STRING,
`Publicado` STRING, `Hits` INT, `Campañas` ARRAY<STRING>"
```

```
# Crea nuestros datos
estáticos data = [ [1, "Jules", "Damji", "https://tinyurl.1", "1/4/2016", 4535, ["twitter",
```

```

    "LinkedIn"]],  

        [2, "Brooke", "Little", "https://tinyurl.2", "5/5/2018", 8908, ["twitter",  

    "LinkedIn"]],  

        [3, "Denny", "Lee", "https://tinyurl.3", "7/6/2019", 7659, ["web",  

    "twitter", "FB", "LinkedIn"]],  

        [4, "Tathagata", "Das", "https://tinyurl.4", "5/12/2018", 10568,  

    ["twitter", "FB"]],  

        [5, "Matei", "Zaharia", "https://tinyurl.5", "14/5/2014", 40578, ["web",  

    "twitter", "FB", "LinkedIn"]],  

        [6, "Reynold", "Xin", "https://tinyurl.6", "2/3/2015", 25568,  

    ["twitter", "LinkedIn"]]  

    ]  
  

# Programa principal  

si __nombre__ == "__principal__":  

# Crear una SparkSession  

    chispa = (Sesión de chispa  

    .constructor  

    .appName("Ejemplo-3_6")  

    .getOrCreate())  

# Crea un DataFrame usando el esquema definido arriba  

    blogs_df = chispa.createDataFrame(datos, esquema)  

# Mostrar el marco de datos; debe reflejar nuestra tabla de arriba  

    blogs_df.mostrar()  

# Imprimir el esquema utilizado por Spark para procesar el DataFrame  

imprimir(blogs_df.printSchema())

```

Ejecutar este programa desde la consola producirá el siguiente resultado:

```

$ chispa-enviar Ejemplo-3_6.py  

...  

+-----+-----+-----+-----+-----+-----+-----+-----+-----+  

|<índice>| Primero | Última | URL | Publicados | Éxitos | Campañas |  

+-----+-----+-----+-----+-----+-----+-----+-----+  

| 1 | Jules | Damji | https://tinyurl.1 | 4/1/2016 | 4535 | [twitter,...] |  

| 2 | Brooke | Little | https://tinyurl.2 | 5/5/2018 | 8908 | [twitter,...] |  

| 3 | Denny | Lee | https://tinyurl.3 | 7/6/2019 | 7659 | [web, twitter,...] |  

| 4 | Tathagata | Das | https://tinyurl.4 | 5/12/2018 | 10568 | [twitter, FB] |  

| 5 | Matei | Zaharia | https://tinyurl.5 | 14/5/2014 | 40578 | [web, twitter, ...] |  

| 6 | Reynold | Xin | https://tinyurl.6 | 2/3/2015 | 25568 | [twitter,...] |  

+-----+-----+-----+-----+-----+-----+-----+-----+  
  

raíz  

|-- Id: entero (anulable = falso)  

|-- Primero: cadena (anulable = falso)  

|-- Último: cadena (anulable = falso)  

|-- Url: cadena (anulable = falso)  

|-- Publicado: cadena (anulable = falso)  

|-- Resultados: entero (anulable = falso)  

|-- Campañas: matriz (anulable = falso)  

|     |-- elemento: cadena (containsNull = false)

```

Si desea utilizar este esquema en otra parte de su código, simplemente ejecute blogs\_df.schema y devolverá la definición del esquema:

```
StructType(Lista(StructField("Id", IntegerType, false),
StructField("Primero", TipoDeCadena, falso),
StructField("Último", StringType, false),
StructField("URL", StringType, false),
StructField("Publicado", StringType, false),
StructField("Accesos", TipoEntero, falso),
StructField("Campañas", ArrayType(StringType, true), false)))
```

Como puede observar, el diseño de DataFrame coincide con el de la [Tabla 3-1](#) junto con los tipos de datos respectivos y la salida del esquema.

Si tuviera que leer los datos de un archivo JSON en lugar de crear datos estáticos, la definición del esquema sería idéntica. Ilustremos el mismo código con un ejemplo de Scala, esta vez leyendo un archivo JSON:

```
// En el paquete
Scala main.scala.chapter3

importar org.apache.spark.sql.SparkSession importar
org.apache.spark.sql.types._

objeto Ejemplo3_7 { def
principal(argumentos: Matriz[Cadena]) {

    val chispa =
        SparkSession .builder .appName("Ejemplo-3_7") .getOrCreate()

    if (args.longitud <= 0) {
        println(" Ejemplo de uso3_7 <ruta del archivo a blogs.json>")
        Sistema.salir(1)
    }
    // Obtener la ruta al archivo JSON val jsonFile
    = args(0)
    // Definir nuestro esquema programáticamente val
    schema = StructType(Array(StructField("Id", IntegerType, false),
    StructField("Primero", StringType, false),
    StructField("Último", StringType, false),
    StructField("Url", StringType, false),
    StructField("Publicado", StringType, false),
    StructField("Accesos", Tipointeger, falso),
    StructField("Campañas", ArrayType(StringType), false)))

    // Crea un DataFrame leyendo el archivo JSON // con un esquema
    predefinido val blogsDF = spark.read.schema(schema).json(jsonFile)

    // Mostrar el esquema DataFrame como salida
    blogsDF.show(false)
```

```
// Imprime el esquema
println(blogsDF.printSchema)
println(blogsDF.esquema)
}
}
```

No es sorprendente que la salida del programa Scala no sea diferente a la del programa

Programa pitón:

Id	Primero	Última	URL	Publicados	Éxitos	Campañas
1	Jules   Damji	https://tinyurl.1	4/1/2016   4535	[twitter, LinkedIn]		
2	Brooke   Little	https://tinyurl.2	5/5/2018   8908	[twitter, LinkedIn]		
3	Denny   Lee	https://tinyurl.3	6/7/2019   7659	[web, twitter,...]		
4	Tathagata   Das	https://tinyurl.4	5/12/2018   10568	[twitter, FB]		
5	Matei   Zaharia	https://tinyurl.5	14/5/2014   40578	[web, twitter, FB,...]		
6	Reynold   Xin	https://tinyurl.6	3/2/2015   25568	[twitter, LinkedIn]		

```
raíz
|-- Id: entero (anulable = verdadero)
|-- Primero: cadena (anulable = verdadero)
|-- Último: cadena (anulable = verdadero)
|-- Url: cadena (anulable = verdadero)
|-- Publicado: cadena (anulable = verdadero)
|-- Resultados: entero (anulable = verdadero)
|-- Campañas: matriz (anulable = verdadero)
|   |-- elemento: cadena (containsNull = true)
```

```
StructType(StructField("Id",IntegerType,true),
           StructField("Primero",TipoDeCadena,verdadero),
           StructField("Último",StringType,true),
           StructField("URL",StringType,true),
           StructField("Publicado",StringType,verdadero),
           StructField("Accesos",TipoEntero,verdadero),
           StructField("Campañas",ArrayType(StringType,true),true))
```

Ahora que tiene una idea de cómo usar esquemas y datos estructurados en DataFrames, concentrémonos en las columnas y filas de DataFrame y lo que significa operar en ellas con la API de marco de datos.

## Columnas y Expresiones

Como se mencionó anteriormente, las columnas con nombre en DataFrames son conceptualmente similares a columnas con nombre en pandas o R DataFrames o en una tabla RDBMS: describen un tipo de campo Puede enumerar todas las columnas por sus nombres y puede realizar operaciones sobre sus valores usando expresiones relacionales o computacionales. En el apoyo de Spark lenguajes portados, las columnas son objetos con métodos públicos (representados por el tipo de columna ).

También puede usar expresiones lógicas o matemáticas en las columnas. Por ejemplo tu podrías crear una expresión simple usando `expr("columnName * 5")` o `(expr("columnName - 5") > col(otraColumnName))`, donde columnName es un tipo Spark (inteligencia, cuerda, etc.). `expr()` es parte de `pyspark.sql.functions` (Python) y Paquetes `org.apache.spark.sql.functions` (Scala). Como cualquier otra función en esos paquetes, `expr()` toma argumentos que Spark analizará como una expresión, computando el resultado.



Scala, Java y Python tienen **métodos públicos asociados con columnas**. Notarás que la documentación de Spark se refiere tanto a columnas y Columna. Column es el nombre del objeto, mientras que `col()` es un estándar . función incorporada de `spark` que devuelve una columna.

Veamos algunos ejemplos de lo que podemos hacer con columnas en Spark. Cada ejemplo es seguido por su salida:

```
// En Scala
scala> import org.apache.spark.sql.functions._
scala> blogsDF.columns
res2: Array[String] = Array(Campañas, Primero, Hits, Id, Último, Publicado, Url)

// Accede a una columna particular con col y devuelve un tipo Columna
scala> blogsDF.col("Id")
res3: org.apache.spark.sql.Column = id

// Usa una expresión para calcular un valor
scala> blogsDF.select(expr("Accesos * 2")).show(2)
// o use col para calcular el valor
scala> blogsDF.select(col("Accesos") * 2).show(2)

+-----+
|(Accesos * 2)|
+-----+
|      9070|
|     17816|
+-----+

// Usa una expresión para calcular los grandes éxitos de los blogs
// Esto agrega una nueva columna, Big Hitters, basada en la expresión condicional
blogsDF.withColumn("Grandes éxitos ", (expr("Visitas > 10000"))).show()

+-----+-----+-----+-----+-----+-----+
| Identificación| Primero| Último| Url| Publicado| Golpes| Campañas| Grandes éxitos|
+-----+-----+-----+-----+-----+-----+
| 1| Julio| Damji|...| 4/1/2016| 4535| [twitter, LinkedIn]| | 2| Brooke| Pequeño|...| 5/5/2018| falso|
| 8908| [twitter, LinkedIn]| | 3| Denny| Lee|...| 7/6/2019| 7659| [web, twitter, FB,...]| | 4| Tathagata| falso|
| El|...| 5/12/2018| 10568| [twitter, Facebook]| | cierto|
```

5	Matei   Zaharia   ...   14/5/2014   40578   [web, twitter, FB ...   Xin   ...		cierto
6  reynold	3/2/2015[25568] [twitter, LinkedIn]		cierto

```
// Concatenar tres columnas, crear una nueva columna y mostrar el
// columna concatenada recién creada
blogsDF
    .withColumn("AuthorsId", (concat(expr("First"), expr("Last"), expr("Id"))))
    .select(col("AutorId"))
    .mostrar(4)
```

+-----+			
	AuthorsId		
+-----+			
, JulesDamji1			
BrookeWenig2			
DennyLee3			
TathagataDas4			
+-----+			

```
// Estas sentencias devuelven el mismo valor, mostrando que
// expr es lo mismo que una llamada al método col
blogsDF.select(expr("Accesos")).show(2)
blogsDF.select(col("Accesos")).show(2)
blogsDF.select("Accesos").show(2)
```

+-----+			
	Golpes		
+-----+			
4535			
8908			
+-----+			

```
// Ordenar por columna "Id" en orden descendente
blogsDF.sort(col("Id").desc).show()
blogsDF.sort($"Id".desc).show()
```

+-----+-----+-----+-----+-----+-----+-----+-----+							
	Campañas	Primer	Golpes	Identificación	Último	Publicado	URL
+-----+-----+-----+-----+-----+-----+-----+-----+							
[twitter, LinkedIn] reynold 25568  6  Xin  2/3/2015 https://tinyurl.6							
[web, twitter, FB...  Matei 40578  5 Zaharia 14/05/2014 https://tinyurl.5							
[twitter, FB]   Tathagata   10568   4   Das   5/12/2018   https://tinyurl.4							
[web, twitter, FB...  Denny  7659  3  Lee  7/6/2019 https://tinyurl.3							
[twitter, LinkedIn] Brooke  8908  2  pequeño  5/5/2018 https://tinyurl.2							
[twitter, LinkedIn]   Julio   4535   1   Damji   4/1/2016   https://tinyurl.1							
+-----+-----+-----+-----+-----+-----+-----+-----+							

En este último ejemplo, las expresiones blogs\_df.sort(col("Id").desc) y blogsDF.sort(\$"Id".desc) son idénticos. Ambos ordenan la columna DataFrame Id nombrado en orden descendente: se usa una función explícita, col("Id"), para devolver un

Columna , mientras que el otro usa \$ antes del nombre de la columna, que es una función en Spark que convierte la columna llamada Id en una columna.



Aquí solo hemos añadido la superficie y hemos empleado solo un par de métodos en los objetos Column . Para obtener una lista completa de todos los métodos públicos para los objetos Column , lo remitimos a la [documentación](#) de Spark .

Los objetos de columna en un DataFrame no pueden existir de forma aislada; cada columna es parte de una fila en un registro y todas las filas juntas constituyen un DataFrame, que como veremos más adelante en el capítulo es realmente un Dataset[Row] en Scala.

## Filas

Una fila en Spark es un [objeto Fila](#) genérico, que contiene una o más columnas. Cada columna puede ser del mismo tipo de datos (por ejemplo, entero o cadena), o pueden tener diferentes tipos (entero, cadena, mapa, matriz, etc.). Debido a que Row es un objeto en Spark y una colección ordenada de campos, puede crear una instancia de Row en cada uno de los idiomas compatibles con Spark y acceder a sus campos mediante un índice que comienza en 0:

```
// En Scala
import org.apache.spark.sql.Row //
Crear una fila val blogRow = Row(6,
"Reynold", "Xin", "https://tinyurl.6", 255568, "3/2 /2015",
Matriz ("twitter", "LinkedIn"))
// Acceso mediante índice para elementos
individuales blogRow(1) res62: Any = Reynold
```

```
# En Python
desde pyspark.sql import Row
blog_row = Row(6, "Reynold", "Xin", "https://tinyurl.6", 255568, "3/2/2015", ["twitter", "LinkedIn"]
]) # acceso usando el índice para elementos individuales blog_row[1]
```

### Reynold

Los objetos de fila se pueden usar para crear marcos de datos si los necesita para una interactividad y exploración rápidas:

```
# En Python
filas = [Fila("Matei Zaharia", "CA"), Fila("Reynold Xin", "CA")]
autores_df =
chispa.createDataFrame(filas, ["Autores", "Estado"])
autores_df.show()
```

```
// En Scala
val filas = Seq(("Matei Zaharia", "CA"), ("Reynold Xin", "CA"))
val autoresDF =
filas.toDF("Autor", "Estado")
autoresDF.show()
```

```
+-----+
|      Autor|Estado|
+-----+
| Mateo el Grande CA |
| Reynold Xin| CA|
+-----+
```

Sin embargo, en la práctica, normalmente querrá leer DataFrames de un archivo como se ilustró anteriormente. En la mayoría de los casos, debido a que sus archivos van a ser enormes, definir un esquema y usarlo es una forma más rápida y eficiente de crear tramas de datos.

Después de haber creado un DataFrame distribuido grande, querrá realizar algunas operaciones de datos comunes en él. Examinemos algunas de las operaciones de Spark que puede realizar con operadores relacionales de alto nivel en las API estructuradas.

#### **Operaciones comunes de marcos** de datos Para

realizar operaciones de datos comunes en marcos de datos, primero deberá cargar un marco de datos desde una fuente de datos que contenga sus datos estructurados. Spark proporciona una interfaz, **DataFrameReader**, que le permite leer datos en un DataFrame desde una miríada de fuentes de datos en formatos como JSON, CSV, Parquet, Text, Avro, ORC, etc. Del mismo modo, para volver a escribir un DataFrame en una fuente de datos en un formato particular, Spark usa **DataFrameWriter**.

#### **Uso de DataFrameReader y**

**DataFrameWriter** La lectura y la escritura son sencillas en Spark debido a estas abstracciones de alto nivel y las contribuciones de la comunidad para conectarse a una amplia variedad de fuentes de datos, incluidas tiendas NoSQL comunes, RDBMS, motores de transmisión como Apache Kafka y Kinesis, y más.

Para comenzar, leamos un archivo CSV grande que contiene datos sobre las llamadas del Departamento de Bomberos de San Francisco.<sup>1</sup> Como se señaló anteriormente, definiremos un esquema para este archivo y usaremos la clase DataFrameReader y sus métodos para decirle a Spark qué hacer. Debido a que este archivo contiene 28 columnas y más de 4380660 registros,<sup>2</sup> es más eficiente definir un esquema que hacer que Spark lo deduzca.

---

1 Estos datos públicos están disponibles en <https://oreil.ly/iDzQK>.

2 El conjunto de datos original tiene más de 60 columnas. Quitamos algunas columnas innecesarias, eliminamos registros con valores nulos o no válidos, y agregó una columna de Retraso adicional.



Si no desea especificar el esquema, Spark puede inferir el esquema de una muestra a un costo menor. Por ejemplo, puede utilizar el opción de relación de muestreo :

```
// En escala
val sampleDF = chispa
    .leer
    .option("Relación de muestreo", 0.001)
    .option("encabezado", verdadero)
    .csv("""/bloques de datos-conjuntos de datos/aprendizaje-chispa-v2/
sf-fuego/sf-fuego-llamadas.csv""")
```

Echemos un vistazo a cómo hacer esto:

```
# En Python, defina un esquema de
pyspark.sql.types import *

# Manera programática de definir un esquema
fire_schema = StructType([StructField('CallNumber', IntegerType(), True),
    StructField('UnitID', StringType(), True),
    StructField('NúmeroIncidente', TipoInteger(), Verdadero),
    StructField('Tipo de llamada', Tipo de cadena(), Verdadero),
    StructField('CallDate', StringType(), True),
    StructField('WatchDate', StringType(), True),
    StructField('CallFinalDisposition', StringType(), True),
    StructField('DisponibleDtTm', StringType(), Verdadero),
    StructField('Dirección', StringType(), Verdadero),
    StructField('Ciudad', StringType(), Verdadero),
    StructField('Código Postal', IntegerType(), Verdadero),
    StructField('Batallón', StringType(), Verdadero),
    StructField('Área de la estación', Tipo de cadena(), Verdadero),
    StructField('Cuadro', StringType(), Verdadero),
    StructField('OriginalPriority', StringType(), True),
    StructField('Prioridad', StringType(), Verdadero),
    StructField('FinalPriority', IntegerType(), True),
    StructField('ALSUnit', BooleanType(), Verdadero),
    StructField('CallTypeGroup', StringType(), True),
    StructField('NumAlarms', IntegerType(), True),
    StructField('TipoUnidad', TipoCadena(), Verdadero),
    StructField('UnitSequenceInCallDispatch', IntegerType(), True),
    StructField('Distrito de prevención de incendios', StringType(), True),
    StructField('SupervisorDistrict', StringType(), True),
    StructField('Vecindario', StringType(), Verdadero),
    StructField('Ubicación', StringType(), Verdadero),
    StructField('RowID', StringType(), True),
    StructField('Retraso', FloatType(), Verdadero)])]

# Use la interfaz DataFrameReader para leer un archivo CSV
sf_fire_file = "/bloques de datos-conjuntos de datos/learning-spark-v2/sf-fire/sf-fire-calls.csv"
fire_df = chispa.read.csv(sf_fire_file, header=True, schema=fire_schema)

// En Scala sería similar
val fireSchema = StructType(Array(StructField("CallNumber", IntegerType, true),
```

```

StructField("UnitID", StringType, verdadero),
StructField("IncidenteNúmero", IntegerType, verdadero),
StructField("CallType", StringType, verdadero),
StructField("Ubicación", StringType, verdadero),
...
...
StructField("Retraso", FloatType, verdadero))))

// Lee el archivo usando CSV DataFrameReader val
sfFireFile="/databricks-datasets/learning-spark-v2/sf-fire/sf-fire-calls.csv" val fireDF =
spark.read.schema(fireSchema).option("encabezado", "verdadero").csv(sfFireFile)

```

La función spark.read.csv() lee el archivo CSV y devuelve un DataFrame de filas y columnas con nombre con los tipos dictados en el esquema.

Para escribir el DataFrame en una fuente de datos externa en el formato de su elección, puede usar la interfaz DataFrameWriter . Al igual que DataFrameReader, admite **múltiples fuentes de datos**. Parquet, un formato de columna popular, es el formato predeterminado; utiliza compresión rápida para comprimir los datos. Si el DataFrame se escribe como Parquet, el esquema se conserva como parte de los metadatos de Parquet. En este caso, las lecturas posteriores en un DataFrame no requieren que proporcione manualmente un esquema.

**Guardar un DataFrame como un archivo Parquet o una tabla SQL.** Una operación de datos común es explorar y transformar sus datos, y luego conservar el DataFrame en formato Parquet o guardarlos como una tabla SQL. Mantener un DataFrame transformado es tan fácil como leerlo. Por ejemplo, para conservar el DataFrame con el que estábamos trabajando como un archivo después de leerlo, haría lo siguiente:

```

// En Scala para guardar como un archivo
Parquet val parquetPath =
fireDF.write.format("parquet").save(parquetPath)

# En Python para guardar como archivo
Parquet parquet_path =
fire_df.write.format("parquet").save(parquet_path)

```

Alternativamente, puede guardarlo como una tabla, que registra los metadatos con el almacén de metadatos de Hive (cubriremos las tablas administradas y no administradas de SQL, los almacenes de metadatos y los datos). Cuadros en el próximo capítulo):

```

// En Scala para guardar como tabla
val parquetTable = ... // nombre de la
fireDF.write.format("parquet").saveAsTable(parquetTable)

# En Python
parquet_table = # nombre de la tabla
fire_df.write.format("parquet").saveAsTable(parquet_table)

```

Veamos algunas operaciones comunes para realizar en DataFrames después de haber leído los datos.

## Transformaciones y acciones

Ahora que tiene un DataFrame distribuido compuesto por llamadas del Departamento de Bomberos de San Francisco en la memoria, lo primero que querrá hacer como desarrollador es examinar sus datos para ver cómo se ven las columnas. ¿Son de los tipos correctos? ¿Alguno de ellos necesita ser convertido a diferentes tipos? ¿Tienen valores nulos ?

En "Transformaciones, acciones y evaluación diferida" en la página 28 del Capítulo 2, pudo ver cómo se usan las transformaciones y acciones para operar en DataFrames y vio algunos ejemplos comunes de cada una. ¿Qué podemos averiguar de las llamadas del Departamento de Bomberos de San Francisco usando estos?

**Proyecciones y ltrós.** Una proyección en el lenguaje relacional es una forma de devolver solo las filas que coinciden con una determinada condición relacional mediante el uso de filtros. En Spark, las proyecciones se realizan con el método select() , mientras que los filtros se pueden expresar con el método filter() o where() . Podemos usar esta técnica para examinar aspectos específicos de nuestro conjunto de datos del Departamento de Bomberos de SF:

```
# En Python
few_fire_df =
    (fire_df .select("IncidentNumber", "AvailableDtTm",
        "CallType") .where(col("CallType") != "Medical Incident")) few_fire_df.show(5,
    truncate=False)

// En Scala val
pocosFireDF = fireDF
    .select("Número de incidente", "DtTm disponible", "Tipo de
    llamada") .where($"Tipo de llamada" != "Incidente médico")
fewFireDF.show(5, false)

+-----+-----+-----+
|Número de incidente|DtTm disponible          |Tipo de llamada   |
+-----+-----+-----+
|2003235 |  |01/11/2002 01:47:00 AM|Incendio Estructural|
|2003235 |  |01/11/2002 01:51:54 AM|Incendio Estructural|
|2003235 |  |01/11/2002 01:47:00 AM|Incendio Estructural|
|2003235 |  |01/11/2002 01:47:00 AM|Incendio Estructural|
|2003235 |  |01/11/2002 01:51:17 AM|Incendio Estructural|
+-----+-----+-----+
solo se muestran las 5 filas superiores
```

¿Qué pasa si queremos saber cuántos CallTypes distintos se registraron como las causas de las llamadas de incendio? Estas consultas simples y expresivas hacen el trabajo:

```
# En Python, devolver el número de distintos tipos de llamadas usando countDistinct() de
pyspark.sql.functions import * (fire_df
```

```
.select("CallType") .where(col("CallType").isNotNull()) .agg(countDistinct("CallType").alias("DistinctCallTypes")) .show()

// En Scala
import org.apache.spark.sql.functions._
```

```
fireDF .select("CallType") .where(col("CallType").isNotNull) .agg(countDistinct('CallType) as 'DistinctCallTypes) .show ()  

+-----+
|Tipos de llamada distintos|
+-----+
|          32|
+-----+
```

Podemos enumerar los distintos tipos de llamadas en el conjunto de datos usando estas consultas:

```
# En Python, filtre solo CallTypes distintos no nulos de todas las filas
(fire_df.select("CallType") .where(col("CallType").isNotNull()) .distinct() .show(10, False) )
```

```
// En Scala
fireDF
```

```
.select("CallType") .where($"CallType".isNotNull()) .distinct() .show(10, false)
```

Salida[20]: 32

```
+-----+
|Tipo de llamada |
+-----+
| Rescate en ascensores / escaleras mecánicas |
|Incendio Marino |
|Emergencia Aeronave |
|Espacio Confinado / Colapso de Estructura|
|Administrativo |
|Alarmas |
|Olor (Extraño / Desconocido) |
|Relámpago (Investigación) |
|Asistencia al ciudadano / Llamada de servicio |
|Materiales peligrosos |
+-----+
solo se muestran las 10 primeras filas
```

**Cambiar el nombre, agregar y eliminar columnas.** A veces desea cambiar el nombre de columnas particulares por razones de estilo o convención, y otras veces por legibilidad o brevedad. Los nombres de las columnas originales en el conjunto de datos del Departamento de Bomberos de SF tenían espacios en ellos. Por ejemplo, el nombre de la columna IncidentNumber era Incident Number. Los espacios en los nombres de las columnas pueden ser problemáticos, especialmente cuando desea escribir o guardar un DataFrame como un archivo Parquet (que lo prohíbe).

Al especificar los nombres de las columnas deseadas en el esquema con StructField, como hicimos nosotros, cambiamos efectivamente todos los nombres en el DataFrame resultante.

Como alternativa, puede cambiar el nombre de las columnas de forma selectiva con el método `withColumnRenamed()`. Por ejemplo, cambiemos el nombre de nuestra columna Delay a ResponseDelayedinMins y echemos un vistazo a los tiempos de respuesta que duraron más de cinco minutos:

```
# En Python
new_fire_df = fire_df.withColumnRenamed("Delay", "ResponseDelayedinMins")
(new_fire_df .select("ResponseDelayedinMins") .where(col("ResponseDelayedinMins") > 5) .show(5, False))
```

```
// En Scala val
newFireDF = fireDF.withColumnRenamed("Delay", "ResponseDelayedinMins")
nuevoFireDF

.select("RespuestaRetrasadaEnMinutos") .where($"ResponseDelayedinMins"
> 5) .show(5, false)
```

Esto nos da una nueva columna renombrada:

```
+-----+
|Respuesta retrasada en minutos|
+-----+
|5.233333 |
|6.933334 |
|6.116667 |7.85 |
|77.333336 |

+-----+
solo se muestran las 5 filas superiores
```



Debido a que las transformaciones de DataFrame son inmutables, cuando cambiamos el nombre de una columna usando `withColumnRenamed()` obtenemos un nuevo DataFrame mientras retenemos el original con el nombre de la columna anterior.

Modificar el contenido de una columna o su tipo son operaciones comunes durante la exploración de datos. En algunos casos, los datos son crudos o sucios, o sus tipos no son susceptibles de

siendo suministrados como argumentos a los operadores relacionales. Por ejemplo, en nuestro conjunto de datos del Departamento de Bomberos de SF, las columnas CallDate, WatchDate y AlarmDtTm son cadenas en lugar de marcas de tiempo de Unix o fechas SQL, las cuales Spark admite y puede manipular fácilmente durante transformaciones o acciones (por ejemplo, durante una fecha- o análisis basado en el tiempo de los datos).

Entonces, ¿cómo los convertimos a un formato más útil? Es bastante simple, gracias a algunos métodos API de alto nivel. spark.sql.functions tiene un conjunto de funciones de marca de fecha/hora hasta/desde como to\_timestamp() y to\_date() que podemos usar solo para este propósito:

```
# En Python
fire_ts_df = (new_fire_df
    .withColumn("IncidentDate", to_timestamp(col("CallDate"), "MM/dd/yyyy"))
    .drop("CallDate") .withColumn("OnWatchDate", to_timestamp(col("WatchDate"),
    "MM /dd/yyyy")) .drop("WatchDate") .withColumn("AvailableDtTS",
    to_timestamp(col("AvailableDtTm"), "MM/dd/yyyy hh:mm:ss a")) .drop("AvailableDtTm "))
```

```
# Seleccione las columnas convertidas
(fire_ts_df .select("IncidentDate",
    "OnWatchDate", "AvailableDtTS") .show(5, False))
```

```
// En Scala
val fireTsDF = newFireDF
    .withColumn("IncidentDate", to_timestamp(col("CallDate"), "MM/dd/yyyy"))
    .drop("CallDate") .withColumn("OnWatchDate", to_timestamp(col("WatchDate"),
    "MM /dd/yyyy")) .drop("WatchDate") .withColumn("AvailableDtTS",
    to_timestamp(col("AvailableDtTm"), "MM/dd/yyyy hh:mm:ss a")) .drop("AvailableDtTm ")
```

```
// Seleccione las columnas convertidas
fireTsDF .select("IncidentDate",
    "OnWatchDate", "AvailableDtTS") .show(5, false)
```

Esas consultas tienen un gran impacto: están sucediendo varias cosas. Desempaquetemos lo que hacen:

1. Convierta el tipo de datos de la columna existente de cadena a un tipo compatible con Spark marca de tiempo
2. Use el nuevo formato especificado en la cadena de formato "MM/dd/yyyy" o "MM/dd/yyyy hh:mm:ss a" cuando corresponda.
3. Despues de convertir al nuevo tipo de datos, suelte() la columna anterior y agregue la nueva especificada en el primer argumento al método withColumn() .
4. Asigne el nuevo DataFrame modificado a fire\_ts\_df.

Las consultas dan como resultado tres nuevas columnas:

```
+-----+-----+-----+
|IncidenteFecha |Fecha de vigilancia |DisponibleDtTS |
+-----+-----+-----+
|2002-01-11 00:00:00|2002-01-10 00:00:00|2002-01-11 01:58:43| |2002-01-11
00:00:00|2002-01-10 00:00:00|2002-01-11 02:10:17| |2002-01-11 00:00:00|
2002-01-10 00:00:00|2002-01-11 01:47:00| |2002-01-11 00:00:00|2002-01-10
00:00:00|2002-01-11 01:51:54| |2002-01-11 00:00:00|2002-01-10 00:00:00|
2002-01-11 01:47:00|
+-----+-----+-----+
```

solo se muestran las 5 filas superiores

Ahora que hemos modificado las fechas, podemos consultar usando funciones de spark.sql.functions como month(), year() y day() para explorar más nuestros datos.

Podríamos averiguar cuántas llamadas se registraron en los últimos siete días, o podríamos ver cuántos años de llamadas del Departamento de Bomberos se incluyen en el conjunto de datos con este consulta:

*# En Python*

```
(fire_ts_df .select(year('IncidentDate')) .distinct() .orderBy(year('IncidentDate')) .show())
```

*// En Scala*

```
fireTsDF .select(year($"IncidentDate")) .distinct() .orderBy(year($"IncidentDate")) .show()
+-----+
|año(FechalIncidente)|
+-----+
| 2000|
| 2001|
| 2002|
| 2003|
| 2004|
| 2005|
```

```

2006|
2007|
2008|
2009|
2010|
2011|
2012|
2013|
2014|
2015|
2016|
2017|
2018|
+-----+

```

Hasta ahora, en esta sección, hemos explorado una serie de operaciones de datos comunes: ing y escribir DataFrames; definir un esquema y usarlo cuando se lee en un DataMarco; guardar un DataFrame como archivo o tabla de Parquet; proyección y filtrado seleccionado columnas de un DataFrame existente; y modificar, renombrar y soltar columnas

Una última operación común es agrupar datos por valores en una columna y agregar los datos de alguna manera, como simplemente contarlos. Este patrón de agrupar y contar es tan común como proyectar y filtrar. Vamos a intentarlo.

**Agregaciones.** ¿Y si queremos saber cuáles son los tipos de llamadas de fuego más habituales? fueron, o qué códigos postales representaron la mayoría de las llamadas? Este tipo de preguntas son común en el análisis y la exploración de datos.

Un puñado de transformaciones y acciones en DataFrames, como groupBy(), orderBy() y count(), ofrecen la capacidad de agregar por nombres de columna y luego recuentos agregados a través de ellos.



Para DataFrames más grandes en los que planea realizar frecuentes o consultas repetidas, podría beneficiarse del almacenamiento en caché. vamos a cubrir Estrategias de almacenamiento en caché de DataFrame y sus beneficios en capítulos posteriores.

Tomemos nuestra primera pregunta: ¿cuáles fueron los tipos más comunes de llamadas de incendio?

```

# en pitón
(fuego_ts_df
 .select("Tipo de llamada")
 .where(col("TipodeLlamada").noesNulo())
 .groupBy("Tipo de llamada")
 .contar()
 .orderBy("recuento", ascendente=False)
 .mostrar(n=10, truncar=False))

```

// En Scala

```
fireTsDF .select("CallType") .where(col("CallType").isNotNull) .groupBy("CallType") .count () .orderBy(desc("count")) .show(10, falso)
```

Tipos de llamada	contar
Incidente Médico  2843475	
Incendio Estructural  578998	
Alarma  483518	
Colisión de tráfico  175507	
Asistencia al ciudadano / Llamada de servicio  65360	
Otros  56961	
Fuego exterior  51603	
Vehículo Incendio  20939	
Salvamento Acuático  20037	
Fuga de Gas (Gases Naturales y LP ) 17284	

De este resultado podemos concluir que el tipo de llamada más común es Incidente médico.



La API de DataFrame también ofrece el método `collect()` , pero para DataFrames extremadamente grandes, esto consume muchos recursos (es caro) y es peligroso, ya que puede causar excepciones de falta de memoria (OOM). A diferencia de `count()`, que devuelve un solo número al controlador, `collect()` devuelve una colección de todos los objetos de fila en todo el marco de datos o el conjunto de datos. Si desea echar un vistazo a algunos registros de fila , es mejor que use `take(n)`, que devolverá solo los primeros *n* objetos de fila del marco de datos.

**Otras operaciones comunes de DataFrame.** Junto con todos los demás que hemos visto, la API de marco de datos proporciona métodos estadísticos descriptivos como `min()`, `max()`, `sum()` y `avg()`. Echemos un vistazo a algunos ejemplos que muestran cómo calcularlos con nuestro conjunto de datos del Departamento de Bomberos de SF.

Aquí calculamos la suma de alarmas, el tiempo de respuesta promedio y los tiempos de respuesta mínimo y máximo para todas las llamadas de incendio en nuestro conjunto de datos, importando las funciones de PySpark de forma Pythonic para no entrar en conflicto con las funciones integradas de Python:

```
# En Python ,
importe pyspark.sql.functions como F
(fire_ts_df .select(F.sum("NumAlarms"),
F.avg("ResponseDelayedinMins"),
```

```
F.min("RespuestaDelayedinMins"), F.max("ResponseDelayedinMins")) .show()

// En Scala
import org.apache.spark.sql.{functions => F}
fireTsDF .select(F.sum("NumAlarms"),
  F.avg("ResponseDelayedinMins"), F.min("ResponseDelayedinMins"),
  F.max("RespuestaDelayedinMins")) .show()

+-----+-----+-----+
|sum(NumAlarms)|avg(Respuesta retrasada en minutos)|min(Respuesta retrasada en minutos)|max(...) |
+-----+-----+-----+
| 4403441| 3.902170335891614| 0.0166666668|1879.6167|
+-----+-----+-----+
```

Para necesidades estadísticas más avanzadas comunes con cargas de trabajo de ciencia de datos, lea la documentación de la API para métodos como `stat()`, `describe()`, `correlación()`, `covarianza()`, `sampleBy()`, `approxQuantile()`, `commonItems()` , etc.

Como puede ver, es fácil redactar y encadenar consultas expresivas con los operadores DSL y API de alto nivel de DataFrames. ¡No podemos imaginar la opacidad y la relativa imposibilidad de leer el código si intentáramos hacer lo mismo con los RDD!

#### Ejemplo de DataFrame de un extremo a otro

Hay muchas posibilidades para el análisis exploratorio de datos, ETL y operaciones de datos comunes en el conjunto de datos públicos del Departamento de Bomberos de San Francisco, más allá de lo que hemos mostrado aquí.

Para abreviar, no incluiremos todo el código de ejemplo aquí, sino el repositorio de [GitHub](#) del libro, proporciona cuadernos de Python y Scala para que intente completar un ejemplo de marco de datos de un extremo a otro utilizando este conjunto de datos. Los cuadernos exploran y responden las siguientes preguntas comunes que puede hacer, utilizando la API de DataFrame y los operadores relacionales de DSL:

- ¿Cuáles fueron todos los diferentes tipos de llamadas de emergencia en 2018? • ¿Qué meses del año 2018 vieron la mayor cantidad de llamadas de emergencia? •
- ¿Qué vecindario en San Francisco generó la mayor cantidad de llamadas de emergencia en 2018? •
- ¿Qué vecindarios tuvieron los peores tiempos de respuesta a las llamadas de incendios en 2018? •
- ¿Qué semana del año en 2018 tuvo la mayor cantidad de llamadas de emergencia? • ¿Existe una correlación entre el vecindario, el código postal y el número de llamadas de bomberos? • ¿Cómo podemos usar archivos Parquet o tablas SQL para almacenar estos datos y volver a leerlos?

Hasta ahora, hemos discutido ampliamente la API de DataFrame, una de las API estructuradas que abarca los componentes MLlib y de transmisión estructurada de Spark, que trataremos más adelante en el libro.

A continuación, cambiaremos nuestro enfoque a la API de conjunto de datos y exploraremos cómo las dos API brindan una interfaz unificada y estructurada a los desarrolladores para programar Spark.

Luego, examinaremos la relación entre las API de RDD, DataFrame y Dataset, y lo ayudaremos a determinar cuándo usar qué API y por qué.

### La API del conjunto de datos

Como se indicó anteriormente en este capítulo, Spark 2.0 **unificó** las API DataFrame y Dataset como API estructuradas con interfaces similares para que los desarrolladores solo tengan que aprender un único conjunto de API. Los conjuntos de datos adquieren dos características: **API con tipo y sin tipo**, como se muestra en la [Figura 3-1](#).

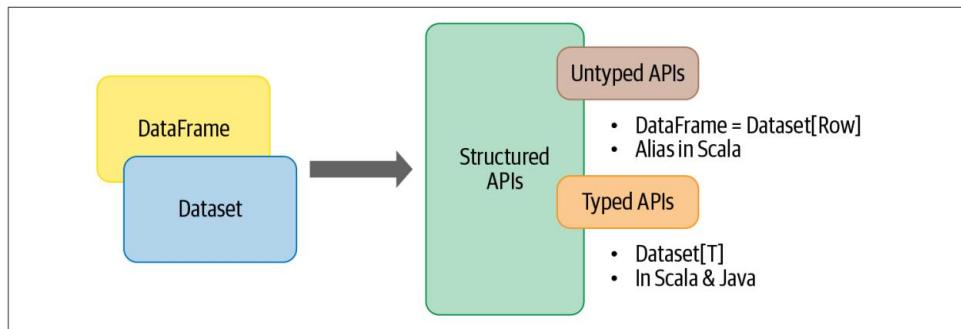


Figura 3-1. API estructuradas en Apache Spark

Conceptualmente, puede pensar en un DataFrame en Scala como un alias para una colección de objetos genéricos, `Dataset[Row]`, donde Row es un objeto JVM genérico sin tipo que puede contener diferentes tipos de campos. Un conjunto de datos, por el contrario, es una colección de objetos JVM fuertemente tipados en Scala o una clase en Java. O, como la [documentación del conjunto de datos](#) lo pone, un

El conjunto de datos es:

una colección fuertemente tipada de objetos específicos del dominio que se pueden transformar en paralelo usando operaciones funcionales o relacionales. Cada conjunto de datos [en Scala] también tiene una vista sin tipo llamada `DataFrame`, que es un conjunto de datos de fila.

### Objetos con tipo, objetos sin tipo y filas genéricas

compatibles con Spark, los conjuntos de datos solo tienen sentido en Java y Scala, mientras que en Python y R solo tienen sentido los marcos de datos. Esto se debe a que Python y R no son de tipo seguro en tiempo de compilación; los tipos se infieren o asignan dinámicamente durante la ejecución, no durante el tiempo de compilación. Lo contrario es cierto en Scala y Java: los tipos están vinculados a

variables y objetos en tiempo de compilación. En Scala, sin embargo, un DataFrame es solo un alias para Dataset[Row] sin tipo. La tabla 3-6 lo resume en pocas palabras.

Tabla 3-6. Objetos con tipo y sin tipo en Spark

Lenguaje	Abstracción principal con tipo y sin tipo	Con o sin escribir
Scala	Dataset[T] y DataFrame (alias para Dataset[Row])	Tanto con tipo como sin tipo
Java	Conjunto de datos<T>	mecongrado
Pítón	Marco de datos	Fila genérica sin tipo
R	Marco de datos	Fila genérica sin tipo

Row es un tipo de objeto genérico en Spark, que contiene una colección de tipos mixtos a los que se puede acceder mediante un índice. Internamente, Spark manipula los objetos Row y los convierte en los tipos equivalentes que se tratan en la Tabla 3-2 y la Tabla 3-3. Por ejemplo, un Int como uno de sus campos en una fila se asignará o convertirá a IntegerType o IntegerType() respectivamente para Scala o Java y Python:

```
// En Scala
import org.apache.spark.sql.Row val fila =
Fila(350, verdadero, "Learning Spark 2E", nulo)

# En Python
desde pyspark.sql importar Fila fila =
Fila (350, Verdadero, "Learning Spark 2E", Ninguno)
```

Usando un índice en el objeto Fila , puede acceder a campos individuales con sus métodos captadores públicos:

```
// En Scala
row.getInt(0) res23:
Int = 350

row.getBoolean(1) res24:
Boolean = true row.getString(2)
res25: String = Learning Spark
2E

# En la fila de
Python [0]
Salida[13]: 350
filas[1]
Salida[14]: Fila
verdadera [2]
Salida[15]: 'Learning Spark 2E'
```

Por el contrario, los objetos con tipo son objetos reales de clase Java o Scala en la JVM. Cada elemento en un conjunto de datos se asigna a un objeto JVM.

## Creación de conjuntos de datos

Al igual que con la creación de tramas de datos a partir de fuentes de datos, al crear un conjunto de datos debe conocer el esquema. En otras palabras, necesita conocer los tipos de datos. Aunque con datos JSON y CSV es posible inferir el esquema, para grandes conjuntos de datos esto es intensivo en recursos (caro). Al crear un conjunto de datos en Scala, la forma más fácil de especificar el esquema para el conjunto de datos resultante es usar una clase de caso. En Java, JavaBean se utilizan clases (discutimos más a fondo las clases de casos de JavaBean y Scala en el [Capítulo 6](#)).

### Scala: clases de casos

Cuando desee instanciar su propio objeto específico de dominio como un conjunto de datos, puede hágalo definiendo una clase de caso en Scala. Como ejemplo, veamos una colección de lecturas ings de dispositivos de Internet de las cosas (IoT) en un archivo JSON (usamos este archivo en el ejemplo de extremo a extremo más adelante en esta sección).

Nuestro archivo tiene filas de cadenas JSON que se ven de la siguiente manera:

```
{"device_id": 198164, "device_name": "sensor-pad-198164owomcJZ", "ip": "80.55.20.25",
"cca2": "PL", "cca3": "POL", "cn": " Polonia", "latitud": 53.080000, "longitud": 18.620000, "escala": "Celsius", "temp": 21, "humedad": 65,
"battery_level": 8, "c02_level": 1408, "lcd": "red", "timestamp": "1458081226051"}
```

Para expresar cada entrada JSON como DevicelotData, un objeto específico de dominio, podemos definir una clase de caso Scala:

```
case class DevicelotData (battery_level: Long, c02_level: Long,
    cca2: Cadena, cca3: Cadena, cn: Cadena, device_id: Largo,
    device_name: Cadena, humedad: Larga, ip: Cadena, latitud: Doble,
    lcd: Cadena, longitud: doble, escala: Cadena, temperatura: larga,
    marca de tiempo: larga)
```

Una vez definido, podemos usarlo para leer nuestro archivo y convertir el Dataset[Row] devuelto en Dataset[DevicelotData] (salida truncada para que quepa en la página):

```
// En escala
val ds = chispa.leer
    .json("/databricks-conjuntos de datos/learning-spark-v2/iot-devices/iot_devices.json")
    .as[DevicelotData]

ds: org.apache.spark.sql.Dataset[DevicelotData] = [battery_level...]

ds.mostrar(5, falso)

+-----+-----+-----+-----+-----+
|nivel_batería|nivel_c02|cca2|cca3|cn|id_dispositivo|...
+-----+-----+-----+-----+-----+
|8|868|EE.UU.|EE.UU.|Estados Unidos|1|...
|7|1473|NO|NOR|Noruega|2|...
|2|1556|IT|ITA|Italia|3|...
```

```
+---+---+---+---+---+
| 6 |1080 |EE.UU . |EE.UU . |Estados Unidos|4 |...
| 4 |931 |PH |PHL |Filipinas |5 |...
+---+---+---+---+---+
solo se muestran las 5 filas superiores
```

### Operaciones de conjuntos de datos

Así como puede realizar transformaciones y acciones en DataFrames, también puede hacerlo con Conjuntos de datos. Dependiendo del tipo de operación, los resultados variarán:

```
// En escala
val filterTempDS = ds.filter({d => {d.temp > 30 && d.humedad > 70}})

filterTempDS: org.apache.spark.sql.Dataset[DeviceLoTData] = [battery_level...]

filterTempDS.show(5, falso)
+-----+-----+-----+-----+-----+
|nivel_bateria|nivel_c02|cca2|cca3|cn |id_dispositivo|...
+-----+-----+-----+-----+-----+
| 0 |1466 |EE.UU . |EE.UU . |Estados Unidos|17 |...
| 9 | 986 |FR |FRA | Francia | 48 | ...
| 8 |1436 |EE.UU . |EE.UU . |Estados Unidos|54 |...
| 4 |1090 |EE.UU . |EE.UU . |Estados Unidos|63 |...
| 4 |1072 |PH |PHL |Filipinas |81 |...
+-----+-----+-----+-----+-----+
solo se muestran las 5 filas superiores
```

En esta consulta, usamos una función como argumento para el método filter() del conjunto de datos.

Este es un método sobrecargado con muchas firmas. La versión que usamos, filter(func: (T) > Boolean): Dataset[T], toma una función lambda, func: (T) > Boolean, como su argumento.

El argumento de la función lambda es un objeto JVM de tipo DeviceLoTData. Como tal, podemos acceder a sus campos de datos individuales usando la notación de punto (.) , como usted lo haría en una clase Scala o JavaBean.

Otra cosa a tener en cuenta es que con DataFrames, expresas tus condiciones filter() como operaciones DSL similares a SQL, que son independientes del idioma (como vimos anteriormente en el fuego llama ejemplos). Con Datasets, usamos expresiones nativas del lenguaje como Scala o Código Java.

Aquí hay otro ejemplo que da como resultado otro conjunto de datos más pequeño:

```
// En escala
case class DeviceTempByCountry(temp: Long, device_name: String, device_id: Long,
                                cca3: Cadena)
valor dsTemp = ds
  .filtrar(d => {d.temp > 25})
  .map(d => (d.temp, d.nombre_dispositivo, d.id_dispositivo, d.cca3))
  .toDF("temp", "nombre_dispositivo", "id_dispositivo", "cca3")
```

```
.as[DeviceTempByCountry]
dsTemp.show(5, false)

+-----+-----+
|temp|nombre_dispositivo |id_dispositivo|ccaa3|
+-----+-----+
|34 | Estados Unidos | Estados Unidos |
| Estados Unidos | JPN |
| Estados Unidos |
+-----+
solo se muestran las 5 filas superiores
```

O puede inspeccionar solo la primera fila de su conjunto de datos:

```
val dispositivo = dsTemp.first()
println(dispositivo)

dispositivo: DeviceTempByCountry =
DeviceTempByCountry(34,meter-gauge-1xbYRYcj,1,USA)
```

Alternativamente, podría expresar la misma consulta usando nombres de columna y luego convertirla a un Dataset[DeviceTempByCountry]:

```
// En Scala
val dsTemp2 = ds
.select($"temp", $"device_name", $"device_id", $"device_id", $"ccaa3") .where("temp >
25") .as[DeviceTempByCountry]
```



Semánticamente, select() es como map() en la consulta anterior, en el sentido de que ambas consultas seleccionan campos y generan resultados equivalentes.

En resumen, las operaciones que podemos realizar en conjuntos de datos (filter(), map(), groupBy(), select(), take(), etc.) son similares a las de DataFrames. En cierto modo, los conjuntos de datos son similares a los RDD en el sentido de que proporcionan una interfaz similar a sus métodos mencionados anteriormente y seguridad en tiempo de compilación, pero con una interfaz de programación orientada a objetos mucho más fácil de leer.

Cuando usamos conjuntos de datos, el motor Spark SQL subyacente maneja la creación, conversión, serialización y deserialización de los objetos JVM. También se ocupa de la gestión de la memoria del montón de Java con la ayuda de los codificadores de conjuntos de datos. (Hablaremos más sobre conjuntos de datos y administración de memoria en el [Capítulo 6](#)).

### Ejemplo de conjunto de datos de un extremo a otro

En este ejemplo de conjunto de datos de un extremo a otro, llevará a cabo un análisis exploratorio de datos, ETL (extracción, transformación y carga) y operaciones de datos similares a los del ejemplo de DataFrame, utilizando el conjunto de datos de IoT. Este conjunto de datos es pequeño y falso, pero nuestro principal objetivo aquí es ilustrar la claridad con la que puede expresar consultas con conjuntos de datos y la legibilidad de esas consultas, tal como lo hicimos con DataFrames.

Nuevamente, por brevedad, no incluiremos aquí todo el código de ejemplo; sin embargo, hemos proporcionado el cuaderno en el repositorio de [GitHub](#). El cuaderno explora las operaciones comunes que puede realizar con este conjunto de datos. Con la API de conjunto de datos, intentamos hacer lo siguiente:

1. Detectar dispositivos defectuosos con niveles de batería por debajo de un umbral.
2. Identificar países infractores con altos niveles de emisiones de CO2.
3. Calcule los valores mínimo y máximo de temperatura, nivel de batería, CO2 y humedad.
4. Ordene y agrupe por temperatura promedio, CO2, humedad y país.

### Marcos de datos frente a conjuntos de datos

A estas alturas, es posible que se pregunte por qué y cuándo debería usar DataFrames o Datasets. En muchos casos, cualquiera de los dos funcionará, dependiendo de los idiomas en los que esté trabajando, pero hay algunas situaciones en las que uno es preferible al otro. Aquí están algunos ejemplos:

- Si quiere decirle a Spark qué hacer, no cómo hacerlo, use DataFrames o Datasets. • Si desea una semántica rica, abstracciones de alto nivel y operadores DSL, utilice marcos de datos o conjuntos de datos.
- Si desea una seguridad de tipos estricta en tiempo de compilación y no le importa crear varios casos clases para un Dataset[T] específico, use Datasets.
- Si su procesamiento exige expresiones de alto nivel, filtros, mapas, agregaciones, cálculos de promedios o sumas, consultas SQL, acceso en columnas o el uso de operadores relacionales en datos semiestructurados, use DataFrames o Datasets.
- Si su procesamiento dicta transformaciones relacionales similares a las consultas tipo SQL, utilizar marcos de datos.
- Si desea aprovechar y beneficiarse de la serialización eficiente de Tungsten ción con codificadores, [uso de conjuntos de datos](#).
- Si desea unificar, optimizar el código y simplificar las API en los componentes de Spark, use DataFrames.

- Si es usuario de R, utilice DataFrames. • Si es un usuario de Python, use DataFrames y desplácese hasta RDD si lo necesita.
- mas control.
- Si desea eficiencia de espacio y velocidad, utilice DataFrames. • Si desea detectar los errores durante la compilación en lugar de durante el tiempo de ejecución, elija la opción API apropiada como se muestra en la Figura 3-2.

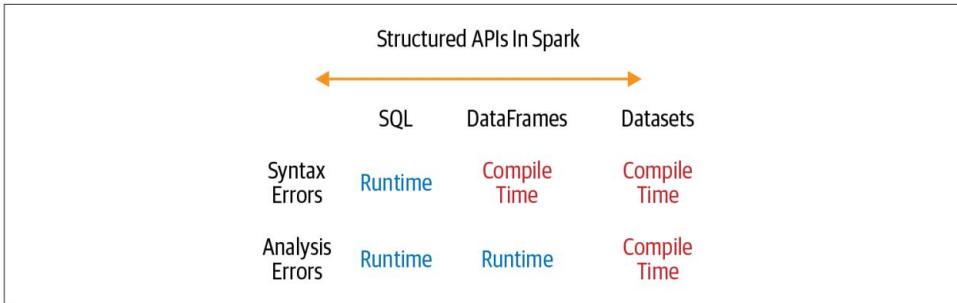


Figura 3-2. Cuando se detectan errores usando las API estructuradas

## Cuándo usar RDD

Usted puede preguntar: ¿Los RDD están siendo relegados a ciudadanos de segunda clase? ¿Están siendo desaprobados? ¡La respuesta es un rotundo no! La API de RDD seguirá siendo compatible, aunque todo el trabajo de desarrollo futuro en Spark 2.x y Spark 3.0 seguirá teniendo una interfaz y semántica DataFrame en lugar de usar RDD.

Hay algunos escenarios en los que querrá considerar el uso de RDD, como cuando usted:

- Están usando un paquete de terceros que está escrito usando RDD • Pueden renunciar a la optimización del código, la utilización eficiente del espacio y el rendimiento beneficios disponibles con DataFrames y Datasets
- Quiere indicar con precisión a Spark cómo hacer una consulta

Además, puede moverse sin problemas entre DataFrames o Datasets y RDD a voluntad usando una simple llamada de método API, df.rdd. (Sin embargo, tenga en cuenta que esto tiene un costo y debe evitarse a menos que sea necesario). Después de todo, los marcos de datos y los conjuntos de datos se construyen sobre los RDD y se descomponen para compactar el código RDD durante la generación del código de etapa completa, que analizamos en la siguiente sección.

Finalmente, las secciones anteriores proporcionaron cierta intuición sobre cómo las API estructuradas en Spark permiten a los desarrolladores usar API fáciles y amigables para redactar consultas expresivas sobre datos estructurados. En otras palabras, le dices a Spark qué hacer, no cómo hacerlo, usando

operaciones de alto nivel y determina la forma más eficiente de crear una consulta y genera un código compacto para usted.

Este proceso de crear consultas eficientes y generar código compacto es el trabajo del motor Spark SQL. Es el sustrato sobre el que se construyen las API estructuradas que hemos estado analizando. Echemos un vistazo debajo del capó a ese motor ahora.

## Spark SQL y el motor subyacente

A nivel programático, Spark SQL permite a los desarrolladores emitir consultas compatibles con ANSI SQL:2003 sobre datos estructurados con un esquema. Desde su introducción en Spark 1.3, Spark SQL ha evolucionado hasta convertirse en un motor sustancial sobre el cual se han construido muchas funcionalidades estructuradas de alto nivel. Además de permitirle realizar consultas similares a las de SQL en sus datos, el motor Spark SQL:

- Unifica los componentes de Spark y permite la abstracción de DataFrames/Datasets en Java, Scala, Python y R, lo que simplifica el trabajo con conjuntos de datos estructurados.
- Se conecta a las tablas y el metaalmacén de Apache Hive. •

Lee y escribe datos estructurados con un esquema específico de formatos de archivos estructurados (JSON, CSV, Text, Avro, Parquet, ORC, etc.) y convierte los datos en tablas temporales.

- Ofrece un shell Spark SQL interactivo para una rápida exploración de datos. •

Proporciona un puente hacia (y desde) herramientas externas a través de conectores JDBC/ODBC de base de datos estándar.

- Genera planes de consulta optimizados y código compacto para la JVM, para fines ejecución.

La figura 3-3 muestra los componentes con los que interactúa Spark SQL para lograr todo esto.

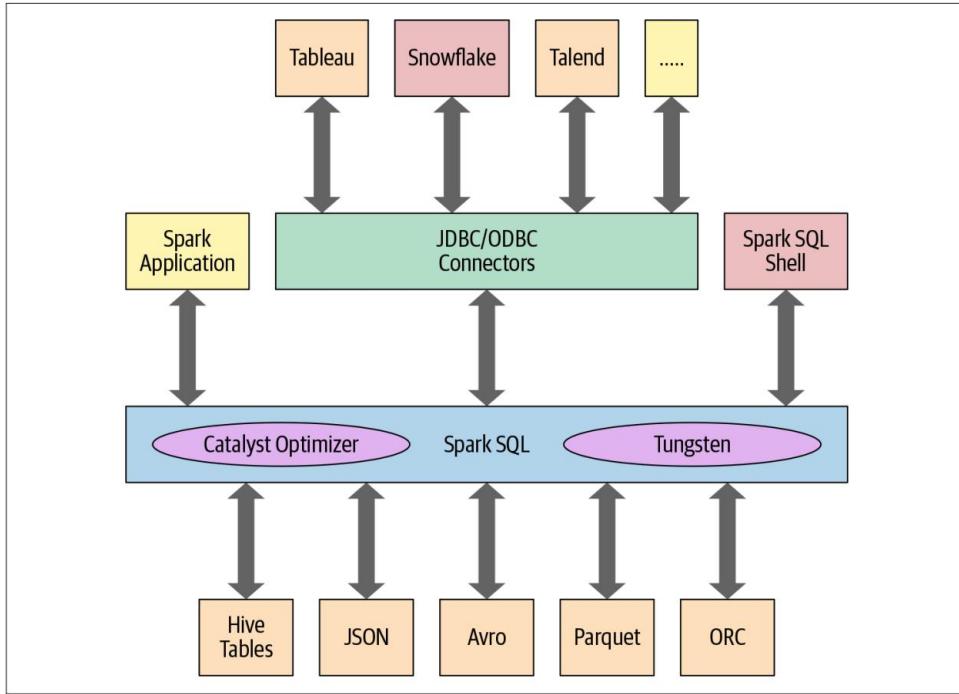


Figura 3-3. Spark SQL y su pila

En el núcleo del motor Spark SQL se encuentran el optimizador Catalyst y Project Tungsten. Juntos, estos admiten las API de DataFrame y Dataset de alto nivel y las consultas SQL. Hablaremos más sobre tungsteno en el [Capítulo 6](#); por ahora, echemos un vistazo más de cerca al optimizador.

#### El optimizador Catalyst El

optimizador Catalyst toma una consulta computacional y la convierte en un plan de ejecución. Pasa por **cuatro fases de transformación**, como se muestra en la [Figura 3-4](#):

1. Análisis 2.

Optimización lógica 3.

Planificación física 4.

Generación de código

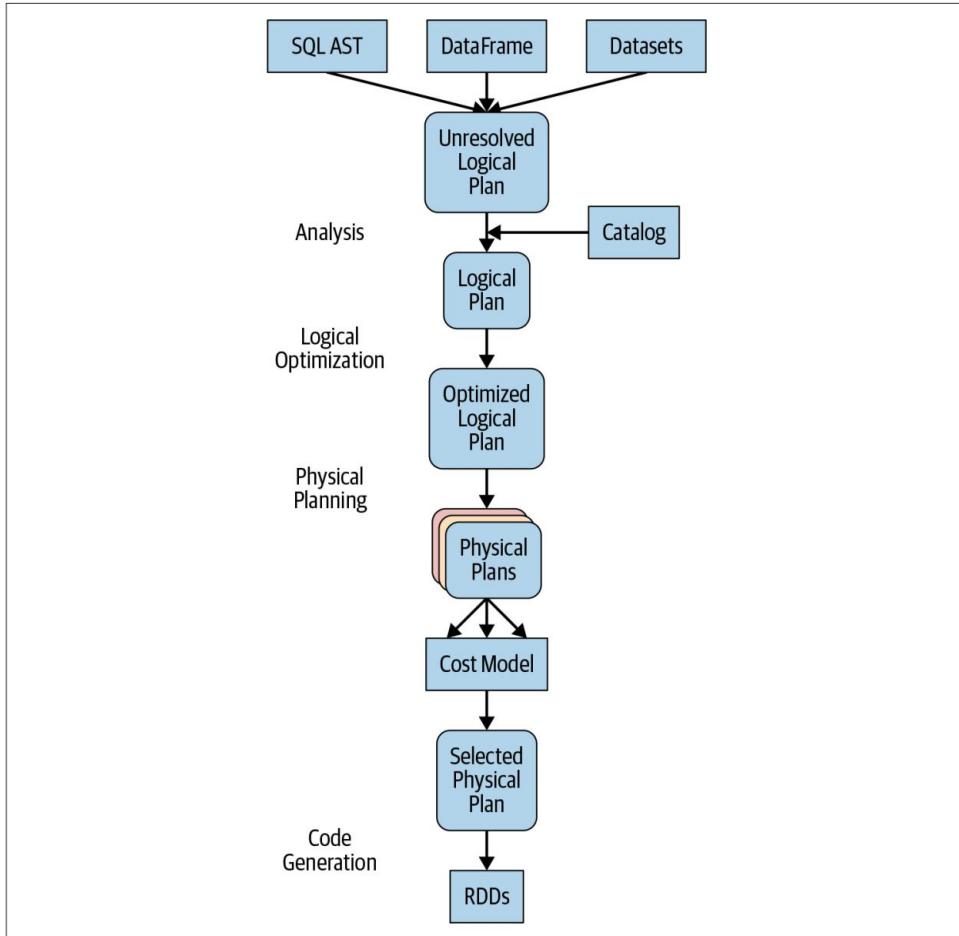


Figura 3-4. El viaje de cuatro fases de un cálculo de Spark

Por ejemplo, considere una de las consultas de nuestro ejemplo de M&Ms en el [Capítulo 2](#). Los dos bloques de código de muestra siguientes pasarán por el mismo proceso y finalmente terminarán con un plan de consulta similar y un código de bytes idéntico para la ejecución. Es decir, independientemente del idioma que use, su cálculo se somete al mismo viaje y el código de bytes resultante probablemente sea el mismo:

```

# En Python
count_mnm_df =
    (mnm_df .select("Estado", "Color",
    "Recuento") .groupBy("Estado",
    "Color") .agg(recuento("Recuento") .alias("Total")) .orderBy("Total", ascendeante=False))
    
```

```
-- En SQL
SELECCIONE Estado, Color, Conteo, suma(Conteo) AS Total
DE MNM_TABLE_NAME
GRUPO POR Estado, Color, Conteo
ORDEN POR Total DESC
```

Para ver las diferentes etapas por las que pasa el código de Python, puede usar el método `count_mnm_df.explain(True)` en el DataFrame. O, para ver los diferentes planes lógicos y físicos, en Scala puede llamar a `df.queryExecution.logical` o `df.queryExecution.optimizedPlan`. (En el [Capítulo 7](#), discutiremos más sobre cómo ajustar y depurar Spark y cómo leer los planes de consulta). Esto nos da el siguiente resultado:

```
count_mnm_df.explain (Verdadero)

== Plan lógico analizado ==
'Ordenar [Total DESC NULLS ÚLTIMO], verdadero
+- Agregado [Estado n.º 10, Color n.º 11], [Estado n.º 10, Color n.º 11, recuento (Recuento n.º 12) COMO... ]
  +- Proyecto [Estado#10, Color#11, Conteo#12]
    +- Relación[Estado#10,Color#11,Recuento#12] csv

== Plan lógico analizado == Estado:
cadena, Color: cadena, Total: bigint Ordenar [Total#24L
DESC NULLS LAST], verdadero +- Agregar [Estado#10,
Color#11], [Estado#10, Color#11 , contar(Cuenta#12) COMO...]
  +- Proyecto [Estado#10, Color#11, Conteo#12]
    +- Relación[Estado#10,Color#11,Recuento#12] csv

== Plan lógico optimizado == Ordenar
[Total#24L DESC NULLS ÚLTIMO], verdadero +- Agregado [Estado#10, Color#11], [Estado#10, Color#11, recuento(Recuento#12) COMO...]
  +- Relación[Estado#10,Color#11,Recuento#12] csv

== Plano físico ==
*(3) Ordenar [Total#24L DESC NULLS LAST], verdadero, 0
+- Partición de rango de intercambio (Total#24L DESC NULLS LAST, 200)
  +- *(2) HashAggregate(teclas=[Estado#10, Color#11], funciones=[recuento(Recuento#12)],
  salida=[Estado#10, Color#11, Total#24L])
    +- Intercambio de partición hash (Estado # 10, Color # 11, 200)
      +- *(1) HashAggregate(teclas=[Estado#10, Color#11],
      funciones=[recuento_parcial(Recuento#12)], salida=[Estado#10, Color#11, recuento#29L])
        +- *(1) FileScan csv [Estado#10,Color#11,Recuento#12] Lote: falso,
Formato: CSV, Ubicación:
InMemoryFileIndex[file:/Users/jules/gits/LearningSpark2.0/chapter2/py/src/... dataset.csv], PartitionFilters:
[], PushedFilters: [], ReadSchema: struct<State :cadena,Color:cadena,Cuenta:int>
```

Consideremos otro ejemplo de cálculo de DataFrame. El siguiente código de Scala pasa por un viaje similar a medida que el motor subyacente optimiza sus planes lógicos y físicos:

```
// En escala
// Users DataFrame leído de una tabla de Parquet val
usersDF = ...
// Evento DataFrame leído de una tabla de Parquet val
eventsDF = ...
// Unir dos DataFrames val
joinDF = usuarios
    .join(eventos, usuarios("id") ===
        eventos("uid")) .filter(eventos("fecha") > "2015-01-01")
```

Después de pasar por una fase de análisis inicial, el optimizador Catalyst transforma y reorganiza el plan de consulta, como se muestra en la [Figura 3-5](#).

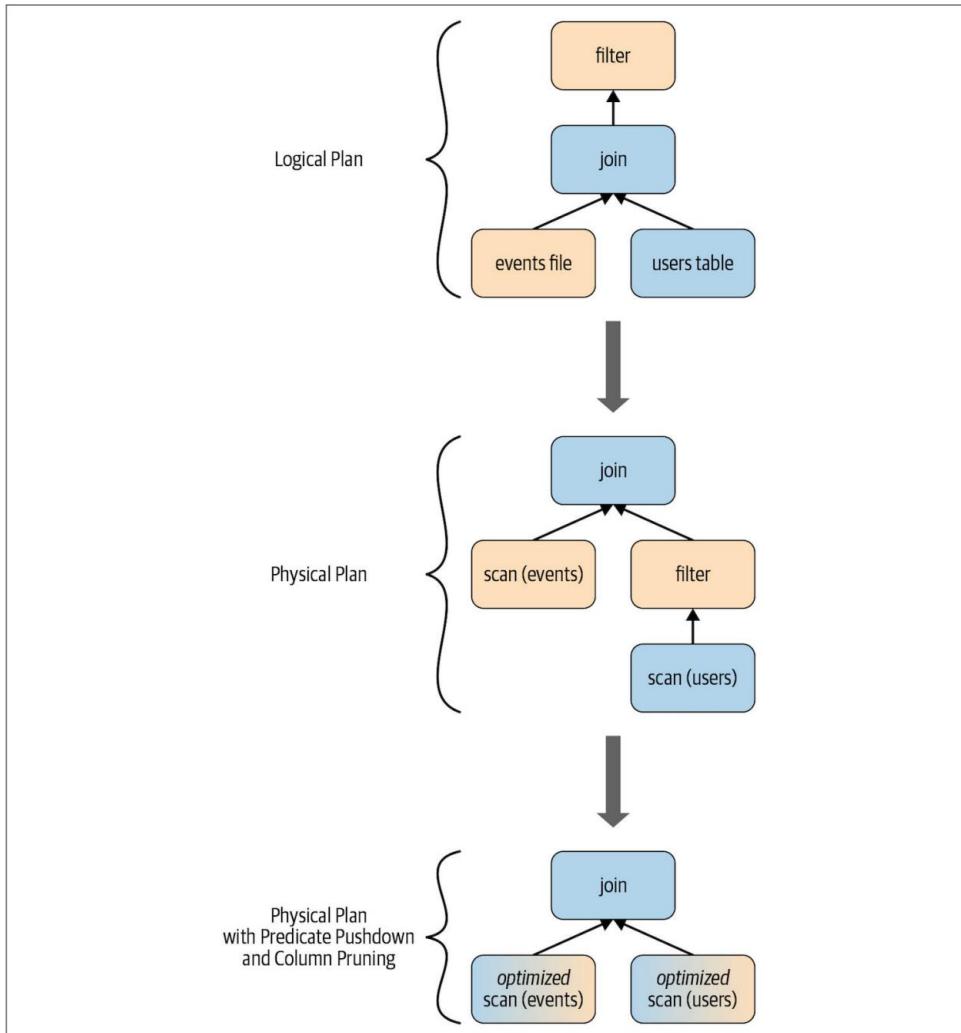


Figura 3-5. Un ejemplo de una transformación de consulta específica

Repasemos cada una de las cuatro fases de optimización de consultas.

#### Fase 1: Análisis

El motor Spark SQL comienza generando un árbol de **sintaxis abstracta (AST)** para la consulta SQL o DataFrame. En esta fase inicial, cualquier columna o nombre de tabla se resolverá consultando un catálogo interno, una interfaz programática para Spark SQL que contiene una lista de nombres de columnas, tipos de datos, funciones, tablas, bases de datos, etc. resuelto con éxito, la consulta pasa a la siguiente fase.

#### Fase 2: Optimización lógica

Como muestra la [Figura 3-4](#), esta fase consta de dos etapas internas. Al aplicar un enfoque de optimización estándar basado en reglas, el optimizador Catalyst primero construirá un conjunto de planes múltiples y luego, utilizando su **optimizador basado en costos (CBO)**, asignar costos a cada plan. Estos planes se presentan como árboles de operadores (como en la [Figura 3-5](#)); pueden incluir, por ejemplo, el proceso de plegamiento constante, reducción de predicados, poda de proyección, simplificación de expresiones booleanas, etc. Este plan lógico es la entrada al plan físico.

#### Fase 3: planificación

**física** En esta fase, Spark SQL genera un plan físico óptimo para el plan lógico seleccionado, utilizando operadores físicos que coinciden con los disponibles en el motor de ejecución de Spark.

#### Fase 4: Generación de

**código** La fase final de la optimización de consultas implica generar código de bytes Java eficiente para ejecutar en cada máquina. Debido a que Spark SQL puede operar en conjuntos de datos cargados en la memoria, Spark puede usar tecnología de compilación de última generación para la generación de código para acelerar la ejecución. En otras palabras, [actúa como un compilador](#). Project Tungsten, que facilita la generación de código de etapa completa, juega un papel aquí.

¿Qué es la generación de código de etapa completa? Es una fase de optimización de consulta física que colapsa toda la consulta en una sola función, eliminando las llamadas a funciones virtuales y empleando registros de CPU para datos intermedios. El motor Tungsten de segunda generación, introducido en Spark 2.0, utiliza este enfoque para generar código RDD compacto para la ejecución final. Esta estrategia simplificada mejora significativamente la eficiencia y el [rendimiento de la CPU](#).



Hemos hablado a nivel conceptual del funcionamiento del motor Spark SQL, con sus dos componentes principales: el optimizador Catalyst y Project Tungsten. El funcionamiento técnico interno está más allá del alcance de este libro; sin embargo, para los curiosos, lo alentamos a consultar las referencias en el texto para obtener discusiones técnicas detalladas.

## Resumen

En este capítulo, profundizamos en las API estructuradas de Spark, comenzando con una mirada a la historia y los méritos de la estructura en Spark.

A través de operaciones de datos comunes ilustrativas y ejemplos de código, demostramos que las API de DataFrame y Dataset de alto nivel son mucho más expresivas e intuitivas que la API de RDD de bajo nivel. Diseñadas para facilitar el procesamiento de grandes conjuntos de datos, las API estructuradas proporcionan operadores específicos de dominio para operaciones de datos comunes, lo que aumenta la claridad y la expresividad de su código.

Exploramos cuándo usar RDD, DataFrames y Datasets, según su uso escenarios de casos.

Y, por último, echamos un vistazo bajo el capó para ver cómo los componentes principales del motor Spark SQL (el optimizador Catalyst y Project Tungsten) son compatibles con las API estructuradas de alto nivel y los operadores DSL. Como vio, sin importar cuál de los lenguajes compatibles con Spark use, una consulta de Spark pasa por el mismo viaje de optimización, desde la construcción del plan lógico y físico hasta la generación de código compacto final.

Los conceptos y ejemplos de código de este capítulo sentaron las bases para los próximos dos capítulos, en los que ilustraremos más la interoperabilidad perfecta entre DataFrames, Datasets y Spark SQL.

## CAPÍTULO 4

# Spark SQL y marcos de datos: Introducción a las fuentes de datos integradas

En el capítulo anterior, explicamos la evolución y la justificación de la estructura en Spark. En particular, discutimos cómo el motor Spark SQL proporciona una base unificada para las API de DataFrame y Dataset de alto nivel. Ahora, continuaremos nuestra discusión sobre DataFrame y exploraremos su interoperabilidad con Spark SQL.

Este capítulo y el siguiente también exploran cómo Spark SQL interactúa con algunos de los componentes externos que se muestran en [la Figura 4-1](#).

En particular, Spark SQL:

- Proporciona el motor sobre el cual se construyen las API estructuradas de alto nivel que exploramos en el [Capítulo 3](#). • Puede leer y escribir datos en una variedad de formatos estructurados (p. ej., JSON, tablas Hive, Parquet, Avro, ORC, CSV).
- Le permite consultar datos utilizando conectores JDBC/ODBC desde fuentes de datos de Business Intelligence (BI) externas como Tableau, Power BI, Talend o RDBMS como MySQL y PostgreSQL.
- Proporciona una interfaz programática para interactuar con datos estructurados almacenados como tablas o vistas en una base de datos desde una aplicación Spark. • Ofrece un shell interactivo para emitir consultas SQL sobre sus datos estructurados. • Compatible [con ANSI Comandos compatibles con SQL:2003 y HiveQL](#).

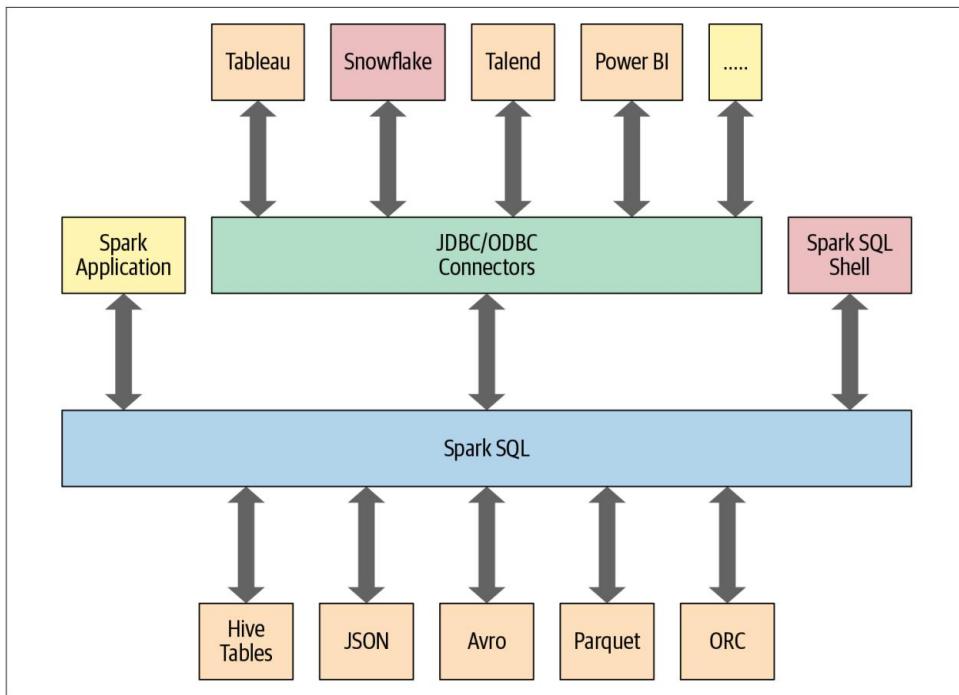


Figura 4-1. Fuentes de datos y conectores Spark SQL

Comencemos con cómo puede usar Spark SQL en una aplicación Spark.

## Uso de Spark SQL en aplicaciones Spark

SparkSession , introducido en Spark 2.0, proporciona un [punto de entrada unificado](#) para programar Spark con las API estructuradas. Puede usar una SparkSession para acceder a la funcionalidad de Spark: simplemente importe la clase y cree una instancia en su código.

Para emitir cualquier consulta SQL, utilice el método `sql()` en la instancia de `SparkSession` , chispa, como `spark.sql("SELECT * FROM myTableName")`. Todas las consultas `spark.sql` ejecutadas de esta manera devuelven un `DataFrame` en el que puede realizar más operaciones de Spark si lo desea, del tipo que exploramos en el [Capítulo 3](#) y sobre las que aprenderá en este capítulo y el siguiente.

## Ejemplos básicos de consultas

En esta sección, veremos algunos ejemplos de consultas sobre el conjunto de datos de Puntualidad de las aerolíneas y Causas de los retrasos en los vuelos, que contiene datos sobre vuelos estadounidenses, incluida la fecha, el retraso, la distancia, el origen y el destino. Está disponible como un archivo CSV con más de un millón de registros. Usando un esquema, leeremos los datos en un DataFrame y registraremos el DataFrame como una vista temporal (más información sobre las vistas temporales en breve) para que podamos consultarlos con SQL.

Los ejemplos de consultas se proporcionan en fragmentos de código, y los cuadernos de Python y Scala que contienen todo el código presentado aquí están disponibles en el repositorio de [GitHub del libro](#).

Estos ejemplos le ofrecerán una idea de cómo usar SQL en sus aplicaciones Spark a través de la interfaz de programación spark.sql. Similar a la API de DataFrame en su estilo declarativo, esta interfaz le permite consultar datos estructurados en sus aplicaciones Spark.

Normalmente, en una aplicación de Spark independiente, creará una instancia de SparkSession manualmente, como se muestra en el siguiente ejemplo. Sin embargo, en un shell de Spark (o cuaderno de bloques de datos), la SparkSession se crea para usted y se puede acceder a ella a través de la variable con el nombre apropiado chispa.

Comencemos leyendo el conjunto de datos en una vista temporal:

```
// En Scala
import org.apache.spark.sql.SparkSession val
spark = 

SparkSession .builder .appName("SparkSQLEExampleApp") .getOrCreate()

// Ruta al conjunto de
datos val csvFile="/databricks-datasets/learning-spark-v2/flights/departuredelays.csv"

// Leer y crear una vista temporal // Inferir
esquema (tenga en cuenta que para archivos más grandes es posible que desee especificar el
esquema) val df = spark.read.format("csv") .option("inferSchema", "true") .option("encabezado",
"veradero") .load(csvFile)

// Crear una vista temporal
df.createOrReplaceTempView("us_delay_flights_tbl")

# En Python
desde pyspark.sql importar SparkSession
# Crear una chispa SparkSession = 

(SparkSession .builder .appName("SparkSQLEExampleApp") .getOrCreate())
```

```
# Ruta al conjunto de
datos csv_file = "/databricks-datasets/learning-spark-v2/flights/departmentdelays.csv"

# Leer y crear una vista temporal # Inferir
esquema (tenga en cuenta que para archivos más
grandes # puede querer especificar el esquema) df =
(spark.read.format("csv")
 .option("inferSchema",
 "true") .option("header",
 "true") .load(csv_file))
df.createOrReplaceTempView("us_delay_flights_tbl")
```



Si desea especificar un esquema, puede usar una cadena con formato DDL. Por ejemplo:

```
// En Scala
val schema = "fecha STRING, retraso INT, distancia INT, origen
STRING, destino STRING"

# En el
esquema de Python = "fecha` STRING, `retraso` INT, `distancia`
INT, `origen` STRING, `destino` STRING "
```

Ahora que tenemos una vista temporal, podemos emitir consultas SQL usando Spark SQL. Estas consultas no son diferentes de las que podría emitir contra una tabla SQL en, por ejemplo, una base de datos MySQL o PostgreSQL. El punto aquí es mostrar que Spark SQL ofrece una interfaz SQL compatible con ANSI: 2003 y demostrar la interoperabilidad entre SQL y DataFrames.

El conjunto de datos de retrasos de vuelos de EE. UU. tiene cinco columnas:

- La columna de fecha contiene una cadena como 02190925. Cuando se convierte, esto se asigna a 02-19 09:25 am.
- La columna de retraso proporciona el retraso en minutos entre el horario programado y el real. horarios de salida. Las salidas anticipadas muestran números negativos.
- La columna de distancia da la distancia en millas desde el aeropuerto de origen hasta el aeropuerto de destino.
- La columna de origen contiene el código de aeropuerto IATA de origen. •

La columna de destino contiene el código de aeropuerto IATA de destino.

Con eso en mente, probemos algunas consultas de ejemplo en este conjunto de datos.

Primero, buscaremos todos los vuelos cuya distancia sea mayor a 1,000 millas:

```
spark.sql("""SELECCIONE distancia, origen, destino DESDE
us_delay_flights_tbl DONDE distancia > 1000 ORDENAR
POR distancia DESC""").show(10)
```

```
+-----+-----+
|distancia|origen|destino|
+-----+-----+
|4330    | HNL | JFK
+-----+-----+
```

solo se muestran las 10 primeras filas

Como muestran los resultados, todos los vuelos más largos fueron entre Honolulu (HNL) y New York (JFK). A continuación, encontraremos todos los vuelos entre San Francisco (SFO) y Chicago (ORD) con al menos dos horas de retraso:

```
spark.sql("""SELECCIONE fecha, demora, origen, destino DESDE
us_delay_flights_tbl DONDE demora > 120 Y ORIGEN = 'SFO' Y
DESTINO = 'ORD'
PEDIDO por retraso DESC""").show(10)
```

```
+-----+-----+
|fecha      |retraso|origen|destino|
+-----+-----+
| 02190925 | 1638 | OFS | ORDEN
| 01031755 | 396  | OFS | ORDEN
| 01022330 | 326  | OFS | ORDEN
| 01051205 | 320  | OFS | ORD
| 01190925 | 297  | OFS | ORD
| 02171115 | 296  | OFS | ORD
| 01071040 | 279  | OFS | ORD
| 01051550 | 274  | OFS | ORD
| 03120730 | 266  | OFS | ORD
| 01261104 | 258  | OFS | ORD
+-----+-----+
```

solo se muestran las 10 primeras filas

Parece que hubo muchos vuelos significativamente retrasados entre estas dos ciudades, en diferentes fechas diferentes. (Como ejercicio, convierta la columna de fecha en un formato legible y busque los días o meses en que estos retrasos fueron más frecuentes. ¿Los retrasos estaban relacionados con meses de invierno o vacaciones?)

Probemos una consulta más complicada donde usamos la cláusula CASE en SQL. A continuación Siguiendo un ejemplo, queremos etiquetar todos los vuelos de EE. UU., independientemente de su origen y destino, con una indicación de los retrasos que experimentaron: Retrasos muy largos (> 6 horas), Retrasos prolongados (2 a 6 horas), etc. Agregaremos estas etiquetas legibles por humanos en una nueva columna llamado Flight\_Delays:

```

spark.sql("""SELECCIONE retraso, origen, destino,
CASO
    CUANDO retraso > 360 ENTONCES 'Retrasos muy largos'
    CUANDO retraso > 120 Y retraso < 360 ENTONCES 'Retrasos largos'
    CUANDO retraso > 60 Y retraso < 120 ENTONCES 'Retrasos cortos'
    CUANDO retraso > 0 y retraso < 60 ENTONCES 'Retrasos tolerables'
    CUANDO retraso = 0 ENTONCES 'Sin retrasos'
    ELSE 'Temprano'
TERMINAR COMO
Flight_Delays FROM
us_delay_flights_tbl ORDER BY origin, delay DESC""").show(10)

+-----+-----+
|retraso|origen|destino|Flight_Delays|
+-----+-----+
|333 |ABE |ATL      |Retrasos largos |
|305 |ABE |ATL      |Retrasos largos |
|275 |ABE |ATL      |Retrasos largos |
|257 |ABE |ATL      |Retrasos largos |
|247 |ABE |DTW      |Retrasos largos |
|247 |ABE |ATL      |Retrasos largos |
|219 |ABE |ORDEN   |Retrasos largos |
|211 |ABE |ATL      |Retrasos largos |
|197 |ABE |DTW      |Retrasos largos |
|192 |ABE |ORD      |Retrasos largos |
+-----+-----+
solo se muestran las 10 primeras filas

```

Al igual que con las API DataFrame y Dataset, con la interfaz spark.sql puede realizar operaciones comunes de análisis de datos como las que exploramos en el capítulo anterior.

Los cálculos se someten a un viaje idéntico en el motor Spark SQL (consulte "[The Catalyst Optimizer](#)" en la página 77 en el Capítulo 3 para obtener más detalles), lo que le brinda los mismos resultados.

Las tres consultas SQL anteriores se pueden expresar con una consulta API de marco de datos equivalente. Por ejemplo, la primera consulta se puede expresar en la API de marco de datos de Python como:

```

# En Python
desde pyspark.sql.functions import col, desc
(df.select("distancia", "origen", "destino") .where(col("distancia")
> 1000) .orderBy(desc("distancia" ))).mostrar(10)

# O
(df.select("distancia", "origen", "destino")
.where("distancia >
1000") .orderBy("distancia", ascensente=False).show(10))

```

Esto produce los mismos resultados que la consulta SQL:

```
+-----+-----+
|distancia|origen|destino|
+-----+-----+
|4330    | HNL | JFK   |
+-----+-----+
```

solo se muestran las 10 primeras filas

Como ejercicio, intente convertir las otras dos consultas SQL para usar la API de DataFrame.

Como muestran estos ejemplos, usar la interfaz Spark SQL para consultar datos es similar a escribir una consulta SQL normal en una tabla de base de datos relacional. Aunque las consultas están en SQL, puede sentir la similitud en la legibilidad y la semántica con el funcionamiento de la API de DataFrame. ciones, que encontró en el [Capítulo 3](#) y explorará más a fondo en el próximo capítulo.

Para permitirle consultar datos estructurados como se muestra en los ejemplos anteriores, Spark gestiona todas las complejidades de la creación y gestión de vistas y tablas, tanto en memoria y en disco. Eso nos lleva a nuestro siguiente tema: cómo se crean tablas y vistas. ados y gestionados.

## Tablas y vistas de SQL

Las tablas contienen datos. Asociados con cada tabla en Spark están sus metadatos relevantes, que son información sobre la tabla y sus datos: el esquema, la descripción, el nombre de la tabla, los datos. nombre base, nombres de columna, particiones, ubicación física donde residen los datos reales, etc. Todo esto se almacena en un metastore central.

En lugar de tener un metastore separado para las tablas de Spark, Spark usa de forma predeterminada el Apache Hive metastore, ubicado en /user/hive/warehouse, para conservar todos los metadatos sobre sus mesas. Sin embargo, puede cambiar la ubicación predeterminada configurando el Spark config variable spark.sql.warehouse.dir a otra ubicación, que se puede establecer en un almacenamiento distribuido local o externo.

### Tablas administradas versus no administradas

Spark le permite crear dos tipos de tablas: administradas y no administradas. Para un hombre-tabla envejecida, Spark administra tanto los metadatos como los datos en el almacén de archivos. Esto podría ser un sistema de archivos local, HDFS o un almacén de objetos como Amazon S3 o Azure Blob. Para

una tabla no administrada, Spark solo administra los metadatos, mientras que usted mismo administra los datos en una fuente de **datos** externa como Casandra.

Con una tabla administrada, dado que Spark administra todo, un comando SQL como DROP TABLE table\_name elimina tanto los metadatos como los datos. Con una tabla no administrada, el mismo comando eliminará solo los metadatos, no los datos reales. Veremos algunos ejemplos de cómo crear tablas administradas y no administradas en la siguiente sección.

### **Creación de tablas y bases de datos SQL Las**

tablas residen dentro de una base de datos. De forma predeterminada, Spark crea tablas en la base de datos predeterminada . Para crear su propio nombre de base de datos, puede emitir un comando SQL desde su aplicación o cuaderno Spark. Usando el conjunto de datos de retrasos de vuelos de EE. UU., creemos una tabla administrada y no administrada. Para comenzar, crearemos una base de datos llamada learn\_spark\_db y le diremos a Spark que queremos usar esa base de datos:

```
// En Scala/Python
spark.sql("CREAR BASE DE DATOS learn_spark_db")
spark.sql("UTILIZAR learn_spark_db")
```

Desde este punto, cualquier comando que emitamos en nuestra aplicación para crear tablas dará como resultado que las tablas se creen en esta base de datos y residan bajo el nombre de la base de datos learn\_spark\_db.

### **Crear una tabla administrada**

Para crear una tabla administrada dentro de la base de datos learn\_spark\_db, puede emitir una consulta SQL como la siguiente:

```
// En Scala/Python
spark.sql("CREATE TABLE Managed_us_delay_flights_tbl (fecha STRING, retraso INT,
distancia INT, cadena de origen, cadena de destino)")
```

Puede hacer lo mismo usando la API de DataFrame de esta manera:

```
# En Python
# Ruta a nuestro archivo CSV de retrasos en los
vuelos de EE. UU. csv_file = "/databricks-datasets/learning-spark-v2/flights/departuredelays.csv"
# Esquema como se define en el ejemplo anterior
schema="fecha STRING, retraso INT, distancia INT, origen STRING, destino STRING" vuelos_df =
chispa.read.csv(archivo_csv, esquema=esquema)
vuelos_df.write.saveAsTable("managed_us_delay_flights_tbl")
```

Ambas declaraciones crearán la tabla administrada us\_delay\_flights\_tbl en la base de datos learn\_spark\_db .

## Creación de una tabla no administrada

**administrada** Por el contrario, puede crear tablas no administradas a partir de sus propias fuentes de datos, por ejemplo, archivos Parquet, CSV o JSON almacenados en un almacén de archivos accesible para su aplicación Spark.

Para crear una tabla no administrada a partir de una fuente de datos, como un archivo CSV, en uso de SQL:

```
spark.sql("""CREATE TABLE us_delay_flights_tbl(fecha STRING, retraso INT,
    distancia INT, CADENA de origen, CADENA de destino)
USO DE LAS OPCIONES csv
(RUTA '/databricks-datasets/learning-spark-v2/flights/departuredelays.csv')""")
```

Y dentro del uso de la API DataFrame:

```
(flights_df .write .option("ruta", "/tmp/data/
us_flights_delay") .saveAsTable("us_delay_flights_tbl"))
```



Para permitirle explorar estos ejemplos, hemos creado cuadernos de ejemplo de Python y Scala que puede encontrar en el repositorio de [Git-Hub del libro](#).

## Creación de vistas

Además de crear tablas, Spark puede crear vistas sobre tablas existentes. Las vistas pueden ser globales (visibles en todas las SparkSessions en un clúster determinado) o de ámbito de sesión (visibles solo para una única SparkSession) y son temporales: desaparecen después de que finaliza su aplicación Spark.

**Creación de vistas** tiene una sintaxis similar a la creación de tablas dentro de una base de datos. Una vez que crea una vista, puede consultarla como lo haría con una tabla. La diferencia entre una vista y una tabla es que las vistas en realidad no contienen los datos; las tablas persisten después de que finaliza su aplicación Spark, pero las vistas desaparecen.

Puede crear una vista a partir de una tabla existente utilizando SQL. Por ejemplo, si desea trabajar solo en el subconjunto del conjunto de datos de retrasos de vuelos de EE. UU. con los aeropuertos de origen de Nueva York (JFK) y San Francisco (SFO), las siguientes consultas crearán vistas globales temporales y temporales que consisten solo en ese pedazo de la mesa:

```
-- En SQL
CREAR O REEMPLAZAR LA VISTA TEMPERATURA GLOBAL us_origin_airport_SFO_global_tmp_view COMO
    SELECCIONE fecha, retraso, origen, destino desde us_delay_flights_tbl WHERE origen =
        'SFO';
```

```
CREAR O REEMPLAZAR TEMP VIEW us_origin_airport_JFK_tmp_view COMO
    SELECCIONE fecha, retraso, origen, destino desde us_delay_flights_tbl DONDE origen =
        'JFK'
```

Puede lograr lo mismo con la API de DataFrame de la siguiente manera:

```
# En Python
df_sfo = spark.sql("SELECCIONE fecha, demora, origen, destino DESDE
                   us_delay_flights_tbl DONDE origen = 'SFO'")
df_jfk = spark.sql("SELECCIONE fecha, demora, origen, destino DESDE
                   us_delay_flights_tbl DONDE origen = 'JFK'")

# Crear una vista temporal y temporal global
df_sfo.createOrReplaceGlobalTempView("us_origin_airport_SFO_global_tmp_view")
df_jfk.createOrReplaceTempView("us_origin_airport_JFK_tmp_view")
```

Una vez que haya creado estas vistas, puede realizar consultas en ellas como lo haría con una tabla. Tenga en cuenta que al acceder a una vista temporal global debe usar el prefijo *global\_temp.<view\_name>*, porque Spark crea vistas temporales globales en una base de datos temporal global llamada *global\_temp*. Por ejemplo:

```
-- En SQL
SELECCIONE * DESDE global_temp.us_origin_airport_SFO_global_tmp_view
```

Por el contrario, puede acceder a la vista temporal normal sin el prefijo *global\_temp*:

```
-- En SQL
SELECCIONE * DESDE us_origin_airport_JFK_tmp_view

// En Scala/Python
spark.read.table("us_origin_airport_JFK_tmp_view")
// O
spark.sql("SELECCIONE * DESDE us_origin_airport_JFK_tmp_view")
```

También puede soltar una vista como lo haría con una tabla:

```
-- En SQL
ABANDONAR LA VISTA SI EXISTE us_origin_airport_SFO_global_tmp_view;
ABANDONAR LA VISTA SI EXISTE us_origin_airport_JFK_tmp_view

// En Scala/Python
spark.catalog.dropGlobalTempView("us_origin_airport_SFO_global_tmp_view")
spark.catalog.dropTempView("us_origin_airport_JFK_tmp_view")
```

### Vistas temporales versus vistas temporales

**globales** La diferencia entre vistas temporales y temporales globales es sutil, puede ser una fuente de confusión leve entre los desarrolladores nuevos en Spark. Una vista temporal está vinculada a una sola SparkSession dentro de una aplicación Spark. Por el contrario, una vista temporal global es visible en varias SparkSessions dentro de una aplicación Spark. Sí, puede crear **múltiples SparkSessions** dentro de una sola aplicación Spark; esto puede ser útil, por ejemplo, en los casos en los que desea acceder (y combinar) datos de dos SparkSessions diferentes que no comparten las mismas configuraciones de metastore de Hive.

## Ver los metadatos

Como se

mencionó anteriormente, Spark administra los metadatos asociados con cada tabla administrada o no administrada. Esto se captura en el [Catálogo](#), una abstracción de alto nivel en Spark SQL para almacenar metadatos. La funcionalidad del Catálogo se amplió en Spark 2.x con nuevos métodos públicos que le permiten examinar los metadatos asociados con sus bases de datos, tablas y vistas. Spark 3.0 lo amplía para usar un catálogo externo (que analizamos brevemente en el [Capítulo 12](#)).

Por ejemplo, dentro de una aplicación Spark, después de crear la chispa variable `SparkSession` , puede acceder a todos los metadatos almacenados a través de métodos como estos:

```
// En Scala/Python
spark.catalog.listDatabases()
spark.catalog.listTables()
spark.catalog.listColumns("us_delay_flights_tbl")
```

Importa el cuaderno desde el repositorio de [GitHub](#) del libro. y pruébalo.

## Almacenamiento en caché de

**tablas SQL** Aunque discutiremos las estrategias de almacenamiento en caché de tablas en el próximo capítulo, vale la pena mencionar aquí que, al igual que los marcos de datos, puede almacenar y recuperar tablas y vistas de SQL.

En [Chispa 3.0](#), Además de otras opciones, puede especificar una tabla como **LAZY**, lo que significa que solo debe almacenarse en caché cuando se usa por primera vez en lugar de inmediatamente:

```
-- En SQL
CACHE [LAZY] TABLE <nombre-tabla>
UNCACHE TABLE <nombre-tabla>
```

## Lectura de tablas en marcos de datos

A menudo,

los ingenieros de datos construyen canalizaciones de datos como parte de sus procesos regulares de ingestión de datos y ETL. Llenan las bases de datos y las tablas de Spark SQL con datos limpios para el consumo de las aplicaciones posteriores.

Supongamos que tiene una base de datos existente, `learn_spark_db`, y una tabla, `us_delay_flights_tbl`, listas para usar. En lugar de leer desde un archivo JSON externo, simplemente puede usar SQL para consultar la tabla y asignar el resultado devuelto a un DataFrame:

```
// En Scala val
usFlightsDF = spark.sql("SELECT * FROM us_delay_flights_tbl") val usFlightsDF2 =
spark.table("us_delay_flights_tbl")
```

```
# En Python
us_flights_df = chispa.sql("SELECCIONE * DE us_delay_flights_tbl")
us_flights_df2 = chispa.table("us_delay_flights_tbl")
```

Ahora tiene una lectura de DataFrame limpia de una tabla Spark SQL existente. También puede leer datos en otros formatos utilizando las fuentes de datos integradas de Spark, lo que le brinda la flexibilidad de interactuar con varios formatos de archivo comunes.

## Orígenes de datos para marcos de datos y tablas SQL

Como se muestra en [la Figura 4-1](#), Spark SQL proporciona una interfaz para una variedad de fuentes de datos. También proporciona un conjunto de métodos comunes para leer y escribir datos hacia y desde estas fuentes de datos mediante la [API de fuentes de datos](#).

En esta sección cubriremos algunas de las [fuentes de datos integradas](#), formatos de archivo disponibles y formas de cargar y escribir datos, junto con opciones específicas relacionadas con estas fuentes de datos. Pero primero, echemos un vistazo más de cerca a dos construcciones API de fuente de datos de alto nivel que dictan la forma en que interactúa con diferentes fuentes de datos: DataFrameReader y DataFrameWriter.

### Lector de tramas de datos

[Lector de tramas de datos](#) es la construcción central para leer datos de una fuente de datos en un DataFrame. Tiene un formato definido y un patrón recomendado de uso:

```
DataFrameReader.format(args).option("clave", "valor").schema(args).load()
```

Este patrón de encadenar métodos es común en Spark y fácil de leer. Lo vimos en el [Capítulo 3](#) cuando exploramos patrones comunes de análisis de datos.

Tenga en cuenta que solo puede acceder a un DataFrameReader a través de una instancia de SparkSession . Es decir, no puede crear una instancia de DataFrameReader. Para obtener un identificador de instancia, use:

```
SparkSession.read //
o
SparkSession.readStream
```

Mientras que read devuelve un identificador a DataFrameReader para leer en un DataFrame desde una fuente de datos estática, readStream devuelve una instancia para leer desde una fuente de transmisión. (Cubriremos la transmisión estructurada más adelante en el libro).

Los argumentos de cada uno de los métodos públicos de DataFrameReader toman valores diferentes.

[La Tabla 4-1](#) los enumera, con un subconjunto de los argumentos admitidos.

Tabla 4-1. Métodos, argumentos y opciones de DataFrameReader

Método	Argumentos	Descripción
format() "parquet", "csv", "txt", "json", "jdbc", "orc", "avro", etc.		Si no especifica este método, el valor predeterminado es Parquet o lo que esté configurado en spark.sql.sources.default.
option() ("modo", {PERMISIVO   FAILFAST   DROPMALFORMED }) ("inferSchema", {verdadero   falso}) ("ruta", "ruta_archivo_fuente_datos")		Una serie de pares clave/valor y opciones. La documentación de Spark muestra algunos ejemplos y explica los diferentes modos y sus acciones. El modo predeterminado es PERMISIVO. El "inferSchema" y las opciones de "modo" son específicas de los formatos de archivo JSON y CSV.
schema() DDL String o StructType, por ejemplo, 'A INT, B STRING' o StructType(...)		Para el formato JSON o CSV, puede especificar inferir el esquema en el método option(). En general, proporcionar un esquema para cualquier formato hace que la carga sea más rápida y garantiza que sus datos se ajusten al esquema esperado.
carga() "/ruta/a/datos/fuente"		La ruta al origen de datos. Esto puede estar vacío si se especifica en la opción ("ruta", "...").

Si bien no enumeraremos exhaustivamente todas las diferentes combinaciones de argumentos y opciones, la [documentación para Python, Scala, R y Java](#) ofrece sugerencias y orientación. Sin embargo, vale la pena mostrar un par de ejemplos:

```
// En scala
// Use el archivo
val de Parquet = """/databricks-datasets/learning-spark-v2/flights/summary-data/parquet/2010-summary.parquet"""
val df = chispa.read.format("parquet").load(archivo)
// Usar Parquet; puede omitir el formato ("parquet") si lo desea, ya que es el valor predeterminado
df2 = spark.read.load(archivo)
// Usar CSV
val df3 =
  spark.read.format("csv") .option("inferSchema",
  "true") .option("header",
  "true") .option("mode",
  "PERMISSIVE") .load("/databricks-conjuntos de datos/learning-spark-v2/flights/summary-data/csv/*")
// Usa JSON
val df4 = spark.read.format("json")
.load("/databricks-conjuntos de datos/learning-spark-v2/flights/summary-data/json/*")
```



En general, no se necesita ningún esquema cuando se lee desde una fuente de datos estática de Parquet; los metadatos de Parquet generalmente contienen el esquema, por lo que se deduce. Sin embargo, para las fuentes de datos de transmisión, deberá proporcionar un esquema. (Cubriremos la lectura de fuentes de datos de transmisión en el [Capítulo 8](#)).

Parquet es la fuente de datos predeterminada y preferida para Spark porque es eficiente, usa almacenamiento en columnas y emplea un algoritmo de compresión rápido. Verá beneficios adicionales más adelante (como la reducción de columnas), cuando cubramos el optimizador Catalyst con mayor profundidad.

### **Escriptor de tramas de datos**

**Escriptor de tramas de datos** hace lo contrario de su contraparte: guarda o escribe datos en una fuente de datos integrada especificada. A diferencia de DataFrameReader, no accede a su instancia desde una SparkSession sino desde el DataFrame que desea guardar. Tiene algunos patrones de uso recomendados:

```
DataFrameWriter.format(argumentos) .opción(argumentos) .bucketBy(argumentos) .partitionBy(argumentos) .save(ruta)
```

```
DataFrameWriter.format(argumentos).opción(argumentos).sortBy(argumentos).saveAsTable(tabla)
```

Para obtener un identificador de instancia, use:

```
DataFrame.escribir  
// o  
DataFrame.writeStream
```

Los argumentos de cada uno de los métodos de DataFrameWriter también toman valores diferentes. Los enumeramos en [la Tabla 4-2](#), con un subconjunto de los argumentos admitidos.

**Tabla 4-2. Métodos, argumentos y opciones de DataFrameWriter**

Método	Argumentos	Descripción
format()	"parquet", "csv", "txt", "json", "jdbc", "orc", "avro", etc.	Si no especifica este método, el valor predeterminado es Parquet o lo que esté configurado en spark.sql.sources.default.
option()	("modo", {agregar   sobrescribir   ignorar   error o errorifex ists}) ("modo", {SaveMode.Overwrite   SaveMode.Append, SaveMode.Ignore, SaveMode.Error   fExists}) ("ruta" , "ruta_a_escribir_a")	Una serie de pares clave/valor y opciones. La <a href="#">documentación de Spark</a> muestra algunos ejemplos. Este es un método sobrecargado. Las opciones de modo predeterminadas son error o error ifexists y SaveMode.ErrorIfExists; lanzan una excepción en tiempo de ejecución si los datos ya existen.

Método	Argumentos	Descripción
dólar	(numBuckets, col, col..., coln)	número de cubos y los nombres de las columnas por los que se van a colocar los cubos.
etBy()		Utiliza el esquema de depósito de Hive en un sistema de archivos.
guardar()	"/ruta/a/datos/fuente"	La ruta para guardar. Esto puede estar vacío si se especifica en la opción ("ruta", "...").
GuardarAsTa se convirtió)	"nombre de la tabla"	La tabla para guardar.

Aquí hay un breve fragmento de ejemplo para ilustrar el uso de métodos y argumentos:

```
// En Scala //
Usar JSON
val ubicación = ...
df.write.format("json").mode("overwrite").save(ubicación)
```

## Parquet

Comenzaremos nuestra exploración de fuentes de datos con **Parquet**, porque es la fuente de datos predeterminada en Spark. Apoyado y ampliamente utilizado por muchos marcos y plataformas de procesamiento de big data, Parquet es un formato de archivo en columnas de código abierto que ofrece muchas optimizaciones de E/S (como la compresión, que ahorra espacio de almacenamiento y permite un acceso rápido a las columnas de datos).

Debido a su eficiencia y estas optimizaciones, recomendamos que después de haber transformado y limpiado sus datos, guarde sus DataFrames en formato Parquet para consumo posterior. (El parquet también es el formato abierto de mesa predeterminado para Delta Lake, que trataremos en el [Capítulo 9](#)).

### Lectura de archivos de Parquet en un

**DataFrame Archivos de Parquet** se almacenan en una estructura de directorio que contiene los archivos de datos, metadatos, varios archivos comprimidos y algunos archivos de estado. Los metadatos en el pie de página contienen la versión del formato de archivo, el esquema y los datos de la columna, como la ruta, etc.

Por ejemplo, un directorio en un archivo Parquet podría contener un conjunto de archivos como este:

```
_SUCCESS
_committed_1799640464332036264
_started_1799640464332036264 part-00000-tid-1799640464332036264-91273258-d7ef-4dc7-<...>-c000.snappy.parquet
```

Puede haber varios archivos comprimidos parte XXXX en un directorio (los nombres que se muestran aquí se han abreviado para que quepan en la página).

Para leer archivos de Parquet en un DataFrame, simplemente especifique el formato y la ruta:

```
// En escala
val file = """/databricks-datasets/learning-spark-v2/flights/summary-data/
    parquet/2010-summary.parquet"""
val df = spark.read.format("parquet").load(archivo)

# en pitón
file = """/databricks-datasets/learning-spark-v2/flights/summary-data/parquet/
    2010-resumen.parquet"""
df = chispa.read.format("parquet").load(archivo)
```

A menos que esté leyendo desde una fuente de datos de transmisión, no es necesario proporcionar el esquema, porque Parquet lo guarda como parte de sus metadatos.

#### Lectura de archivos de Parquet en una tabla Spark SQL

Además de leer archivos de Parquet en un Spark DataFrame, también puede crear un Spark Tabla no administrada de SQL o vista directamente usando SQL:

```
-- En SQL
CREAR O REEMPLAZAR VISTA TEMPORAL us_delay_flights_tbl
    UTILIZAR parquet
    OPCIONES (
        ruta "/databricks-datasets/learning-spark-v2/flights/summary-data/parquet/
            2010-resumen.parquet" )
```

Una vez que haya creado la tabla o vista, puede leer datos en un DataFrame usando SQL, como vimos en algunos ejemplos anteriores:

```
// En escala
chispa.sql("SELECCIONE * DE us_delay_flights_tbl").show()

# en pitón
chispa.sql("SELECCIONE * DE us_delay_flights_tbl").show()
```

Ambas operaciones devuelven los mismos resultados:

DEST_COUNTRY_NAME ORIGIN_COUNTRY_NAME recuento
Estados Unidos  Rumanía  1
Estados Unidos  Irlanda  264
Estados Unidos  India  69
Egipto  Estados Unidos  24
Guinea Ecuatorial Estados Unidos  1
Estados Unidos  Singapur  25
Estados Unidos  Granada  54
Costa Rica  Estados Unidos  477
Senegal  Estados Unidos  29
Estados Unidos  Islas Marshall  44

solo se muestran las 10 primeras filas

## Escribir marcos de datos en archivos

**de Parquet** Escribir o guardar un marco de datos como una tabla o archivo es una operación común en Spark. Para escribir un DataFrame, simplemente use los métodos y argumentos del DataFrame Writer descritos anteriormente en este capítulo, proporcionando la ubicación para guardar los archivos de Parquet. Por ejemplo:

// En Scala

```
df.write.format("parquet") .mode("overwrite") .option("compression", "snappy") .save("/tmp/data/parquet/df_parquet")
```

# En Python

```
(df.write.format("parquet") .mode("overwrite") .option("compression", "snappy") .save("/tmp/data/parquet/df_parquet"))
```



Recuerde que Parquet es el formato de archivo predeterminado. Si no incluye el método format() , el DataFrame aún se guardará como un archivo Parquet.

Esto creará un conjunto de archivos Parquet compactos y comprimidos en la ruta especificada.

Dado que aquí usamos Snappy como nuestra opción de compresión, tendremos archivos comprimidos Snappy. Para abreviar, este ejemplo generó solo un archivo; normalmente, puede haber una docena o más de archivos creados:

```
-rw-r--r-- 1 jules wheel 0 19 de mayo 10:58 _SUCCESS -rw-r--r--  
1 jules wheel 966 19 de mayo 10:58 part-00000-<...>-c000.snappy .parquet
```

## Escribir DataFrames en tablas Spark

**SQL** Escribir un DataFrame en una tabla SQL es tan fácil como escribir en un archivo: solo use saveAsTable() en lugar de save(). Esto creará una tabla administrada llamada us\_delay\_flights\_tbl:

// En Scala

```
df.write .mode("overwrite") .saveAsTable("us_delay_flights_tbl")
```

# En Python

```
(df.write .mode("overwrite") .saveAsTable("us_delay_flights_tbl"))
```

Para resumir, Parquet es el formato de archivo de origen de datos integrado preferido y predeterminado en Spark, y ha sido adoptado por muchos otros marcos. Te recomendamos que use este formato en sus procesos de ingesta de datos y ETL.

## JSON

La notación de objetos de JavaScript (JSON) también es un formato de datos popular. Llegó a prominencia como un formato fácil de leer y analizar en comparación con XML. Tiene dos representaciones: **modo unifilar** y **modo multilínea**. Ambos modos son compatibles con Spark.

En el modo de una sola línea, **cada línea denota un solo objeto JSON**, mientras que en multilínea modo, todo el objeto multilínea constituye un solo objeto JSON. Para leer en este modo, establezca multiLine en verdadero en el método option().

### Leer un archivo JSON en un DataFrame

Puede leer un archivo JSON en un DataFrame de la misma manera que lo hizo con Parquet, solo especifique "json" en el método format():

```
// En escala
val file = "/databricks-datasets/learning-spark-v2/flights/summary-data/json/*"
val df = spark.read.format("json").load(archivo)

# en pitón
archivo = "/bloques de datos-conjuntos de datos/aprendizaje-chispa-v2/vuelos/resumen-datos/json/*"
df = spark.read.format("json").load(archivo)
```

### Leer un archivo JSON en una tabla Spark SQL

También puede crear una tabla SQL a partir de un archivo JSON como lo hizo con Parquet:

```
-- En SQL
CREAR O REEMPLAZAR VISTA TEMPORAL us_delay_flights_tbl
    UTILIZANDO json
    OPCIONES (
        ruta "/databricks-datasets/learning-spark-v2/flights/summary-data/json/*"
    )
```

Una vez que se crea la tabla, puede leer datos en un DataFrame usando SQL:

```
// En Scala/Python
chispa.sql("SELECCIONE * DE us_delay_flights_tbl").show()

+-----+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|recuento|
+-----+-----+-----+
|Estados Unidos    |Rumania          |15 |
|Estados Unidos    |Croacia           |1 |
|Estados Unidos    |Irlanda           |344 |
|Egipto            |Estados Unidos   |15 |
```

Estados Unidos	India	62
Estados Unidos	Singapur	1
Estados Unidos	Granada	62
Costa Rica	Estados Unidos	588
Senegal	Estados Unidos	40
Moldavia	Estados Unidos	1
+-----+-----+-----+		

solo se muestran las 10 primeras filas

#### Escribir tramas de datos en archivos JSON

Guardar un DataFrame como un archivo JSON es simple. Especifique el apropiado

Métodos y argumentos de DataFrameWriter , y proporcione la ubicación para guardar el JSON archivos a:

```
// En escala
df.write.format("json")
    .mode("sobrescribir")
    .option("compresión", "rápido")
    .save("/tmp/datos/json/df_json")

# en pitón
(df.write.format("json")
    .mode("sobrescribir")
    .option("compresión", "rápido")
    .save("/tmp/datos/json/df_json"))
```

Esto crea un directorio en la ruta especificada con un conjunto de JSON compacto archivos:

```
-rw-r--r-- 1 rueda de julio 0 16 de mayo 14:44 _ÉXITO
-rw-r--r-- 1 jules rueda 71 16 de mayo 14:44 part-00000-<...>-c000.json
```

#### Opciones de fuente de datos JSON

La Tabla 4-3 describe las opciones comunes de JSON para DataFrameReader y marco de datos

Escritor. Para obtener una lista completa, lo remitimos a la documentación.

Tabla 4-3. Opciones de JSON para DataFrameReader y DataFrameWriter

Nombre de la propiedad	Valores	Significado	Alcance
compresión	ninguno, sin comprimir, bzip2, desinflar, gzip, lz4, o rápido	Utilice este códec de compresión para escribir. Tenga en cuenta que leer solo detectará la compresión o el códec del archivo extensión.	Escribe
formato de fecha	aaaa-MM-dd o DateTime yoFormatter	Use este formato o cualquier formato de DateTime de Java Formateador.	Leer/ escribe
multilínea	cierto, falso	Utilice el modo multilínea. El valor predeterminado es falso (una sola línea modo).	Leer
permitir Sin comillas Nombres de campo	cierto, falso	Permitir nombres de campo JSON sin comillas. El valor predeterminado es falso. Leer	

## CSV

Tan utilizado como los archivos de texto sin formato, este formato de archivo de texto común captura cada dato o campo delimitado por una coma; cada línea con campos separados por comas representa un registro. Aunque una coma es el separador predeterminado, puede usar otros delimitadores para separar campos en los casos en que las comas sean parte de sus datos. Las hojas de cálculo populares pueden generar archivos CSV, por lo que es un formato popular entre los analistas de datos y de negocios.

### Leer un archivo CSV en un DataFrame

Al igual que con las otras fuentes de datos integradas, puede usar los métodos y argumentos de DataFrameReader para leer un archivo CSV en un DataFrame:

```
// En Scala val
file = "/databricks-datasets/learning-spark-v2/flights/summary-data/csv/**" val schema = "DEST_COUNTRY_NAME
STRING, ORIGIN_COUNTRY_NAME STRING, count INT"

val df =


// Salir si hay algún error
// Reemplaza cualquier dato nulo con comillas
spark.read.format("csv") .schema(schema) .option("header", "true") .option("mode", "FAILFAST") .option("nullValue", "") .cargar archivo

# En Python
file = "/databricks-datasets/learning-spark-v2/flights/summary-data/csv/**" schema =
"DEST_COUNTRY_NAME STRING, ORIGIN_COUNTRY_NAME STRING, count INT" df =
(spark.read.format("csv ") .option("header", "true") .schema(schema) .option("mode", "FAILFAST") # Salir
si hay algún error .option("nullValue", "")


# Reemplazar cualquier campo de datos nulos con
comillas .load(archivo))
```

### Leer un archivo CSV en una tabla Spark SQL

Crear una tabla SQL a partir de una fuente de datos CSV no es diferente de usar Parquet o JSON:

```
-- En SQL
CREAR O REEMPLAZAR VISTA TEMPORAL us_delay_flights_tbl
USAR csv
OPCIONES
( ruta "/databricks-datasets/learning-spark-v2/flights/summary-data/csv/**", encabezado "true",
inferSchema "true", modo "FAILFAST"

)
```

Una vez que haya creado la tabla, puede leer datos en un DataFrame usando SQL como antes:

```
// En Scala/Python
chispa.sql("SELECCIONE * DE us_delay_flights_tbl").show(10)

+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|recuento|
+-----+-----+
|Estados Unidos    |Rumania          |1 |
|Estados Unidos    |Irlanda           |264 |
|Estados Unidos    |India             |69 |
|Egipto            |Estados Unidos   |24 |
|Guinea Ecuatorial|Estados Unidos   |1 |
|Estados Unidos    |Singapur         |25 |
|Estados Unidos    |Granada          |54 |
|Costa Rica        |Estados Unidos   |477 |
|Senegal           |Estados Unidos   |29 |
|Estados Unidos    |Islas Marshall   |44 |
+-----+
solo se muestran las 10 primeras filas
```

#### Escribir tramas de datos en archivos CSV

Guardar un DataFrame como un archivo CSV es simple. Especifique el DataFrameWriter apropiado métodos y argumentos, y proporcione la ubicación para guardar los archivos CSV en:

```
// En escala
df.write.format("csv").mode("overwrite").save("/tmp/data/csv/df_csv")

# en pitón
df.write.format("csv").mode("overwrite").save("/tmp/data/csv/df_csv")
```

Esto genera una carpeta en la ubicación especificada, poblada con un montón de comÿ archivos comprimidos y compactos:

```
-rw-r--r-- 1 rueda de julio 0 16 de mayo 12:17 _ÉXITO
-rw-r--r-- 1 jules rueda 36 16 de mayo 12:17 part-00000-251690eb-<...>-c000.csv
```

#### Opciones de fuente de datos CSV

La Tabla 4-4 describe algunas de las opciones comunes de CSV para [DataFrameReader](#) y [datos](#)

**Escritor de marcos.** Debido a que los archivos CSV pueden ser complejos, hay muchas opciones disponibles; para lista completa le remitimos a la documentación.

Tabla 4-4. Opciones de CSV para DataFrameReader y DataFrameWriter

Valores	Significado	Alcance	
ninguno, bzip2, desinflar, gzip, lz4 o rápido	Utilice este códec de compresión para escribir.	Escribe	
formato de fecha	aaaa-MM-dd o fecha y hora formateador	Use este formato o cualquier formato de Java's Date Formateador de tiempo.	Leer/ escribe
multilínea	cierto, falso	Utilice el modo multilínea. El valor predeterminado es falso (modo de una sola línea).	Leer
inferSchema	cierto, falso	Si es verdadero, Spark determinará los datos de la columna tipos. El valor predeterminado es falso.	Leer
sep	Cualquier personaje	Utilice este carácter para separar valores de columna en una hilera. El delimitador predeterminado es una coma (,).	Leer/ escribe
escapar	Cualquier personaje	Utilice este carácter para escapar de las comillas. El valor predeterminado es \.	Leer/ escribe
encabezamiento	cierto, falso	Indica si la primera línea es un encabezado denotando el nombre de cada columna. El valor predeterminado es falso.	Leer/ escribe

## Euro

Introducido en [Spark 2.4](#) como fuente de datos integrada, el [formato Avro](#) se utiliza, por ejemplo, por ejemplo, por [Apache Kafka](#) para la serialización y deserialización de mensajes. Ofrece muchos beneficios ajustes, incluido el mapeo directo a JSON, velocidad y eficiencia, y enlaces disponibles para muchos lenguajes de programación.

### Lectura de un archivo Avro en un DataFrame

La lectura de un archivo Avro en un DataFrame usando DataFrameReader es consistente en el uso con las otras fuentes de datos que hemos discutido en esta sección:

```
// En escala
val df = chispa.read.format("avro")
    .load("/databricks-conjuntos de datos/learning-spark-v2/flights/summary-data/avro/*")
df.mostrar(falso)

# en pitón
df = (chispa.leer.formato("avro")
      .load("/databricks-conjuntos de datos/learning-spark-v2/flights/summary-data/avro/*"))
df.show(truncar=False)
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	recuento
Estados Unidos	Rumania	1
Estados Unidos	Irlanda	264
Estados Unidos	India	69
Egipto	Estados Unidos	24
Guinea Ecuatorial	Estados Unidos	1

```
| Estados Unidos | Singapur | 25 |
| Estados Unidos | Granada | 54 |
| Costa Rica | Estados Unidos | 477 |
| Senegal | Estados Unidos | 29 |
| Estados Unidos | Islas Marshall | 44 |
+-----+
solo se muestran las 10 primeras filas
```

#### Lectura de un archivo Avro en una tabla Spark SQL

Nuevamente, crear tablas SQL usando una fuente de datos de Avro no es diferente de usar Parquet, JSON o CSV:

```
-- En SQL
CREAR O REEMPLAZAR VISTA TEMPORAL episodio_tbl
    UTILIZANDO avro
    OPCIONES (
        ruta "/databricks-datasets/learning-spark-v2/flights/summary-data/avro/*"
    )
```

Una vez que haya creado una tabla, puede leer datos en un DataFrame usando SQL:

```
// En escala
chispa.sql("SELECCIONE * DE episodio_tbl").show(falso)

# en pitón
chispa.sql("SELECCIONAR * DE episodio_tbl").show(truncar=False)

+-----+
| DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|recuento|
+-----+
| Estados Unidos | Rumanía | 1 |
| Estados Unidos | Irlanda | 264 |
| Estados Unidos | India | 69 |
| Egipto | Estados Unidos | 24 |
| Guinea Ecuatorial|Estados Unidos | 1 |
| Estados Unidos | Singapur | 25 |
| Estados Unidos | Granada | 54 |
| Costa Rica | Estados Unidos | 477 |
| Senegal | Estados Unidos | 29 |
| Estados Unidos | Islas Marshall | 44 |
+-----+
solo se muestran las 10 primeras filas
```

#### Escribir tramas de datos en archivos Avro

Escribir un DataFrame como un archivo Avro es simple. Como de costumbre, especifique los Datos apropiados Métodos y argumentos de FrameWriter , y proporcione la ubicación para guardar los archivos Avro para:

```
// En escala
df.escribir
    .formato ("avro")
    .modo("sobrescribir")
    .guardar("/tmp/datos/avro/df_avro")

# en pitón
(df.escribir
    .formato ("avro")
    .modo("sobrescribir")
    .guardar("/tmp/datos/avro/df_avro"))
```

Esto genera una carpeta en la ubicación especificada, poblada con un montón de comódulos archivos comprimidos y compactos:

```
-rw-r--r-- 1 rueda de julio 0 17 de mayo 11:54 _ÉXITO
-rw-r--r-- 1 rueda de julio 526 17 de mayo 11:54 part-00000-fdf70f4-<...>-c000.avro
```

#### Opciones de fuente de datos de Avro

**La Tabla 4-5** describe las opciones comunes para DataFrameReader y DataFrameWriter. UN

La lista completa de opciones se encuentra en la [documentación](#).

Tabla 4-5. Opciones de Avro para DataFrameReader y DataFrameWriter

Nombre de la propiedad	Defecto	Significado	Alcance
avroSchema	Ninguna	Esquema Avro opcional proporcionado por un usuario en formato JSON. Los datos el tipo y el nombre de los campos de registro deben coincidir con los datos de entrada de Avro o Datos de Catalyst (tipo de datos interno de Spark); de lo contrario, la lectura/escritura la acción fallará.	Leer/ escribe
recordName	nivel superior Registro	Nombre de registro de nivel superior en el resultado de escritura, que se requiere en Avro <small>Especificaciones</small>	Escribe
recordNamespace	" "	Registre el espacio de nombres en el resultado de escritura.	Escribe
ignorarExtensión	verdadero	Si esta opción está habilitada, todos los archivos (con y sin el .avro extensión) están cargados. De lo contrario, los archivos sin la extensión .avro se ignorado	Leer
compresión	rápido	Le permite especificar el códec de compresión para usar en la escritura. Los códecs admitidos actualmente son sin comprimir, ágiles, deflate, bzip2 y xz. Si esta opción no está configurada, el valor en spark.sql.avro.compression Se tiene en cuenta sion.codec .	Escribe

#### ORCO

Como formato de archivo en columnas optimizado adicional, Spark 2.x admite un formato [vectorizado](#)

**Lector ORC.** Dos configuraciones de Spark dictan qué implementación ORC usar.

Cuando spark.sql.orc.impl se establece en native y spark.sql.orc.enableVectorize

dReader se establece en verdadero, Spark usa el lector ORC vectorizado. Un [lector vectorizado](#)

lee bloques de filas (a menudo 1024 por bloque) en lugar de una fila a la vez, simplificando operaciones y reducir el uso de la CPU para operaciones intensivas como escaneos, filtros, agregados gaciones y uniones.

Para tablas Hive ORC SerDe (serialización y deserialización) creadas con SQL comando UTILIZANDO LAS OPCIONES DE HIVE (formato de archivo 'ORC'), se usa el lector vectorizado cuando el parámetro de configuración Spark spark.sql.hive.convertMetastoreOrc es establecido en verdadero.

### Leer un archivo ORC en un DataFrame

Para leer en un DataFrame usando el lector vectorizado ORC, puede usar el normal Métodos y opciones de DataFrameReader :

```
// En el archivo
Scala val = "/databricks-datasets/learning-spark-v2/flights/summary-data/orc/*"
val df = chispa.leer.formato("orc").cargar(archivo)
df.mostrar(10, falso)

# en pitón
file = "/databricks-datasets/learning-spark-v2/flights/summary-data/orc/*"
df = spark.read.format("orc").option("ruta", archivo).load()
df.mostrar(10, Falso)

+-----+-----+
|DEST_COUNTRY_NAME|ORIGIN_COUNTRY_NAME|recuento|
+-----+-----+
|Estados Unidos    |Rumanía |1 |
|Estados Unidos    |Irlanda |264 |
|Estados Unidos    |India   |69 |
|Egipto           |Estados Unidos |24 |
|Guinea Ecuatorial|Estados Unidos |1 |
|Estados Unidos    |Singapur |25 |
|Estados Unidos    |Granada |54 |
| Costa Rica      |Estados Unidos |477 |
|Senegal          |Estados Unidos |29 |
|Estados Unidos    |Islas Marshall |44 |
+-----+
solo se muestran las 10 primeras filas
```

### Lectura de un archivo ORC en una tabla Spark SQL

No hay diferencia con Parquet, JSON, CSV o Avro al crear una vista SQL utilizando una fuente de datos ORC:

```
-- En SQL
CREAR O REEMPLAZAR VISTA TEMPORAL us_delay_flights_tbl
    USAR orc
    OPCIONES (
        ruta "/databricks-datasets/learning-spark-v2/flights/summary-data/orc/*"
    )
```

Una vez que se crea una tabla, puede leer datos en un DataFrame usando SQL como de costumbre:

```
// En Scala/Python
chispa.sql("SELECCIONE * DE us_delay_flights_tbl").show()
```

DEST_COUNTRY_NAME	ORIGIN_COUNTRY_NAME	recuento
Estados Unidos	Rumanía	1
Estados Unidos	Irlanda	264
Estados Unidos	India	69
Egipto	Estados Unidos	24
Guinea Ecuatorial	Estados Unidos	1
Estados Unidos	Singapur	25
Estados Unidos	Granada	54
Costa Rica	Estados Unidos	477
Senegal	Estados Unidos	29
Estados Unidos	Islas Marshall	44

solamente se muestran las 10 primeras filas

#### Escribir tramas de datos en archivos ORC

Escribir un DataFrame transformado después de leerlo es igualmente simple usando el Método de DataFrameWriter :

```
// En scala
df.write.format("orc")
  .mode("sobrescribir")
  .option("compresión", "rápido")
  .save("/tmp/data/orc/df_orc")

# en pitón
(df.write.format("orc")
  .mode("sobrescribir")
  .option("compresión", "rápido")
  .save("/tmp/datos/orc/vuelos_orc"))
```

El resultado será una carpeta en la ubicación especificada que contiene algo de ORC comprimido archivos:

```
-rw-r--r-- 1 rueda de julio -rw-r--r-- 0 16 mayo 17:23 _ÉXITO
1 rueda de julio 547 16 de mayo 17:23 part-00000-<...>-c000.snappy.orc
```

#### Imágenes

En Spark 2.4, la comunidad introdujo una nueva fuente de datos, [archivos de imagen](#), apoyar marcos de aprendizaje profundo y aprendizaje automático como TensorFlow y PyTorch.

Para aplicaciones de aprendizaje automático basadas en visión artificial, carga y procesamiento conjuntos de datos de imagen es importante.

## Leer un archivo de imagen en un DataFrame

Al igual que con todos los formatos de archivo anteriores, puede usar los métodos DataFrameReader y opciones para leer en un archivo de imagen como se muestra aquí:

```
// En escala
import org.apache.spark.ml.source.image

val imageDir = "/databricks-datasets/learning-spark-v2/cctvVideos/train_images/"
val imágenesDF = spark.read.format("imagen").load(imageDir)

imágenesDF.printSchema

imagesDF.select("imagen.altura", "imagen.ancho", "imagen.nCanales", "imagen.modo",
    "etiqueta").show(5, falso)
# en pitón
de la imagen de importación de pyspark.ml

image_dir = "/databricks-conjuntos de datos/learning-spark-v2/cctvVideos/train_images/"
images_df = chispa.read.format("imagen").load(image_dir)
imágenes_df.printSchema()

raíz
|- imagen: estructura (anulable = verdadero)
    |- origen: cadena (anulable = verdadero)
    |- altura: entero (anulable = verdadero)
    |- ancho: entero (anulable = verdadero)
    |- nChannels: entero (anulable = verdadero)
    |- modo: entero (anulable = verdadero)
    |- datos: binario (anulable = verdadero)
    ||||| |-- etiqueta: entero (anulable = verdadero)

images_df.select("imagen.altura", "imagen.ancho", "imagen.nCanales", "imagen.modo",
    "etiqueta").show(5, truncado=False)

+-----+-----+-----+-----+
|alto|ancho|nCanales|modo|etiqueta|
+-----+-----+-----+-----+
|288|384|3|288| |16|0|16| | | |
|384|3|288|384|3| |1|16|0| | |
|288|384|3|288| |16|0|16| |
|384|3| |0| | | |
+-----+-----+-----+-----+
solo se muestran las 5 filas superiores
```

## Archivos

**binarios** Spark 3.0 agrega soporte para **archivos binarios como fuente de datos**. DataFrameReader convierte cada archivo binario en una única fila (registro) de DataFrame que contiene el contenido sin procesar y los metadatos del archivo. La fuente de datos del archivo binario produce un DataFrame con las siguientes columnas:

- ruta: tipo de cadena
- tiempo de modificación: tipo de marca de tiempo
- longitud: tipo largo
- contenido: tipo binario

### Lectura de un archivo binario en un

**DataFrame** Para leer archivos binarios, especifique el formato de la fuente de datos como un archivo binario. Puede cargar archivos con rutas que coincidan con un patrón global determinado y conservar el comportamiento de la detección de particiones con la opción de fuente de datos pathGlobFilter. Por ejemplo, el siguiente código lee todos los archivos JPG del directorio de entrada con cualquier directorio particionado:

```
// En Scala val
path = "/databricks-datasets/learning-spark-v2/cctvVideos/train_images/" val binaryFilesDF =
spark.read.format("binaryFile")
.option("pathGlobFilter", "*.jpg") .load(ruta)
binaryFilesDF.show(5)

# En Python
ruta = "/databricks-datasets/learning-spark-v2/cctvVideos/train_images/" binary_files_df =
(spark.read.format("binaryFile") .option("pathGlobFilter", "*.jpg") .load(ruta)) binary_files_df.show(5)
```

camino	modificaciónTiempo	longitud	contenido	etiqueta
archivo:/Usuarios/jules...[2020-02-12 12:04:24] 55037	[FF D8 FF E0 00 1...]	archivo:/Usuarios/jules...[2020-02-12 12:04:24] 54634	[FF D8 FF E0 00 1...]	archivo:/Usuarios/jules...[2020-02-12 12:04:24] 54624

solo se muestran las 5 filas superiores

Para ignorar el descubrimiento de datos de partición en un directorio, puede configurar recursiveFile Búsqueda de "verdadero":

```
// En Scala val
binaryFilesDF = spark.read.format("binaryFile")
  .option("pathGlobFilter",
  "*.jpg") .option("recursiveFileLookup",
  "true") .load(ruta) binaryFilesDF.show(5)
```

```
# En Python
binary_files_df = (spark.read.format("binaryFile")
  .option("pathGlobFilter",
  "*.jpg") .option("recursiveFileLookup",
  "true") .load(ruta)) binary_files_df.show(5)
```

camino	modificación	Tiempo	longitud	contenido
archivo:/Usuarios/jules... 2020-02-12 12:04:24  55037 [FF D8 FF E0 00 1...]	archivo:/Usuarios/jules... 2020-02-12 12:04:24  54634 [FF D8 FF E0 00 1...]	archivo:/Usuarios/jules... 2020-02-12 12:04:24  54624 [FF D8 FF E0 00 1...]	archivo:/Usuarios/jules... 2020-02-12 12:04:24  54505 [FF D8 FF E0 00 1...]	archivo:/Usuarios/jules... 2020-02-12 12:04:24  54475 [FF D8 FF E0 00 1...]
1...]				
solo se muestran las 5 filas superiores				

Tenga en cuenta que la columna de etiqueta está ausente cuando la opción recursiveFileLookup se establece en "true".

Actualmente, la fuente de datos del archivo binario no admite la escritura de un DataFrame en el formato de archivo original.

En esta sección, obtuvo un recorrido por cómo leer datos en un DataFrame desde una variedad de formatos de archivo admitidos. También le mostramos cómo crear vistas y tablas temporales a partir de las fuentes de datos integradas existentes. Ya sea que esté utilizando la API de DataFrame o SQL, las consultas producen resultados idénticos. Puede examinar algunas de estas consultas en el cuaderno disponible en el repositorio de [GitHub](#) para este libro

## Resumen

En resumen, este capítulo exploró la interoperabilidad entre la API de DataFrame y Spark SQL. En particular, obtuvo una idea de cómo usar Spark SQL para:

- Cree tablas administradas y no administradas usando Spark SQL y DataFrame API.
- Leer y escribir en varias fuentes de datos integradas y formatos de archivo.
- Emplear la interfaz programática spark.sql para emitir consultas SQL en la estructura de datos almacenados como tablas o vistas de Spark SQL.
- Examine el catálogo de Spark para inspeccionar metadatos asociados con tablas y vistas.

- Utilice las API DataFrameWriter y DataFrameReader .

A través de los fragmentos de código del capítulo y los cuadernos disponibles en el repositorio de [GitHub del libro](#), tienes una idea de cómo usar DataFrames y Spark SQL. Continuando con esta línea, el próximo capítulo explora más a fondo cómo Spark interactúa con las fuentes de datos externas que se muestran en [la Figura 4-1](#). Verá algunos ejemplos más detallados de transformaciones y la interoperabilidad entre la API de DataFrame y Spark SQL.

## CAPÍTULO 5

# Spark SQL y marcos de datos: Interactuar con fuentes de datos externas

En el capítulo anterior, exploramos la interacción con las fuentes de datos integradas en Spark. También echamos un vistazo más de cerca a la API de DataFrame y su interoperabilidad con Spark SQL. En este capítulo, nos centraremos en cómo Spark SQL interactúa con componentes externos. Específicamente, discutimos cómo Spark SQL le permite:

- Utilice funciones definidas por el usuario para Apache Hive y Apache Spark.
- Conéctese con fuentes de datos externas como bases de datos JDBC y SQL, PostgreSQL, MySQL, Tableau, Azure Cosmos DB y MS SQL Server.
- Trabajar con tipos simples y complejos, funciones de orden superior y relaciones comunes.

También veremos algunas opciones diferentes para consultar Spark usando Spark SQL, como el shell Spark SQL, Beeline y Tableau.

## Spark SQL y Apache Hive

Spark SQL es un componente fundamental de Apache Spark que integra el procesamiento relacional con la API de programación funcional de Spark. Su génesis fue en un trabajo [anterior sobre Shark](#). Shark se creó originalmente sobre la base de código de Hive sobre Apache Spark<sup>1</sup> y se convirtió en uno de los primeros motores de consulta SQL interactivos en los sistemas Hadoop. Demostró que era posible tener lo [mejor de ambos mundos](#); tan rápido como un almacén de datos empresarial y escalable tan bien como Hive/MapReduce.

---

<sup>1</sup> El motor Spark SQL actual ya no usa el código de Hive en su implementación.

Spark SQL permite a los programadores de Spark aprovechar los beneficios de un rendimiento más rápido y una programación relacional (p. ej., consultas declarativas y almacenamiento optimizado), así como llamar a bibliotecas analíticas complejas (p. ej., aprendizaje automático). Como se discutió en el capítulo anterior, a partir de Apache Spark 2.x, SparkSession proporciona un único punto de entrada unificado para manipular datos en Spark.

### Funciones definidas por el usuario

Si bien Apache Spark tiene una gran cantidad de funciones integradas, la flexibilidad de Spark permite que los ingenieros de datos y los científicos de datos también definan sus propias funciones. Estas se conocen como funciones definidas por el usuario (UDF).

#### Spark SQL UDF

El beneficio de crear sus propios PySpark o Scala UDF es que usted (y otros) podrán utilizarlos dentro de Spark SQL. Por ejemplo, un científico de datos puede envolver un modelo de ML dentro de una UDF para que un analista de datos pueda consultar sus predicciones en Spark SQL sin necesariamente comprender los aspectos internos del modelo.

Este es un ejemplo simplificado de cómo crear una UDF de Spark SQL. Tenga en cuenta que las UDF funcionan por sesión y no se conservarán en el metastore subyacente:

```
// En escala
// Crear función al cubo val
cubed = (s: Long) => {
    s * s * s
}

// Registrar UDF
spark.udf.register("cubed", cubed)

// Crear vista temporal
spark.range(1, 9).createOrReplaceTempView("udf_test")

# En Python
desde pyspark.sql.types import LongType

# Crear función cubicada
def cubed(s): return s
    s * s

# Registrar UDF
spark.udf.register("cubed", cubed, LongType())

# Generar vista temporal
spark.range(1, 9).createOrReplaceTempView("udf_test")
```

Ahora puede usar Spark SQL para ejecutar cualquiera de estas funciones cubed() :

```
// En Scala/Python //
Consulta el UDF en cubos
spark.sql("SELECT id, cubed(id) AS id_cubed FROM udf_test").show()

+---+-----+
| id|id_cubed|
+---+-----+
| 1|      1|
| 2|      8|
| 3|     27|
| 4|     64|
| 5|    125|
| 6|    216|
| 7|   343|
| 8|   512|
+---+-----+
```

#### Orden de evaluación y comprobación de nulos en

**Spark SQL** Spark SQL (esto incluye SQL, la API de DataFrame y la API de conjunto de datos) no garantiza el orden de evaluación de las subexpresiones. Por ejemplo, la siguiente consulta no garantiza que la cláusula s is NOT NULL se ejecute antes que la cláusula strlen(s) > 1 :

```
spark.sql("SELECCIONE s DE test1 DONDE s NO ES NULO Y strlen(s) > 1")
```

Por lo tanto, para realizar una comprobación nula adecuada , se recomienda hacer lo siguiente:

1. Haga que la propia UDF reconozca los valores nulos y realice una verificación de valores nulos dentro de la UDF.
2. Utilice las expresiones IF o CASE WHEN para realizar la comprobación de valores nulos e invocar la UDF en un rama condicional.

#### Aceleración y distribución de PySpark UDF con Pandas UDF Uno de

los problemas predominantes anteriores con el uso de PySpark UDF era que tenían un rendimiento más lento que Scala UDF. Esto se debió a que las UDF de PySpark requerían el movimiento de datos entre JVM y Python, lo cual era bastante costoso. Para resolver este problema, [las UDF de Pandas](#) (también conocidos como UDF vectorizados) se introdujeron como parte de Apache Spark 2.3. Una UDF de Pandas usa Apache Arrow para transferir datos y Pandas para trabajar con los datos. Defina una UDF de Pandas usando la palabra clave pandas\_udf como decorador, o para envolver la función en sí. Una vez que los datos están en [formato Apache Arrow](#) , ya no es necesario serializar/decapar los datos, ya que ya están en un formato consumible por el proceso de Python. En lugar de operar en entradas individuales fila por fila, está operando en una Serie Pandas o DataFrame (es decir, ejecución vectorizada).

Desde Apache Spark 3.0 con Python 3.6 y superior, las UDF de Pandas se dividieron en dos categorías de API: UDF de Pandas y API de funciones de Pandas.

#### Pandas UDF

Con Apache Spark 3.0, las UDF de Pandas deducen el tipo de UDF de Pandas a partir de las sugerencias de tipo de Python en las UDF de Pandas, como `pandas.Series`, `pandas.DataFrame`, `Tuple` e `Iterator`. Anteriormente, necesitaba definir y especificar manualmente cada tipo de UDF de Pandas. Actualmente, los casos admitidos de sugerencias de tipo Python en Pandas UDF son Serie a serie, Iterador de serie a Iterador de serie, Iterador de serie múltiple a Iterador de serie y Serie a escalar (un valor único).

#### API de funciones de pandas

Las API de funciones de Pandas le permiten aplicar directamente una función de Python local a un marco de datos de PySpark donde tanto la entrada como la salida son instancias de Pandas. Para Spark 3.0, las API de funciones de Pandas compatibles son mapa agrupado, mapa, mapa coagrupado.

Para obtener más información, consulte “[UDF de Pandas rediseñadas con sugerencias de tipo de Python](#)” en la página 354 en el [Capítulo 12](#).

El siguiente es un ejemplo de una UDF de Pandas escalar para Spark 3.0:<sup>2</sup>

```
# en pitón
# Importar pandas
import pandas como pd

# Importe varias funciones SQL de pyspark, incluidas pandas_udf de
pyspark.sql.functions import col, pandas_udf de pyspark.sql.types import LongType

# Declarar la función cubicada def
cubed(a: pd.Series) -> pd.Series:
    devolver un * un * un

# Crea la UDF de pandas para la función en cubos cubed_udf =
pandas_udf(cubed, returnType=LongType())
```

El fragmento de código anterior declara una función llamada `cubed()` que realiza una operación cubicada. Esta es una función normal de Pandas con la llamada adicional `cubed_udf = pandas_udf(cubed, returnType=LongType())` para crear nuestra UDF de Pandas.

---

<sup>2</sup> Tenga en cuenta que existen ligeras diferencias cuando se trabaja con Pandas UDF entre Spark 2.3, 2.4, y 3.0.

Comencemos con una Serie Pandas simple (como se define para x) y luego apliquemos la función local cubed() para el cálculo al cubo:

```
# Crea una Serie de Pandas
x = pd.Series([1, 2, 3])

# La función para pandas_udf ejecutada con datos locales de Pandas
print(cubed(x))
```

La salida es la siguiente:

```
0      1
1      8
2     27
tipo: int64
```

Ahora cambiemos a un Spark DataFrame. Podemos ejecutar esta función como una UDF vectorizada de Spark de la siguiente manera:

```
# Crear un Spark DataFrame, 'spark' es un SparkSession existente df =
spark.range(1, 4)

# Ejecutar la función como un UDF vectorizado de Spark
df.select("id", cubed_udf(col("id"))).show()
```

Aquí está la salida:

```
+---+-----+
| id|cubed(id)|
+---+-----+
| 1|          1|
2| 3|          8|
              27|
+---+-----+
```

A diferencia de una función local, el uso de una UDF vectorizada dará como resultado la ejecución de trabajos de Spark; la función local anterior es una función de Pandas ejecutada solo en el controlador Spark. Esto se vuelve más evidente al ver la interfaz de usuario de Spark para una de las etapas de esta función pandas\_udf ([Figura 5-1](#)).



Para profundizar más en las UDF de Pandas, consulte la [documentación de las funciones definidas por el usuario de Pandas](#).

## Details for Stage 28 (Attempt 0)

**Total Time Across All Tasks:** 73 ms

**Locality Level Summary:** Process local: 1

**Associated Job Ids:** 28

▼DAG Visualization

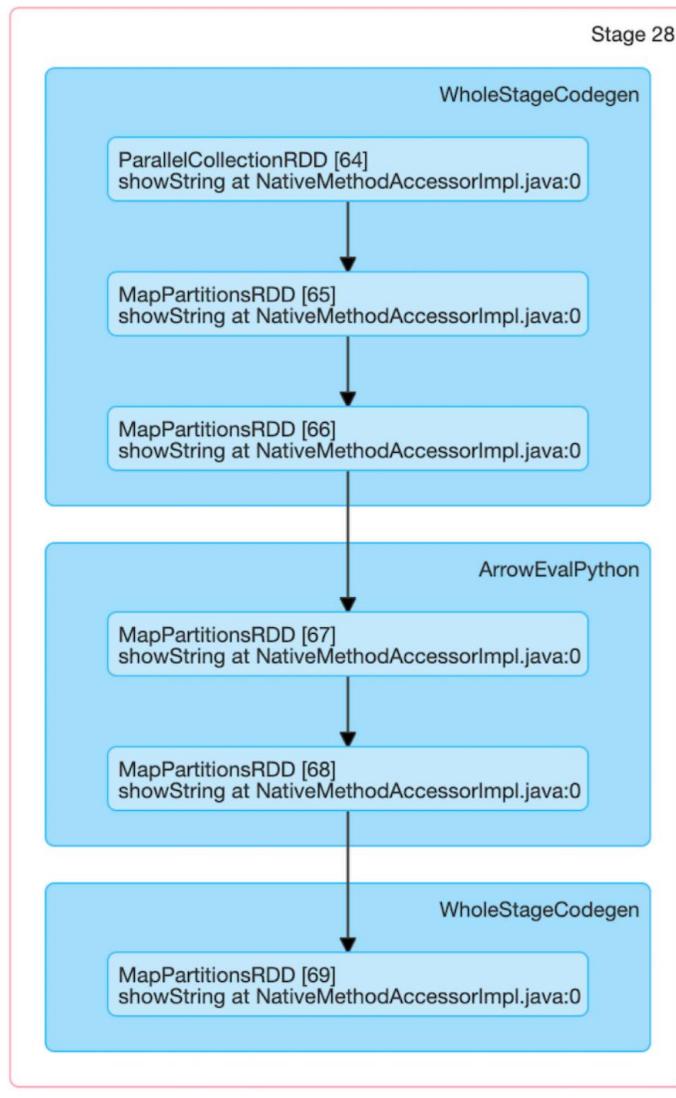


Figura 5-1. Etapas de Spark UI para ejecutar un UDF de Pandas en un Spark DataFrame

Al igual que muchos trabajos de Spark, el trabajo comienza con `parallelize()` para enviar datos locales (lotes binarios de Arrow) a los ejecutores y llama a `mapPartitions()` para convertir los lotes binarios de Arrow al formato de datos interno de Spark, que se puede distribuir a los trabajadores de Spark. Hay una serie de pasos de `WholeStageCodegen`, que representan un avance fundamental en el rendimiento (gracias a la [generación de código de etapa completa de Project Tungsten](#), lo que mejora significativamente la eficiencia y el rendimiento de la CPU). Pero es el paso `ArrowE valPython` el que identifica que (en este caso) se está ejecutando una UDF de Pandas.

## Consultas con Spark SQL Shell, Beeline y Tableau

Hay varios mecanismos para consultar Apache Spark, incluido el shell Spark SQL, la utilidad Beeline CLI y herramientas de generación de informes como Tableau y Power BI.

En esta sección, incluimos instrucciones para Tableau; para Power BI, consulte la [documentación](#).

### Usando el shell SQL de Spark

Una herramienta conveniente para ejecutar consultas Spark SQL es la CLI `spark-sql`. Si bien esta utilidad se comunica con el servicio Metastore de Hive en modo local, no se comunica con el [servidor Thrift JDBC/ODBC](#). (también conocido como Spark Thrif Server o STS). El STS permite a los clientes JDBC/ODBC ejecutar consultas SQL sobre los protocolos JDBC y ODBC en Apache Spark.

Para iniciar Spark SQL CLI, ejecute el siguiente comando en la carpeta `$SPARK_HOME`:

```
./bin/spark-sql
```

Una vez que haya iniciado el shell, puede usarlo para realizar consultas Spark SQL de forma interactiva. Echemos un vistazo a algunos ejemplos.

#### crear una tabla

Para crear una nueva tabla Spark SQL permanente, ejecute la siguiente instrucción:

```
spark-sql> CREATE TABLE personas (nombre STRING, edad int);
```

Su salida debe ser similar a esta, teniendo en cuenta la creación de la tabla `personas` de Spark SQL, así como la ubicación de su archivo (`/usuario/colmena/almacén/personas`):

20/01/11 22:42:16 WARN HiveMetaStore: Ubicación: archivo:/usuario/colmena/almacén/personas especificadas para tabla no externa :personas Tiempo empleado: 0,63 segundos

## Insertar datos en la tabla

Puede insertar datos en una tabla Spark SQL ejecutando una declaración similar a:

```
INSERTAR EN personas SELECCIONAR nombre, edad DESDE ...
```

Como no depende de la carga de datos de una tabla o archivo preexistente, puede insertar datos en la tabla mediante declaraciones **INSERT...VALUES**. Estas tres declaraciones insertan tres personas (sus nombres y edades, si se conocen) en la tabla de personas :

```
spark-sql> INSERTAR EN VALORES de personas ("Michael", NULL); Tiempo  
empleado: 1,696 segundos spark-sql> INSERT INTO people VALUES ("Andy",  
30); Tiempo empleado: 0,744 segundos spark-sql> INSERTAR EN VALORES de  
personas ("Samantha", 19); Tiempo empleado: 0,637 segundos spark-sql>
```

## Ejecutar una consulta

**Spark SQL** Ahora que tiene datos en su tabla, puede ejecutar consultas Spark SQL contra ellos. Comencemos por ver qué tablas existen en nuestra metatienda:

```
spark-sql> MOSTRAR TABLAS;  
personas predeterminadas falseTiempo  
empleado: 0,016 segundos, Obtuvo 1 fila(s)
```

A continuación, averigüemos cuántas personas en nuestra tabla son menores de 20 años:

```
spark-sql> SELECT * FROM personas DONDE edad < 20; Samantha  
19 Tiempo empleado: 0,593 segundos, Obtenido 1 fila(s)
```

Además, veamos quiénes son los individuos que no especificaron su edad:

```
spark-sql> SELECCIONE el nombre DE las personas DONDE la edad ES  
NULA; Michael Tiempo empleado: 0,272 segundos, Obtuvo 1 fila(s)
```

## Trabajar con Beeline

Si ha trabajado con Apache Hive, es posible que esté familiarizado con la herramienta de línea de comandos **Beeline**, una utilidad común para ejecutar consultas HiveQL contra HiveServer2.

Beeline es un cliente JDBC basado en **SQLLine CLI**. Puede usar esta misma utilidad para ejecutar consultas Spark SQL en el servidor Spark Thrift. Tenga en cuenta que el servidor Thrift JDBC/ODBC actualmente implementado corresponde a HiveServer2 en Hive 1.2.1. Puede probar el servidor JDBC con el siguiente script Beeline que viene con Spark o Hive 1.2.1.

### Inicie el servidor Thrift

Para iniciar el servidor Spark Thrift JDBC/ODBC, ejecute el siguiente comando desde la carpeta \$SPARK\_HOME :

```
./sbin/start-thriftserver.sh
```



Si aún no ha iniciado su controlador y trabajador de Spark, ejecute el siguiente comando antes de start-thriftserver.sh:

```
./sbin/start-all.sh
```

### Conéctese al servidor Thrift a través de Beeline

Para probar el servidor Thrift JDBC/ODBC usando Beeline, ejecute el siguiente comando:

```
./bin/línea recta
```

Luego configure Beeline para conectarse al servidor Thrift local:

```
!conectar jdbc:colmena2://hostlocal:10000
```



De forma predeterminada, Beeline está en modo no seguro. Por lo tanto, el nombre de usuario es su inicio de sesión (por ejemplo, usuario@learningspark.org) y la contraseña está en blanco.

### Ejecutar una consulta de Spark SQL con

**Beeline** Desde aquí, puede ejecutar una consulta de Spark SQL de forma similar a como ejecutaría una consulta de Hive con Beeline. Aquí hay algunas consultas de muestra y su salida:

```
0: jdbc:hive2://localhost:10000> MOSTRAR tablas;
```

base de datos	nombretabla	esTemporal
predeterminado	gente	falso

1 fila seleccionada (0,417 segundos)

```
0: jdbc:hive2://localhost:10000> SELECCIONAR * DE personas;
```

nombre	edad
Samanta	19
Andy	30
Miguel	NULO

3 filas seleccionadas (1.512 segundos)

0: jdbc:colmena2://localhost:10000>

#### **Detener el servidor Thrift**

Una vez que haya terminado, puede detener el servidor Thrift con el siguiente comando:

`./sbin/stop-thriftserver.sh`

#### **Trabajar con Tableau** Al igual

que ejecutar consultas a través de Beeline o Spark SQL CLI, puede conectar su herramienta de BI favorita a Spark SQL a través del servidor Thrift JDBC/ODBC. En esta sección, le mostraremos cómo conectar Tableau Desktop (versión 2019.2) a su instancia local de Apache Spark.



Deberá tener el **Spark ODBC de Tableau** versión del controlador 1.2.0 o superior ya instalada. Si ha instalado (o actualizado) Tableau 2018.1 o superior, este controlador ya debería estar preinstalado.

#### **Inicie el servidor Thrift**

Para iniciar el servidor Spark Thrift JDBC/ODBC, ejecute el siguiente comando desde la carpeta `$SPARK_HOME` :

`./sbin/start-thriftserver.sh`



Si aún no ha iniciado su controlador y trabajador de Spark, ejecute el siguiente comando antes de `start-thriftserver.sh`:

`./sbin/start-all.sh`

## Iniciar Tableau

Si inicia Tableau por primera vez, verá un cuadro de diálogo Conectar que le permitirá conectarse a una gran cantidad de fuentes de datos. De forma predeterminada, la opción Spark SQL no se incluirá en el menú "A un servidor" de la izquierda (consulte la [Figura 5-2](#)).

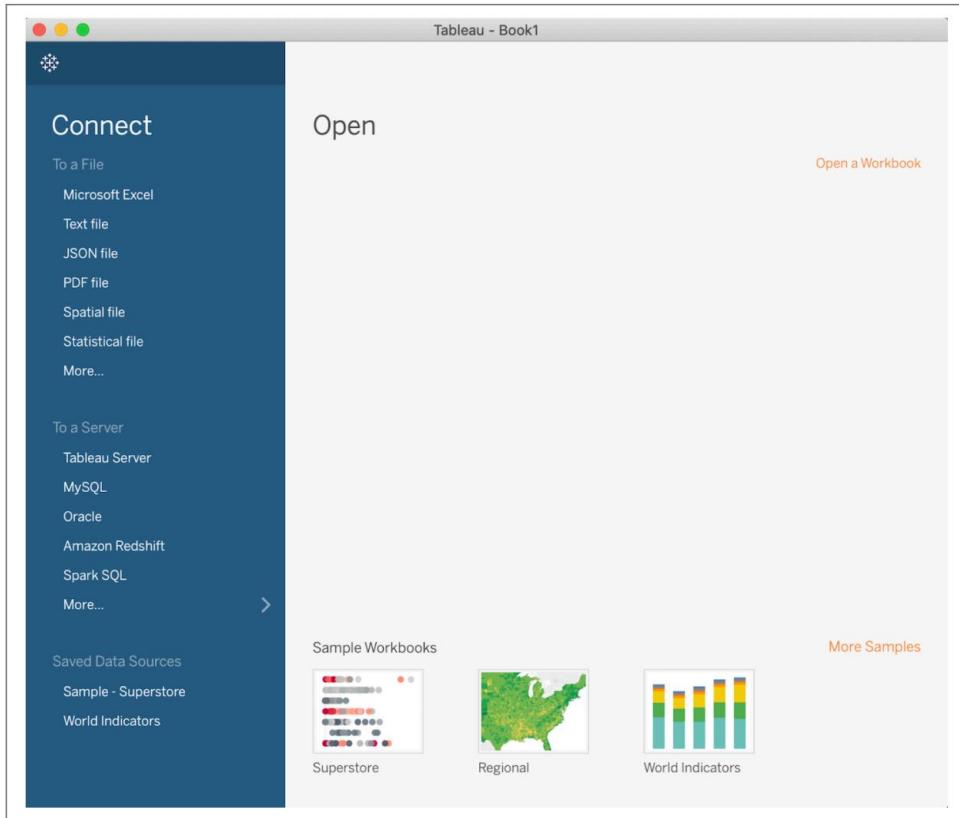


Figura 5-2. Cuadro de diálogo de conexión de Tableau

Para acceder a la opción Spark SQL, haga clic en Más... en la parte inferior de esa lista y luego elija Spark SQL de la lista que aparece en el panel principal, como se muestra en la Figura 5-3.

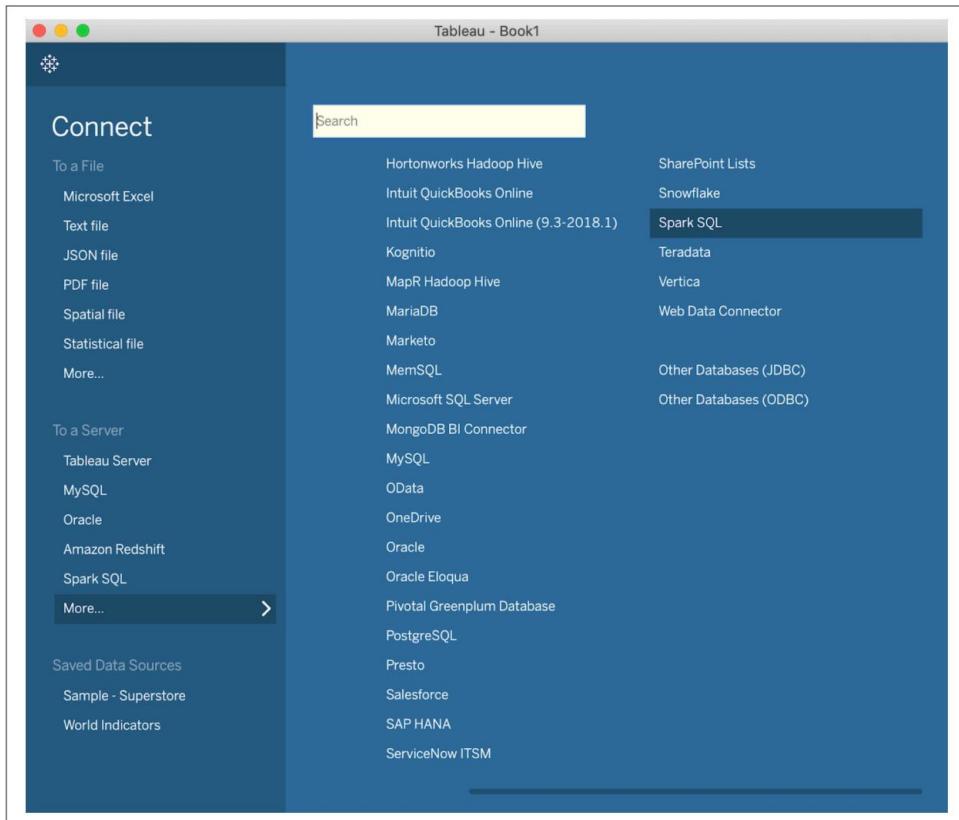


Figura 5-3. Elija Más... > Spark SQL para conectarse a Spark SQL

Esto abrirá el cuadro de diálogo Spark SQL ([Figura 5-4](#)). Mientras se conecta a una instancia local de Apache Spark, puede usar el modo de autenticación de nombre de usuario no seguro con los siguientes parámetros:

- Servidor: servidor local
- Puerto: 10000 (predeterminado)
- Tipo: SparkThriftServer (predeterminado)
- Autenticación: Nombre de usuario
- Nombre de usuario: su inicio de sesión, por ejemplo,  
usuario@learningspark.org • Requerir SSL: no marcado

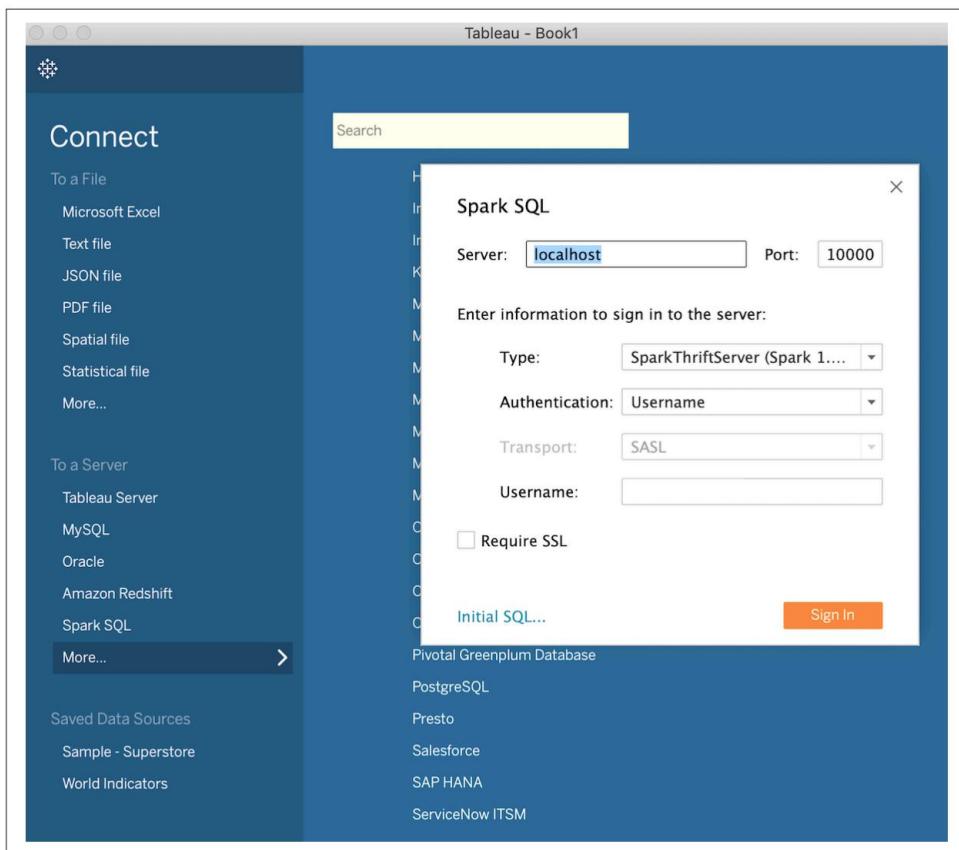


Figura 5-4. El cuadro de diálogo Spark SQL

Una vez que se haya conectado con éxito a la fuente de datos Spark SQL, verá una vista de Conexiones de fuente de datos similar a la [Figura 5-5](#).

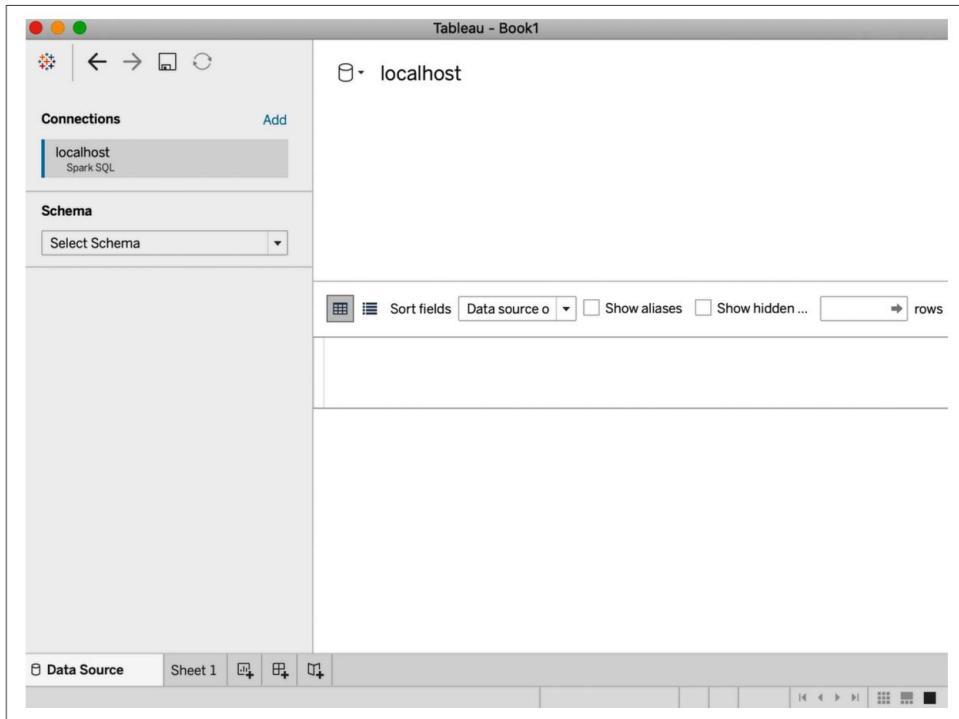


Figura 5-5. Vista Conexiones de fuentes de datos de Tableau, conectada a una instancia local de Spark

En el menú desplegable Seleccionar esquema a la izquierda, elija "predeterminado". Luego ingrese el nombre de la tabla que desea consultar (vea la [Figura 5-6](#)). Tenga en cuenta que puede hacer clic en el ícono de la lupa para obtener una lista completa de las tablas disponibles.

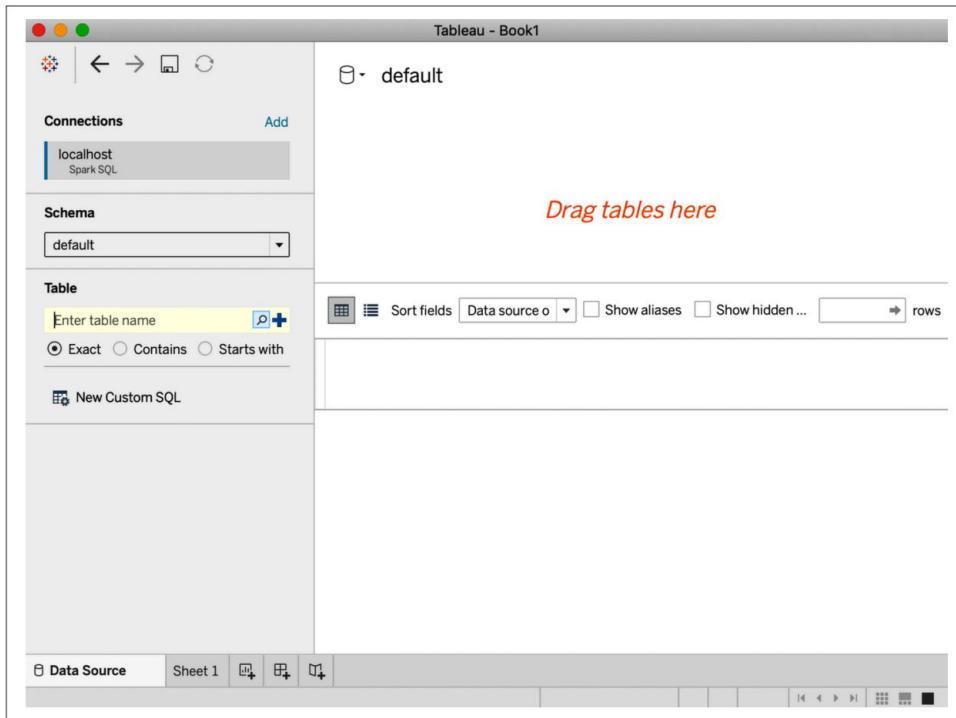


Figura 5-6. Seleccione un esquema y una tabla para consultar



Para obtener más información sobre el uso de Tableau para conectarse a una base de datos Spark SQL, consulte la [documentación de Spark SQL](#) de Tableau y la documentación de Databricks [Tableau](#).

Ingrese personas como el nombre de la tabla, luego arrastre y suelte la tabla desde el lado izquierdo en el cuadro de diálogo principal (en el espacio marcado "Arrastrar tablas aquí"). Debería ver algo como la [Figura 5-7](#).

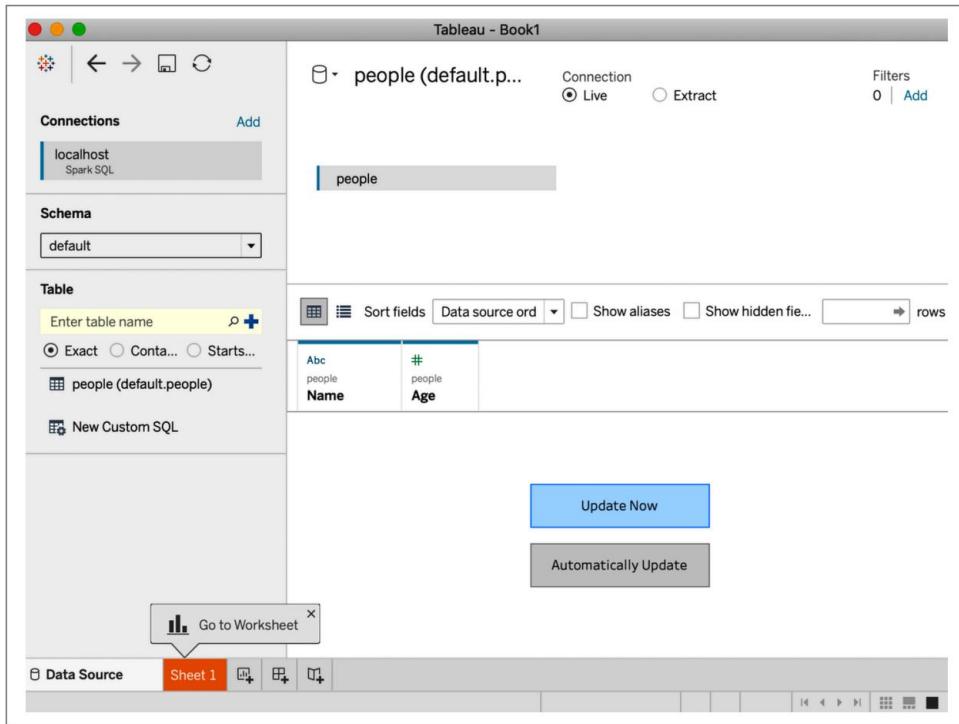


Figura 5-7. Conexión a la tabla de personas en su instancia local de Spark

Haga clic en Actualizar ahora y, bajo las cubiertas, Tableau consultará su fuente de datos Spark SQL ([Figura 5-8](#)).

Ahora puede ejecutar consultas en su fuente de datos de Spark, unir tablas y más, al igual que con cualquier otra fuente de datos de Tableau.

The screenshot shows the Tableau Data Source interface. On the left, there's a sidebar with 'Connections' (localhost Spark SQL selected), 'Schema' (default), and 'Table' (people). The main area displays the 'people' table with three rows: Michael (Age null), Samantha (Age 19), and Andy (Age 30). A tooltip 'Go to Worksheet' points to the 'Sheet 1' tab at the bottom.

Name	Age
Michael	null
Samantha	19
Andy	30

Figura 5-8. Vista de tabla de la hoja de trabajo de Tableau consultando una fuente de datos local de Spark

#### Detener el servidor Thrift

Una vez que haya terminado, puede detener el servidor Thrift con el siguiente comando:

```
./sbin/stop-thriftserver.sh
```

## Fuentes de datos externas

En esta sección, nos centraremos en cómo usar Spark SQL para conectarse a fuentes de datos externas, comenzando con bases de datos JDBC y SQL.

### Bases de datos JDBC y SQL

Spark SQL incluye una API de origen de datos que puede leer datos de otras bases de datos mediante **JDBC**. Simplifica la consulta de estas fuentes de datos, ya que devuelve los resultados como un marco de datos, lo que proporciona todos los beneficios de Spark SQL (incluido el rendimiento y la capacidad de unirse a otras fuentes de datos).

Para comenzar, deberá especificar el controlador JDBC para su fuente de datos JDBC y deberá estar en el classpath de Spark. Desde la carpeta `$SPARK_HOME`, emitirá un comando como el siguiente:

```
./bin/spark-shell --driver-class-path $base de datos.jar --jars $base de datos.jar
```

Utilizando la API de origen de datos, las tablas de la base de datos remota se pueden cargar como un Vista temporal DataFrame o Spark SQL. Los usuarios pueden especificar la conexión JDBC propiedades en las opciones de fuente de datos. [La tabla 5-1](#) contiene algunos de los más comunes. propiedades de conexión (sin distinción entre mayúsculas y minúsculas) que admite Spark.

Tabla 5-1. Propiedades de conexión comunes

Propiedad	Descripción
nombre	
usuario, paso palabra	Normalmente se proporcionan como propiedades de conexión para iniciar sesión en las fuentes de datos.
URL	URL de conexión de JDBC, por ejemplo, jdbc:postgresql://localhost/test?user=fred&pass palabra=secreto.
dbtable	Tabla JDBC para leer o escribir. No puede especificar las opciones dbtable y query al mismo tiempo.
consulta	Consulta que se usará para leer datos de Apache Spark, por ejemplo, SELECT column1, column2, ..., umnN FROM [table subquery]. No puede especificar las opciones de consulta y dbtable al mismo tiempo.
conductor	Nombre de clase del controlador JDBC que se usará para conectarse a la URL especificada.

Para obtener la lista completa de propiedades de conexión, consulte la [documentación de Spark SQL](#).

## La importancia de dividir

Al transferir grandes cantidades de datos entre Spark SQL y un JDBC externo fuente, es importante particionar su fuente de datos. Todos sus datos están pasando una conexión de controlador, que puede saturar y ralentizar significativamente el rendimiento de su extracción, así como potencialmente saturar los recursos de su fuente sistema. Si bien estas propiedades de JDBC son opcionales, para cualquier operación a gran escala es muy recomendable utilizar las propiedades que se muestran en [la Tabla 5-2](#).

Tabla 5-2. Propiedades de conexión de partición

Nombre de la propiedad	Descripción
númParticiones	El número máximo de particiones que se pueden utilizar para el paralelismo en la lectura de tablas y escritura. Esto también determina el número máximo de conexiones JDBC simultáneas.
columna de partición	Al leer una fuente externa, la columna de partición es la columna que se usa para determinar las particiones; nota, la columna de partición debe ser un número, fecha o columna de marca de tiempo.
límite inferior	Establece el valor mínimo de la columna de partición para el paso de la partición.
limitesuperior	Establece el valor máximo de la columna de partición para el paso de partición.

Echemos un vistazo a un [ejemplo](#) . para ayudarle a comprender cómo funcionan estas propiedades.

Supongamos que usamos la siguiente configuración:

- número de particiones: 10
- límite inferior: 1000
- límite superior: 10000

Entonces el paso es igual a 1000 y se crearán 10 particiones. Este es el equivalente a ejecutar estas 10 consultas (una para cada partición):

- SELECCIONE \* DESDE la tabla DONDE particiónColumna ENTRE 1000 y 2000
- SELECCIONE \* DESDE la tabla DONDE particiónColumna ENTRE 2000 y 3000
- ...
- SELECCIONE \* DESDE la tabla DONDE particiónColumna ENTRE 9000 y 10000

Si bien no lo abarcan todo, los siguientes son algunos consejos que se deben tener en cuenta al usar estas propiedades:

- Un buen punto de partida para numPartitions es usar un múltiplo del número de trabajadores de Spark. Por ejemplo, si tiene cuatro nodos trabajadores de Spark, quizás comience con 4 u 8 particiones. Pero también es importante tener en cuenta qué tan bien su sistema fuente puede manejar las solicitudes de lectura. Para los sistemas que tienen ventanas de procesamiento, puede maximizar la cantidad de solicitudes simultáneas al sistema de origen; para los sistemas que carecen de ventanas de procesamiento (por ejemplo, un sistema OLTP que procesa datos continuamente), debe reducir la cantidad de solicitudes simultáneas para evitar la saturación del sistema de origen.
- Inicialmente, calcule el límite inferior y el límite superior en función de los valores reales mínimo y máximo de la columna de partición . Por ejemplo, si elige {numPartitions:10, lowerBound: 1000, upperBound: 10000}, pero todos los valores están entre 2000 y 4000, entonces solo 2 de las 10 consultas (una para cada partición) realizarán todas las trabajo. En este escenario, una mejor configuración sería {numPartitions:10, lowerBound: 2000, upperBound: 4000}.
- Elija una columna de partición que se pueda distribuir uniformemente para evitar la desviación de datos. Por ejemplo, si la mayor parte de su columna de partición tiene el valor 2500, con {numPartitions: 10, lowerBound: 1000, upperBound: 10000} la mayor parte del trabajo lo realizará la tarea que solicita los valores entre 2000 y 3000. En su lugar, elija una columna de partición diferente o, si es posible, genere una nueva (quizás un hash de varias columnas) para distribuir sus particiones de manera más uniforme.

## PostgreSQL

Para conectarse a una base de datos PostgreSQL, cree o descargue el jar JDBC de [Maven](#) y agréguelo a su classpath. Luego inicie un Spark shell (spark-shell o pyspark), especificando ese jar:

```
bin/chispa-shell --jars postgresql-42.2.6.jar
```

Los siguientes ejemplos muestran cómo cargar y guardar en una base de datos PostgreSQL utilizando la API de fuente de datos Spark SQL y JDBC en Scala:

*// En Scala //*

Opción de lectura 1: carga de datos desde una fuente JDBC usando el método de carga `val`

```
jdbcDF1 = spark.read
```

```
.format("jdbc") .option("url", "jdbc:postgresql:  
[DBSERVER]") .option("dbtable", "[SCHEMA].  
[TABLENAME]").option("user", "[ NOMBRE DE  
USUARIO]").option("contraseña", "[CONTRASEÑA]").load()
```

```
// Opción de lectura 2: carga de datos desde una fuente JDBC usando el método jdbc // Crear  
propiedades de conexión import java.util.Properties val cxnProp = new Properties()  
cxnProp.put("usuario", "[NOMBRE DE USUARIO]") cxnProp.put ("contraseña", "[CONTRASEÑA]")
```

*// Cargar datos usando las propiedades de conexión `val` jdbcDF2*  
`= spark.read.jdbc("jdbc:postgresql:[DBSERVER]", "[SCHEMA].  
[TABLENAME]", cxnProp)`

```
// Opción de escritura 1: Guardar datos en una fuente JDBC utilizando el método de guardado  
jdbcDF1.write.format("jdbc") .option("url", "jdbc:postgresql:[DBSERVER]").option("dbtable",  
" [ESQUEMA][NOMBRE DE LA TABLA]").option("usuario", "[NOMBRE DE  
USUARIO]").option("contraseña", "[CONTRASEÑA]").save()
```

```
// Opción de escritura 2: Guardar datos en una fuente JDBC usando el método jdbc  
jdbcDF2.write.jdbc("jdbc:postgresql:[DBSERVER]", "[SCHEMA].[TABLENAME]", cxnProp)
```

Y aquí está cómo hacerlo en PySpark:

# en pitón

# Opción de lectura 1: carga de datos desde una fuente JDBC usando el método de carga

```
jdbcDF1 = (chispa
```

```

.read .format("jdbc") .option("url", "jdbc:postgresql://
[DBSERVER]") .option("dbtable", "[SCHEMA].
[TABLENAME]").option("usuario ", "[NOMBRE DE
USUARIO]").option("contraseña", "[CONTRASEÑA]").load()

# Opción de lectura 2: carga de datos desde una fuente JDBC utilizando el método jdbc
= (spark .read .jdbc("jdbc:postgresql://[DBSERVER]", "[SCHEMA].[TABLENAME]",

properties={"usuario": "[NOMBRE DE USUARIO]", "contraseña": "[CONTRASEÑA]"}))

# Opción de escritura 1: Guardar datos en una fuente JDBC usando el método de guardar
(jdbcDF1 .write .format("jdbc") .option("url", "jdbc:postgresql://[DBSERVER]") .option("dbtable",
"[ESQUEMA].[NOMBRE DE LA TABLA]").option("usuario", "[NOMBRE DE
USUARIO]").option("contraseña", "[CONTRASEÑA]").save())

# Opción de escritura 2: Guardar datos en una fuente JDBC usando el método jdbc
(jdbcDF2 .write .jdbc("jdbc:postgresql:[DBSERVER]", "[SCHEMA].[TABLENAME]",

properties={"usuario": "[NOMBRE DE USUARIO]", "contraseña": "[CONTRASEÑA]"}))

```

## MySQL

Para conectarse a una base de datos MySQL, cree o descargue el jar JDBC de [Maven](#) o [MySQL](#) (lo último es más fácil!) y agréguelo a su classpath. Luego inicie un shell Spark (spark-shell o pyspark), especificando ese jar:

```
bin/spark-shell --jars mysql-conector-java_8.0.16-bin.jar
```

Los siguientes ejemplos muestran cómo cargar datos y guardarlos en una base de datos MySQL utilizando la API de fuente de datos Spark SQL y JDBC en Scala:

```

// En Scala //
Carga de datos desde una fuente JDBC usando load val jdbcDF =
spark .read .format("jdbc") .option("url", "jdbc:mysql://
[DBSERVER]:3306/[DATABASE] ") .option("controlador",
"com.mysql.jdbc.Driver") .option("dbtable", "[NOMBRE DE LA
TABLA]").option("usuario", "[NOMBRE DE USUARIO]").option("contraseña ",
"[CONTRASEÑA]").cargar()

```

```
// Guardar datos en una fuente JDBC usando save
jdbcDF .write .format("jdbc") .option("url", "jdbc:mysql://
[DBSERVER]:3306/[DATABASE]") .option("driver ",
"com.mysql.jdbc.Driver") .option("dbtable", "[NOMBRE DE
LA TABLA]") .option("usuario", "[NOMBRE DE USUARIO]") .option("contraseña",
"[CONTRASEÑA]" ) .guardar()
```

Y aquí está cómo hacerlo en Python:

```
# En Python #
Cargando datos desde una fuente JDBC usando load jdbcDF =
(spark .read

.format("jdbc") .option("url", "jdbc:mysql://[DBSERVER]:3306/
[DATABASE]") .option("driver", "com.mysql.jdbc.Driver") . option("dbtable",
"[NOMBRE DE LA TABLA]") .option("usuario", "[NOMBRE DE
USUARIO]") .option("contraseña", "[CONTRASEÑA]") .load())
```

```
# Guardar datos en una fuente JDBC usando save
(jdbcDF .write .format("jdbc") .option("url", "jdbc:mysql://
[DBSERVER]:3306/[DATABASE]") .option("driver ",
"com.mysql.jdbc.Driver") .option("dbtable", "[NOMBRE
DE LA TABLA]") .option("usuario", "[NOMBRE DE USUARIO]") .option("contraseña",
"[CONTRASEÑA]" ) .guardar())
```

## Azure Cosmos DB

Para conectarse a una base de datos de Azure Cosmos DB, cree o descargue el archivo jar de JDBC de [Maven](#) o [GitHub](#) y agréguelo a su classpath. Luego inicie un shell Scala o PySpark, especificando este jar (tenga en cuenta que este ejemplo usa Spark 2.4):

```
bin/spark-shell --jars azure-cosmosdb-spark_2.4.0_2.11-1.3.5-uber.jar
```

También tiene la opción de usar --packages para extraer el conector de [Spark Packages](#) . usando sus coordenadas Maven:

```
exportar PKG="com.microsoft.azure:azure-cosmosdb-spark_2.4.0_2.11:1.3.5" bin/spark-shell --
paquetes $PKG
```

Los siguientes ejemplos muestran cómo cargar datos y guardarlos en una base de datos de Azure Cosmos DB mediante la API de origen de datos Spark SQL y JDBC en Scala y PySpark. Tenga en cuenta que es común usar la configuración query\_custom para hacer uso de los diversos índices dentro de Cosmos DB:

```
// En Scala //
Importar las bibliotecas necesarias import
com.microsoft.azure.cosmosdb.spark.schema._ import
com.microsoft.azure.cosmosdb.spark._ import
com.microsoft.azure.cosmosdb.spark.config.Config

// Cargando datos desde Azure Cosmos DB //
Configure la conexión a su colección val query = "SELECT
c.colA, c.colB FROM c WHERE c.origin = 'SEA'" val readConfig = Config(Map(
    "Punto final" -> "https://[CUENTA].documents.azure.com:443/", "Masterkey" ->
    "[MASTER KEY]", "Database" -> "[DATABASE]", "PreferredRegions" -> "Centro de EE.
    UU.; Este de EE. UU. 2;", "Colección" -> "[COLECCIÓN]", "Ratio de muestreo" -> "1.0",
    "consulta_personalizada" -> consulta )))

// Conéctese a través de azure-cosmosdb-spark para crear Spark DataFrame val df =
spark.read.cosmosDB(readConfig) df.count

// Guardar datos en Azure Cosmos DB
// Configurar la conexión a la colección receptora val writeConfig =
Config(Map(
    "Punto final" -> "https://[CUENTA].documents.azure.com:443/", "Masterkey" ->
    "[MASTER KEY]", "Database" -> "[DATABASE]", "PreferredRegions" -> "Centro de EE.
    UU.; Este de EE. UU. 2;", "Colección" -> "[COLECCIÓN]", "Tamaño de lote de escritura"
    -> "100" ))

// Upsert DataFrame a Azure Cosmos DB import
org.apache.spark.sql.SaveMode
df.write.mode(SaveMode.Overwrite).cosmosDB(writeConfig)

# En Python #
Carga de datos desde Azure Cosmos DB # Consulta
de configuración de lectura = "SELECCIONE c.colA ,
c.colB DESDE c DONDE c.origin = 'SEA'" readConfig = { "Punto final" : "https://
[CUENTA] .documents.azure.com:443/", "Masterkey" : "[MASTER KEY]", "Database" :
"[DATABASE]", "preferredRegions" : "Centro de EE. UU.; Este de EE. UU. 2",
```

```

    "Colección" : "[COLECCIÓN]", "Ratio de
    muestreo" : "1.0", "schema_sampleSize" :
    "1000", "query_pageSize" : "2147483647",
    "query_custom" : consulta

}

# Conéctese a través de azure-cosmosdb-spark para crear Spark DataFrame df =
(spark .read .format("com.microsoft.azure.cosmosdb.spark") .options(**readConfig) .load())

# Cuenta el número de vuelos df.count()

# Guardar datos en Azure Cosmos DB #
Configuración de escritura writeConfig = { "Punto
final" : "https://[CUENTA].documents.azure.com:443",
"Masterkey" : "[MASTER KEY]", "Database" : "[BASE DE DATOS]", "Colección" :
"[COLECCIÓN]", "Upsert" : "true" }

# Inserte el DataFrame en Azure Cosmos DB
(df.write .format("com.microsoft.azure.cosmosdb.spark") .options(**writeConfig) .save())

```

Para obtener más información, consulte la [documentación de Azure Cosmos DB](#).

## Servidor MS SQL

Para conectarse a una base de datos de MS SQL Server, [descargue el jar de JDBC](#) y agréguelo a su classpath. Luego inicie un shell de Scala o PySpark, especificando este jar:

```
bin/spark-shell --jars mssql-jdbc-7.2.2.jre8.jar
```

Los siguientes ejemplos muestran cómo cargar datos y guardarlos en una base de datos de MS SQL Server utilizando la API de fuente de datos Spark SQL y JDBC en Scala y PySpark:

```

// En escala
// Cargando datos desde una fuente JDBC
// Configurar jdbcUrl val
jdbcUrl = "jdbc:sqlserver://[DBSERVER]:1433;database=[DATABASE]"

// Crea un objeto Properties() para contener los parámetros.
// Tenga en cuenta que puede crear la URL de JDBC sin pasar los // parámetros de
usuario/contraseña directamente. val cxnProp = nuevas propiedades()
cxnProp.put("usuario", "[NOMBRE DE USUARIO]") cxnProp.put("contraseña",
"[CONTRASEÑA]") cxnProp.put("controlador", "com.microsoft.sqlserver.
jdbc.SQLServerDriver")

// Cargar datos usando las propiedades de conexión val jdbcDF
= spark.read.jdbc(jdbcUrl, "[TABLENAME]", cxnProp)

// Guardar datos en una fuente JDBC
jdbcDF.write.jdbc(jdbcUrl, "[TABLENAME]", cxnProp)

# en pitón
# Configurar jdbcUrl jdbcUrl
= "jdbc:sqlserver://[DBSERVER]:1433;base de datos=[BASE DE DATOS]"

# Carga de datos desde una fuente JDBC
jdbcDF = (spark .read .format("jdbc") .option("url",
jdbcUrl) .option("dbtable",
"[TABLENAME]") .option("user", " [NOMBRE
DE USUARIO]") .option("contraseña",
"[CONTRASEÑA]) .load())

# Guardar datos en una fuente JDBC
(jdbcDF .write .format("jdbc") .option("url",
jdbcUrl) .option("dbtable",
"[TABLENAME]") .option("user",
"[USERNAME ]") .opción("contraseña",
"[CONTRASEÑA]) .guardar())

```

## Otras fuentes externas

Hay solo algunas de las muchas fuentes de datos externas a las que Apache Spark se puede conectar; otras fuentes de datos populares incluyen:

- Apache Cassandra
- Copo de nieve
- MongoDB

## Funciones de orden superior en DataFrames y Spark SQL

Debido a que los tipos de datos complejos son amalgamas de tipos de datos simples, es tentador manipularlos directamente. Hay dos soluciones típicas para manipular tipos de datos complejos:

- Explosión de la estructura anidada en filas individuales, aplicación de alguna función y luego recreación de la estructura anidada
- Creación de una función definida por el usuario

Estos enfoques tienen la ventaja de permitirle pensar en el problema en formato tabular. Por lo general, implican (pero no se limitan a) el uso de funciones de utilidad como `get_json_object()`, `from_json()`, `to_json()`, `explode()` y `selectExpr()`.

Echemos un vistazo más de cerca a estas dos opciones.

### Opción 1: Explotar y recopilar

En esta instrucción SQL anidada, primero explotamos (valores), lo que crea una nueva fila (con la identificación) para cada elemento (valor) dentro de los valores:

```
-- En SQL
SELECCIONE id, collect_list(valor + 1) valores AS
DESDE (SELECCIONE id, EXPLOTAR (valores) COMO valor
       DESDE la tabla) x
AGRUPAR POR ID
```

Mientras que `collect_list()` devuelve una lista de objetos con duplicados, la instrucción GROUP BY requiere operaciones aleatorias, lo que significa que el orden de la matriz recopilada no es necesariamente el mismo que el de la matriz original. Como los valores pueden tener cualquier cantidad de dimensiones (una matriz muy amplia y/o muy larga) y estamos haciendo un GRUPO POR, este enfoque podría ser muy costoso.

### Opción 2: función definida por el usuario

Para realizar la misma tarea (agregar 1 a cada elemento en valores), también podemos crear un UDF que usa `map()` para iterar a través de cada elemento (valor) y realizar la operación de suma:

```
-- En SQL
SELECCIONE id, collect_list(valor + 1) valores AS
DESDE (SELECCIONE id, EXPLOTAR (valores) COMO valor
       DESDE la tabla) x
AGRUPAR POR ID
```

Entonces podríamos usar este UDF en Spark SQL de la siguiente manera:

```
spark.sql("SELECCIONE id, plusOneInt(valores) COMO valores DE la tabla").show()
```

Si bien esto es mejor que usar explotar() y recolectar\_list() ya que no habrá ninguna problemas de pedido, el proceso de serialización y deserialización en sí puede ser costoso.

Sin embargo, también es importante tener en cuenta que collect\_list() puede hacer que los ejecutores experimentar problemas de falta de memoria para grandes conjuntos de datos, mientras que el uso de UDF viate estos problemas.

## Funciones integradas para tipos de datos complejos

En lugar de usar estas técnicas potencialmente costosas, es posible que pueda usar algunas de las funciones integradas para tipos de datos complejos incluidos como parte de Apache Spark 2.4 y después. Algunos de los más comunes se enumeran en [la Tabla 5-3](#) (tipos de matriz) y [Tabla 5-4](#) (tipos de mapas); para ver la lista completa, consulte [este cuaderno](#) en los ladrillos de datos documentación.

Tabla 5-3. Funciones de tipo matriz

Función/Descripción	Consulta	Producción
array_distinct(array<T>): matriz<T>	SELECCIONE array_distinct(array(1, 2, 3, nulo));	[1,2,3, nulo]
Elimina duplicados dentro de una matriz		
array_intersect(matriz<T>, matriz<T>): matriz<T>	SELECCIONE array_inter secta(matriz(1, 2, 3), matriz(1, 3, 5));	[1,3]
Devuelve la intersección de dos matrices. sin duplicados		
matriz_union(matriz<T>, matriz<T>): matriz<T>	SELECCIONE array_union(array(1, 2, 3), array(1, 3, 5));	[1,2,3,5]
Devuelve la unión de dos matrices. sin duplicados		
matriz_excepto(matriz<T>, matriz<T>): matriz<T>	SELECCIONE arreglo_excepto(arreglo(1, 2, 3), arreglo(1, 3, 5));	[2]
Devuelve elementos en array1 pero no en array2, sin duplicados		
array_join(matriz<Cadena>, Cadena[, Cadena]): Cadena	SELECCIONE array_join(array('hola', 'mundo'), ''');	Hola Mundo
Concatena los elementos de un arreglo usando un delimitador		
matriz_max(matriz<T>): T	SELECCIONE array_max(matriz(1, 20, nulo, 3));	20
Devuelve el valor máximo dentro del formación; se omiten los elementos nulos		
matriz_min(matriz<T>): T	SELECCIONE array_min(matriz(1, 20, nulo, 3));	1
Devuelve el valor mínimo dentro del formación; se omiten los elementos nulos		

Función descriptiva	Consulta	Producción
posición_matriz(matriz<T>, T): largo	SELECCIONE posición_arreglo(arreglo(3, 2, 1), 1);	3
Devuelve el índice (basado en 1) del primer elemento de la matriz dada como un Long		
array_remove(matriz<T>, T): matriz<T>	SELECCIONE array_remove(array(1, 2, 3, nulo), 3);	[1,2, nulo]
Elimina todos los elementos que son iguales a el elemento dado de la matriz dada		
matrices_superposición(matriz<T>, matriz<T>): matriz<T>	SELECCIONE matrices_superposición (matriz (1, 2, 3), matriz (3, 4, 5));	verdadero
Devuelve verdadero si array1 contiene at menos un elemento no nulo también presente en array2		
ordenación_matriz(matriz<T>: matriz<T>)	SELECT array_sort(array('b', 'd', null, 'c', 'a'));	["a","b","c","d",null]
Ordena la matriz de entrada en orden ascendente orden, con elementos nulos colocados en el final de la matriz		
concat(matriz<T>, ...): matriz<T>	SELECCIONE concat(matriz(1, 2, 3), matriz(4, 5), matriz(6));	[1,2,3,4,5,6]
Concatena cadenas, binarios, arreglos, etc.		
aplanar(matriz<matriz<T>>): matriz<T>	SELECCIONE aplanar (arreglo (arreglo (1, 2), arreglo (3, 4)));	[1,2,3,4]
Aplana una matriz de matrices en una sola formación		
matriz_repetir(T, Int): matriz<T>	SELECCIONA array_repeat('123', 3); ["123","123","123"]	
Devuelve una matriz que contiene el elemento especificado el número especificado de tiempos		
inversa (matriz <T>): matriz<T>	SELECCIONE inversa (matriz (2, 1, 4, 3));	[3,4,1,2]
Devuelve una cadena invertida o una matriz con el orden inverso de los elementos		
secuencia(T, T[, T]): matriz<T>	SELECCIONAR secuencia (1, 5); SELECCIONAR secuencia (5, 1); SELECCIONE secuencia(to_date('2018-01-01'), to_date('2018-03-01'), entre valor 1 mes);	[1,2,3,4,5] [5,4,3,2,1] ["2018-01-01", "2018-02-01", "2018-03-01"]
Genera una matriz de elementos a partir de comienzo a parada (inclusive) por incremental paso		
barajar(matriz<T>): matriz<T>	SELECCIONE barajar (matriz (1, 20, nulo, 3));	[nulo,3,20,1]
Devuelve una permutación aleatoria de la matriz dada		

Función descriptiva	Consulta	Producción
segmento(matriz<T>, Int, Int): matriz<T>  Devuelve un subconjunto de la matriz dada a partir del índice dado (contando desde el final si el índice es negativo), de la longitud especificada	SELECCIONE segmento (matriz (1, 2, 3, 4), -2, 2);	[3,4]
matriz_zip(matriz<T>, matriz<U>, ...): matriz<estructura<T, U, ...>>  Devuelve una matriz combinada de estructuras	SELECCIONE arreglos_zip(arreglo(1, 2), arreglo(2, 3), arreglo(3, 4));	[{"0":1,"1":2,"2":3}, {"0":2,"1":3,"2":4}]
elemento_en(matriz<T>, Int): T/  Devuelve el elemento de la matriz dada en el índice dado (basado en 1)	SELECCIONE elemento_en(matriz(1, 2, 3), 2);	2
cardinalidad(matriz<T>): Int  Un alias de tamaño; devuelve el tamaño de la matriz dada o un mapa	SELECCIONE la cardinalidad (matriz ('b', 'd', 'c', 'a'));	4

Tabla 5-4. Funciones del mapa

Función/Descripción	Consulta	Producción
map_from_arrays(array<K>, array<V>): map<K, V>  Crea un mapa a partir del par dado de clave/valor arreglos; los elementos en las claves no deben ser nulos	SELECCIONE  map_from_arrays(matriz(1.0, 3.0), matriz('2', '4'));	{"1.0": "2", "3.0": "4"}
map_from_entries(matriz<estructura<K, V>>): mapa<K, V>  Devuelve un mapa creado a partir de la matriz dada	SELECCIONE  map_from_entries(array(struct(1, 'a'), struct(2, 'b')));	{"1": "a", "2": "b"}
map_concat(mapa<K, V>, ...): mapa<K, V>  Devuelve la unión de los mapas de entrada.	SELECCIONE map_concat(mapa(1, 'a', 2, 'b'), mapa(2, 'c', 3, 'd'));	{"1": "a", "2": "c", "3": "d"}
element_at(mapa<K, V>, K): V  Devuelve el valor de la clave dada, o nulo si la clave no está contenida en el mapa	SELECCIONE element_at(mapa(1, 'a', 2, 'b'), 2);	b
cardinalidad(matriz<T>): Int  Un alias de tamaño; devuelve el tamaño de lo dado matriz o un mapa	SELECCIONE la cardinalidad (mapa (1, 'a', 2, 'b'));	2

## Funciones de orden superior

Además de las funciones integradas mencionadas anteriormente, existen funciones de orden superior. ciones que toman funciones lambda anónimas como argumentos. Un ejemplo de una función de orden superior es la siguiente:

-- En  
**transformación SQL** (valores, valor -> expresión lambda)

La función transform() toma una matriz (valores) y una función anónima (expresión lambda) como entrada. La función crea de forma transparente una nueva matriz aplicando la función anónima a cada elemento y luego asignando el resultado a la matriz de salida (similar al enfoque UDF, pero más eficiente).

Vamos a crear un conjunto de datos de muestra para que podamos ejecutar algunos ejemplos:

```
# En Python
desde pyspark.sql.types import *
esquema = StructType([StructField("celsius", ArrayType(IntegerType()))])

t_list = [[35, 36, 32, 30, 40, 42, 38]], [[31, 32, 34, 55, 56]] t_c =
chispa.createDataFrame(t_list, esquema) t_c.createOrReplaceTempView("tC" )

# Mostrar el marco de
datos t_c.show()

// En Scala //
Crear DataFrame con dos filas de dos arreglos (tempc1, tempc2) val t1 = Array(35,
36, 32, 30, 40, 42, 38) val t2 = Array(31, 32, 34, 55 , 56) val tC = Seq(t1,
t2).toDF("celsius") tC.createOrReplaceTempView("tC")

// Mostrar el marco de datos
tC.show()
```

Aquí está la salida:

```
+-----+
|      centígrados|
+-----+
|[35, 36, 32, 30, ...] |[31, 32,
34, 55, 56]|
+-----+
```

Con el DataFrame anterior, puede ejecutar las siguientes consultas de función de orden superior.

### **transformar()**

transformar(matriz<T>, función<T, U>): matriz<U>

La función transform() produce una matriz al aplicar una función a cada elemento de la matriz de entrada (similar a una función map() ):

```
// En Scala/Python //
Calcula Fahrenheit a partir de Celsius para una serie de temperaturas spark.sql("""
```

SELECCIONE grados centígrados,

```
transformar(celsius, t -> ((t * 9) div 5) + 32) como farenheit
DESDE tC
""").show()
```

	centígrados	farenheit
35, 36, 32, 30, ...	95, 96, 89, 86, ...	
31, 32, 34, 55, 56]	87, 89, 93, 131, ...	

**filtro()**

```
filtro(matriz<T>, función<T, booleana>): matriz<T>
```

La función filter () produce una matriz que consta solo de los elementos de la entrada matriz para la cual la función booleana es verdadera:

```
// En Scala/Python
// Temperaturas de filtro > 38C para una variedad de temperaturas
chispa.sql("""
SELECCIONE
    celsius, filtro (celsius, t -> t > 38) tan alto
    DESDE tC
""").show()
```

	centígrados	alto
35, 36, 32, 30, ...	40, 42]	
31, 32, 34, 55, 56]	55, 56]	

**existe()**

```
existe(matriz<T>, función<T, V, Booleano>): Booleano
```

La función existe() devuelve verdadero si la función booleana se cumple para cualquier elemento en la matriz de entrada:

```
// En Scala/Python
// ¿Hay una temperatura de 38C en el conjunto de temperaturas?
chispa.sql("""
SELECCIONE
    celsius, existe (celsius, t -> t = 38) como umbral
    DESDE tC
""").show()
```

	centígrados	umbral
35, 36, 32, 30, ...		cierto

[[31, 32, 34, 55, 56]]	falso
------------------------	-------

**reducir()**

```
reducir(matriz<T>, B, función<B, T, B>, función<B, R>)
```

La función reduce () reduce los elementos de la matriz a un solo valor fusionando los elementos en un búfer B usando la función <B, T, B> y aplicando un acabado function<B, R> en el búfer final:

```
// En Scala/Python
// Calcular la temperatura promedio y convertir a F
chispa.sql("""
SELECCIONE celsius,
    reduza (
        celsius,
        0, (t, acc)
        -> t + acc, acc -> (tamaño
            div acc (celsius) * 9 div 5) + 32
        ) como avgFahrenheit
DESDE tC
""").show()
```

centígrados	promedio Fahrenheit
[[35, 36, 32, 30, ...]]	[[31, 32, 34, 55, 56]]
	96
	105

## Operaciones comunes de DataFrames y Spark SQL

Parte del poder de Spark SQL proviene de la amplia gama de operaciones de DataFrame (también conocidas como operaciones de conjuntos de datos sin tipo) que admite. La lista de operaciones es bastante extenso e incluye:

- Funciones agregadas
- Funciones de colección
- Funciones de fecha y hora
- Funciones matemáticas
- Funciones misceláneas
- Funciones no agregadas
- Funciones de clasificación
- Funciones de cadena

- Funciones UDF
- Funciones de ventana

Para ver la lista completa, consulte la [documentación de Spark SQL](#).

En este capítulo, nos centraremos en las siguientes operaciones relacionales comunes:

- Uniones y uniones •

Ventanas

- Modificaciones

Para realizar estas operaciones de DataFrame, primero prepararemos algunos datos. En el siguiente fragmento de código, nosotros:

1. Importe dos archivos y cree dos DataFrames, uno para la información del aeropuerto (airportsna) mación y otra para retrasos en vuelos de EE . UU . (departureDelays).
2. Usando expr(), convierta las columnas de retardo y distancia de STRING a INT.
3. Cree una tabla más pequeña, foo, en la que podamos centrarnos para nuestros ejemplos de demostración; contiene solo información sobre tres vuelos con origen en Seattle (SEA) hasta el destino de San Francisco (SFO) durante un pequeño intervalo de tiempo.

Empecemos:

```
// En Scala
importar org.apache.spark.sql.functions._

// Establecer rutas de
archivo val delaysPath =
  "/databricks-datasets/learning-spark-v2/flights/departuredelays.csv" val airportsPath =
  "/databricks-datasets/learning-spark-v2/flights/airport-codes-na.txt"

// Obtener conjunto de datos de
aeropuertos val airports = spark.read
  .option("header",
  "true") .option("inferSchema",
  "true") .option("delimiter",
  "\t") .csv(airportsPath)
airports.createOrReplaceTempView("airports_na")

// Obtener el conjunto de datos de retrasos de
salida val delays = spark.read
  .option("header", "true") .csv(delaysPath) .withColumn("retraso",
  expr("CAST(retraso como INT) como retraso")) .withColumn("distancia", expr("CAST(distancia como INT) como distancia"))
retrasos.createOrReplaceTempView("retrasos_de_salida")
```

```

// Crear una pequeña tabla temporal val
foo = delays.filter(
    expr("origen == 'SEA' Y destino == 'SFO' Y fecha como '01010%'"
        Y retraso > 0"))
    .createOrReplaceTempView("foo")

# En Python
# Establecer rutas
de archivos desde pyspark.sql.functions
import expr tripdelaysFilePath = "/databricks-
datasets/learning-spark-v2/flights/departuredelays.csv" airportsnaFilePath = "/databricks-
datasets/learning-spark-v2/flights /códigos-aeropuerto-na.txt"

# Obtener el conjunto de datos
de aeropuertos airportsna =

(spark.read .format("csv") .options(header="true", inferSchema="true",
sep="\t") .load(airportsnaFilePath))

aeropuertosna.createOrReplaceTempView("aeropuertos_na")

# Obtener el conjunto de datos de retrasos en
la salida de salidaDelays = (spark.read

.format("csv") .options(header="true") .load(tripdelaysFilePath))

Retrasos de salida = (retrasos de
salida .withColumn("retraso", expr("CAST(retraso como INT) como
retraso")) .withColumn("distancia", expr("CAST(distancia como INT) como distancia")))

Retrasos en la salida.createOrReplaceTempView("Retrasos en la salida")

# Crear una tabla pequeña temporal
foo = (departureDelays .filter(expr("origen
== 'SEA' y destino == 'SFO' y fecha como '01010%' y retraso > 0")))
    .createOrReplaceTempView("foo")

```

El marco de datos de retrasos de salida contiene datos sobre >1,3 millones de vuelos, mientras que el marco de datos foo contiene solo tres filas con información sobre vuelos de SEA a SFO para un rango de tiempo específico, como se indica en el siguiente resultado:

```

// Scala/Python
spark.sql("SELECCIONAR * DE airports_na LIMIT 10").show()

+-----+-----+
| Ciudad|Estado|País|ATA|
+-----+-----+
| Abbotsford| BC| Canadá| YXX|
| Aberdeen| SD| USA| ABR|

```

Abilene	Texas	Estados Unidos   ITB
Akron	OH	Estados Unidos   CAK
Álamoso	CO	Estados Unidos   COMO
Albany	GA	Estados Unidos   DBY
Albany	Nueva York	Estados Unidos   ALB
Albuquerque	NM	Estados Unidos   ABQ
Alejandría	Los Ángeles	Estados Unidos   AEX
Allentown	PA	Estados Unidos   ABE

```
chispa.sql("SELECCIONE * DESDE el LÍMITE de retrasos de salida 10").show()
```

fecha	traslado	distanza	origen	destino	
01011245	6	602	ABE	ATL	
01020600	-8	369	ABE	DTW	
01021245	-2	602	ABE	ATL	
01020605	-4	602	ABE	ATL	
01031245	-4	602	ABE	ATL	
01030605	0	602	ABE	ATL	
01041243	10	602	ABE	ATL	
01040605	28	602	ABE	ATL	
01051245	88	602	ABE	ATL	
01050605	9	602	ABE	ATL	

```
chispa.sql("SELECCIONAR * DE foo").mostrar()
```

fecha retraso distancia origen destino	OFS
[01010710 31  ] 590  MAR	OFS
01010955 104  ] 590  MAR	OFS
01010730 5  ] 590  MAR	OFS

En las siguientes secciones, ejecutaremos ejemplos de union, join y windowing con estos datos

## **sindicatos**

Un patrón común dentro de Apache Spark es unir dos DataFrames diferentes con el mismo esquema juntos. Esto se puede lograr usando el método `union()`:

```
// Escala  
// Unión de dos tablas  
val bar = retrasos.union(foo)  
barra.createOrReplaceTempView("barra")  
bar.filter(expr("""origen == 'MAR' Y destino == 'SFO'  
Y fecha COMO '010101' Y retraso > 0""")).show()
```

```
# en pitón
# Unión de dos mesas
bar = retrasos en la salida.union(foo)
barra.createOrReplaceTempView("barra")

# Mostrar la unión (filtrado por SEA y SFO en un rango de tiempo específico)
bar.filter(expr("""origen == 'MAR' Y destino == 'SFO'
Y fecha COMO '01010%' Y retraso > 0""").show()
```

La barra DataFrame es la unión de foo con retrasos. Utilizando los mismos criterios de filtrado da como resultado la barra DataFrame, vemos una duplicación de los datos foo , como se esperaba:

```
-- En SQL
chispa.sql("""
SELECCIONE *
DESDE
barra DONDE origen = 'MAR'
Y destino = 'SFO'
Y fecha COMO '01010%'
Y retraso > 0
""").show()

+-----+-----+-----+
| fecha|retraso|distancia|origen|destino|
+-----+-----+-----+
|01010710| 31|      590|MAR|    OFS|
|01010955| 104|      590|MARI|    OFS|
|01010730| 5|      590|MARI|    OFS|
|01010710| 31|      590|MARI|    OFS|
|01010955| 104|      590|MARI|    OFS|
|01010730| 5|      590|MARI|    OFS|
+-----+-----+-----+
```

## Uniones

Una operación común de DataFrame es unir dos DataFrames (o tablas). Por defecto, una unión Spark SQL es una unión interna, con las opciones interna, cruzada, externo, completo, completo\_externo , izquierdo, izquierdo\_externo , derecho, derecho\_externo, izquierdo\_semi y izquierdo\_anti. Hay más información disponible en la [documentación](#) . (esto es aplicable a Scala y Python).

El siguiente ejemplo de código realiza el valor predeterminado de una unión interna entre el aire Portsna y foo Marcos de datos:

```
// En escala
foo.join(
  aeropuertos.as('air'),
  $"aire.IATA" === $"origen"
).select("Ciudad", "Estado", "fecha", "retraso", "distancia", "destino").show()

# en pitón
# Unir los datos de retrasos de salida (foo) con la información del aeropuerto
```

```

foo.join(
    aeropuertos,
    aeropuertos.IATA == foo.origin
).select("Ciudad", "Estado", "fecha", "retraso", "distancia", "destino").show()

-- En SQL
chispa.sql("""
SELECCIONE una.Ciudad, un.Estado, f.fecha, f.retraso, f.distancia, f.destino
DESDE foo f
ÚNETE a airports_na a
ON a.IATA = f.origen
""").show()

```

El código anterior le permite ver la fecha, el retraso, la distancia y el destino información de foo DataFrame unida a la información de ciudad y estado de el marco de datos de los aeropuertos :

Ciudad	Estado	fecha	retraso	distancia	destino
Seattle	WA	0101710	31	590	OFS
Seattle	WA	01010955	104	590	OFS
Seattle	WA	01010730	5	590	OFS

## ventanas

Una [función de ventana](#) usa valores de las filas en una ventana (un rango de filas de entrada) para devuelve un conjunto de valores, normalmente en forma de otra fila. Con funciones de ventana, es posible operar en un grupo de filas y aún devolver un valor único para cada fila de entrada. En esta sección, mostraremos cómo usar la función de ventana `dense_rank()` .

ción; hay muchas otras funciones, como se indica en [la Tabla 5-5](#).

Tabla 5-5. Funciones de ventana

sql	API de marco de datos
<b>Clasificación de funciones</b>	
rank()	rango()
rango_denso()	rango denso ()
rango_porcentaje()	rangoporcentaje()
ntile()	ntil()
numero_fila()	numero de fila()
<b>Funciones analíticas</b>	
cume_dist()	distanciaCume ()
primer_valor() primerValor()	
último_valor()	último valor ()
retraso ()	retraso ()
Plomo()	Plomo()

Comencemos con una revisión de los TotalDelays (calculados por sum(Delay)) experimentados por vuelos con origen en Seattle (SEA), San Francisco (SFO) y la ciudad de Nueva York (JFK) e ir a un conjunto específico de ubicaciones de destino, como se indica a continuación consulta:

```
-- En SQL
DROP TABLE SI EXISTE salidaDelaysWindow ;

CREAR TABLA salidaDelaysWindow AS
SELECCIONE origen, destino, SUMA (retraso) AS TotalDelays
    DESDE salidaRetrasos
    DONDE origen EN ('SEA', 'SFO', 'JFK')
        Y destino EN ('SEA', 'SFO', 'JFK', 'DEN', 'ORD', 'LAX', 'ATL')
    AGRUPAR POR origen, destino;

SELECCIONE * DE la ventana de retrasos de salida
```

origen	destino	TotalDelays
JFK	PEDIDO	5608
MAR	LAX	9359
JFK	OF\$	35619
OF\$	PEDIDO	27412
JFK	DEN	4315
OF\$	DEN	18688
OF\$	MAR	17080
MAR	OF\$	22293
JFK	ATL	12141
OF\$	ATL	5091
MAR	DEN	13645
MAR	ATL	4535
MAR	PEDIDO	10041
JFK	MAR	7856
JFK	LAX	35755
OF\$	JFK	24100
OF\$	LAX	40798
MAR	JFK	4667

¿Qué pasaría si para cada uno de estos aeropuertos de origen quisiera encontrar los tres destinos que experimentó la mayor cantidad de retrasos? Puede lograr esto ejecutando tres consultas diferentes. ies para cada origen y luego unir los resultados, así:

```
-- En SQL
SELECCIONE origen, destino, SUMA (TotalDelays) AS TotalDelays
    DESDE salidaDelaysWindow
    WHERE origen = '[ORIGEN]'
    AGRUPAR POR origen, destino
    ORDEN POR SUMA(TotalDelays) DESC
    LÍMITE 3
```

donde [ORIGIN] son los tres valores de origen diferentes de JFK, SEA y SFO.

Pero un mejor enfoque sería utilizar una función de ventana como `dense_rank()` para realizar formar el siguiente cálculo:

```
-- En SQL
chispa.sql"""
SELECCIONE origen, destino, TotalDelays, rango
DESDE (
    SELECCIONE origen, destino, TotalDelays, dense_rank()
    OVER (PARTICIÓN POR origen ORDEN POR TotalDelays DESC) como rango
    DESDE salidaDelaysWindow
) t
DONDE rango <= 3
""").show()

+-----+-----+
|origen|destino|TotalDelays|rango|
+-----+-----+
|MAR|OFS|22293|1|
|MAR|DEN|13645|2|
|MAR|PEDIDO|10041|3|
|OFS|LAX|40798|1|
|OFS|PEDIDO|27412|2|
|OFS|JFK|24100|3|
|JFK|LAX|35755|1|
|JFK|OFS|35619|2|
|JFK|ATL|12141|3|
+-----+-----+
```

Mediante el uso de la función de ventana `dense_rank()` , podemos determinar rápidamente que el destino Las naciones con los peores retrasos para las tres ciudades de origen fueron:

- Seattle (SEA): San Francisco (SFO), Denver (DEN) y Chicago (ORD)
- San Francisco (SFO): Los Ángeles (LAX), Chicago (ORD) y Nueva York (JFK)
- Nueva York (JFK): Los Ángeles (LAX), San Francisco (SFO) y Atlanta (ATL)

Es importante tener en cuenta que cada grupo de ventanas debe caber en un único ejecutor y se compondrá en una sola partición durante la ejecución. Por lo tanto, necesita asegúrese de que sus consultas no sean ilimitadas (es decir, limite el tamaño de su ventana).

## Modificaciones

Otra operación común es realizar modificaciones al DataFrame. Mientras Los DataFrames en sí mismos son inmutables, puede modificarlos a través de operaciones que crear nuevos DataFrames diferentes, con diferentes columnas, por ejemplo. (Recordar de capítulos anteriores que los RDD subyacentes son inmutables, es decir, no pueden ser

cambiado, para garantizar que haya un linaje de datos para las operaciones de Spark). Comencemos con nuestro pequeño ejemplo anterior de DataFrame:

```
// En Scala/Python
foo.mostrar()

+-----+-----+-----+
| fecha|retraso|distancia|origen|destino|
+-----+-----+-----+
| 01010710| 31| | 590| MAR| OFS|
| 01010955| 104| | 590| MAR| OFS|
| 01010730| 5| | 590| MAR| OFS|
+-----+-----+-----+
```

### Agregar nuevas columnas

Para agregar una nueva columna al foo DataFrame, use el método `withColumn()` :

```
// En escala
importar org.apache.spark.sql.functions.expr
val foo2 = foo.withColumn(
    "estado",
    expr("CASO CUANDO retraso <= 10 ENTONCES 'A tiempo' ELSE 'Retrasado' FIN")
)

# en pitón
desde pyspark.sql.functions import expr
foo2 = (foo.withColumn(
    "estado",
    expr("CASO CUANDO retraso <= 10 ENTONCES 'A tiempo' ELSE 'Retrasado' FIN")
))
```

El recién creado `foo2` DataFrame tiene el contenido del `foo` DataFrame original más la columna de estado adicional definida por la sentencia CASE :

```
// En Scala/Python
foo2.mostrar()

+-----+-----+-----+-----+
| fecha|retraso|distancia|origen|destino| estado|
+-----+-----+-----+-----+
| 01010710| 31| | 590| MAR| OFS|Retrasado|
| 01010955| 104| | 590| MAR| OFS|Retrasado|
| 01010730| 5| | 590| MAR| SFO|a tiempo|
+-----+-----+-----+-----+
```

### Tirando columnas

Para soltar una columna, use el método `drop()` . Por ejemplo, eliminemos la columna de retraso . umn ya que ahora tenemos una columna de estado , agregada en la sección anterior:

```
// En escala
valor foo3 = foo2.drop("retraso")
foo3.mostrar()

# en pitón
foo3 = foo2.drop("retraso")
foo3.mostrar()

+-----+
| fecha|distancia|origen|destino| estado|
+-----+
| 01010710|      590| MAR|      OFS|Retrasado|
| 01010955|      590| MAR|      OFS|Retrasado|
| 01010730|      590| MAR|      SFO|a tiempo|
+-----+
```

**Cambiar el nombre de las columnas**

Puede cambiar el nombre de una columna usando el método rename() :

```
// En escala
val foo4 = foo3.withColumnRenamed("estado", "estado_vuelo")
foo4.mostrar()

# en pitón
foo4 = foo3.withColumnRenamed("estado", "estado_vuelo")
foo4.mostrar()

+-----+
| fecha|distancia|origen|destino|flight_status|
+-----+
| 01010710|      590| MAR|      OFS|      Retrasado|
| 01010955|      590| MAR|      OFS|      Retrasado|
| 01010730|      590| MAR|      OFS|      A tiempo|
+-----+
```

**pivotante**

Cuando trabaje con sus datos, a veces necesitará intercambiar las columnas por las filas, es decir, **pivote sus datos**. Tomemos algunos datos para demostrar este concepto:

```
-- En SQL
SELECCIONE destino, CAST(SUBSTRING(fecha, 0, 2) COMO int) COMO mes, retraso
DESDE salidaRetrasos
DONDE origen = 'MAR'

+-----+
|destino|mes|retraso|
+-----+
| PEDIDO | 1|    92|
| JFK| 1|    -7|
| DFW| 1|    -5|
|  | 1|    -3|
+-----+
```

```
+-----+-----+
| DFW | 1| -3|
| DFW | 1| 1|
| PEDIDO | 1| -10|
| DFW | 1| -6|
| DFW | 1| -2|
| PEDIDO | 1| -3|
+-----+
```

solo se muestran las 10 primeras filas

Girar le permite colocar nombres en la columna del mes (en lugar de 1 y 2 puede mostrar enero y febrero, respectivamente) así como realizar cálculos agregados (en este caso media y máx) sobre los retrasos por destino y mes:

```
-- En SQL
SELECCIONE * DESDE (
SELECCIONE destino, CAST(SUBSTRING(fecha, 0, 2) COMO int) COMO mes, retraso
DESDE los retrasos en la salida DONDE el origen = 'MAR'
)
PIVOTE (
CAST(PROMEDIO(retraso) COMO DECIMAL(4, 2)) COMO AvgDelay, MAX(retraso) COMO MaxDelay
PARA el mes EN (1 ENE, 2 FEB)
)
ORDENAR POR destino
```

```
+-----+-----+-----+-----+
|destino|JAN_AvgDelay|JAN_MaxDelay|FEB_AvgDelay|FEB_MaxDelay|
+-----+-----+-----+-----+
| ABQ | 19.86| 316| 11.42| 69|
| ANC| 4.44| 149| 7.90| 141|
| ATL| 11.98| 397| 7.73| 145|
| APAGADO| 3.48| 50| -0.21| 18|
| BOS| 7.84| 110| 14.58| 152|
| BUR| -2.03| 56| -1.89| 78|
| CLÉ| 16.00| 27| nulo| nulo|
| CLT| 2.53| 41| 12.96| 228|
| COS| 5.32| 82| 12.18| 203|
| CVG| -0.50| 4| nulo| nulo|
| DCA| -1.15| 50| 0.07| 34|
| DEN | 13.13| 425| 12.95| 625|
| DFW| 7.95| 247| 12.57| 356|
| DTWI| 9.18| 107| 3.47| 77|
| EEE| 9.63| 236| 5.20| 212|
| IFA | 1.84| 160| 4.21| 60|
| GRASA| 1.36| 119| 5.22| 232|
| FLL| 2.94| 54| 3.50| 40|
| GEG | 2.28| 63| 2.87| 60|
| HDN | -0.44| 27| -6.50| 0|
+-----+-----+-----+-----+
```

solo se muestran las 20 filas superiores

## Resumen

Este capítulo exploró cómo Spark SQL interactúa con componentes externos. Discutimos la creación de funciones definidas por el usuario, incluidas las UDF de Pandas, y presentamos algunas opciones para ejecutar consultas Spark SQL (incluido el shell Spark SQL, Beeline y Tableau). Luego proporcionamos ejemplos de cómo usar Spark SQL para conectarse con una variedad de fuentes de datos externas, como bases de datos SQL, PostgreSQL, MySQL, Tableau, Azure Cosmos DB, MS SQL Server y otros.

Exploramos las funciones integradas de Spark para tipos de datos complejos y dimos algunos ejemplos de cómo trabajar con funciones de orden superior. Finalmente, discutimos algunos operadores relacionales comunes y mostramos cómo realizar una selección de operaciones de DataFrame.

En el próximo capítulo, exploramos cómo trabajar con conjuntos de datos, los beneficios de las operaciones fuertemente tipadas y cuándo y por qué usarlas.



## CAPÍTULO 6

# Spark SQL y conjuntos de datos

En los Capítulos 4 y 5, cubrimos Spark SQL y la API de DataFrame. Analizamos cómo conectarse a fuentes de datos integradas y externas, echamos un vistazo al motor Spark SQL y exploramos temas como la interoperabilidad entre SQL y DataFrame, la creación y administración de vistas y tablas, y DataFrame y SQL avanzados. transformaciones.

Aunque presentamos brevemente la API de conjuntos de datos en el [Capítulo 3](#), analizamos los aspectos más destacados de cómo los conjuntos de datos (colecciones distribuidas fuertemente tipadas) se crean, almacenan, serializan y deserializan en Spark.

En este capítulo, profundizaremos en la comprensión de los conjuntos de datos: exploraremos el trabajo con conjuntos de datos en Java y Scala, cómo Spark administra la memoria para acomodar construcciones de conjuntos de datos como parte de la API de alto nivel y los costos asociados con el uso Conjuntos de datos.

## API única para Java y Scala

Como recordará del [Capítulo 3 \(Figura 3-1 y Tabla 3-6\)](#), los conjuntos de datos ofrecen una API unificada y singular para objetos fuertemente tipados. Entre los lenguajes admitidos por Spark, solo Scala y Java están fuertemente tipados; por lo tanto, Python y R solo admiten la API de DataFrame sin tipo.

Los conjuntos de datos son objetos tipificados específicos del dominio que se pueden operar en paralelo mediante programación funcional o los operadores DSL con los que está familiarizado de la API de tramas de datos.

Gracias a esta singular API, los desarrolladores de Java ya no corren el riesgo de quedarse atrás. Por ejemplo, cualquier interfaz futura o cambios de comportamiento en Scala groupBy(), flatMap(), map(),

o la API filter() también será la misma para Java, porque es una interfaz singular que es común a ambas implementaciones.

### Clases de casos de Scala y JavaBeans para conjuntos de datos

Si recuerda del [Capítulo 3 \(Tabla 3-2\)](#), Spark tiene tipos de datos internos, como StringType, BinaryType, IntegerType, BooleanType y MapType, que utiliza para mapear sin problemas los tipos de datos específicos del lenguaje en Scala y Java. durante las operaciones de Spark. Este mapeo se realiza a través de codificadores, que discutiremos más adelante en este capítulo.

Para crear Dataset[T], donde T es su objeto escrito en Scala, necesita una [clase de caso](#) que define el objeto. Usando nuestros datos de ejemplo del [Capítulo 3 \(Tabla 3-1\)](#), supongamos que tenemos un archivo JSON con millones de entradas sobre bloggers que escriben sobre Apache Spark en el siguiente formato:

```
{id: 1, primero: "Jules", último: "Damji", URL: "https://tinyurl.1", fecha: "4/1/2016",
visitas: 4535, campañas: {"twitter", "LinkedIn"}},
...
{id: 87, primero: "Brooke", último: "Wenig", url: "https://tinyurl.2", fecha: "5/5/2018", visitas:
8908, campañas: {"twitter", "LinkedIn"}}
```

Para crear un conjunto de datos [Bloggers] distribuido, primero debemos definir una clase de caso de Scala que defina cada campo individual que comprende un objeto de Scala. Esta clase de caso sirve como modelo o esquema para el objeto escrito Bloggers:

```
// En el caso
de Scala , clase Bloggers(id:Int, first:String, last:String, url:String, date:String, hits: Int,
Campaigns:Array[String])
```

Ahora podemos leer el archivo desde la fuente de datos:

```
val bloggers = "../data/bloggers.json" val
bloggersDS =
  chispa .read .format("json") .option("ruta",
bloggers) .load() .as[Bloggers]
```

Cada fila de la recopilación de datos distribuidos resultante es del tipo Bloggers.

De manera similar, puede crear una clase JavaBean de tipo Bloggers en Java y luego usar codificadores para crear un Dataset<Bloggers>:

```
// En Java
import org.apache.spark.sql.Encoders;
import java.io.Serializable;

Bloggers de clase pública implementa Serializable
{ private int id; cadena privada primero;
```

```

última cadena privada ; URL
de cadena privada ; Fecha
de cadena privada ; aciertos
internos privados ;
campañas privadas de Array[String] ;

// captadores y definidores de JavaBean int
getID() { return id; } void setID(int i) { id = i; }

String getFirst() { volver primero; } void establecerPrimero(String f) { primero = f; }
String getLast() { devuelve el último; } void setUltimo(String l) { último = l; }
String getURL() { devolver url; } void setURL(Cadena u) { url = u; }
String getDate() { fecha de regreso ; }
Void setDate(String d) { fecha = d; } int getHits()
{ devolver resultados; } void setHits(int h) { hits = h; }

Array[String] getCampaigns() { devolver campañas; } void setCampaigns(Array[String] c) { campañas = c; }

// Crear codificador
Codificador<Bloggers> BloggerEncoder = Encoders.bean(Bloggers.class); Cadena bloggers =
"../bloggers.json"
Conjunto de datos<Bloggers>bloggersDS = chispa

.read .format("json") .option("ruta",
bloggers) .load() .as(BloggerEncoder);

```

Como puede ver, la creación de conjuntos de datos en Scala y Java requiere un poco de previsión, ya que debe conocer todos los nombres y tipos de columnas individuales para las filas que está leyendo. A diferencia de DataFrames, donde opcionalmente puede dejar que Spark infiera el esquema, la API de conjunto de datos requiere que defina sus tipos de datos con anticipación y que su clase de caso o clase JavaBean coincida con su esquema.



Los nombres de los campos en la definición de clase de caso de Scala o de clase de Java deben coincidir con el orden en la fuente de datos. Los nombres de columna para cada fila en los datos se asignan automáticamente a los nombres correspondientes en la clase y los tipos se conservan automáticamente.

Puede usar una clase de caso de Scala existente o una clase de JavaBean si los nombres de los campos coinciden con sus datos de entrada. Trabajar con la API de Dataset es tan fácil, conciso y declarativo como trabajar con DataFrames. Para la mayoría de las transformaciones del conjunto de datos,

puede usar los mismos operadores relacionales que aprendió en los capítulos anteriores.

Examinemos algunos aspectos del trabajo con un conjunto de datos de muestra.

## Trabajar con conjuntos de datos

Una forma simple y dinámica de crear un conjunto de datos de muestra es usar una instancia de SparkSession . En este escenario, con fines ilustrativos, creamos dinámicamente un objeto Scala con tres campos: uid (identificación única para un usuario), uname (cadena de nombre de usuario generada aleatoriamente) y uso (minutos de uso del servidor o servicio).

### Creación de datos de muestra

Primero, generaremos algunos datos de muestra:

```
// En Scala
import scala.util.Random._

Nuestra clase de caso para la clase de
caso del conjunto de datos Uso(uid:Int, uname:String, uso: Int)
val r = new scala.util.Random(42)
// Crear 1000 instancias de scala Usage class // Esto
genera datos sobre la marcha val data = for (i <- 0 to 1000)

rendimiento (Uso(i, "usuario-" + r.alphanumeric.take(5).mkString(""),
r.nextInt(1000)))
// Crear un conjunto de datos de uso tipeado data
val dsUsage = spark.createDataset(data)
dsUsage.show(10)

+-----+
|uid|      uname|uso|
+-----+
| 0|usuario-Gpi2C| 525| | 1|
|usuario-DgXDi| 502| | 2|
|usuario-M66yO| 170| | 3|
|usuario-xTOn6| 913| | 4|
|usuario-3xGSz| 246| | 5|
|usuario-2aWRN| 727| | 6|
|usuario-EZZY1| 65| | 7|
|usuario-ZIZMZ| 935| | 8|
|usuario-VjxeG| 756| | 9|
|usuario-iqf1P| 3| |
+-----+
solo se muestran las 10 primeras filas
```

En Java la idea es similar, pero tenemos que usar Encoders explícitos (en Scala, Spark maneja esto implícitamente):

```

// En Java
importar org.apache.spark.sql.Encoders;
importar org.apache.commons.lang3.RandomStringUtils;
importar java.io.Serializable;
importar java.util.Random;
importar java.util.ArrayList;
importar java.util.List;

// Crear una clase Java como Bean
El uso de la clase pública implementa Serializable {

    fluido int ; // Identificación de usuario
    Cadena uname; // nombre de usuario
    uso int ; // uso

    Uso público (int uid, String uname, uso int ) {
        este.uid = uid;
        esto.uname = uname;
        this.uso = uso;
    }
    // Getters y setters de JavaBean public
    int getUid() { return this.uid; }
    public void setUid(int uid) { this.uid = uid; }
    public String getUsername() { devuelve esto.uname; }
    public void setUsername(String uname) { this.uname = uname; }
    public int getUsage() { return this.usage; }
    public void setUsage(int uso) { this.usage = uso; }

    Uso público () {
    }

    Cadena pública a Cadena () {
        return "uid: " + this.uid + "", uname: + this.usage + "" + este.uname + "",
        uso: + "";
    }
}

// Crear un Codificador explícito
Encoder <Usage> usoEncoder = Encoders.bean(Usage.class);
Rand aleatorio = new Random();
rand.setSeed(42);
List<Uso> datos = new ArrayList<Uso>()

// Crear 1000 instancias de la clase de uso de Java
para (int i = 0; i < 1000; i++) {
    datos.add(nuevo Uso(i, "usuario" +
        RandomStringUtils.randomAlphanumeric(5),
        rand.nextInt(1000)));

// Crear un conjunto de datos de datos escritos de uso
Conjunto de datos<Uso> dsUsage = chispa.createDataset(datos, usoEncoder);

```



El conjunto de datos generado entre Scala y Java diferirá porque el algoritmo de semilla aleatoria puede ser diferente. Por lo tanto, los resultados de la consulta de Scala y Java serán diferentes.

Ahora que tenemos nuestro conjunto de datos generado, `dsUsage` , realicemos algunas de las transformaciones comunes que hemos hecho en capítulos anteriores.

### Transformación de datos de muestra

Recuerde que los conjuntos de datos son colecciones fuertemente tipadas de objetos específicos del dominio. Estos objetos se pueden transformar en paralelo mediante operaciones funcionales o relacionales. Ejemplos de estas transformaciones incluyen `map()`, `reduce()`, `filter()`, `select()` y `added()`. Como ejemplos de [funciones de orden superior](#), estos métodos pueden tomar lambdas, cierres o funciones como argumentos y devolver los resultados. Como tales, se prestan bien a [la programación funcional](#).

Scala es un lenguaje de programación funcional y, más recientemente, también se han agregado a Java lambdas, argumentos funcionales y cierres. Probemos un par de funciones de orden superior en Spark y usemos construcciones de programación funcional con los datos de muestra que creamos anteriormente.

### Funciones de orden superior y programación

**funcional** Para un ejemplo simple, usemos `filter()` para devolver todos los usuarios en nuestro `dsUsage` Dataset cuyo uso excede los 900 minutos. Una forma de hacer esto es usar una expresión funcional como argumento para el método `filter()` :

```
// En Scala
import org.apache.spark.sql.functions._
dsUsage .filter(d => d.usage >
  900) .orderBy(desc("usage")) .show(5, false)
```

Otra forma es definir una función y proporcionar esa función como argumento para `filter()`:

```
def filterWithUsage(u: Uso) = u.usage > 900
dsUsage.filter(filterWithUsage(_)).orderBy(desc("uso")).show(5)

+-----+
|uid0|      uname|uso|
+-----+
|561|usuario-5n2xY| 999| |
|113|usuario-nnAXr| 999| |
|605|usuario-NL6c4| 999| |
|634|usuario-L0wci| 999|
```

```
|805|usuario-LX27o| 996|
+-----+

```

solo se muestran las 5 filas superiores

En el primer caso usamos una expresión lambda, {d.usage > 900}, como argumento para el método filter() , mientras que en el segundo caso definimos una función Scala, def filterWithUsage(u: Usage) = u.usage > 900. En ambos casos, el método filter() itera sobre cada fila del objeto Uso en el conjunto de datos distribuido y aplica la expresión o ejecuta la función, devolviendo un nuevo conjunto de datos de tipo Uso para las filas donde el valor de la expresión o función es verdadero (Consulte la [documentación de Scala](#) para obtener detalles de la firma del método).

En Java, el argumento de filter() es del tipo `FilterFunction<T>`. Esto se puede definir en línea de forma anónima o con una función con nombre. Para este ejemplo, definiremos nuestra función por nombre y la asignaremos a la variable f. La aplicación de esta función en filter() devolverá un nuevo conjunto de datos con todas las filas para las que nuestra condición de filtro es

verdadero:

```
// En Java
// Definir una función de filtro Java
FilterFunction<Uso> f = new FilterFunction<Uso>() {
    llamada booleana pública (Uso u)
    { return (u.uso > 900);
}
};

// Usa filter con nuestra función y ordena los resultados en orden descendente
dsUsage.filter(f).orderBy(col("usage").desc()).show(5);
```

```
+-----+
|uid|nombre      |usol
+-----+

```

```
|67|usuario-qCGvZ|997 ||
878|usuario-J2HUU|994 ||
668|usuario-pz2Lk|992 ||
750|usuario-0zWqR|991 ||
242|usuario-g0kF6|989 |
+-----+
```

solo se muestran las 5 filas superiores

No todas las lambdas o argumentos funcionales deben evaluar valores booleanos ; también pueden devolver valores calculados. Considere este ejemplo utilizando la función de orden superior map(), donde nuestro objetivo es averiguar el costo de uso para cada usuario cuyo valor de uso supera un cierto umbral para que podamos ofrecerles a esos usuarios un precio especial por minuto.

```
// En Scala //
Usa una expresión lambda if-then-else y calcula un valor dsUsage.map(u =>
{if (u.usage > 750) u.usage * .15 else u.usage * .50 })
.mostrar(5, falso)
// Definir una función para calcular el uso
```

```

def computeCostUsage(uso: Int): Double = { si (uso >
  750) uso * 0.15 uso si no } * 0.50

// Usar la función como argumento para map()
dsUsage.map(u => {computeCostUsage(u.usage)}).show(5, false)
+-----+
|valor |
+-----+
|262.5 ||
251.0 ||
85,0 ||
136.95|| 
123.0 |
+-----+
solo se muestran las 5 filas superiores

```

Para usar `map()` en Java, debe definir `MapFunction<T>`. Puede ser una clase anónima o una clase definida que amplíe `MapFunction<T>`. Para este ejemplo, lo usamos en línea, es decir, en la misma llamada al método:

```

// En Java //
Definir una MapFunction en línea
dsUsage.map((MapFunction<Usage, Double>) u -> { if
  (u.usage > 750) return u.usage else
    * 0,15;

      volver u.uso      * 0,50;
}, Codificadores.DOUBLE()).show(5); // Necesitamos especificar explícitamente el codificador
+-----+
|valor |
+-----+
|65,0 ||
114.45|| 
124.0 ||
132,6 ||
145,5 |
+-----+
solo se muestran las 5 filas superiores

```

Aunque hemos calculado valores para el costo de uso, no sabemos con qué usuarios están asociados los valores calculados. ¿Cómo obtenemos esta información?

Los pasos son simples:

1. Cree una clase de caso Scala o una clase JavaBean, `UsageCost`, con un campo adicional o columna denominada costo.
2. Defina una función para calcular el costo y utilícela en el método `map()`.

Así es como se ve esto en Scala:

```
// En escala
// Crear una nueva clase de caso con un campo adicional, clase de caso
de costo UsageCost(uid: Int, uname:String, use: Int, cost: Double)

// Calcular el costo de uso con Uso como parámetro // Devolver un
nuevo objeto, CostoUso def calcularUsoCostoUso(u: Uso):
CostoUso = { val v = if (u.uso > 750) u.uso * 0.15 else u.uso
CostoUso(u.uid, u.uname, u.uso, v) * 0.50

}

// Usar map() en nuestro conjunto de datos
original dsUsage.map(u => {computeUserCostUsage(u)}).show(5)

+-----+
|uid|      uname|uso| costo|
+-----+
| 0|usuario-Gpi2C| 525| 262.5| | 1|
usuario-DgXDi| 502| 251.0| | 2|
usuario-M66yO| 170| 85.0| | 3|usuario-
xTOn6| 913|136.95| | 4|usuario-3xGSz|
246| 123.0|
+-----+
solo se muestran las 5 filas superiores
```

Ahora tenemos un conjunto de datos transformado con una nueva columna, costo, calculado por la función en nuestra transformación map() , junto con todas las demás columnas.

Del mismo modo, en Java, si queremos el costo asociado con cada usuario, debemos definir una clase JavaBean UsageCost y MapFunction<T>. Para ver el ejemplo completo de JavaBean, consulte el repositorio de [GitHub del libro](#); por brevedad, solo mostraremos MapFunction<T> en línea aquí:

```
// En Java //
Obtenga el codificador para la clase JavaBean
Encoder<UsageCost> useCostEncoder = Encoders.bean(UsageCost.class);

// Aplicar la función map() a nuestros datos
dsUsage.map( (MapFunction<Usage, UsageCost>) u -> {
    doble v = 0,0;
    (u.uso > 750) v = u.uso * 0.15; else v = u.uso * 0.50; return new UsageCost(u.uid,
    u.uname,u.usage, v); }, codificador de costo de uso). mostrar (5);

+-----+
| costo|uid|      uname|uso|
+-----+
| 65.0| 0|usuario-xSyzf| 130| |114.45|
1|usuario-iOI72| 763| | 124.0| 2|
usuario-QHRUK| 248|
```

```
| 132.6| 3|usuario-8GTjo| 884| |
145.5| 4|usuario-U4cU1| 970|
+-----+
solo se muestran las 5 filas superiores
```

Hay algunas cosas que observar sobre el uso de funciones y conjuntos de datos de orden superior:

- Estamos utilizando objetos JVM tipificados como argumentos para las funciones.
  - Estamos usando la notación de puntos (de la programación orientada a objetos) para acceder a campos individuales dentro del objeto JVM escrito, lo que facilita la lectura.
  - Algunas de nuestras funciones y firmas lambda pueden tener seguridad de tipos, lo que garantiza la detección de errores en el tiempo de compilación e instruye a Spark sobre qué tipos de datos trabajar, qué operaciones realizar, etc. • Nuestro código es legible, expresivo y conciso, usando Java o Funciones del lenguaje Scala
- turas en expresiones lambda.
- Spark proporciona el equivalente de map() y filter() sin construcciones funcionales de orden superior tanto en Java como en Scala, por lo que no está obligado a usar programación funcional con Datasets o DataFrames. En su lugar, puede simplemente utilizar operadores DSL condicionales o expresiones SQL: por ejemplo, dsUsage.filter("uso > 900") o dsUsage(\$"uso" > 900). (Para obtener más información sobre esto, consulte ["Costos del uso de conjuntos de datos"](#) en la página 170).
  - Para los conjuntos de datos, usamos codificadores, un mecanismo para convertir de manera eficiente los datos entre JVM y el formato binario interno de Spark para sus tipos de datos (más información en ["Codificadores de conjuntos de datos"](#) en la página 168).



Las funciones de orden superior y la programación funcional no son exclusivas de Spark Datasets; también puede usarlos con DataFrames. Recuerde que un DataFrame es un Dataset[Row], donde Row es un objeto JVM genérico sin tipo que puede contener diferentes tipos de campos. La firma del método toma expresiones o funciones que operan en Fila, lo que significa que el tipo de datos de cada Fila puede ser un valor de entrada para la expresión o función.

#### **Conversión de tramas de datos en**

**conjuntos de datos** Para una verificación sólida de tipos de consultas y construcciones, puede convertir tramas de datos en conjuntos de datos. Para convertir un DataFrame df existente en un Dataset de tipo SomeCaseClass, simplemente use la notación df.as[SomeCaseClass]. Vimos un ejemplo de esto antes:

```
// En Scala
val bloggersDS = chispa .leer

.format("json") .option("ruta", "/data/bloggers/
bloggers.json") .load() .as[Bloggers]
```

spark.read.format("json") devuelve un DataFrame<Row>, que en Scala es un alias de tipo para Dataset[Row]. El uso de .as[Bloggers] le indica a Spark que use codificadores, que se analizan más adelante en este capítulo, para serializar/deserializar objetos de la representación de la memoria interna de Spark en objetos de JVM Bloggers .

## Gestión de memoria para conjuntos de datos y marcos de datos

Spark es un motor intensivo de big data distribuido en memoria, por lo que su uso eficiente de la memoria es crucial para su velocidad de ejecución.<sup>1</sup> A lo largo de su historial de lanzamiento, el uso de la memoria de Spark ha **evolucionado significativamente**:

- Spark 1.0 usaba objetos Java basados en RDD para el almacenamiento, serialización y deserialización de la memoria, lo cual era costoso en términos de recursos y lento. Además, el almacenamiento se asignó en el montón de Java, por lo que estaba a merced de la recolección de basura (GC) de JVM para grandes conjuntos de datos.
- Spark 1.x presentó **el Proyecto Tungsten**. Una de sus características destacadas fue un nuevo formato interno basado en filas para diseñar conjuntos de datos y marcos de datos en la memoria fuera del montón, utilizando compensaciones y punteros. Spark usa un mecanismo eficiente llamado codificadores para serializar y deserializar entre la JVM y su formato Tungsten interno. La asignación de memoria fuera del montón significa que Spark está menos sobrecargado por GC.
- Spark 2.x introdujo el **motor Tungsten de segunda generación**, con generación de código de etapa completa y diseño de memoria basado en columnas vectorizadas. Basada en ideas y técnicas de compiladores modernos, esta nueva versión también aprovechó las arquitecturas modernas de CPU y caché para un acceso rápido a datos paralelos con el enfoque de "instrucción única, datos múltiples" (SIMD).

---

<sup>1</sup> Para obtener más detalles sobre cómo Spark administra la memoria, consulte las referencias proporcionadas en el texto y las presentaciones "[Apache Spark Memory Management](#)" y "[Profundización en el Proyecto Tungsten acercando Spark al Bare Metal](#)".

### Codificadores de conjuntos de datos

Los codificadores convierten los datos en la memoria fuera del montón del formato Tungsten interno de Spark a objetos JVM Java. En otras palabras, serializan y deserializan objetos Dataset del formato interno de Spark a objetos JVM, incluidos los tipos de datos primitivos. Por ejemplo, un Encoder[T] se convertirá del formato Tungsten interno de Spark a Dataset[T].

Spark tiene soporte integrado para generar codificadores automáticamente para tipos primitivos (por ejemplo, cadena, entero, largo), clases de casos de Scala y JavaBeans. En comparación con la serialización y deserialización de Java y Kryo, los codificadores Spark son **significativamente más rápidos**.

En nuestro ejemplo anterior de Java, creamos explícitamente un codificador:

```
Codificador<UsageCost> useCostEncoder = Encoders.bean(UsageCost.class);
```

Sin embargo, para Scala, Spark genera automáticamente el código de bytes para estos convertidores eficientes. Echemos un vistazo al formato interno basado en filas de tungsteno de Spark.

### El formato interno de Spark versus el formato de objeto de Java

Los objetos de Java tienen grandes gastos generales: información de encabezado, código hash, información de Unicode, etc. Incluso una simple cadena de Java como "abcd" ocupa 48 bytes de almacenamiento, en lugar de los 4 bytes que podría esperar. Imagine la sobrecarga para crear, por ejemplo, un objeto MyClass(Int, String, String) .

En lugar de crear objetos basados en JVM para conjuntos de datos o marcos de datos, Spark asigna memoria Java fuera del montón para diseñar sus datos y emplea codificadores para convertir los datos de la representación en memoria a objetos JVM. Por ejemplo, la [Figura 6-1](#) muestra cómo se almacenaría internamente el objeto JVM MyClass(Int, String, String) .

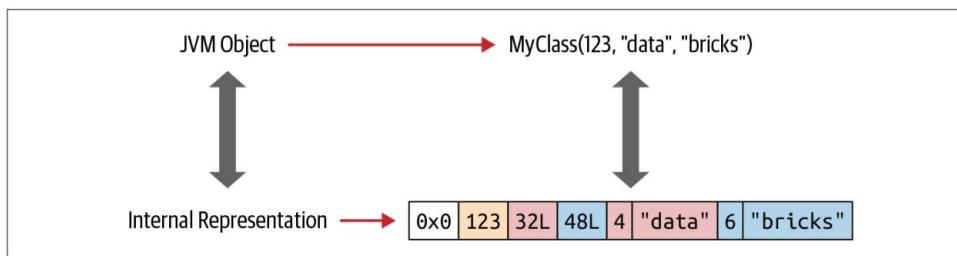


Figura 6-1. Objeto JVM almacenado en memoria Java fuera del montón contigua administrada por Spark

Cuando los datos se almacenan de esta manera contigua y son accesibles mediante aritmética de punteros y compensaciones, los codificadores pueden serializar o deserializar rápidamente esos datos. ¿Qué significa eso?

## Serialización y Deserialización (SerDe)

Un concepto no nuevo en la computación distribuida, donde los datos viajan con frecuencia a través de la red entre los nodos de la computadora en un clúster, la serialización y la deserialización es el proceso mediante el cual el remitente codifica (serializa) un objeto escrito en una presentación o formato binario y lo decodifica (deserializado) del formato binario en su respectivo objeto de tipo de datos por parte del receptor.

Por ejemplo, si el objeto JVM MyClass en la [figura 6-1](#) tuviera que compartirse entre nodos en un clúster de Spark, el remitente lo serializaría en una matriz de bytes y el receptor lo deserializaría nuevamente en un objeto JVM de tipo MyClass.

La JVM tiene su propio serializador y deserializador Java incorporado, pero es ineficiente porque (como vimos en la sección anterior) los objetos Java creados por la JVM en la memoria del montón están hinchados. Por lo tanto, el proceso es lento.

Aquí es donde los codificadores de Dataset vienen al rescate, por varias razones:

- El formato binario interno de tungsteno de Spark (consulte las Figuras [6-1](#) y [6-2](#)) almacena objetos fuera de la memoria del montón de Java y es compacto, por lo que esos objetos ocupan menos espacio.
- Los codificadores pueden serializarse rápidamente atravesando la memoria usando aritmética de punteros con direcciones de memoria y compensaciones ([Figura 6-2](#)).
- En el extremo receptor, los codificadores pueden deserializar rápidamente la representación binaria en la representación interna de Spark. Los codificadores no se ven obstaculizados por las pausas de recolección de basura de JVM.

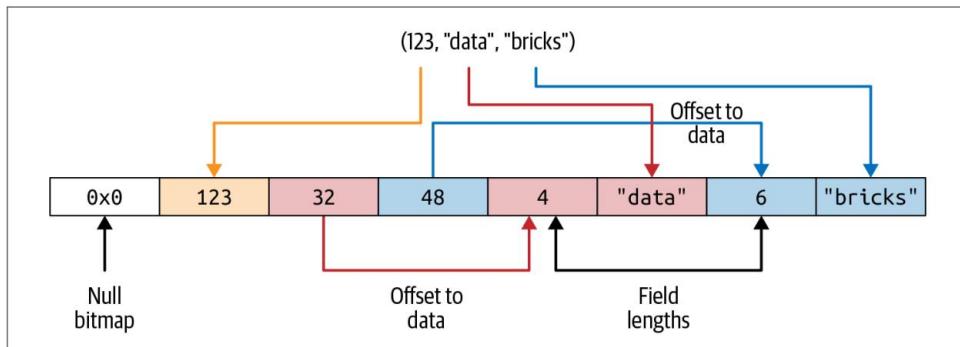


Figura 6-2. Formato interno basado en filas de tungsteno de Spark

Sin embargo, la mayoría de las cosas buenas de la vida tienen un precio, como veremos a continuación.

## Costos de usar conjuntos de datos

En “Frames de datos versus conjuntos de datos” en la página 74 en el Capítulo 3, describimos algunos de los beneficios del uso de conjuntos de datos, pero estos beneficios tienen un costo. Como se indicó en la sección anterior, cuando los conjuntos de datos se pasan a funciones de orden superior, como `filter()`, `map()` o `flatMap()` que toman lambdas y argumentos funcionales, existe un costo asociado con la deserialidad del formato Tungsten interno de Spark. en el objeto JVM.

En comparación con otros serializadores utilizados antes de que se introdujeron los codificadores en Spark, este costo es menor y tolerable. Sin embargo, en conjuntos de datos más grandes y muchas consultas, este costo se acumula y puede afectar el rendimiento.

### Estrategias para mitigar los costos Una

estrategia para mitigar el exceso de serialización y deserialización es usar expresiones DSL en sus consultas y evitar el uso excesivo de lambdas como funciones anónimas como argumentos para funciones de orden superior. Debido a que las lambdas son anónimas y opacas para el optimizador de Catalyst hasta el tiempo de ejecución, cuando las usa, no puede discernir de manera eficiente lo que está haciendo (no le está diciendo a Spark qué hacer) y, por lo tanto, no puede optimizar sus consultas (consulte “Catalyst Optimizer” en la página 77 en el Capítulo 3).

La segunda estrategia es encadenar sus consultas de tal manera que se minimice la serialización y la deserialización. Encadenar consultas juntas es una práctica común en Spark.

Ilustremos con un ejemplo sencillo. Supongamos que tenemos un conjunto de datos de tipo Persona, donde Persona se define como una clase de caso de Scala:

```
// En escala
Persona(id: entero, nombre: cadena, segundo nombre: cadena, apellido : cadena, género: cadena, fecha de nacimiento: cadena, ssn: cadena, salario: cadena)
```

Queremos emitir un conjunto de consultas a este conjunto de datos, utilizando la programación funcional.

Examinemos un caso en el que redactamos una consulta de manera ineficiente, de tal manera que, sin darnos cuenta, incurrimos en el costo de serialización y deserialización repetidas:

```
import java.util.Calendar val
firstYear = Calendar.getInstance.get(Calendar.YEAR) - 40

personaDS

// Todos los mayores de 40:
lambda-1 .filter(x => x.birthDate.split("-"))(0).toInt > firstYear )

// Todos ganan más de 80K .filter($"salario"
> 80000)
```

```
// El apellido comienza con J: lambda-2 .filter(x
=> x.lastName.startsWith("J"))

// El primer nombre comienza con
D .filter($"firstName".startsWith("D")) .count()
```

Como puede observar en la [Figura 6-3](#), cada vez que pasamos de lambda a DSL (filtro `($"salary" > 8000)`) incurrimos en el costo de serializar y deserializar el objeto Person JVM.

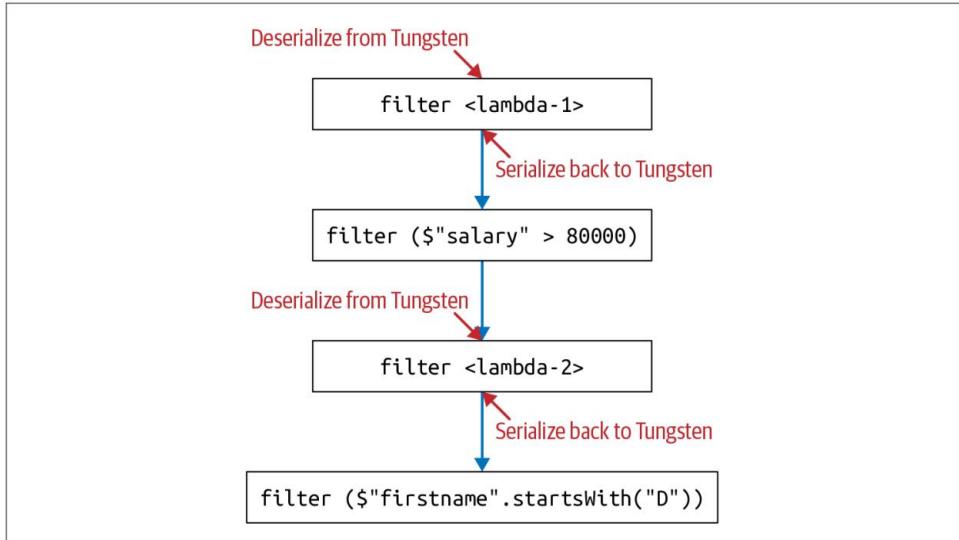


Figura 6-3. Una forma ineficiente de encadenar consultas con lambdas y DSL

Por el contrario, la siguiente consulta usa solo DSL y no lambdas. Como resultado, es mucho más eficiente: no se requiere serialización/deserialización para toda la consulta compuesta y encadenada:

```
personDS .filter(year($"birthDate") > firstYear) // Todos los mayores de
40 .filter($"salary" > 80000) // Todos ganan más de
80K .filter($"lastName".startsWith("J")) // El apellido comienza con
J .filter($"firstName".startsWith("D")) // El primer nombre comienza con D .count()
```

Para los curiosos, pueden ver la diferencia de tiempo entre las dos ejecuciones en el cuaderno de este capítulo en el repositorio de [GitHub del libro](#).

## Resumen

En este capítulo, explicamos cómo trabajar con conjuntos de datos en Java y Scala. Exploramos cómo Spark administra la memoria para acomodar construcciones de conjuntos de datos como parte de su API unificada y de alto nivel, y consideramos algunos de los costos asociados con el uso de conjuntos de datos y cómo mitigar esos costos. También le mostramos cómo usar las construcciones de programación funcional de Java y Scala en Spark.

Finalmente, echamos un vistazo bajo el capó a cómo los codificadores serializan y deserializan desde el formato binario interno de tungsteno de Spark a objetos JVM.

En el próximo capítulo, veremos cómo optimizar Spark examinando estrategias de E/S eficientes, optimizando y ajustando configuraciones de Spark, y qué atributos y señales buscar al depurar aplicaciones de Spark.

## CAPÍTULO 7

# Optimización y ajuste de aplicaciones Spark

En el capítulo anterior, explicamos cómo trabajar con conjuntos de datos en Java y Scala. Exploramos cómo Spark administra la memoria para acomodar construcciones de conjuntos de datos como parte de su API unificada y de alto nivel, y consideramos los costos asociados con el uso de conjuntos de datos y cómo mitigar esos costos.

Además de mitigar los costos, también queremos considerar cómo optimizar y ajustar Spark. En este capítulo, analizaremos un conjunto de configuraciones de Spark que permiten optimizaciones, veremos la familia de estrategias de combinación de Spark e inspeccionaremos la interfaz de usuario de Spark en busca de pistas sobre el mal comportamiento.

## Optimización y ajuste de Spark para la eficiencia

Si bien Spark tiene muchas configuraciones para [sintonizar](#), este libro solo cubrirá un puñado de las configuraciones más importantes y comúnmente ajustadas. Para obtener una lista completa agrupada por temas funcionales, puede consultar la [documentación](#).

### Visualización y establecimiento de configuraciones de Apache Spark

Existen tres formas de obtener y establecer las propiedades de Spark. La primera es a través de un conjunto de archivos de configuración. En el directorio \$SPARK\_HOME de su implementación (donde instaló Spark), hay varios archivos de configuración: conf/spark-defaults.conf.template, conf/log4j.properties.template y conf/spark-env.sh.template. Cambiar los valores predeterminados en estos archivos y guardarlos sin el sufijo .template indica a Spark que use estos nuevos valores.



Los cambios de configuración en el archivo conf/spark-defaults.conf se aplican al clúster de Spark y a todas las aplicaciones de Spark enviadas al clúster.

La segunda forma es especificar las configuraciones de Spark directamente en su aplicación Spark o en la línea de comando al enviar la aplicación con spark-submit, usando el indicador --conf :

```
spark-submit --conf spark.sql.shuffle.partitions=5 --conf
"spark.executor.memory=2g" --class main.scala.chapter7.SparkConfig_7_1 jars/main scala-
chapter7_2.12-1.0.jar
```

Así es como haría esto en la propia aplicación Spark:

```
// En Scala
importar org.apache.spark.sql.SparkSession

def printConfigs(session: SparkSession) = { // Obtener
    conf val mconf = session.conf.getAll // Imprimirlas
    para (k <- mconf.keySet) { println(s"${k} -> ${mconf(k)}\n") }

}

def main(args: Array[String]) { // Crear
    una sesión val spark =
        SparkSession.builder .config("spark.sql.shuffle.partitions",
            5) .config("spark.executor.memory", "2g"
            ") .master("local[*]") .appName("SparkConfig") .getOrCreate()

    printConfigs(spark)
    spark.conf.set("spark.sql.shuffle.partitions",
        spark.sparkContext.defaultParallelism)
    println(" ***** Configuración aleatoria de particiones en paralelismo predeterminado")
    printConfigs(spark) }

    spark.driver.host -> 10.8.154.34
    spark.driver.port -> 55243 spark.app.name
    -> SparkConfig spark.executor.id ->
    controlador spark.master -> local[*]
    spark.executor.memory -> 2g chispa.app.id
    -> local-1580162894307
    chispa.sql.shuffle.particiones -> 5
```

La tercera opción es a través de una interfaz programática a través de Spark Shell. Al igual que con todo lo demás en Spark, las API son el principal método de interacción. A través del objeto `SparkSession`, puede acceder a la mayoría de los ajustes de configuración de Spark.

En un Spark REPL, por ejemplo, este código de Scala muestra las configuraciones de Spark en un host local donde Spark se inicia en modo local (para obtener detalles sobre los diferentes modos disponibles, consulte "Modos de implementación" en la página 12 en el Capítulo 1):

```
// En Scala //
mconf es un Map[String, String] scala> val
mconf = spark.conf.getAll
...
scala> for (k <- mconf.keySet) { println(s"${k} -> ${mconf(k)}\n") }

spark.driver.host -> 10.13.200.101
spark.driver.port -> 65204 spark.repl.class.uri
-> spark://10.13.200.101:65204/classes spark.jars -> spark.repl.class.outputDir
-> /private/var/folders/jz/qg062ynx5v39wwmfmxph5nn... spark.app.name ->
Spark shell spark.submit.pyFiles -> spark.ui.showConsoleProgress -> true spark.executor.id -> driver
spark.submit.deployMode -> cliente spark.master -> local[*] spark.home -> /Users/julesdamji/spark/spark-3.0.0-
preview2-bin-hadoop2.7 spark.sql.catalogImplementation -> hive spark.app.id -> local-1580144503745
```

También puede ver solo las configuraciones de Spark específicas de Spark SQL:

```
// En Scala
spark.sql("SET -v").select("clave", "valor").show(5, falso)
# En Python
spark.sql("SET -v").select("clave", "valor").show(n=5, truncar=False)

+-----+-----+
|clave |valor |
+-----+-----+
|spark.sql.adaptive.habilitado |falso |spark.sql.adaptive.nonEmptyPartitionRatioForBroadcastJoin |
|0.2 |spark.sql.adaptive.shuffle.fetchShuffleBlocksInBatch.enabled|true |||
spark.sql.adaptive.shuffle.localShuffleReader.enabled |true |||
spark.sql.adaptive.shuffle.maxNumPostShufflePartitions |<undefined>

+-----+
solo se muestran las 5 filas superiores
```

Como alternativa, puede acceder a la configuración actual de Spark a través de la pestaña Entorno de la interfaz de usuario de Spark, que trataremos más adelante en este capítulo, como valores de solo lectura, como se muestra en la Figura 7-1.

The screenshot shows the Apache Spark 3.0.0-preview2 user interface with the 'Environment' tab selected. The 'Runtime Information' section contains the following table:

Name	Value
Java Home	/Library/Java/JavaVirtualMachines/jdk1.8.0_241.jdk/Contents/Home/jre
Java Version	1.8.0_241 (Oracle Corporation)
Scala Version	version 2.12.10

The 'Spark Properties' section contains the following table:

Name	Value
spark.app.id	local-1591198945354
spark.app.name	Spark shell

Figura 7-1. La pestaña Entorno de la interfaz de usuario de Spark 3.0

Para establecer o modificar una configuración existente mediante programación, primero verifique si la propiedad es modificable. `spark.conf.isModifiable("<config_name>")` devolverá verdadero o falso. Todas las configuraciones modificables se pueden establecer en nuevos valores utilizando la API:

```
// En Scala
scala> spark.conf.get("spark.sql.shuffle.partitions") res26:
String = 200 scala> spark.conf.set("spark.sql.shuffle.partitions",
5) scala> spark .conf.get("spark.sql.shuffle.partitions") res28:
Cadena = 5

# En Python
>>> chispa.conf.get("chispa.sql.shuffle.particiones")
'200' >>> chispa.conf.set("chispa.sql.shuffle.particiones",
5) >>> chispa .conf.get("chispa.sql.shuffle.particiones") '5'
```

Entre todas las formas en que puede configurar las propiedades de Spark, un orden de precedencia determina qué valores se respetan. Primero se leerán los valores o indicadores definidos en `spark-defaults.conf`, seguidos de los proporcionados en la línea de comando con envío de `chispa` y, finalmente, los establecidos a través de `SparkSession` en la aplicación Spark. Todas estas propiedades se fusionarán y cualquier propiedad duplicada que se restablezca en la aplicación Spark tendrá prioridad. Del mismo modo, los valores proporcionados en la línea de comando reemplazarán las configuraciones en el archivo de configuración, siempre que no se sobreescrbían en la propia aplicación.

Ajustar o proporcionar las configuraciones correctas ayuda con el rendimiento, como verá en la siguiente sección. Las recomendaciones aquí se derivan de las observaciones de los profesionales en la comunidad y se centran en cómo maximizar la utilización de los recursos del clúster para que Spark acomode cargas de trabajo a gran escala.