

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.

Escalado de Spark para cargas de trabajo

grandes Las cargas de trabajo grandes de Spark suelen ser trabajos por lotes: algunas se ejecutan todas las noches, mientras que otras se programan a intervalos regulares durante el día. En cualquier caso, estos trabajos pueden procesar decenas de terabytes de datos o más. Para evitar fallas en el trabajo debido a la escasez de recursos o la degradación gradual del rendimiento, hay varias configuraciones de Spark que puede habilitar o modificar. Estas configuraciones afectan a tres componentes de Spark: el controlador de Spark, el ejecutor y el servicio de reproducción aleatoria que se ejecuta en el ejecutor.

La responsabilidad del controlador de Spark es coordinarse con el administrador del clúster para lanzar ejecutores en un clúster y programar tareas de Spark en ellos. Con grandes cargas de trabajo, puede tener cientos de tareas. Esta sección explica algunas configuraciones que puede modificar o habilitar para optimizar la utilización de sus recursos, paralelizar tareas y evitar cuellos de botella para una gran cantidad de tareas. Algunas de las ideas y conocimientos de optimización se derivaron de empresas de big data como Facebook que usan Spark a escala de terabytes, que compartieron con la comunidad de Spark en Spark + AI Summit.¹

Asignación de recursos estática frente a

dinámica Cuando especifica recursos informáticos como argumentos de la línea de comandos para el envío rápido, como hicimos anteriormente, limita el límite. Esto significa que si se necesitan más recursos más adelante, ya que las tareas se ponen en cola en el controlador debido a una carga de trabajo mayor que la anticipada, Spark no puede acomodar ni asignar recursos adicionales.

Si, en cambio, utiliza la [configuración de asignación dinámica de recursos de Spark](#), el controlador Spark puede solicitar más o menos recursos informáticos a medida que la demanda de grandes cargas de trabajo fluya y disminuya. En escenarios en los que sus cargas de trabajo son dinámicas, es decir, varían en su demanda de capacidad informática, el uso de la asignación dinámica ayuda a adaptarse a los picos repentinos.

Un caso de uso en el que esto puede ser útil es la transmisión, donde el volumen del flujo de datos puede ser desigual. Otro es el análisis de datos bajo demanda, donde puede tener un gran volumen de consultas SQL durante las horas pico. Habilitar la asignación dinámica de recursos permite a Spark lograr una mejor utilización de los recursos, liberando ejecutores cuando no están en uso y adquiriendo nuevos cuando es necesario.



Además de cuando se trabaja con cargas de trabajo grandes o variables, la asignación dinámica también es útil [en un entorno de múltiples inquilinos](#), donde Spark se puede implementar junto con otras aplicaciones o servicios en YARN, Mesos o Kubernetes. Tenga en cuenta, sin embargo, que las cambiantes demandas de recursos de Spark pueden afectar a otras aplicaciones que demandan recursos al mismo tiempo.

¹ Consulte "[Ajuste de Apache Spark para cargas de trabajo a gran escala](#)" y "[Hive Bucketing en Apache Spark](#)".

Para habilitar y configurar la asignación dinámica, puede usar configuraciones como las siguientes. Tenga en cuenta que los números aquí son arbitrarios; la configuración adecuada dependerá de la naturaleza de su carga de trabajo y debe ajustarse en consecuencia. Algunas de estas configuraciones no se pueden configurar dentro de un Spark REPL, por lo que deberá configurarlas mediante programación:

```
spark.dynamicAllocation.enabled true  
spark.dynamicAllocation.minExecutors 2  
spark.dynamicAllocation.schedulerBacklogTimeout 1m  
spark.dynamicAllocation.maxExecutors 20  
spark.dynamicAllocation.executorIdleTimeout 2min
```

Por defecto, spark.dynamicAllocation.enabled se establece en falso. Cuando se habilita con la configuración que se muestra aquí, el controlador Spark solicitará que el administrador del clúster cree dos ejecutores para comenzar, como mínimo (spark.dynamicAllocation.minExecutors). A medida que aumenta el trabajo pendiente de la cola de tareas, se solicitarán nuevos ejecutores cada vez que se exceda el tiempo de espera del trabajo pendiente (spark.dynamicAllocation.schedulerBacklogTimeout). En este caso, siempre que existan tareas pendientes que no hayan sido programadas durante más de 1 minuto, el conductor solicitará que se lance un nuevo ejecutor para programar las tareas atrasadas, hasta un máximo de 20 (spark.dynamicAllocation.maxExecutors). Por el contrario, si un ejecutor finaliza una tarea y está inactivo durante 2 minutos (spark.dynamicAllocation.executorIdleTimeout), el controlador Spark lo terminará.

Configurar la memoria de los ejecutores de Spark y el servicio

de reproducción aleatoria No basta con habilitar la asignación dinámica de recursos. También debe comprender cómo Spark distribuye y utiliza la memoria del ejecutor para que los ejecutores no se queden sin memoria ni tengan problemas con la recolección de elementos no utilizados de JVM.

La cantidad de memoria disponible para cada ejecutor está controlada por spark.executor.memory. Esto se divide en tres secciones, como se muestra en [la Figura 7-2](#): memoria de ejecución, memoria de almacenamiento y memoria reservada. La división predeterminada es 60 % para la memoria de ejecución y 40 % para el almacenamiento, después de permitir 300 MB para la memoria reservada, para protegerse contra errores OOM. La [documentación](#) de Spark informa que esto funcionará para la mayoría de los casos, pero puede ajustar qué fracción de spark.executor.memory desea que cada sección use como línea de base. Cuando no se utiliza la memoria de almacenamiento, Spark puede adquirirla para usarla en la memoria de ejecución con fines de ejecución, y viceversa.

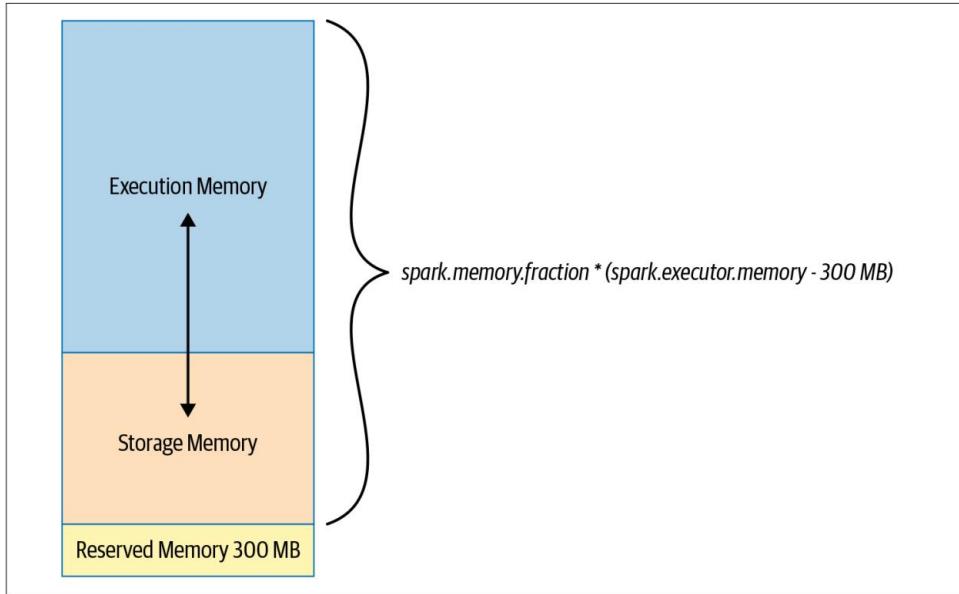


Figura 7-2. Disposición de la memoria del ejecutor

La memoria de ejecución se usa para mezclas, combinaciones, clasificaciones y agregaciones de Spark. Dado que diferentes consultas pueden requerir diferentes cantidades de memoria, la fracción (`spark.memory.fraction` es 0.6 por defecto) de la memoria disponible para dedicar a esto puede ser difícil de ajustar pero es fácil de ajustar. Por el contrario, la memoria de almacenamiento se utiliza principalmente para almacenar en caché estructuras de datos de usuario y particiones derivadas de DataFrames.

Durante las operaciones de asignación y reproducción aleatoria, Spark escribe y lee los archivos de reproducción aleatoria del disco local, por lo que hay una gran actividad de E/S. Esto puede resultar en un cuello de botella, porque las configuraciones predeterminadas no son óptimas para trabajos de Spark a gran escala. Saber qué configuraciones modificar puede mitigar este riesgo durante esta fase de un trabajo de Spark.

En la Tabla 7-1, capturamos algunas configuraciones recomendadas para ajustar de modo que los procesos de mapeo, derrame y combinación durante estas operaciones no se vean obstaculizados por E/S ineficientes y para permitir que estas operaciones empleen memoria de búfer antes de escribir las particiones aleatorias finales al disco. **Sintonización del servicio de reproducción aleatoria** ejecutarse en cada ejecutor también puede ayudar a aumentar el rendimiento general para grandes cargas de trabajo de Spark.

Tabla 7-1. Configuraciones de Spark para modificar E/S durante las operaciones de mapa y reproducción aleatoria

Configuración	Valor predeterminado, recomendación y descripción
chispa.driver.memoria	es 1 g (1 GB). Esta es la cantidad de memoria asignada al controlador Spark para recibir datos de los ejecutores. Esto a menudo se cambia durante el envío de Spark con --driver-memory.
chispa.shuffle.archivo.búfer	Solo cambie esto si espera que el controlador reciba grandes cantidades de datos de operaciones como recopilar (), o si se queda sin memoria del controlador.
chispa.archivo.transferir a	El valor predeterminado es verdadero. Establecerlo en falso forzará a Spark a usar el búfer de archivos para transferir archivos antes de finalmente escribirlos en el disco; esto disminuirá la actividad de E/S.
chispa.shuffle.unsafe.file.out.put.buffer	El valor predeterminado es 32 KB. Esto controla la cantidad de almacenamiento en búfer posible al fusionar archivos durante las operaciones aleatorias. En general, los valores grandes (p. ej., 1 MB) son más apropiados para cargas de trabajo más grandes, mientras que el valor predeterminado puede funcionar para cargas de trabajo más pequeñas.
chispa.io.compression.lz4.block.Tamaño	El valor predeterminado es 32 KB. Aumentar a 512 KB. Puede disminuir el tamaño de la reproducción aleatoria archivo aumentando el tamaño comprimido del bloque.
servicio.shuffle.de.chispa.index.cache.size	El valor predeterminado es 100 m. Las entradas de caché están limitadas al espacio de memoria especificado en bytes.
chispa.shuffle.registro. se acabó el tiempo	El valor predeterminado es 5000 ms. Aumente a 120000 ms.
chispa.shuffle.registro.max Intentos	El valor predeterminado es 3. Aumente a 5 si es necesario.



Las recomendaciones de esta tabla no funcionarán para todas las situaciones, pero deberían darle una idea de cómo ajustar estas configuraciones en función de su carga de trabajo. Al igual que con todo lo demás en el ajuste de rendimiento, debe experimentar hasta encontrar el equilibrio adecuado.

Maximizar el paralelismo de

Spark Gran parte de la eficiencia de Spark se debe a su capacidad para ejecutar múltiples tareas en paralelo a escala. Para comprender cómo puede maximizar el paralelismo, es decir, leer y procesar tantos datos en paralelo como sea posible, debe analizar cómo Spark lee los datos en la memoria desde el almacenamiento y qué significan las particiones para Spark.

En el lenguaje de gestión de datos, una partición es una forma de organizar los datos en un subconjunto de fragmentos configurables y legibles o bloques de datos contiguos en el disco. Estos subconjuntos de datos pueden leerse o procesarse de forma independiente y en paralelo, si es necesario, por más de un hilo en un proceso. Esta independencia es importante porque permite un paralelismo masivo del procesamiento de datos.

Spark es vergonzosamente eficiente en el procesamiento de sus tareas en paralelo. Como aprendió en el [Capítulo 2](#), para cargas de trabajo a gran escala, un trabajo de Spark tendrá muchas etapas, y dentro de

cada etapa habrá muchas tareas. Spark programará, en el mejor de los casos, un subproceso por tarea por núcleo, y cada tarea procesará una partición distinta. Para optimizar la utilización de recursos y maximizar el paralelismo, lo ideal es al menos tantas particiones como núcleos haya en el ejecutor, como se muestra en la [Figura 7-3](#). Si hay más particiones que núcleos en cada ejecutor, todos los núcleos se mantienen ocupados. Puede pensar en las particiones como unidades atómicas de paralelismo: un solo subproceso que se ejecuta en un solo núcleo puede funcionar en una sola partición.

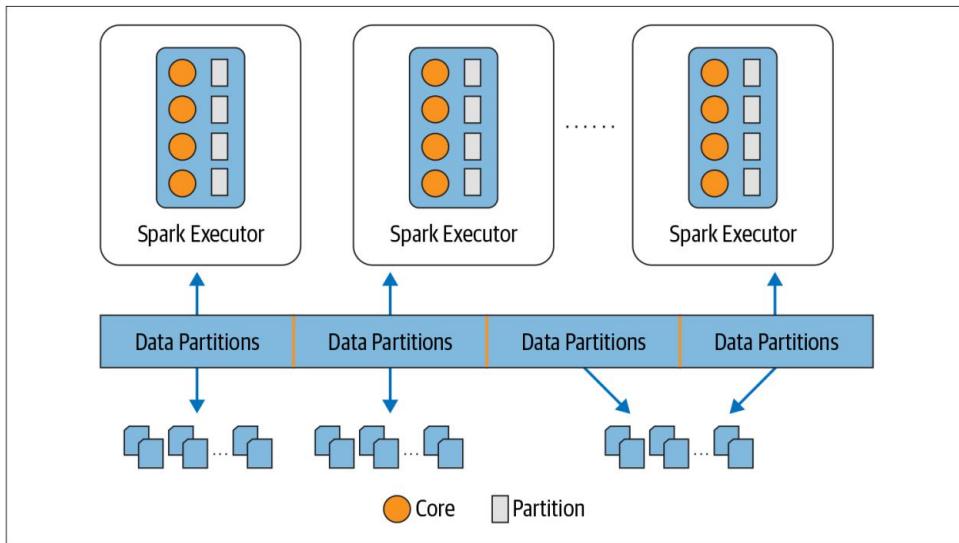


Figura 7-3. Relación de tareas, núcleos, particiones y paralelismo de Spark

Cómo se crean las particiones. Como se mencionó anteriormente, las tareas de Spark procesan datos a medida que las particiones leen del disco a la memoria. Los datos en el disco se distribuyen en fragmentos o bloques de archivos contiguos, según el almacen. De forma predeterminada, los bloques de archivos en los almacenes de datos varían en tamaño de 64 MB a 128 MB. Por ejemplo, en HDFS y S3, el tamaño predeterminado es 128 MB (esto es configurable). Una colección contigua de estos bloques constituye una partición.

El tamaño de una partición en Spark está dictado por `spark.sql.files.maxPartitionBytes`.

El valor predeterminado es 128 MB. Puede reducir el tamaño, pero eso puede resultar en lo que se conoce como el "problema de archivos pequeños": muchos archivos de partición pequeños, lo que introduce una cantidad excesiva de E/S de disco y degradación del rendimiento gracias a las operaciones del sistema de archivos, como abrir, cerrar y listar. directorios, que en un sistema de archivos distribuido puede ser lento.

Las particiones también se crean cuando usa explícitamente ciertos métodos de la API de DataFrame. Por ejemplo, mientras crea un DataFrame grande o lee un archivo grande del disco, puede indicar explícitamente a Spark que cree una cierta cantidad de particiones:

```
// En Scala
val ds = spark.read.textFile("../README.md").repartition(16) ds:
org.apache.spark.sql.Dataset[String] = [value: string]

ds.rdd.getNumPartitions
res5: Int = 16

val numDF = chispa.rango (1000L * 1000 * 1000) .repartición (16)
numDF.rdd.getNumPartitions

numDF: org.apache.spark.sql.Dataset[Largo] = [id: bigint] res12:
Int = 16
```

Finalmente, las particiones aleatorias se crean durante la etapa aleatoria. De forma predeterminada, el número de particiones aleatorias se establece en 200 en `spark.sql.shuffle.partitions`. Puede ajustar este número según el tamaño del conjunto de datos que tenga, para reducir la cantidad de pequeñas particiones que se envían a través de la red a las tareas de los ejecutores.



El valor predeterminado para `spark.sql.shuffle.partitions` es demasiado alto para cargas de trabajo más pequeñas o de transmisión; es posible que desee reducirlo a un valor más bajo, como la cantidad de núcleos en los ejecutores o menos.

Creadas durante operaciones como `groupBy()` o `join()`, también conocidas como transformaciones amplias, las particiones aleatorias consumen recursos de E/S tanto de red como de disco. Durante estas operaciones, la reproducción aleatoria derramará los resultados en los discos locales de los ejecutores en la ubicación especificada en `spark.local.directory`. Tener discos SSD de alto rendimiento para esta operación aumentará el rendimiento.

No existe una fórmula mágica para establecer el número de particiones aleatorias para la etapa aleatoria; la cantidad puede variar según su caso de uso, el conjunto de datos, la cantidad de núcleos y la cantidad de memoria del ejecutor disponible; es un enfoque de prueba y error.² Además de escalar Spark para grandes cargas de trabajo, para aumentar su rendimiento, Querrá considerar el almacenamiento en caché o la persistencia de sus marcos de datos o tablas a los que se accede con frecuencia.

Exploramos varias opciones de almacenamiento en caché y persistencia en la siguiente sección.

² Para obtener algunos consejos sobre la configuración de particiones aleatorias, consulte "Ajuste de Apache Spark para cargas de trabajo a gran escala", "Hive Bucketing en Apache Spark", y "Por qué debería preocuparse por el diseño de datos en el sistema de archivos".

Almacenamiento en caché y persistencia de datos

¿Cuál es la diferencia entre almacenamiento en caché y persistencia? En Spark son sinónimos. Dos llamadas API, `cache()` y `persist()`, ofrecen estas capacidades. Este último proporciona más control sobre cómo y dónde se almacenan sus datos: en la memoria y en el disco, serializados y no serializados. Ambos contribuyen a mejorar el rendimiento de las tramas de datos o las tablas a las que se accede con frecuencia.

Marco de datos.cache()

`cache()` almacenará tantas particiones leídas en la memoria entre los ejecutores de Spark como la memoria lo permita (consulte la [Figura 7-2](#)). Si bien un DataFrame puede almacenarse en caché de manera fraccionada, las particiones no pueden almacenarse en caché de manera fraccionada (por ejemplo, si tiene 8 particiones pero solo caben 4,5 particiones en la memoria, solo 4 se almacenarán en caché). Sin embargo, si no todas sus particiones están almacenadas en caché, cuando desee acceder a los datos nuevamente, las particiones que no están almacenadas en caché deberán volver a calcularse, lo que ralentizará su trabajo de Spark.

Veamos un ejemplo de cómo el almacenamiento en caché de un DataFrame grande mejora el rendimiento al acceder a un DataFrame:

```
// En Scala //
Crear un DataFrame con 10 millones de registros
val df = spark.range(1 * 10000000).toDF("id").withColumn("square", $"id" * $"id") df.cache () // Almacenar en
caché los datos df.count() // Materializar el caché

res3: Long = 10000000 El
comando tardó 5,11 segundos

df.count() // Ahora obtén el caché res4: Long
= 10000000 El comando tomó 0.44 segundos
```

El primer `count()` materializa el caché, mientras que el segundo accede al caché, lo que resulta en un tiempo de acceso casi 12 veces más rápido para este conjunto de datos.



Cuando usa `cache()` o `persist()`, el DataFrame no se almacena completamente en caché hasta que invoca una acción que pasa por cada registro (por ejemplo, `count()`). Si usa una acción como `take(1)`, solo se almacenará en caché una partición porque Catalyst se da cuenta de que no necesita calcular todas las particiones solo para recuperar un registro.

Al observar cómo se almacena un DataFrame en un ejecutor en un host local, como se muestra en la [Figura 7-4](#), podemos ver que todos caben en la memoria (recuerde que, en un nivel bajo, los DataFrame están respaldados por RDD).

The screenshot shows the Apache Spark 3.0.0-preview2 Storage UI. At the top, there are tabs for Jobs, Stages, Storage (which is selected), Environment, Executors, and SQL. To the right of the tabs, it says "Spark shell application UI".

RDD Storage Info for *(1) Project [id#0L, (id#0L * id#0L) AS square#4L] +- *(1) Range (0, 10000000, step=1, splits=12)

Storage Level: Disk Memory Deserialized 1x Replicated
 Cached Partitions: 12
 Total Partitions: 12
 Memory Size: 86.2 MiB
 Disk Size: 0.0 B

Data Distribution on 1 Executors

Host	On Heap Memory Usage	Off Heap Memory Usage	Disk Usage
10.0.1.5:60233	86.2 MiB (280.1 MiB Remaining)	0.0 B (0.0 B Remaining)	0.0 B

12 Partitions

Page: 1 | 1 Pages. Jump to 1 . Show 100 items in a page.

Block Name ▲	Storage Level	Size in Memory	Size on Disk	Executors
rdd_3_0	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_1	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_10	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_11	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_2	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_3	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_4	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_5	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_6	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_7	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_8	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233
rdd_3_9	Memory Deserialized 1x Replicated	7.2 MiB	0.0 B	10.0.1.5:60233

Figura 7-4. Caché distribuido en 12 particiones en la memoria del ejecutor

DataFrame.persist()

`persist(StorageLevel.LEVEL)` tiene matices, lo que brinda control sobre cómo se almacenan en caché sus datos a través de `StorageLevel`. La Tabla 7-2 resume los diferentes niveles de almacenamiento. Los datos en el disco siempre se serializan utilizando la serialización `Java o Kryo`.

Tabla 7-2. niveles de almacenamiento

Nivel de almacenamiento	Descripción
SOLO_MEMORIA	datos se almacenan directamente como objetos y se almacenan solo en la memoria.
MEMORY_ONLY_SER	Los datos se serializan como una representación de matriz de bytes compacta y se almacenan solo en la memoria. Para usarlo, tiene que ser deserializado a un costo.
MEMORIA_Y_DISCO	Los datos se almacenan directamente como objetos en la memoria, pero si no hay suficiente memoria, el resto se serializa y se almacena en el disco.
DISK_ONLY	Los datos se serializan y almacenan en disco.
OFF_HEAP	Los datos se almacenan fuera del montón. La memoria fuera del montón se usa en Spark para el almacenamiento y la ejecución de consultas; consulte "Configuración de la memoria de los ejecutores de Spark y el servicio de reproducción aleatoria" en la página 178.
MEMORY_AND_DISK_SER	Como MEMORY_AND_DISK, pero los datos se serializan cuando se almacenan en la memoria. (Los datos siempre son serializado cuando se almacena en el disco).



Cada StorageLevel (excepto OFF_HEAP) tiene un LEVEL_NAME_2 equivalente, lo que significa replicar dos veces en dos ejecutores de Spark diferentes: MEMORY_ONLY_2, MEMORY_AND_DISK_SER_2, etc. opción para programar una tarea local a una copia de los datos.

Veamos el mismo ejemplo que en la sección anterior, pero usando el método persist() :

```
// En Scala
import org.apache.spark.storage.StorageLevel

// Crear un DataFrame con 10 millones de registros val df =
spark.range(1 * 10000000).toDF("id").withColumn("square", $"id" * $"id") df.persist(StorageLevel.DISK_ONLY) // Serializar los datos
y almacenarlos en caché en el disco df.count() // Materializar el caché

res2: Long = 10000000 El
comando tardó 2,08 segundos

df.count() // Ahora obténalo del caché res3: Long = 10000000
El comando tomó 0.38 segundos
```

Como puede ver en la Figura 7-5, los datos se conservan en el disco, no en la memoria. Para eliminar la persistencia de sus datos almacenados en caché, simplemente llame a DataFrame.unpersist().

The screenshot shows the Apache Spark 3.0.0-preview2 interface. The top navigation bar includes links for Jobs, Stages, Storage, Environment, Executors, and SQL, with Storage being the active tab. A sub-header below the navigation bar reads "RDD Storage Info for *(1) Project [id#0L, (id#0L * id#0L) AS square#4L] +- *(1) Range (0, 10000000, step=1, splits=12)". Below this, detailed storage statistics are listed: Storage Level: Disk Serialized 1x Replicated; Cached Partitions: 12; Total Partitions: 12; Memory Size: 0.0 B; Disk Size: 86.0 MiB. A section titled "Data Distribution on 1 Executors" contains a table showing memory usage per host. The table has four columns: Host, On Heap Memory Usage, Off Heap Memory Usage, and Disk Usage. One row is present for the host 10.0.1.5:60655, showing 0.0 B (366.3 MiB Remaining) for On Heap and 0.0 B (0.0 B Remaining) for Off Heap, with a Disk Usage of 86.0 MiB. A section titled "12 Partitions" displays a table of 12 data blocks. The table has five columns: Block Name, Storage Level, Size in Memory, Size on Disk, and Executors. All 12 blocks are named rdd_3_x where x is a digit from 0 to 9. They all have a Storage Level of "Disk Serialized 1x Replicated", a Size in Memory of 0.0 B, a Size on Disk of 7.2 MiB, and are assigned to the Executor 10.0.1.5:60655.

Host	On Heap Memory Usage	Off Heap Memory Usage	Disk Usage
10.0.1.5:60655	0.0 B (366.3 MiB Remaining)	0.0 B (0.0 B Remaining)	86.0 MiB

Block Name	Storage Level	Size in Memory	Size on Disk	Executors
rdd_3_0	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_1	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_10	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_11	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_2	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_3	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_4	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_5	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_6	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_7	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_8	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655
rdd_3_9	Disk Serialized 1x Replicated	0.0 B	7.2 MiB	10.0.1.5:60655

Figura 7-5. Caché distribuido en 12 particiones en disco ejecutor

Finalmente, no solo puede almacenar en caché DataFrames, sino que también puede almacenar en caché las tablas o vistas derivadas de DataFrames. Esto les da nombres más legibles en la interfaz de usuario de Spark. Por ejemplo:

```
// En Scala
df.createOrReplaceTempView("dfTable")
spark.sql("CACHE TABLE dfTable")
spark.sql("SELECT count(*) FROM dfTable").show()
```

```
+-----+
|cuenta(1)|
+-----+
|10000000|
+-----+
```

El comando tomó 0.56 segundos

Cuándo almacenar en caché y persistir

Los casos de uso comunes para el almacenamiento en caché son escenarios en los que querrá acceder a un gran conjunto de datos repetidamente para realizar consultas o transformaciones. Algunos ejemplos incluyen:

- DataFrames comúnmente utilizados durante la capacitación de aprendizaje automático iterativo • DataFrames a los que se accede comúnmente para realizar transformaciones frecuentes durante ETL o construir canalizaciones de datos

Cuándo no almacenar en caché y persistir

No todos los casos de uso dictan la necesidad de almacenar en caché. Algunos escenarios que pueden no garantizar el almacenamiento en caché de sus DataFrames incluyen:

- DataFrames que son demasiado grandes para caber en la memoria • Una transformación económica en un DataFrame que no requiere un uso frecuente, independientemente del tamaño

Como regla general, debe usar el almacenamiento en caché de memoria con prudencia, ya que puede incurrir en costos de recursos al serializar y deserializar, según el nivel de almacenamiento utilizado.

A continuación, cambiaremos nuestro enfoque para analizar un par de operaciones comunes de unión de Spark que desencadenan un costoso movimiento de datos, lo que exige recursos informáticos y de red del clúster, y cómo podemos aliviar este movimiento mediante la organización de los datos.

Se une una familia de chispas

Las operaciones de unión son un tipo común de transformación en el análisis de big data en el que dos conjuntos de datos, en forma de tablas o marcos de datos, se fusionan sobre una clave de coincidencia común. De forma similar a las bases de datos relacionales, las API Spark DataFrame y Dataset y Spark SQL ofrecen una serie de transformaciones de unión: uniones internas, uniones externas, uniones izquierdas, uniones derechas, etc. Todas estas operaciones desencadenan una gran cantidad de movimiento de datos entre los ejecutores de Spark.

En el corazón de estas transformaciones está cómo Spark calcula qué datos producir, qué claves y datos asociados escribir en el disco, y cómo transferir esas claves y datos a los nodos como parte de operaciones como `groupBy()`, `join()`, `agg ()`, ordenar por () y reducir por clave (). Este movimiento se conoce comúnmente como el shuffle.

Spark tiene [cinco estrategias de unión distintas](#) mediante el cual intercambia, mueve, ordena, agrupa y fusiona datos entre ejecutores: la combinación hash de transmisión (BHJ), la combinación hash aleatoria (SHJ), la combinación aleatoria de clasificación (SMJ), la combinación de bucle anidado de transmisión (BNLJ) y la combinación aleatoria -y unión de bucle anidado replicado (también conocida como unión de producto cartesiano). Nos centraremos solo en dos de estos aquí (BHJ y SMJ), porque son los más comunes que encontrará.

Unión hash de difusión

También conocida como unión solo del lado del mapa, la unión hash de transmisión se emplea cuando dos conjuntos de datos, uno pequeño (que cabe en la memoria del controlador y el ejecutor) y otro lo suficientemente grande como para evitar el movimiento, necesitan unirse sobre ciertas condiciones o columnas. Usando una **variable de transmisión** de Spark , el conjunto de datos más pequeño es transmitido por el controlador a todos los ejecutores de Spark, como se muestra en la [Figura 7-6](#), y posteriormente se une con el conjunto de datos más grande en cada ejecutor. Esta estrategia evita el gran intercambio.

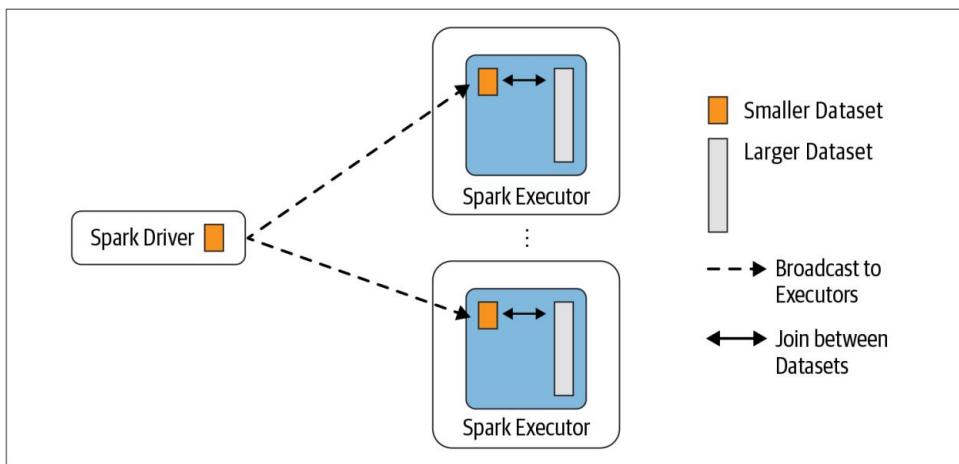


Figura 7-6. BHJ: el conjunto de datos más pequeño se transmite a todos los ejecutores

De forma predeterminada, Spark utilizará una unión de difusión si el conjunto de datos más pequeño tiene menos de 10 MB. Esta configuración se establece en `spark.sql.autoBroadcastJoinThreshold`; puede disminuir o aumentar el tamaño dependiendo de la cantidad de memoria que tenga en cada ejecutor y en el controlador. Si está seguro de que tiene suficiente memoria, puede usar una unión de transmisión con tramas de datos de más de 10 MB (incluso hasta 100 MB).

Un caso de uso común es cuando tiene un conjunto común de claves entre dos marcos de datos, uno con menos información que el otro, y necesita una vista combinada de ambos. Por ejemplo, considere un caso simple en el que tiene un gran conjunto de datos de jugadores de fútbol de todo el mundo, `playersDF`, y un conjunto de datos más pequeño de clubes de fútbol para los que juegan, `clubsDF`, y desea unirlos en una clave común:

```
// En Scala
import org.apache.spark.sql.functions.broadcast val
joinDF = playersDF.join(broadcast(clubsDF), "key1 === key2")
```



En este código, obligamos a Spark a realizar una unión de transmisión, pero recurrirá a este tipo de unión de forma predeterminada si el tamaño del conjunto de datos más pequeño está por debajo del umbral `spark.sql.autoBroadcastJoin`.

El BHJ es la unión más fácil y rápida que ofrece Spark, ya que no implica ninguna mezcla del conjunto de datos; todos los datos están disponibles localmente para el ejecutor después de una transmisión. Solo debe asegurarse de tener suficiente memoria tanto en el lado del controlador Spark como en el lado de los ejecutores para almacenar el conjunto de datos más pequeño en la memoria.

En cualquier momento después de la operación, puede ver en el plan físico qué operación de unión se realizó ejecutando:

se unió a DF.explicar (modo)

En Spark 3.0, puede usar `joinDF.explain('mode')` para mostrar un resultado legible y digerible. Los modos incluyen 'simple', 'extendido', 'codegen', 'costo' y 'formateado'.

Cuándo usar una combinación hash de transmisión

Utilice este tipo de unión bajo las siguientes condiciones para obtener el máximo beneficio:

- Cuando cada clave dentro de los conjuntos de datos más pequeños y más grandes se cifra en el mismo parámetro título de Spark
- Cuando un conjunto de datos es mucho más pequeño que el otro (y dentro de la configuración predeterminada de 10 MB, o más si tiene suficiente memoria)
- Cuando solo deseé realizar una unión equitativa, para combinar dos conjuntos de datos en función de las claves no ordenadas coincidentes. • Cuando no le preocupe el uso excesivo del ancho de banda de la red o los errores OOM, porque el conjunto de datos más pequeño se transmitirá a todos los ejecutores de Spark.

Especificar un valor de -1 en `spark.sql.autoBroadcastJoinThreshold` hará que Spark recurra siempre a una combinación de combinación de clasificación aleatoria, que analizamos a continuación.

Shuffle Sort Merge Join El

algoritmo sort-merge es una forma eficiente de combinar dos grandes conjuntos de datos en una clave común que se puede ordenar, única y se puede asignar o almacenar en la misma partición, es decir, dos datos conjuntos con una clave hashable común que terminan estando en la misma partición. Desde la perspectiva de Spark, esto significa que todas las filas dentro de cada conjunto de datos con la misma clave tienen un hash en la misma partición en el mismo ejecutor.

Obviamente, esto significa que los datos deben colocarse o intercambiarse entre ejecutores.

Como su nombre lo indica, este esquema de unión tiene dos fases: una fase de clasificación seguida de una fase de fusión. La fase de clasificación ordena cada conjunto de datos por su clave de combinación deseada; la fase de fusión itera sobre cada clave en la fila de cada conjunto de datos y fusiona las filas si las dos claves coinciden.

De forma predeterminada, SortMergeJoin está habilitado a través de spark.sql.join.preferSortMerge Join. Aquí hay un fragmento de código de un cuaderno de aplicaciones independientes disponibles para este capítulo en el repositorio de [GitHub del libro](#). La idea principal es tomar dos marcos de datos grandes, con un millón de registros, y unirlos en dos claves comunes, uid == id_usuarios.

Estos datos son sintéticos pero ilustran el punto:

```
// En Scala
importar scala.util.Random //
Mostrar preferencia sobre otras uniones para grandes conjuntos de datos //
Deshabilitar unión de difusión // Generar datos

...
spark.conf.set("spark.sql.autoBroadcastJoinThreshold", "-1")

// Genere algunos datos de muestra para dos conjuntos de
datos var state = scala.collection.mutable.Map[Int, String]() var items =
scala.collection.mutable.Map[Int, String]() val rnd = new scala.util.Random(42)

// Inicializar estados y artículos comprados estados ++
(0 -> "AZ", 1 -> "CO", 2-> "CA", 3-> "TX", 4 -> "NY", 5-> " MI") artículos += (0 -> "SKU-0", 1 ->
"SKU-1", 2-> "SKU-2", 3-> "SKU-3", 4 -> "SKU-4",
5-> "SKU-5")

// Crear tramas de datos
val usersDF = (0 a 1000000).map(id => (id, s"user_${id}", s"user_${id}
@databricks.com", state(rnd.nextInt(5 )))) .toDF("uid", "inicio de sesión",
"correo electrónico", "estado_usuario") val ordersDF = (0 a 1000000) .map(r
=> (r, r, rnd.nextInt(10000), 10 * r* 0.2d, estados(rnd.nextInt(5)),
items(rnd.nextInt(5)))) .toDF("transaction_id", "cantidad", "users_id",
"cantidad", "estado", "artículos")

// Hacer la unión
val usersOrdersDF = ordersDF.join(usersDF, $"users_id" === $"uid")

// Mostrar los resultados combinados
usersOrdersDF.show(false)

+-----+-----+-----+-----+
|id_transacción|cantidad|id_usuarios|cantidad |estado|elementos|uid|...|estado_usuario|
+-----+-----+-----+-----+
|3916 |          |3916 |148 |36384      | 7832.0 | CA | SKU-1 | 148 | ... | CO |
|36384 |          |148           |72768.0 | NY | SKU-2 | 148 | ... | CO
+-----+-----+-----+-----+
```

41839	41 839 148	83678.0 CA SKU-3 148 ... CO	
48212	48 212 148 48	96424.0 CA SKU-4 148 ... CO	
48484	484 148 50	96968.0 TX SKU-3 148 ... CO	
50514	514 148 65	101028.0 CO SKU-0 148 ... CO	
65694	694 148 65	131388.0 TX SKU-4 148 ... CO	
65723	723 148 93	131446.0 CA SKU-1 148 ... CO	
93125	125 148 107	186250.0 NY SKU-3 148 ... CO	
107097	097 148 148	214194.0 TX SKU-2 148 ... CO	
111297	289245 148	222594.0 AZ SKU-3 148 ... CO	
117195	314077 148	234390.0 TX SKU-4 148 ... CO	
253407	322170 148	506814.0 NY SKU-4 148 ... CO	
267180	344627 148	534360.0 AZ SKU-0 148 ... CO	
283187	345611 148	566374.0 AZ SKU-3 148 ... CO	
289245		578490.0 AZ SKU-0 148 ... CO	
314077		628154.0 CO SKU-3 148 ... CO	
322170		644340.0 TX SKU-3 148 ... CO	
344627		689254.0 NY SKU-3 148 ... CO	
		691222.0 TX SKU-3 148 ... CO	

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

solo se muestran las 20 filas superiores

Al examinar nuestro plan de ejecución final, notamos que Spark empleó un SortMergeJoin, como era de esperar, para unir los dos DataFrames. La operación de intercambio es el barajado de los resultados de la operación del mapa en cada ejecutor:

```
usuariosPedidosDF.explain()

== Plano físico ==
InMemoryTableScan [transaction_id#40, cantidad#41, users_id#42, cantidad#43,
estado#44, elementos#45, uid#13, inicio de sesión#14, correo electrónico#15, estado_usuario#16]
    +- InMemoryRelation [id_transacción#40, cantidad#41, id_usuarios#42, cantidad#43,
estado#44, elementos#45, uid#13, inicio de sesión#14, correo electrónico#15, estado_usuario#16],
StorageLevel (disco, memoria, deserializado, 1 réplica)
    +- *(3) SortMergeJoin [users_id#42], [uid#13], Interno
        :- *(1) Ordenar [users_id#42 ASC NULLS FIRST], falso, 0
        :  +- Intercambio de partición hash (users_id#42, 16), verdadero, [id=#56]
        :      +- LocalTableScan [transaction_id#40, cantidad#41, users_id#42,
cantidad#43, estado#44, artículos#45]
        +- *(2) Ordenar [uid#13 ASC NULLS PRIMERO], falso, 0
            +- Intercambiar partición hash (uid#13, 16), verdadero, [id=#57]
                +- LocalTableScan [uid#13, inicio de sesión#14, correo electrónico#15, estado_usuario#16]
```

Además, la interfaz de usuario de Spark (que analizaremos en la siguiente sección) muestra tres etapas para todo el trabajo: las operaciones Exchange y Sort ocurren en la etapa final, seguidas de la fusión de los resultados, como se muestra en las Figuras 7-7 y 7-8. El intercambio es costoso y requiere que las particiones se mezclen a través de la red entre ejecutores

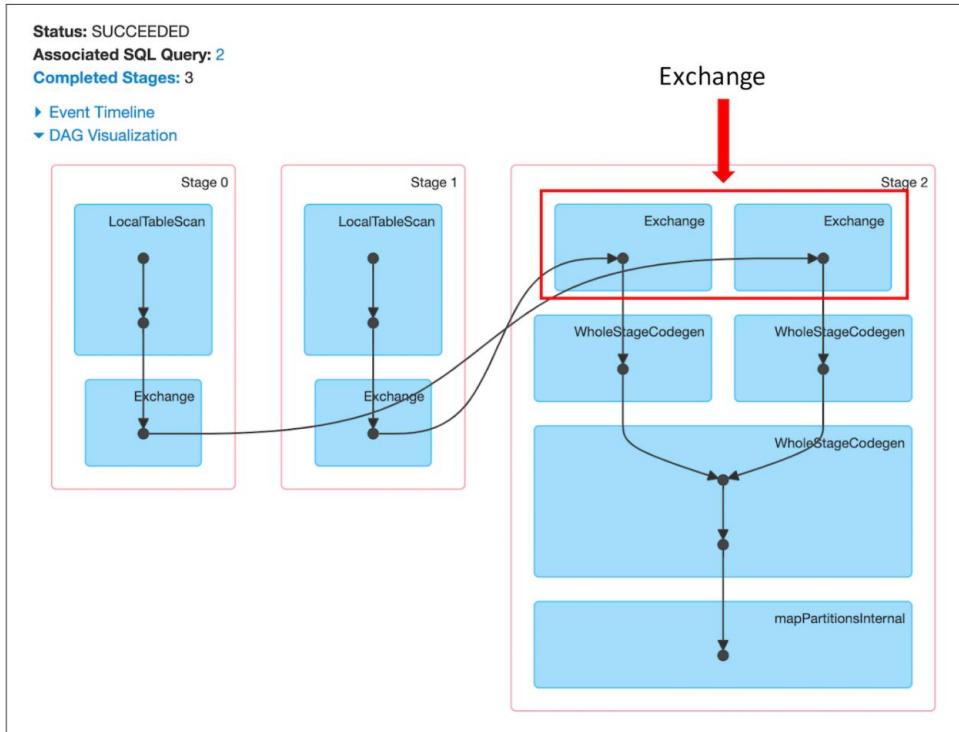


Figura 7-7. Antes del baldeo: etapas del Spark

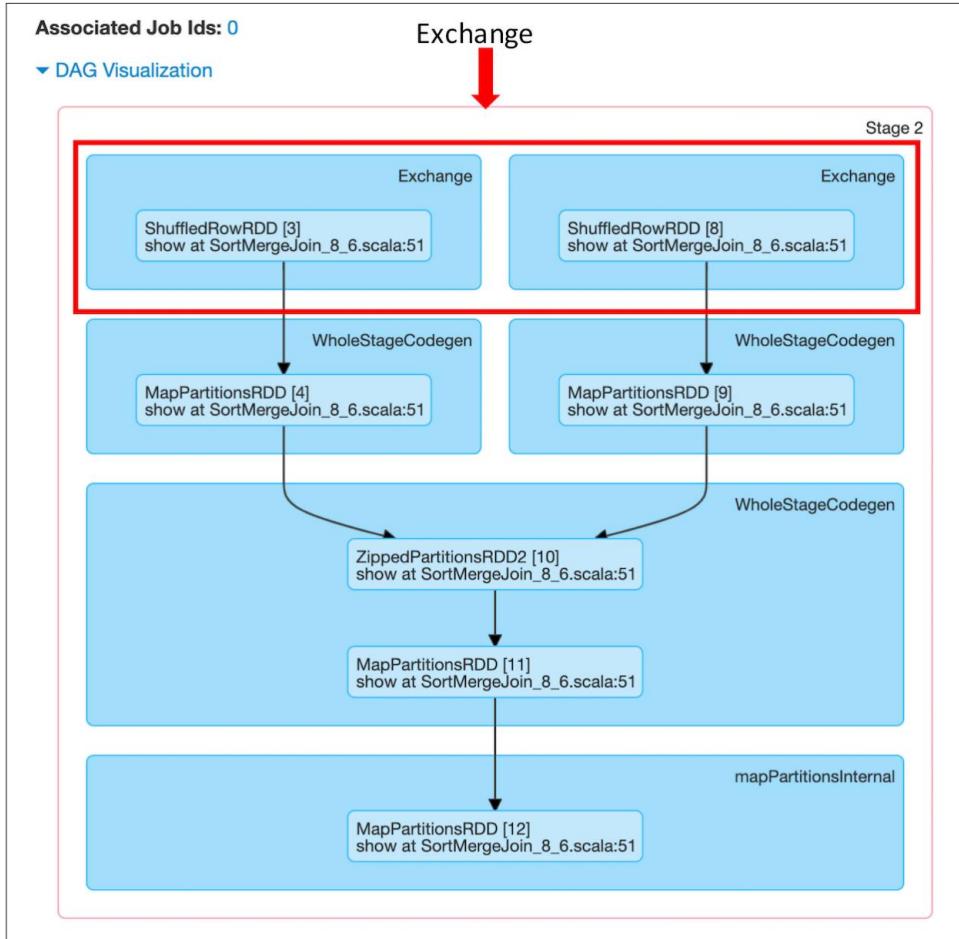


Figura 7-8. Antes del depósito: se requiere intercambio

Optimización de la combinación de

combinación de clasificación aleatoria Podemos eliminar el paso de intercambio de este esquema si creamos cubos particionados para columnas o claves ordenadas comunes en las que queremos realizar combinaciones de igualdad frecuentes. Es decir, podemos crear un número explícito de cubos para almacenar columnas ordenadas específicas (una clave por cubo). Preordenar y reorganizar los datos de esta manera aumenta el rendimiento, ya que nos permite omitir la costosa operación de intercambio e ir directamente a WholeStageCodegen.

En el siguiente fragmento de código del cuaderno de este capítulo (disponible en el repositorio de [GitHub del libro](#)) clasificamos y clasificamos por las columnas users_id y uid en las que nos uniremos, y guardamos los depósitos como tablas administradas por Spark en formato Parquet:

```

// En escala
importar org.apache.spark.sql.functions._
importar org.apache.spark.sql.SaveMode

// Guardar como tablas administradas agrupandolas en formato Parquet
usuariosDF.orderBy(asc("uid"))
    .write.format("parqué")
    .bucketBy(8, "uid")
    .mode(ModoGuardar.Sobrescribir)
    .saveAsTable("UsersTbl")

pedidosDF.orderBy(asc("users_id"))
    .write.format("parqué")
    .bucketBy(8, "users_id")
    .mode(ModoGuardar.Sobrescribir)
    .saveAsTable("OrdersTbl")

// Almacenar en caché las tablas
spark.sql(" TblUsuarios de la TABLA DE CACHE")
spark.sql("CACHE TABLE OrdersTbl")

// Leerlos de nuevo en
val userBucketDF = spark.table("UsersTbl")
val ordersBucketDF = spark.table("OrdersTbl")

// Haz la unión y muestra los resultados
val joinUsersOrdersBucketDF = ordersBucketDF
    .join(usersBucketDF, $"users_id" === "$uid")

joinUsersOrdersBucketDF.show (falso)

+-----+-----+-----+-----+-----+
|id_transacción|cantidad|id_usuarios|cantidad |estado|elementos|uid|...|estado_usuario|
+-----+-----+-----+-----+-----+
| 144179      | 144179 | 22 |      | 288358.0 | TX | SKU-4 | 22 | ... | CO | |
| 145352      | 145352 | 22 |      | 290704.0 | NY | SKU-0 | 22 | ... | CO |
| 168648      | 168648 | 22 |      | 337296.0 | TX | SKU-2 | 22 | ... | CO |
| 173682      | 173682 | 22 |      | 347364.0 | NY | SKU-2 | 22 | ... | CO |
| 397577      | 397577 | 22 |      | 795154.0 | CA | SKU-3 | 22 | ... | CO |
| 403974      | 403974 | 22 |      | 807948.0 | CO | SKU-2 | 22 | ... | CO |
| 405438      | 405438 | 22 |      | 810876.0 | NY | SKU-1 | 22 | ... | CO |
| 417886      | 417886 | 22 |      | 835772.0 | CA | SKU-3 | 22 | ... | CO |
| 420809      | 420809 | 22 |      | 841618.0 | NY | SKU-4 | 22 | ... | CO |
| 659905      | 22 | 916827 | 22 |      | 1319810.0 | AZ | SKU -1 | 22 | ... | CO |
| 899422      | 919106 | 22 |      | 1798844.0 | TX | SKU-4 | 22 | ... | CO |
| 906616      | 921921 | 22 |      | 1813232.0 | CO | SKU-2 | 22 | ... | CO |
| 916292      | 926777 | 22 |      | 1832584.0 | TX | SKU-0 | 22 | ... | CO |
|           | 124630 | 22 |      | 1833654.0 | TX | SKU-1 | 22 | ... | CO |
|           |        |     |      | 1838212.0 | TX | SKU-1 | 22 | ... | CO |
|           |        |     |      | 1843842.0 | AZ | SKU-4 | 22 | ... | CO |
|           |        |     |      | 1853554.0 | CO | SKU-2 | 22 | ... | CO |
|           |        |     |      | 249260.0 | CO | SKU-0 | 22 | ... | CO |
+-----+-----+-----+-----+-----+

```

129823	129823 22	259646.0 NY SKU-4 22 ... CO	
132756	132756 22	265512.0 AZ SKU-2 22 ... CO	
+-----+-----+-----+-----+			
solo se muestran las 20 filas superiores			

La salida unida está ordenada por uid y users_id, porque guardamos las tablas ordenadas en orden ascendente. Como tal, no hay necesidad de ordenar durante SortMergeJoin. Mirar En la interfaz de usuario de Spark ([Figura 7-9](#)), podemos ver que omitimos el intercambio y fuimos directamente a WholeStageCodegen.

El plan físico también muestra que no se realizó ningún intercambio , en comparación con el plan físico. plan antes de la cubeta:

```
unirUsuariosPedidosCuboDF.explain()

== Plano físico ==
*(3) SortMergeJoin [users_id#165], [uid#62], Interno
:- *(1) Ordenar [users_id#165 ASC NULLS FIRST], falso, 0
: +- *(1) El filtro no es nulo (users_id#165)
:     +- Escanear la tabla en memoria `OrdersTbl` [transaction_id#163, cantidad#164,
id_usuarios#165, cantidad#166, estado#167, artículos#168], [no es nulo(id_usuarios#165)]
:         +- InMemoryRelation [transaction_id#163, cantidad#164, users_id#165,
cantidad n.º 166, estado n.º 167, elementos n.º 168], nivel de almacenamiento (disco, memoria, deserializado, 1
rélicas)
:             +- *(1) De columna a fila
:                 +- Parquet FileScan
...
...
```

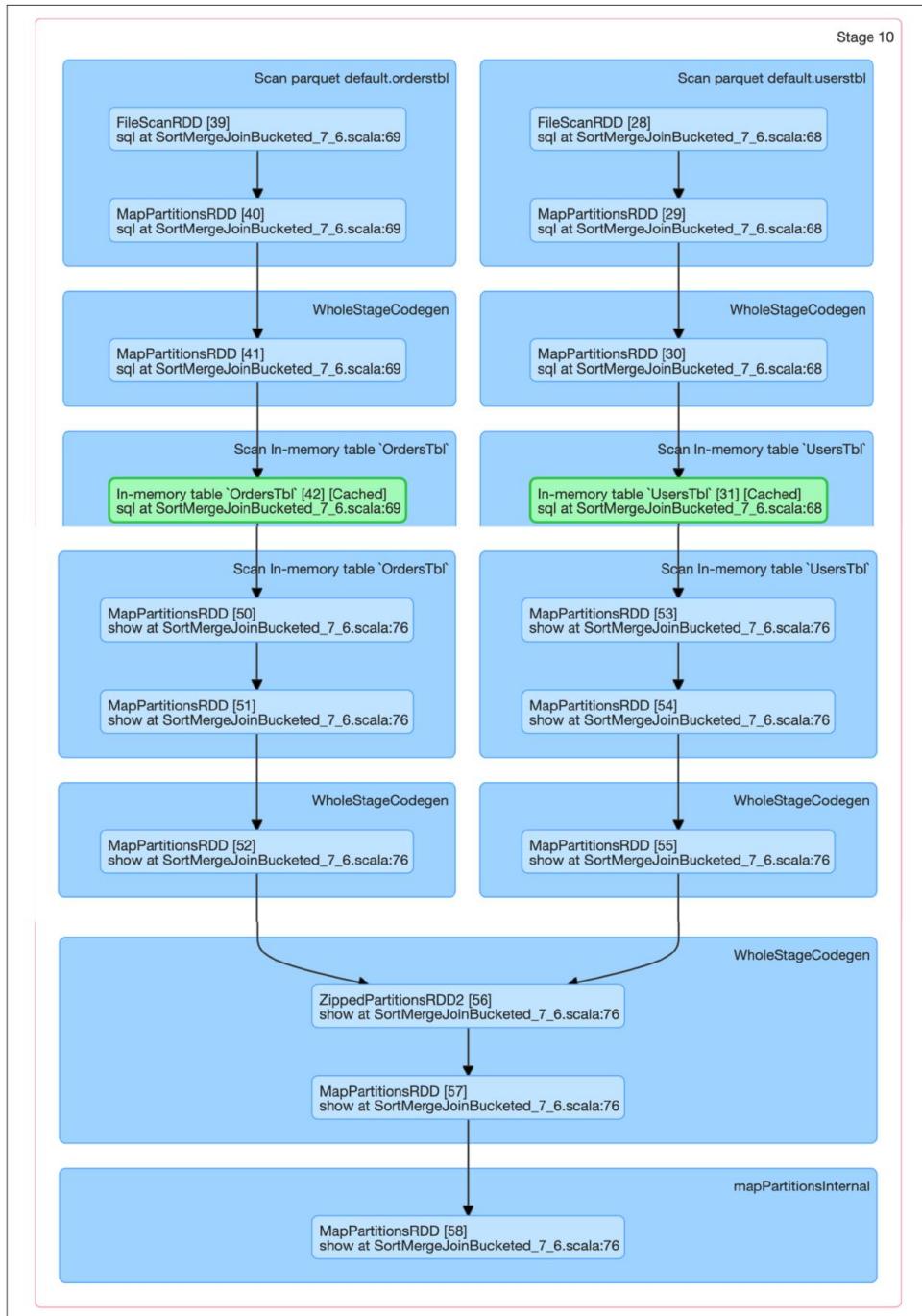


Figura 7-9. Después del agrupamiento: no se requiere intercambio

Cuándo usar una combinación de combinación de orden aleatorio

Utilice este tipo de unión bajo las siguientes condiciones para obtener el máximo beneficio:

- Cuando cada clave dentro de dos grandes conjuntos de datos se puede ordenar y cifrar en la misma partición mediante Spark. •

Cuando desea realizar solo uniones equitativas para combinar dos conjuntos de datos en función de

claves ordenadas coincidentes

- Cuando desee evitar las operaciones de intercambio y clasificación para guardar grandes cambios en la red

Hasta ahora, hemos cubierto aspectos operativos relacionados con el ajuste y la optimización de Spark, y cómo Spark intercambia datos durante dos operaciones comunes de combinación. También demostramos cómo puede aumentar el rendimiento de una operación de combinación de combinación de clasificación aleatoria mediante el uso de depósitos para evitar grandes intercambios de datos.

Como ha visto en las figuras anteriores, la interfaz de usuario de Spark es una forma útil de visualizar estas operaciones. Muestra las métricas recopiladas y el estado del programa, revelando una gran cantidad de información y pistas sobre posibles cuellos de botella en el rendimiento. En la sección final de este capítulo, analizamos qué buscar en la interfaz de usuario de Spark.

Inspección de la interfaz de usuario de Spark

Spark proporciona una interfaz de usuario web elaborada que nos permite inspeccionar varios componentes de nuestras aplicaciones. Ofrece detalles sobre el uso de la memoria, trabajos, etapas y tareas, así como líneas de tiempo de eventos, registros y varias métricas y estadísticas que pueden brindarle información sobre lo que sucede en sus aplicaciones Spark, tanto a nivel del controlador Spark como de manera individual, ejecutores individuales.

Un trabajo de envío de chispa iniciará la interfaz de usuario de Spark y podrá conectarse a ella en el host local (en modo local) o a través del controlador de Spark (en otros modos) en el puerto predeterminado. 4040.

Viaje a través de las pestañas de la interfaz de usuario de Spark

La interfaz de usuario de Spark tiene seis pestañas, como se muestra en la [Figura 7-10](#), cada una de las cuales brinda oportunidades para la exploración. Echemos un vistazo a lo que nos revela cada pestaña.



Figura 7-10. Pestañas de la IU de Spark

Esta discusión se aplica a Spark 2.x y Spark 3.0. Si bien gran parte de la interfaz de usuario es la misma en Spark 3.0, también agrega una séptima pestaña, Streaming estructurado. Esto se muestra en el [Capítulo 12](#).

Trabajos y Etapas

Como aprendió en el [Capítulo 2](#), Spark divide una aplicación en trabajos, etapas y tareas. Las pestañas Trabajos y Etapas le permiten navegar a través de estos y profundizar en un nivel granular para examinar los detalles de las tareas individuales. Puede ver su estado de finalización y revisar las métricas relacionadas con E/S, consumo de memoria, duración de la ejecución, etc.

[La Figura 7-11](#) muestra la pestaña Trabajos con la Línea de tiempo de eventos ampliada, que muestra cuándo se agregaron o eliminaron ejecutores del clúster. También proporciona una lista tabular de todos los trabajos completados en el clúster. La columna Duración indica el tiempo que tardó cada trabajo (identificado por el Id. de trabajo en la primera columna) en finalizar. Si este tiempo es alto, es una buena indicación de que es posible que desee investigar las etapas de ese trabajo para ver qué tareas pueden estar causando retrasos. Desde esta página de resumen también puede acceder a una página de detalles para cada trabajo, incluida una visualización de DAG y una lista de etapas completadas.

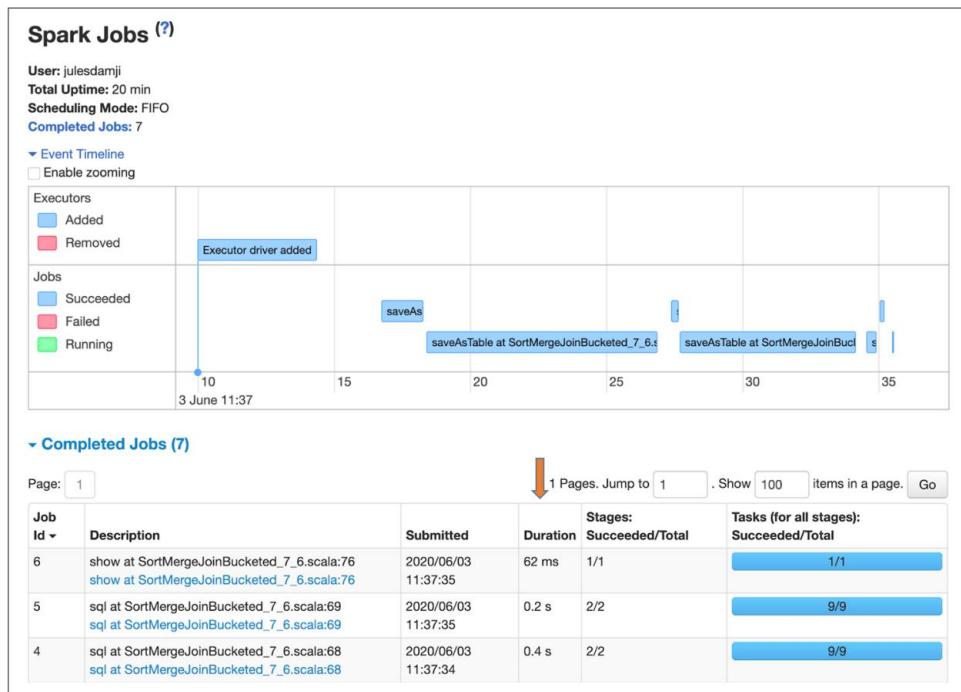


Figura 7-11. La pestaña Trabajos ofrece una vista de la línea de tiempo del evento y una lista de todos los trabajos completados

La pestaña Etapas proporciona un resumen del estado actual de todas las etapas de todos los trabajos en la aplicación. También puede acceder a una página de detalles para cada etapa, que proporciona un DAG y métricas sobre sus tareas (Figura 7-12). Además de otras estadísticas opcionales, puede ver la duración promedio de cada tarea, el tiempo dedicado a la recolección de elementos no utilizados (GC) y la cantidad de bytes/registros aleatorios leídos. Si los datos aleatorios se leen desde ejecutores remotos, un tiempo alto de bloqueo de lectura aleatoria puede indicar problemas de E/S. Un tiempo de GC alto indica que hay demasiados objetos en el montón (sus ejecutores pueden tener poca memoria). Si el tiempo de tarea máximo de una etapa es mucho mayor que la mediana, probablemente tenga datos sesgados causados por una distribución desigual de datos en sus particiones. Busque estos signos reveladores.

The screenshot shows the 'Stages' tab in the Spark UI. It includes three main sections:

- Summary Metrics for 1 Completed Tasks:** A table showing metrics for completed tasks. The columns are Metric, Min, 25th percentile, Median, 75th percentile, and Max. The data is as follows:

Metric	Min	25th percentile	Median	75th percentile	Max
Duration	48.0 ms	48.0 ms	48.0 ms	48.0 ms	48.0 ms
GC Time	0.0 ms	0.0 ms	0.0 ms	0.0 ms	0.0 ms
Input Size / Records	813.3 KiB / 4	813.3 KiB / 4	813.3 KiB / 4	813.3 KiB / 4	813.3 KiB / 4

 A note below says "Showing 1 to 3 of 3 entries".
- Aggregated Metrics by Executor:** A table showing metrics for executors. The columns are Executor ID, Logs, Address, Task Time, Total Tasks, Failed Tasks, Killed Tasks, Succeeded Tasks, Blacklisted, and Shuffle Read Size / Records. The data is as follows:

Executor ID	Logs	Address	Task Time	Total Tasks	Failed Tasks	Killed Tasks	Succeeded Tasks	Blacklisted	Shuffle Read Size / Records
driver		10.0.1.5:61630	0.2 s	1	0	0	1	false	253.8 KiB / 12411

 A note below says "Showing 1 to 1 of 1 entries". There are navigation buttons for 'Previous' (disabled), page 1, and 'Next'.
- Tasks (1):** A table showing task details. The columns are Index, Task ID, Attempt, Status, Locality level, Executor ID, Host, Logs, Launch Time, Duration, and GC Time. The data is as follows:

Index	Task ID	Attempt	Status	Locality level	Executor ID	Host	Logs	Launch Time	Duration	GC Time
0	24	0	SUCCESS	NODE_LOCAL	driver	10.0.1.5		2020-06-03 20:06:43	0.1 s	

 A note below says "Showing 1 to 1 of 1 entries". There are navigation buttons for 'Previous' (disabled), page 1, and 'Next'.

Figura 7-12. La pestaña Etapas proporciona detalles sobre las etapas y sus tareas.

También puede ver métricas agregadas para cada ejecutor y un desglose de las tareas individuales en esta página.

Ejecutores

La pestaña Ejecutores proporciona información sobre los ejecutores creados para la aplicación.

Como puede ver en la [Figura 7-13](#), puede profundizar en las minucias de los detalles sobre el uso de recursos (disco, memoria, núcleos), el tiempo dedicado al GC, la cantidad de datos escritos y leídos durante la reproducción aleatoria, etc.

Executors												
Show Additional Metrics												
Summary												
RDD ▲ Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input	Shuffle Read	Shuffle Write	Blacklisted
Active(1) 0	8.6 KiB / 366.3 MiB	0.0 B	12	0	0	25	25	13 s (3 s)	0.0 B	253.8 KiB	4 MiB	0
Dead(0) 0	0.0 B / 0.0 B	0.0 B	0	0	0	0	0	0.0 ms (0.0 ms)	0.0 B	0.0 B	0.0 B	0
Total(1) 0	8.6 KiB / 366.3 MiB	0.0 B	12	0	0	25	25	13 s (3 s)	0.0 B	253.8 KiB	4 MiB	0

Executors												
Show 20 ▾ entries Search: <input type="text"/>												
Executor ID	▲ Address	Status	RDD Blocks	Storage Memory	Disk Used	Cores	Active Tasks	Failed Tasks	Complete Tasks	Total Tasks	Task Time (GC Time)	Input
driver	10.0.1.5:61630	Active	0	8.6 KiB / 366.3 MiB	0.0 B	12	0	0	25	25	13 s (3 s)	0.0 B

Showing 1 to 1 of 1 entries Previous 1 Next

Figura 7-13. La pestaña Ejecutores muestra estadísticas granulares y métricas sobre los ejecutores utilizados por su aplicación Spark

Además de las estadísticas de resumen, puede ver cómo cada ejecutor individual usa la memoria y con qué propósito. Esto también ayuda a examinar el uso de recursos cuando ha utilizado el método `cache()` o `persist()` en un DataFrame o una tabla administrada, que analizamos a continuación.

Almacenamiento En el código Spark en "Shuffle Sort Merge Join" almacenamos en caché dos tablas administradas después del depósito. La pestaña Almacenamiento, que se muestra en la [Figura 7-14](#), brinda información sobre cualquier tabla o trama de datos almacenada en caché por la aplicación como resultado del método `cache()` o `persist()`.

▼ RDDs						
ID	RDD Name	Storage Level	Cached Partitions	Fraction Cached	Size in Memory	Size on Disk
2	LocalTableScan [uid#13, login#14, email#15, user_state#16]	Disk Serialized 1x Replicated	12	100%	0.0 B	4.4 MiB
16	LocalTableScan [transaction_id#40, quantity#41, users_id#42, amount#43, state#44, items#45]	Disk Serialized 1x Replicated	12	100%	0.0 B	1771.0 KiB
31	In-memory table 'UsersTbl'	Disk Memory Deserialized 1x Replicated	8	100%	4.4 MiB	0.0 B
42	In-memory table 'OrdersTbl'	Disk Memory Deserialized 1x Replicated	8	100%	2.0 MiB	0.0 B

Figura 7-14. La pestaña Almacenamiento muestra detalles sobre el uso de la memoria

Yendo un poco más allá al hacer clic en el enlace "Tabla en memoria `UsersTbl`" en la Figura 7-14 , se muestra cómo la tabla se almacena en caché en la memoria y en el disco en 1 ejecutor y 8 particiones; este número corresponde a la cantidad de cubos que creado para esta tabla (ver Figura 7-15).

RDD Storage Info for In-memory table 'UsersTbl'				
Storage Level: Disk Memory Deserialized 1x Replicated				
Cached Partitions: 8				
Total Partitions: 8				
Memory Size: 4.4 MiB				
Disk Size: 0.0 B				
Data Distribution on 1 Executors				
Host	On Heap Memory Usage	Off Heap Memory Usage	Disk Usage	
10.0.1.5:61782	4.4 MiB (359.8 MiB Remaining)	0.0 B (0.0 B Remaining)	0.0 B	
Page: 1	1 Pages. Jump to <input type="text" value="1"/> . Show <input type="text" value="100"/> items in a page.	<input type="button" value="Go"/>		
Block Name ▲	Storage Level	Size in Memory	Size on Disk	Executors
rdd_31_0	Memory Deserialized 1x Replicated	558.3 KiB	0.0 B	10.0.1.5:61782
rdd_31_1	Memory Deserialized 1x Replicated	556.3 KiB	0.0 B	10.0.1.5:61782
rdd_31_2	Memory Deserialized 1x Replicated	565.7 KiB	0.0 B	10.0.1.5:61782
rdd_31_3	Memory Deserialized 1x Replicated	560.5 KiB	0.0 B	10.0.1.5:61782
rdd_31_4	Memory Deserialized 1x Replicated	564.6 KiB	0.0 B	10.0.1.5:61782
rdd_31_5	Memory Deserialized 1x Replicated	564.3 KiB	0.0 B	10.0.1.5:61782
rdd_31_6	Memory Deserialized 1x Replicated	558.1 KiB	0.0 B	10.0.1.5:61782
rdd_31_7	Memory Deserialized 1x Replicated	557.6 KiB	0.0 B	10.0.1.5:61782

Figura 7-15. Spark UI que muestra la distribución de tablas en caché en la memoria del ejecutor

SQL Los efectos de las consultas Spark SQL que se ejecutan como parte de su aplicación Spark son rastreables y visibles a través de la pestaña SQL. Puede ver cuándo se ejecutaron las consultas y por qué trabajos, y su duración. Por ejemplo, en nuestro ejemplo SortMergeJoin ejecutamos algunas consultas; todos ellos se muestran en la [Figura 7-16](#), con enlaces para profundizar más.

SQL				
Completed Queries: 9				
▼ Completed Queries (9)				
Page: <input type="button" value="1"/>		1 Pages. Jump to <input type="button" value="1"/> . Show <input type="button" value="100"/> items in a page. <input type="button" value="Go"/>		
ID	Description	Submitted	Duration	Job IDs
8	show at SortMergeJoinBucketed_8_6.scala:69 +details	2020/02/27 18:44:34	0.2 s	[6]
7	sql at SortMergeJoinBucketed_8_6.scala:63 +details	2020/02/27 18:44:33	0.4 s	[5]
6	sql at SortMergeJoinBucketed_8_6.scala:63 +details	2020/02/27 18:44:33	0.4 s	
5	sql at SortMergeJoinBucketed_8_6.scala:62 +details	2020/02/27 18:44:33	0.7 s	[4]
4	sql at SortMergeJoinBucketed_8_6.scala:62 +details	2020/02/27 18:44:33	0.8 s	
3	saveAsTable at SortMergeJoinBucketed_8_6.scala:60	2020/02/27 18:44:31	1 s	[2][3]

Figura 7-16. La pestaña SQL muestra detalles sobre las consultas SQL completadas

Al hacer clic en la descripción de una consulta, se muestran los detalles del plan de ejecución con todos los operadores físicos, como se muestra en la [Figura 7-17](#). Debajo de cada operador físico del plan (aquí, la tabla Scan In-memory, HashAggregate y Exchange) se encuentran las métricas de SQL.

Estas métricas son útiles cuando queremos inspeccionar los detalles de un operador físico y descubrir qué ocurrió: cuántas filas se escanearon, cuántos bytes aleatorios se escribieron, etc.

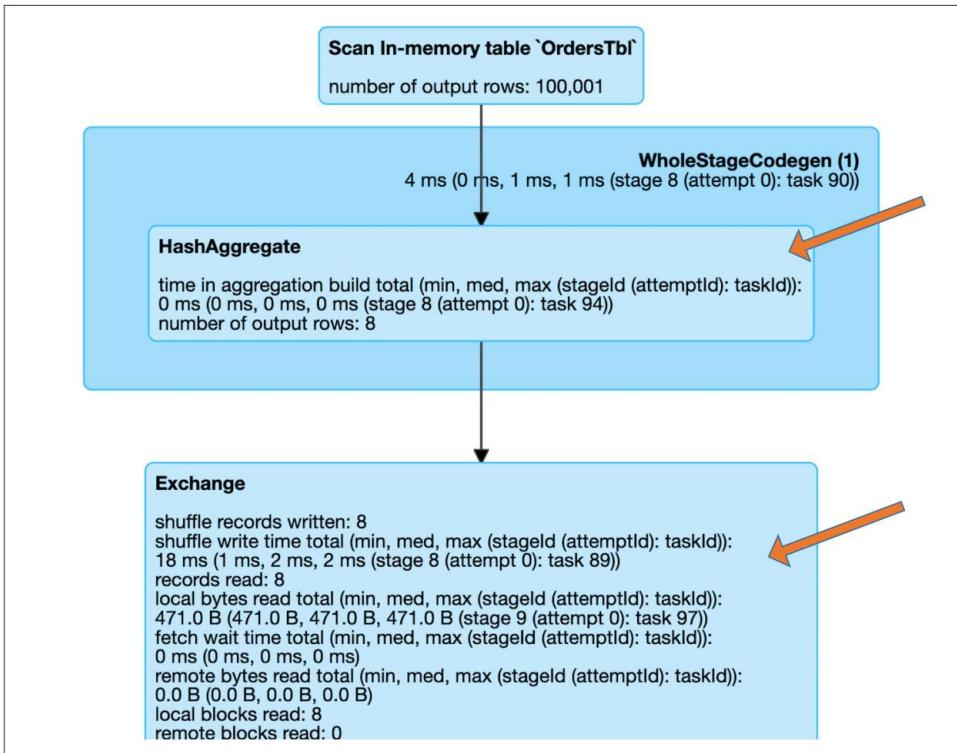


Figura 7-17. Spark UI que muestra estadísticas detalladas sobre una consulta SQL

Medioambiente

La pestaña Entorno, que se muestra en la [Figura 7-18](#), es tan importante como las demás. Conocer el entorno en el que se ejecuta su aplicación Spark revela muchas pistas que son útiles para la resolución de problemas. De hecho, es imperativo saber qué variables de entorno están configuradas, qué jars están incluidos, qué propiedades de Spark están configuradas (y sus respectivos valores, especialmente si modificó algunas de las configuraciones mencionadas en ["Optimización y ajuste de Spark para la eficiencia"](#) en página 173), qué propiedades del sistema están configuradas, qué entorno de tiempo de ejecución (como JVM o la versión de Java) se usa, etc. Todos estos detalles de solo lectura son una mina de oro de información que complementa sus esfuerzos de investigación si nota algún comportamiento anormal. en su aplicación Spark.

Environment

Runtime Information

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

Spark Properties

Name	Value
spark.app.id	local-1591215877337
spark.app.name	SortMergeJoinBucketed
spark.driver.host	10.0.1.5
spark.driver.port	61781
spark.executor.id	driver
spark.jars	file:/Users/julesdamji/gits/LearningSparkV2/chapter7/scala/jars/scala-chapter7_2.12-1.0.jar

Hadoop Properties

System Properties

Classpath Entries

Resource	Source
/Users/julesdamji/spark/spark-3.0.0-preview2-bin-hadoop2.7/conf/	System Classpath
/Users/julesdamji/spark/spark-3.0.0-preview2-bin-hadoop2.7/jars/HikariCP-2.5.1.jar	System Classpath
/Users/julesdamji/spark/spark-3.0.0-preview2-bin-hadoop2.7/jars/JLargeArrays-1.5.jar	System Classpath

Figura 7-18. La pestaña Entorno muestra las propiedades de tiempo de ejecución de su clúster de Spark

Depuración de aplicaciones

de Spark En esta sección, hemos navegado por las distintas pestañas de la interfaz de usuario de Spark. Como ha visto, la interfaz de usuario proporciona una gran cantidad de información que puede usar para depurar y solucionar problemas con sus aplicaciones Spark. Además de lo que hemos cubierto aquí, también proporciona acceso a los registros stdout/stderr del controlador y del ejecutor, donde puede haber registrado información de depuración.

La depuración a través de la interfaz de usuario es un proceso diferente a recorrer paso a paso una aplicación en su IDE favorito, más como investigar, seguir rastros de migas de pan, aunque

si prefiere ese enfoque, también puede depurar una aplicación Spark en un IDE como [IntelliJ IDEA](#) en un anfitrión local.

Las pestañas de la interfaz de usuario de Spark 3.0 revelan migas de pan perspicaces sobre lo que sucedió, junto con el acceso a los registros stdout/stderr del controlador y del ejecutor, donde podría haber registrado información de depuración.

Inicialmente, esta pléthora de información puede ser abrumadora para un novato. Pero con el tiempo comprenderá qué buscar en cada pestaña y podrá detectar y diagnosticar anomalías con mayor rapidez. Los patrones se aclararán y al visitar con frecuencia estas pestañas y familiarizarse con ellas después de ejecutar algunos ejemplos de Spark, se acostumbrará a ajustar e inspeccionar sus aplicaciones de Spark a través de la interfaz de usuario.

Resumen

En este capítulo, hemos discutido una serie de técnicas de optimización para ajustar sus aplicaciones Spark. Como vio, al ajustar algunas de las configuraciones predeterminadas de Spark, puede mejorar el escalado para grandes cargas de trabajo, mejorar el paralelismo y minimizar el hambre de memoria entre los ejecutores de Spark. También tuvo una idea de cómo puede usar el almacenamiento en caché y las estrategias persistentes con los niveles apropiados para acelerar el acceso a los conjuntos de datos que usa con frecuencia, y examinamos dos combinaciones de uso común que Spark emplea durante las agregaciones complejas y demostramos cómo agrupar los marcos de datos por claves ordenadas. , puede omitir costosas operaciones aleatorias.

Finalmente, para obtener una perspectiva visual del rendimiento, la interfaz de usuario de Spark completó la imagen. Aunque la interfaz de usuario es informativa y detallada, no es equivalente a la depuración por pasos en un IDE; sin embargo, le mostramos cómo puede convertirse en un detective de Spark al examinar y obtener información de las métricas y estadísticas, los datos de uso de memoria y cómputo y los seguimientos de ejecución de consultas SQL disponibles en la media docena de pestañas de la interfaz de usuario de Spark.

En el próximo capítulo, nos sumergiremos en la transmisión estructurada y le mostraremos cómo las API estructuradas que aprendió en los capítulos anteriores le permiten escribir aplicaciones de transmisión y por lotes de manera continua, lo que le permite generar datos confiables. lagos y oleoductos.

CAPÍTULO 8

Transmisión estructurada

En capítulos anteriores, aprendió a usar API estructuradas para procesar volúmenes de datos muy grandes pero finitos. Sin embargo, a menudo los datos llegan continuamente y deben procesarse en tiempo real. En este capítulo, discutiremos cómo las mismas API estructuradas también se pueden usar para procesar flujos de datos.

Evolución del motor de procesamiento de Apache Spark Stream

El procesamiento de flujo se define como el procesamiento continuo de un sinfín de flujos de datos. Con la llegada de Big Data, los sistemas de procesamiento de flujo pasaron de motores de procesamiento de un solo nodo a motores de procesamiento distribuido de múltiples nodos. Tradicionalmente, el procesamiento de flujo distribuido se ha implementado con un modelo de procesamiento de registro a la vez, como se ilustra en la Figura 8-1.

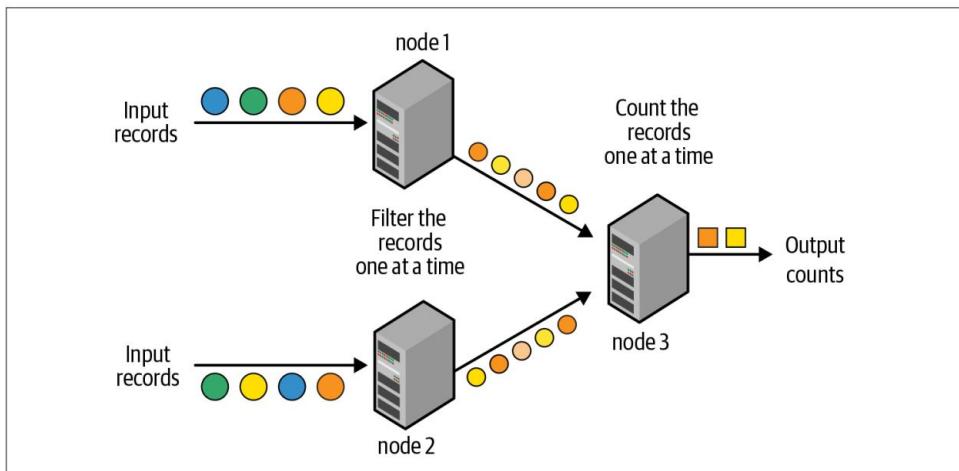


Figura 8-1. Modelo tradicional de procesamiento de registro a la vez

La canalización de procesamiento se compone de un gráfico dirigido de nodos, como se muestra en la Figura 8-1; cada nodo recibe continuamente un registro a la vez, lo procesa y luego envía los registros generados al siguiente nodo en el gráfico. Este modelo de procesamiento puede lograr latencias muy bajas, es decir, la canalización puede procesar un registro de entrada y la salida resultante se puede generar en milisegundos. Sin embargo, este modelo no es muy eficiente para recuperarse de fallas de nodos y nodos rezagados (es decir, nodos que son más lentos que otros); puede recuperarse de una falla muy rápido con una gran cantidad de recursos adicionales de comutación por error, o usar recursos adicionales mínimos pero recuperarse lentamente.¹

El advenimiento del procesamiento de flujo de microlotes

Este enfoque tradicional fue desafiado por Apache Spark cuando introdujo Spark Streaming (también llamado DStreams). Introdujo la idea del procesamiento de flujo de microlotes, donde el cómputo de flujo se modela como una serie continua de pequeños trabajos de procesamiento por lotes de estilo mapa/reducción (por lo tanto, "microlotes") en pequeños fragmentos de datos de flujo. . Esto se ilustra en la Figura 8-2.

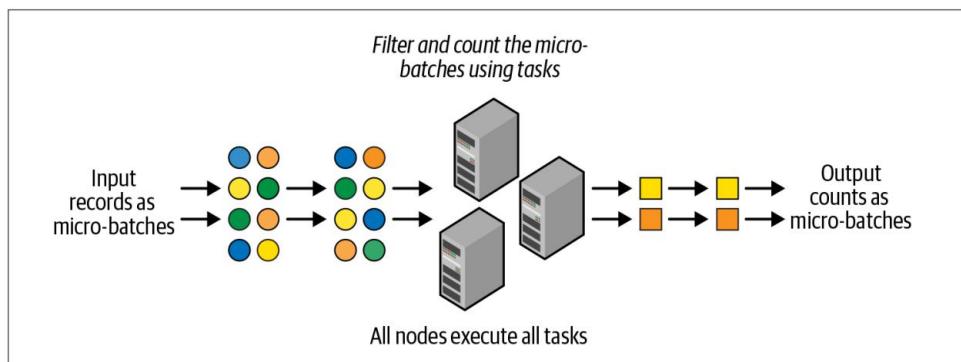


Figura 8-2. La transmisión estructurada utiliza un modelo de procesamiento de microlotes

Como se muestra aquí, Spark Streaming divide los datos del flujo de entrada en, por ejemplo, microlotes de 1 segundo. Cada lote se procesa en el clúster de Spark de forma distribuida con pequeñas tareas deterministas que generan el resultado en microlotes.

Dividir el cómputo de transmisión en estas pequeñas tareas nos brinda dos ventajas sobre el modelo tradicional de operador continuo:

¹ Para obtener una explicación más detallada, consulte el artículo de investigación original “[Discretized Streams: Fault-Tolerant Stream Processing at Scale](#)” por Matei Zaharia et al. (2013).

- La programación ágil de tareas de Spark puede recuperarse muy rápida y eficientemente de fallas y ejecutores rezagados al reprogramar una o más copias de las tareas en cualquiera de los otros ejecutores.
- La naturaleza determinista de las tareas asegura que los datos de salida sean los mismos sin importar cuántas veces se vuelva a ejecutar la tarea. Esta característica crucial permite que Spark Streaming brinde garantías de procesamiento exactamente una vez de extremo a extremo, es decir, los resultados de salida generados serán tales que cada registro de entrada se procesó exactamente una vez.

Esta tolerancia a fallas eficiente tiene el costo de la latencia: el modelo de microlotes no puede lograr latencias de nivel de milisegundos; suele conseguir latencias de unos pocos segundos (tan bajas como medio segundo en algunos casos). Sin embargo, hemos observado que para la gran mayoría de los casos de uso de procesamiento de flujo, los beneficios del procesamiento de microlotes superan el inconveniente de las latencias de segunda escala. Esto se debe a que la mayoría de las canalizaciones de transmisión tienen al menos una de las siguientes características:

- El pipeline no necesita latencias inferiores a unos pocos segundos. Por ejemplo, cuando la salida de transmisión solo va a ser leída por trabajos por hora, no es útil generar salida con latencias inferiores a un segundo.
- Hay mayores retrasos en otras partes del oleoducto. Por ejemplo, si las escrituras de un sensor en Apache Kafka (un sistema para ingerir flujos de datos) se procesan por lotes para lograr un mayor rendimiento, ninguna cantidad de optimización en los sistemas de procesamiento posteriores puede hacer que la latencia de un extremo a otro sea inferior a la del procesamiento por lotes. retrasos

Además, la API DStream se creó sobre la API RDD por lotes de Spark. Por lo tanto, DStreams tenía la misma semántica funcional y modelo de tolerancia a fallas que los RDD.

Spark Streaming demostró así que es posible que un único motor de procesamiento unificado proporcione API y semántica consistentes para cargas de trabajo por lotes, interactivas y de transmisión. Este cambio de paradigma fundamental en el procesamiento de transmisiones impulsó a Spark Streaming a convertirse en uno de los motores de procesamiento de transmisiones de código abierto más utilizados.

Lecciones aprendidas de Spark Streaming (DStreams)

A pesar de todas las ventajas, la API de DStream no estuvo exenta de fallas. Aquí hay algunas áreas clave de mejora que se identificaron:

Ausencia de una única API para el procesamiento por lotes y secuencias Aunque los DStreams y RDD tienen API consistentes (es decir, las mismas operaciones y la misma semántica), los desarrolladores aún tenían que reescribir explícitamente su código para usar diferentes clases al convertir sus trabajos por lotes en trabajos de transmisión.

Falta de separación entre los planes lógicos y físicos Spark Streaming ejecuta las operaciones de DStream en la misma secuencia en la que fueron especificadas por el desarrollador. Dado que los desarrolladores especifican efectivamente el plan físico exacto, no hay margen para las optimizaciones automáticas, y los desarrolladores tienen que optimizar manualmente su código para obtener el mejor rendimiento.

Falta de soporte nativo para ventanas de tiempo de eventos

DStreams define operaciones de ventana basadas solo en el momento en que Spark Streaming recibe cada registro (conocido como tiempo de procesamiento). Sin embargo, muchos casos de uso necesitan calcular agregados en ventana en función del momento en que se generaron los registros (conocido como tiempo del evento) en lugar de cuándo se recibieron o procesaron. La falta de soporte nativo de ventanas de tiempo de eventos dificultó que los desarrolladores construyeran dichas canalizaciones con Spark Streaming.

Estos inconvenientes dieron forma a la filosofía de diseño de Structured Streaming, que discutiremos a continuación.

La filosofía de la transmisión estructurada Basado

en estas lecciones de DStreams, la transmisión estructurada se diseñó desde cero con una filosofía central: para los desarrolladores, escribir canalizaciones de procesamiento de transmisión debería ser tan fácil como escribir canalizaciones por lotes. En pocas palabras, los principios rectores de la transmisión estructurada son:

Un único modelo de programación unificado y una interfaz para el procesamiento por lotes y secuencias
Este modelo unificado ofrece una interfaz de API simple para cargas de trabajo por lotes y de transmisión. Puede usar SQL familiar o consultas DataFrame similares a lotes (como las que aprendió en los capítulos anteriores) en su transmisión como lo haría en un lote, dejando de lidiar con las complejidades subyacentes de la tolerancia a fallas, las optimizaciones y las demoras. datos al motor. En las próximas secciones, examinaremos algunas de las consultas que podría escribir.

Una definición más amplia de procesamiento de flujo

Las aplicaciones de procesamiento de big data se han vuelto lo suficientemente complejas como para que la línea entre el procesamiento en tiempo real y el procesamiento por lotes se haya desdibujado significativamente. El objetivo de Structured Streaming era ampliar su aplicabilidad del procesamiento de flujo tradicional a una clase más amplia de aplicaciones; cualquier aplicación que procese datos de forma periódica (p. ej., cada pocas horas) a continua (como las aplicaciones de transmisión tradicionales) debe poder expresarse mediante transmisión estructurada.

A continuación, analizaremos el modelo de programación utilizado por Structured Streaming.

El modelo de programación de transmisión estructurada

"Tabla" es un concepto bien conocido con el que los desarrolladores están familiarizados cuando crean aplicaciones por lotes. La transmisión estructurada extiende este concepto a las aplicaciones de transmisión al tratar una transmisión como una tabla sin límites que se agrega continuamente, como se ilustra en la figura 8-3.

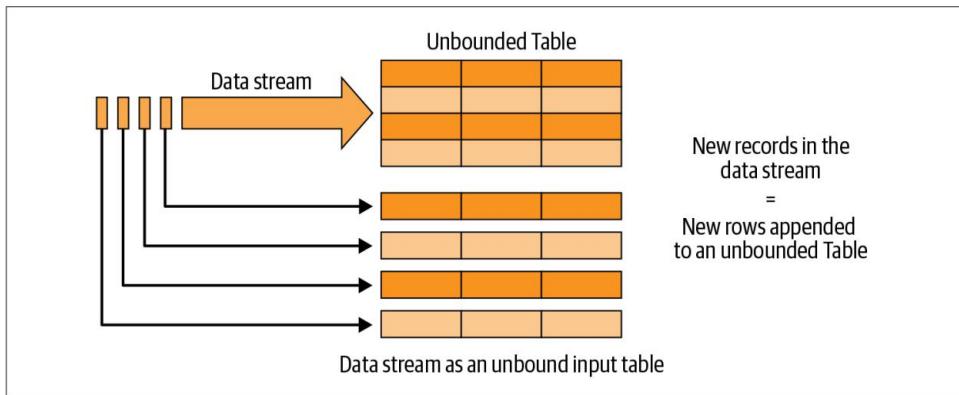


Figura 8-3. El modelo de programación de Flujo Estructurado: el flujo de datos como una tabla ilimitada

Cada nuevo registro recibido en el flujo de datos es como una nueva fila que se agrega a la tabla de entrada ilimitada. La transmisión estructurada en realidad no retendrá toda la entrada, pero la salida producida por la transmisión estructurada hasta el momento T será equivalente a tener toda la entrada hasta T en una tabla limitada y estática y ejecutar un trabajo por lotes en la tabla.

Como se muestra en la figura 8-4, el desarrollador define una consulta en esta tabla de entrada conceptual, como si fuera una tabla estática, para calcular la tabla de resultados que se escribirá en un sumidero de salida. La transmisión estructurada convertirá automáticamente esta consulta por lotes en un plan de ejecución de transmisión. Esto se denomina incrementalización: la transmisión estructurada determina qué estado debe mantenerse para actualizar el resultado cada vez que llega un registro. Finalmente, los desarrolladores especifican políticas de activación para controlar cuándo actualizar los resultados. Cada vez que se activa un activador, el flujo estructurado busca nuevos datos (es decir, una nueva fila en la tabla de entrada) y actualiza el resultado de forma incremental.

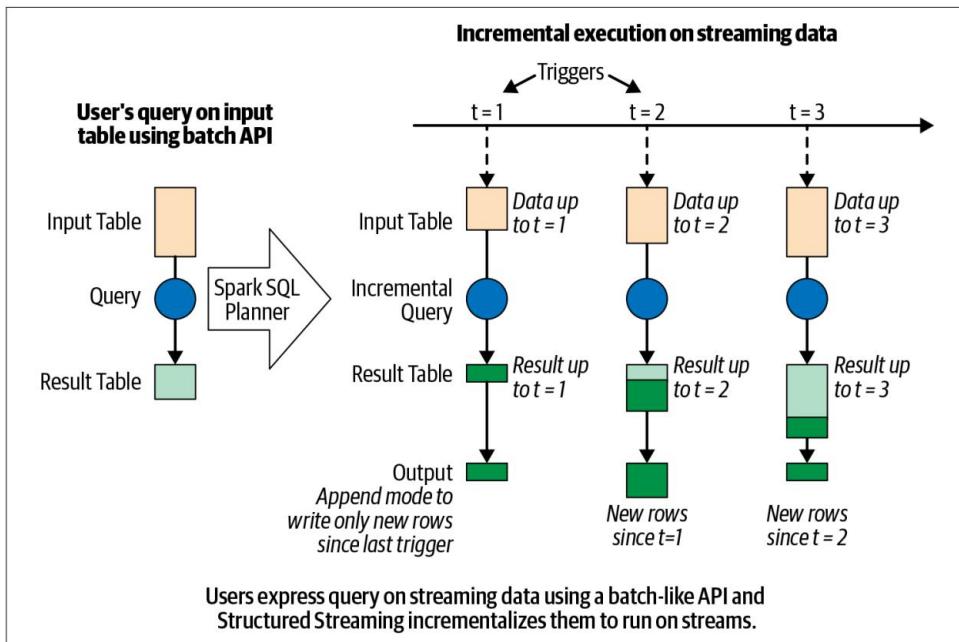


Figura 8-4. El modelo de procesamiento de transmisión estructurada

La última parte del modelo es el modo de salida. Cada vez que se actualice la tabla de resultados, el desarrollador querrá escribir las actualizaciones en un sistema externo, como un sistema de archivos (p. ej., HDFS, Amazon S3) o una base de datos (p. ej., MySQL, Cassandra). Por lo general, queremos escribir la salida de forma incremental. Para ello, Structured Streaming proporciona tres modos de salida:

Modo de adición

Solo las nuevas filas añadidas a la tabla de resultados desde el último activador se escribirán en el almacenamiento externo. Esto es aplicable solo en consultas donde las filas existentes en la tabla de resultados no pueden cambiar (por ejemplo, un mapa en un flujo de entrada).

Modo de actualización

Solo las filas que se actualizaron en la tabla de resultados desde el último activador se cambiarán en el almacenamiento externo. Este modo funciona para sumideros de salida que se pueden actualizar en el lugar, como una tabla MySQL.

Modo completo

Toda la tabla de resultados actualizada se escribirá en el almacenamiento externo.



A menos que se especifique el modo completo, la transmisión estructurada no materializará completamente la tabla de resultados. Solo se mantendrá la información suficiente (conocida como "estado") para garantizar que los cambios en la tabla de resultados se puedan calcular y se puedan generar las actualizaciones.

Pensar en los flujos de datos como tablas no solo facilita la conceptualización de los cálculos lógicos de los datos, sino que también facilita su expresión en código.

Dado que el DataFrame de Spark es una representación programática de una tabla, puede usar la API de DataFrame para expresar sus cálculos en la transmisión de datos. Todo lo que necesita hacer es definir un DataFrame de entrada (es decir, la tabla de entrada) desde una fuente de datos de transmisión y luego aplicar operaciones en el DataFrame de la misma manera que lo haría en un DataFrame definido en una fuente por lotes.

En la siguiente sección, verá lo fácil que es escribir consultas de Streaming Estructurado utilizando DataFrames.

Los fundamentos de una consulta de transmisión estructurada

En esta sección, cubriremos algunos conceptos de alto nivel que deberá comprender para desarrollar consultas de transmisión estructurada. Primero veremos los pasos clave para definir e iniciar una consulta de transmisión, luego analizaremos cómo monitorear la consulta activa y administrar su ciclo de vida.

Cinco pasos para definir una consulta de transmisión

Como se explicó en la sección anterior, la transmisión estructurada utiliza la misma API de DataFrame que las consultas por lotes para expresar la lógica de procesamiento de datos. Sin embargo, hay algunas diferencias clave que debe conocer para definir una consulta de transmisión estructurada. En esta sección, exploraremos los pasos necesarios para definir una consulta de transmisión mediante la creación de una consulta simple que lea secuencias de datos de texto en un socket y cuente las palabras.

Paso 1: Definir las fuentes de

entrada Al igual que con las consultas por lotes, el primer paso es definir un DataFrame desde una fuente de transmisión. Sin embargo, al leer fuentes de datos por lotes, necesitamos spark.read para crear un DataFrameReader, mientras que con las fuentes de transmisión necesitamos spark.readStream para crear un DataStreamReader. DataStreamReader tiene la mayoría de los mismos métodos que DataFrameReader, por lo que puede usarlo de manera similar. Aquí hay un ejemplo de cómo crear un DataFrame a partir de un flujo de datos de texto que se recibirá a través de una conexión de socket:

```
# En Python
chispa = SparkSession...
líneas =
(chispa .readStream.format("socket") .option("host", "localhost") .option("puerto", 9999) .load())

// En Scala
val chispa = SparkSession... val
líneas =
chispa .readStream.format("socket") .option("host", "localhost") .option("puerto", 9999) .load()
```

Este código genera las líneas DataFrame como una tabla ilimitada de datos de texto separados por saltos de línea que se leen desde localhost:9999. Tenga en cuenta que, al igual que las fuentes por lotes con spark.read, esto no comienza a leer inmediatamente los datos de transmisión; solo establece las configuraciones necesarias para leer los datos una vez que la consulta de transmisión se inicia explícitamente.

Además de los sockets, Apache Spark admite de forma nativa la lectura de flujos de datos de Apache Kafka y todos los diversos formatos basados en archivos que admite DataFrameReader (Parquet, ORC, JSON, etc.). Los detalles de estas fuentes y sus opciones admitidas se analizan más adelante en este capítulo. Además, una consulta de transmisión puede definir múltiples fuentes de entrada, tanto de transmisión como por lotes, que se pueden combinar mediante operaciones de DataFrame como uniones y uniones (también discutidas más adelante en este capítulo).

Paso 2: Transformar datos

Ahora podemos aplicar las operaciones habituales de DataFrame, como dividir las líneas en palabras individuales y luego contarlas, como se muestra en el siguiente código:

```
# En Python
desde pyspark.sql.functions import *
palabras = líneas.select(split(col("valor"), "\\s").alias("palabra")) recuentos =
palabras.agruparPor("palabra") .contar()

// En Scala
import org.apache.spark.sql.functions._ val
palabras = líneas.select(split(col("valor"), "\\s").as("palabra")) val cuenta =
palabras .groupBy("palabra").count()
```

counts es un DataFrame de transmisión (es decir, un DataFrame en datos de transmisión ilimitados) que representa los recuentos de palabras en ejecución que se calcularán una vez que se inicie la consulta de transmisión y los datos de entrada de transmisión se procesen continuamente.

Tenga en cuenta que estas operaciones para transformar las líneas que transmiten DataFrame funcionarían exactamente de la misma manera si las líneas fueran un DataFrame por lotes. En general, la mayoría de DataFrame

las operaciones que se pueden aplicar en un DataFrame por lotes también se pueden aplicar en un DataFrame de transmisión. Para comprender qué operaciones son compatibles con la transmisión estructurada, debe reconocer las dos clases generales de transformaciones de datos:

Transformaciones sin

estado Operaciones como select(), filter(), map(), etc. no requieren ninguna información de las filas anteriores para procesar la fila siguiente; cada fila se puede procesar por sí misma. La falta de “estado” previo en estas operaciones las convierte en apátridas. Las operaciones sin estado se pueden aplicar tanto a tramas de datos por lotes como de transmisión.

Transformaciones con

estado Por el contrario, una operación de agregación como count() requiere mantener el estado para combinar datos en varias filas. Más específicamente, cualquier operación de DataFrame que implique agrupar, unir o agregar son transformaciones con estado. Si bien muchas de estas operaciones son compatibles con la transmisión estructurada, algunas combinaciones de ellas no son compatibles porque es computacionalmente difícil o inviable calcularlas de manera incremental.

Las operaciones con estado admitidas por Flujo estructurado y cómo administrar su estado en tiempo de ejecución se analizan más adelante en este capítulo.

Paso 3: definir el receptor de salida y el

modo de salida Después de transformar los datos, podemos definir cómo escribir los datos de salida procesados con DataFrame.writeStream (en lugar de DataFrame.write, que se usa para datos por lotes). Esto crea un DataStreamWriter que, similar a DataFrameWriter, tiene métodos adicionales para especificar lo siguiente:

- Detalles de escritura de salida (dónde y cómo escribir la salida) •
- Detalles de procesamiento (cómo procesar datos y cómo recuperarse de fallas)

Comencemos con los detalles de escritura de salida (nos centraremos en los detalles de procesamiento en el siguiente paso). Por ejemplo, el siguiente fragmento muestra cómo escribir los recuentos finales en la consola:

```
# En el
escritor de Python = counts.writeStream.format("console").outputMode("complete")
// En Scala
val escritor = counts.writeStream.format("console").outputMode("complete")
```

Aquí hemos especificado "consola" como receptor de transmisión de salida y "completo" como modo de salida. El modo de salida de una consulta de transmisión especifica qué parte de la salida actualizada se debe escribir después de procesar nuevos datos de entrada. En este ejemplo, a medida que se procesa una parte de los nuevos datos de entrada y se actualizan los recuentos de palabras, podemos optar por imprimir en la consola los recuentos de todas las palabras vistas hasta ahora (que

es, modo completo), o solo aquellas palabras que se actualizaron en la última parte de los datos de entrada. Esto se decide por el modo de salida especificado, que puede ser uno de los siguientes (como ya vimos en "El modelo de programación de transmisión estructurada" en la página 211:

Modo de adición

Este es el modo predeterminado, donde solo las nuevas filas agregadas a la tabla de resultados/marco de datos (por ejemplo, la tabla de conteos) desde el último activador se enviarán al sumidero. Semánticamente, este modo garantiza que cualquier fila que se genere nunca será modificada o actualizada por la consulta en el futuro. Por lo tanto, el modo de adición solo es compatible con aquellas consultas (p. ej., consultas sin estado) que nunca modificarán los datos de salida anteriores. Por el contrario, nuestra consulta de recuento de palabras puede actualizar los recuentos generados previamente; por lo tanto, no admite el modo de adición.

Modo completo En

en este modo, todas las filas de la tabla de resultados/DataFrame se mostrarán al final de cada disparo. Esto es compatible con consultas en las que es probable que la tabla de resultados sea mucho más pequeña que los datos de entrada y, por lo tanto, es factible que se conserve en la memoria. Por ejemplo, nuestra consulta de recuento de palabras admite el modo completo porque es probable que los datos de recuento sean mucho más pequeños que los datos de entrada.

Modo de

actualización En este modo, solo las filas de la tabla de resultados/marco de datos que se actualizaron desde el último activador se mostrarán al final de cada activador. Esto contrasta con el modo de adición, ya que la consulta puede modificar las filas de salida y generarlas nuevamente en el futuro. La mayoría de las consultas admiten el modo de actualización.



Los detalles completos sobre los modos de salida admitidos por diferentes consultas se pueden encontrar en la última Guía de [programación de transmisión estructurada](#).

Además de escribir la salida en la consola, la transmisión estructurada admite de forma nativa escrituras de transmisión en archivos y Apache Kafka. Además, puede escribir en ubicaciones arbitrarias utilizando los métodos API `foreachBatch()` y `foreach()`. De hecho, puede usar `foreachBatch()` para escribir salidas de transmisión utilizando fuentes de datos por lotes existentes (pero perderá las garantías de exactamente una vez). Los detalles de estos sumideros y sus opciones admitidas se analizan más adelante en este capítulo.

Paso 4: especificar los detalles del

procesamiento El paso final antes de iniciar la consulta es especificar los detalles de cómo procesar los datos. Continuando con nuestro ejemplo de conteo de palabras, vamos a especificar los detalles de procesamiento de la siguiente manera:

```
# En Python
checkpointDir = "..."
escritor2 = (escritor
    .trigger(processingTime="1
    second") .option("checkpointLocation", checkpointDir))

// En Scala
importar org.apache.spark.sql.streaming._
val checkpointDir = "..." val escritor2 = escritor

    .trigger(Trigger.ProcessingTime("1
    segundo")) .option("checkpointLocation", checkpointDir)
```

Aquí hemos especificado dos tipos de detalles usando el DataStreamWriter que creamos con DataFrame.writeStream:

Detalles de

activación Esto indica cuándo activar el descubrimiento y el procesamiento de datos de transmisión recientemente disponibles. Hay cuatro opciones:

Predeterminado Cuando el activador no se especifica explícitamente, de forma predeterminada, la consulta de transmisión ejecuta datos en microlotes donde el siguiente microlote se activa tan pronto como se completa el microlote anterior.

Tiempo de procesamiento con intervalo de activación

Puede especificar explícitamente el activador ProcessingTime con un intervalo y la consulta activará microlotes en ese intervalo fijo.

Una vez

En este modo, la consulta de transmisión ejecutará exactamente un microlote: procesa todos los datos nuevos disponibles en un solo lote y luego se detiene. Esto es útil cuando desea controlar la activación y el procesamiento desde un programador externo que reiniciará la consulta utilizando cualquier programa personalizado (por ejemplo, para controlar el costo ejecutando una consulta solo una **vez al día**).

Continuo

Este es un modo experimental (a partir de Spark 3.0) en el que la consulta de transmisión procesará datos continuamente en lugar de microlotes. Si bien solo un pequeño subconjunto de operaciones de DataFrame permite usar este modo, puede proporcionar una latencia mucho más baja (tan baja como milisegundos) que los modos de activación de microlotes. Consulte la última **Guía de programación de transmisión estructurada** para obtener la información más actualizada.

Ubicación del punto de control

Este es un directorio en cualquier sistema de archivos compatible con HDFS donde una consulta de transmisión guarda su información de progreso, es decir, qué datos se han procesado con éxito. En caso de falla, estos metadatos se utilizan para reiniciar la consulta fallida exactamente donde la dejó. Por lo tanto, es necesario establecer esta opción para la recuperación de fallas con garantías de una sola vez.

Paso 5: Inicie la consulta

Una vez que se ha especificado todo, el paso final es iniciar la consulta, lo cual puedes hacer con lo siguiente:

```
# En Python
streamingQuery = escritor2.start()

// En Scala
val streamingQuery = escritor2.start()
```

El objeto devuelto de tipo streamingQuery representa una consulta activa y se puede usar para administrar la consulta, que trataremos más adelante en este capítulo.

Tenga en cuenta que start() es un método sin bloqueo, por lo que regresará tan pronto como la consulta haya comenzado en segundo plano. Si desea que el subprocesso principal se bloquee hasta que finalice la consulta de transmisión, puede usar streamingQuery.awaitTermination(). Si la consulta falla en segundo plano con un error, awaitTermination() también fallará con la misma excepción.

Puede esperar hasta un tiempo de espera con awaitTermination(timeoutMillis), y puede detener explícitamente la consulta con streamingQuery.stop().

Poniéndolo todo junto

Para resumir, aquí está el código completo para leer flujos de datos de texto en un socket, contar las palabras e imprimir los conteos en la consola:

```
# En Python
desde pyspark.sql.functions import *
chispa = SparkSession... líneas =
(chispa .readStream.format("socket") .option("host",
"localhost") .option("puerto", 9999) .carga())

palabras = líneas.select(split(col("valor"), "\\s").alias("palabra"))
cuentas =
palabras.groupBy("palabra").count()
checkpointDir = "..." streamingQuery =
(cuenta .writeStream .format("consola")
```

```

.outputMode("completo") .trigger(processingTime="1
segundo") .option("checkpointLocation",
checkpointDir) .start() streamingQuery.awaitTermination()

// En Scala
import org.apache.spark.sql.functions._
import org.apache.spark.sql.streaming._ val
spark = SparkSession... val lines =
spark .readStream.format("socket") .option
("host", "host local") .option("puerto",
9999) .load()

val palabras = líneas.select(split(col("valor"), "\\\s").as("palabra")) val cuenta =
palabras.groupBy("palabra").count()

val checkpointDir = "..." val
streamingQuery =
counts.writeStream .format("console") .outputMode("complete") .trigger(Trigger.ProcessingTime("1 segundo")) .option("ch

```

Una vez iniciada la consulta, un subproceso en segundo plano lee continuamente nuevos datos de la fuente de transmisión, los procesa y los escribe en los receptores de transmisión. A continuación, echemos un vistazo rápido debajo del capó de cómo se ejecuta esto.

Bajo el capó de una consulta de transmisión activa

Una vez que se inicia la consulta, la siguiente secuencia de pasos transcurre en el motor, como se muestra en la [Figura 8-5](#). Las operaciones de DataFrame se convierten en un plan lógico, que es una representación abstracta del cálculo que utiliza Spark SQL para planificar una consulta:

1. Spark SQL analiza y optimiza este plan lógico para garantizar que se pueda ejecutar de manera incremental y eficiente en la transmisión de datos.
2. Spark SQL inicia un subproceso en segundo plano que ejecuta continuamente lo siguiente

bucle: 2

2 Este bucle de ejecución se ejecuta para modos de activación basados en microlotes (es decir, ProcessingTime y Once), pero no para el modo de disparo continuo .

una. Según el intervalo de activación configurado, el subprocesso verifica las fuentes de transmisión para verificar la disponibilidad de nuevos datos. b. Si está disponible, los nuevos datos se ejecutan ejecutando un microlote. A partir del plan lógico optimizado, se genera un plan de ejecución de Spark optimizado que lee los nuevos datos del origen, calcula de forma incremental el resultado actualizado y escribe la salida en el receptor de acuerdo con el modo de salida configurado.

C. Para cada microlote, el rango exacto de datos procesados (p. ej., el conjunto de archivos o el rango de compensaciones de Apache Kafka) y cualquier estado asociado se guardan en la ubicación del punto de control configurado para que la consulta pueda reprocesar de manera determinista el rango exacto. si es necesario.

3. Este bucle continúa hasta que finaliza la consulta, lo que puede ocurrir por uno de los siguientes motivos: a. Ha ocurrido una falla en la consulta (ya sea un error de procesamiento o una falla en

el clúster).

b. La consulta se detiene explícitamente mediante streamingQuery.stop(). C. Si el activador se establece en Una vez, la consulta se detendrá por sí sola después de ejecutar un solo microlote que contenga todos los datos disponibles.

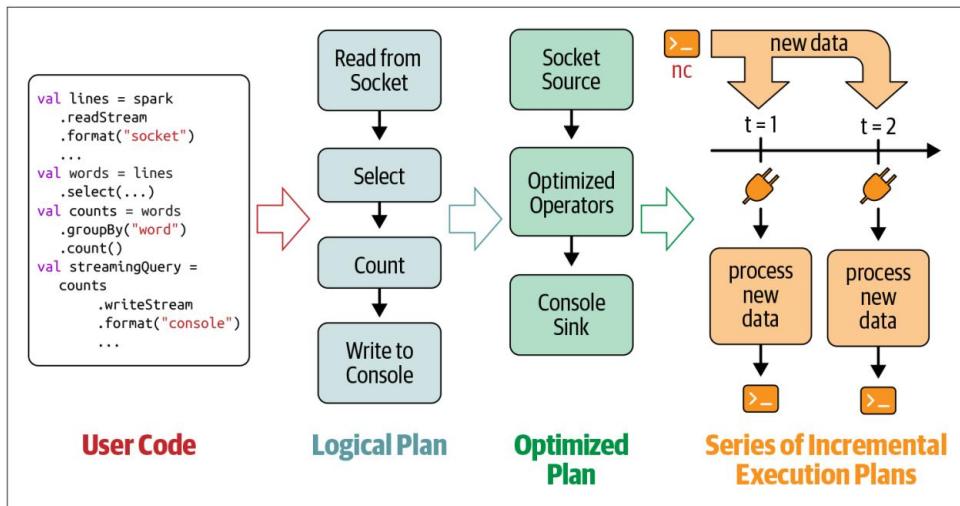


Figura 8-5. Ejecución incremental de consultas de transmisión



Un punto clave que debe recordar acerca de la transmisión estructurada es que debajo se usa Spark SQL para ejecutar los datos. Como tal, se utiliza toda la potencia del motor de ejecución hiperoptimizado de Spark SQL para maximizar el rendimiento del procesamiento de secuencias, lo que proporciona ventajas de rendimiento clave.

A continuación, analizaremos cómo reiniciar una consulta de transmisión después de la finalización y el ciclo de vida de una consulta de transmisión.

Recuperación de fallas con garantías de una sola vez

Para reiniciar una consulta finalizada en un proceso completamente nuevo, debe crear una nueva SparkSession, redefinir todos los marcos de datos e iniciar la consulta de transmisión en el resultado final utilizando la misma ubicación de punto de control que la utilizada cuando la consulta se inició por primera vez. Para nuestro ejemplo de conteo de palabras, simplemente puede volver a ejecutar el fragmento de código completo que se muestra anteriormente, desde la definición de chispa en la primera línea hasta el inicio final () en la última línea.

La ubicación del punto de control debe ser la misma en todos los reinicios porque este directorio contiene la identidad única de una consulta de transmisión y determina el ciclo de vida de la consulta. Si se elimina el directorio de puntos de control o se inicia la misma consulta con un directorio de puntos de control diferente, es como comenzar una nueva consulta desde cero. Específicamente, los puntos de control tienen información a nivel de registro (por ejemplo, compensaciones de Apache Kafka) para rastrear el rango de datos que estaba procesando el último microlote incompleto. La consulta reiniciada usará esta información para comenzar a procesar registros precisamente después del último microlote completado con éxito. Si la consulta anterior había planificado un microlote pero finalizó antes de completarse, la consulta reiniciada reprocesará el mismo rango de datos antes de procesar nuevos datos. Junto con la ejecución de tareas deterministas de Spark, la salida regenerada será la misma que se esperaba antes del reinicio.

La transmisión estructurada puede garantizar garantías de extremo a extremo exactamente una vez (es decir, la salida es como si cada registro de entrada se procesara exactamente una vez) cuando se cumplen las siguientes condiciones:

Fuentes de transmisión reproducibles

El rango de datos del último microlote incompleto se puede volver a leer desde la fuente.

Cálculos deterministas

Todas las transformaciones de datos producen de manera determinista el mismo resultado cuando se les dan los mismos datos de entrada.

Sumidero de transmisión

idempotente El sumidero puede identificar microlotes reejecutados e ignorar escrituras duplicadas que pueden ser causadas por reinicios.

Tenga en cuenta que nuestro ejemplo de conteo de palabras no proporciona garantías de una sola vez porque la fuente del socket no se puede reproducir y el sumidero de la consola no es idempotente.

Como nota final con respecto al reinicio de consultas, es posible realizar modificaciones menores a una consulta entre reinicios. Aquí hay algunas maneras en que puede modificar la consulta:

Transformaciones de tramas de datos

Puede realizar modificaciones menores a las transformaciones entre reinicios. Por ejemplo, en nuestro ejemplo de recuento de palabras de transmisión, si desea ignorar las líneas que tienen secuencias de bytes dañadas que pueden bloquear la consulta, puede agregar un filtro en la transformación:

```
# En Python
# isCorruptedUdf = udf para detectar corrupción en la cadena

filteredLines = lines.filter("isCorruptedUdf(value) = false") palabras =
filteredLines.select(split(col("value"), "\\s").alias("word"))

// En Scala //
val isCorruptedUdf = udf para detectar corrupción en la cadena

val filteredLines = lines.filter("isCorruptedUdf(value) = false") val palabras =
filteredLines.select(split(col("value"), "\\s").as("word"))
```

Al reiniciar con este DataFrame de palabras modificadas , la consulta reiniciada aplicará el filtro a todos los datos procesados desde el reinicio (incluido el último microlote incompleto), evitando que la consulta vuelva a fallar.

Opciones de fuente y sumidero

El hecho de que una opción readStream o writeStream se pueda cambiar entre reinicios depende de la semántica de la fuente o el sumidero específico. Por ejemplo, no debe cambiar las opciones de host y puerto para el origen del socket si los datos se van a enviar a ese host y puerto. Pero puede agregar una opción al sumidero de la consola para imprimir hasta cien recuentos modificados después de cada activación:

```
writeStream.format("consola").option("numRows", "100")...
```

Detalles del

procesamiento Como se discutió anteriormente, la ubicación del punto de control no debe cambiarse entre reinicios. Sin embargo, otros detalles como el intervalo de activación se pueden cambiar sin romper las garantías de tolerancia a fallas.

Para obtener más información sobre el conjunto limitado de cambios que se permiten entre reinicios, consulte la [Guía de programación de transmisión estructurada más reciente](#).

Monitoreo de una consulta activa

parte importante de la ejecución de una canalización de transmisión en producción es el seguimiento de su estado. La transmisión estructurada proporciona varias formas de realizar un seguimiento del estado y las métricas de procesamiento de una consulta activa.

Consulta del estado actual mediante

StreamingQuery Puede consultar el estado actual de una consulta activa mediante la instancia de StreamingQuery . Aquí hay dos métodos:

Obtenga métricas actuales usando StreamingQuery. Cuando una consulta procesa algunos datos en un microlote, consideramos que ha realizado algún progreso. lastProgress() devuelve información sobre el último microlote completado. Por ejemplo, imprimir el objeto devuelto (StreamingQueryProgress en Scala/Java o un diccionario en Python) producirá algo como esto:

```
// En Scala/Python {  
  
    "id": "ce011fdc-8762-4dcb-84eb-a77333e28109", "runId":  
    "88e2ff94-edc0-45a8-b687-6316fbef529a", "nombre": "Mi  
    consulta", "marca de tiempo": "2016-12-14T18:45:24.873Z",  
    "numInputRows": 10, "inputRowsPerSecond": 120.0,  
    "processedRowsPerSecond": 200.0, "durationMs":  
    { "triggerExecution": 3, "getOffset": 2 }, "stateOperators":  
    [ ], "fuentes": [ { "descripción": "KafkaSource[Subscribe[tema-0]]",  
    "startOffset": { "tema-0": { "2": 0, "1": 1, "0": 1  
  
    } },  
    "compensación  
    final": { "tema-0":  
        { "2": 0, "1":  
        134, "0": 534  
  
    } },  
    "numInputRows": 10,  
    "inputRowsPerSecond": 120.0,  
    "processedRowsPerSecond": 200.0
```

```

    },
    "fregadero" :
      { "descripción" : "MemorySink"
    }
}

```

Algunas de las columnas destacadas son:

id
Identificador único vinculado a la ubicación de un punto de control. Esto permanece igual a lo largo de la vida de una consulta (es decir, entre reinicios).

ID de ejecución
Identificador único para la instancia actual (re)iniciada de la consulta. Esto cambia con cada reinicio.

numInputRows

Número de filas de entrada que se procesaron en el último microlote.

inputRowsPerSecond

Tasa actual a la que se generan las filas de entrada en el origen (promedio durante la duración del último microlote).

filas procesadas por segundo

Tasa actual a la que el sumidero procesa y escribe las filas (promedio durante la duración del último microlote). Si esta tasa es constantemente más baja que la tasa de entrada, entonces la consulta no puede procesar los datos tan rápido como los genera la fuente. Este es un indicador clave del estado de la consulta.

fuentes y sumidero

Proporciona detalles específicos de fuente/sumidero de los datos procesados en el último lote.

Obtenga el estado actual usando StreamingQuery.status(). Esto proporciona información sobre lo que está haciendo el subproceso de consulta en segundo plano en este momento. Por ejemplo, imprimir el objeto devuelto producirá algo como esto:

```
// En Scala/Python {
  "mensaje": "Esperando que lleguen los datos",
  "isDataAvailable": falso, "isTriggerActive": falso}
```

Publicación de métricas con Dropwizard

Metrics Spark admite métricas de informes a través de una biblioteca popular llamada [Dropwizard Metrics](#). Esta biblioteca permite que las métricas se publiquen en muchos marcos de monitoreo populares (Ganglia, Graphite, etc.). Estas métricas no están habilitadas de forma predeterminada para las consultas de transmisión estructurada debido a su alto volumen de datos informados. Para habilitarlos, aparte

desde la configuración de Dropwizard Metrics para Spark, debe establecer explícitamente la configuración de `SparkSession spark.sql.streaming.metricsEnabled` en verdadero antes de iniciar su consulta.

Tenga en cuenta que solo un subconjunto de la información disponible a través de `StreamingQuery.lastProgress()` se publica a través de Dropwizard Metrics. Si desea publicar continuamente más información de progreso en ubicaciones arbitrarias, debe escribir oyentes personalizados, como se explica a continuación.

Publicación de métricas mediante StreamingQueryListeners

personalizados `StreamingQueryListener` es una interfaz de escucha de eventos con la que puede injectar lógica arbitraria para publicar métricas de forma continua. Esta API para desarrolladores solo está disponible en Scala/Java. Hay dos pasos para usar oyentes personalizados:

1. Defina su oyente personalizado. La interfaz `StreamingQueryListener` proporciona tres métodos que su implementación puede definir para obtener tres tipos de eventos relacionados con una consulta de transmisión: inicio, progreso (es decir, se ejecutó un activador) y terminación. Aquí hay un ejemplo:

```
// En Scala
import org.apache.spark.sql.streaming._ val
myListener = new StreamingQueryListener() {
    override def onQueryStarted(evento: QueryStartedEvent): Unit =
        { println("Consulta iniciada: " + evento.id)

    } invalidar def en ConsultaTerminada(evento: ConsultaTerminadaEvento): Unidad
        = { println("Consulta terminada: " + evento.id)

    } override def onQueryProgress(evento: QueryProgressEvent): Unit =
        { println("Consulta realizada progreso: " + evento.progreso)
    }
}
```

2. Agregue su oyente a `SparkSession` antes de iniciar la consulta:

```
// En Scala
spark.streams.addListener(myListener)
```

Después de agregar el oyente, todos los eventos de consultas de transmisión que se ejecutan en esta sesión de Spark comenzarán a llamar a los métodos del oyente.

Fuentes y sumideros de datos de transmisión

Ahora que hemos cubierto los pasos básicos que necesita para expresar una consulta de transmisión estructurada de un extremo a otro, examinemos cómo usar las fuentes y los sumideros de datos de transmisión integrados. Como recordatorio, puede crear tramas de datos a partir de fuentes de transmisión mediante `SparkSession.readStream()` y escribir la salida de una trama de datos resultante mediante `DataFrame.writeStream()`. En cada caso, puede especificar el tipo de fuente usando el método `format()`. Más adelante veremos algunos ejemplos concretos.

archivos

La transmisión estructurada admite la lectura y escritura de flujos de datos hacia y desde archivos en los mismos formatos que los admitidos en el procesamiento por lotes: texto sin formato, CSV, JSON, Parquet, ORC, etc. Aquí analizaremos cómo operar la transmisión estructurada en archivos.

Lectura de archivos

La transmisión estructurada puede tratar los archivos escritos en un directorio como un flujo de datos. Aquí hay un ejemplo:

```
# En Python
desde pyspark.sql.types import *
inputDirectoryOfJsonFiles = ...

fileSchema =
    (StructType() .add(StructField("clave",
        IntegerType()) .add(StructField("valor", IntegerType())))

inputDF =
    (chispa .readStream

        .format("json") .schema(fileSchema) .load(inputDirectoryOfJsonFiles))

// En Scala
import org.apache.spark.sql.types._ val
inputDirectoryOfJsonFiles = ...

val fileSchema = new
    StructType() .add("clave",
        IntegerType) .add("valor", IntegerType)

val inputDF = chispa.readStream

        .format("json") .schema(fileSchema) .load(inputDirectoryOfJsonFiles)
```

El DataFrame de transmisión devuelto tendrá el esquema especificado. Aquí hay algunos puntos clave para recordar al usar archivos:

- Todos los archivos deben tener el mismo formato y se espera que tengan el mismo esquema. Por ejemplo, si el formato es "json", todos los archivos deben estar en formato JSON con un registro JSON por línea. El esquema de cada registro JSON debe coincidir con el especificado con `readStream()`. La violación de estas suposiciones puede dar lugar a un análisis incorrecto (por ejemplo, valores nulos inesperados) o errores de consulta.
- Cada archivo debe aparecer en la lista de directorios de forma atómica, es decir, el archivo completo debe estar disponible para su lectura a la vez y, una vez que está disponible, el archivo no se puede actualizar ni modificar. Esto se debe a que Structured Streaming procesará el archivo cuando el motor lo encuentre (utilizando la lista de directorios) e internamente lo marcará como procesado. Cualquier cambio en ese archivo no será procesado.
- Cuando hay varios archivos nuevos para procesar, pero solo puede seleccionar algunos de ellos en el siguiente microlote (p. ej., debido a los límites de velocidad), seleccionará los archivos con las marcas de tiempo más antiguas. Sin embargo, dentro del microlote no existe un orden predefinido de lectura de los archivos seleccionados; todos ellos serán leídos en paralelo.



Esta fuente de archivos de transmisión admite varias opciones comunes, incluidas las opciones específicas del formato de archivo admitidas por `spark.read()` (consulte “[Fuentes de datos para marcos de datos y tablas SQL](#)” [en la página 94](#) en el [Capítulo 4](#)) y varias opciones específicas de transmisión (por ejemplo, `maxFilesPerTrigger` para limitar la tasa de procesamiento de archivos). Ver la [guía de programación](#) para detalles completos.

Escribir en

archivos La transmisión estructurada admite la escritura de resultados de consultas de transmisión en archivos en los mismos formatos que las lecturas. Sin embargo, solo es compatible con el modo de adición, porque si bien es fácil escribir nuevos archivos en el directorio de salida (es decir, agregar datos a un directorio), es difícil modificar los archivos de datos existentes (como se esperaría con los modos de actualización y finalización). También es compatible con la partición. Aquí hay un ejemplo:

```
# En Python
salidaDir = ...
checkpointDir = ...
resultDF = ...

streamingQuery =
(resultDF.writeStream .format("parquet") .option("path",
outputDir) .option("checkpointLocation", checkpointDir) .start())
```

```
// En Scala
val outputDir = ...
val checkpointDir = val ...
resultDF = ...

val streamingQuery = ...

resultDF .writeStream .format("parquet") .option("path",
outputDir) .option("checkpointLocation", checkpointDir) .start()
```

En lugar de usar la opción "ruta" , puede especificar el directorio de salida directamente como inicio (dirección de salida).

Algunos puntos clave para recordar:

- La transmisión estructurada logra garantías de extremo a extremo exactamente una vez al escribir en archivos al mantener un registro de los archivos de datos que se han escrito en el directorio. Este registro se mantiene en el subdirectorio _spark_metadata. Cualquier consulta de Spark en el directorio (no en sus subdirectorios) usará automáticamente el registro para leer el conjunto correcto de archivos de datos para que se mantenga la garantía de una sola vez (es decir, no se lean datos duplicados ni archivos parciales). Tenga en cuenta que es posible que otros motores de procesamiento no conozcan este registro y, por lo tanto, es posible que no brinden la misma garantía.
- Si cambia el esquema del DataFrame resultante entre reinicios, el directorio de salida tendrá datos en múltiples esquemas. Estos esquemas deben reconciliarse al consultar el directorio.

apache kafka

apache kafka es un popular sistema de publicación/suscripción que se usa ampliamente para el almacenamiento de flujos de datos. La transmisión estructurada tiene soporte integrado para leer y escribir en Apache Kafka.

Lectura desde

Kafka Para realizar lecturas distribuidas desde Kafka, debe usar opciones para especificar cómo conectarse a la fuente. Digamos que desea suscribirse a los datos del tema "eventos".

Así es como puede crear un DataFrame de transmisión:

```
# En Python
inputDF =

(spark .readStream .format("kafka") .option("kafka.bootstrap.servers",
"host1:port1,host2:port2") .option("subscribe", "events") .load ())
```

```
// En escala
val inputDF = chispa
    .readStream
    .formato ("kafka")
    .option("kafka.bootstrap.servidores", "host1:puerto1,host2:puerto2")
    .option("suscribirse", "eventos")
    .carga()
```

El DataFrame devuelto tendrá el esquema descrito en [la Tabla 8-1](#).

Tabla 8-1. Esquema del DataFrame generado por la fuente Kafka

Nombre de la columna	Tipo de columna	Descripción
llave	binario	Datos clave del registro en bytes.
valor	binario	Valorar los datos del registro como bytes.
tema	cuerda	Tema de Kafka en el que se encontraba el registro. Esto es útil cuando se suscribe a varios temas.
dividir	En t	Partición del tema de Kafka en el que se encontraba el registro.
compensar	largo	Valor de compensación del registro.
marca de tiempo	largo	Marca de tiempo asociada con el registro.
timestampType	int	Enumeración para el tipo de sello de tiempo asociado al registro.

También puede optar por suscribirse a varios temas, un patrón de temas o incluso un especial. partición específica de un tema. Además, puede elegir si desea leer solo datos nuevos en los temas suscritos o procesar todos los datos disponibles en esos temas. Puede incluso lea datos de Kafka de consultas por lotes, es decir, trate los temas de Kafka como tablas. Ver la [guía de integración de Kafka](#) para más detalles.

Escribiendo a Kafka

Para escribir en Kafka, Structured Streaming espera que el resultado DataFrame tenga un pocas columnas de nombres y tipos específicos, como se describe en [la Tabla 8-2](#).

Tabla 8-2. Esquema de DataFrame que se puede escribir en el receptor de Kafka

Nombre de la columna	tipo	Descripción
llave (opcional)	o binario	Si está presente, los bytes se escribirán como la clave de registro de Kafka; de lo contrario, la clave estará vacío.
valor (requerido)	cadena o binario	Los bytes se escribirán como el valor de registro de Kafka.
tema (requerido sólo si "tema" no es especificado como opción)	cuerda	Si no se especifica "tema" como opción, esto determina el tema para escribir el clave/valor para. Esto es útil para distribuir las escrituras en varios temas. Si el se ha especificado la opción "tema" , este valor se ignora.

Puede escribir en Kafka en los tres modos de salida, aunque no se recomienda el modo completo, ya que generará repetidamente los mismos registros. Aquí hay un ejemplo concreto de cómo escribir el resultado de nuestra consulta de conteo de palabras anterior en Kafka en modo de actualización:

```
# En Python
cuenta = ... # DataFrame[palabra: cadena, cuenta: largo]
streamingQuery =
(cuenta .selectExpr( "cast(word
    as string) as key", "cast(count as
    string) as value")

.writeStream .format("kafka") .option("kafka.bootstrap.servers",
"host1:puerto1,host2:puerto2") .option("tema",
"wordCounts") .outputMode("actualizar") .option ("ubicación del punto de
control!", dirección del punto de control) .start())

// En Scala
val cuenta = ... // DataFrame[palabra: cadena, cuenta: largo] val
streamingQuery = cuenta .selectExpr( "cast(word as string) as key",
"cast(count as string) as value" )

.writeStream .format("kafka") .option("kafka.bootstrap.servers",
"host1:puerto1,host2:puerto2") .option("tema",
"wordCounts") .outputMode("actualizar") .option ("Ubicación del punto de
control!", Dirección del punto de control) .start()
```

Consulte la [Guía de integración de Kafka](#) para más detalles.

Fuentes y sumideros de transmisión personalizados

En esta sección, analizaremos cómo leer y escribir en sistemas de almacenamiento que no tienen soporte integrado en transmisión estructurada. En particular, verá cómo usar los métodos `foreachBatch()` y `foreach()` para implementar una lógica personalizada para escribir en su almacenamiento.

Escritura en cualquier sistema

de almacenamiento Hay dos operaciones que le permiten escribir el resultado de una consulta de transmisión en sistemas de almacenamiento arbitrarios: `foreachBatch()` y `foreach()`. Tienen casos de uso ligeramente diferentes: mientras que `foreach()` permite una lógica de escritura personalizada en cada fila, `foreach Batch()` permite operaciones arbitrarias y una lógica personalizada en la salida de cada micro lote. Exploraremos su uso con más detalle.

Usando foreachBatch(). foreachBatch() le permite especificar una función que se ejecuta en la salida de cada microlote de una consulta de transmisión. Toma dos parámetros: un DataFrame o Dataset que tiene la salida de un micro-lote, y el identificador único del micro-lote. Como ejemplo, supongamos que queremos escribir el resultado de nuestra consulta de recuento de palabras anterior en [Apache Cassandra](#). A partir de [Spark Cassandra Connector 2.4.2](#), no hay soporte para escribir DataFrames de transmisión. Pero puede usar el soporte de DataFrame por lotes del conector para escribir la salida de cada lote (es decir, recuentos de palabras actualizados) en Cassandra, como se muestra aquí:

```
# En Python
hostAddr = "<dirección IP>"
keyspaceName = "<keyspace>"
tableName = "<tableName>"

chispa.conf.set("chispa.cassandra.conexión.host", hostAddr)

def writeCountsToCassandra(updatedCountsDF, lotId):
    # Use la fuente de datos por lotes de Cassandra para escribir los recuentos
    # actualizados (updatedCountsDF
    .escribe

    .format("org.apache.spark.sql.cassandra") .mode("append") .options(tableName, keyspace=keyspaceName) .save()

streamingQuery =
    (cuenta .writeStream .foreachBatch(writeCountsToCassandra) .outputMode("update") .option("checkpointLocation", checkpointDir))

// En Scala
import org.apache.spark.sql.DataFrame

val hostAddr = "<dirección IP>" val
keyspaceName = "<keyspace>" val
tableName = "<tableName>"

chispa.conf.set("chispa.cassandra.conexión.host", hostAddr)

def writeCountsToCassandra(updatedCountsDF: DataFrame, batchId: Long) { // Usa la fuente
    de datos por lotes de Cassandra para escribir los recuentos actualizados
    updatedCountsDF .write .format("org.apache.spark.sql.cassandra") .options(Map("table" ->
        tableName, "keyspace" -> keyspaceName)) .mode("append") .save()

}
```

```
val streamingQuery =

    recuentos .writeStream .foreachBatch(writeCountsToCassandra
    _).outputMode("update") .option("checkpointLocation", checkpointDir) .start()
```

Con foreachBatch(), puede hacer lo siguiente:

Reutilizar fuentes de datos por lotes

existentes Como se muestra en el ejemplo anterior, con foreachBatch() puede usar fuentes de datos por lotes existentes (es decir, fuentes que admiten la escritura de tramas de datos por lotes) para escribir la salida de consultas de transmisión.

Escribir en varias ubicaciones

Si desea escribir la salida de una consulta de transmisión en varias ubicaciones (p. ej., un almacén de datos OLAP y una base de datos OLTP), simplemente puede escribir la trama de datos/conjunto de datos de salida varias veces. Sin embargo, cada intento de escritura puede hacer que se vuelvan a calcular los datos de salida (incluida la posible relectura de los datos de entrada). Para evitar recálculos, debe almacenar en caché el marco de datos de salida por lotes, escribirlo en varias ubicaciones y luego quitarlo de la memoria caché:

```
# En Python
def writeCountsToMultipleLocations(updatedCountsDF, batchId):
    updatedCountsDF.persist() updatedCountsDF.write.format(...).save() #
    Ubicación 1 updatedCountsDF.write.format(...).save() # Ubicación 2
    actualizadoCountsDF.unpersist()
```

// En Scala

```
def
```

```
writeCountsToMultipleLocations( UpdatedCountsDF: DataFrame,
    batchId: Long) { UpdatedCountsDF.persist()
    UpdatedCountsDF.write.format(...).save() // Ubicación 1 UpdatedCountsDF.write.format(...). save() // Ubicación 2
}
```

Aplicar operaciones adicionales de DataFrame

Muchas operaciones de la API de DataFrame no son compatibles³ en la transmisión de DataFrames porque la transmisión estructurada no admite la generación de planes incrementales en esos casos. Con foreachBatch(), puede aplicar algunas de estas operaciones en cada salida de microlote. Sin embargo, tendrá que razonar sobre la semántica de extremo a extremo de realizar la operación usted mismo.

³ Para obtener la lista completa de operaciones no admitidas, consulte la Guía de programación de transmisión estructurada.



`foreachBatch()` solo proporciona garantías de escritura al menos una vez. Puede obtener garantías de una sola vez utilizando el ID de lote para deduplicar varias escrituras de microlotes reejecutados.

Usando `foreach()`. Si `foreachBatch()` no es una opción (por ejemplo, si no existe un escritor de datos por lotes correspondiente), puede expresar su lógica de escritor personalizado mediante `foreach()`. Específicamente, puede expresar la lógica de escritura de datos dividiéndola en tres métodos: `abrir()`, `procesar()` y `cerrar()`. La transmisión estructurada utilizará estos métodos para escribir cada partición de los registros de salida. Aquí hay un ejemplo abstracto:

```
# en pitón
# Variación 1: Usando la función def
process_row(fila):
    # Escribir fila en almacenamiento
    aprobar

consulta = streamingDF.writeStream.foreach(process_row).start()

# Variación 2: Uso de la clase ForeachWriter class
ForeachWriter:
    def abierto(auto , ID de partición, ID de época):
        # Conexión abierta al almacén de datos
        # Devuelve True si la escritura debe continuar
        # Este método es opcional en Python
        # Si no se especifica, la escritura continuará automáticamente y devolverá True

    proceso de definición (uno mismo, fila):
        # Escribir una cadena en el almacén de datos usando una conexión abierta
        # Este método NO es opcional en Python
        aprobar

    def close(self, error): # Cierra la
        conexión. Este método es opcional en Python
        aprobar

resultDF.writeStream.foreach(ForeachWriter()).start()

// En Scala
import org.apache.spark.sql.ForeachWriter val
foreachWriter = new ForeachWriter[String] { // escrito con Strings

    def abrir(partitionId: Long, epochId: Long): Boolean = {
        // Abrir conexión al almacén de datos
        // Devuelve verdadero si la escritura debe continuar
    }

    def proceso(registro: Cadena): Unidad = {
        // Escribir una cadena en el almacén de datos usando una conexión abierta
    }
}
```

```

    }

    def close(errorOrNull: Throwble ): Unidad = {
    //Cerrar la conexión
}
}

resultadoDSofStrings.writeStream.foreach(foreachWriter).start()

```

La semántica detallada de estos métodos tal como se ejecutan se discuten en el [Estructurado Guía de programación de transmisión.](#)

Lectura desde cualquier sistema

de almacenamiento Desafortunadamente, a partir de Spark 3.0, las API para crear fuentes y receptores de transmisión personalizados aún son experimentales. La iniciativa DataSourceV2 en Spark 3.0 presenta las API de transmisión, pero aún no se han declarado estables. Por lo tanto, no existe una forma oficial de leer desde sistemas de almacenamiento arbitrarios.

Transformaciones de datos

En esta sección, vamos a profundizar en las transformaciones de datos compatibles con la transmisión estructurada. Como se discutió brevemente anteriormente, solo las operaciones de DataFrame que se pueden ejecutar de forma incremental son compatibles con la transmisión estructurada. Estas operaciones se clasifican ampliamente en operaciones sin estado y con estado. Definiremos cada tipo de operación y explicaremos cómo identificar qué operaciones tienen estado.

Ejecución incremental y estado de transmisión Como

discutimos en “[Debajo del capó de una consulta de transmisión activa](#)” en la página 219, el optimizador Catalyst en Spark SQL convierte todas las operaciones de DataFrame en un plan lógico optimizado. El planificador Spark SQL, que decide cómo ejecutar un plan lógico, reconoce que se trata de un plan lógico de transmisión que necesita operar en flujos de datos continuos. En consecuencia, en lugar de convertir el plan lógico en un plan de ejecución física único, el planificador genera una secuencia continua de planes de ejecución. Cada plan de ejecución actualiza el DataFrame del resultado final de forma incremental, es decir, el plan procesa solo una parte de los datos nuevos de los flujos de entrada y posiblemente algún resultado parcial intermedio calculado por el plan de ejecución anterior.

Cada ejecución se considera como un microlote, y el resultado intermedio parcial que se comunica entre las ejecuciones se denomina “estado” de transmisión. Las operaciones de marcos de datos se pueden clasificar en términos generales en operaciones sin estado y con estado en función de si ejecutar la operación de forma incremental requiere mantener un estado. En el resto de esta sección, exploraremos la distinción entre operaciones sin estado y con estado y cómo su presencia en una consulta de transmisión requiere una configuración de tiempo de ejecución y una administración de recursos diferentes.



Algunas operaciones lógicas son fundamentalmente poco prácticas o muy caras de computar de forma incremental y, por lo tanto, no se admiten en la transmisión estructurada. Por ejemplo, cualquier intento de iniciar una consulta de transmisión con una operación como `cube()` o `rollup()` arrojará una `UnsupportedOperationException`.

Transformaciones sin estado

Todas las operaciones de proyección (por ejemplo, `seleccionar()`, `explotar()`, `mapa()`, `planoMapa()`) y las operaciones de selección (por ejemplo, `filtrar()`, `donde()`) procesan cada registro de entrada individualmente sin necesidad de información de las filas anteriores. . Esta falta de dependencia de los datos de entrada anteriores los convierte en operaciones sin estado.

Una consulta de transmisión que solo tiene operaciones sin estado admite los modos de salida agregar y actualizar, pero no el modo completo. Esto tiene sentido: dado que cualquier fila de salida procesada de dicha consulta no puede ser modificada por ningún dato futuro, se puede escribir en todos los sumideros de transmisión en modo de adición (incluidos los de solo adición, como archivos de cualquier formato). Por otro lado, tales consultas naturalmente no combinan información entre registros de entrada y, por lo tanto, es posible que no reduzcan el volumen de datos en el resultado.

El modo completo no es compatible porque almacenar los datos de resultados en constante crecimiento suele ser costoso. Esto está en marcado contraste con las transformaciones con estado, como discutiremos Siguiente.

Transformaciones con estado

El ejemplo más simple de una transformación con estado es `DataFrame.groupBy().count()`, que genera un conteo continuo de la cantidad de registros recibidos desde el comienzo de la consulta. En cada microlote, el plan incremental suma el conteo de nuevos registros al conteo anterior generado por el microlote anterior. Este cómputo parcial comunicado entre planes es el estado. Este estado se mantiene en la memoria de los ejecutores de Spark y se señala en la ubicación configurada para tolerar fallas. Si bien Spark SQL administra automáticamente el ciclo de vida de este estado para garantizar resultados correctos, por lo general, debe ajustar algunas perillas para controlar el uso de recursos para mantener el estado. En esta sección, vamos a explorar cómo los diferentes operadores con estado administran su estado bajo el capó.

Administración de estado distribuida y tolerante

a fallas Recuerde de los Capítulos 1 y 2 que una aplicación Spark que se ejecuta en un clúster tiene un controlador y uno o más ejecutores. El programador de Spark que se ejecuta en el controlador divide las operaciones de alto nivel en tareas más pequeñas y las coloca en colas de tareas y, a medida que los recursos están disponibles, los ejecutores extraen las tareas de las colas para ejecutarlas. Cada microlote en una consulta de transmisión esencialmente realiza un conjunto de tareas de este tipo que lee datos nuevos de fuentes de transmisión y escribe resultados actualizados en receptores de transmisión. Para consultas de procesamiento de flujo con estado, además de escribir en sumideros, cada microlote de tareas genera datos de estado intermedio que serán consumidos por el siguiente microlote. Esta generación de datos de estado está completamente dividida y distribuida (ya que toda la lectura, escritura y procesamiento se realiza en Spark) y se almacena en caché en la memoria del ejecutor para un consumo eficiente. Esto se ilustra en la figura 8-6, que muestra cómo se gestiona el estado en nuestra consulta de recuento de palabras de transmisión original.

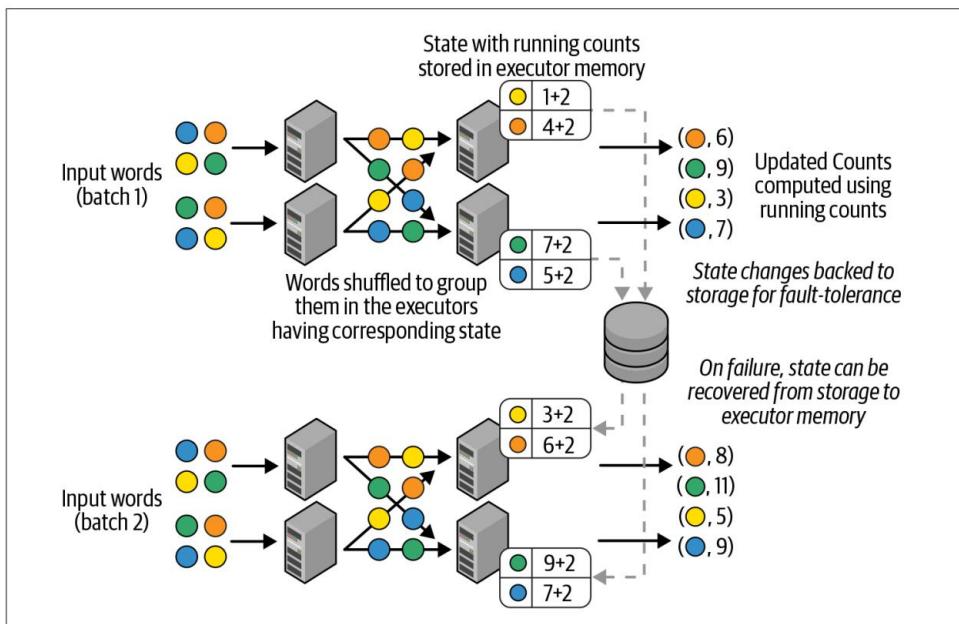


Figura 8-6. Gestión de estado distribuido en Streaming estructurado

Cada microlote lee un nuevo conjunto de palabras, las mezcla dentro de los ejecutores para agruparlas, calcula los recuentos dentro del microlote y finalmente los agrega a los recuentos continuos para producir los nuevos recuentos. Estos nuevos recuentos son tanto la salida como el estado del siguiente microlote y, por lo tanto, se almacenan en caché en la memoria de los ejecutores. El siguiente microlote de datos se agrupa entre ejecutores exactamente de la misma manera que antes, de modo que cada palabra siempre es procesada por el mismo ejecutor y, por lo tanto, puede leer y actualizar localmente su cuenta corriente.

Sin embargo, no es suficiente mantener este estado en la memoria, ya que cualquier falla (ya sea de un ejecutor o de toda la aplicación) hará que se pierda el estado en memoria. Para evitar pérdidas, guardamos sincrónicamente la actualización de estado de clave/valor como registros de cambios en la ubicación del punto de control proporcionada por el usuario. Estos cambios se co-versionan con los rangos de compensación procesados en cada lote, y la versión requerida del estado se puede reconstruir automáticamente leyendo los registros marcados. En caso de cualquier falla, Structured Streaming puede volver a ejecutar el microlote fallido al reprocesar los mismos datos de entrada junto con el mismo estado que tenía antes de ese microlote, produciendo así los mismos datos de salida que tendría si no había fallado. Esto es fundamental para garantizar las garantías exactamente una vez de extremo a extremo.

En resumen, para todas las operaciones con estado, la transmisión estructurada garantiza la corrección de la operación al guardar y restaurar automáticamente el estado de forma distribuida. Dependiendo de la operación con estado, todo lo que debe hacer es ajustar la política de limpieza de estado de modo que las claves y los valores antiguos se puedan eliminar automáticamente del estado almacenado en caché. Esto es lo que discutiremos a continuación.

Tipos de operaciones con

estado La esencia del estado de transmisión es conservar resúmenes de datos anteriores. A veces, los resúmenes antiguos deben limpiarse del estado para dejar espacio para nuevos resúmenes. En función de cómo se haga esto, podemos distinguir dos tipos de operaciones con estado:

Operaciones con estado administradas

Estos identifican y limpian automáticamente el estado antiguo, en función de una definición específica de operación de "antiguo". Puede ajustar lo que se define como antiguo para controlar el uso de recursos (por ejemplo, la memoria del ejecutor utilizada para almacenar el estado). Las operaciones que entran en esta categoría son aquellas para:

- Agregaciones de transmisión •
- Uniones de transmisión a transmisión
- Deduplicación de transmisión

Operaciones con estado no administradas

Estas operaciones le permiten definir su propia lógica de limpieza de estado personalizada. Las operaciones en esta categoría son:

- MapGroupsWithState
- FlatMapGroupsWithState

Estas operaciones le permiten definir operaciones arbitrarias con estado (sesionización, etc.).

Cada una de estas operaciones se analiza en detalle en las siguientes secciones.

Agregaciones de transmisión con estado

La transmisión estructurada puede ejecutar de forma incremental la mayoría de las operaciones de agregación de DataFrame. Puede agregar datos por claves (p. ej., conteo de palabras de transmisión) y/o por tiempo (p. ej., conteo de registros recibidos cada hora). En esta sección, vamos a discutir la semántica y los detalles operativos de ajustar estos diferentes tipos de agregaciones de transmisión. También analizaremos brevemente los pocos tipos de agregaciones que no son compatibles con la transmisión. Comencemos con agregaciones que no involucran tiempo.

Agregaciones no basadas en el tiempo

Las agregaciones que no implican tiempo se pueden clasificar en términos generales en dos categorías:

Agregaciones globales

Agregaciones en todos los datos de la secuencia. Por ejemplo, supongamos que tiene un flujo de lecturas de sensores como un DataFrame de transmisión llamado sensorReadings .

Puede calcular el conteo acumulado del número total de lecturas recibidas con la siguiente consulta:

```
# En Python
runningCount = sensorReadings.groupBy().count()

// En Scala
val runningCount = sensorReadings.groupBy().count()
```



No puede usar operaciones de agregación directa como DataFrame.count() y Dataset.reduce() en la transmisión de DataFrame. Esto se debe a que, para los DataFrames estáticos, estas operaciones devuelven inmediatamente los agregados calculados finales, mientras que para los DataFrames de transmisión, los agregados deben actualizarse continuamente. Por lo tanto, siempre debe usar Data Frame.groupBy() o Dataset.groupByKey() para las agregaciones en la transmisión de DataFrames.

Agregaciones agrupadas

Agregaciones dentro de cada grupo o clave presente en el flujo de datos. Por ejemplo, si sensorReadings contiene datos de varios sensores, puede calcular la lectura promedio móvil de cada sensor (por ejemplo, para configurar un valor de referencia para cada sensor) con lo siguiente:

```
# En Python
valores de línea base = lecturas del sensor.groupBy("sensorId").mean("valor")

// En Scala
val valores de línea base = lecturas del sensor.groupBy("sensorId").mean("valor")
```

Además de conteos y promedios, los marcos de datos de transmisión admiten los siguientes tipos de agregaciones (similares a los marcos de datos por lotes):

Todas las funciones de agregación

integradas sum(), mean(), stddev(), countDistinct(), collect_set(), approx_count_distinct(), etc. Consulte la documentación de la API ([Python](#) y [Scala](#)) para más detalles.

Múltiples agregaciones calculadas juntas

Puede aplicar múltiples funciones de agregación para que se calculen juntas de la siguiente manera:

```
# En Python
desde pyspark.sql.functions import *
multipleAggs =
    ( sensorReadings .groupBy("sensorId") .agg(count(**),
        mean("value").alias("baselineValue"), collect_set("errorCode ").alias("todos los códigos de error")))
// En Scala
import org.apache.spark.sql.functions.* val
multipleAggs =
    sensorReadings .groupBy("sensorId") .agg(count(**),
        mean("value").alias("baselineValue"),
        collect_set("CódigoError").alias("Todos los Códigos Error"))
```

Funciones de agregación definidas por el

usuario Se admiten todas las funciones de agregación definidas por el usuario. Consulte la [guía de programación de Spark SQL](#) para obtener más detalles sobre las funciones de agregación definidas por el usuario sin tipo y con tipo.

Con respecto a la ejecución de tales agregados de transmisión, ya hemos ilustrado en secciones anteriores cómo se mantienen los agregados en ejecución como un estado distribuido.

Además de esto, hay dos puntos muy importantes que recordar para las agregaciones que no se basan en el tiempo: el modo de salida que se utilizará para dichas consultas y la planificación del uso de recursos por estado. Estos se discuten hacia el final de esta sección. A continuación, analizaremos las agregaciones que combinan datos dentro de ventanas de tiempo.

Agregaciones con ventanas de tiempo de eventos

En muchos casos, en lugar de ejecutar agregaciones en todo el flujo, desea agregaciones sobre datos agrupados por ventanas de tiempo. Continuando con nuestro ejemplo de sensor, supongamos que se espera que cada sensor envíe como máximo una lectura por minuto y queremos detectar si algún sensor está informando una cantidad inusualmente alta de veces. Para encontrar tales anomalías, podemos contar el número de lecturas recibidas de cada sensor en intervalos de cinco minutos. Además, para mayor solidez, deberíamos calcular el intervalo de tiempo en función del momento en que se generaron los datos en el sensor y no en función del momento en que se generó la información.

se recibieron los datos, ya que cualquier retraso en el tránsito sesgaría los resultados. En otras palabras, queremos usar la hora del evento, es decir, la marca de tiempo en el registro que representa cuándo se generó la lectura. Digamos que sensorReadings DataFrame tiene la marca de tiempo de generación como una columna llamada eventTime. Podemos expresar esta cuenta de cinco minutos de la siguiente manera:

```
# En Python
desde pyspark.sql.functions import *
(sensorReadings .groupBy("sensorId",
    window("eventTime", "5 minute")) .count())
```

```
// En Scala
importar org.apache.spark.sql.functions.*
sensorReadings .groupBy("sensorId",
    window("eventTime", "5 minute")) .count()
```

La clave a tener en cuenta aquí es la función window() , que nos permite expresar las ventanas de cinco minutos como una columna de agrupación calculada dinámicamente. Cuando se inicie, esta consulta hará lo siguiente para cada lectura del sensor:

- Use el valor eventTime para calcular la ventana de tiempo de cinco minutos en la que cae la lectura del sensor.
- Agrupe la lectura según el grupo compuesto (<ventana calculada>, ID del sensor).
- Actualizar el conteo del grupo compuesto.

Entendamos esto con un ejemplo ilustrativo. La figura 8-7 muestra cómo se asignan algunas lecturas de sensor a grupos de ventanas de cinco minutos (es decir, no superpuestas) en función de la hora de su evento. Las dos líneas de tiempo muestran cuándo recibió cada evento

serán procesados por Flujo estructurado y la marca de tiempo en los datos del evento (generalmente, la hora en que se generó el evento en el sensor).

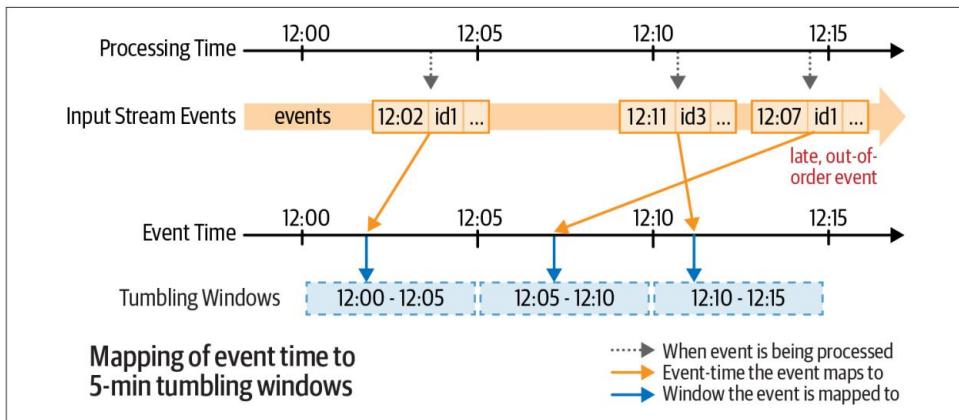


Figura 8-7. Mapeo del tiempo del evento a las ventanas giratorias

Cada ventana de cinco minutos sobre el tiempo del evento se considera para la agrupación en base a la cual se calcularán los conteos. Tenga en cuenta que los eventos pueden llegar tarde y fuera de orden en términos de tiempo del evento. Como se muestra en la figura, el evento con tiempo de evento 12:07 se recibió y procesó después del evento con tiempo 12:11. Sin embargo, independientemente de cuándo lleguen, cada evento se asigna al grupo adecuado en función de la hora del evento. De hecho, según la especificación de la ventana, cada evento se puede asignar a varios grupos. Por ejemplo, si desea calcular los recuentos correspondientes a ventanas de 10 minutos que se deslizan cada 5 minutos, puede hacer lo siguiente:

```
# En Python
(sensorReadings
    .groupBy("sensorId", ventana("eventTime", "10 minutos", "5 minutos")) .count())
```

// En Scala

```
sensorReadings .groupBy("sensorId", ventana("eventTime", "10 minutos", "5
    minutos")) .count()
```

En esta consulta, cada evento se asignará a dos ventanas superpuestas, como se ilustra en la Figura 8-8.

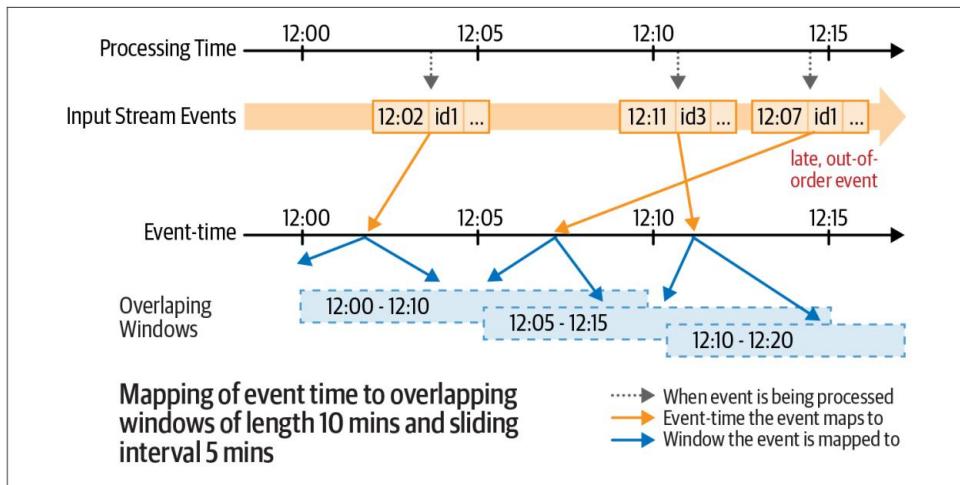


Figura 8-8. Mapeo del tiempo del evento a múltiples ventanas superpuestas

Cada tupla única de (*<ventana de tiempo asignada>*, sensorId) se considera un grupo generado dinámicamente para el cual se calcularán los recuentos. Por ejemplo, el evento [eventTime = 12:07, sensorId = id1] se asigna a dos ventanas de tiempo y, por lo tanto, a dos grupos (12:00-12:10, id1) y (12:05-12:15, id1). Los conteos para estas dos ventanas se incrementan en 1 cada uno. La Figura 8-9 ilustra esto para los eventos mostrados anteriormente.

Suponiendo que los registros de entrada se procesaron con un intervalo de activación de cinco minutos, las tablas en la parte inferior de la Figura 8-9 muestran el estado de la tabla de resultados (es decir, los conteos) en cada uno de los microlotes. A medida que avanza el tiempo del evento, se crean automáticamente nuevos grupos y sus agregados se actualizan automáticamente. Los eventos tardíos y fuera de servicio se manejan automáticamente, ya que simplemente actualizan los más antiguos. grupos

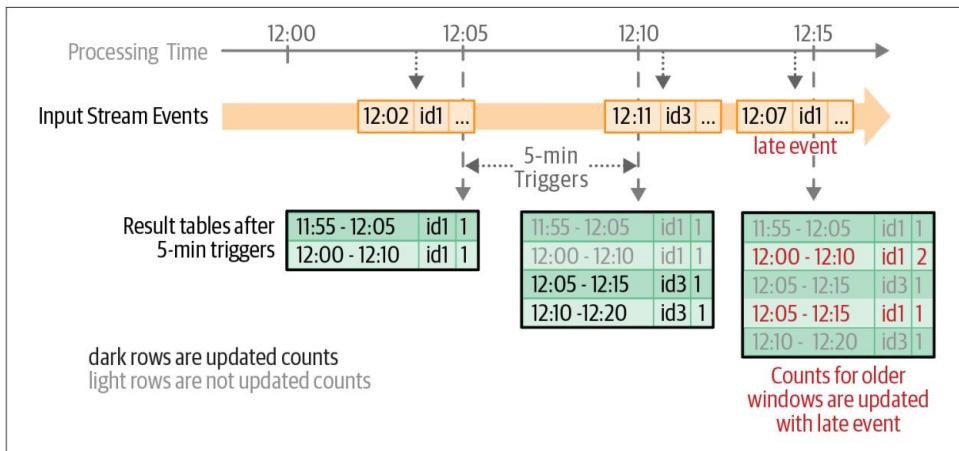


Figura 8-9. Recuentos actualizados en la tabla de resultados después de cada activación de cinco minutos

Sin embargo, desde el punto de vista del uso de los recursos, esto plantea un problema diferente: el crecimiento indefinido del tamaño del estado. A medida que se crean nuevos grupos correspondientes a las últimas ventanas de tiempo, los grupos más antiguos continúan ocupando la memoria de estado, esperando que se actualice cualquier dato tardío. Incluso si en la práctica existe un límite en cuanto a la demora de los datos de entrada (por ejemplo, los datos no pueden tener más de siete días de retraso), la consulta no conoce esa información. Por lo tanto, no sabe cuándo considerar una ventana como "demasiado antigua para recibir actualizaciones" y eliminarla del estado. Para proporcionar un límite de retraso a una consulta (y evitar un estado ilimitado), puede especificar marcas de agua, como veremos a continuación.

Manejo de datos atrasados con

marcas de agua Una marca de agua se define como un umbral móvil en el tiempo del evento que va detrás del tiempo máximo del evento visto por la consulta en los datos procesados. La brecha final, conocida como retraso de la marca de agua, define cuánto tiempo esperará el motor para que lleguen los datos tardíos. Al conocer el punto en el que no llegarán más datos para un grupo determinado, el motor puede finalizar automáticamente los agregados de ciertos grupos y eliminarlos del estado. Esto limita la cantidad total de estado que el motor debe mantener para calcular los resultados de la consulta.

Por ejemplo, suponga que sabe que los datos de su sensor no se retrasarán más de 10 minutos. Luego puede configurar la marca de agua de la siguiente manera:

En Python

```
( sensorReadings .withWatermark("eventTime", "10
minutes") .groupBy("sensorId", window("eventTime", "10 minutes", "5
minutes")) .mean("value"))
```

// En Scala

```
sensorReadings .withWatermark("eventTime", "10
minutes") .groupByKey("sensorId", window("eventTime", "10 minutes", "5
minute")) .mean("value")
```

Tenga en cuenta que debe llamar a `withWatermark()` antes de `groupBy()` y en la misma columna de marca de tiempo que la utilizada para definir ventanas. Cuando se ejecuta esta consulta, Structured Streaming rastreará continuamente el valor máximo observado de la columna `eventTime` y, en consecuencia, actualizará la marca de agua, filtrará los datos "demasiado tarde" y borrará el estado anterior. Es decir, se ignorarán todos los datos retrasados por más de 10 minutos, y todas las ventanas de tiempo que tengan más de 10 minutos más que los datos de entrada más recientes (por tiempo de evento) se borrarán del estado. Para aclarar cómo se ejecutará esta consulta, considere la línea de tiempo en la [Figura 8-10](#) que muestra cómo se procesó una selección de registros de entrada.

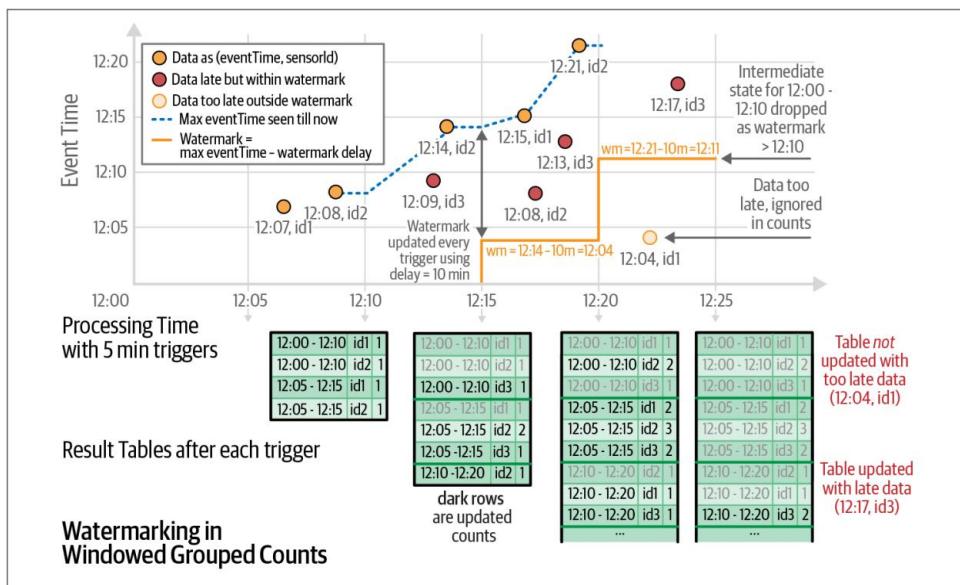


Figura 8-10. Ilustración de cómo el motor rastrea el tiempo máximo de evento entre eventos, actualiza la marca de agua y, en consecuencia, maneja los datos atrasados

Esta figura muestra un gráfico bidimensional de registros procesados en términos de sus tiempos de procesamiento (eje x) y sus tiempos de evento (eje y). Los registros se procesan en microlotes de cinco minutos y se marcan con círculos. Las tablas en la parte inferior muestran el estado de la tabla de resultados después de que se completa cada microlote.

Cada registro fue recibido y procesado después de todos los registros a su izquierda. Considere los dos registros [12:15, id1] (procesado alrededor de las 12:17) y [12:13, id3] (procesado alrededor de las 12:18). El registro de id3 se consideró tardío (y por lo tanto marcado en sólido)

rojo) porque fue generado por el sensor antes del registro para id1 pero fue procesado después de este último. Sin embargo, en el microlote para el intervalo de tiempo de procesamiento de 12:15 a 12:20, la marca de agua utilizada fue 12:04, que se calculó en función del tiempo máximo del evento visto hasta el microlote anterior (es decir, 12 horas). :14 menos el retraso de 10 minutos de la marca de agua). Por lo tanto, el registro tardío [12:13, id3] no se consideró demasiado tarde y se contó con éxito. Por el contrario, en el siguiente micro lote, se consideró que el registro [12:04, id1] era demasiado tarde en comparación con la nueva marca de agua de 12:11 y se descartó.

Puede configurar el retraso de la marca de agua en función de los requisitos de su aplicación: los valores más grandes para este parámetro permiten que los datos lleguen más tarde, pero a costa de un mayor tamaño de estado (es decir, uso de memoria) y viceversa.

Garantías semánticas con marcas de agua. Antes de concluir esta sección sobre marcas de agua, consideremos la garantía semántica precisa que proporciona la marca de agua. Una marca de agua de 10 minutos garantiza que el motor nunca perderá ningún dato que se retrase menos de 10 minutos en comparación con la hora del último evento visto en los datos de entrada. Sin embargo, la garantía es estricta solo en una dirección. No se garantiza que los datos retrasados por más de 10 minutos se eliminen, es decir, se pueden agregar. Si un registro de entrada con más de 10 minutos de retraso se agregará o no depende del momento exacto en que se recibió el registro y cuándo se activó el procesamiento de micro lotes.

Modos de salida

admitidos A diferencia de las agregaciones de transmisión que no involucran tiempo, las agregaciones con ventanas de tiempo pueden usar los tres modos de salida. Sin embargo, hay otras implicaciones con respecto a la limpieza del estado que debe tener en cuenta, según el modo:

Modo de

actualización En este modo, cada microlote generará solo las filas donde se actualizó el agregado. Este modo se puede utilizar con todo tipo de agregaciones. Específicamente para las agregaciones de ventana de tiempo, la marca de agua garantizará que el estado se limpie regularmente. Este es el modo más útil y eficiente para ejecutar consultas con agregaciones de transmisión. Sin embargo, no puede usar este modo para escribir agregados a sumideros de transmisión de solo agregar, como cualquier formato basado en archivos como Parquet y ORC (a menos que use Delta Lake, que discutiremos en el próximo capítulo).

Modo completo

En este modo, cada microlote generará todos los agregados actualizados, independientemente de su antigüedad o si contienen cambios. Si bien este modo se puede usar en todos los tipos de agregaciones, para las agregaciones de ventana de tiempo, usar el modo completo significa que el estado no se limpiará incluso si se especifica una marca de agua. La salida de todos los agregados requiere todo el estado anterior y, por lo tanto, los datos de agregación deben conservarse

incluso si se ha definido una marca de agua. Use este modo en agregaciones de ventanas de tiempo con precaución, ya que esto puede generar un aumento indefinido en el tamaño del estado y el uso de la memoria.

Modo de adición

Este modo solo se puede usar con agregaciones en ventanas de tiempo de eventos y con la marca de agua habilitada. Recuerde que el modo de adición no permite que cambien los resultados de salida anteriores. Para cualquier agregación sin marcas de agua, cada agregado puede actualizarse con cualquier dato futuro y, por lo tanto, estos no pueden generarse en el modo de adición. Solo cuando la marca de agua está habilitada en agregaciones en ventanas de tiempo de eventos, la consulta sabe cuándo un agregado no se actualizará más.

Por lo tanto, en lugar de generar las filas actualizadas, el modo agregar genera cada clave y su valor agregado final solo cuando la marca de agua garantiza que el agregado no se actualizará nuevamente. La ventaja de este modo es que le permite escribir agregados en sumideros de transmisión de solo agregar (por ejemplo, archivos). La desventaja es que la salida se retrasará por la duración de la marca de agua: la consulta tiene que esperar a que la marca de agua final supere la ventana de tiempo de una clave antes de que se pueda finalizar su agregado.

Uniones de transmisión

La transmisión estructurada admite la unión de un conjunto de datos de transmisión con otro conjunto de datos estático o de transmisión. En esta sección, exploraremos qué tipos de uniones (internas, externas, etc.) son compatibles y cómo usar marcas de agua para limitar el estado almacenado para uniones con estado. Comenzaremos con el caso simple de unir un flujo de datos y un conjunto de datos estático.

Combinaciones estáticas de flujo

Muchos casos de uso requieren unir un flujo de datos con un conjunto de datos estático. Por ejemplo, consideremos el caso de la monetización de anuncios. Suponga que es una empresa de publicidad que muestra anuncios en sitios web y gana dinero cuando los usuarios hacen clic en ellos. Supongamos que tiene un conjunto de datos estático de todos los anuncios que se mostrarán (conocidos como impresiones) y otro flujo de eventos para cada vez que los usuarios hacen clic en los anuncios mostrados. Para calcular los ingresos por clic, debe hacer coincidir cada clic en el flujo de eventos con la impresión de anuncio correspondiente en la tabla. Primero representemos los datos como dos marcos de datos, uno estático y uno de transmisión, como se muestra aquí:

```
# En Python
# Static DataFrame [adId: String, impressionTime: Timestamp, ...] # lectura de
su fuente de datos estáticosimpresionesStatic = spark.read . ...
# Streaming DataFrame [adId: String, clickTime: Timestamp, ...] # leyendo
desde su fuente de transmisiónclicksStream = spark.readStream . ...
```

// En Scala //

Marco de datos estático [adId: cadena, tiempo de impresión: marca de tiempo, ...] //
lectura de su fuente de datos estáticos **val** impresiones estáticas = chispa.lectura. ...

// Streaming DataFrame [adId: String, clickTime: Timestamp, ...] // lectura de su
fuente de transmisión **val** clicksStream = spark.readStream. ...

Para hacer coincidir los clics con las impresiones, simplemente puede aplicar una unión equitativa interna entre ellos usando la columna común de adId :

Coincidencia
en Python = clicksStream.join (impresiones estáticas, "adId")

// En Scala
val coincidió = clicksStream.join(impressionsStatic, "adId")

Este es el mismo código que habría escrito si tanto las impresiones como los clics fueran marcos de datos estáticos; la única diferencia es que usa spark.read() para el procesamiento por lotes y spark.readStream() para una secuencia.

Cuando se ejecuta este código, cada microlote de clics se une internamente con la tabla de impresiones estáticas para generar el flujo de salida de eventos coincidentes.

Además de las uniones internas, la transmisión estructurada también admite dos tipos de uniones externas estáticas de transmisión:

- Unión externa izquierda cuando el lado izquierdo es un DataFrame de transmisión •

Unión externa derecha cuando el lado derecho es un DataFrame de transmisión

Los otros tipos de uniones externas (por ejemplo, exterior completo y exterior izquierdo con un marco de datos de transmisión a la derecha) no se admiten porque no son fáciles de ejecutar de forma incremental.

En ambos casos admitidos, el código es exactamente como sería para una combinación externa izquierda/derecha entre dos tramas de datos estáticas:

Coincidencia
en Python = clicksStream.join (impresiones estáticas, "adId", "leftOuter")

// En Scala
val coincidió = clicksStream.join(impressionsStatic, Seq("adId"), "leftOuter")

Hay algunos puntos clave a tener en cuenta acerca de las uniones estáticas de flujo:

- Las uniones estáticas de flujo son operaciones sin estado y, por lo tanto, no requieren ningún tipo de marca de agua.
- El DataFrame estático se lee repetidamente mientras se une con la transmisión de datos de cada microlote, por lo que puede almacenar en caché el DataFrame estático para acelerar las lecturas. • Si los datos subyacentes en la fuente de datos en la que se definió el DataFrame estático cambian, la consulta de transmisión puede ver esos cambios.

en el comportamiento específico de la fuente de datos. Por ejemplo, si el DataFrame estático se definió en archivos, los cambios en esos archivos (p. ej., anexos) no se recuperarán hasta que se reinicie la consulta de transmisión.

En este ejemplo de flujo estático, hicimos una suposición importante: que la tabla de impresiones es una tabla estática. En realidad, se generará un flujo de nuevas impresiones a medida que se muestren nuevos anuncios. Si bien las uniones estáticas de flujo son buenas para enriquecer los datos en un flujo con información estática adicional (o que cambia lentamente), este enfoque es insuficiente cuando ambas fuentes de datos cambian rápidamente. Para eso, necesita uniones de flujo a flujo, que discutiremos a continuación.

Uniones de flujo a flujo

El desafío de generar uniones entre dos flujos de datos es que, en cualquier momento, la vista de cualquiera de los conjuntos de datos está incompleta, lo que hace que sea mucho más difícil encontrar coincidencias entre las entradas. Los eventos coincidentes de los dos flujos pueden llegar en cualquier orden y pueden retrasarse arbitrariamente. Por ejemplo, en nuestro caso de uso de publicidad, un evento de impresión y su evento de clic correspondiente pueden llegar fuera de orden, con retrasos arbitrarios entre ellos. La transmisión estructurada da cuenta de tales retrasos al almacenar en búfer los datos de entrada de ambos lados como estado de transmisión y verificar continuamente las coincidencias a medida que se reciben nuevos datos. La idea conceptual se esboza en la [figura 8-11](#).

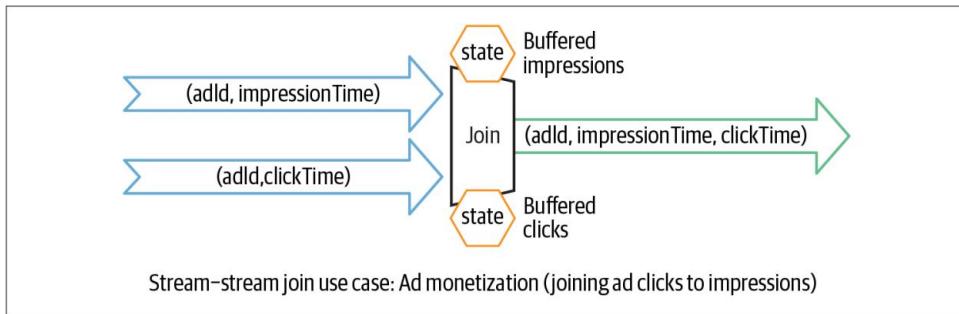


Figura 8-11. Monetización de anuncios usando una combinación de flujo a flujo

Consideremos esto con más detalle, primero con uniones internas y luego con uniones externas.

Uniones internas con marca de agua opcional

Supongamos que hemos redefinido nuestro DataFrame de impresiones para que sea un DataFrame de transmisión. Para obtener el flujo de impresiones coincidentes y sus clics correspondientes, podemos usar el mismo código que usamos anteriormente para uniones estáticas y uniones estáticas de flujo:

```

# En Python
# Streaming DataFrame [adId: String, impressionTime: Timestamp, ...] impresiones
= chispa.readStream. ...

# Streaming DataFrame[adId: String, clickTime: Timestamp, ...] clics =
chispa.readStream. ... coincidente = impresiones.join(clics, "adId")

// En Scala //
Streaming DataFrame [adId: String, impressionTime: Timestamp, ...] val impresiones
= chispa.readStream. ...

// Streaming DataFrame[adId: String, clickTime: Timestamp, ...] val clicks =
spark.readStream. ... valor coincidente = impresiones.join(clicks, "adId")

```

Aunque el código es el mismo, la ejecución es completamente diferente. Cuando se ejecuta esta consulta, el motor de procesamiento reconocerá que se trata de una unión de secuencia a secuencia en lugar de una unión de secuencia estática. El motor almacenará en búfer todos los clics e impresiones como estado, y generará una impresión y un clic coincidentes tan pronto como un clic recibido coincida con una impresión almacenada en búfer (o viceversa, dependiendo de cuál se recibió primero). Visualicemos cómo funciona esta unión interna usando la línea de tiempo de ejemplo de eventos en la [Figura 8-12](#).

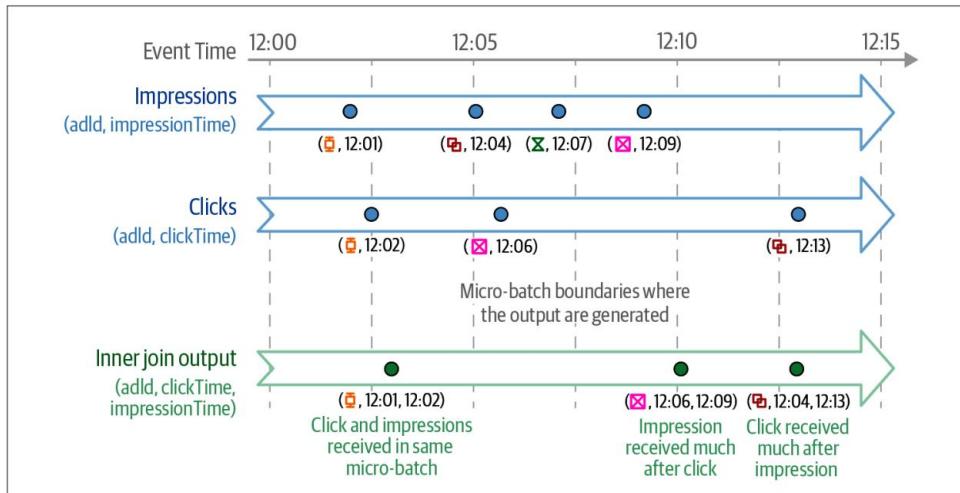


Figura 8-12. Cronología ilustrativa de clics, impresiones y su salida conjunta

En la [Figura 8-12](#), los puntos azules representan los tiempos de los eventos de impresión y clic que se recibieron en diferentes microlotes (separados por líneas grises discontinuas). A los efectos de esta ilustración, suponga que cada evento se recibió realmente a la misma hora del reloj de pared que la hora del evento. Tenga en cuenta los diferentes escenarios en los que se unen los eventos relacionados. Ambos eventos con adId = y fueron

recibidos en el mismo microlote, por lo que su salida conjunta fue generada por ese microlote. Sin embargo, para adId = γ la impresión se recibió a las 12:04, mucho antes que su clic correspondiente a las 12:13. La transmisión estructurada recibirá primero la impresión a las 12:04 y la almacenará en el estado. Por cada clic recibido, el motor intentará unirlo con todas las impresiones almacenadas (y viceversa). Finalmente, en un microlote posterior que se ejecuta alrededor de las 12:13, el motor recibe el clic para adId = γ y genera la salida conjunta.

Sin embargo, en esta consulta, no hemos dado ninguna indicación de cuánto tiempo el motor debe almacenar en búfer un evento para encontrar una coincidencia. Por lo tanto, el motor puede amortiguar un evento para siempre y acumular una cantidad ilimitada de estado de transmisión. Para limitar el estado de transmisión que mantienen las uniones de flujo a flujo, debe conocer la siguiente información sobre su caso de uso:

- ¿Cuál es el rango de tiempo máximo entre la generación de los dos eventos en sus respectivas fuentes? En el contexto de nuestro caso de uso, supongamos que un clic puede ocurrir entre cero segundos y una hora después de la impresión correspondiente. • ¿Cuál es la duración máxima que un evento puede demorar en tránsito entre la fuente y el motor de procesamiento? Por ejemplo, los clics en anuncios de un navegador pueden retrasarse debido a la conectividad intermitente y llegar mucho más tarde de lo esperado y fuera de servicio. Digamos que las impresiones y los clics se pueden retrasar como máximo dos y tres horas, respectivamente.

Estos límites de retraso y restricciones de tiempo de evento se pueden codificar en las operaciones de DataFrame usando marcas de agua y condiciones de rango de tiempo. En otras palabras, deberá realizar los siguientes pasos adicionales en la unión para garantizar la limpieza del estado:

1. Defina retrasos de marca de agua en ambas entradas, de modo que el motor sepa cómo la entrada puede retrasarse (similar a las agregaciones de transmisión).
2. Defina una restricción en el tiempo del evento en las dos entradas, de modo que el motor pueda determinar cuándo no se requerirán las filas antiguas de una entrada (es decir, no satisfarán la restricción de tiempo) para las coincidencias con la otra entrada. Esta restricción se puede definir de una de las siguientes maneras: a. Condiciones de combinación de rango de tiempo (p. ej., condición de combinación = "hora izquierda ENTRE la hora derecha Y la hora derecha + INTERVALO 1 HORA")
- b. Únase en ventanas de tiempo de eventos (por ejemplo, condición de unión = "leftTimeWindow = ventana de tiempo derecha")

En nuestro caso de uso de publicidad, nuestro código de unión interno se volverá un poco más complicado:

```

# En Python
# Definir marcas de
agua impresionesConMarcaDeAgua =
    (impresiones .selectExpr("adId AS impresiónAdId", "tiempo de
    impresión") .withWatermark("tiempo de impresión", "2 horas"))

clicksWithWatermark =
    (clics .selectExpr("adId AS clickAdId",
    "clickTime") .withWatermark("clickTime", "3 hours"))

# Unión interna con condiciones de rango de
tiempo (impresionesConMarcaDeAgua.join(clicsConMarcaDeAgua,
expr("""
    clickAdId = impresiónAdId Y clickTime
    ENTRE impresionTime Y impresionTime + intervalo 1 hora"""))

// En escala
// Definir marcas de agua
val impresionesConMarcaDeAgua = impresiones
    .selectExpr("adId AS impresiónAdId", "tiempo de
    impresión") .withWatermark("tiempo de impresión", "2 horas ")

val clicksWithWatermark = clics
    .selectExpr("adId AS clickAdId",
    "clickTime") .withWatermark("clickTime", "3 horas")

// Unión interna con condiciones de rango de
tiempo impresionesConMarcaDeAgua.join(clicsConMarcaDeAgua,
expr("""
    clickAdId = impresiónAdId Y clickTime
    ENTRE impresionTime Y impresionTime + intervalo 1 hora"""))

```

Con estas limitaciones de tiempo para cada evento, el motor de procesamiento puede calcular automáticamente cuánto tiempo deben almacenarse en búfer los eventos para generar resultados correctos y cuándo se pueden eliminar los eventos del estado. Por ejemplo, evaluará lo siguiente (ilustrado en la [Figura 8-13](#)):

- Las impresiones deben almacenarse en el búfer durante un máximo de cuatro horas (en el tiempo del evento), ya que un clic de tres horas de retraso puede coincidir con una impresión realizada hace cuatro horas (es decir, tres horas de retraso + hasta una hora de retraso entre la impresión y haga clic). • Por el contrario, los clics deben almacenarse en el búfer durante un máximo de dos horas (en el tiempo del evento), ya que una impresión de dos horas de retraso puede coincidir con un clic recibido hace dos horas.

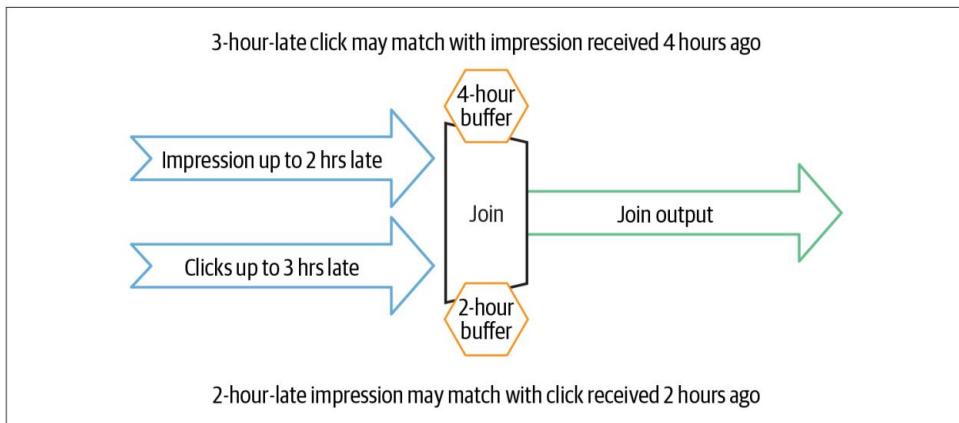


Figura 8-13. La transmisión estructurada calcula automáticamente los umbrales para la limpieza del estado utilizando retrasos de marca de agua y condiciones de rango de tiempo

Hay algunos puntos clave para recordar acerca de las uniones internas:

- Para uniones internas, especificar marcas de agua y restricciones de tiempo de evento son opcionales. En otras palabras, a riesgo de un estado potencialmente ilimitado, puede optar por no especificarlos. Solo cuando se especifican ambos obtendrá la limpieza del estado.
- Similar a las garantías proporcionadas por la marca de agua en las agregaciones, una demora de marca de agua de dos horas garantiza que el motor nunca dejará de funcionar o no coincidirá con los datos que tengan menos de dos horas de retraso, pero los datos con retraso de más de dos horas pueden o pueden no ser procesado.

Combinaciones externas con

marca de agua La combinación interna anterior generará solo aquellos anuncios para los que se hayan recibido ambos eventos. En otras palabras, los anuncios que no recibieron clics no se informarán en absoluto. En su lugar, es posible que desee que se informen todas las impresiones de anuncios, con o sin los datos de clics asociados, para permitir un análisis adicional más adelante (por ejemplo, tasas de clics). Esto nos lleva a las uniones externas de flujo a flujo. Todo lo que necesita hacer para implementar esto es especificar el tipo de unión externa:

```
# En Python #
Combinación externa izquierda con condiciones de intervalo de
tiempo (impresionesConMarcaDeAgua.join(clicsConMarcaDeAgua,
expr"""
clickAdd = impresionAdd Y clickTime ENTRE
impresionTime Y impresionTime + intervalo 1 hora""", "leftOuter")) # solo cambio: establece el tipo de combinación
externa

// En Scala
// Combinación externa izquierda con condiciones de rango de tiempo
```

```

impresionesConMarcaDeAgua.join(clicsConMarcaDeAgua,
expr"""
clickAdId = impresionAdId Y clickTime
ENTRE impresionTime Y impresionTime + intervalo 1 hora"""),
"leftOuter" // Solo cambio:
establezca el tipo de unión externa

```

Como se esperaba de las uniones externas, esta consulta comenzará a generar resultados para cada impresión, con o sin (es decir, usando valores NULL) los datos de clic. Sin embargo, hay algunos puntos adicionales a tener en cuenta sobre las uniones externas:

- A diferencia de las uniones internas, el retraso de la marca de agua y las restricciones de tiempo de evento no son opcionales para las uniones externas. Esto se debe a que, para generar los resultados NULL, el motor debe saber cuándo un evento no va a coincidir con nada más en el futuro. Para obtener resultados correctos de combinación externa y limpieza de estado, se deben especificar las restricciones de marca de agua y tiempo de evento.
- En consecuencia, los resultados NULL externos se generarán con un retraso ya que el motor tiene que esperar un tiempo para asegurarse de que no haya ni haya coincidencias. Este retraso es el tiempo máximo de almacenamiento en búfer (con respecto al tiempo del evento) calculado por el motor para cada evento como se explicó en la sección anterior (es decir, cuatro horas para impresiones y dos horas para clics).

Cálculos arbitrarios con estado

Muchos casos de uso requieren una lógica más complicada que las operaciones SQL que hemos discutido hasta ahora. Por ejemplo, supongamos que desea realizar un seguimiento de los estados (p. ej., iniciado sesión, ocupado, inactivo) de los usuarios mediante el seguimiento de sus actividades (p. ej., clics) en tiempo real. Para construir esta canalización de procesamiento de transmisión, deberá realizar un seguimiento del historial de actividad de cada usuario como un estado con una estructura de datos arbitraria y aplicar continuamente cambios arbitrariamente complejos en la estructura de datos en función de las acciones del usuario. La operación `mapGroupsWithState()` y su equivalente más flexible `flatMapGroupsWithState()` están diseñadas para casos de uso analítico tan complejos.



A partir de Spark 3.0, estas dos operaciones solo están disponibles en Scala y Java.

En esta sección, comenzaremos con un ejemplo simple con `mapGroupsWithState()` para ilustrar los cuatro pasos clave para modelar datos de estado personalizados y definir operaciones personalizadas en ellos. Luego discutiremos el concepto de tiempos de espera y cómo puede usarlos para caducar el estado que no se ha actualizado por un tiempo. Terminaremos con `flatMapGroupsWithState()`, que le brinda aún más flexibilidad.

Modelado de operaciones arbitrarias con estado con mapGroupsWithState()

El estado con un esquema arbitrario y transformaciones arbitrarias en el estado se modela como una función definida por el usuario que toma la versión anterior del valor del estado y los nuevos datos como entradas, y genera el estado actualizado y el resultado calculado como salidas. Programáticamente en Scala, tendrá que definir una función con la siguiente firma (K, V, S y U son tipos de datos, como se explica en breve):

```
// En Scala
def arbitraryStateUpdateFunction( clave:
    K, newDataForKey: Iterator[V],
    previousStateForKey: GroupState[S]
```

): tu

Esta función se proporciona a una consulta de transmisión mediante las operaciones groupByKey() y mapGroupsWithState(), de la siguiente manera:

```
// En Scala
val inputDataset: Dataset[V] = // input streaming Dataset
```

inputDataset .groupByKey(keyFunction) // keyFunction() genera la clave a partir de la
entrada .mapGroupsWithState(arbitraryStateUpdateFunction)

Cuando se inicia esta consulta de transmisión, en cada microlote, Spark llamará a esta arbitraryStateUpdateFunction() para cada clave única en los datos del microlote.

Echemos un vistazo más de cerca a cuáles son los parámetros y qué valores de parámetros Spark llamará a la función con:

clave:

KK es el tipo de datos de las claves comunes definidas en el estado y la entrada. Spark llamará a esta función para cada clave única en los datos.

newDataForKey: Iterador[V]

V es el tipo de datos del conjunto de datos de entrada. Cuando Spark llama a esta función para una tecla, este parámetro tendrá todos los nuevos datos de entrada correspondientes a esa tecla. Tenga en cuenta que el orden en que los objetos de datos de entrada estarán presentes en el iterador no está definido.

anteriorEstadoParaClave: EstadoGrupo[S]

S es el tipo de datos del estado arbitrario que va a mantener, y Group State[S] es un objeto contenedor con tipo que proporciona métodos para acceder y administrar el valor del estado. Cuando Spark llama a esta función para una clave, este objeto proporcionará el valor de estado establecido la vez anterior que Spark llamó a esta función para esa clave (es decir, para uno de los microlotes anteriores).

U es el tipo de datos de la salida de la función.



Hay un par de parámetros adicionales que debe proporcionar. Todos los tipos (K, V, S, U) deben ser codificables por los codificadores de Spark SQL. En consecuencia, en `mapGroupsWithState()`, debe proporcionar los codificadores escritos para S y U , ya sea implícitamente en Scala o explícitamente en Java. Consulte “[Codificadores de conjuntos de datos](#)” en la [página 168](#) en el [Capítulo 6](#) para obtener más detalles.

Examinemos cómo expresar la función de actualización de estado deseada en este formato con un ejemplo. Digamos que queremos comprender el comportamiento del usuario en función de sus acciones. Conceptualmente, es bastante simple: en cada micro-lote, para cada usuario activo, usaremos las nuevas acciones realizadas por el usuario y actualizaremos el "estado" del usuario. Mediante programación, podemos definir la función de actualización de estado con los siguientes pasos:

1. Defina los tipos de datos. Necesitamos definir los tipos exactos de K, V, S y U. En este caso, usaremos lo siguiente: a.

Datos de entrada (V) = clase de caso `UserAction(userId: String, action: Cuerda)`

- b. Claves (K) = Cadena (es decir, el ID de usuario)
- c. Estado (S) = clase de caso `UserStatus (ID de usuario: cadena, activo: booleano)`
- d. Salida (U) = `UserStatus`, ya que queremos generar el último estado del usuario

Tenga en cuenta que todos estos tipos de datos son compatibles con los codificadores.

2. Defina la función. Según los tipos elegidos, traduzcamos la idea conceptual en código. Cuando se llama a esta función con nuevas acciones de usuario, hay dos situaciones principales que debemos manejar: si existe un estado anterior (es decir, estado de usuario anterior) para esa clave (es decir, ID de usuario) o no. En consecuencia, inicializaremos el estado del usuario o actualizaremos el estado existente con las nuevas acciones. Actualizaremos explícitamente el estado con el nuevo conteo acumulado y, finalmente, devolveremos el par actualizado `userId-userStatus` :

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

def updateUserStatus( userId: String, newActions: Iterator[UserAction], state: GroupState[UserStatus]): UserStatus = {

  val userStatus = state.getOption.getOrElse(
    nuevo estado de usuario (ID de usuario, falso)
  )
  ...
}
```

```

} newActions.foreach { action =>
    userStatus.updateWith(acción)

} state.update(userStatus)
devuelve el estado del usuario
}

```

3. Aplicar la función sobre las acciones. Agruparemos el conjunto de datos de acciones de entrada usando groupByKey() y luego aplicaremos la función updateUserStatus usando mapGroupsWithState():

```
// En Scala
val userActions: Dataset[UserAction] = val ...
lastStatuses = userActions .groupByKey(userAction
    => userAction.userId) .mapGroupsWithState(updateUserStatus
    _)
```

Una vez que comenzemos esta consulta de transmisión con la salida de la consola, veremos que se imprimen los estados de usuario actualizados.

Antes de pasar a temas más avanzados, hay algunos puntos notables para recordar:

- Cuando se llama a la función, no hay un orden bien definido para los registros de entrada en el nuevo iterador de datos (por ejemplo, newActions). Si necesita actualizar el estado con los registros de entrada en un orden específico (p. ej., en el orden en que se realizaron las acciones), entonces debe reordenarlos explícitamente (p. ej., según la marca de tiempo del evento o algún otro orden). IDENTIFICACIÓN). De hecho, si existe la posibilidad de que las acciones se lean fuera de orden desde la fuente, debe considerar la posibilidad de que un microlote futuro pueda recibir datos que deberían procesarse antes que los datos del lote actual. En ese caso, debe almacenar en búfer los registros como parte del estado.
- En un microlote, la función se llama una vez en una clave solo si el microlote tiene datos para esa clave. Por ejemplo, si un usuario se vuelve inactivo y no proporciona nuevas acciones durante mucho tiempo, entonces, de forma predeterminada, la función no se llamará durante mucho tiempo. Si desea actualizar o eliminar el estado en función de la inactividad de un usuario durante un período prolongado, debe usar tiempos de espera, que analizaremos en la siguiente sección.
- El motor de procesamiento incremental supone que la salida de mapGroupsWithState() son registros de clave/valor actualizados continuamente, de forma similar a la salida de las agregaciones. Esto limita qué operaciones se admiten en la consulta después de mapGroupsWithState() y qué sumideros se admiten. Por ejemplo, no se admite agregar la salida a archivos. Si desea aplicar un estado arbitrario

operaciones con mayor flexibilidad, entonces debe usar flatMapGroupsWith State(). Discutiremos eso después de los tiempos de espera.

Uso de tiempos de espera para administrar grupos inactivos

En el ejemplo anterior de seguimiento de sesiones de usuarios activos, a medida que más usuarios se activan, la cantidad de claves en el estado seguirá aumentando, al igual que la memoria utilizada por el estado. Ahora, en un escenario del mundo real, es probable que los usuarios no permanezcan activos todo el tiempo. Puede que no sea muy útil mantener el estado de los usuarios inactivos en el estado, ya que no volverá a cambiar hasta que esos usuarios vuelvan a estar activos. Por lo tanto, es posible que deseemos eliminar explícitamente toda la información de los usuarios inactivos. Sin embargo, un usuario no puede realizar explícitamente ninguna acción para volverse inactivo (por ejemplo, cerrar sesión explícitamente), y es posible que debamos definir la inactividad como la falta de cualquier acción durante un umbral de duración. Esto se vuelve complicado de codificar en la función, ya que la función no se llama para un usuario hasta que haya nuevas acciones de ese usuario.

Para codificar la inactividad basada en el tiempo, mapGroupsWithState() admite tiempos de espera que se definen de la siguiente manera:

- Cada vez que se llama a la función en una tecla, se puede establecer un tiempo de espera en la tecla en función de una duración o una marca de tiempo de umbral. • Si esa clave no recibe ningún dato, por lo que se cumple la condición de tiempo de espera, la clave se marca como "tiempo de espera agotado". El siguiente microlote llamará a la función en esta clave con tiempo de espera agotado incluso si no hay datos para esa clave en ese microlote. En esta llamada de función especial, el nuevo iterador de datos de entrada estará vacío (ya que no hay datos nuevos) y GroupState.hasTimedOut() devolverá verdadero. Esta es la mejor manera de identificar dentro de la función si la llamada se debió a nuevos datos o a un tiempo de espera.

Hay dos tipos de tiempos de espera, basados en nuestras dos nociones de tiempo: tiempo de procesamiento y tiempo de evento. El tiempo de espera del tiempo de procesamiento es el más simple de usar de los dos, por lo que comenzaremos con eso.

Tiempos de espera de tiempo

de procesamiento Los tiempos de espera de tiempo de procesamiento se basan en la hora del sistema (también conocida como la hora del reloj de pared) de la máquina que ejecuta la consulta de transmisión y se definen de la siguiente manera: si una clave recibió datos por última vez en la marca de tiempo del sistema T, y la hora actual la marca de tiempo es mayor que ($T + <\text{tiempo de espera}>$), entonces la función se volverá a llamar con un nuevo iterador de datos vacío.

Investigaremos cómo usar los tiempos de espera actualizando nuestro ejemplo de usuario para eliminar el estado de un usuario en función de una hora de inactividad. Haremos tres cambios:

- En `mapGroupsWithState()`, especificaremos el tiempo de espera como `GroupStateTimeout.out.ProcessingTimeTimeout`.
- En la función de actualización de estado, antes de actualizar el estado con nuevos datos, tenemos que verificar si el estado ha expirado o no. En consecuencia, actualizaremos o eliminaremos el estado.
- Además, cada vez que actualicemos el estado con nuevos datos, estableceremos el tiempo de espera duración.

Aquí está el código actualizado:

```
// En Scala
def updateUserStatus( userId: String, newActions: Iterator[UserAction], state: GroupState[UserStatus]): UserStatus = {

  if (!state.hasTimedOut) { val userStatus = state.getOption.getOrElse {
    nuevo estado de usuario (ID de usuario, falso)

    } newActions.foreach { action => userStatus.updateWith(action) }
    state.update(userStatus).setTimeoutDuration("1 hora") // Establecer la duración del tiempo de espera return userStatus
  }

} else {
  { val userStatus = state.get()
    state.remove() // Elimina el estado cuando se agota el tiempo de espera return userStatus.asInactive() // Devuelve el estado del usuario inactivo
  }
}

val lastStatuses = userActions
  .groupByKey (userAction =>
    userAction.userId) .mapGroupsWithState (
      GroupStateTimeout.ProcessingTimeTimeout
      ( updateUserStatus _)
    )
}
```

Esta consulta limpiará automáticamente el estado de los usuarios para los que la consulta no ha procesado ningún dato durante más de una hora. Sin embargo, hay algunos puntos a tener en cuenta sobre los tiempos de espera:

- El tiempo de espera establecido por la última llamada a la función se cancela automáticamente cuando se vuelve a llamar a la función, ya sea por los nuevos datos recibidos o por el tiempo de espera. Por lo tanto, cada vez que se llama a la función, la duración del tiempo de espera o la marca de tiempo deben establecerse explícitamente para habilitar el tiempo de espera. • Dado que los tiempos de espera se procesan durante los microlotes, el momento de su ejecución es impreciso y depende en gran medida del intervalo de activación y

tiempos de procesamiento de microlotes. Por lo tanto, no se recomienda utilizar tiempos de espera para un control de tiempo preciso.

- Mientras que los tiempos de espera de tiempo de procesamiento son fáciles de razonar, no son resistentes a las ralentizaciones y los tiempos de inactividad. Si la consulta de transmisión sufre un tiempo de inactividad de más de una hora, después del reinicio, se agotará el tiempo de espera de todas las claves en el estado porque ha pasado más de una hora desde que cada clave recibió datos. Se pueden producir tiempos de espera a gran escala similares si la consulta procesa los datos más lentamente de lo que llegan al origen (por ejemplo, si los datos llegan y se almacenan en búfer en Kafka). Por ejemplo, si el tiempo de espera es de cinco minutos, una caída repentina en la velocidad de procesamiento (o un pico en la velocidad de llegada de datos) que provoque un retraso de cinco minutos podría producir tiempos de espera falsos. Para evitar tales problemas, podemos usar un tiempo de espera de tiempo de evento, que discutiremos a continuación.

Tiempos de espera de eventos

En lugar de la hora del reloj del sistema, el tiempo de espera de un evento se basa en la hora del evento en los datos (similar a las agregaciones basadas en el tiempo) y una marca de agua definida en la hora del evento. Si una tecla está configurada con una marca de tiempo de espera específica de T (es decir, no una duración), entonces esa tecla expirará cuando la marca de agua exceda T si no se recibieron nuevos datos para esa tecla desde la última vez que se llamó a la función. . Recuerde que la marca de agua es un umbral móvil que se retrasa con respecto al tiempo máximo del evento visto mientras se procesan los datos. Por lo tanto, a diferencia del tiempo del sistema, la marca de agua avanza en el tiempo al mismo ritmo que se procesan los datos. Esto significa que (a diferencia de los tiempos de espera de tiempo de procesamiento) cualquier ralentización o tiempo de inactividad en el procesamiento de consultas no causará errores falsos.

tiempos de espera

Modifiquemos nuestro ejemplo para usar un tiempo de espera de evento. Además de los cambios que ya hicimos para usar el tiempo de espera de procesamiento, haremos los siguientes cambios:

- Defina marcas de agua en el conjunto de datos de entrada (suponga que la clase UserAction tiene un campo eventTimestamp). Recuerde que el umbral de la marca de agua representa la cantidad aceptable de tiempo durante el cual los datos de entrada pueden retrasarse y estar fuera de servicio.
- Actualice mapGroupsWithState() para usar EventTimeTimeout. • Actualizar la función para establecer la marca de tiempo del umbral en el que ocurrirá el tiempo de espera. Tenga en cuenta que los tiempos de espera de eventos no permiten establecer una duración de tiempo de espera, como los tiempos de espera de procesamiento. Discutiremos la razón de esto más adelante. En este ejemplo, calcularemos este tiempo de espera como la marca de agua actual más una hora.

Aquí está el ejemplo actualizado:

```
// En Scala
def updateUserStatus( userId: String, newActions: Iterator[UserAction], state: GroupState[UserStatus]):UserStatus = {
    if (!state.hasTimedOut) { // No se llamó debido al tiempo de espera val
        userStatus = if (state.getOption.getorElse {
            nuevo estado de usuario ()
        } newActions.foreach { action => userStatus.updateWith(action) }
        state.update(userStatus)
        // Establecer la marca de tiempo de espera en la marca de agua
        actual + 1 hora state.setTimeoutTimestamp(state.getCurrentWatermarkMs, "1
        hour") return userStatus } else { val userStatus = state.get() state.remove()
        return userStatus.asInactive() }

    }
}

val lastStatuses = userActions
    .withWatermark("eventTimestamp", "10
    minutes") .groupByKey(userAction =>

    userAction.userId) .mapGroupsWithState( GroupStateTimeout.EventTimeTimeout)( updateUserStatus _)
```

Esta consulta será mucho más robusta ante tiempos de espera falsos causados por reinicios y demoras en el procesamiento.

Aquí hay algunos puntos a tener en cuenta sobre los tiempos de espera de eventos:

- A diferencia del ejemplo anterior con tiempos de espera de tiempo de procesamiento, hemos usado `GroupState.setTimeoutTimestamp()` en lugar de `GroupState.setTimeoutDuration()`. Esto se debe a que con los tiempos de espera de tiempo de procesamiento, la duración es suficiente para calcular la marca de tiempo futura exacta (es decir, la hora actual del sistema + la duración especificada) cuando ocurrirá el tiempo de espera, pero este no es el caso para los tiempos de espera de eventos. Diferentes aplicaciones pueden querer usar diferentes estrategias para calcular la marca de tiempo del umbral. En este ejemplo, simplemente lo calculamos en función de la marca de agua actual, pero una aplicación diferente puede optar por calcular la marca de tiempo de tiempo de espera de una clave en función de la marca de tiempo de tiempo de evento máxima vista para esa clave (seguida y guardada como parte del estado) .
- La marca de tiempo de tiempo de espera debe establecerse en un valor mayor que la marca de agua actual. Esto se debe a que se espera que se agote el tiempo de espera cuando la marca de tiempo cruce la marca de agua, por lo que no es lógico establecer la marca de tiempo en un valor que ya sea mayor que la marca de agua actual.

Antes de pasar de los tiempos de espera, una última cosa que debe recordar es que puede usar estos mecanismos de tiempo de espera para un procesamiento más creativo que los tiempos de espera de duración fija. Por ejemplo, puede implementar una tarea aproximadamente periódica (digamos, cada hora) en el estado guardando la marca de tiempo de ejecución de la última tarea en el estado y usándola para establecer la duración del tiempo de espera del tiempo de procesamiento, como se muestra en este fragmento de código:

```
// En Scala
timeoutDurationMs = lastTaskTimestampMs + periodIntervalMs -
groupState.getCurrentProcessingTimeMs()
```

Generalización con atMapGroupsWithState()

Hay dos limitaciones clave con mapGroupsWithState() que pueden limitar la flexibilidad que queremos implementar en casos de uso más complejos (por ejemplo, sesionizaciones encadenadas):

- Cada vez que se llama a mapGroupsWithState() , debe devolver uno y solo un registro. Para algunas aplicaciones, en algunos activadores, es posible que no desee generar nada en absoluto.
- Con mapGroupsWithState(), debido a la falta de más información sobre la función de actualización de estado opaco, el motor asume que los registros generados son pares de datos clave/valor actualizados. En consecuencia, razona sobre las operaciones aguas abajo y permite o desautoriza algunas de ellas. Por ejemplo, el DataFrame generado usando mapGroupsWithState() no se puede escribir en el modo de adición a los archivos.

Sin embargo, es posible que algunas aplicaciones deseen generar registros que puedan considerarse anexos.

flatMapGroupsWithState() supera estas limitaciones, a costa de una sintaxis un poco más compleja. Tiene dos diferencias con mapGroupsWithState():

- El tipo de devolución es un iterador, en lugar de un único objeto. Esto permite que la función para devolver cualquier número de registros o, si es necesario, ningún registro en absoluto.
- Toma otro parámetro, llamado modo de salida del operador (que no debe confundirse con los modos de salida de consulta que discutimos anteriormente en el capítulo), que define si los registros de salida son registros nuevos que se pueden agregar (Modo de salida.Anexar) o actualizar registros de clave/valor (OutputMode.Update).

Para ilustrar el uso de esta función, ampliamos nuestro ejemplo de seguimiento de usuarios (hemos eliminado los tiempos de espera para simplificar el código). Por ejemplo, si queremos generar alertas solo para ciertos cambios de usuario y queremos escribir las alertas de salida en archivos, podemos hacer lo siguiente:

```
// En Scala
def getUserAlerts( userId:
  String, newActions:
  Iterator[UserAction],
```

```

estado: GroupState[UserStatus]): Iterador[UserAlert] = {

  val userStatus = state.getOption.getOrElse {
    nuevo estado de usuario (ID de usuario, falso)
  }

  } newActions.foreach { action =>
    userStatus.updateWith(action) }
  state.update(userStatus)

  // Generar cualquier número de alertas
  return userStatus.generateAlerts().tolerator }

  val alertas de usuario = acciones de usuario
  .groupByKey(userAction =>
  userAction.userId) .flatMapGroupsWithState( OutputMode.Append,
  GroupStateTimeout.NoTimeout)( getUserAlerts)
}

```

La optimización del rendimiento

La transmisión estructurada utiliza el motor Spark SQL y, por lo tanto, se puede ajustar con los mismos parámetros que se analizaron para Spark SQL en los capítulos 5 y 7. Sin embargo, a diferencia de los trabajos por lotes que pueden procesar de gigabytes a terabytes de datos, los trabajos de microlotes generalmente procesan datos mucho más pequeños. volúmenes de datos. Por lo tanto, un clúster de Spark que ejecuta consultas de transmisión generalmente debe ajustarse de manera ligeramente diferente. Aquí hay algunas consideraciones a tener en cuenta:

Aprovisionamiento de recursos de clúster

Dado que los clústeres de Spark que ejecutan consultas de transmisión se ejecutarán las 24 horas del día, los 7 días de la semana, es importante aprovisionar los recursos de manera adecuada. El aprovisionamiento insuficiente de los recursos puede hacer que las consultas de transmisión se retrasen (con microlotes que tardan cada vez más), mientras que el aprovisionamiento excesivo (p. ej., núcleos asignados pero no utilizados) puede generar costos innecesarios. Además, la asignación debe realizarse en función de la naturaleza de las consultas de transmisión: las consultas sin estado generalmente necesitan más núcleos y las consultas con estado generalmente necesitan más memoria.

Número de particiones para las consultas

aleatorias Para las consultas de transmisión estructurada, el número de particiones aleatorias generalmente debe establecerse mucho más bajo que para la mayoría de las consultas por lotes: dividir demasiado el cálculo aumenta los gastos generales y reduce el rendimiento. Además, las mezclas debido a operaciones con estado tienen gastos generales de tareas significativamente más altos debido a los puntos de control.

Por lo tanto, para consultas de transmisión con operaciones con estado e intervalos de activación de unos pocos segundos a minutos, se recomienda ajustar el número de reproducción aleatoria.

particiones del valor predeterminado de 200 a un máximo de dos a tres veces el número de núcleos asignados.

Establecimiento de límites de tasa de origen para la estabilidad

Una vez que las configuraciones y los recursos asignados se han optimizado para las velocidades de datos de entrada esperadas de una consulta, es posible que los aumentos repentinos en las velocidades de datos puedan generar trabajos inesperadamente grandes y la consiguiente inestabilidad. Además del enfoque costoso del aprovisionamiento excesivo, puede protegerse contra la inestabilidad utilizando límites de tasa de origen. Establecer límites en las fuentes admitidas (p. ej., Kafka y archivos) evita que una consulta consuma demasiados datos en un solo microlote. Los datos de aumento permanecerán almacenados en el búfer en la fuente y la consulta eventualmente se pondrá al día. Sin embargo, tenga en cuenta lo siguiente:

- Establecer el límite demasiado bajo puede hacer que la consulta subutilice los recursos asignados y se quede atrás de la tasa de entrada.
- Los límites no protegen eficazmente contra aumentos sostenidos en la tasa de entrada. Mientras se mantiene la estabilidad, el volumen de datos almacenados en búfer y sin procesar crecerá indefinidamente en el origen, al igual que las latencias de un extremo a otro.

Múltiples consultas de transmisión en la misma aplicación Spark

Ejecutar múltiples consultas de transmisión en el mismo SparkContext o SparkSession puede conducir a un intercambio de recursos detallado. Sin embargo:

- La ejecución de cada consulta utiliza continuamente recursos en el controlador Spark (es decir, la JVM donde se ejecuta). Esto limita la cantidad de consultas que el controlador puede ejecutar simultáneamente. Alcanzar esos límites puede provocar un cuello de botella en la programación de tareas (es decir, subutilizar los ejecutores) o exceder los límites de memoria.
- Puede garantizar una asignación de recursos más justa entre consultas en el mismo contexto configurándolas para que se ejecuten en grupos de programadores separados. Establezca la propiedad local de subprocessos de SparkContext , spark.scheduler.pool , en un valor de cadena diferente para cada transmisión:

// En Scala //

*Ejecutar la consulta de transmisión 1 en el
programador pool1* spark.sparkContext.setLocalProperty("spark.scheduler.pool",
"pool1") df.writeStream.queryName("query1").format("parquet").start(camino1)

// Ejecute la consulta de transmisión2 en el

programador pool2 spark.sparkContext.setLocalProperty("spark.scheduler.pool",
"pool2") df.writeStream.queryName("query2").format("parquet").start(path2)

```
# En Python  
# Ejecute la consulta de transmisión 1 en el  
programador pool1 spark.sparkContext.setLocalProperty("spark.scheduler.pool",  
"pool1") df.writeStream.queryName("query1").format("parquet").start(path1)  
  
# Ejecute la consulta de transmisión2 en el  
programador pool2 spark.sparkContext.setLocalProperty("spark.scheduler.pool",  
"pool2") df.writeStream.queryName("query2").format("parquet").start(path2)
```

Resumen

Este capítulo exploró la escritura de consultas de transmisión estructurada mediante la API de DataFrame. Específicamente, discutimos:

- La filosofía central de Structured Streaming y el modelo de procesamiento de tratar los flujos de datos de entrada como tablas ilimitadas
- Los pasos clave para definir, iniciar, reiniciar y monitorear las consultas de transmisión • Cómo usar varias fuentes y sumideros de transmisión integrados y escribir sumideros de transmisión
- Cómo usar y ajustar operaciones con estado administradas, como agregaciones de transmisión y uniones de flujo a flujo. • Técnicas para expresar cálculos personalizados con estado.

Al trabajar con los fragmentos de código en el capítulo y los cuadernos en el repositorio de [GitHub del libro](#), Obtendrá una idea de cómo usar la transmisión estructurada de manera efectiva. En el próximo capítulo, exploraremos cómo puede administrar la lectura y escritura de datos estructurados simultáneamente desde cargas de trabajo por lotes y de transmisión.

CAPÍTULO 9

Creación de lagos de datos fiables con Apache Spark

En los capítulos anteriores, aprendió cómo usar Apache Spark de manera fácil y efectiva para crear canalizaciones de procesamiento de datos escalables y de alto rendimiento. Sin embargo, en la práctica, expresar la lógica de procesamiento solo resuelve la mitad del problema de extremo a extremo de construir una canalización. Para un ingeniero de datos, un científico de datos o un analista de datos, el objetivo final de crear canalizaciones es consultar los datos procesados y obtener información de ellos. La elección de la solución de almacenamiento determina la solidez y el rendimiento de extremo a extremo (es decir, desde los datos sin procesar hasta los conocimientos) de la canalización de datos.

En este capítulo, primero analizaremos las características clave de una solución de almacenamiento que debe tener en cuenta. Luego, analizaremos dos amplias clases de soluciones de almacenamiento, bases de datos y lagos de datos, y cómo usar Apache Spark con ellos. Finalmente, presentaremos la próxima ola de soluciones de almacenamiento, llamadas casas lago, y exploraremos algunos de los nuevos motores de procesamiento de código abierto en este espacio.

La importancia de una solución de almacenamiento óptima

Estas son algunas de las propiedades que se desean en una solución de almacenamiento:

Escalabilidad y rendimiento

La solución de almacenamiento debe poder escalar al volumen de datos y proporcionar el rendimiento de lectura/escritura y la latencia que requiere la carga de trabajo.

Compatibilidad con

transacciones Las cargas de trabajo complejas suelen leer y escribir datos al mismo tiempo, por lo que la compatibilidad con [transacciones ACID](#) es fundamental para garantizar la calidad de los resultados finales.

Compatibilidad con diversos formatos de datos La solución de almacenamiento debe poder almacenar datos no estructurados (p. ej., archivos de texto como registros sin procesar), datos semiestructurados (p. ej., datos JSON) y datos estructurados (p. ej., datos tabulares).

Compatibilidad con diversas cargas de trabajo

La solución de almacenamiento debe ser capaz de admitir una amplia gama de cargas de trabajo comerciales, que incluyen:

- Cargas de trabajo SQL como análisis de BI tradicional • Cargas de trabajo por lotes como trabajos ETL tradicionales que procesan datos no estructurados sin procesar • Cargas de trabajo de transmisión como monitoreo y alertas en tiempo real • Cargas de trabajo de ML e IA como recomendaciones y predicciones de abandono

Apertura La

compatibilidad con una amplia gama de cargas de trabajo a menudo requiere que los datos se almacenen en formatos de datos abiertos. Las API estándar permiten acceder a los datos desde una variedad de herramientas y motores. Esto permite que la empresa utilice las herramientas más óptimas para cada tipo de carga de trabajo y tome las mejores decisiones comerciales.

Con el tiempo, se han propuesto diferentes tipos de soluciones de almacenamiento, cada una con sus ventajas y desventajas únicas con respecto a estas propiedades. En este capítulo, exploraremos cómo evolucionaron las soluciones de almacenamiento disponibles de bases de datos a lagos de datos, y cómo usar Apache Spark con cada una de ellas. Luego centraremos nuestra atención en la próxima generación de soluciones de almacenamiento, a menudo denominadas data lakehouses, que pueden proporcionar lo mejor de ambos mundos: la escalabilidad y flexibilidad de los data lakes con las garantías transaccionales de las bases de datos.

bases de datos

Durante muchas décadas, las bases de datos han sido la solución más confiable para construir almacenes de datos para almacenar datos críticos para el negocio. En esta sección, exploraremos la arquitectura de las bases de datos y sus cargas de trabajo, y cómo usar Apache Spark para cargas de trabajo de análisis en bases de datos. Finalizaremos esta sección con una discusión sobre las limitaciones de las bases de datos para admitir cargas de trabajo modernas que no sean SQL.

Una breve introducción a las bases de datos

Las bases de datos están diseñadas para almacenar datos estructurados como tablas, que se pueden leer mediante consultas SQL. Los datos deben adherirse a un esquema estricto, lo que permite que un sistema de gestión de bases de datos cooptime en gran medida el almacenamiento y el procesamiento de datos. Es decir, acoplan estrechamente su disposición interna de los datos e índices en archivos en disco con sus motores de procesamiento de consultas altamente optimizados, lo que proporciona cálculos muy rápidos en

los datos almacenados junto con sólidas garantías ACID transaccionales en todas las operaciones de lectura/escritura.

Las cargas de trabajo de SQL en las bases de datos se pueden clasificar en términos generales en dos categorías, de la siguiente manera:

Cargas de trabajo de procesamiento de transacciones en línea

(OLTP) Al igual que las transacciones de cuentas bancarias, las cargas de trabajo de OLTP suelen ser consultas simples de alta simultaneidad y baja latencia que leen o actualizan algunos registros a la vez.

Procesamiento analítico en línea (OLAP)

Las cargas de trabajo de OLAP, como los informes periódicos, suelen ser consultas complejas (que implican agregados y uniones) que requieren análisis de alto rendimiento en muchos registros.

Es importante tener en cuenta que Apache Spark es un motor de consulta diseñado principalmente para cargas de trabajo OLAP, no para cargas de trabajo OLTP. Por lo tanto, en el resto del capítulo vamos a centrar nuestra discusión en soluciones de almacenamiento para cargas de trabajo analíticas. A continuación, veamos cómo se puede usar Apache Spark para leer y escribir en bases de datos.

Ler y escribir en bases de datos con Apache Spark

Gracias al ecosistema de conectores en constante crecimiento, Apache Spark puede conectarse a una amplia variedad de bases de datos para leer y escribir datos. Para las bases de datos que tienen controladores JDBC (p. ej., PostgreSQL, MySQL), puede utilizar la fuente de datos JDBC integrada junto con los archivos jar de controladores JDBC adecuados para acceder a los datos. Para muchas otras bases de datos modernas (por ejemplo, Azure Cosmos DB, Snowflake), hay conectores dedicados que puede invocar con el nombre de formato adecuado. Varios ejemplos se analizaron en detalle en el [Capítulo 5](#). Esto hace que sea muy fácil aumentar sus almacenes de datos y bases de datos con cargas de trabajo y casos de uso basados en Apache Spark.

Limitaciones de las bases de datos

Desde el siglo pasado, las bases de datos y las consultas SQL han sido conocidas como excelentes soluciones de construcción para las cargas de trabajo de BI. Sin embargo, la última década ha visto dos nuevas tendencias importantes en las cargas de trabajo analítico:

Crecimiento en el tamaño

de los datos Con la llegada de Big Data, ha habido una tendencia global en la industria para medir y recopilar todo (vistas de página, clics, etc.) para comprender las tendencias y los comportamientos de los usuarios. Como resultado, la cantidad de datos recopilados por cualquier empresa u organización ha aumentado de gigabytes hace un par de décadas a terabytes y petabytes en la actualidad.

Crecimiento en la diversidad de análisis Junto

con el aumento en la recopilación de datos, existe la necesidad de conocimientos más profundos.

Esto ha llevado a un crecimiento explosivo de análisis complejos como el aprendizaje automático y el aprendizaje profundo.

Se ha demostrado que las bases de datos son bastante inadecuadas para adaptarse a estas nuevas tendencias, debido a las siguientes limitaciones:

El escalado horizontal de las bases de datos es

extremadamente costoso Aunque las bases de datos son extremadamente eficientes en el procesamiento de datos en una sola máquina, la tasa de crecimiento de los volúmenes de datos ha superado con creces el crecimiento de las capacidades de rendimiento de una sola máquina. La única forma de avanzar para los motores de procesamiento es escalar horizontalmente, es decir, usar varias máquinas para procesar datos en paralelo. Sin embargo, la mayoría de las bases de datos, especialmente las de código abierto, no están diseñadas para escalar horizontalmente para realizar un procesamiento distribuido. Las pocas soluciones de bases de datos industriales que pueden mantenerse al día con los requisitos de procesamiento de forma remota tienden a ser soluciones propietarias que se ejecutan en hardware especializado y, por lo tanto, son muy costosas de adquirir y mantener.

Las bases de datos no admiten muy bien los análisis que no están basados en

SQL. Las bases de datos almacenan datos en formatos complejos (a menudo patentados) que normalmente están muy optimizados para que solo los lea el motor de procesamiento SQL de esa base de datos. Esto significa que otras herramientas de procesamiento, como el aprendizaje automático y los sistemas de aprendizaje profundo, no pueden acceder de manera eficiente a los datos (excepto leyendo de manera ineficiente todos los datos de la base de datos). Las bases de datos tampoco se pueden ampliar fácilmente para realizar análisis no basados en SQL, como el aprendizaje automático.

Estas limitaciones de las bases de datos llevaron al desarrollo de un enfoque completamente diferente para almacenar datos, conocido como lagos de datos.

Lagos de datos

A diferencia de la mayoría de las bases de datos, un lago de datos es una solución de almacenamiento distribuido que se ejecuta en hardware básico y se escala horizontalmente con facilidad. En esta sección, comenzaremos con una discusión sobre cómo los lagos de datos satisfacen los requisitos de las cargas de trabajo modernas, luego veremos cómo Apache Spark se integra con los lagos de datos para hacer que las cargas de trabajo se escalen a datos de cualquier tamaño. Finalmente, exploraremos el impacto de los sacrificios arquitectónicos realizados por los lagos de datos para lograr la escalabilidad.

Una breve introducción a los lagos de datos

La arquitectura del lago de datos, a diferencia de las bases de datos, desacopla el sistema de almacenamiento distribuido del sistema informático distribuido. Esto permite que cada sistema se escala según lo necesite la carga de trabajo. Además, los datos se guardan como archivos con formatos abiertos, de modo que cualquier motor de procesamiento puede leerlos y escribirlos utilizando API estándar. Esta idea fue popularizada a fines de la década de 2000 por Hadoop File System (HDFS) del [proyecto Apache Hadoop](#), que a su vez se inspiró en gran medida en el trabajo de investigación "El sistema de archivos de Google" de Sanjay Ghemawat, Howard Gobioff y Shun-Tak Leung.

Las organizaciones crean sus lagos de datos eligiendo de forma independiente lo siguiente:

Sistema de

almacenamiento Eligen ejecutar HDFS en un clúster de máquinas o utilizar cualquier almacén de objetos en la nube (p. ej., AWS S3, Azure Data Lake Storage o Google Cloud Storage).

Formato de

archivo Según las cargas de trabajo posteriores, los datos se almacenan como archivos en formatos estructurados (p. ej., Parquet, ORC), semiestructurados (p. ej., JSON) o, a veces, incluso no estructurados (p. ej., texto, imágenes, audio, video) .

Motor(es) de procesamiento

Nuevamente, dependiendo de los tipos de cargas de trabajo analítico que se realizarán, se elige un motor de procesamiento. Puede ser un motor de procesamiento por lotes (p. ej., Spark, Presto, Apache Hive), un motor de procesamiento de secuencias (p. ej., Spark, Apache Flink) o una biblioteca de aprendizaje automático (p. ej., Spark MLlib, scikit-learn, R).

Esta flexibilidad, la capacidad de elegir el sistema de almacenamiento, el formato de datos abiertos y el motor de procesamiento que mejor se adapte a la carga de trabajo en cuestión, es la mayor ventaja de los lagos de datos sobre las bases de datos. En general, para las mismas características de rendimiento, los lagos de datos suelen proporcionar una solución mucho más económica que las bases de datos. Esta ventaja clave ha llevado al crecimiento explosivo del ecosistema de big data. En la siguiente sección, analizaremos cómo puede usar Apache Spark para leer y escribir formatos de archivo comunes en cualquier sistema de almacenamiento.

Lectura y escritura en lagos de datos con Apache Spark

Apache Spark es uno de los mejores motores de procesamiento para crear su propio lago de datos, ya que proporciona todas las características clave que requieren:

Compatibilidad con diversas cargas de trabajo

Spark proporciona todas las herramientas necesarias para manejar una amplia gama de cargas de trabajo, incluido el procesamiento por lotes, las operaciones ETL, las cargas de trabajo de SQL mediante Spark SQL, el procesamiento de flujo mediante transmisión estructurada (discutido en el [Capítulo 8](#)) y el aprendizaje automático mediante MLlib (discutido en el [Capítulo 10](#)) , Entre muchos otros.

Soporte para diversos formatos de archivo

En el [Capítulo 4](#), exploramos en detalle cómo Spark tiene soporte integrado para formatos de archivo no estructurados, semiestructurados y estructurados.

Soporte para diversos sistemas de archivos.

Spark admite el acceso a datos desde cualquier sistema de almacenamiento que admita las API del sistema de archivos de Hadoop. Dado que esta API se ha convertido en el estándar de facto en el ecosistema de big data, la mayoría de los sistemas de almacenamiento locales y en la nube proporcionan implementaciones para ella, lo que significa que Spark puede leer y escribir en la mayoría de los sistemas de almacenamiento.

Sin embargo, para muchos sistemas de archivos (especialmente aquellos basados en almacenamiento en la nube, como AWS S3), debe configurar Spark para que pueda acceder al sistema de archivos de manera segura. Además, los sistemas de almacenamiento en la nube a menudo no tienen la misma semántica de operación de archivos que se espera de un sistema de archivos estándar (por ejemplo, coherencia eventual en S3), lo que puede generar resultados inconsistentes si no configura Spark en consecuencia. Consulte la [documentación sobre la integración en la nube](#) para detalles.

Limitaciones de los lagos de datos

Los lagos de datos no están exentos de fallas, la más notoria de las cuales es la falta de garantías transaccionales. Específicamente, los lagos de datos no brindan garantías ACID sobre:

Atomicidad y aislamiento Los

motores de procesamiento escriben datos en lagos de datos como muchos archivos de forma distribuida. Si la operación falla, no hay ningún mecanismo para revertir los archivos ya escritos, lo que deja datos potencialmente dañados (el problema se agrava cuando las cargas de trabajo simultáneas modifican los datos porque es muy difícil proporcionar aislamiento entre archivos sin un nivel superior). mecanismos).

Coherencia La

falta de atomicidad en las escrituras fallidas hace que los lectores obtengan una vista inconsistente de los datos. De hecho, es difícil garantizar la calidad de los datos incluso en datos escritos con éxito. Por ejemplo, un problema muy común con los lagos de datos es escribir accidentalmente archivos de datos en un formato y esquema inconsistente con los datos existentes.

Para evitar estas limitaciones de los lagos de datos, los desarrolladores emplean todo tipo de trucos.

Aquí están algunos ejemplos:

- Las grandes colecciones de archivos de datos en los lagos de datos a menudo se “dividen” en subdirectorios basados en el valor de una columna (por ejemplo, una gran tabla de Hive con formato Parquet dividida por fecha). Para lograr modificaciones atómicas de los datos existentes, a menudo se reescriben subdirectorios completos (es decir, se escriben en un directorio temporal y luego se intercambian las referencias) solo para actualizar o eliminar algunos registros. • Las programaciones de los trabajos de actualización de datos (p. ej., trabajos diarios de ETL) y los trabajos de consulta de datos (p. ej., trabajos de informes diarios) a menudo se escalonan para evitar el acceso simultáneo a los datos y las incoherencias causadas por ello.

Los intentos de eliminar estos problemas prácticos han llevado al desarrollo de nuevos sistemas, como las casas de los lagos.

Lakehouses: el siguiente paso en la evolución de

Soluciones de almacenamiento

Lakehouse es un nuevo paradigma que combina los mejores elementos de lagos de datos y almacenes de datos para cargas de trabajo OLAP. Los Lakehouses están habilitados por un nuevo diseño de sistema que proporciona funciones de administración de datos similares a las bases de datos directamente en el almacenamiento escalable y de bajo costo que se usa para los lagos de datos. Más específicamente, proporcionan las siguientes características:

Compatibilidad con transacciones

Al igual que las bases de datos, las casas lago proporcionan garantías ACID en presencia de cargas de trabajo simultáneas.

Cumplimiento y gobernanza del esquema

Lakehouses evita que los datos con un esquema incorrecto se inserten en una tabla y, cuando sea necesario, el esquema de la tabla puede evolucionar explícitamente para adaptarse a los datos en constante cambio. El sistema debe ser capaz de razonar sobre la integridad de los datos y debe tener mecanismos sólidos de gobernanza y auditoría.

Compatibilidad con diversos tipos de datos en formatos abiertos

A diferencia de las bases de datos, pero similares a los lagos de datos, las casas lago pueden almacenar, refinar, analizar y acceder a todos los tipos de datos necesarios para muchas aplicaciones de datos nuevos, ya sean estructurados, semiestructurados o no estructurados. Para permitir que una amplia variedad de herramientas accedan a ellos de manera directa y eficiente, los datos deben almacenarse en formatos abiertos con API estandarizadas para leerlos y escribirlos.

Compatibilidad con diversas cargas de trabajo

Impulsados por la variedad de herramientas que leen datos mediante API abiertas, los lagos permiten diversas cargas de trabajo para operar con datos en un solo repositorio. Desglosar silos de datos aislados (es decir, múltiples repositorios para diferentes categorías de datos) permite a los desarrolladores crear más fácilmente soluciones de datos diversas y complejas, desde SQL tradicional y análisis de transmisión hasta aprendizaje automático.

Compatibilidad con upserts y deletes

Casos de uso complejos como [cambio de captura de datos \(CDC\)](#) y [dimensión de cambio lento \(SCD\)](#). Las operaciones requieren que los datos de las tablas se actualicen continuamente. Lakehouse permite que los datos se eliminen y actualicen simultáneamente con garantías transaccionales.

Los Lakehouses de

gobierno de datos proporcionan las herramientas con las que puede razonar sobre la integridad de los datos y auditar todos los cambios de datos para el cumplimiento de las políticas.

Actualmente, existen algunos sistemas de código abierto, como Apache Hudi, Apache Iceberg y Delta Lake, que se pueden usar para construir lagos con estas propiedades.

alto nivel, los tres proyectos tienen una arquitectura similar inspirada en principios de bases de datos bien conocidos. Todos son formatos abiertos de almacenamiento de datos que hacen lo siguiente:

- Almacenar grandes volúmenes de datos en formatos de archivo estructurados en sistemas de archivos escalables.
- Mantener un registro de transacciones para registrar una línea de tiempo de cambios atómicos en los datos
(Al igual que las bases de datos).
- Utilice el registro para definir versiones de los datos de la tabla y proporcione garantías de aislamiento de instantáneas entre lectores y escritores.
- Soporte de lectura y escritura en tablas usando Apache Spark.

Dentro de estos grandes trazos, cada proyecto tiene características únicas en términos de API, rendimiento y nivel de integración con las API de origen de datos de Apache Spark. Los exploraremos a continuación. Tenga en cuenta que todos estos proyectos están evolucionando rápidamente y, por lo tanto, algunas de las descripciones pueden estar desactualizadas en el momento en que las está leyendo. Consulte la documentación en línea de cada proyecto para obtener la información más actualizada.

Apache Hudi

Inicialmente construido por [Uber Engineering](#), [Apache Hudi](#), un Acrónimo de Hadoop Update Delete and Incremental: es un formato de almacenamiento de datos diseñado para upserts y eliminaciones incrementales sobre datos de estilo clave/valor. Los datos se almacenan como una combinación de formatos de columnas (p. ej., archivos Parquet) y formatos basados en filas (p. ej., archivos Avro para registrar cambios incrementales sobre archivos Parquet). Además de las características comunes mencionadas anteriormente, admite:

- Upserting con indexación rápida y conectable • Publicación
atómica de datos con soporte de reversión • Lectura de cambios
incrementales en una tabla • Puntos de guardado para recuperación de
datos • Tamaño de archivo y administración de diseño mediante estadísticas
- Compactación asíncrona de datos de filas y columnas

Apache Iceberg Construido

originalmente en [Netflix](#), [iceberg apache](#) es otro formato de almacenamiento abierto para grandes conjuntos de datos. Sin embargo, a diferencia de Hudi, que se enfoca en alterar los datos clave/valor, Iceberg se enfoca más en el almacenamiento de datos de uso general que escala a petabytes en una sola tabla y tiene propiedades de evolución de esquema. En concreto, proporciona las siguientes características adicionales (además de las comunes):

- Evolución del esquema agregando, quitando, actualizando, renombrando y reordenando columnas, campos y/o estructuras anidadas
- Partición oculta, que bajo las cubiertas crea los valores de partición para las filas en una mesa
- Evolución de la partición, donde automáticamente realiza una operación de metadatos para actualizar el diseño de la tabla a medida que cambia el volumen de datos o los patrones de consulta
- Viaje en el tiempo, que le permite consultar una instantánea de tabla específica por ID o por marca de tiempo
- Retroceder a versiones anteriores para corregir errores • Aislamiento serializable, incluso entre múltiples escritores simultáneos

Iago delta

Iago delta es un proyecto de código abierto alojado por Linux Foundation, construido por los creadores originales de Apache Spark. Al igual que los demás, es un formato de almacenamiento de datos abierto que proporciona garantías transaccionales y permite la aplicación y evolución del esquema. También proporciona varias otras características interesantes, algunas de las cuales son únicas.

Delta Lake apoya:

- Transmisión de lectura y escritura en tablas mediante fuentes de transmisión estructurada y se hunde
- Operaciones de actualización, eliminación y combinación (para upserts), incluso en Java, Scala y API de Python
- Evolución del esquema, ya sea alterando explícitamente el esquema de la tabla o fusionando implícitamente el esquema de un DataFrame con el de la tabla durante la escritura del DataFrame. (De hecho, la operación de combinación en Delta Lake admite sintaxis avanzada para actualizaciones/inserciones/eliminaciones condicionales, actualización de todas las columnas juntas, etc., como verá más adelante en este capítulo).
- Viaje en el tiempo, que le permite consultar una instantánea de tabla específica por ID o por marca de tiempo
- Retroceder a versiones anteriores para corregir errores • Aislamiento serializable entre múltiples escritores simultáneos que ejecutan cualquier SQL, operaciones por lotes o de transmisión

En el resto de este capítulo, exploraremos cómo se puede usar un sistema de este tipo, junto con Apache Spark, para construir un lago que proporcione las propiedades antes mencionadas. De estos tres sistemas, hasta ahora, Delta Lake tiene la integración más estrecha con las fuentes de datos de Apache Spark (tanto para cargas de trabajo por lotes como de transmisión) y operaciones SQL (p. ej., MERGE). Por lo tanto, utilizaremos Delta Lake como vehículo para una mayor exploración.



Este proyecto se llama Delta Lake por su analogía con el streaming. Los arroyos fluyen hacia el mar para crear deltas: aquí es donde se acumulan todos los sedimentos y, por lo tanto, donde se cultivan los cultivos valiosos. ¡A Jules S. Damji (uno de nuestros coautores) se le ocurrió esto!

Construcción de lagos con Apache Spark y Delta Lake

En esta sección, vamos a echar un vistazo rápido a cómo se pueden usar Delta Lake y Apache Spark para construir lagos. Específicamente, exploraremos lo siguiente:

- Lectura y escritura de tablas de Delta Lake con Apache Spark • Cómo Delta Lake permite escrituras por lotes y de transmisión simultáneas con ACID garantías
- Cómo Delta Lake garantiza una mejor calidad de los datos al aplicar el esquema en todas las escrituras, al tiempo que permite la evolución del esquema explícito
- Creación de canalizaciones de datos complejas mediante operaciones de actualización, eliminación y combinación, todo de los cuales aseguran las garantías ACID
- Auditir el historial de operaciones que modificaron una tabla de Delta Lake y retroceder en el tiempo consultando versiones anteriores de la tabla

Los datos que utilizaremos en esta sección son una versión modificada (un subconjunto de columnas en formato Parquet) de los [datos públicos de préstamos de Lending Club¹](#). Incluye todos los préstamos financieros de préstamo incluye la información del solicitante proporcionada por el solicitante, así como el estado actual del préstamo (actual, atrasado, pagado en su totalidad, etc.) y la información del último pago.

Configuración de Apache Spark con Delta Lake

Puede configurar Apache Spark para que se vincule a la biblioteca de Delta Lake de una de las siguientes maneras:

Configurar un shell interactivo

Si usa Apache Spark 3.0, puede iniciar un shell de PySpark o Scala con Delta Lake usando el siguiente argumento de la línea de comandos:

`--paquetes io.delta:delta-core_2.12:0.7.0`

Por ejemplo:

`pyspark --paquetes io.delta:delta-core_2.12:0.7.0`

Si está ejecutando Spark 2.4, debe usar Delta Lake 0.6.0.

¹ Una vista completa de los datos está disponible en [este archivo de Excel](#).

Configurar un proyecto independiente de Scala/Java usando coordenadas de Maven

Si desea crear un proyecto utilizando los archivos binarios de Delta Lake desde el repositorio de Maven Central, puede agregar las siguientes coordenadas de Maven a las dependencias del proyecto:

```
<dependencia>
<groupId>io.delta</groupId>
<artifactId>delta-core_2.12</artifactId>
<version>0.7.0</version>
</dependencia>
```

Nuevamente, si está ejecutando Spark 2.4, debe usar Delta Lake 0.6.0.



Ver la [documentación de Delta Lake](#) para obtener la información más actualizada.

Carga de datos en una tabla de Delta Lake

Si está acostumbrado a crear lagos de datos con Apache Spark y cualquiera de los formatos de datos estructurados, por ejemplo, Parquet, entonces es muy fácil migrar las cargas de trabajo existentes para usar el formato de Delta Lake. Todo lo que tiene que hacer es cambiar todas las operaciones de lectura y escritura de DataFrame para usar `format("delta")` en lugar de `format("parquet")`. Proberemos esto con algunos de los datos de préstamos antes mencionados, que están disponibles [como un archivo de Parquet](#). Primero, leamos estos datos y guardémoslos como una tabla de Delta Lake:

```
// En Scala //
Configure la ruta de datos de origen
val sourcePath = "/databricks-datasets/learning-spark-v2/loans/loan-
risks.snappy.parquet"

// Configurar la ruta del lago Delta
val deltaPath = "/tmp/loans_delta"

// Crear la tabla Delta con los mismos datos de préstamos
```

```
chispa .read .format("parquet") .load(sourcePath) .write .format("delta") .save(deltaPath)

// Crear una vista de los datos llamada préstamos_delta
chispa .read .format("delta")
```

```

.load(ruta delta)
.createOrReplaceTempView("préstamos_delta")

# en pitón
# Configurar la ruta de datos de origen
sourcePath = "/databricks-datasets/learning-spark-v2/loans/
préstamos riesgos.snappy.parquet"

# Configurar la ruta del lago Delta
deltaPath = "/tmp/préstamos_delta"

# Cree la tabla Delta Lake con los mismos datos de préstamos
(spark.read.format("parquet").load(origenPath)
 .write.format("delta").save(ruta delta))

# Crear una vista sobre los datos llamada préstamos_delta
spark.read.format("delta").load(deltaPath).createOrReplaceTempView("loans_delta")

```

Ahora podemos leer y explorar los datos tan fácilmente como cualquier otra tabla:

// En Scala/Python

```
// Recuento de filas de préstamos
chispa.sql("SELECCIONE el recuento (*) DE préstamos_delta").show()
```

```
+-----+
|cuenta(1)|
+-----+
| 14705|
+-----+
```

```
// Primeras 5 filas de la tabla de préstamos
chispa.sql("SELECCIONE * DE préstamos_delta LÍMITE 5").show()
```

	id_de_préstamo	importe_financiado	importe_pagado	estado_dirección	CA
0	1000	182.22	1000	Washington	
1	361.19	1000	176.26	Texas	
2	1000	1000.0	1000	Aceptar	
3	249.98			Pensilvania	
4					

Carga de flujos de datos en una tabla de Delta Lake

Al igual que

con los marcos de datos estáticos, puede modificar fácilmente sus trabajos de transmisión estructurada existentes para escribir y leer en una tabla de Delta Lake configurando el formato en "delta". Supongamos que tiene un flujo de datos de préstamos nuevos como un DataFrame llamado newLoanStreamDF, que tiene el mismo esquema que la tabla. Puede agregar a la tabla de la siguiente manera:

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

val newLoanStreamDF = ... // Streaming DataFrame con datos de nuevos préstamos
val checkpointDir = // Directorio para newLoanStreamDF.writeStream val streamingQuery

.streamingQuery
    .format("delta") .option("ubicación del punto de control",
    dirección del punto de
    control) .trigger(Trigger.ProcessingTime("10 segundos")) .start(deltaPath)

# En Python
newLoanStreamDF = # Streaming DataFrame con datos de nuevos préstamos
checkpointDir = # Directorio para puntos de control de streaming streamingQuery
(newLoanStreamDF.writeStream .format("delta") .option("checkpointLocation",
    checkpointDir) .trigger(processingTime = "10 segundos" ) .start(rutadelta))
```

Con este formato, como cualquier otro, el Streaming estructurado ofrece garantías de extremo a extremo exactamente una vez. Sin embargo, Delta Lake tiene algunas ventajas adicionales sobre los formatos tradicionales como JSON, Parquet u ORC:

Permite escrituras de trabajos por lotes y de transmisión en la misma tabla

Con otros formatos, los datos escritos en una tabla desde un trabajo de transmisión estructurada sobrescribirán cualquier dato existente en la tabla. Esto se debe a que los metadatos mantenidos en la tabla para garantizar las garantías de una sola vez para las escrituras de transmisión no tienen en cuenta otras escrituras que no son de transmisión. La gestión avanzada de metadatos de Delta Lake permite escribir datos tanto por lotes como de transmisión.

Permite que múltiples trabajos de transmisión agreguen datos a la misma

tabla. La misma limitación de metadatos con otros formatos también evita que múltiples consultas de transmisión estructurada se agreguen a la misma tabla. Los metadatos de Delta Lake mantienen información de transacción para cada consulta de transmisión, lo que permite que cualquier cantidad de consultas de transmisión se escriban simultáneamente en una tabla con garantías de una sola vez.

Proporciona garantías ACID incluso en escrituras simultáneas

A diferencia de los formatos integrados, Delta Lake permite operaciones por lotes y de transmisión simultáneas para escribir datos con garantías ACID.

Hacer cumplir el esquema en la escritura para evitar la corrupción de datos

datos Un problema común con la administración de datos con Spark usando formatos comunes como JSON, Parquet y ORC es la corrupción accidental de datos causada por la escritura de datos con formato incorrecto. Dado que estos formatos definen el diseño de datos de archivos individuales y no de toda la tabla, no existe ningún mecanismo para evitar que cualquier trabajo de Spark escriba archivos con diferentes esquemas en las tablas existentes. Esto significa que no hay garantías de coherencia para toda la tabla de muchos archivos de Parquet.

El formato Delta Lake registra el esquema como metadatos a nivel de tabla. Por lo tanto, todas las escrituras en una tabla de Delta Lake pueden verificar si los datos que se escriben tienen un esquema compatible con el de la tabla. Si no es compatible, Spark generará un error antes de que se escriban y confirmen datos en la tabla, lo que evitara la corrupción accidental de datos. Probemos esto tratando de escribir algunos datos con una columna adicional, cerrada, que indique si el préstamo ha sido rescindido. Tenga en cuenta que esta columna no existe en la tabla:

```
// En Scala val
préstamoActualizaciones = Seq(
  (1111111L, 1000, 1000.0, "TX", falso, (2222222L,
  2000, 0.0, "CA", verdadero))
  .toDF("loan_id", "funded_amnt", "paid_amnt", "addr_state", "closed")

préstamoActualizaciones.escribir.format("delta").modo("añadir").guardar(ruta delta)

# En Python
desde pyspark.sql.functions import *

cols = ['loan_id', 'funded_amnt', 'paid_amnt', 'addr_state', 'closed'] items = [ (1111111, 1000, 1000.0,
'TX', True), (2222222, 2000, 0.0, 'CA', Falso) ]

préstamoActualizaciones = (chispa.createDataFrame(elementos, columnas )
  .withColumn("funded_amnt", col("funded_amnt").cast("int")))
préstamoActualizaciones.write.format("delta").mode("append").save(deltaPath)
```

Esta escritura fallará con el siguiente mensaje de error:

```
org.apache.spark.sql.AnalysisException: se detectó una discrepancia de esquema al escribir en la tabla Delta (ID de
tabla: 48bfa949-5a09-49ce-96cb-34090ab7d695).
Para habilitar la migración de esquemas, establezca :
'.option("mergeSchema", "true")'.
```

Esquema de tabla:
raíz
-- id_prestamo: largo (anulable = verdadero) --
monto_financiado: entero (anulable = verdadero) --
pago_imuesto: doble (anulable = verdadero) -- addr_state:
cadena (anulable = verdadero)

```
Fecha del esquema:  
raíz  
-- id_prestamo: largo (anulable =  
verdadero) -- monto_financiado: entero (anulable =  
verdadero) -- pago_impuesto: doble (anulable =  
verdadero) -- addr_state: cadena (anulable =  
verdadero) -- cerrado: booleano (anulable = verdadero ) )
```

Esto ilustra cómo Delta Lake bloquea las escrituras que no coinciden con el esquema de la tabla. Sin embargo, también da una pista sobre cómo evolucionar realmente el esquema de la tabla usando la opción `mergeSchema`, como se explica a continuación.

Esquemas en evolución para acomodar datos cambiantes

En nuestro mundo de datos en constante cambio, es posible que deseemos agregar esta nueva columna a la tabla. Esta nueva columna se puede agregar explícitamente configurando la opción "`mergeSchema`" en "`true`":

// En Scala

```
préstamosActualizaciones.escribir.formato("delta").modo("añadir") .opción("mergeSchema", "true") .save(deltaPath)
```

En Python

```
(loanUpdates.write.format("delta").mode("append") .option("mergeSchema",  
"true") .save(deltaPath))
```

Con esto, la columna cerrada se agregará al esquema de la tabla y se agregarán nuevos datos.

Cuando se leen las filas existentes, el valor de la nueva columna se considera NULL. En Spark 3.0, también puede usar el comando SQL DDL ALTER TABLE para agregar y modificar columnas.

Transformación de datos existentes

Delta Lake admite los comandos DML ACTUALIZAR, ELIMINAR y COMBINAR, que le permiten crear canalizaciones de datos complejas. Estos comandos se pueden invocar mediante Java, Scala, Python y SQL, lo que brinda a los usuarios la flexibilidad de utilizar los comandos con cualquier API con la que estén familiarizados, utilizando DataFrames o tablas. Además, cada una de estas operaciones de modificación de datos asegura las garantías de ACID.

Exploraremos esto con algunos ejemplos de casos de uso del mundo real.

Actualización de datos a x

errores Un caso de uso común al administrar datos es corregir errores en los datos. Supongamos que, al revisar los datos, nos dimos cuenta de que todos los préstamos asignados a addr_state = 'OR' deberían haberse asignado a addr_state = 'WA'. Si la tabla de préstamos fuera una tabla de Parquet, para realizar dicha actualización necesitaríamos:

1. Copie todas las filas que no se ven afectadas en una nueva tabla.
2. Copie todas las filas que se ven afectadas en un DataFrame, luego realice los datos modificación.
3. Inserte las filas del DataFrame anotado anteriormente en la nueva tabla.
4. Elimine la tabla anterior y cambie el nombre de la tabla nueva al nombre de la tabla anterior.

En Spark 3.0, que agregó soporte directo para operaciones SQL DML como ACTUALIZAR, ELIMINAR y COMBINAR, en lugar de realizar manualmente todos estos pasos, simplemente puede ejecutar el comando ACTUALIZAR de SQL. Sin embargo, con una tabla de Delta Lake, los usuarios también pueden ejecutar esta operación mediante el uso de las API programáticas de Delta Lake de la siguiente manera:

```
// En Scala
import io.delta.tables.DeltaTable
import org.apache.spark.sql.functions._

val deltaTable = DeltaTable.forPath(spark, deltaPath)
deltaTable.update( col("addr_state") === "OR", Map("addr_state"
-> lit("WA")))
```

```
# En Python
desde la importación de delta.tables *

deltaTable = DeltaTable.forPath(spark, deltaPath)
deltaTable.update("addr_state = 'OR'", {"addr_state": "'WA'"})
```

Eliminación de datos

relacionados con el usuario Con políticas de protección de datos como el [Reglamento general de protección de datos \(RGPD\)](#) de la UE entrando en vigor, ahora es más importante que nunca poder eliminar los datos de los usuarios de todas sus tablas. Digamos que es obligatorio que elimine los datos de todos los préstamos que se han pagado por completo. Con Delta Lake, puede hacer lo siguiente:

```
// En Scala
val deltaTable = DeltaTable.forPath(spark, deltaPath)
deltaTable.delete("funded_amnt >= paid_amnt")

# En Python
deltaTable = DeltaTable.forPath (spark, deltaPath)
deltaTable.delete ("funded_amnt > = paid_amnt")
```

De manera similar a las actualizaciones, con Delta Lake y Apache Spark 3.0 puede ejecutar directamente el comando DELETE SQL en la tabla.

Alterar datos de cambio en una tabla usando merge ()

Un caso de uso común es la captura de datos modificados, donde debe replicar los cambios de fila realizados en una tabla OLTP en otra tabla para las cargas de trabajo OLAP. Para continuar con nuestro ejemplo de datos de préstamos, supongamos que tenemos otra tabla de información de préstamos nuevos, algunos de los cuales son préstamos nuevos y otros son actualizaciones de préstamos existentes. Además, supongamos que esta tabla de cambios tiene el mismo esquema que la tabla de préstamos_delta . Puede modificar estos cambios en la tabla mediante la operación DeltaTable.merge() , que se basa en el comando MERGE SQL:

// En Scala

```
deltaTable.alias("t") .merge(loanUpdates.alias("s"), "t.loan_id =  
s.loan_id") .whenMatched.updateAll() .whenNotMatched.insertAll() .execute()
```

En Python

```
(deltaTable.alias("t") .merge(loanUpdates.alias("s"), "t.loan_id =  
s.loan_id") .whenMatchedUpdateAll() .whenNotMatchedInsertAll() .execute())
```

Como recordatorio, puede ejecutar esto como un comando SQL MERGE a partir de Spark 3.0.

Además, si tiene una transmisión de dichos cambios capturados, puede aplicar esos cambios continuamente mediante una consulta de transmisión estructurada. La consulta puede leer los cambios en los microlotes (consulte el Capítulo 8) desde cualquier fuente de transmisión y usar fore achBatch() para aplicar los cambios en cada microlote a la tabla Delta Lake.

Deduplicación de datos durante la inserción mediante combinación de solo inserción

La operación de fusión en Delta Lake admite una sintaxis extendida más allá de la especificada por el estándar ANSI, que incluye funciones avanzadas como las siguientes:

Eliminar acciones

Por ejemplo, COMBINAR... CUANDO COINCIDE ENTENTCES ELIMINAR.

Condiciones de la

cláusula Por ejemplo, MERGE .CUANDO COINCIDE Y <condición> ENTENTCES

Acciones opcionales

Todas las cláusulas MATCHED y NOT MATCHED son opcionales.

Sintaxis de

estrella Por ejemplo, ACTUALIZAR * e INSERTAR * para actualizar/insertar todas las columnas en la tabla de destino con columnas coincidentes del conjunto de datos de origen. Las API de Delta Lake equivalentes son updateAll() e insertAll(), que vimos en la anterior sección.

Esto le permite expresar muchos casos de uso más complejos con poco código. Por ejemplo, digamos que desea llenar la tabla de préstamos_delta con datos históricos sobre préstamos anteriores. Pero es posible que algunos de los datos históricos ya se hayan insertado en la tabla y no deseé actualizar esos registros porque pueden contener información más actualizada. Puede deduplicar por el id_de_préstamo mientras realiza la inserción ejecutando la siguiente operación de combinación con solo la acción INSERTAR (dado que la acción ACTUALIZAR es opcional):

// En Scala

```
deltaTable.alias("t").merge(historicalUpdates.alias("s"), "t.loan_id =  
s.loan_id").whenNotMatched.insertAll().execute()
```

En Python

```
(deltaTable.alias("t").merge(historicalUpdates.alias("s"), "t.loan_id =  
s.loan_id").whenNotMatchedInsertAll().execute())
```

Hay casos de uso aún más complejos, como CDC con eliminaciones y tablas SCD, que se simplifican con la sintaxis de combinación extendida. Consulte la [documentación](#) para más detalles y ejemplos.

Auditoría de cambios de datos con el historial de operaciones

Todos los cambios en su tabla de Delta Lake se registran como confirmaciones en el registro de transacciones de la tabla. A medida que escribe en una tabla o directorio de Delta Lake, cada operación se versiona automáticamente. Puede consultar el historial de operaciones de la tabla como se indica en el siguiente fragmento de código:

// En Scala/Python
deltaTable.history().show()

Por defecto, esto mostrará una tabla enorme con muchas versiones y muchas columnas. En su lugar, imprimamos algunas de las columnas clave de las últimas tres operaciones:

// En Scala

```
deltaTable.history(3).select("versión", "marca de tiempo", "operación", "parámetros  
de operación").show(false)
```

En Python

```
(deltaTable .history(3) .select("version", "timestamp", "operation",
    "operationParameters") .show(truncate=False))
```

Esto generará la siguiente salida:

versión	marca de tiempo	operación	parámetros de operación
5	2020-04-07	MERGE	[[predicado -> (t.`loan_id` = s.`loan_id`)] [[predicado ->
4	2020-04-07	MERGE	(t.`loan_id` = s.`loan_id`)]
3	2020-04-07	ELIMINAR	[[predicado -> ["(CAST(`funded_amnt` ...

Tenga en cuenta la operación y los parámetros de operación que son útiles para auditar los cambios.

Consulta de instantáneas anteriores de una tabla con viajes en el tiempo

Puede consultar instantáneas de versiones anteriores de una tabla mediante las opciones "versionAsOf" y "timestampAsOf" de DataFrameReader . Aquí están algunos ejemplos:

// En Scala

```
spark.read .format("delta") .option("timestampAsOf", "2020-01-01") // marca de tiempo después de la
                     creación de la tabla .load(deltaPath)
```

```
chispa.read.format("delta")
    .option("versionAsOf",
    "4") .load(deltaPath)
```

En Python

```
(spark.read .format("delta") .option("timestampAsOf", "2020-01-01") # marca de tiempo después de la
                     creación de la tabla .load(deltaPath))
```

```
(chispa.leer.formato("delta")
    .option("versionAsOf",
    "4") .load(deltaPath))
```

Esto es útil en una variedad de situaciones, tales como:

- Reproducir experimentos e informes de aprendizaje automático volviendo a ejecutar el trabajo en una versión de tabla específica
- Comparar los cambios de datos entre diferentes versiones para la auditoría •

Revertir los cambios incorrectos leyendo una instantánea anterior como un DataFrame y sobrescribiendo la tabla con ella

Resumen

Este capítulo examinó las posibilidades de crear lagos de datos confiables con Apache Spark. En resumen, las bases de datos han resuelto problemas de datos durante mucho tiempo, pero no cumplen con los diversos requisitos de los casos de uso y las cargas de trabajo modernos. Los lagos de datos se crearon para paliar algunas de las limitaciones de las bases de datos y Apache Spark es una de las mejores herramientas para crearlos. Sin embargo, los lagos de datos aún carecen de algunas de las funciones clave que brindan las bases de datos (p. ej., las garantías de ACID). Lakehouses son la próxima generación de soluciones de datos, cuyo objetivo es proporcionar las mejores funciones de bases de datos y lagos de datos y cumplir con todos los requisitos de diversos casos de uso y cargas de trabajo.

Exploramos brevemente un par de sistemas de código abierto (Apache Hudi y Apache Iceberg) que se pueden usar para construir lagos, luego analizamos más de cerca Delta Lake, un formato de almacenamiento de código abierto basado en archivos que, junto con Apache Spark, es un gran bloque de construcción para las casas del lago. Como viste, proporciona lo siguiente:

- Garantías transaccionales y gestión de esquemas, como bases de datos • Escalabilidad y apertura, como lagos de datos • Compatibilidad con cargas de trabajo simultáneas por lotes y transmisión con garantías ACID • Compatibilidad con la transformación de datos existentes mediante operaciones de actualización, eliminación y combinación que garantizan las garantías ACID Soporte para control de versiones, auditoría del historial de operaciones y consulta de versiones anteriores.
versiones

En el próximo capítulo, exploraremos cómo comenzar a construir modelos ML usando MLlib de Spark.

CAPÍTULO 10

Aprendizaje automático con MLlib

Hasta este momento, nos hemos centrado en las cargas de trabajo de ingeniería de datos con Apache Spark. La ingeniería de datos suele ser un paso previo a la preparación de sus datos para las tareas de aprendizaje automático (ML), que serán el tema central de este capítulo. Vivimos en una era en la que el aprendizaje automático y las aplicaciones de inteligencia artificial son una parte integral de nuestras vidas. Lo más probable es que nos demos cuenta o no, todos los días entramos en contacto con modelos de ML para fines tales como recomendaciones y anuncios de compras en línea, detección de fraudes, clasificación, reconocimiento de imágenes, comparación de patrones y más. Estos modelos de ML impulsan decisiones comerciales importantes para muchas empresas.

Según [este estudio de McKinsey](#), El 35 % de lo que los consumidores compran en Amazon y el 75 % de lo que ven en Netflix están impulsados por recomendaciones de productos basadas en el aprendizaje automático. Construir un modelo que funcione bien puede hacer o deshacer a las empresas.

En este capítulo, lo ayudaremos a comenzar a construir modelos ML usando [MLlib](#), la biblioteca de aprendizaje automático de facto en Apache Spark. Comenzaremos con una breve introducción al aprendizaje automático, luego cubriremos las mejores prácticas para el aprendizaje automático distribuido y la ingeniería de características a escala (si ya está familiarizado con los fundamentos del aprendizaje automático, puede pasar directamente a ["Diseño de canalizaciones de aprendizaje automático"](#) en página 289). A través de los fragmentos de código cortos presentados aquí y los cuadernos disponibles en el repositorio de [GitHub](#) del libro , aprenderá cómo crear modelos básicos de ML y usar MLlib.



Este capítulo cubre las API de Scala y Python; Si está interesado en usar R (sparklyr) con Spark para el aprendizaje automático, lo invitamos a consultar [Mastering Spark with R](#). de Javier Luraschi, Kevin Kuo y Edgar Ruiz (O'Reilly).

¿Qué es el aprendizaje automático?

El aprendizaje automático está recibiendo mucha publicidad en estos días, pero ¿qué es exactamente? En términos generales, el aprendizaje automático es un proceso para extraer patrones de sus datos, utilizando estadísticas, álgebra lineal y optimización numérica. El aprendizaje automático se puede aplicar a problemas como predecir el consumo de energía, determinar si hay o no un gato en su video o agrupar elementos con características similares.

Hay algunos tipos de aprendizaje automático, incluidos el aprendizaje supervisado, semisupervisado, no supervisado y de refuerzo. Este capítulo se centrará principalmente en el aprendizaje automático supervisado y solo abordará el aprendizaje no supervisado. Antes de sumergirnos, analicemos brevemente las diferencias entre ML supervisado y no supervisado.

Aprendizaje supervisado

En el **aprendizaje automático supervisado**, sus datos consisten en un conjunto de registros de entrada, cada uno de los cuales tiene etiquetas asociadas, y el objetivo es predecir la(s) etiqueta(s) de salida dada una nueva entrada sin etiquetar. Estas etiquetas de salida pueden ser discretas o continuas, lo que nos lleva a los dos tipos de aprendizaje automático supervisado: clasificación y regresión.

En un problema de clasificación, el objetivo es separar las entradas en un conjunto discreto de clases o etiquetas. Con la clasificación binaria, hay dos etiquetas discretas que desea predecir, como "perro" o "no perro", como se muestra en la [Figura 10-1](#).



Figura 10-1. Ejemplo de clasificación binaria: perro o no perro

Con la clasificación multiclase, también conocida como multinomial, puede haber tres o más etiquetas discretas, como predecir la raza de un perro (p. ej., pastor australiano, golden retriever o caniche, como se muestra en la [figura 10-2](#)).



Figura 10-2. Ejemplo de clasificación multinomial: pastor australiano, golden retriever o caniche

En problemas de regresión, el valor a predecir es un número continuo, no una etiqueta. Esto significa que puede predecir valores que su modelo no ha visto durante el entrenamiento, como se muestra en la Figura 10-3. Por ejemplo, puede crear un modelo para predecir las ventas diarias de helado dada la temperatura. Su modelo podría predecir el valor de \$77,67, incluso si ninguno de los pares de entrada/salida en los que se entrenó contenía ese valor.

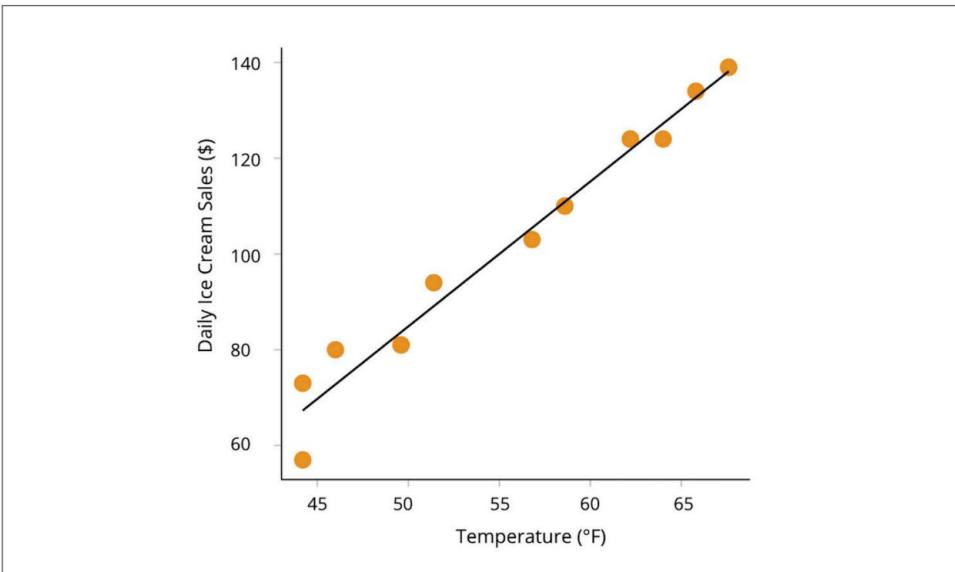


Figura 10-3. Ejemplo de regresión: predicción de ventas de helados en función de la temperatura

La Tabla 10-1 enumera algunos algoritmos de aprendizaje automático supervisado de uso común que están disponibles en [Spark MLlib](#), con una nota sobre si se pueden usar para regresión, clasificación o ambas.

Tabla 10-1. Algoritmos populares de clasificación y regresión

Algoritmo	
regresión lineal	regresión
Regresión logística	(lo sabemos, ¡tiene regresión en el nombre!)
Árboles de decisión	Ambas cosas
Árboles potenciados por gradiente	Ambas cosas
Bosques aleatorios	Ambas cosas
bayesiana ingenua	Clasificación
Clasificación de las máquinas de vectores de soporte (SVM)	

Aprendizaje no supervisado La

obtención de los datos etiquetados que requiere el aprendizaje automático supervisado puede ser muy costosa y/o inviable. Aquí es donde **el aprendizaje automático no supervisado** entra en juego. En lugar de predecir una etiqueta, ML no supervisado lo ayuda a comprender mejor la estructura de sus datos.

Como ejemplo, considere los datos no agrupados originales a la izquierda en la [Figura 10-4](#).

No existe una etiqueta verdadera conocida para cada uno de estos puntos de datos (x_1, x_2), pero al aplicar el aprendizaje automático no supervisado a nuestros datos, podemos encontrar los grupos que se forman naturalmente, como se muestra a la derecha.

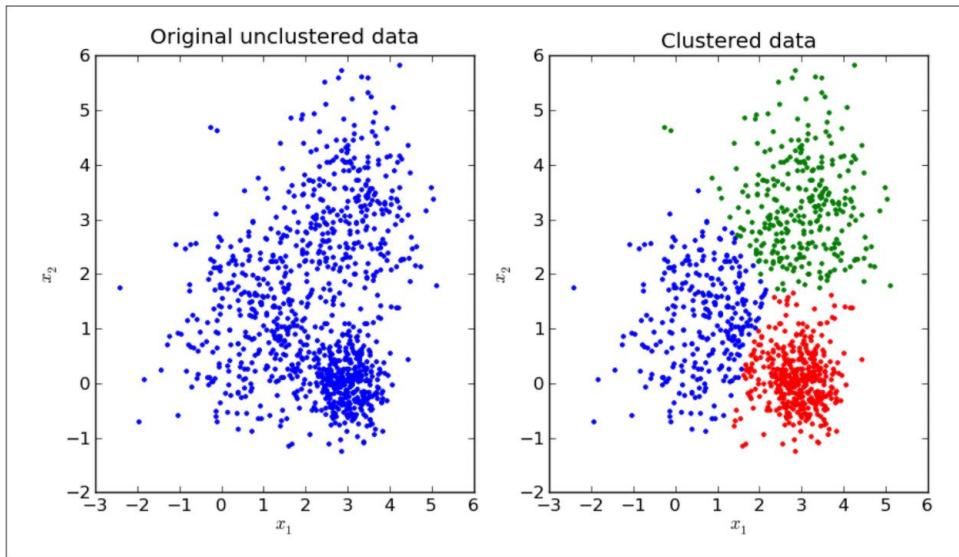


Figura 10-4. Ejemplo de agrupamiento

El aprendizaje automático no supervisado se puede utilizar para la detección de valores atípicos o como un paso de preprocessamiento para el aprendizaje automático supervisado, por ejemplo, para [reducir la dimensionalidad](#).

(es decir, número de dimensiones por dato) del conjunto de datos, que es útil para reducir los requisitos de almacenamiento o simplificar las tareas posteriores. Algunos algoritmos de aprendizaje **automático no supervisados en MLlib** incluyen k-medias, asignación latente de Dirichlet (LDA) y modelos de mezcla gaussiana.

¿Por qué Spark para el aprendizaje automático?

Spark es un motor de análisis unificado que proporciona un ecosistema para la ingesta de datos, la ingeniería de características, el entrenamiento de modelos y la implementación. Sin Spark, los desarrolladores necesitarían muchas herramientas dispares para realizar este conjunto de tareas y aún podrían tener problemas con la escalabilidad.

Spark tiene dos paquetes de aprendizaje automático: **spark.mllib** y **chispa.ml**. spark.mllib es la API de aprendizaje automático original, basada en la API de RDD (que ha estado en modo de mantenimiento desde Spark 2.0), mientras que spark.ml es la API más nueva, basada en marcos de datos. El resto de este capítulo se centrará en el uso del paquete spark.ml y en cómo diseñar canalizaciones de aprendizaje automático en Spark. Sin embargo, usamos "MLlib" como término general para referirnos a ambos paquetes de bibliotecas de aprendizaje automático en Apache Spark.

Con spark.ml, los científicos de datos pueden usar un ecosistema para la preparación de datos y la creación de modelos, sin necesidad de reducir la muestra de sus datos para que quepan en una sola máquina. spark.ml se enfoca en la escalabilidad horizontal $O(n)$, donde el modelo escala linealmente con la cantidad de puntos de datos que tiene, por lo que puede escalar a cantidades masivas de datos. En el siguiente capítulo, discutiremos algunas de las compensaciones involucradas en la elección entre un marco distribuido como spark.ml y un marco de un solo nodo como **scikit-learn**. (aprender). Si ha usado scikit-learn anteriormente, muchas de las API en spark.ml le resultarán bastante familiares, pero hay algunas diferencias sutiles que discutiremos.

Diseño de canalizaciones de aprendizaje automático

En esta sección, cubriremos cómo crear y ajustar canalizaciones de ML. El concepto de canalizaciones es común en muchos marcos de ML como una forma de organizar una serie de operaciones para aplicar a sus datos. En MLlib, la **API de Pipeline** proporciona una API de alto nivel construida sobre DataFrames para organizar su flujo de trabajo de aprendizaje automático. La Pipeline API está compuesta por una serie de transformadores y estimadores, de los que hablaremos en profundidad más adelante.

A lo largo de este capítulo, utilizaremos el conjunto de datos de vivienda de San Francisco de **Inside Airbnb**. Contiene información sobre los alquileres de Airbnb en San Francisco, como la cantidad de habitaciones, la ubicación, los puntajes de revisión, etc., y nuestro objetivo es construir un modelo para predecir los precios de alquiler por noche para los listados en esa ciudad. Este es un problema de regresión, porque el precio es una variable continua. Lo guiaremos a través del flujo de trabajo por el que pasaría un científico de datos para abordar este problema, incluida la ingeniería de características,

construcción de modelos, ajuste de hiperparámetros y evaluación del rendimiento del modelo. Este conjunto de datos es bastante desordenado y puede ser difícil de modelar (¡como la mayoría de los conjuntos de datos del mundo real!), así que si está experimentando por su cuenta, no se sienta mal si sus primeros modelos no son excelentes.

La intención de este capítulo no es mostrarle todas las API en MLlib, sino equiparlo con las habilidades y el conocimiento para comenzar a usar MLlib para construir canalizaciones de un extremo a otro. Antes de entrar en detalles, definamos algo de terminología de MLlib:

Transformer

Acepta un DataFrame como entrada y devuelve un nuevo DataFrame con una o más columnas añadidas.

Los transformadores no aprenden ningún parámetro de sus datos y simplemente aplican transformaciones basadas en reglas para preparar los datos para el entrenamiento del modelo o generar predicciones usando un modelo MLlib entrenado. Tienen un método `.transform()`.

Estimador

Aprende (o "ajusta") los parámetros de su DataFrame a través de un método `.fit()` y devuelve un modelo, que es un transformador.

Pipeline

Organiza una serie de transformadores y estimadores en un solo modelo. Si bien los conductos en sí mismos son estimadores, la salida de `pipeline.fit()` devuelve un `PipelineModel`, un transformador.

Si bien estos conceptos pueden parecer bastante abstractos en este momento, los fragmentos de código y los ejemplos de este capítulo lo ayudarán a comprender cómo se unen. Pero antes de que podamos construir nuestros modelos ML y usar transformadores, estimadores y canalizaciones, necesitamos cargar nuestros datos y realizar alguna preparación de datos.

Ingestión y exploración de datos

Hemos

preprocesado ligeramente los datos en nuestro conjunto de datos de ejemplo para eliminar valores atípicos (por ejemplo, Airbnbs publicados por \$0/noche), convertido todos los números enteros en dobles y seleccionado un subconjunto informativo de los más de cien campos. Además, para cualquier valor numérico faltante en nuestras columnas de datos, imputamos el valor de la mediana y agregamos una columna indicadora (el nombre de la columna seguido de `_na`, como `dormitorios_na`). De esta manera, el modelo de ML o el analista humano pueden interpretar cualquier valor en esa columna como un valor imputado, no como un valor real. Puede ver el cuaderno de preparación de datos en el repositorio de [GitHub del libro](#). Tenga en cuenta que hay muchas otras formas de manejar los valores perdidos, que están fuera del alcance de este libro.

Echemos un vistazo rápido al conjunto de datos y el esquema correspondiente (con el resultado que muestra solo un subconjunto de las columnas):

```
# en pitón
filePath = """/databricks-conjuntos de datos/learning-spark-v2/sf-airbnb/
sf-airbnb-clean.parquet"""
airbnbDF = chispa.read.parquet(filePath)
airbnbDF.select("barrio_limpio", "tipo_habitación", "dormitorios", "baños",
"número_de_reseñas", "precio").show(5)

// En escala
val vía_archivo =
  "/databricks-datasets/learning-spark-v2/sf-airbnb/sf-airbnb-clean.parquet"
val airbnbDF = chispa.read.parquet(filePath)
airbnbDF.select("barrio_limpio", "tipo_habitación", "dormitorios", "baños",
"número_de_reseñas", "precio").show(5)

+-----+-----+-----+-----+
|barrio_limpio| room_type|dormitorios|baños|número_...|precio|
+-----+-----+-----+-----+
| Western Addition| Casa/apto. entero| 1.0| 1.0| 180,0|170,0|
| Bernal Heights| Casa/apto. entero| 2.0| 1.0| 111,0|235,0|
| Haight Ashbury| Habitación privada| 1.0| 4.0| 17,0| 65,0|
| Haight Ashbury| Habitación privada| 1.0| 4.0| 8,0| 65,0|
| Western Addition| Casa/apto. entero| 2.0| 1.5| 27,0|785,0|
+-----+-----+-----+-----+
```

Nuestro objetivo es predecir el precio por noche de una propiedad de alquiler, dadas nuestras características.



Antes de que los científicos de datos puedan llegar a la construcción de modelos, necesitan explorar y comprender sus datos. A menudo usarán Spark para agrupar sus datos, luego usar bibliotecas de visualización de datos como [mat](#) y [tramlab](#) para visualizar los datos. Dejaremos la exploración de datos como un ejercicio para el lector.

Creación de conjuntos de datos de prueba y entrenamiento

Antes de comenzar la ingeniería y el modelado de características, dividiremos nuestro conjunto de datos en dos grupos: entrenar y probar. Dependiendo del tamaño de su conjunto de datos, su proporción de entrenamiento/prueba puede variar, pero muchos científicos de datos usan 80/20 como una división estándar de entrenamiento/prueba. podrías estar preguntando, "¿Por qué no usar todo el conjunto de datos para entrenar el modelo?" El problema es que si construimos un modelo en todo el conjunto de datos, es posible que el modelo se "sobreajuste" a los datos de entrenamiento que proporcionamos, y no tendríamos más datos con los que evaluar qué tan bien se generaliza a datos nunca antes vistos. Los modelos de rendimiento en el conjunto de prueba es un indicador de qué tan bien funcionará en datos no vistos (es decir, en la naturaleza o en producción), asumiendo que los datos siguen distribuciones similares. Esta división se representa en la figura 10-5.

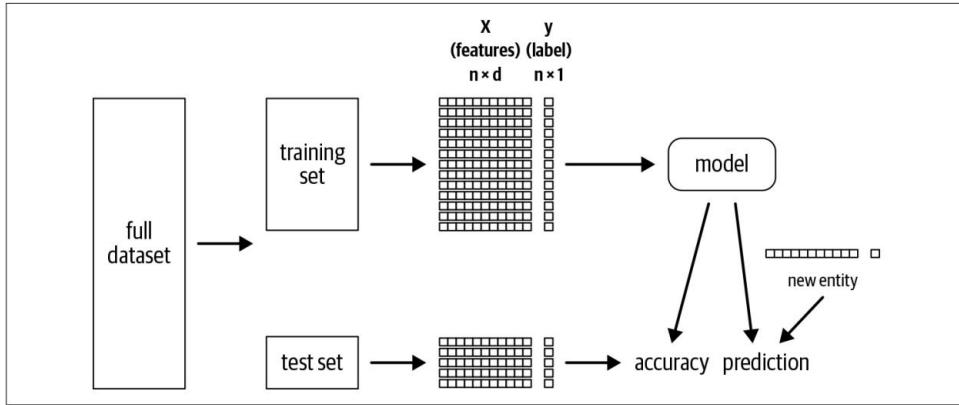


Figura 10-5. División de tren/prueba

Nuestro conjunto de entrenamiento consta de un conjunto de características, X, y una etiqueta, y. Aquí usamos X mayúscula para denotar una matriz con dimensiones $n \times d$, donde n es el número de puntos de datos (o ejemplos) y d es el número de características (así es como llamamos a los campos o columnas en nuestro DataFrame). Usamos y minúscula para denotar un vector, con dimensiones $n \times 1$; para cada ejemplo, hay una etiqueta.

Se utilizan diferentes métricas para medir el rendimiento del modelo. Para problemas de clasificación, una métrica estándar es la precisión, o porcentaje, de predicciones correctas. Una vez que el modelo tenga un desempeño satisfactorio en el conjunto de entrenamiento usando esa métrica, aplicaremos el modelo a nuestro conjunto de prueba. Si funciona bien en nuestro conjunto de prueba de acuerdo con nuestras métricas de evaluación, entonces podemos estar seguros de que hemos creado un modelo que se "generalizará" a datos no vistos.

Para nuestro conjunto de datos de Airbnb, mantendremos el 80 % para el conjunto de entrenamiento y reservaremos el 20 % de nuestros datos para el conjunto de prueba. Además, estableceremos una semilla aleatoria para la reproducibilidad, de modo que si volvemos a ejecutar este código obtendremos los mismos puntos de datos para nuestros conjuntos de datos de entrenamiento y prueba, respectivamente. El valor de la semilla en sí no debería importar, pero a los científicos de datos a menudo les gusta establecerlo en 42, ya que esa es la respuesta a la [última pregunta de la vida](#):

```

# En Python
trainDF, testDF = airbnbDF.randomSplit([.8, .2], seed=42)
print(f"""Hay {trainDF.count()} filas en el conjunto de entrenamiento y {testDF.count()} en el conjunto de prueba""")

// En Scala
val Array(trainDF, testDF) = airbnbDF.randomSplit(Array(.8, .2), seed=42)
println(s"""Hay ${trainDF.count()} filas en el conjunto de entrenamiento, y ${testDF.count()} en el conjunto de prueba""")
  
```

Esto produce la siguiente salida:

Hay 5780 filas en el conjunto de entrenamiento y 1366 en el conjunto de prueba

Pero, ¿qué sucede si cambiamos la cantidad de ejecutores en nuestro clúster Spark? El optimizador Catalyst determina la **forma óptima de particionar sus datos** en función de los recursos de su clúster y el tamaño de su conjunto de datos. Dado que los datos en un Spark DataFrame se dividen en filas y cada trabajador realiza su división independientemente de los demás trabajadores, si los datos en las particiones cambian, el resultado de la división (por división aleatoria ()) no será el mismo .

Si bien puede corregir la configuración de su clúster y su semilla para asegurarse de obtener resultados consistentes, nuestra recomendación es dividir sus datos una vez y luego escribirlos en su propia carpeta de entrenamiento/prueba para que no tenga estos problemas de reproducibilidad.



Durante su análisis exploratorio, debe almacenar en caché el conjunto de datos de entrenamiento porque accederá a él muchas veces durante el proceso de aprendizaje automático. Consulte la sección sobre "[Almacenamiento en caché y persistencia de datos](#)" en la página 183 en el [Capítulo 7](#).

Preparación de características con transformadores

Ahora que hemos dividido nuestros datos en conjuntos de prueba y entrenamiento, preparamos los datos para construir un modelo de regresión lineal que prediga el precio dada la cantidad de habitaciones. En un ejemplo posterior, incluiremos todas las características relevantes, pero por ahora asegúremos de tener la mecánica en su lugar. La regresión lineal (como muchos otros algoritmos en Spark) requiere que todas las características de entrada estén contenidas dentro de un solo vector en su DataFrame. Por lo tanto, necesitamos transformar nuestros datos.

Los transformadores en Spark aceptan un DataFrame como entrada y devuelven un nuevo DataFrame con una o más columnas añadidas. No aprenden de sus datos, sino que aplican transformaciones basadas en reglas mediante el método `transform()` .

Para la tarea de poner todas nuestras características en un solo vector, usaremos el transformador `Assembler VectorAssembler` . `VectorAssembler` toma una lista de columnas de entrada y crea un nuevo DataFrame con una columna adicional, a la que llamaremos características. Combina los valores de esas columnas de entrada en un solo vector:

```
# En Python
desde pyspark.ml.feature import VectorAssembler
vecAssembler = VectorAssembler(inputCols=["dormitorios"], outputCol="características")
vecTrainDF = vecAssembler.transform(trainDF) vecTrainDF.select("dormitorios",
"características", "precio").mostrar(10)

// En Scala
importar org.apache.spark.ml.feature.VectorAssembler
val vecAssembler = new
    VectorAssembler() .setInputCols(Array("dormitorios")) .setOutputCol("características")
val vecTrainDF = vecAssembler.transform(trainDF)
vecTrainDF.select("dormitorios", "características", "precio").show(10)
```

```
+-----+-----+
|dormitorios|características|precio|
+-----+-----+
| 1.0| [1.0]|200.0|
| 1.0| [1.0]|130.0|
| 1.0| [1.0]| 95.0|
| 1.0| [1.0]|250.0|
| 3.0| [3.0]|250.0|
| 1.0| [1.0]|115.0|
| 1.0| [1.0]|105.0|
| 1.0| [1.0]| 86.0|
| 1.0| [1.0]|100.0|
| 2.0| [2.0]|220.0|
+-----+-----+
```

Notarás que en el código de Scala, tuvimos que instanciar el nuevo VectorAssembler objeto así como el uso de métodos setter para cambiar las columnas de entrada y salida. En Python, tiene la opción de pasar los parámetros directamente al constructor de VectorAssembler o usar los métodos setter, pero en Scala solo puedes usar el setter métodos.

Cubrimos los fundamentos de la regresión lineal a continuación, pero si ya está familiarizado con el algoritmo, salte a ["Uso de estimadores para construir modelos"](#) en la página 295.

Comprender la regresión lineal

regresión lineal modela una relación lineal entre su variable dependiente (o etiqueta) y una o más variables independientes (o características). En nuestro caso, queremos encajar un modelo de regresión lineal para predecir el precio de un alquiler de Airbnb dado el número de dormitorios

En la [figura 10-6](#), tenemos una sola característica x y una salida y (esta es nuestra variable dependiente). Poder). La regresión lineal busca ajustar una ecuación para una línea a x e y, que para escalar variables se pueden expresar como $y = mx + b$, donde m es la pendiente y b es el desplazamiento o interceptar.

Los puntos indican los pares verdaderos (x, y) de nuestro conjunto de datos, y la línea sólida indica la línea de mejor ajuste para este conjunto de datos. Los puntos de datos no se alinean perfectamente, por lo que Por lo general, piense en la regresión lineal como el ajuste de un modelo a y $\hat{y} = mx + b + \epsilon$, donde ϵ (ϵ son errores extraídos independientemente por registro x de alguna distribución. Estos son los errores entre las predicciones de nuestro modelo y los valores verdaderos. A menudo pensamos en ϵ como siendo gaussiana, o distribuida normalmente. Las líneas verticales sobre la línea de regresión indique ϵ positivo (o residuos), donde sus valores verdaderos están por encima del valor predicho \hat{y} , y las líneas verticales debajo de la línea de regresión indican residuos negativos. El objetivo de la regresión lineal es encontrar una línea que minimice el cuadrado de estos residuos. Notará que la línea puede extrapolarse predicciones para puntos de datos que no ha visto.

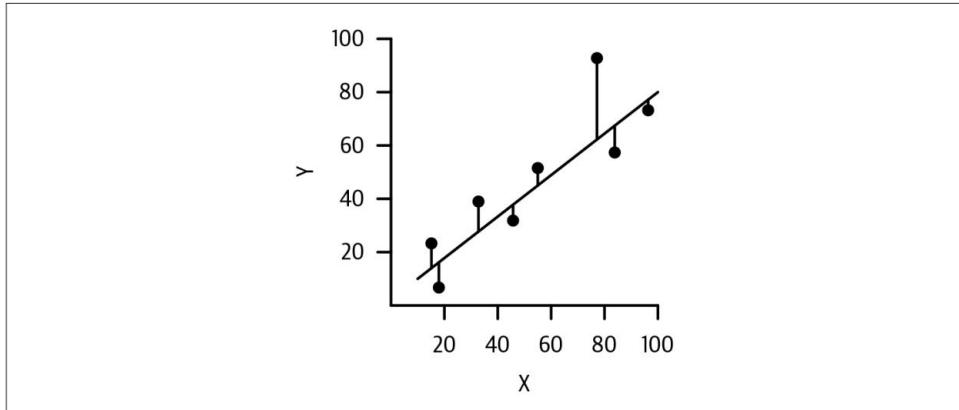


Figura 10-6. Regresión lineal univariante

La regresión lineal también se puede extender para manejar múltiples variables independientes. Si tuviéramos tres características como entrada, $x = [x_1, x_2, x_3]$, entonces podríamos modelar y como $y \sim w_0 + w_1x_1 + w_2x_2 + w_3x_3 + \epsilon$. En este caso, hay un coeficiente (o peso) separado para cada característica y una única intersección (w_0 en lugar de b aquí). El proceso de estimar los coeficientes y el intercepto de nuestro modelo se denomina aprendizaje (o ajuste) de los parámetros del modelo. Por ahora, nos centraremos en el ejemplo de regresión univariada de predecir el precio dada la cantidad de habitaciones, y volveremos a la regresión lineal multivariante en un momento.

Uso de estimadores para construir

modelos Despues de configurar nuestro vectorAssembler, tenemos nuestros datos preparados y transformados en un formato que espera nuestro modelo de regresión lineal. En Spark, **regresión lineal** es un tipo de estimador: toma un marco de datos y devuelve un modelo. Los estimadores aprenden parámetros de sus datos, tienen un método `estimator_name.fit()` y se evalúan con entusiasmo (es decir, inician trabajos de Spark), mientras que los transformadores se evalúan con pereza. Algunos otros ejemplos de estimadores incluyen Imputer, DecisionTreeClassifier y Random ForestRegressor.

Notarás que nuestra columna de entrada para la regresión lineal (características) es la salida de nuestro vectorAssembler:

```
# En Python
desde pyspark.ml.regression import LinearRegression lr =
LinearRegression(featuresCol="features", labelCol="price") lrModel =
lr.fit(vecTrainDF)

// En Scala
import org.apache.spark.ml.regression.LinearRegression val
lr = new LinearRegression() .setFeaturesCol("features")
```

```
.setLabelCol ("precio")
```

```
val lrModel = lr.fit(vecTrainDF)
```

lr.fit() devuelve un modelo de **regresión lineal** (lrModel), que es un transformador. En otras palabras, la salida del método fit() de un estimador es un transformador. Una vez que el estimador ha aprendido los parámetros, el transformador puede aplicar estos parámetros a nuevos puntos de datos para generar predicciones. Inspeccionemos los parámetros que aprendió:

En Python

```
m = round(lrModel.coefficients[0], 2) b =
round(lrModel.intercept, 2) print(f"""La fórmula
para la línea de regresión lineal es precio = {m}*dormitorios + {b}""")
```

// En Scala

```
val m = lrModel.coefficients(0) val b =
lrModel.intercept println(f"""La fórmula
para la línea de regresión lineal es precio = $m%1.2f*dormitorios +
$b%1.2f""")
```

Esto imprime:

La fórmula para la línea de regresión lineal es precio = 123,68*habitaciones + 47,51

Creando un Pipeline Si

queremos aplicar nuestro modelo a nuestro conjunto de prueba, entonces necesitamos preparar esos datos de la misma manera que el conjunto de entrenamiento (es decir, pasarlo a través del ensamblador vectorial). A menudo, las canalizaciones de preparación de datos tendrán varios pasos, y se vuelve engorroso recordar no solo qué pasos aplicar, sino también el orden de los pasos. Esta es la motivación para la **API Pipeline**: simplemente especifica las etapas por las que desea que pasen sus datos, en orden, y Spark se encarga del procesamiento por usted. Proporcionan al usuario una mejor reutilización y organización del código. En Spark, Pipelines son estimadores, mientras que PipelineModels (Pipelines ajustados) son transformadores.

Construyamos nuestra canalización ahora:

En Python

```
desde pyspark.ml import Pipeline
pipeline = Pipeline(stages=[vecAssembler, lr]) pipelineModel
= pipeline.fit(trainDF)
```

// En Scala

```
import org.apache.spark.ml.Pipeline val
pipeline = new Pipeline().setStages(Array(vecAssembler, lr)) val pipelineModel =
pipeline.fit(trainDF)
```

Otra ventaja de usar Pipeline API es que determina qué etapas son estimadores/transformadores para usted, por lo que no tiene que preocuparse por especificar name.fit() versus name.transform() para cada una de las etapas.

Dado que pipelineModel es un transformador, es sencillo aplicarlo a nuestros datos de prueba

establecer también:

```
# en pitón
predDF = pipelineModel.transform(testDF)
predDF.select("dormitorios", "características", "precio", "predicción").show(10)

// En escala
val predDF = pipelineModel.transform(testDF)
predDF.select("dormitorios", "características", "precio", "predicción").show(10)

+-----+-----+
|dormitorios|características| precio | predicción |
+-----+-----+
| 1.0 [1.0] | 85.0 | 171.18598011578285 |          |
| 1.0 [1.0] | 45.0 | 171.18598011578285 |          |
| 1.0 [1.0] | 70.0 | 171.18598011578285 |          |
| 1.0 [1.0] | 128.0 | 171.18598011578285 |          |
| 1.0 [1.0] | 159.0 | 171.18598011578285 |          |
| 2.0 [2.0] | 250.0 | 294.8617264977757 |          |
| 1.0 [1.0] | 99.0 | 171.18598011578285 |          |
| 1.0 [1.0] | 95.0 | 171.18598011578285 |          |
| 1.0 [1.0] | 100.0 | 171.18598011578285 |          |
| 1.0 [1.0] | 2010.0 | 171.18598011578285 |          |
+-----+
```

En este código construimos un modelo usando solo una característica, dormitorios (puedes encontrar el notebook para este capítulo en el repositorio de [GitHub del libro](#)). Sin embargo, es posible que desee construya un modelo utilizando todas sus características, algunas de las cuales pueden ser categóricas, como host_is_superhost. Las características categóricas toman valores discretos y no tienen características intrínsecas. sic ordering: los ejemplos incluyen ocupaciones o nombres de países. En la siguiente sección consideraremos una solución sobre cómo tratar este tipo de variables, conocida como one-hot codificación

Codificación one-hot

En la canalización que acabamos de crear, solo teníamos dos etapas y nuestra regresión lineal el modelo solo usó una función. Echemos un vistazo a cómo construir un poco más complejo tubería que incorpora todas nuestras características numéricas y categóricas.

La mayoría de los modelos de aprendizaje automático en MLlib esperan valores numéricos como entrada, representación enviados como vectores. Para convertir valores categóricos en valores numéricos, podemos usar un técnica llamada codificación one-hot (OHE). Supongamos que tenemos una columna llamada Animal y tenemos tres tipos de animales: Perro, Gato y Pez. No podemos pasar los tipos de cadena. en nuestro modelo ML directamente, por lo que debemos asignar un mapeo numérico, como este:

```
Animal = {"Perro", "Gato", "Pez"}
"Perro" = 1, "Gato" = 2, "Pez" = 3
```

Sin embargo, al utilizar este enfoque, hemos introducido algunas relaciones falsas en nuestro conjunto de datos que no existían antes. Por ejemplo, ¿por qué le asignamos a Gato el doble del valor de Perro? Los valores numéricos que usamos no deben introducir ninguna relación en nuestro conjunto de datos. En cambio, queremos crear una columna separada para cada valor distinto en nuestra columna Animal :

```
"Perro" = [ 1, 0, 0]
"Gato" = [0, 1, 0]
"Pez" = [0, 0, 1]
```

Si el animal es un perro, tiene un uno en la primera columna y ceros en el resto. Si es un gato, tiene un uno en la segunda columna y ceros en el resto. El orden de las columnas es irrelevante. Si ha usado pandas antes, notará que esto hace lo mismo que [pandas.get_dummies\(\)](#).

Si tuviéramos un zoológico de 300 animales, ¿la OHE aumentaría masivamente el consumo de memoria/recursos informáticos? ¡No con Chispa! Spark usa internamente un **SparseVector** cuando la mayoría de las entradas son 0, como suele ser el caso después de OHE, por lo que no desperdicia espacio almacenando valores 0 . Echemos un vistazo a un ejemplo para comprender mejor cómo funcionan los SparseVectors :

```
VectorDenso(0, 0, 0, 7, 0, 2, 0, 0, 0, 0)
Vector disperso (10, [3, 5], [7, 2])
```

El **vector denso** en este ejemplo contiene 10 valores, todos menos 2 de los cuales son 0. Para crear un **SparseVector**, debemos realizar un seguimiento del tamaño del vector, los índices de los elementos distintos de cero y los valores correspondientes en esos índices. En este ejemplo, el tamaño del vector es 10, hay dos valores distintos de cero en los índices 3 y 5, y los valores correspondientes en esos índices son 7 y 2.

Hay algunas formas de codificar sus datos en caliente con Spark. Un enfoque común es usar el **StringIndexer** y **OneHotEncoder**. Con este enfoque, el primer paso es aplicar el estimador StringIndexer para convertir valores categóricos en índices de categoría. Estos índices de categoría están ordenados por frecuencias de etiquetas, por lo que la etiqueta más frecuente obtiene el índice 0, lo que nos brinda resultados reproducibles en varias ejecuciones de los mismos datos.

Una vez que haya creado sus índices de categoría, puede pasarlo como entrada al OneHotEncoder (**OneHotEncoderEstimator** si usa Spark 2.3/2.4). El codificador OneHotEncoder asigna una columna de índices de categoría a una columna de vectores binarios. Eche un vistazo a la [Tabla 10-2](#) para ver las diferencias en las API StringIndexer y OneHotEncoder de Spark 2.3/2.4 a 3.0.

Tabla 10-2. Cambios en StringIndexer y OneHotEncoder en Spark 3.0

	Spark 2.3 y 2.4	Spark 3.0
Indizador de cadenas	Columna única como entrada/salida	Múltiples columnas como entrada/salida
OneHotEncoder	Obsoleto	Múltiples columnas como entrada/salida
OneHotEncoderEstimator	Múltiples columnas como entrada/salida	N/A

El siguiente código demuestra cómo codificar en caliente nuestras características categóricas. En nuestro conjunto de datos, cualquier columna de tipo cadena se trata como una característica categórica, pero a veces es posible que tenga características numéricas que desee tratar como categóricas o viceversa. Deberá identificar cuidadosamente qué columnas son numéricas y cuáles son categóricas:

```
# En Python
desde pyspark.ml.feature import OneHotEncoder, StringIndexer

categoricalCols = [campo para (campo, tipo de datos) en trainDF.dtypes si tipo de datos ==
                   "cadena"] indexOutputCols = [x + "índice" para x en categoricalCols]
oheOutputCols = [x + "OHE" para x en categoricalCols]

stringIndexer = StringIndexer(inputCols=categoricalCols, outputCols=indexOutputCols,
                               handleInvalid="skip") oheEncoder =
                               OneHotEncoder(inputCols=indexOutputCols,
                               outputCols=oheOutputCols)

numericCols = [campo para (campo, tipo de datos) en trainDF.dtypes if ((tipo de datos
                   == "doble") & (campo != "precio"))]
ensambladorInputs = oheOutputCols + numericCols vecAssembler
= VectorAssembler(inputCols=ensambladorInputs, outputCol="características")

// En Scala
importar org.apache.spark.ml.feature.{OneHotEncoder, StringIndexer}

val categoricalCols = trainDF.dtypes.filter(_.2 == "StringType").map(_.1) val indexOutputCols =
categoricalCols.map(_ + "Index") val oheOutputCols = categoricalCols.map(_ + "OHE")

val stringIndexer = new
StringIndexer() .setInputCols(categoricalCols) .setOutputCols(indexOutputCols) .setHandleInvalid("skip")

val oheEncoder = new
OneHotEncoder() .setInputCols(indexOutputCols) .setOutputCols(oheOutputCols)

val numericCols = trainDF.dtypes.filter( case (field, dataType) => dataType == "DoubleType" &&
field != "price").map(_.1)
```

```
val ensambladorInputs = oheOutputCols ++ numericCols val
vecAssembler = new

VectorAssembler().setInputCols(ensambladorInputs).setOutputCol("características")
```

Ahora puede que se pregunte: "¿Cómo maneja StringIndexer las nuevas categorías que aparecen en el conjunto de datos de prueba, pero no en el conjunto de datos de entrenamiento?" Hay un parámetro handleInvalid que especifica cómo desea manejarlos. Las opciones son omitir (filtrar filas con datos no válidos), error (lanzar un error) o mantener (poner datos no válidos en un cubo adicional especial, en el índice numLabels). Para este ejemplo, simplemente omitimos los registros no válidos.

Una dificultad con este enfoque es que debe decirle a StringIndexer explícitamente qué características deben tratarse como características categóricas. Podrías usar [VectorIndexer](#) para detectar automáticamente todas las variables categóricas, pero es computacionalmente costoso ya que tiene que iterar sobre cada columna y detectar si tiene menos de maxCategories valores distintos. maxCategories es un parámetro que especifica el usuario y determinar este valor también puede ser difícil.

Otro enfoque es usar [RFormula](#). La sintaxis para esto está inspirada en el lenguaje de programación R. Con RFormula, proporciona su etiqueta y qué características desea incluir. Admite un subconjunto limitado de operadores R, incluidos ~, ., :, + y -. Por ejemplo, puede especificar formula = "y ~ dormitorios + baños", lo que significa predecir y dados solo dormitorios y baños, o formula = "y ~ .", lo que significa usar todas las funciones disponibles (y excluye automáticamente y de las características). RFormula automáticamente StringIndex y OHE todas sus columnas de cadena , convertirá sus columnas numéricas a tipo doble y combinará todo esto en un solo vector usando VectorAssembler debajo del capó. Por lo tanto, podemos reemplazar todo el código anterior con una sola línea y obtendremos el mismo resultado:

```
# En Python
desde pyspark.ml.feature import RFormula

rFormula = RFormula(formula="precio ~ .",
                     FeaturesCol= "características",
                     labelCol="precio",
                     handleInvalid="skip")

// En Scala
importar org.apache.spark.ml.feature.RFormula

val rFormula = new
  RFormula().setFormula("precio
~ .").setFeaturesCol("características").setLabelCol("precio").setHandleInvalid("saltar")
```

La desventaja de que RFormula combine automáticamente StringIndexer y OneHotEncoder es que no se requiere ni se recomienda la codificación one-hot para todos los algoritmos. Por ejemplo, los algoritmos basados en árboles pueden manejar variables categóricas directamente si solo usa StringIndexer para las características categóricas. No es necesario codificar en caliente las funciones categóricas para los métodos basados en árboles y, a menudo, [empeorará los modelos basados en árboles](#). Desafortunadamente, no existe una solución única para la ingeniería de características, y el enfoque ideal está estrechamente relacionado con los algoritmos posteriores que planea aplicar a su conjunto de datos.



Si alguien más realiza la ingeniería de funciones por usted, asegúrese de que documenten cómo generaron esas funciones.

Una vez que haya escrito el código para transformar su conjunto de datos, puede agregar un modelo de regresión lineal utilizando todas las funciones como entrada.

Aquí, ponemos toda la preparación de características y la creación de modelos en la canalización y la aplicamos a nuestro conjunto de datos:

```
# En Python
lr = LinearRegression(labelCol="price", FeaturesCol= "features") pipeline =
Pipeline(etapas = [ stringIndexer, oheEncoder, vecAssembler, lr])
# O use RFormula #
pipeline = Pipeline(etapas = [rFormula, lr])

pipelineModel = pipeline.fit(trainDF) predDF =
pipelineModel.transform(testDF)
predDF.select("características", "precio", "predicción").show(5)

// En Scala
val lr = new

LinearRegression() .setLabelCol("price") .setFeaturesCol("features") val pipeline = new Pipeline()
.setStages(Array( stringIndexer, oheEncoder, vecAssembler, lr))
// O use RFormula //
val pipeline = new Pipeline().setStages(Array(rFormula, lr))

val pipelineModel = pipeline.fit(trainDF) val predDF
= pipelineModel.transform(testDF)
predDF.select("características", "precio", "predicción").show(5)

+-----+-----+
|      características|precio|      predicción|
+-----+-----+
|(98,[0,3,6,7,23,4...| 85,0| 55,80250714362137| |(98,
[0,3,6,7,23,4...| 45,0| 22,74720286761658| |( 98,
[0,3,6,7,23,4...| 70,0| 27,115811183814913|
```

```
|(98,[0,3,6,7,13,4...]|128.0|-91.60763412465076|)(98,
[0,3,6,7,13,4...]|159.0| 94.70374072351933|
+-----+-----+
```

Como puede ver, la columna de funciones se representa como `SparseVector`. Hay 98 funciones después de la codificación one-hot, seguidas de los índices distintos de cero y luego los valores mismos. Puede ver la salida completa si pasa `truncate=False` a `show()`.

¿Cómo está funcionando nuestro modelo? Puede ver que, si bien algunas de las predicciones pueden considerarse "cercanas", otras están lejos (¿un precio negativo para un alquiler?). A continuación, evaluaremos numéricamente qué tan bien funciona nuestro modelo en todo nuestro conjunto de pruebas.

Evaluación de modelos

Ahora que hemos construido un modelo, necesitamos evaluar qué tan bien funciona. En `spark.ml` hay evaluadores de clasificación, regresión, agrupación y clasificación (introducidos en Spark 3.0). Dado que se trata de un problema de regresión, utilizaremos el **error cuadrático medio (RMSE)** y rendimiento **R²** (pronunciado "R-squared") para evaluar el modelo de

RMSE

RMSE es una métrica que va de cero a infinito. Cuanto más cerca esté de cero, mejor.

Repasemos la fórmula matemática paso a paso:

1. Calcule la diferencia (o error) entre el valor verdadero y_i y el valor pronosticado \hat{y}_i (pronunciado y -sombrero, donde el "sombrero" indica que es un valor pronosticado de la cantidad debajo del sombrero):

$$\text{Error} = y_i - \hat{y}_i$$

2. Elevar al cuadrado la diferencia entre y_i y \hat{y}_i para que nuestros residuos positivos y negativos no se cancelen. Esto se conoce como el error al cuadrado:

$$\text{Error cuadrático (SE)} = (y_i - \hat{y}_i)^2$$

3. Luego sumamos el error cuadrático para todos los n de nuestros puntos de datos, conocido como la suma de los errores cuadráticos (SSE) o la suma de los residuos cuadráticos:

$$\text{Suma de errores al cuadrado (SSE)} = \sum_{i=1}^n (y_i - \hat{y}_i)^2$$

4. Sin embargo, el SSE crece con la cantidad de registros n en el conjunto de datos, por lo que queremos normalizarlo por la cantidad de registros. Nos da el error cuadrático medio (MSE), una métrica de regresión muy utilizada:

$$\text{Error cuadrático medio (MSE)} = \frac{1}{n} \sum_{y_0=1}^n (y_i - \hat{y}_i)^2$$

5. Si nos detenemos en MSE, nuestro término de error está en la escala de la unidad². A menudo tomamos la raíz cuadrada del MSE para devolver el error a la escala de la unidad original, lo que nos da el error cuadrático medio (RMSE):

$$\text{Error cuadrático medio (RMSE)} = \sqrt{\frac{1}{n} \sum_{y_0=1}^n (y_i - \hat{y}_i)^2}$$

Evaluemos nuestro modelo usando RMSE:

```
# En Python
desde pyspark.ml.e Evaluation import RegressionEvaluator
regressionEvaluator = RegressionEvaluator(
    PredicciónCol="predicción",
    labelCol="precio", metricName="rmse")
rmse =
regressionEvaluator.evaluate(predDF) print(f"RMSE es
{rmse:.1f}")

// En Scala
importar org.apache.spark.ml.evaluacion.RegressionEvaluator val
regressionEvaluator = new
```

```
RegressionEvaluator() .setPredictionCol("prediction") .setLabelCol("price") .setMetricName("rmse") val rmse = regressionEvaluator.
```

Esto produce la siguiente salida:

RME es 220,6

Interpretación del valor de RMSE. Entonces, ¿cómo sabemos si 220,6 es un buen valor para el RMSE? Hay varias formas de interpretar este valor, una de las cuales es construir un modelo de referencia simple y calcular su RMSE para compararlo. Un modelo de referencia común para las tareas de regresión es calcular el valor promedio de la etiqueta en el conjunto de entrenamiento \bar{y} (barra y pronunciada), luego predecir \hat{y} para cada registro en el conjunto de datos de prueba y calcular el RMSE resultante (el código de ejemplo es disponible en el repositorio de [GitHub del libro](#)). Si prueba esto, verá que nuestro modelo de referencia tiene un RMSE de 240,7, por lo que superamos nuestro

base. Si no supera la línea de base, es probable que algo haya fallado en el proceso de creación de su modelo.



Si se trata de un problema de clasificación, es posible que desee predecir la clase más prevalente como su modelo de referencia.

Tenga en cuenta que la unidad de su etiqueta impacta directamente en su RMSE. Por ejemplo, si su etiqueta es altura, entonces su RMSE será mayor si usa centímetros en lugar de metros como unidad de medida. Podría disminuir arbitrariamente el RMSE usando una unidad diferente, por lo que es importante comparar su RMSE con una línea de base.

También hay algunas métricas que, naturalmente, le dan una intuición de cómo se está desempeñando en comparación con una línea de base, como R^2 , que comentamos a continuación.

R 2

A pesar del nombre R^2 , que contiene "cuadrado", los valores de R^2 van desde infinito negativo hasta 1. echemos un vistazo a las matemáticas detrás de esta métrica. R^2 se calcula de la siguiente manera:

$$R^2 = 1 - \frac{SS_{\text{res}}}{SS_{\text{tot}}}$$

donde SS_{tot} es la suma total de cuadrados si siempre predices \bar{y} :

$$SS_{\text{tot}} = \sum_{y_0=1}^{n_{\text{obs}}} (\bar{y} - y_{0, \text{obs}})^2$$

y SS_{res} es la suma de los residuos al cuadrado de las predicciones de su modelo (también conocida como la suma de los errores al cuadrado, que usamos para calcular el RMSE):

$$SS_{\text{res}} = \sum_{y_0=1}^{n_{\text{obs}}} (\hat{y}_{0, \text{pred}} - y_{0, \text{obs}})^2$$

Si su modelo predice perfectamente todos los puntos de datos, entonces su $SS_{\text{res}} = 0$, lo que hace que su $R^2 = 1$. Y si su $SS_{\text{res}} = SS_{\text{tot}}$, entonces la fracción es $1/1$, por lo que su R^2 es 0 . ~~Este resultado se realiza de si mismo~~ que siempre prediciendo el valor promedio, \bar{y} .

Pero, ¿qué sucede si su modelo funciona peor que siempre prediciendo \bar{y} y su SS_{res} es realmente grande? Entonces tu R^2 es negativa, debería R^2 en realidad puede ser negativo! Si tu R^2

reevalúe su proceso de modelado. Lo bueno de usar R necesariamente necesitá es que tu no definir un modelo de referencia para comparar.

Si queremos cambiar nuestro evaluador de regresión para usar el R^2 , en lugar de redefinir el evaluador de regresión R, podemos establecer el nombre de la métrica usando la propiedad setter:

En Python

```
r2 = regressionEvaluator.setMetricName("r2").evaluate(predDF) print(f"R2 es {r2}")
```

// En Scala

```
val r2 = regressionEvaluator.setMetricName("r2").evaluate(predDF) println(s"R2 es $r2")
```

La salida es:

R2 es 0.159854

nuestra R^2 es positivo, pero está muy cerca de 0. Una de las razones por las que nuestro modelo no está funcionando demasiado bien es porque nuestra etiqueta, el precio, parece tener **una distribución logarítmica normal**. Si una distribución es log-normal, significa que si tomamos el logaritmo del valor, el resultado parece una distribución normal. El precio a menudo tiene una distribución logarítmica normal. Si piensas en los precios de alquiler en San Francisco, la mayoría cuesta alrededor de \$200 por noche, ¡pero hay algunos que alquilan por miles de dólares la noche! Puede ver la distribución de nuestros precios de Airbnb para nuestro conjunto de datos de capacitación en la [Figura 10-7](#).

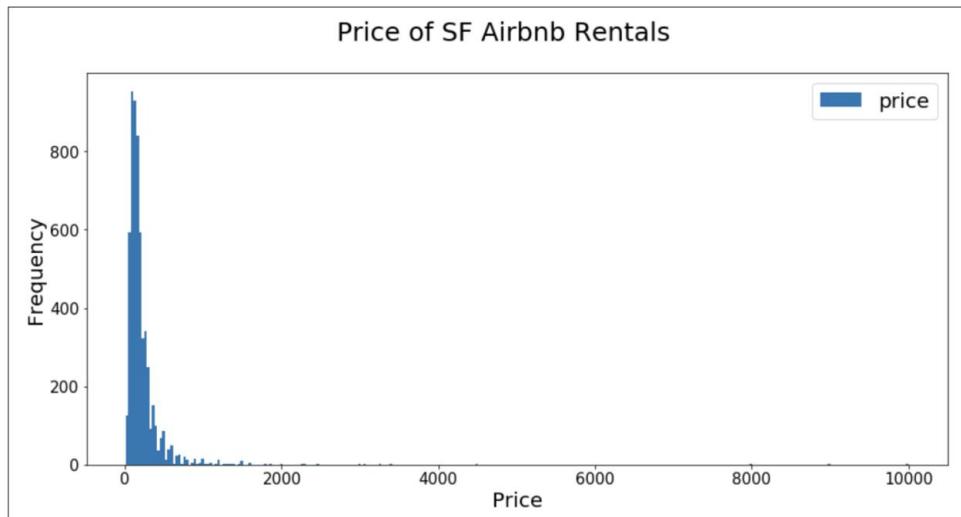


Figura 10-7. Distribución de precios de vivienda en San Francisco

Echemos un vistazo a la distribución resultante si, en cambio, miramos el logaritmo del precio ([Figura 10-8](#)).



Figura 10-8. Distribución del precio logarítmico de la vivienda en San Francisco

Puede ver aquí que nuestra distribución de precios logarítmicos se parece un poco más a una distribución normal. Como ejercicio, intente construir un modelo para predecir el precio en la escala logarítmica, luego exponga la predicción para sacarla de la escala logarítmica y evalúe su modelo. El código también se puede encontrar en el cuaderno de este capítulo en el repositorio de [GitHub del libro](#). Debería ver que su RMSE disminuye y su R

² aumenta para este conjunto de datos.

Guardar y cargar modelos

Ahora que hemos construido y evaluado un modelo, guardémoslo en un almacenamiento persistente para reutilizarlo más tarde (o en el caso de que nuestro clúster se caiga, no tenemos que volver a calcular el modelo). Guardar modelos es muy similar a escribir DataFrames: la API es `model.write().save(path)`. Opcionalmente, puede proporcionar el comando `overwrite()` para sobrescribir cualquier dato contenido en esa ruta:

```
# En Python
pipelinePath = "/tmp/lr-pipeline-model"
pipelineModel.write().overwrite().save(pipelinePath)

// En Scala
val pipelinePath = "/tmp/lr-pipeline-model"
pipelineModel.write.overwrite().save(pipelinePath)
```

Cuando carga sus modelos guardados, debe especificar el tipo de modelo que está cargando nuevamente (por ejemplo, ¿era un Modelo de regresión lineal o un Modelo de regresión logística ?). Por esta razón, le recomendamos que siempre coloque sus transformadores/estimadores en un Pipeline, de modo que para todos sus modelos cargue un PipelineModel y solo necesite cambiar la ruta del archivo al modelo:

```
# En Python
desde pyspark.ml importar PipelineModel
guardadoPipelineModel = PipelineModel.load(pipelinePath)

// En Scala
importar org.apache.spark.ml.PipelineModel
val guardadoPipelineModel = PipelineModel.load(pipelinePath)
```

Después de cargar, puede aplicarlo a nuevos puntos de datos. Sin embargo, no puede usar los pesos de este modelo como parámetros de inicialización para entrenar un nuevo modelo (a diferencia de comenzar con pesos aleatorios), ya que Spark no tiene el concepto de "arranque en caliente". Si su conjunto de datos cambia ligeramente, tendrá que volver a entrenar todo el modelo de regresión lineal desde cero.

Con nuestro modelo de regresión lineal construido y evaluado, exploremos cómo funcionan algunos otros modelos en nuestro conjunto de datos. En la siguiente sección, exploraremos modelos basados en árboles y veremos algunos hiperparámetros comunes para ajustar con el fin de mejorar el rendimiento del modelo.

Ajuste de hiperparámetros

Cuando los científicos de datos hablan de ajustar sus modelos, a menudo hablan de ajustar hiperparámetros para mejorar el poder predictivo del modelo. Un hiperparámetro es un atributo que define sobre el modelo antes del entrenamiento y no se aprende durante el proceso de entrenamiento (no debe confundirse con los parámetros, que se aprenden en el proceso de entrenamiento). La cantidad de árboles en su bosque aleatorio es un ejemplo de un hiperparámetro.

En esta sección, nos centraremos en el uso de modelos basados en árboles como ejemplo de procedimientos de ajuste de hiperparámetros, pero los mismos conceptos también se aplican a otros modelos. Una vez que configuremos la mecánica para realizar el ajuste de hiperparámetros con spark.ml, analizaremos las formas de optimizar la canalización. Comencemos con una breve introducción a los árboles de decisión, seguido de cómo podemos usarlos en spark.ml.

Modelos basados en árboles

Los modelos basados en árboles, como los árboles de decisión, los árboles potenciados por gradientes y los bosques aleatorios, son modelos relativamente simples pero potentes que son fáciles de interpretar (es decir, es fácil explicar las predicciones que hacen). Por lo tanto, son bastante populares para tareas de aprendizaje automático. Llegaremos a los bosques aleatorios en breve, pero primero debemos cubrir los fundamentos de los árboles de decisión.

Árboles de decisión

Como solución lista para usar, los árboles de decisión se adaptan bien a la minería de datos. Son relativamente rápidos de construir, altamente interpretables y de escala invariable (es decir, estandarizar o escalar las características numéricas no cambia el rendimiento del árbol). Entonces, ¿qué es un árbol de decisión?

Un árbol de decisión es una serie de reglas if-then-else aprendidas de sus datos para tareas de clasificación o regresión. Supongamos que estamos tratando de construir un modelo para predecir si alguien aceptará o no una oferta de trabajo, y las características incluyen salario, tiempo de viaje, café gratis, etc. Si ajustamos un árbol de decisión a este conjunto de datos, podríamos obtener un modelo que se parece a la Figura 10-9.

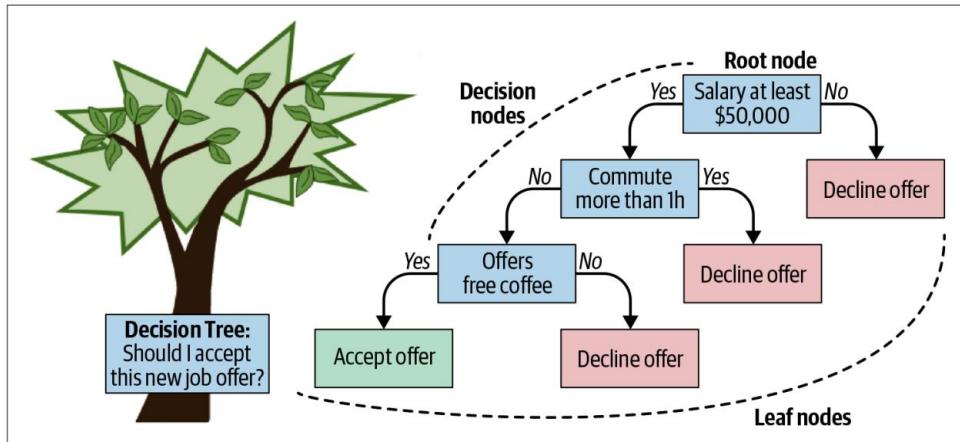


Figura 10-9. Ejemplo de árbol de decisión

El nodo en la parte superior del árbol se llama la "raíz" del árbol porque es la primera característica en la que nos "dividimos". Esta característica debería brindar la división más informativa; en este caso, si el salario es inferior a \$ 50,000, la mayoría de los candidatos rechazarán la oferta de trabajo. El nodo "Rechazar oferta" se conoce como "nodo hoja" ya que no hay otras divisiones que salgan de ese nodo; está al final de una rama. (Sí, es un poco divertido que lo llamemos "árbol" de decisión, ¡pero dibuje la raíz del árbol en la parte superior y las hojas en la parte inferior!)

Sin embargo, si el salario ofrecido es superior a \$50 000, pasamos a la siguiente característica más informativa del árbol de decisiones, que en este caso es el tiempo de viaje. Incluso si el salario supera los \$ 50,000, si el viaje es más largo que una hora, la mayoría de las personas rechazarán la oferta de trabajo.



No entraremos en detalles sobre cómo determinar qué características le darán la mayor ganancia de información aquí, pero si está interesado, consulte el Capítulo 9 de [Los elementos del aprendizaje estadístico](#), por Trevor Hastie, Robert Tibshirani y Jerome Friedman (Springer).

La última característica de nuestro modelo es el café gratis. En este caso, el árbol de decisión muestra que si el salario es superior a \$ 50,000, el viaje es menos de una hora y hay café gratis, entonces la mayoría de la gente aceptará nuestra oferta de trabajo (¡si fuera tan simple!). Como recurso de seguimiento, [R2D3](#) tiene una gran visualización de cómo funcionan los árboles de decisión.



Es posible dividir en la misma función varias veces en un solo árbol de decisión, pero cada división se producirá con un valor diferente.

La profundidad de un árbol de decisión es el camino más largo desde el nodo raíz hasta cualquier nodo hoja dado. En la [Figura 10-9](#), la profundidad es tres. Los árboles que son muy profundos son propensos a sobreajustarse o memorizar ruido en su conjunto de datos de entrenamiento, pero los árboles que son demasiado superficiales no se adaptarán bien a su conjunto de datos (es decir, podrían haber captado más señal de los datos).

Con la esencia de un árbol de decisión explicado, reanudemos el tema de la preparación de características para los árboles de decisión. Para los árboles de decisión, no tiene que preocuparse por estandarizar o escalar sus características de entrada, porque esto no tiene impacto en las divisiones, pero sí debe tener cuidado con la forma en que prepara sus características categóricas.

Los métodos basados en árboles pueden manejar naturalmente variables categóricas. En spark.ml, solo necesita pasar las columnas categóricas al StringIndexer, y el árbol de decisiones puede encargarse del resto. Ajustemos un árbol de decisión a nuestro conjunto de datos:

```
# En Python
desde pyspark.ml.regression import DecisionTreeRegressor

dt = DecisionTreeRegressor(labelCol="precio")

# Filtre solo para columnas numéricas (y excluya el precio, nuestra etiqueta)
numericCols = [field for (field, dataType) in trainDF.dtypes if ((dataType ==
    "doble") & (field != "price"))]

# Combine la salida de StringIndexer definida anteriormente y las columnas
# numéricas
ensambladorInputs = indexOutputCols + numericCols
vecAssembler = VectorAssembler(inputCols=ensambladorInputs, outputCol="características")

# Combinar etapas en etapas de
canalización = [ stringIndexer, vecAssembler, dt]
```

```

pipeline = Pipeline(stages=stages)
pipelineModel = pipeline.fit(trainDF) # Esta línea debería generar un error

// En Scala
import org.apache.spark.ml.regression.DecisionTreeRegressor

val dt = new
DecisionTreeRegressor() .setLabelCol("precio")

// Filtra solo para columnas numéricas (y excluye el precio, nuestra etiqueta) val
numericCols = trainDF.dtypes.filter{ case (field, dataType) => dataType ==
"DoubleType" && field != "price"}.map(_._1)

// Combina la salida de StringIndexer definida anteriormente y las columnas numéricas
val ensambladorInputs = indexOutputCols ++ numericCols val vecAssembler = new
VectorAssembler() .setInputCols(ensambladorInputs) .setOutputCol("características")

// Combinar etapas en canalización
val etapas = Array(stringIndexer, vecAssembler, dt) val
canalización = new Pipeline() .setStages(etapas)

val pipelineModel = pipeline.fit(trainDF) // Esta línea debería generar un error

```

Esto produce el siguiente error:

java.lang.IllegalArgumentException: el requisito falló: DecisionTree requiere que maxBins (= 32) sea al menos tan grande como la cantidad de valores en cada característica categórica, pero la característica categórica 3 tiene 36 valores. Consideré eliminar esta y otras características categóricas con una gran cantidad de valores, o agregue más ejemplos de capacitación.

Podemos ver que hay un problema con el parámetro maxBins . ¿Qué hace ese parámetro? maxBins determina la cantidad de contenedores en los que se discretizan o dividen sus entidades continuas. Este paso de discretización es crucial para realizar el entrenamiento distribuido. No hay un parámetro maxBins en scikit -learn porque todos los datos y el modelo residen en una sola máquina. En Spark, sin embargo, los trabajadores tienen todas las columnas de los datos, pero solo un subconjunto de las filas. Por lo tanto, al comunicarnos sobre qué características y valores dividir, debemos asegurarnos de que todos estén hablando de los mismos valores de división, que obtenemos de la discretización común configurada en el momento del entrenamiento. Echemos un vistazo a la [Figura 10-10](#), que muestra el [PLANETA](#) implementación de árboles de decisión distribuidos, para obtener una mejor comprensión del aprendizaje automático distribuido e ilustrar el parámetro maxBins .

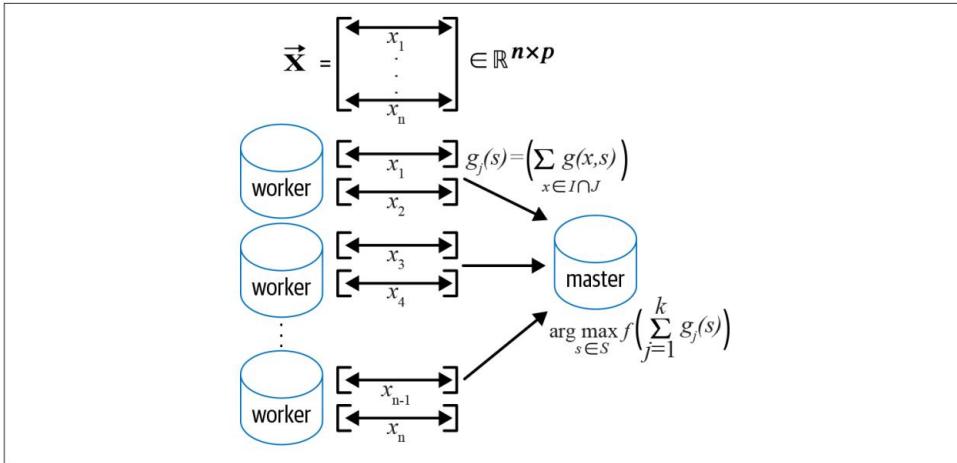


Figura 10-10. PLANET implementación de árboles de decisión distribuidos (fuente: <https://oreil.ly/RAvP>)

Cada trabajador tiene que calcular estadísticas de resumen para cada función y cada punto de división posible, y esas estadísticas se agregarán entre los trabajadores. MLlib requiere que maxBins sea lo suficientemente grande para manejar la discretización de las columnas categóricas. El valor predeterminado para maxBins es 32 y teníamos una columna categórica con 36 valores distintos, por lo que obtuvimos el error antes. Si bien podríamos aumentar maxBins a 64 para representar con mayor precisión nuestras características continuas, eso duplicaría la cantidad de divisiones posibles para las variables continuas, lo que aumentaría en gran medida nuestro tiempo de cálculo. En su lugar, establezcamos maxBins en 40 y volvamos a entrenar la canalización. Notará aquí que estamos usando el método setter setMaxBins() para modificar el árbol de decisiones en lugar de redefinirlo por completo:

```
# En Python
dt.setMaxBins(40)
pipelineModel = pipeline.fit(trainDF)

// En Scala
dt.setMaxBins(40)
val pipelineModel = pipeline.fit(trainDF)
```



Debido a las diferencias en la implementación, a menudo no obtendrá exactamente los mismos resultados al construir un modelo con scikit-learn versus MLlib. Sin embargo, está bien. La clave es comprender por qué son diferentes y ver qué parámetros están bajo su control para que funcionen de la manera que los necesita. Si está transfiriendo cargas de trabajo de scikit-learn a MLlib, le recomendamos que eche un vistazo a [spark.ml](#) y [scikit-aprender](#) documentación para ver qué parámetros difieren y ajustar esos parámetros para obtener resultados comparables para los mismos datos. Una vez que los valores estén lo suficientemente cerca, puede escalar su modelo MLlib a tamaños de datos más grandes que scikit-learn no puede manejar.

Ahora que hemos construido con éxito nuestro modelo, podemos extraer las reglas if-then-else aprendidas por el árbol de decisión:

```
# En Python
dtModel = pipelineModel.stages[-1]
print(dtModel.toDebugString)

// En Scala
val dtModel =
  pipelineModel.stages.last.asInstanceOf[org.apache.spark.ml.regression.DecisionTreeRegressionModel]
println(dtModel.toDebugString)

DecisionTreeRegressionModel: uid=dtr_005040f1efac, profundidad=5, numNodes=47, ...
  Si (característica 12 <= 2.5)
  Si (característica 12 <= 1.5)
    Si (función 5 en {1.0,2.0})
      Si (función 4 en {0.0,1.0,3.0,5.0,9.0,10.0,11.0,13.0,14.0,16.0,18.0,24.0})
        Si (función 3 en
{0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,11.0,12.0,13.0,14.0,...})
          Predecir: 104.23992784125075
        De lo contrario (función 3 no en {0.0,1.0,2.0,3.0,4.0,5.0,6.0,7.0,8.0,9.0,10.0,...})
          Predecir: 250.7111111111111
  ...

```

Esto es solo un subconjunto de la impresión, pero notará que es posible dividir en la misma función más de una vez (p. ej., función 12), pero con diferentes valores de división. Observe también la diferencia entre cómo se divide el árbol de decisión en características numéricas frente a características categóricas: para características numéricas, comprueba si el valor es menor o igual que el umbral, y para características categóricas, comprueba si el valor está en ese conjunto o no.

También podemos extraer las puntuaciones de importancia de las características de nuestro modelo para ver las características más importantes:

```
# En Python
importar pandas como pd

featureImp =
pd.DataFrame( list(zip(vecAssembler.getInputCols(), dtModel.featureImportances)),
```

```

columnas = ["característica", "importancia"])
featureImp.sort_values(por="importancia", ascendente=False)

// En escala
val featureImp = vecAssembler
    .getInputCols.zip(dtModel.featureImportances.toArray)
val columnas = Array("característica", "Importancia")
val featureImpDF = chispa.createDataFrame(featureImp).toDF(columnas: _*)

featureImpDF.orderBy($"Importancia".desc).show()

```

Rasgo	Importancia
dormitorios	0.283406
cancel_policyIndex	0.167893
instant_bookableIndex	0.140081
property_typeIndex	0.128179
number_of_reviews	0.126233
vecindario_cleansedIndex	0.056200
longitud	0.038810
mínimo_noches	0.029473
camas	0.015218
indice_tipo_habitación	0.010905
acomoda	0.003603

Si bien los árboles de decisión son muy flexibles y fáciles de usar, no siempre son los más modelo preciso. Si tuviéramos que calcular nuestra R^2 en el conjunto de datos de prueba, en realidad obtener una puntuación negativa! Eso es peor que solo predecir el promedio. (Puedes ver esto en el cuaderno de este capítulo en el repositorio de [GitHub del libro](#)).

Veamos cómo mejorar este modelo mediante el uso de un enfoque de conjunto que combina diferentes diferentes modelos para lograr un mejor resultado: bosques aleatorios.

Bosques aleatorios

Conjuntos trabajar adoptando un enfoque democrático. Imagina que hay muchos M&M en un frasco. Le pides a cien personas que adivinen el número de M&M y luego tomas la promedio de todas las conjeturas. El promedio probablemente esté más cerca del valor real que la mayoría de las conjeturas individuales. Ese mismo concepto se aplica a los modelos de aprendizaje automático. Si construyes muchos modelos y combinás/promedias sus predicciones, serán más robustos que los producidos por cualquier modelo individual.

Bosques aleatorios son un conjunto de árboles de decisión con dos ajustes clave:

Bootstrapping de muestras por filas

Bootstrapping es una técnica para simular nuevos datos mediante el muestreo con reemplazo de sus datos originales. Cada árbol de decisión se entrena en una muestra de arranque diferente de su conjunto de datos, lo que produce árboles de decisión ligeramente diferentes, y luego agrega sus predicciones. Esta técnica se conoce como **agregación bootstrap**, o embolsado. En una implementación típica de bosque aleatorio, cada árbol muestra la misma cantidad de puntos de datos con el reemplazo del conjunto de datos original, y esa cantidad se puede controlar a través del parámetro `subsamplingRate`.

Selección aleatoria de características por columnas

El principal inconveniente del embolsado es que todos los árboles están altamente correlacionados y, por lo tanto, aprenden patrones similares en sus datos. Para mitigar este problema, cada vez que deseé realizar una división, solo considere un subconjunto aleatorio de las columnas ($1/3$ de las funciones para `RandomForestRegressor` y `#features` para `RandomForestClassifier`). Debido a esta aleatoriedad que introduce, por lo general desea que cada árbol sea bastante poco profundo. Podría estar pensando: cada uno de estos árboles funcionará peor que cualquier árbol de decisión único, entonces, ¿cómo podría ser mejor este enfoque? Resulta que cada uno de los árboles aprende algo diferente sobre su conjunto de datos, y la combinación de esta colección de aprendices "débiles" en un conjunto hace que el bosque sea mucho más sólido que un solo árbol de decisión.

La figura 10-11 ilustra un bosque aleatorio en tiempo de entrenamiento. En cada división, considera 3 de las 10 características originales para dividir; finalmente, elige lo mejor de entre ellos.

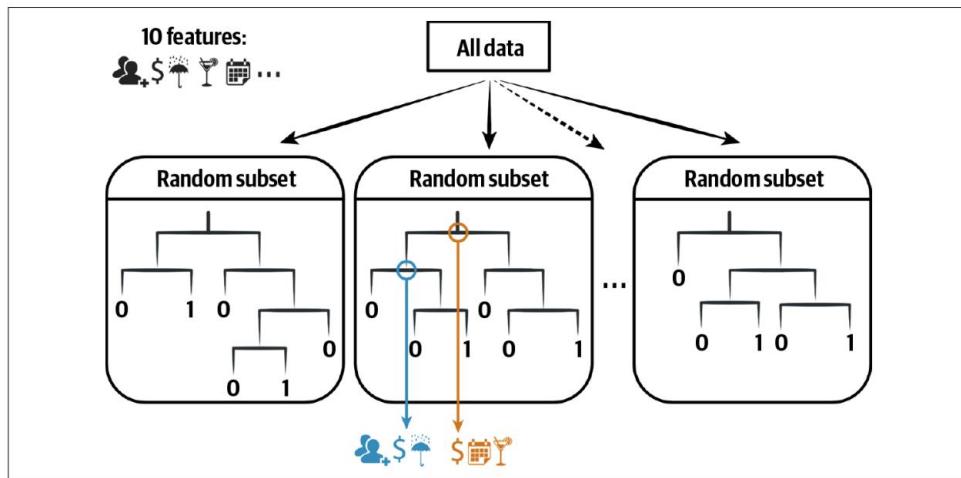


Figura 10-11. Entrenamiento de bosque aleatorio

Las API para bosques aleatorios y árboles de decisión son similares y ambos se pueden aplicar a tareas de regresión o clasificación:

```
# En Python
desde pyspark.ml.regression import RandomForestRegressor rf
= RandomForestRegressor(labelCol="price", maxBins=40, seed=42)

// En Scala
import org.apache.spark.ml.regression.RandomForestRegressor val
rf = new
RandomForestRegressor() .setLabelCol("price") .setMaxBins(40) .setSeed(42)
```

Una vez que haya entrenado su bosque aleatorio, puede pasar nuevos puntos de datos a través de los diferentes árboles entrenados en el conjunto.

Como muestra la [figura 10-12](#), si crea un bosque aleatorio para la clasificación, pasa el punto de prueba a través de cada uno de los árboles del bosque y recibe una mayoría de votos entre las predicciones de los árboles individuales. (Por el contrario, en la regresión, el bosque aleatorio simplemente promedia esas predicciones). Aunque cada uno de estos árboles tiene menos rendimiento que cualquier árbol de decisión individual, la colección (o conjunto) en realidad proporciona un modelo más sólido.

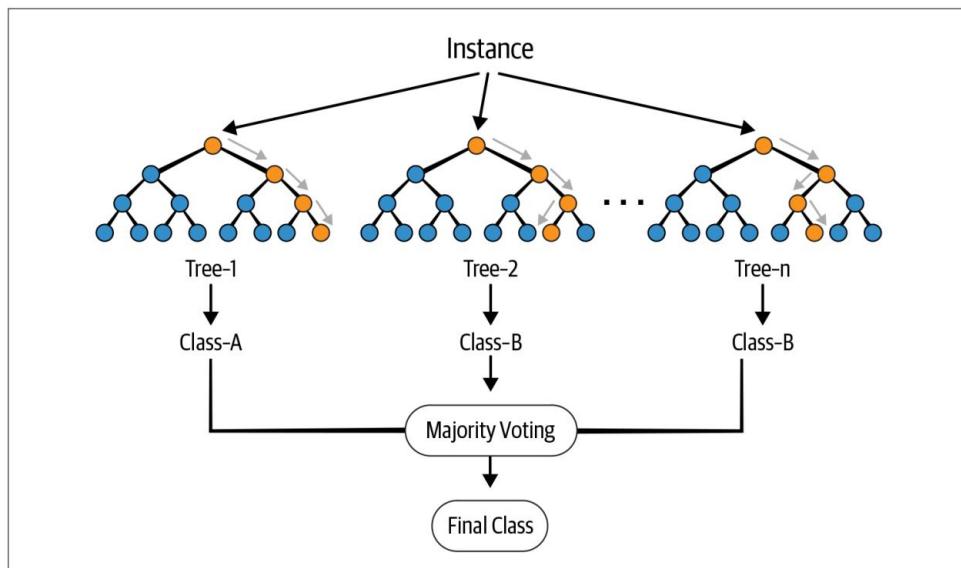


Figura 10-12. Predicciones aleatorias de bosques

Los bosques aleatorios realmente demuestran el poder del aprendizaje automático distribuido con Spark, ya que cada árbol se puede construir independientemente de los otros árboles (por ejemplo, no necesita construir el árbol 3 antes de construir el árbol 10). Además, dentro de cada nivel del árbol, puede paralelizar el trabajo para encontrar las divisiones óptimas.

Entonces, ¿cómo determinamos cuál debe ser la cantidad óptima de árboles en nuestro bosque aleatorio o la profundidad máxima de esos árboles? Este proceso se denomina **ajuste de hiperparámetros**.

A diferencia de un parámetro, un hiperparámetro es un valor que controla el proceso de aprendizaje o la estructura de su modelo, y no se aprende durante el entrenamiento. Tanto el número de árboles como la profundidad máxima son ejemplos de hiperparámetros que puede ajustar para bosques aleatorios. Ahora cambiemos nuestro enfoque a cómo podemos descubrir y evaluar el mejor modelo de bosque aleatorio ajustando algunos hiperparámetros.

Validación cruzada de pliegues k

¿Qué conjunto de datos debemos usar para determinar los valores óptimos de hiperparámetros? Si usamos el conjunto de entrenamiento, es probable que el modelo se sobreajuste o memorice los matices de nuestros datos de entrenamiento. Esto significa que será menos probable generalizar a datos no vistos. Pero si usamos el conjunto de prueba, eso ya no representará datos "no vistos", por lo que no podremos usarlo para verificar qué tan bien se generaliza nuestro modelo. Por lo tanto, necesitamos otro conjunto de datos que nos ayude a determinar los hiperparámetros óptimos: el conjunto de datos de validación.

Por ejemplo, en lugar de dividir nuestros datos en una división de entrenamiento/prueba 80/20, como hicimos anteriormente, podemos hacer una división 60/20/20 para generar conjuntos de datos de capacitación, validación y prueba, respectivamente. Luego, podemos construir nuestro modelo en el conjunto de entrenamiento, evaluar el rendimiento en el conjunto de validación para seleccionar la mejor configuración de hiperparámetros y aplicar el modelo al conjunto de prueba para ver qué tan bien se desempeña en los datos nuevos. Sin embargo, una de las desventajas de este enfoque es que perdemos el 25 % de nuestros datos de entrenamiento (80 % -> 60 %), que podrían haberse utilizado para ayudar a mejorar el modelo. Esto motiva el uso de la técnica de validación cruzada k-fold para resolver este problema.

Con este enfoque, en lugar de dividir el conjunto de datos en conjuntos separados de entrenamiento, validación y prueba, lo dividimos en conjuntos de entrenamiento y prueba como antes, pero usamos los datos de entrenamiento tanto para el entrenamiento como para la validación. Para lograr esto, dividimos nuestros datos de entrenamiento en k subconjuntos o "pliegues" (p. ej., tres). Luego, para una configuración de hiperparámetro dada, entrenamos nuestro modelo en $k-1$ pliegues y evaluamos en el pliegue restante, repitiendo este proceso k veces. [La figura 10-13](#) ilustra este enfoque.

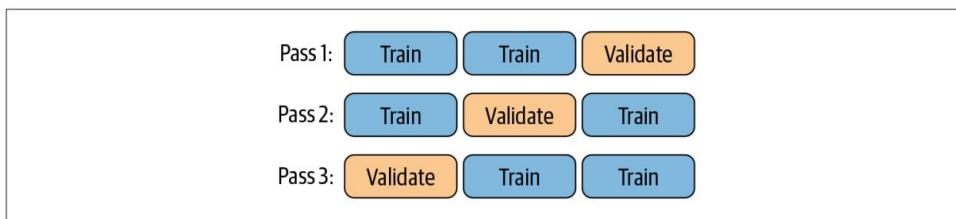


Figura 10-13. validación cruzada k-fold

Como muestra esta figura, si dividimos nuestros datos en tres partes, nuestro modelo primero se entrena en la primera y segunda parte (o divisiones) de los datos y se evalúa en la tercera parte. Luego construimos el mismo modelo con los mismos hiperparámetros en el primer y tercer pliegue.

de los datos y evaluar su rendimiento en el segundo pliegue. Por último, construimos el modelo en el segundo y tercer pliegue y lo evaluamos en el primer pliegue. Luego promediamos el rendimiento de esos tres (o k) conjuntos de datos de validación como un indicador de qué tan bien funcionará este modelo en datos no vistos, ya que cada punto de datos tuvo la oportunidad de ser parte del conjunto de datos de validación exactamente una vez. A continuación, repetimos este proceso para todas nuestras diferentes configuraciones de hiperparámetros para identificar la óptima.

Determinar el espacio de búsqueda de sus hiperparámetros puede ser difícil y, a menudo, realizar una búsqueda aleatoria de hiperparámetros **superar a una búsqueda de cuadrícula estructurada**.

Hay bibliotecas especializadas, como [Hyperopt](#), para ayudarlo a identificar las configuraciones óptimas de **hiperparámetros**, que tratamos en el [capítulo 11](#).

Para realizar una búsqueda de hiperparámetros en Spark, siga los siguientes pasos:

1. Defina el estimador que desea evaluar.
2. Especifique qué hiperparámetros desea variar, así como sus respectivos valores, utilizando [ParamGridBuilder](#).
3. Defina un evaluador para especificar qué métrica usar para comparar los diversos modelos
4. Usa el [CrossValidator](#) para realizar una validación cruzada, evaluando cada una de las variantes nuestros modelos.

Comencemos definiendo nuestro estimador de tubería:

```
# En Python
pipeline = Pipeline(etapas = [stringIndexer, vecAssembler, rf])

// En Scala val
pipeline = new
  Pipeline() .setStages(Array(stringIndexer, vecAssembler, rf))
```

Para nuestro ParamGridBuilder, variaremos nuestra maxDepth para que sea 2, 4 o 6 y numTrees (la cantidad de árboles en nuestro bosque aleatorio) para que sea 10 o 100. Esto nos dará una cuadrícula de 6 (3 x 2) diferentes Configuraciones de hiperparámetros en total:

```
(maxDepth=2, numTrees=10)
(maxDepth=2, numTrees=100)
(maxDepth=4, numTrees=10)
(maxDepth=4, numTrees=100)
(maxDepth=6, numTrees=10)
(maxDepth=6, numTrees=100)

# En Python
desde pyspark.ml.tuning import ParamGridBuilder paramGrid =
(ParamGridBuilder()
 .addGrid(rf.maxDepth, [2, 4,
 6]) .addGrid(rf.numTrees, [10, 100]) .build())
```

```
// En Scala
import org.apache.spark.ml.tuning.ParamGridBuilder val
paramGrid = new ParamGridBuilder() .addGrid(rf.maxDepth,
  Array(2, 4, 6)) .addGrid(rf.numTrees, Array(10,
  100)) .construir()
```

Ahora que hemos configurado nuestra cuadrícula de hiperparámetros, debemos definir cómo evaluar cada uno de los modelos para determinar cuál funcionó mejor. Para esta tarea, usaremos RegressionEvaluator y usaremos RMSE como nuestra métrica de interés:

```
# En el
evaluador de Python = RegressionEvaluator(labelCol="price",
  PredicciónCol="predicción",
  metricName="rmse")

// En Scala
val evaluador = new RegressionEvaluator()

.setLabelCol("precio") .setPredictionCol("predicción") .setMetricName("rmse")
```

Realizaremos nuestra validación cruzada de k-fold utilizando CrossValidator, que acepta un estimador, un evaluador y un estimadorParamMaps para que sepa qué modelo usar, cómo evaluar el modelo y qué hiperparámetros establecer para el modelo. También podemos establecer el número de pliegues en los que queremos dividir nuestros datos (numFolds=3), así como establecer una semilla para que tengamos divisiones reproducibles en los pliegues (semilla=42). Luego, ajustemos este validador cruzado a nuestro conjunto de datos de entrenamiento:

```
# En Python
desde pyspark.ml.tuning import CrossValidator

cv = CrossValidator(estimator=tubería,
  evaluador=evaluador,
  estimadorParamMaps=paramGrid,
  numFolds=3, seed=42) cvModel =
  cv.fit(trainDF)

// En Scala
importar org.apache.spark.ml.tuning.CrossValidator

val cv = new

CrossValidator() .setEstimator(pipeline) .setEvaluator(evaluador) .setEstimatorParamMaps(paramGrid) .setNumFo
```

La salida nos dice cuánto tiempo tomó la operación:

El comando tomó 1.07 minutos

Entonces, ¿cuántos modelos acabamos de entrenar? Si respondió 18 (6 configuraciones de hiperparámetros x validación cruzada de 3 veces), está cerca. Una vez que haya identificado la configuración óptima de hiperparámetros, ¿cómo combina esos tres (o k) modelos juntos? Si bien algunos modelos pueden ser bastante fáciles de promediar juntos, otros no lo son. Por lo tanto, Spark vuelve a entrenar su modelo en todo el conjunto de datos de entrenamiento una vez que ha identificado la configuración óptima de hiperparámetros, por lo que al final entrenamos 19 modelos. Si desea retener los modelos intermedios entrenados, puede configurar Collect SubModels=True en CrossValidator.

Para inspeccionar los resultados del validador cruzado, puede echar un vistazo a avgMetrics:

```
# En la lista
de Python(zip(cvModel.getEstimatorParamMaps(), cvModel.avgMetrics))

// En Scala
cvModel.getEstimatorParamMaps.zip(cvModel.avgMetrics)
```

Aquí está la salida:

```
res1: Matriz[({org.apache.spark.ml.param.ParamMap, Doble})] =
Matriz(({ rfr_a132fb1ab6c8-maxDepth: 2, rfr_a132fb1ab6c8-numTrees:
10 },303.99522869739343),
({ rfr_a132fb1ab6c8-2maxDepthfr:162ab6c8-2maxDepthfr:16
-numTrees: 100}, 299.56501993529474), ({rfr_a132fb1ab6c8-maxDepth:
4, rfr_a132fb1ab6c8-numTrees: 10}, 310.63687030886894),
({rfr_a132fb1ab6c8-maxDepth: 4, rfr_a132fb1ab6c8-numTrees:
100}, 294.7369599168999), ({rfr_a132fb1ab6c8-MAXDEPTH : 6,
rfr_a132fb1ab6c8-numTrees: 10 },312.6678169109293),
({ rfr_a132fb1ab6c8-maxDepth: 6, rfr_a132fb1ab6c8-numTrees:
100 },292.101039874209))
```

Podemos ver que el mejor modelo de nuestro CrossValidator (el que tiene el RMSE más bajo) tenía maxDepth=6 y numTrees=100. Sin embargo, esto tomó mucho tiempo para funcionar. En la siguiente sección, veremos cómo podemos disminuir el tiempo para entrenar nuestro modelo mientras mantenemos el mismo rendimiento del modelo.

Optimización de canalizaciones

Si su código tarda lo suficiente como para pensar en mejorarlo, entonces debe optimizarlo. En el código anterior, aunque cada uno de los modelos en el validador cruzado es técnicamente independiente, spark.ml en realidad entrena la colección de modelos secuencialmente en lugar de en paralelo. En Spark 2.3, un **parallelismo** El parámetro fue introducido para resolver este problema. Este parámetro determina el número de modelos a entrenar en paralelo, que a su vez se ajustan en paralelo. De la [guía de ajuste de chispa](#):

El valor del paralelismo debe elegirse cuidadosamente para maximizar el paralelismo sin exceder los recursos del clúster, y es posible que los valores más grandes no siempre conduzcan a un mejor rendimiento. En términos generales, un valor de hasta 10 debería ser suficiente para la mayoría de los clústeres.

Establezcamos este valor en 4 y veamos si podemos entrenar más rápido:

```
# En Python
cvModel = cv.setParallelism(4).fit(trainDF)

// En Scala val
cvModel = cv.setParallelism(4).fit(trainDF)
```

La respuesta es sí:

El comando tomó 31.45 segundos

Hemos reducido el tiempo de entrenamiento a la mitad (de 1,07 minutos a 31,45 segundos), ¡pero aún podemos mejorarla aún más! Hay otro truco que podemos usar para acelerar el entrenamiento del modelo: colocar el validador cruzado dentro de la canalización (p. ej., Pipeline(stages=[..., cv]) en lugar de colocar la canalización dentro del validador cruzado (p. ej., CrossValidator(esti mator=pipeline, ...)). Cada vez que el validador cruzado evalúa la canalización, ejecuta cada paso de la canalización para cada modelo, incluso si algunos de los pasos no cambian, como el StringIndexer. Por Al reevaluar cada paso en la canalización, estamos aprendiendo el mismo mapeo de StringIndexer una y otra vez, aunque no cambie.

Si, en cambio, colocamos nuestro validador cruzado dentro de nuestra tubería, entonces no estaremos reevaluando el StringIndexer (o cualquier otro estimador) cada vez que probemos un modelo diferente:

```
# En Python cv
= CrossValidator(estimador=rf, evaluador=evaluador,
estimadorParamMaps=paramGrid,
numFolds=3, parallelismo=4, semilla=42)
```

```
pipeline = Pipeline(etapas=[stringIndexer, vecAssembler, cv]) pipelineModel =
pipeline.fit(trainDF)

// En Scala val
cv = new CrossValidator()
```

```
.setEstimator(rf).setEvaluator(evaluador).setEstimatorParamMaps(paramGrid).setNumFolds(3).setParallelism(4).setSeed(42)

tubería val = nueva tubería ()
.setStages(Array(stringIndexer, vecAssembler, cv)) val pipelineModel =
pipeline.fit(trainDF)
```

Este recorta cinco segundos de nuestro tiempo de entrenamiento:

El comando tomó 26.21 segundos

Gracias al parámetro de paralelismo y a la reorganización del orden de nuestra canalización, esa última ejecución fue la más rápida y, si la aplica al conjunto de datos de prueba, verá que obtiene los mismos resultados. Aunque estas ganancias fueron del orden de segundos, las mismas técnicas se aplican a conjuntos de datos y modelos mucho más grandes, con un ahorro de tiempo correspondientemente mayor. Puede intentar ejecutar este código usted mismo accediendo al cuaderno en el repositorio de [GitHub del libro](#).

Resumen

En este capítulo, cubrimos cómo crear canalizaciones con Spark MLlib, en particular, su paquete de API basado en DataFrame, spark.ml. Discutimos las diferencias entre transformadores y estimadores, cómo componerlos usando la API Pipeline y algunas métricas diferentes para evaluar modelos. Luego exploramos cómo usar la validación cruzada para realizar el ajuste de hiperparámetros para ofrecer el mejor modelo, así como consejos para optimizar la validación cruzada y el entrenamiento del modelo en Spark.

Todo esto establece el contexto para el próximo capítulo, en el que analizaremos las estrategias de implementación y las formas de administrar y escalar las canalizaciones de aprendizaje automático con Spark.

CAPÍTULO 11

Gestión, implementación y escalado de máquinas

Canalizaciones de aprendizaje con Apache Spark

En el capítulo anterior, cubrimos cómo crear canalizaciones de aprendizaje automático con MLlib. Este capítulo se centrará en cómo administrar e implementar los modelos que entrena. Al final de este capítulo, podrá usar MLflow para rastrear, reproducir e implementar sus modelos MLlib, analizar las dificultades y las compensaciones entre varios escenarios de implementación de modelos y diseñar soluciones escalables de aprendizaje automático. Pero antes de analizar la implementación de modelos, analicemos primero algunas prácticas recomendadas para la administración de modelos a fin de preparar sus modelos para la implementación.

Gestión de modelos

Antes de implementar su modelo de aprendizaje automático, debe asegurarse de que puede reproducir y realizar un seguimiento del rendimiento del modelo. Para nosotros, la reproducibilidad de extremo a extremo de las soluciones de aprendizaje automático significa que debemos poder reproducir el código que generó un modelo, el entorno utilizado en el entrenamiento, los datos con los que se entrenó y el modelo en sí. A todos los científicos de datos les encanta recordarle que configure sus semillas para que pueda reproducir sus experimentos (p. ej., para la división de prueba/entrenamiento, cuando se usan modelos con aleatoriedad inherente, como bosques aleatorios). Sin embargo, hay muchos más aspectos que contribuyen a la reproducibilidad además del establecimiento de semillas, y algunos de ellos son mucho más sutiles. Aquí están algunos ejemplos:

Control de versiones

de la biblioteca Cuando un científico de datos le entrega su código, es posible que mencione o no las bibliotecas dependientes. Si bien puede averiguar qué bibliotecas se requieren revisando los mensajes de error, no estará seguro de qué versiones de biblioteca usaron, por lo que probablemente instalará las más recientes. Pero si su código se creó en una versión anterior de una biblioteca, que puede estar aprovechando algunos valores predeterminados

comportamiento que difiere de la versión que instaló, el uso de la última versión puede hacer que el código se rompa o que los resultados difieran (por ejemplo, considere cómo [XGBoost](#) cambió [la forma en que maneja los valores faltantes](#) en v0.90).

Evolución de los

datos Suponga que construye un modelo el 1 de junio de 2020 y realiza un seguimiento de todos sus hiperparámetros, bibliotecas, etc. Luego intenta reproducir el mismo modelo el 1 de julio de 2020, pero la canalización se rompe o los resultados difieren porque los datos subyacentes han cambiado, lo que podría suceder si alguien agregara una columna adicional o un orden de magnitud más de datos después de la compilación inicial.

Orden de ejecución

Si un científico de datos le entrega su código, debería poder ejecutarlo de arriba a abajo sin errores. Sin embargo, los científicos de datos son conocidos por ejecutar las cosas sin orden o ejecutar la misma celda con estado varias veces, lo que hace que sus resultados sean muy difíciles de reproducir. (¡También pueden registrar una copia del código con diferentes hiperparámetros que los que se usaron para entrenar el modelo final!)

Operaciones paralelas

Para maximizar el rendimiento, las GPU ejecutarán muchas operaciones en paralelo. Sin embargo, el orden de ejecución no siempre está garantizado, lo que puede generar resultados no deterministas. Este es un problema conocido con funciones como `tf.reduce_sum()` y al agregar números de punto flotante (que tienen una precisión limitada): el orden en que los agrega puede generar resultados ligeramente diferentes, que pueden exacerbarse en muchas iteraciones.

La incapacidad de reproducir sus experimentos a menudo puede ser un obstáculo para lograr que las unidades de negocios adopten su modelo o lo pongan en producción. Si bien puede crear sus propias herramientas internas para rastrear sus modelos, datos, versiones de dependencia, etc., pueden volverse obsoletas, frágiles y requerir un esfuerzo de desarrollo significativo para mantenerlas. Igualmente importante es tener estándares de toda la industria para administrar modelos de modo que puedan compartirse fácilmente con los socios. Existen herramientas de código abierto y propietarias que pueden ayudarnos a reproducir nuestros experimentos de aprendizaje automático al abstraer muchas de estas dificultades comunes. Esta sección se centrará en MLflow, ya que tiene la integración más estrecha con MLlib de las herramientas de gestión de modelos de código abierto actualmente disponibles.

ML bajo

[MLflow](#) es una plataforma de código abierto que ayuda a los desarrolladores a reproducir y compartir experimentos, administrar modelos y mucho más. Proporciona interfaces en Python, R y Java/Scala, así como una API REST. Como se muestra en la [Figura 11-1](#), MLflow tiene cuatro componentes principales:

Seguimiento

Proporciona API para registrar parámetros, métricas, versiones de código, modelos y artefactos como gráficos y texto.

Proyectos

Un formato estandarizado para empaquetar sus proyectos de ciencia de datos y sus dependencias para ejecutarlos en otras plataformas. Le ayuda a gestionar el proceso de formación de modelos.

Modelos

Un formato estandarizado para empaquetar modelos para implementar en diversos entornos de ejecución. Proporciona una API coherente para cargar y aplicar modelos, independientemente del algoritmo o la biblioteca utilizada para construir el modelo.

Registro

Un repositorio para realizar un seguimiento del linaje del modelo, las versiones del modelo, las transiciones de etapa y las anotaciones.

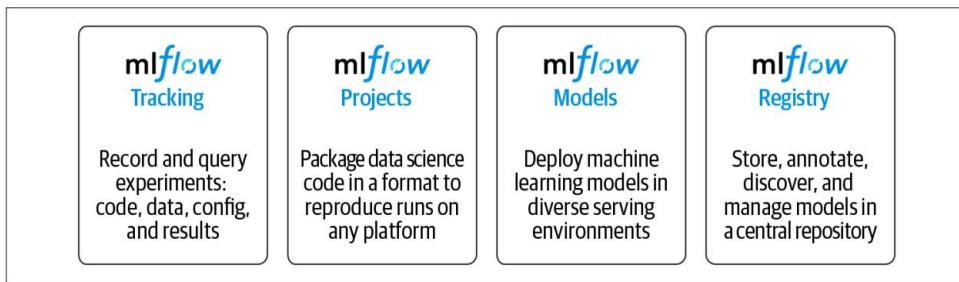


Figura 11-1. Componentes de flujo de ML

Hagamos un seguimiento de los experimentos del modelo MLlib que ejecutamos en el [Capítulo 10](#) para la reproducibilidad.

Luego veremos cómo entran en juego los otros componentes de MLflow cuando analicemos la implementación del modelo.

Para comenzar con MLflow, simplemente ejecute pip install mlflow en su host local.

Seguimiento MLflow Tracking es una API de registro que es independiente de las bibliotecas y los entornos que realmente realizan el entrenamiento. Está organizado en torno al concepto de ejecuciones, que son ejecuciones de código de ciencia de datos. Las ejecuciones se agregan en experimentos, de modo que muchas ejecuciones pueden ser parte de un experimento determinado.

El servidor de seguimiento de MLflow puede albergar muchos experimentos. Puede iniciar sesión en el servidor de seguimiento mediante un cuaderno, una aplicación local o un trabajo en la nube, como se muestra en la [Figura 11-2](#).

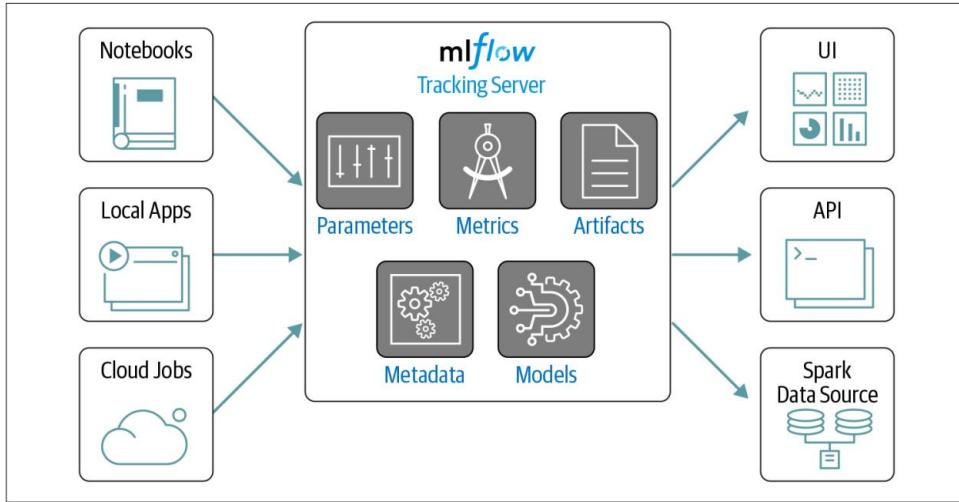


Figura 11-2. Servidor de seguimiento de flujo de aprendizaje automático

Examinemos algunas cosas que se pueden registrar en el servidor de seguimiento:

Parámetros

Entradas de clave/valor a su código, por ejemplo, hiperparámetros como num_trees o max_depth en su bosque aleatorio

Métrica

Valores numéricos (pueden actualizarse con el tiempo), por ejemplo, RMSE o valores de precisión

Artefactos

Archivos, datos y modelos, por ejemplo, imágenes matplotlib o archivos Parquet

Metadatos

Información sobre la ejecución, como el código fuente que ejecutó la ejecución o la versión del código (p. ej., la cadena hash de confirmación de Git para la versión del código)

Modelos

El(es) modelo(s) que entrenaste

De forma predeterminada, el servidor de seguimiento registra todo en el sistema de archivos, pero puede **especificar una base de datos** para consultas más rápidas, como los parámetros y las métricas. Agreguemos el seguimiento de MLflow a nuestro código de bosque aleatorio del [Capítulo 10](#):

```
# En Python
desde pyspark.ml import Pipeline from
pyspark.ml.feature import StringIndexer, VectorAssembler from
pyspark.ml.regression import RandomForestRegressor from pyspark.ml.e
Evaluation import RegressionEvaluator

filePath = """/databricks-datasets/learning-spark-v2/sf-airbnb/ sf-airbnb-clean.parquet"""
airbnbDF = spark.read.parquet(filePath) (trainDF, testDF) = airbnbDF.randomSplit( [.8,
.2], semilla=42)

categoricalCols = [field for (field, dataType) in trainDF.dtypes if dataType == "string"]
indexOutputCols = [x + "Index" for x in categoricalCols]
stringIndexer = StringIndexer(inputCols=categoricalCols, outputCols=indexOutputCols,
handleInvalid=" saltear")

numericCols = [campo para (campo, tipo de datos ) en trainDF.dtypes if ((tipo de
datos == "doble") & (campo != "precio"))]
ensambladorInputs = indexOutputCols + numericCols
vecAssembler = VectorAssembler(inputCols=ensambladorInputs,
outputCol="características")

rf = RandomForestRegressor(labelCol="price", maxBins=40, maxDepth=5,
número de árboles = 100, semilla = 42)

tubería = tubería (etapas = [stringIndexer, vecAssembler, rf])
```

Para comenzar a iniciar sesión con MLflow, deberá iniciar una ejecución con mlflow.start_run(). En lugar de llamar explícitamente a mlflow.end_run(), los ejemplos de este capítulo usarán una cláusula with para finalizar automáticamente la ejecución al final del bloque with :

```
# En Python
importar mlflow
importar mlflow.spark
importar pandas como pd

con mlflow.start_run(run_name="random-forest") como ejecución:
    # Parámetros de registro: número_árboles y
    profundidad_máxima mlflow.log_param("número_árboles",
    rf.getNumTrees()) mlflow.log_param("profundidad_máxima", rf.getMaxDepth())

    # Modelo de
    registro pipelineModel = pipeline.fit(trainDF)
    mlflow.spark.log_model(pipelineModel, "modelo")

    # Métricas de registro: RMSE y R2
    predDF = pipelineModel.transform(testDF)
    regressionEvaluator = RegressionEvaluator(predictionCol="prediction",
                                              labelCol="precio")
    rmse = regressionEvaluator.setMetricName("rmse").evaluate(predDF)
```

```
r2 = regressionEvaluator.setMetricName("r2").evaluate(predDF)
mlflow.log_metrics({"rmse": rmse, "r2": r2})

# Artefacto de registro: puntuaciones de importancia
de características rfModel = pipelineModel.stages[-1]
pandasDF = (pd.DataFrame(list(zip(vecAssembler.getInputCols(),
                                 rfModel.featureImportances))),
            columns=["característica",
                      "importancia"] ) .sort_values(por="importancia", ascendente=False)

# Primero escriba en el sistema de archivos local, luego dígale a MLflow dónde encontrar
ese archivo pandasDF.to_csv("feature-importance.csv", index=False) mlflow.log_artifact("feature-
importance.csv")
```

Examinemos la interfaz de usuario de MLflow, a la que puede acceder ejecutando `mlflow ui` en su terminal y navegando a `http://localhost:5000/`. La figura 11-3 muestra una captura de pantalla de la interfaz de usuario.

The screenshot shows the MLflow UI for an experiment with ID 0. It displays a table of runs, each with a checkbox, Date, Run Name, User, Source, Version, max_depth, num_trees, and r2 metrics. One run is selected, showing 2020-04-21, random-forest, brookewenig, spark_mlflow.p, 45978e, 5, 100, and 0.22794251... The interface includes a search bar, filter dropdowns for State (Active), and buttons for Compare, Delete, and Download CSV.

	Date	Run Name	User	Source	Version	max_depth	num_trees	r2
<input type="checkbox"/>	2020-04-21	random-forest	brookewenig	spark_mlflow.p	45978e	5	100	0.22794251...

Figura 11-3. La interfaz de usuario de MLflow

La interfaz de usuario almacena todas las ejecuciones de un experimento determinado. Puede buscar en todas las ejecuciones, filtrar aquellas que cumplan con criterios particulares, comparar ejecuciones una al lado de la otra, etc. Si lo desea, también puede exportar el contenido como un archivo CSV para analizarlo localmente. Haga clic en la ejecución en la interfaz de usuario denominada "bosque aleatorio". Debería ver una pantalla como la Figura 11-4.

Default > random-forest

Date : 2020-04-21 15:28:51	Source : spark_mlflow.py	Git Commit : 45978e03fbfd093d71947efa7f02b5a6434bc559
User: brookewenig	Duration: 12.2s	Status: FINISHED

▼ Notes

None

▼ Parameters

Name	Value
max_depth	5
num_trees	100

▼ Metrics

Name	Value
r2	0.228
rmse	211.5

► Tags

▼ Artifacts

► model feature-importance.csv	Full Path: file:///Users/brookewenig/PycharmProjects/LearningSparkv2/mlruns/0... Size: 1.28KB	
<pre>feature,importance bedrooms,0.15843143150583575 accommodates,0.13279647009992607 neighbourhood_cleansedIndex,0.10999025254162696 beds,0.09951358167394081 cancellation_policyIndex,0.0851694066862927</pre>		

Figura 11-4. Ejecución aleatoria del bosque

Notará que realiza un seguimiento del código fuente utilizado para esta ejecución de MLflow, además de almacenar todos los parámetros, métricas, etc. correspondientes. Puede agregar notas sobre esta ejecución en texto libre, así como etiquetas. No puede modificar los parámetros ni las métricas una vez finalizada la ejecución.

También puede consultar el servidor de seguimiento utilizando MlflowClient o REST API:

```
# En Python
desde mlflow.tracking import MlflowClient

cliente = MlflowClient() se
ejecuta = client.search_runs(run.info.experiment_id,
                             order_by=["attributes.start_time desc"],
                             max_results=1)

run_id = carreras[0].info.run_id
carreras[0].data.metrics
```

Esto produce la siguiente salida:

```
{'r2': 0.22794251914574226, 'rmse': 211.5096898777315}
```

Hemos alojado este código como un [proyecto de MLflow](#) en el repositorio de [GitHub](#) para este libro, para que pueda experimentar ejecutándolo con diferentes valores de hiperparámetros para max_depth y num_trees. El archivo YAML dentro del proyecto MLflow especifica las dependencias de la biblioteca para que este código pueda ejecutarse en otros entornos:

En Python

```
mlflow.run("https://github.com/databricks/LearningSparkV2/#mlflow-project-example",
    parameters={"max_depth": 5, "num_trees": 100})
```

O en la línea de comando

```
ejecutar mlflow https://github.com/databricks/LearningSparkV2/#mlflow-project-example -P
max_depth=5 -P num_trees=100
```

Ahora que ha rastreado y reproducido sus experimentos, analicemos las diversas opciones de implementación disponibles para sus modelos MLlib.

Opciones de implementación de modelos con MLlib

Implementar modelos de aprendizaje automático significa algo diferente para cada organización y caso de uso. Las restricciones comerciales impondrán diferentes requisitos de latencia, rendimiento, costo, etc., que dictarán qué modo de implementación del modelo es adecuado para la tarea en cuestión, ya sea por lotes, transmisión, en tiempo real o móvil/integrado. La implementación de modelos en sistemas móviles/integrados está fuera del alcance de este libro, por lo que nos centraremos principalmente en las otras opciones. La [tabla 11-1](#) muestra el [rendimiento](#) y [latencia](#) compensaciones para estas tres opciones de implementación para generar predicciones. Nos preocupamos tanto por la cantidad de solicitudes simultáneas como por el tamaño de esas solicitudes, y las soluciones resultantes se verán bastante diferentes.

Tabla 11-1. Comparación por lotes, transmisión y en tiempo real

Latencia de rendimiento		Ejemplo de aplicación	
Lote	Alto	Alta (horas a días)	Predicción de abandono de clientes
Medio de transmisión			Precios dinámicos
Baja en tiempo real	Bajo (milisegundos)		Ofertas de anuncios en línea

El procesamiento por lotes genera predicciones en un horario regular y escribe los resultados en un almacenamiento persistente para que se sirvan en otro lugar. Por lo general, es la opción de implementación más barata y fácil, ya que solo necesita pagar por el cómputo durante su ejecución programada. El procesamiento por lotes es mucho más eficiente por punto de datos porque acumula menos gastos generales cuando se amortizan en todas las predicciones realizadas. Este es particularmente el caso con Spark, debido a la sobrecarga de comunicación entre el controlador y los ejecutores; no querrá hacer predicciones punto de datos a la vez.

¡tiempo! Sin embargo, su principal inconveniente es la latencia, ya que normalmente se programa con un período de horas o días para generar el siguiente lote de predicciones.

La transmisión ofrece un buen compromiso entre el rendimiento y la latencia. Continuamente hará predicciones sobre micro lotes de datos y obtendrá sus predicciones en segundos o minutos. Si usa transmisión estructurada, casi todo su código se verá idéntico al caso de uso por lotes, lo que facilita alternar entre estas dos opciones. Con la transmisión, tendrá que pagar por las máquinas virtuales o los recursos informáticos que utiliza para mantenerse en funcionamiento de forma continua, y asegurarse de haber configurado la transmisión correctamente para que sea tolerante a fallas y proporcione almacenamiento en búfer si hay picos en los datos entrantes.

La implementación en tiempo real prioriza la latencia sobre el rendimiento y genera predicciones en unos pocos milisegundos. Su infraestructura deberá admitir el equilibrio de carga y poder escalar a muchas solicitudes simultáneas si hay un gran aumento en la demanda (por ejemplo, para minoristas en línea durante las vacaciones). A veces, cuando la gente dice "implementación en tiempo real", se refiere a extraer predicciones precalculadas en tiempo real, pero aquí nos referimos a generar predicciones de modelos en tiempo real. La implementación en tiempo real es la única opción para la que Spark no puede cumplir con los requisitos de latencia, por lo que para usarla deberá exportar su modelo fuera de Spark. Por ejemplo, si pretende usar un extremo REST para la inferencia de modelos en tiempo real (por ejemplo, calcular predicciones en menos de 50 ms), MLlib no cumple con los requisitos de latencia necesarios para esta aplicación, como se muestra en la [figura 11-5](#). Necesitará obtener su modelo y preparación de características fuera de Spark, lo que puede llevar mucho tiempo y ser difícil.

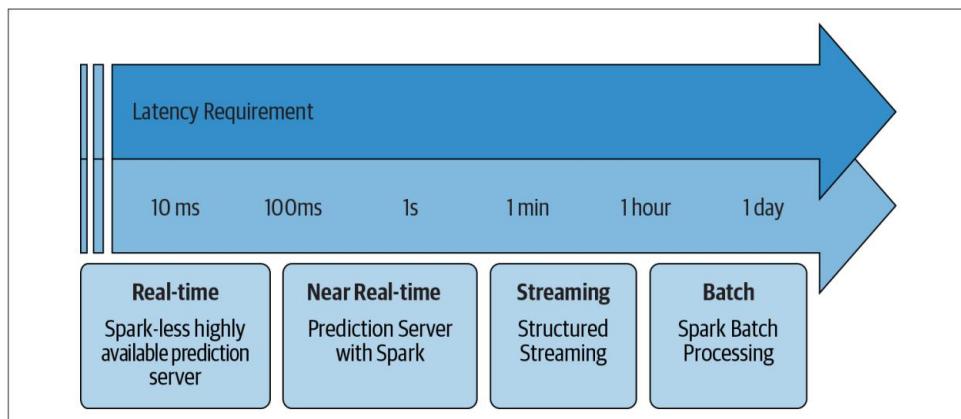


Figura 11-5. Opciones de implementación para MLlib

Antes de comenzar el proceso de modelado, debe definir los requisitos de implementación de su modelo. MLlib y Spark son solo algunas de las herramientas de su caja de herramientas, y debe comprender cuándo y dónde deben aplicarse. El resto de esta sección

analiza las opciones de implementación para MLlib con más profundidad y luego consideraremos las opciones de implementación con Spark para modelos que no son MLlib.

Lote

Las implementaciones por lotes representan la mayoría de los casos de uso para implementar modelos de aprendizaje automático y esta es posiblemente la opción más fácil de implementar. Ejecutará un trabajo regular para generar predicciones y guardará los resultados en una tabla, base de datos, lago de datos, etc. para el consumo posterior. De hecho, ya vio cómo generar predicciones por lotes en el [Capítulo 10](#) con MLlib. `model.transform()` de MLlib aplicará el modelo en paralelo a todas las particiones de su DataFrame:

```
# En Python
# Cargue el modelo guardado con
MLflow import mlflow.spark
pipelineModel = mlflow.spark.load_model(f"runs:{run_id}/model")

# Generar predicciones
inputDF = spark.read.parquet("/databricks-datasets/learning-spark-v2/
sf-airbnb/sf-airbnb-clean.parquet")

predDF = pipelineModel.transform(inputDF)
```

Algunas cosas a tener en cuenta con las implementaciones por lotes son:

¿Con qué frecuencia generará predicciones?

Hay un compromiso entre latencia y rendimiento. Obtendrá un mayor rendimiento al agrupar muchas predicciones, pero luego el tiempo que lleva recibir cualquier predicción individual será mucho más largo, lo que retrasará su capacidad para actuar sobre estas predicciones.

¿Con qué frecuencia volverá a entrenar al modelo?

A diferencia de bibliotecas como sklearn o TensorFlow, MLlib no admite actualizaciones en línea ni inicios en caliente. Si desea volver a entrenar su modelo para incorporar los datos más recientes, tendrá que volver a entrenar todo el modelo desde cero, en lugar de aprovechar los parámetros existentes. En cuanto a la frecuencia del reentrenamiento, algunas personas establecerán un trabajo regular para reentrenar el modelo (por ejemplo, una vez al mes), mientras que otras [monitorearán activamente la desviación del modelo](#) para identificar cuándo necesitan volver a capacitarse.

¿Cómo versionarás el modelo?

Puede usar el [Registro de modelos de MLflow](#) para realizar un seguimiento de los modelos que está utilizando y controlar cómo se transfieren a/desde la puesta en escena, la producción y el archivo. Puede ver una captura de pantalla del Registro del modelo en la [Figura 11-6](#). También puede usar el registro modelo con las otras opciones de implementación.

Name	Latest Version	Staging	Production	Last Modified
Model A	Version 1	Version 1	-	2019-10-16 22:51:19
Model B	Version 1	-	-	2019-10-16 22:51:52

Figura 11-6. Registro del modelo de flujo de ML

Además de usar la interfaz de usuario de MLflow para administrar sus modelos, también puede administrarlos mediante programación. Por ejemplo, una vez que haya registrado su modelo de producción, tiene un URI coherente que puede usar para recuperar la última versión:

```
# Recuperar el modelo de producción
más reciente model_production_uri = f"models:{model_name}/"
"production" model_production = mlflow.spark.load_model(model_production_uri)
```

Streaming En

lugar de esperar un trabajo por hora o por la noche para procesar sus datos y generar predicciones, el Streaming estructurado puede realizar inferencias de forma continua sobre los datos entrantes. Si bien este enfoque es más costoso que una solución por lotes, ya que tiene que pagar continuamente por el tiempo de cómputo (y obtener un menor rendimiento), obtiene el beneficio adicional de generar predicciones con mayor frecuencia para que pueda actuar sobre ellas antes. Las soluciones de transmisión en general son más complicadas de mantener y monitorear que las soluciones por lotes, pero ofrecen una latencia más baja.

Con Spark es muy fácil convertir tus predicciones por lotes en predicciones de transmisión, y prácticamente todo el código es el mismo. La única diferencia es que cuando lee los datos, necesita usar spark.readStream() en lugar de spark.read() y cambiar la fuente de los datos. En el siguiente ejemplo, vamos a simular la lectura en transmisión de datos mediante la transmisión en un directorio de archivos de Parquet. Notará que estamos especificando un esquema a pesar de que estamos trabajando con archivos de Parquet. Esto se debe a que necesitamos definir el esquema a priori cuando trabajamos con datos de transmisión. En este ejemplo, usaremos el modelo de bosque aleatorio entrenado en nuestro conjunto de datos de Airbnb del capítulo anterior para realizar estas predicciones de transmisión. Cargaremos el modelo guardado usando MLflow. Hemos dividido el archivo de origen en cien archivos Parquet pequeños para que pueda ver cómo cambia la salida en cada intervalo de activación:

```
# En Python
# Carga el modelo guardado con
MLflow pipelineModel = mlflow.spark.load_model(f"runs:{run_id}/model")
```

```

# Configurar datos de transmisión
simulados repartitionedPath = "/databricks-datasets/learning-spark-v2/sf-airbnb/sf-
    airbnb-clean-100p.parquet"
esquema = chispa.leer.parquet(rutareparticionada).esquema

streamingData =
    (spark .readStream .schema(schema) # Puede establecer
        el esquema de esta manera .option("maxFilesPerTrigger",
        1) .parquet(repartitionedPath))

# Generar predicciones
streamPred = pipelineModel.transform(streamingData)

```

Después de generar estas predicciones, puede escribirlas en cualquier ubicación de destino para recuperarlas más tarde (consulte el [Capítulo 8](#) para obtener sugerencias sobre transmisión estructurada). Como puede ver, el código prácticamente no cambia entre los escenarios por lotes y de transmisión, lo que hace que MLlib sea una excelente solución para ambos. Sin embargo, según las exigencias de latencia de su tarea, es posible que MLlib no sea la mejor opción. Con Spark, hay una sobrecarga significativa involucrada en la generación del plan de consulta y la comunicación de la tarea y los resultados entre el conductor y el trabajador. Por lo tanto, si necesita predicciones de latencia realmente baja, deberá exportar su modelo fuera de Spark.

Casi en tiempo real

Si su caso de uso requiere predicciones del orden de cientos de milisegundos a segundos, puede crear un servidor de predicción que use MLlib para generar las predicciones.

Si bien este no es un caso de uso ideal para Spark porque está procesando cantidades muy pequeñas de datos, obtendrá una latencia más baja que con las soluciones de transmisión o por lotes.

Patrones de exportación de modelos para la inferencia en

tiempo real Hay algunos dominios en los que se requiere la inferencia en tiempo real, incluida la detección de fraudes, la recomendación de anuncios y similares. Si bien hacer predicciones con una pequeña cantidad de registros puede lograr la baja latencia requerida para la inferencia en tiempo real, deberá lidiar con el equilibrio de carga (manejar muchas solicitudes simultáneas), así como con la geolocalización en tareas críticas de latencia. Existen soluciones administradas populares, como [AWS SageMaker](#) y [Azure ML](#), que proporcionan soluciones de servicio de modelos de baja latencia. En esta sección, le mostraremos cómo exportar sus modelos MLlib para que puedan implementarse en esos servicios.

Una forma de exportar su modelo fuera de Spark es volver a implementar el modelo de forma nativa en Python, C, etc. Si bien puede parecer simple extraer los coeficientes del modelo, exportar todos los pasos de preprocesamiento e ingeniería de funciones junto con ellos (OneHotEncoder, VectorAssembler, etc.) rápidamente se vuelve problemático y es muy propenso a errores.

Hay algunas bibliotecas de código abierto, como **MLeap** y **ONNX**, eso puede ayudarlo a exportar automáticamente un subconjunto compatible de los modelos MLlib para eliminar su dependencia de Spark. Sin embargo, en el momento de escribir este artículo, la empresa que desarrolló MLeap ya no lo admite. MLeap tampoco es compatible todavía con Scala 2.12/Spark 3.0.

ONNX (Open Neural Network Exchange), por otro lado, se ha convertido en el estándar abierto de facto para la interoperabilidad del aprendizaje automático. Algunos de ustedes pueden recordar otros formatos de interoperabilidad de ML, como PMML (Lenguaje de marcado de modelo predictivo), pero esos nunca ganaron la misma tracción que ONNX tiene ahora. ONNX es muy popular en la comunidad de aprendizaje profundo como una herramienta que permite a los desarrolladores cambiar fácilmente entre bibliotecas e idiomas, y en el momento de escribir este artículo tiene soporte experimental para MLlib.

En lugar de exportar modelos de MLlib, existen otras bibliotecas de terceros que se integran con Spark y que son convenientes para implementar en escenarios en tiempo real, como **XGBoost**. y **agua** con gas de H2O.ai (cuyo nombre se deriva de una combinación de H2O y Spark).

XGBoost es uno de **los algoritmos más exitosos** en **competiciones de Kaggle** para problemas de datos estructurados, y es una biblioteca muy popular entre los científicos de datos. Aunque XGBoost no es técnicamente parte de MLlib, la **biblioteca XGBoost4J-Spark** le permite integrar XGBoost distribuido en sus canalizaciones MLlib. Una ventaja de XGBoost es la facilidad de implementación: después de entrenar su canalización de MLlib, puede extraer el modelo de XGBoost y guardarlo como un modelo que no sea de Spark para servir en Python, como se demuestra aquí:

```
// En Scala
val xgboostModel =
  xgboostPipelineModel.stages.last.asInstanceOf[XGBoostRegressionModel]
xgboostModel.nativeBooster.saveModel(nativeModelPath)

# En Python
importa xgboost como
xgb bst = xgb.Booster({'nthread': 4})
bst.load_model("xgboost_native_model")
```



En el momento de escribir este artículo, la API XGBoost distribuida solo está disponible en Java/Scala. Se incluye un ejemplo completo en el repositorio de [Git- Hub del libro](#).

Ahora que ha aprendido acerca de las diferentes formas de exportar modelos de MLlib para usar en entornos de servicio en tiempo real, analicemos cómo podemos aprovechar Spark para modelos que no son de MLlib.

Aprovechamiento de Spark para modelos que no son MLlib

Como se mencionó anteriormente, MLlib no siempre es la mejor solución para sus necesidades de aprendizaje automático. Es posible que no cumpla con los requisitos de inferencia de latencia súper baja o que tenga soporte integrado para el algoritmo que le gustaría usar. Para estos casos, aún puede aprovechar Spark, pero no MLlib. En esta sección, discutiremos cómo puede usar Spark para realizar inferencias distribuidas de modelos de un solo nodo usando Pandas UDF, realizar ajustes de hiperparámetros y escalar ingeniería de características.

Pandas UDF

Si bien MLlib es fantástico para el entrenamiento distribuido de modelos, no está limitado a usar MLlib solo para hacer predicciones por lotes o de transmisión con Spark; puede crear funciones personalizadas para aplicar sus modelos previamente entrenados a escala, conocidas como funciones definidas por el usuario (UDF, cubierto en [el Capítulo 5](#)). Un caso de uso común es construir un modelo de scikit-learn o TensorFlow en una sola máquina, quizás en un subconjunto de sus datos, pero realizar una inferencia distribuida en todo el conjunto de datos usando Spark.

Si define su propia UDF para aplicar un modelo a cada registro de su DataFrame en Python, opte por [pandas UDF](#) para una serialización y deserialización optimizadas, como se discutió en el [Capítulo 5](#). Sin embargo, si su modelo es muy grande, entonces hay una gran sobrecarga para que Pandas UDF cargue repetidamente el mismo modelo para cada lote en el mismo proceso de trabajo de Python. En Spark 3.0, las UDF de Pandas pueden aceptar un iterador de `pandas.Series` o `pandas.DataFrame` para que pueda cargar el modelo solo una vez en lugar de cargarlo para cada serie en el iterador. Para obtener más detalles sobre las novedades de Apache Spark 3.0 con Pandas UDF, consulte el [Capítulo 12](#).



Si los trabajadores almacenaron en caché los pesos del modelo después de cargarlo por primera vez, las llamadas posteriores de la misma UDF con la misma carga del modelo serán significativamente más rápidas.

En el siguiente ejemplo, usaremos `mapInPandas()`, introducido en Spark 3.0, para aplicar un modelo scikit-learn a nuestro conjunto de datos de Airbnb. `mapInPandas()` toma un iterador de `pandas.DataFrame` como entrada y genera otro iterador de `pandas.DataFrame`. Es flexible y fácil de usar si su modelo requiere todas sus columnas como entrada, pero requiere serialización/deserialización de todo el DataFrame (a medida que se pasa a su entrada). Puede controlar el tamaño de cada `pandas.DataFrame` con la configuración `spark.sql.execution.arrow.maxRecordsPerBatch`. Una copia completa del código para generar el modelo está disponible en el repositorio de [GitHub de este libro](#), pero aquí solo nos centraremos en cargar el modelo scikit-learn guardado de MLflow y aplicarlo a nuestro Spark DataFrame:

```
# En Python
import mlflow.sklearn
import pandas as pd

def predecir (iterador):
    model_path = f"runs/{run_id}/random-forest-model" model =
    mlflow.sklearn.load_model(model_path) # Carga el modelo para las
    características en el iterador:
    rendimiento pd.DataFrame (modelo. predecir (características))

df.mapInPandas(predict, "predicción doble").show(3)

+-----+
|     predicción|
+-----+
| 90.4355866254844|
255.3459534312323|
499.625544914651|
+-----+
```

Además de aplicar modelos a escala con una UDF de Pandas, también puede usarlos para paralelizar el proceso de creación de muchos modelos. Por ejemplo, es posible que desee crear un modelo para cada tipo de dispositivo IoT para predecir el tiempo de falla. Puede usar `pyspark.sql.GroupedData.applyInPandas()` (introducido en Spark 3.0) para esta tarea. La función toma un `pandas.DataFrame` y devuelve otro `pandas.DataFrame`. El repositorio de GitHub del libro contiene un ejemplo completo del código para construir un modelo por tipo de dispositivo IoT y rastrear los modelos individuales con MLflow; solo se incluye un fragmento aquí por brevedad:

```
# En Python
df.groupBy("device_id").applyInPandas(build_model, schema=trainReturnSchema)
```

El `groupBy()` causará una mezcla completa de su conjunto de datos, y debe asegurarse de que su modelo y los datos de cada grupo puedan caber en una sola máquina. Algunos de ustedes pueden estar familiarizados con `pyspark.sql.GroupedData.apply()` (por ejemplo, `df.groupBy("device_id").apply(build_model)`), pero esa API quedará obsoleta en versiones futuras de Spark a favor de `pyspark.sql.GroupedData.applyInPandas()`.

Ahora que ha visto cómo aplicar UDF para realizar inferencia distribuida y paralelizar la creación de modelos, veamos cómo usar Spark para el ajuste de hiperparámetros distribuidos.

Spark para el ajuste de hiperparámetros distribuidos

Incluso si no tiene la intención de hacer una inferencia distribuida o no necesita las capacidades de entrenamiento distribuidas de MLlib, aún puede aprovechar Spark para el ajuste de hiperparámetros distribuidos. Esta sección cubrirá dos bibliotecas de código abierto en particular: Joblib e Hyperopt.

Joblib

Según su documentación, **Joblib** es "un conjunto de herramientas para proporcionar canalización ligera en Python". Tiene un backend Spark para distribuir tareas en un clúster Spark. Joblib se puede utilizar para el ajuste de hiperparámetros, ya que transmite automáticamente una copia de sus datos a todos sus trabajadores, que luego crean sus propios modelos con diferentes hiperparámetros en sus copias de los datos. Esto le permite entrenar y evaluar múltiples modelos en paralelo. Todavía tiene la limitación fundamental de que un solo modelo y todos los datos tienen que caber en una sola máquina, pero puede paralelizar trivialmente la búsqueda de hiperparámetros, como se muestra en la [Figura 11-7](#).

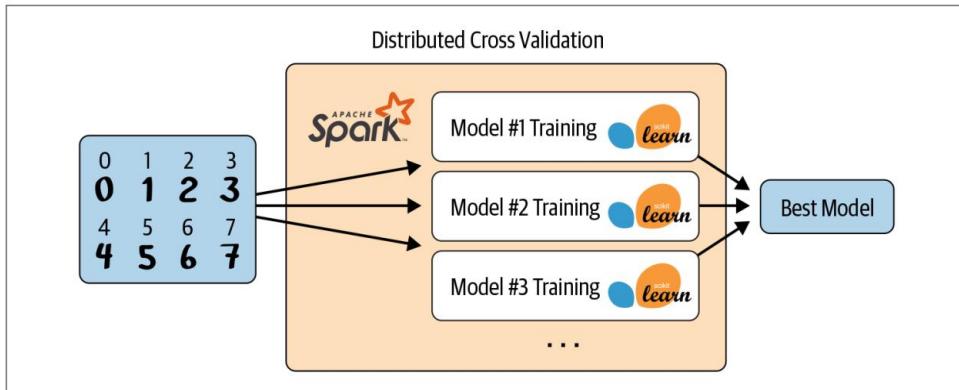


Figura 11-7. Búsqueda distribuida de hiperparámetros

Para usar Joblib, instálelo a través de pip install joblibspark. Asegúrese de estar utilizando la versión 0.21 o posterior de scikit learn y la versión 2.4.4 o posterior de pyspark . Aquí se muestra un ejemplo de cómo realizar una validación cruzada distribuida, y el mismo enfoque también funcionará para el ajuste de hiperparámetros distribuidos:

```
# En Python
desde sklearn.utils import parallel_backend from
sklearn.ensemble import RandomForestRegressor from
sklearn.model_selection import train_test_split from
sklearn.model_selection import GridSearchCV import pandas
as pd from joblibspark import register_spark

register_spark() # Registrar backend de Spark

df = pd.read_csv("/dbfs/databricks-datasets/learning-spark-v2/sf-airbnb/
    sf-airbnb-numeric.csv")
X_train, X_test, y_train, y_test = train_test_split(df.drop(["price"], axis=1),
    df[["precio"]].valores.ravel(), estado_aleatorio=42)

rf = RandomForestRegressor(random_state=42)
param_grid = {"max_depth": [2, 5, 10], "n_estimators": [20, 50, 100]}
```

```

gscv = GridSearchCV (rf, param_grid, cv = 3)

con paralelo_backend("chispa", n_trabajos=3):
    gscv.fit(X_tren, y_tren)

    imprimir (gscv.cv_resultados_)

```

Consulte la documentación de **GridSearchCV** de scikit-learn para obtener una explicación de los parámetros devueltos por el validador cruzado.

hiperóptico

hiperóptico es una biblioteca de Python para "optimización en serie y en paralelo en espacios de búsqueda incómodos, que pueden incluir dimensiones de valor real, discretas y condicionales".

Puede instalarlo a través de pip install hyperopt. Hay dos formas principales de **escalar Hyperopt con Apache Spark**:

- Usar Hyperopt de una sola máquina con un algoritmo de entrenamiento distribuido (p. ej., MLlib)
- Uso de Hyperopt distribuido con algoritmos de entrenamiento de una sola máquina con el Clase SparkTrials

Para el primer caso, no hay nada especial que deba configurar para usar MLlib con Hyperopt frente a cualquier otra biblioteca. Entonces, echemos un vistazo al último caso: Hyperopt distribuido con modelos de un solo nodo. Desafortunadamente, no puede combinar la evaluación de hiperparámetros distribuidos con modelos de entrenamiento distribuidos en el momento de escribir este artículo. El ejemplo de código completo para parallelizar la búsqueda de hiperparámetros para un **Keras** el modelo se puede encontrar en el repositorio de **GitHub del libro**; Aquí solo se incluye un fragmento para ilustrar los componentes clave de Hyperopt:

```

# En Python
import hyperopt

best_hyperparameters = hyperopt.fmin( fn =
    función_entrenamiento , espacio =
    espacio_búsqueda, algo = hyperopt.tpe.suggest,
    max_evals = 64, pruebas =
    hyperopt.SparkTrials(paralelismo=4))

```

fmin() genera nuevas configuraciones de hiperparámetros para usar en su función de entrenamiento y las pasa a SparkTrials. SparkTrials ejecuta lotes de estas tareas de capacitación en paralelo como un trabajo de Spark de una sola tarea en cada ejecutor de Spark. Cuando finaliza la tarea Spark, devuelve los resultados y la pérdida correspondiente al controlador. Hyperopt utiliza estos nuevos resultados para calcular mejores configuraciones de hiperparámetros para futuras tareas. Esto permite una escalabilidad horizontal masiva del ajuste de hiperparámetros. MLflow también

se integra con Hyperopt, por lo que puede realizar un seguimiento de los resultados de todos los modelos que ha entrenado como parte de su ajuste de hiperparámetros.

Un parámetro importante para SparkTrials es el paralelismo. Esto determina el número máximo de ensayos para evaluar simultáneamente. Si el paralelismo = 1, entonces está entrenando cada modelo secuencialmente, pero puede obtener mejores modelos haciendo un uso completo de los algoritmos adaptativos. Si establece paralelismo = max_evals (el número total de modelos para entrenar), entonces solo está haciendo una búsqueda aleatoria.

Cualquier número entre 1 y max_evals le permite tener un compromiso entre escalabilidad y adaptabilidad. De forma predeterminada, el paralelismo se establece en la cantidad de ejecutores de Spark. También puede especificar un tiempo de espera para limitar el número máximo de segundos que fmin() puede tomar.

Incluso si MLlib no es adecuado para su problema, es de esperar que pueda ver el valor de usar Spark en cualquiera de sus tareas de aprendizaje automático.

koalas

pandas es una biblioteca de manipulación y análisis de datos muy popular en Python, pero se limita a ejecutarse en una sola máquina. **koalas** es una biblioteca de código abierto que implementa la API de Pandas DataFrame sobre Apache Spark, lo que facilita la transición de Pandas a Spark. Puede instalarlo con pip install koalas y luego simplemente reemplazar cualquier lógica pd (Pandas) en su código con ks (Koalas). De esta forma, puede ampliar sus análisis con Pandas sin necesidad de reescribir por completo su base de código en PySpark.

Aquí hay un ejemplo de cómo cambiar su código Pandas a Koalas (deberá tener PySpark ya instalado):

```
# En pandas
import pandas como pd
pdf = pd.read_csv(csv_path, header=0, sep=";", quotechar="")
pdf["duration_new"] = pdf["duration"] + 100

# En koalas
import databricks.koalas como ks
kdf = ks.read_csv(file_path, header=0, sep=";", quotechar="")
kdf["duration_new"] = kdf["duration"] + 100
```

Si bien Koalas tiene como objetivo implementar todas las funciones de Pandas eventualmente, no todas están implementadas todavía. Si hay una funcionalidad que necesita que Koalas no proporciona, siempre puede cambiar al uso de las API de Spark llamando a kdf.to_spark(). Alternativamente, puede llevar los datos al controlador llamando a kdf.to_pandas() y usar la API de Pandas (¡tenga cuidado de que el conjunto de datos no sea demasiado grande o puede bloquear el controlador!).

Resumen

En este capítulo, cubrimos una variedad de mejores prácticas para administrar e implementar canalizaciones de aprendizaje automático. Vio cómo MLflow puede ayudarlo a rastrear y reproducir experimentos y empaquetar su código y sus dependencias para implementar en otros lugares.

También discutimos las principales opciones de implementación (por lotes, transmisión y en tiempo real) y sus compensaciones asociadas. MLlib es una solución fantástica para el entrenamiento de modelos a gran escala y los casos de uso por lotes/transmisión, pero no superará a un modelo de un solo nodo para la inferencia en tiempo real en pequeños conjuntos de datos. Sus requisitos de implementación impactan directamente en los tipos de modelos y marcos que puede usar, y es fundamental analizar estos requisitos antes de comenzar el proceso de creación de modelos.

En el próximo capítulo, destacaremos algunas características nuevas clave en Spark 3.0 y cómo puede incorporarlas en sus cargas de trabajo de Spark.

CAPÍTULO 12

Epílogo: Apache Spark 3.0

En el momento en que escribimos este libro, Apache Spark 3.0 aún no se había lanzado oficialmente; todavía estaba en desarrollo y nos pusimos a trabajar con Spark 3.0.0-preview2. Todos los ejemplos de código de este libro se probaron con Spark 3.0.0-preview2 y no deberían funcionar de manera diferente con la versión oficial de Spark 3.0.

Siempre que fue posible en los capítulos, cuando fue relevante, mencionamos cuándo las características eran nuevas incorporaciones o comportamientos en Spark 3.0. En este capítulo, examinamos los cambios.

Las correcciones de errores y las mejoras de funciones son numerosas, por lo que, para abreviar, destacamos solo una selección de los cambios y funciones notables relacionados con los componentes de Spark.

Algunas de las nuevas características son, bajo el capó, avanzadas y más allá del alcance de este libro, pero las mencionamos aquí para que pueda explorarlas cuando la versión esté disponible en general.

Spark Core y Spark SQL

Consideremos primero qué hay de nuevo bajo las sábanas. Se han introducido una serie de cambios en Spark Core y el motor Spark SQL para ayudar a acelerar las consultas. Una forma de acelerar las consultas es leer menos datos mediante la eliminación dinámica de particiones. Otra es adaptar y optimizar los planes de consulta durante la ejecución.

Eliminación dinámica de particiones

La idea detrás [de la eliminación dinámica de particiones \(DPP\)](#) es omitir los datos que no necesita en los resultados de una consulta. El escenario típico en el que DPP es óptimo es cuando se unen dos tablas: una tabla de hechos (particionada en varias columnas) y una tabla de dimensiones (sin particiones), como se muestra en la [figura 12-1](#). Normalmente, el filtro está en el lado no particionado de la tabla (Fecha, en nuestro caso). Por ejemplo, considere esta consulta común sobre dos tablas, Ventas y Fecha:

-- En SQL
SELECCIONE * DE Ventas ÚNASE EN Ventas.fecha = Fecha.fecha

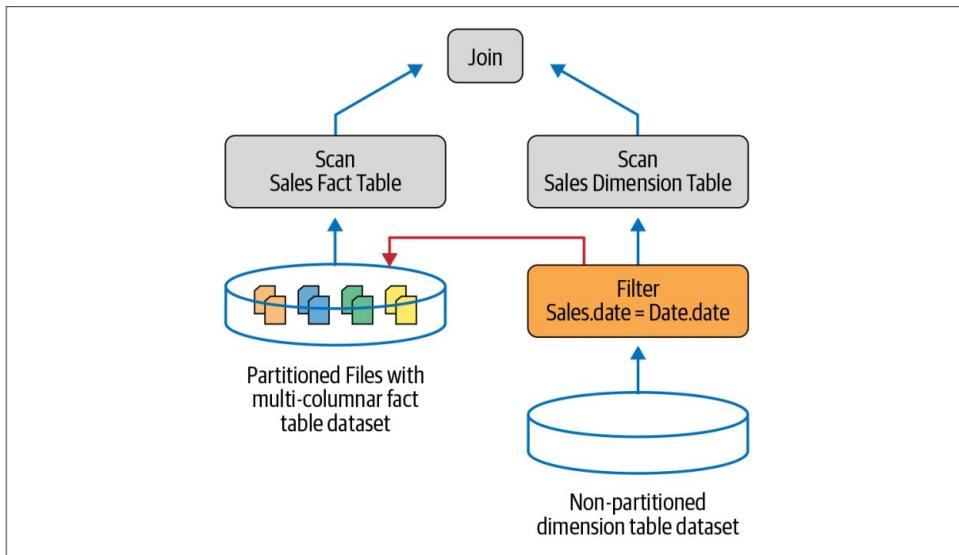


Figura 12-1. El filtro dinámico se inyecta desde la tabla de dimensiones a la tabla de hechos

La técnica de optimización clave en DPP es tomar el resultado del filtro de la tabla de dimensiones e inyectarlo en la tabla de hechos como parte de la operación de escaneo para limitar la lectura de datos, como se muestra en la [Figura 12-1](#).

Considere un caso donde la tabla de dimensiones es más pequeña que la tabla de hechos y realizamos una unión, como se muestra en la [Figura 12-2](#). En este caso, lo más probable es que Spark haga una unión de transmisión (discutido en el [Capítulo 7](#)). Durante esta unión, Spark llevará a cabo los siguientes pasos para minimizar la cantidad de datos escaneados de la tabla de hechos más grande:

1. En el lado de la dimensión de la combinación, Spark creará una tabla hash a partir de la tabla de dimensiones, también conocida como relación de construcción, como parte de esta consulta de filtro.
2. Spark conectará el resultado de esta consulta a la tabla hash y lo asignará a una variable de transmisión, que se distribuye a todos los ejecutores involucrados en esta operación de combinación.
3. En cada ejecutor, Spark probará la tabla hash transmitida para determinar qué filas correspondientes para leer de la tabla de hechos.
4. Finalmente, Spark inyectará este filtro dinámicamente en la operación de escaneo de archivos de la tabla de hechos y reutilizará los resultados de la variable de transmisión. De esta forma, como parte de la operación de escaneo de archivos en la tabla de hechos, solo se escanean las particiones que coinciden con el filtro y solo se leen los datos necesarios.

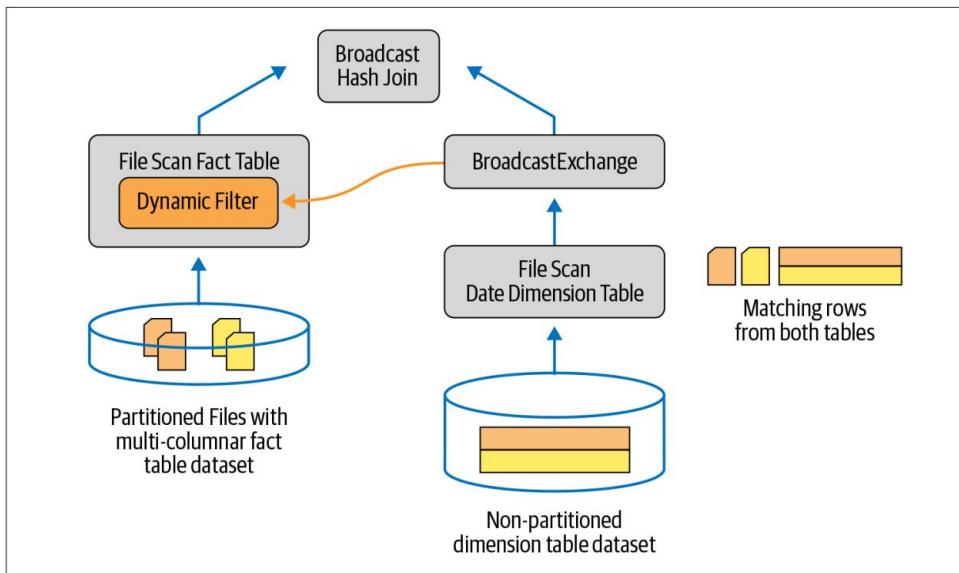


Figura 12-2. Spark inyecta un filtro de tabla de dimensiones en la tabla de hechos durante una unión de transmisión

Habilitado de forma predeterminada para que no tenga que configurarlo explícitamente, todo esto sucede de forma dinámica cuando realiza uniones entre dos tablas. Con la optimización de DPP, Spark 3.0 puede funcionar mucho mejor con consultas de esquema en estrella.

Ejecución adaptable de consultas Otra

forma en que Spark 3.0 optimiza el rendimiento de las consultas es adaptando su plan de ejecución física en tiempo de ejecución. **Ejecución adaptable de consultas (AQE)** vuelve a optimizar y ajusta los planes de consulta en función de las estadísticas de tiempo de ejecución recopiladas en el proceso de ejecución de la consulta. Intenta hacer lo siguiente en tiempo de ejecución:

- Reduzca el número de reductores en la etapa de reproducción aleatoria disminuyendo el número de particiones de reproducción aleatoria.
- Optimice el plan de ejecución física de la consulta, por ejemplo, convirtiendo SortMergeJoin en BroadcastHashJoin cuando corresponda.
- Manejar el sesgo de datos durante una combinación.

Todas estas medidas de adaptación tienen lugar durante la ejecución del plan en tiempo de ejecución, como se muestra en la [Figura 12-3](#). Para usar AQE en Spark 3.0, establezca la configuración `spark.sql.adaptive.enabled` en `true`.

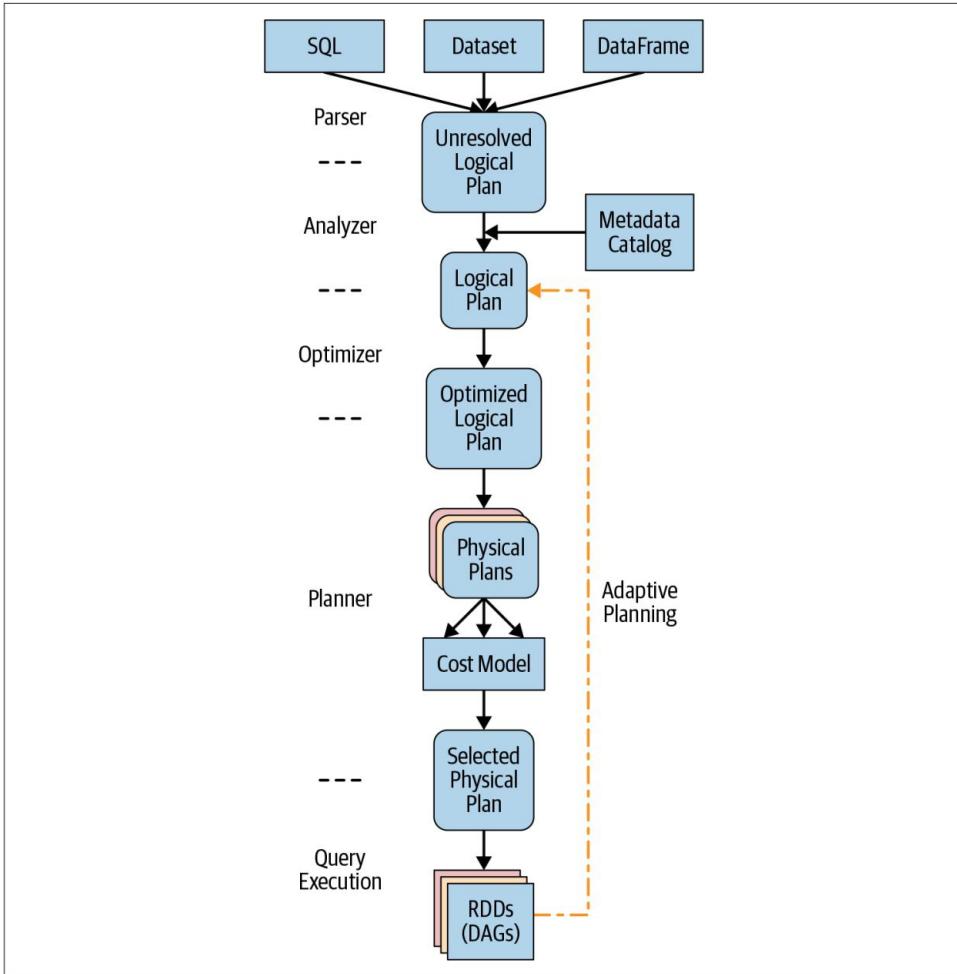


Figura 12-3. AQE reexamina y reoptimiza el plan de ejecución en tiempo de ejecución

El marco AQE

Las operaciones de Spark en una consulta se canalizan y ejecutan en procesos paralelos, pero un intercambio aleatorio o de difusión rompe esta canalización, porque la salida de una etapa se necesita como entrada para la siguiente etapa (consulte “[Paso 3: Comprensión de los conceptos de la aplicación Spark](#)” en la [página 25](#) del [Capítulo 2](#)). Estos puntos de ruptura se denominan puntos de materialización en una etapa de consulta y presentan una oportunidad para volver a optimizar y examinar la consulta, como se ilustra en [la figura 12-4](#).

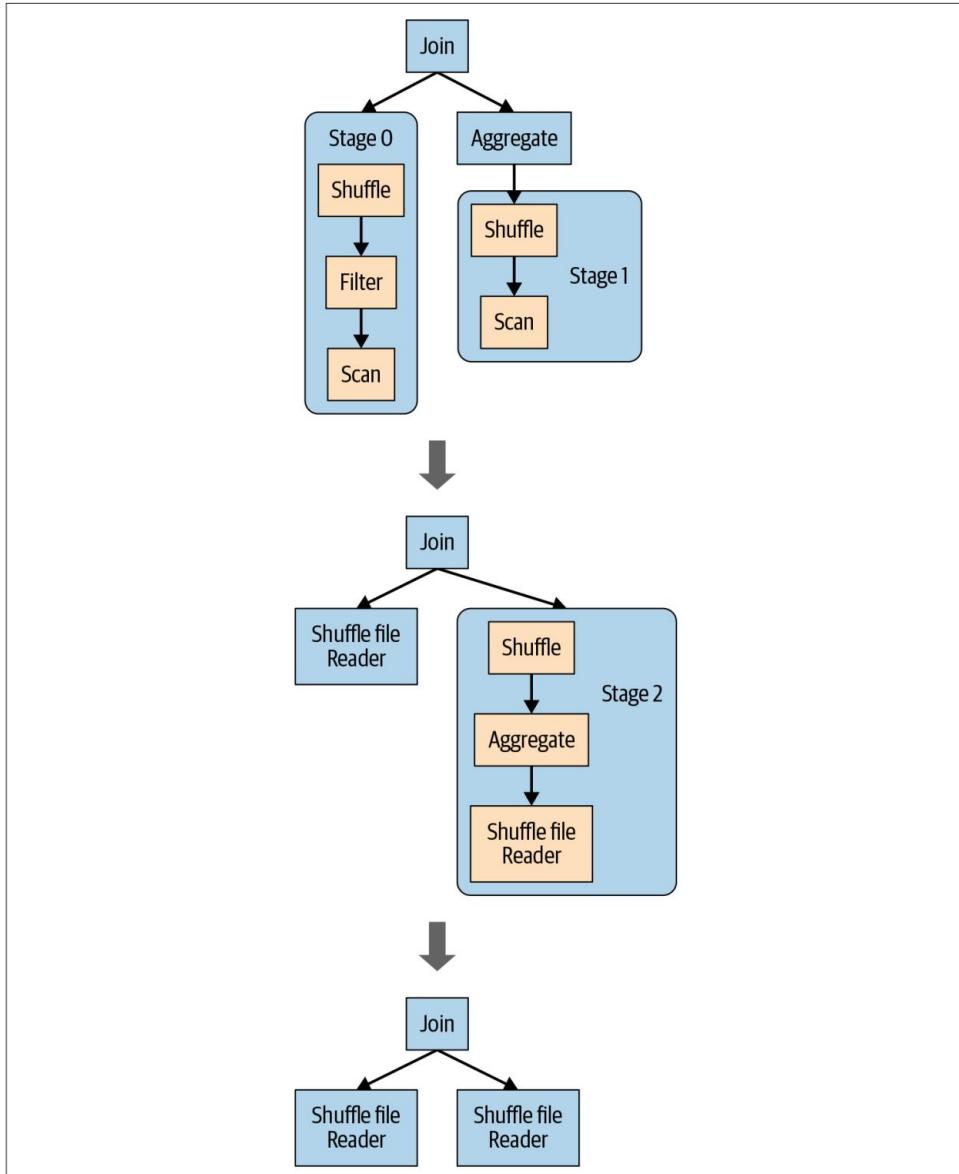


Figura 12-4. Un plan de consulta reoptimizado en el marco AQE

Estos son los pasos conceptuales sobre los que itera el marco AQE, como se muestra en esta figura:

1. Se ejecutan todos los nodos hoja, como las operaciones de escaneo, de cada etapa.
2. Una vez que el punto de materialización termina de ejecutarse, se marca como completo y todas las estadísticas relevantes obtenidas durante la ejecución se actualizan en su plan lógico.
3. En función de estas estadísticas, como el número de particiones leídas, los bytes de datos leídos, etc., el marco ejecuta el optimizador Catalyst nuevamente para comprender si puede:
 - a. Fusionar el número de particiones para reducir el número de reductores para leer mezclar datos.
 - b. Reemplace una combinación de combinación de clasificación, según el tamaño de las tablas leídas, con una transmisión entrar.
- C. Intente remediar una unión sesgada.
- d. Cree un nuevo plan lógico optimizado, seguido de un nuevo plan físico optimizado plan.

Este proceso se repite hasta que se ejecutan todas las etapas del plan de consulta.

En resumen, esta reoptimización se realiza dinámicamente, como se muestra en la [Figura 12-3](#), y el objetivo es fusionar dinámicamente las particiones aleatorias, disminuir la cantidad de reductores necesarios para leer los datos de salida aleatorios, cambiar las estrategias de unión si corresponde y remediar cualquier problema. uniones sesgadas.

Dos configuraciones de Spark SQL dictan cómo AQE reducirá la cantidad de reductores:

- `spark.sql.adaptive.coalescePartitions.enabled` (establecido en verdadero)
- `spark.sql.adaptive.skewJoin.enabled` (establecido en verdadero)

En el momento de escribir este artículo, el blog de la comunidad de Spark 3.0, la documentación y los ejemplos no se habían publicado públicamente, pero en el momento de la publicación deberían haberlo hecho. Estos recursos le permitirán obtener información más detallada si desea ver cómo funcionan estas funciones bajo el capó, incluida la forma en que puede inyectar sugerencias de combinación de SQL, que se analiza a continuación.

Sugerencias de combinación de SQL

Además de las sugerencias de BROADCAST existentes para combinaciones, Spark 3.0 agrega sugerencias de combinación para todas [las estrategias de combinación de Spark](#) (consulte “[Una familia de Spark Joins](#)” en la página 187 en el [Capítulo 7](#)). Aquí se proporcionan ejemplos para cada tipo de unión.

Combinación de combinación de clasificación aleatoria (SMJ)

Con estas nuevas sugerencias, puede sugerirle a Spark que realice un SortMergeJoin al unir las tablas a y b o clientes y pedidos, como se muestra en los siguientes ejemplos. Puede agregar una o más sugerencias a una declaración SELECT dentro de /*+ ... */ bloques de comentarios:

```
SELECT /*+ MERGE(a, b) */ id FROM a JOIN b ON a.key = b.key SELECT /*+
MERGE(clientes, pedidos) */ * FROM clientes, pedidos DONDE
pedidos.custId = clientes.custId
```

Unión hash de difusión (BHJ)

De manera similar, para una combinación hash de transmisión, puede proporcionar una pista a Spark de que prefiere una combinación de transmisión. Por ejemplo, aquí transmitimos la mesa a para que se una con la mesa b y los clientes de la mesa que se unan con los pedidos de la mesa:

```
SELECT /*+ BROADCAST(a) */ id FROM a JOIN b ON a.key = b.key SELECT /*+
BROADCAST(clientes) */ * FROM clientes, pedidos DONDE
pedidos.custId = clientes.custId
```

Combinación hash aleatoria (SHJ)

Puede ofrecer sugerencias de manera similar para realizar uniones hash aleatorias, aunque esto se encuentra con menos frecuencia que las dos estrategias de unión admitidas anteriores:

```
SELECT /*+ SHUFFLE_HASH(a, b) */ id FROM a JOIN b ON a.key = b.key SELECT /*+
SHUFFLE_HASH(clientes, pedidos) */ * FROM clientes, pedidos DONDE
pedidos.custId = clientes.custId
```

Combinación de bucle anidado de reproducción aleatoria y replicación (SNLJ)

Finalmente, la combinación de bucle anidado de reproducción aleatoria y replicación se adhiere a una forma y sintaxis similares:

```
SELECCIONE /*+ SHUFFLE_REPLICATE_NL(a, b) */ id DESDE a ÚNASE b
```

Catalog Plugin API y DataSourceV2

Para no limitarse solo a la tienda de metadatos y el catálogo de Hive, la API experimental DataSourceV2 de Spark 3.0 amplía el ecosistema de Spark y ofrece a los desarrolladores tres capacidades principales. Específicamente, es:

- Permite conectar una fuente de datos externa para la gestión de catálogos y tablas.
- Admite la inserción de predicados en fuentes de datos adicionales con formatos de archivo compatibles como ORC, Parquet, Kafka, Cassandra, Delta Lake y Apache Iceberg.
- Proporciona API unificadas para transmisión y procesamiento por lotes de fuentes de datos para sumideros y fuentes

Dirigida a los desarrolladores que desean ampliar la capacidad de Spark para usar fuentes y receptores externos, la API de catálogo proporciona API programáticas y de SQL para crear, modificar, cargar y eliminar tablas del catálogo conectable especificado. El catálogo proporciona una abstracción jerárquica de funcionalidades y operaciones realizadas en diferentes niveles, como se muestra en la [Figura 12-5](#).

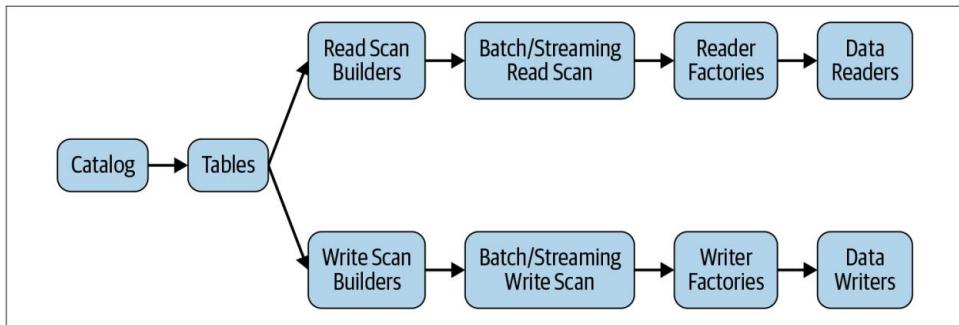


Figura 12-5. Nivel jerárquico de funcionalidad de la API del complemento de catálogo

La interacción inicial entre Spark y un conector específico es resolver una relación con su objeto Table real . El catálogo define cómo buscar tablas en este conector.

Además, Catalog puede definir cómo modificar sus propios metadatos, lo [que permite operaciones](#) como CREAR TABLA, ALTERAR TABLA, etc.

Por ejemplo, en SQL ahora puede emitir comandos para crear espacios de nombres para su catálogo. Para usar un catálogo conectable, habilite las siguientes configuraciones en su archivo spark defaults.conf:

```
spark.sql.catalog.ndb_catalog com.ndb.ConnectorImpl # implementación del conector
spark.sql.catalog.ndb_catalog.option1 valor1 spark.sql.catalog.ndb_catalog.option2 valor2
```

Aquí, el conector al catálogo de fuentes de datos tiene dos opciones: opción1->valor1 y opción2->valor2. Una vez que se han definido, los usuarios de la aplicación en Spark o SQL pueden usar los métodos API

DataFrameReader y DataFrameWriter o los comandos Spark SQL con estas opciones definidas como métodos para la manipulación de fuentes de datos. Por ejemplo:

```
-- En SQL
MOSTRAR TABLAS
ndb_catalog; CREAR TABLA
ndb_catalog.table_1; SELECCIONE * de
ndb_catalog.table_1; ALTERAR TABLA ndb_catalog.table_1

// En Scala
df.writeTo("ndb_catalog.table_1") val
dfNBD = chispa.read.table("ndb_catalog.table_1")
    .opción("opción1",
    "valor1") .opción("opción2", "valor2")
```

Si bien estas API de complementos de catálogo amplían la capacidad de Spark para utilizar fuentes de datos externas como sumideros y fuentes, todavía son experimentales y no deben usarse en producción. Una guía detallada de su uso está más allá del alcance de este libro, pero lo alentamos a consultar la documentación de la versión para obtener información adicional si desea escribir un conector personalizado para una fuente de datos externa como un catálogo para administrar sus tablas externas y sus asociados. metadatos



Los fragmentos de código anteriores son ejemplos de cómo puede verse su código después de haber definido e implementado sus conectores de catálogo y haberlos rellenado con datos.

Programador compatible con aceleradores

[proyecto hidrogeno](#), una iniciativa comunitaria para unir IA y big data, tiene tres objetivos principales: implementar el modo de ejecución de barrera, la programación consciente del acelerador y el intercambio de datos optimizado. Una implementación básica del [modo de ejecución de barrera](#) se introdujo en Apache Spark 2.4.0. En Spark 3.0, un [programador](#) básico se ha implementado para aprovechar los aceleradores de hardware como las GPU en las plataformas de destino donde Spark se implementa en modo independiente, YARN o Kubernetes.

Para que Spark aproveche estas GPU de manera organizada para cargas de trabajo especializadas que las usan, debe especificar los recursos de hardware disponibles a través de configuraciones. Luego, su aplicación puede descubrirlos con la ayuda de un script de descubrimiento. Habilitar el uso de GPU es un proceso de tres pasos en su aplicación Spark:

1. Escriba un script de detección que descubra las direcciones de las GPU subyacentes disponibles en cada ejecutor de Spark. Este script se establece en la siguiente configuración de Spark:

```
spark.worker.resource.gpu.discoveryScript=/ruta/a/script.sh
```

2. Configure la configuración para que sus ejecutores de Spark usen estas GPU descubiertas:

```
chispa.executor.resource.gpu.cantidad=2  
chispa.tarea.recurso.gpu.cantidad=1
```

3. Escriba código RDD para aprovechar estas GPU para su tarea:

```
importar org.apache.spark.BarrierTaskContext  
val rdd = ...  
rdd.barrier.mapPartitions { it =>  
    val context = BarrierTaskContext.getcontext.barrier() val gpus =  
    context.resources().get("gpu").get.addresses // iniciar proceso  
    externo que aprovecha GPU launchProcess(gpus)  
}
```



Estos pasos aún son experimentales y se continuará con el desarrollo en futuras versiones de Spark 3.x para admitir el descubrimiento continuo de recursos de GPU, tanto en la línea de comando (con envío de chispa) como en el nivel de tarea de Spark.

Transmisión estructurada

Para inspeccionar cómo le va a sus trabajos de transmisión estructurada con el flujo y reflujo de datos durante el curso de la ejecución, la interfaz de usuario de Spark 3.0 tiene una nueva pestaña de transmisión estructurada junto con las otras pestañas que exploramos en el [Capítulo 7](#). Esta pestaña ofrece dos conjuntos de estadísticas : información agregada sobre los trabajos de consulta de transmisión completados ([Figura 12-6](#)) y estadísticas detalladas sobre las consultas de transmisión, incluida la tasa de entrada, la tasa de proceso, la cantidad de filas de entrada, la duración del lote y la duración de la operación ([Figura 12-7](#)).

Streaming Query									
▼Active Streaming Queries (4)									
Name	Status	Id	Run ID	Start Time	Duration	Avg Input /sec	Avg Process /sec	Lastest Batch	
display_query_1	RUNNING	0b23d2af-7394-4cc7-9dd0-021d830e77fd	456c184b-46fb-4ce4-87e5-5955d0afd563	2020/06/03 23:01:16	3 minutes 5 seconds	33.98	35.72	24	
display_query_4	RUNNING	8311ea6a-07a3-40cf-b469-4aa58e71bc35	0afb8817-691f-416b-9366-ff935138ab33	2020/06/03 23:04:08	14 seconds 385 ms	52.85	61.66	7	
display_query_2	RUNNING	156e2285-fe84-4840-9383-824e0d9c5250	88c7a058-85fc-4d10-bdb0-4e1d9db5c8f1	2020/06/03 23:03:43	38 seconds 572 ms	56.29	58.60	19	
display_query_3	RUNNING	dc7ea927-5631-4325-05da	b9945a08-0c4f-4166-067a	2020/06/03 23:03:55	26 seconds 074 ms	52.50	57.03	13	

Figura 12-6. Pestaña de transmisión estructurada que muestra estadísticas agregadas de un trabajo de transmisión completado



La captura de pantalla de la [Figura 12-7](#) se tomó con Spark 3.0.0-preview2; con la versión final, debería ver el nombre de la consulta y el ID en el identificador de nombre en la página de la interfaz de usuario.

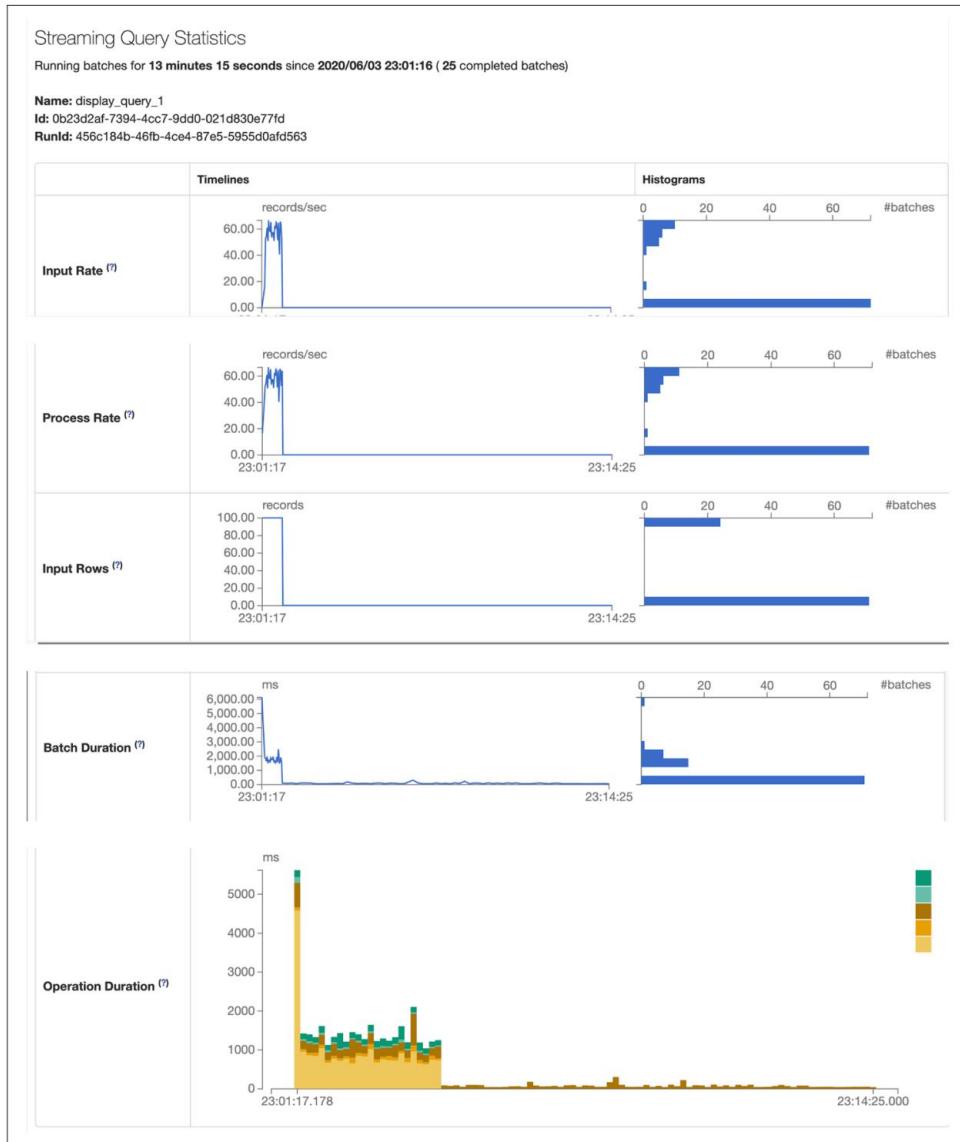


Figura 12-7. Mostrar estadísticas detalladas de un trabajo de transmisión completado

No se requiere configuración; todas las configuraciones funcionan directamente desde la instalación de Spark 3.0, con los siguientes valores predeterminados:

- `chispa.sql.streaming.ui.enabled=true`
- `chispa.sql.transmisión.ui.retenidoProgressUpdates=100`

- chispa.sql.streaming.ui.retainQueries=100

PySpark, UDF de Pandas y API de funciones de Pandas

Spark 3.0 requiere pandas versión 0.23.2 o superior para emplear cualquier método relacionado con pandas, como DataFrame.toPandas() o SparkSession.createDataFrame(pandas.DataFrame).

Además, requiere PyArrow versión 0.12.1 o posterior para usar la funcionalidad de PyArrow como pandas_udf(), DataFrame.toPandas() y SparkSession.createDataFrame(pandas.DataFrame) con spark.sql.execution.arrow.enabled configuración establecida en verdadero. La siguiente sección presentará nuevas características en Pandas UDF.

UDF de Pandas rediseñadas con sugerencias de tipo de Python

Las UDF de Pandas en Spark 3.0 se rediseñaron aprovechando las sugerencias de tipo de Python. Esto le permite expresar UDF de forma natural sin requerir el tipo de evaluación. Los UDF de Pandas ahora son más "Pythonic" y pueden definir por sí mismos lo que se supone que debe ingresar y generar el UDF, en lugar de especificarlo mediante, por ejemplo, @pandas_udf ("largo", PandasUDFType.SCALAR) como lo hizo en Spark 2.4 .

Aquí hay un ejemplo:

```
# Pandas UDF en Spark 3.0
import pandas as pd
from pyspark.sql.functions import pandas_udf

@pandas_udf("largo")
def pandas_plus_one(v: pd.Series) -> pd.Series:
    devuelve v + 1
```

Este nuevo formato proporciona varios beneficios, como un análisis estático más sencillo. Puede aplicar las nuevas UDF de la misma manera que antes:

```
df = chispa.rango(3)
df.withColumn("plus_one", pandas_plus_one("id")).show()

+---+-----+
| id|más_uno|
+---+-----+
| 0|      1|
| 1|      2|
| 2|      3|
+---+-----+
```

Compatibilidad con iteradores en Pandas UDF

Las UDF de Pandas se usan muy comúnmente para cargar un modelo y realizar operaciones distribuidas. La inferencia para el aprendizaje automático de un solo nodo y los modelos de aprendizaje profundo. Sin embargo, si un modelo es muy grande, entonces hay una gran sobrecarga para que Pandas UDF repita repetidamente cargue el mismo modelo para cada lote en el mismo proceso de trabajo de Python.

En Spark 3.0, las UDF de Pandas pueden [aceptar un iterador](#) de pandas.Series o pandas.DataFrame, como se muestra aquí:

de escribir iterador de importación

```
@pandas_udf('largo')
def pandas_plus_one(iterador: Iterador[pd.Series]) -> Iterador[pd.Series]:
    mapa de retorno (lambda s: s + 1, iterador)

df.withColumn("plus_one", pandas_plus_one("id")).show()

+---+-----+
| id|más_uno|
+---+-----+
| 0|      1|
| 1|      2|
| 2|      3|
+---+-----+
```

Con este soporte, puede cargar el modelo solo una vez en lugar de cargarlo para cada serie en el iterador. El siguiente pseudocódigo ilustra cómo hacer esto:

```
@pandas_udf (...)
def predecir (iterador):
    modelo = # cargar modelo
    para características en el iterador:
        modelo de rendimiento. predecir (características)
```

Nuevas API de funciones de Pandas

Spark 3.0 presenta algunos tipos nuevos de UDF de Pandas que son útiles cuando desea aplicar una función en un DataFrame completo en lugar de en columnas, como `mapInPandas()`, presentado en el [Capítulo 11](#). Estos toman un iterador de pandas.DataFrame como entrada y salida de otro iterador de pandas.DataFrame:

```
def pandas_filter(iterador:
    Iterador[pd.DataFrame]) -> Iterador[pd.DataFrame]:
    para pdf en iterador:
        producir pdf[pdf.id == 1]

df.mapInPandas(pandas_filter, esquema=df.esquema).show()

+---+
| identificación |
+---+
| 1|
+---+
```

Puede controlar el tamaño de pandas.DataFrame especificándolo en la configuración de `spark.sql.execution.arrow.maxRecordsPerBatch`. Tenga en cuenta que el tamaño de entrada y el tamaño de salida no tienen que coincidir, a diferencia de la mayoría de las UDF de Pandas.



Todos los datos de un cogrupo se cargarán en la memoria, lo que significa que si hay datos sesgados o ciertos grupos son demasiado grandes para caber en la memoria, podría tener problemas de OOM.

Spark 3.0 también presenta UDF Pandas de mapas coagrupados. La función `applyInPandas()` toma dos pandas.DataFrames que comparten una clave común y aplica una función a cada cogrupo. Los pandas.DataFrames devueltos luego se combinan como un solo marco de datos. Al igual que con `mapInPandas()`, no hay restricciones en la longitud de los pandas.DataFrame devueltos. Aquí hay un ejemplo:

```
df1 = chispa.createDataFrame([(1201,
    1, 1.0), (1201, 2, 2.0), (1202, 1, 3.0), (1202, 2, 4.0)], ("hora", "id", "v1"))
df2 = chispa.createDataFrame([(1201, 1, "x"), (1201, 2, "y")], ("hora", "id", "v2"))

def asof_join(izquierda: pd.DataFrame, derecha: pd.DataFrame) -> pd.DataFrame: return
    pd.merge_asof(izquierda, derecha, on="tiempo", by="id")

df1.groupby("id").cogrupo
    (df2.groupby("id"))
).applyInPandas(asof_join, "time int, id int, v1 double, v2 string").show()
```

```
+-----+
|tiempo| identificación | v1| v2|
+-----+
|1201| 1|1.0| x| 1202|
|3.0| x| 1201| 2|2.0|
y| 1202| 2|4.0| y|
+-----+
```

Funcionalidad modificada

Enumerar todos los cambios de funcionalidad en Spark 3.0 transformaría este libro en un ladrillo de varias pulgadas de grosor. Entonces, en aras de la brevedad, mencionaremos algunos notables aquí y lo dejaremos consultar las notas de la versión de Spark 3.0 para obtener detalles completos y todos los matices tan pronto como estén disponibles.

Idiomas compatibles y obsoletos Spark 3.0

es compatible con Python 3 y JDK 11, y se requiere la versión 2.12 de Scala. Todas las versiones de Python anteriores a la 3.6 y Java 8 están obsoletas. Si utiliza estas versiones obsoletas, recibirá mensajes de advertencia.

Cambios en las API de DataFrame y Dataset En versiones

anteriores de Spark, los AP de Dataset y DataFrame habían obsoleto el método unionAll() . En Spark 3.0 esto se invirtió y unionAll() ahora es un alias del método union() .

Además, las versiones anteriores de Dataset.groupByKey() de Spark dieron como resultado un conjunto de datos agrupado con la clave falsamente nombrada como valor cuando la clave no era de tipo estructura (int, cadena, matriz, etc.). Como tal, la agregación resulta de ds.groupByKey().count() en la consulta cuando se muestra, de manera contraria a la intuición, como (valor, recuento). Esto se ha corregido para dar como resultado (clave, conteo), que es más intuitivo. Por ejemplo:

```
// En Scala val
ds = chispa.createDataset(Seq(20, 3, 3, 2, 4, 8, 1, 1, 3)) ds.show(5)
```

```
+----+
|valor|
+----+
| 20|
| 3|
| 3|
| 2|
| 4|
+----+
```

```
ds.groupByKey(k=> k).count.show(5)

+-----+
|clave|recuento(1)|
+-----+
| 1 |      2|
3| |      3|
20| |      1|
4| 8|      1|
               1|
+-----+
solo se muestran las 5 filas superiores
```

Sin embargo, puede conservar el formato anterior si lo prefiere configurando spark.sql.legacy.dataset.nameNonStructGroupingKeyAsValue en verdadero.

Comandos de explicación de SQL y

DataFrame Para una mejor legibilidad y formato, Spark 3.0 presenta la capacidad DataFrame.explain (FORMAT_MODE) para mostrar diferentes vistas de los planes que genera el optimizador Catalyst. Las opciones FORMAT_MODE incluyen "simple" (predeterminado), "extendido", "coste", "codegen" y "formateado". Aquí hay una ilustración simple:

```
// En Scala
val strings =
  chispa.read.text("/databricks-datasets/learning-spark-v2/SPARK_README.md")
valor filtrado = cadenas.filtro($"valor".contains("Spark")) filtrado.cuenta()

# En cadenas
de Python = chispa
.read.text("/databricks-conjuntos de datos/learning-spark-v2/SPARK_README.md")
filtrado = strings.filter(strings.value.contains("Spark")) filtered.count()

// En Scala
filtrado.explain("simple")

# En Python
filtered.explain(mode="simple")

== Plano físico ==
*(1) Proyecto [valor#72]
+- *(1) Filtro (no es nulo(valor#72) Y Contiene(valor#72, Spark))
   +- Texto de FileScan [valor #72] Lote: falso, Filtros de datos: [no es nulo (valor #72),
   Contiene (valor n.º 72, Spark)], Formato: Texto, Ubicación:
   InMemoryFileIndex[dbfs:/databricks-datasets/learning-spark-v2/SPARK_README.md], PartitionFilters:
   [], PushedFilters: [IsNotNull(value), StringContains (valor,Spark)], ReadSchema: struct<valor:cadena>

// En Scala
filtered.explain("formateado")
```

```
# En Python
filtered.explain(mode="formatted")

== Plano físico ==
* Proyecto (3)
+- * Filtro (2)
  +- Escanear texto (1)

(1) Salida de
texto escaneado [1]: [valor #72]
Por lotes: falso
Ubicación: InMemoryFileIndex [dbfs:/databricks-datasets/learning-spark-v2/...
PushedFilters: [IsNotNull(valor), StringContains(valor,Spark)]
ReadSchema: struct<valor:cadena>

(2) Filtrar [ id de código de generación : 1]
Entrada [1]: [valor #72]
Condición : (no es nulo (valor n. º 72) Y contiene (valor n. º 72, Spark))

(3) Proyecto [ id de código de generación : 1]
Salida [1]: [valor#72]
Entrada [1]: [valor #72]

-- En SQL
EXPLICAR FORMATO
SELECCIONAR *
DESDE tmp_spark_readme
DONDE valor como "%Spark%"

== Plano físico ==
* Proyecto (3)
+- * Filtro (2)
  +- Escanear texto (1)

(1) Salida de
texto escaneado [1]: [valor #2016]
Por lotes: falso
Ubicación: InMemoryFileIndex [dbfs: /databricks-datasets/learning-spark-
v2/SPARK_README.md]
PushedFilters: [IsNotNull(valor), StringContains(valor,Spark)]
ReadSchema: struct<valor:cadena>

(2) Filtrar [id de código de generación : 1]
Entrada [1]: [valor#2016]
Condición : (no es nulo (valor # 2016) Y Contiene (valor # 2016, Spark))

(3) Proyecto [id de código de generación : 1]
Salida [1]: [valor#2016]
Entrada [1]: [valor#2016]
```

Para ver el resto de los modos de formato en acción, puede probar el cuaderno en el repositorio de [GitHub del libro](#). Consulte también las [guías de migración](#) de Spark 2.x a Spark 3.0.

Resumen

Este capítulo proporcionó un resumen rápido de las nuevas funciones en Spark 3.0. Nos tomamos la libertad de mencionar algunas características avanzadas que vale la pena mencionar. Operan bajo el capó y no a nivel de API. En particular, echamos un vistazo a la poda de partición dinámica (DPP) y la ejecución de consulta adaptativa (AQE), dos optimizaciones que mejoran el rendimiento de Spark en el momento de la ejecución. También exploramos cómo la API de catálogo experimental amplía el ecosistema de Spark a almacenes de datos personalizados para fuentes y receptores para datos por lotes y de transmisión, y observamos el nuevo programador en Spark 3.0 que le permite aprovechar las GPU en los ejecutores.

Complementando nuestra discusión sobre la interfaz de usuario de Spark en el [Capítulo 7](#), también le mostramos la nueva pestaña Streaming estructurado, que proporciona estadísticas acumuladas sobre trabajos de streaming, visualizaciones adicionales y métricas detalladas en cada consulta.

Las versiones de Python anteriores a la 3.6 están obsoletas en Spark 3.0, y las UDF de Pandas se han rediseñado para admitir sugerencias de tipo de Python e iteradores como argumentos. Hay Pandas UDF que permiten transformar un DataFrame completo, así como combinar dos DataFrames coagrupados en un nuevo DataFrame.

Para una mejor legibilidad de los planes de consulta, `DataFrame.explain(FORMAT_MODE)` y `EXPLAIN FORMAT_MODE` en SQL muestran diferentes niveles y detalles de planes lógicos y físicos. Además, los comandos SQL ahora pueden tomar sugerencias de unión para toda la familia de uniones compatibles con Spark.

Si bien no pudimos enumerar todos los cambios en la última versión de Spark en este breve capítulo, le recomendamos que explore las notas de la versión cuando se publique Spark 3.0 para obtener más información. Además, para obtener un resumen rápido de los cambios que enfrentan los usuarios y detalles sobre cómo migrar a Spark 3.0, le recomendamos que consulte las guías de migración.

Como recordatorio, todo el código de este libro se probó en Spark 3.0.0-preview2 y debería funcionar con Spark 3.0 cuando se lance oficialmente. Esperamos que haya disfrutado leyendo este libro y aprendido de este viaje con nosotros. ¡Le agradecemos su atención!

Índice

UNProgramador consciente del acelerador, [351](#)ACM (Asociación de Maquinaria Informática), [6](#)acciones, [28-30, 61](#)adición de columnas, [63, 152](#)agg(), [187](#) agregado(), [162](#)agregaciones, [66, 239-246](#)

propiedad

allowUnquotedFieldNames, [101](#)Amazon S3, [89](#)AMPLab, [3](#)Fase de análisis (Spark SQL), [81](#)funciones analíticas, [149](#)Formato de flecha Apache, [115](#)Apache Cassandra, [89, 137, 231](#)Colmena Apache, [89, 113-155](#)Apache Hudi, [272](#)Iceberg apache, [272](#)

Apache Kafka

sobre, [8, 15](#)lectura de, [228](#)Streaming estructurado y, [228-230](#) escribiendo a, [229](#)Apache Meses, [12, 177](#)Grupos de reunión de Apache Spark, [16](#)Append mode (Structured Streaming), [212, 215, 245](#)aplicaciones, Spark about, [26](#) conceptos de, [25-28](#)depuración, [204](#) controlador y ejecutores, [10, 12](#)programas de controlador, [10](#)optimización y ajuste, [173-205](#) usandoSpark SQL en, [84-89](#) métodoapproxQuantile(), [68](#) approx_count_distinct(), [239](#)AQE (ejecución de consulta adaptativa), [345-348](#)cálculos arbitrarios con estado, [253-261](#) funciones de tipo matriz, [139-141](#) función arrays_overlap(), [139](#)función array_distinct(), [139](#) función array_except(), [139](#)función array_intersect(), [139](#) array_join() función, [139](#)función array_max(), [139](#) función array_min(), [139](#)función array_position(), [139](#) función array_remove(), [139](#)función array_repeat(), [139](#) función array_sort(), [139](#)función array_union(), [139](#) función array_zip(), [139](#)artefactos, [326](#)ASF (Fundación de Software Apache), [2](#)AST (árbol de sintaxis abstracta), [81](#)atomicidad, de lagos de datos, [270](#)cambios de datos de auditoría con historial de operaciones, [282](#)método avg(), [67](#)propiedad de esquema avro, [106](#)Avro, como fuente de datos para DataFrames y tablas SQL, [104](#)AWS SageMaker, [334](#)Azure Cosmos DB, [134](#)Azur ML, [334](#)

B

embolsado, 313
 modo de ejecución de barrera, 351
 implementación por lotes, 332
 Beeline, consultando con, 119
 BHJ (unión hash de transmisión), 188, 349
 grandes datos, 1
 Tabla grande,
 1 directorio bin, 21, 21
 archivos binarios, como fuente de datos para DataFrames
 y tablas SQL, 110 muestras de arranque, 313 variables
 de transmisión, 188, 344 método bucketBy(), 96 fuentes de
 datos integradas, 83-112, 94 funciones integradas, 139-141,
 239 bytecode, 7, 23, 25

selección aleatoria de características por,
 313 cambio de nombre, 63, 153 archivos
 de valores separados por comas (archivos CSV), 102
 adopción/expansión de la comunidad, de Spark, 16
 Modo completo (streaming estructurado), 212, 215, 245

tipos de datos complejos, 49, 139-141
 propiedad de compresión, 101, 103, 106 función
 de cálculo, 44 función concat(), 139 archivo
 conf.spark-defaults.conf, 173 ajustes de
 configuración, 173-176 utilizados en este libro,
 visualización xviii , 173-176 configurar Spark,
 con Delta Lake, 274 consistencia, de lagos
 de datos, 270 aplicaciones continuas, 15
 Modo continuo (Streaming estructurado),
 217 Modelo de Streaming continuo, 8 Modo de
 disparo continuo, 219 método de correlación(), 68

C

cache(), 183-187
 almacenamiento en caché,
 93, 183-187 cardinalidad() función,
 139 clase de caso, 71, 158
 sentencia CASE, 152
 Casandra, 89, 137, 231
 Catálogo API, 93, 349-351
 Optimizador de catalizadores, xvi, 16, 77-82, 170
 CBO (optimizador basado en costos), 81
 CDC (cambio de captura de datos), 271
 puntos de control, 217, 262 clasificación,
 286-287, 292, 304 condiciones de cláusula,
 281 modo cliente, 12 método close(), 233
 administradores de clúster, 10, 12, 176, 178
 recurso de clúster aprovisionamiento, 262
 agrupamiento, 286, 288, 302 ejemplos de
 código, utilizando, xviii

costos

atenuante, 170 de
 bases de datos, 267
 de latencia, 209
 count(), 29, 66, 183, 215
 countDistinct(), 239
 Ejemplo de conteo de M&Ms, 35-39 método
 de covarianza(), 68
 CrossValidator, 317
 Archivos CSV, como fuente de datos para DataFrames y
 tablas SQL, 102
 función cube(), 235 función
 cubed(), 116
 customStateUpdateFunction(), 254

D

Fase de generación de código (Spark SQL), 81
 codegen, habilitación en Spark SQL, 189 cogroup(),
 356 método collect(), 67 collect_list(), 138
 collect_set(), 239

DAG (gráfico acíclico dirigido), 4, 27 datos

Objeto de columna, 54
 columnas agregando,
 63, 152 eliminando,
 63, 152 en DataFrames,
 54

adaptando cambios, 279 auditando
 cambios con historial de operaciones,
 282
 deduplicación, 281
 diversidad de formatos para soluciones de almacenamiento,
 265
 gobierno de, como una característica de las casas del lago,
 271
 crecimiento en tamaño de, 267

- cargar en tablas de Delta Lake, [275](#) transformar, [214](#), [279-282](#) actualizar, [280](#) corrupción de datos, hacer cumplir el esquema en escritura para evitar, [278-279](#) directorio de datos, [21](#) tareas de ingeniería de datos, [15](#) ingenieros de datos, [xv](#), [15](#) evolución de datos, [323](#) datos ingesta, [290](#) lagos de datos, [265-284](#) sobre, [268](#) bases de datos, [266-268](#) lagos, [271](#) limitaciones de, [270](#) soluciones de almacenamiento óptimas, [265](#) lectura desde, [269](#) escritura en, [269](#) tareas de ciencia de datos, [14](#) científicos de datos, [xv](#) fuentes de datos sobre, [89](#) incorporado, [94](#) para DataFrames, [94-112](#) para tablas SQL, [94-112](#) streaming, [226-234](#)
- API de fuentes de datos, [94](#)
- tipo de datos sobre, [48](#)
- complejo, [49](#) estructurado, [49](#) soporte para diversos, como una característica de las casas del lago, [271](#) bases de datos sobre, [266](#) limitaciones de, [267](#) lectura de, [267](#) escritura de, [267](#)
- DataFrameWriter sobre, [5](#) como fuente de datos para DataFrames y tablas SQL, [96](#) usando, [58-60](#)
- Edición de la comunidad de bricks, [xviii](#), [34](#)
- API de DataFrame
- sobre, [16](#), [47](#)
 - columnas, [54](#)
 - operaciones comunes, [58-68](#) creación de DataFrames, [50-54](#) tipos de datos, [48](#) ejemplo de, [68](#) expresiones, [54](#) filas, [57](#) esquemas, [50-54](#)
- API de conjunto de datos
- acerca de, [16](#), [69](#)
 - cambios en, [357](#)
 - creación de conjuntos de datos, [71](#)
 - Operaciones de conjunto de datos, [72](#) ejemplo de, [74](#) objetos escritos, [69](#)
- conjuntos de datos
- en comparación con DataFrames, [74](#) convertir DataFrames en, [166](#)
- tipos de datos estructurados, [49](#)
- DataFrame.cache(), [183](#)
- DataFrame.persist(), [184](#)
- DataFrameReader sobre, [5](#) como fuente de datos para DataFrames y tablas SQL, [94](#) usando, [58-60](#) DataFrames, [144-155](#) cambios a, [357](#) en comparación con conjuntos de datos, [74](#) conversión a Conjuntos de datos, [166](#) creación, [50-54](#) fuentes de datos para, [94-112](#) funciones de orden superior en, [138-144](#) evaluación perezosa y, [29](#) administración de memoria para, [167](#) lectura de archivos Avro, [104](#) lectura de archivos binarios, [110](#) lectura de archivos CSV en, [102](#) lectura de datos en, [93](#) lectura de archivos de imagen, [108](#) lectura de archivos JSON en, [100](#) lectura de archivos ORC en, [107](#) lectura de archivos Parquet en, [97](#) comandos de explicación SQL, [358](#) transmisión, [214](#) transformaciones, [222](#) escritura en archivos Avro, [105](#) escritura en CSV, [103](#) escritura en archivos JSON, [101](#) escritura en archivos ORC, [108](#) escritura en archivos Parquet, [99](#) escritura en tablas Spark SQL, [99](#)

costos de uso, 170
 creación, 71
 codificadores, 168
 JavaBeans para, 158
 administración de memoria para, 167
 operaciones, 72 API única, 157-160

Spark SQL y, 157-172
 trabajando con, 160-167
 DataSourceV2, 349-351
 propiedad dateFormat, 101, 103 función day(), 65
 cadena DDL (lenguaje de definición de datos), 51
 depuración de Spark, 204 árboles de decisión, 308-313
 deduplicación de datos, 281 acciones de eliminación, 271, 281 eliminación de datos relacionados con el usuario, 280

Delta Lake
 sobre, 273
 casas del lago de construcción con Apache Spark y, 274-283
 configurar Apache Spark con, 274 cargar datos en tablas, 275 cargar flujos de datos en tablas, 277 DenseVector, 298 función dense_rank(), 151 dependencias, 44 modos de implementación, 12, 330-335 idiomas obsoletos, 357 método describe(), 68 desarrolladores , Spark y, 14-17 directorios, 21 flujos discretizados, 9 DISK_ONLY nivel de almacenamiento, 184 datos distribuidos, particiones y, 12 ejecución distribuida, 10-14 ajuste de hiperparámetro distribuido, 337-340 gestión de estado distribuido, 236 notación de punto (), 72 descargando Spark, 19-22 DPP (poda de partición dinámica), 343-345 controlador, 10 programas de controlador, 10 método drop(), 64, 152 columnas de eliminación, 63, 152 Dropwizard Metrics, 224 DSL (lenguaje específico del dominio), 44 API DStream, 209

DStreams (flujos discretizados), 9 asignación dinámica de recursos, 177

E facilidad de uso, de Spark, 5 Los elementos del aprendizaje estadístico (Hastie, Tibshirani y Friedman), 309 función element_at(), 139, 139 codificadores (conjuntos de datos), 168 codificación, one-hot, 297 garantías de extremo a extremo exactamente una vez, 221 enfoque de conjunto, 313 pestaña Entorno (interfaz de usuario de Spark), 203 errores, corrección, 280 estimadores , 290, 295 método estimator_name.fit(), 295 ETL (extraer, transformar y cargar), 15 evaluar modelos, 302-306 orden de evaluación, 115 tiempos de espera de eventos, 259 ventanas de tiempo de eventos, agregaciones con, 239-246

garantías de una sola vez, 221 archivo de ejemplos, 21 ejecución, orden de, 323 albaceas, 12

Pestaña Ejecutores (Spark UI), 200 existe() función, 143 experimentos, 325 explotar() función, 138, 235 exportar patrones, para inferencia en tiempo real, 334 función expr(), 145 expresiones, en DataFrames, 54 extensibilidad, de Spark, 5 fuentes de datos externas, 113-155

Apache Casandra, 137 Azure Cosmos DB, 134 DataFrames comunes, 144-155 funciones de orden superior en DataFrames y Chispa SQL, 138-144 Base de datos JDBC, 129-131 MongoDB, 137 Servidor MS SQL, 136 mysql, 133 PostgreSQL, 132 Copo de nieve, 137 Operaciones Spark SQL, 144-155 Base de datos SQL, 129-131 Tabla, 122-129

- Tolerancia a fallas F , 2, 9, 15, 185, 209, 222
 administración de estado tolerante a fallas, 236
 formatos de archivo
 sobre, 76
 Archivos CSV, 102
 lagos de datos y 269
 compatibilidad con diversidad de 269
 archivos
 acerca de,
 21 lectura de, 226
 Streaming estructurado y, 226 escribir
 en, 227 sistemas de archivos, 89, 269
 método filter(), 28, 29, 61, 72, 73, 143, 157, 162, 170, 215, 235 filtrado, DataFrames y, 61 método fit() , 296
 ajuste, 295 método flatMap(), 157, 170, 215, 235
 flatMapGroupsWithState(), 253, 256, 261 función flatten(), 139
 fmin(), 339 método foreach(), 216, 230, 233-234 foreachBatch()
 () método, 216, 230, 281 formato() método, 94, 96, 100
 frequentItems() método, 68
- G**
 Friedman, Jerome, Los elementos de la estadística
 Aprendizaje, 309
 función from_json(), 138
 programación funcional, funciones de orden superior y, 162-167 funcionalidad, cambiada, 357-359
- recolección de basura, 167, 178, 199
 RGPD (Reglamento General de Protección de Datos), 280
 generalización, con flatMapGroupsWithState(), 261
 filas genéricas, 69
 métodos getter, 70
 función get_json_object(), 138
 GFS (sistema de archivos de Google), 1
 Ghemawat, Sanjay, The Google File System, 268 agregaciones globales, 238 vistas temporales globales, 92
 Gobioff, Howard, El sistema de archivos de Google, 268
 google, 1
- El sistema de archivos de Google (Ghemawat, Gobioff y Leung), 268
 GraphFrames, 9
 interfaz gráfica de usuario, 31
 Biblioteca GraphX, 6, 9
 GridSearchCV, 339
 Declaración GROUP BY, 138 método
 groupBy(), 30, 66, 73, 157, 182, 187, 244, 337 groupByKey(), 254, 256 UDF de Pandas agrupados agrupados, 116
 agregaciones agrupadas, 238 UDF de Pandas de mapas agrupados, 116
- H**
 Hadoop, 2, 268
 Hadoop HILO, 12
 Hastie, Trevor, Los elementos de la estadística
 Aprendizaje, 309
 Base H, 5
 HDFS (Sistema de archivos distribuido de Hadoop), 2, 268 API estructuradas de alto nivel, 25 funciones de orden superior, 138-144, 162-167
 Colmena, 89, 113-155
 Hive ORC SerDe (serialización y deserialización) tablas, 107
- objetivo HiveContext, 11
 HiveServer2, 120
 Hyperopt, 339
 configuraciones de hiperparámetros, 317 ajuste de hiperparámetros, 307 distribución, 337-340
 validación cruzada de k-fold, 316-319
 canalizaciones de optimización, 320-321
 modelos basados en árboles, 307-316
- I**
 columna de identificación (StreamingQuery), 224 propiedad ignoreExtension, 106 imágenes, como fuente de datos para DataFrames y tablas SQL, 108
 ejecución incremental, 234
 incrementalización, 211 propiedad inferSchema, 103 uniones internas, 248-252 definición de fuentes de entrada y salida, 213 formatos de archivo

- sobre, 76
CSV, 102 lagos
de datos y 269 soporte
para diversas cargas de trabajo, 269 columnas

instalación de R, 21
grupos interactivos, administración mediante tiempos de espera, 257-261 shell interactivo, 274 aislamiento, de lagos de datos, 270 compatibilidad con iteradores, en Pandas UDF, 355
- J Java, 55, 157-160
Serialización de Java, 184
java.io.Serializable, 158 Clase
JavaBean, 71 JavaBeans, para conjuntos de datos, 158 Base de datos JDBC, 89, 129-131 JDK (Java Development Kit), 41 Joblib, 338 trabajos, 26, 27 Pestaña Trabajos (interfaz de usuario de Spark), 198 operaciones de unión sobre, 187 unión hash de difusión (BHJ), 188 combinación de clasificación aleatoria (SMJ), 189-197 unión(), 182, 187 uniones sobre, 148 BHJ (unión hash de difusión), 349 SHJ (combinación hash aleatoria), 349 SMJ (combinación de combinación de clasificación aleatoria), 349 SNLJ (combinación de bucle anidado de reproducción aleatoria), 349 Spark SQL, 348 archivos JSON (notación de objetos de JavaScript), 53, 100 como una fuente de datos para DataFrames y tablas SQL, 100
- Validación cruzada k-fold, 316-319
Guía de integración de Kafka, 229
Kaggle, 335
Karau, Holden, XV
Kay, Alan, 3
koalas, 340
- Konwinski, Andy, XV
Biblioteca de serialización Kryo, 168, 170, 184
Kubernetes, 12, 21
Kuo, Kevin, Mastering Spark with R, xvi, 285
- ## L
- casas del lago
Apache Hudi, 272
Apache Iceberg, edificio 272 con Apache Spark y Delta Lake, 274-283
- Delta Lake, 273
funciones de , 271
lambda, 170 idiomas, 357 latencia, 209, 330
evaluación perezosa, 28-30 nodo hoja, de árboles de decisión, 308 aprendizaje, 295
- Datos de préstamos del club de préstamos, 274
Leung, Shun-Tak, The Google File System, 268 versiones de biblioteca, 323 linaje, 29 regresión lineal, 294
- Modelo de regresión lineal, 295, 306
Linux Foundation, 273 método load(), 94 cargar y guardar
fluxos de dados en tabelas de Delta Lake, 277 formatos de arquivo
- acerca de, 76 archivos CSV, 102 lagos de datos y, 269 soporte para diversidad de, 269 sistemas de archivos, 89, 269 en tabelas de Delta Lake, 275 datos estructurados, utilizando Spark, 91 modelos de carga, 306 máquina local, 23 log-normalmente distribuida, 305 Log4j.properties.template, 37 registro, 325, 327 Fase de optimización lógica (Spark SQL), 81 regresión logística, 287 LogisticRegressionModel, 306 propiedad lowerBound, 130 método lr.fit(), 296 Luraschi, Javier, Mastering Spark with R, xvi , 285

M

aprendizaje automático (ML)
 sobre, 286 creación de
 modelos usando estimadores, 295 creación de canalizaciones, 296-302 creación de conjuntos de datos de prueba, 291-293 creación de conjuntos de datos de entrenamiento, 291-293 ingestión de datos, 290 diseño de canalizaciones, 289-307 evaluación de modelos, 302-306 exploración, 290 ajuste de hiperparámetros, 307-321 regresión lineal, 294 modelos de carga, 306 razones para usar Spark, 289 modelos de ahorro, 306 supervisados, 286 no supervisados, 288 con MLlib, 285-321 ingenieros de aprendizaje automático, xv transformaciones con estado administradas, 237 tablas administradas, 89 funciones de mapa, 139 método map(), 73, 138, 157, 162, 163, 170, 215,

arquitectura de microservicios, Spark Streaming, método 208 min(), 67 costos de mitigación, 170 MLeap, 334 MLflow, 323, 324-330 MLflow Model Registry, 332 MLflow Models, 332 MLflow Projects, 330 MLflow Tracking, 325-330 MLlib library acerca de, xv, 6, 7 (ver también aprendizaje automático (ML)) aprendizaje automático (ML) con, 285-321 opciones de implementación de modelos con, 330 model.transform(), 332 modelos acerca de, 326 construcción usando estimadores, 295 evaluación, 302-306 cargando, 306 administrando, 323-330 guardando, 306 basado en árbol, 307-316 componente de modelos (MLflow), 324 modificaciones, 151-155 modularidad, de Spark, 5 MongoDB, 137 función mes(), 65 MR (MapReduce), 1 MS SQL Server, 136 propiedad multilínea, 101, 103 entorno multiusuario, 177 base de datos MySQL, 86, 133

235

unión solo del lado del mapa,
 188 mapGroupsWithState(), 253, 256, 261 método mapInPandas(), 336, 356 mapPartitions(), 119 función map_concat(), 139 función map_form_arrays(), 139 función map_from_entries(), 139

Dominar Spark con R (Luraschi, Kuo y Ruiz), xvi, 285
 objeto matriz, 292
 Maven, 133, 134
 método max(), 67
 método mean(), 239
 gestión de memoria, para conjuntos de datos y datos. Marcos, 167
 MEMORY_AND_DISK nivel de almacenamiento, 184
 MEMORY_AND_DISK_SER nivel de almacenamiento, 184
 MEMORY_ONLY nivel de almacenamiento, 184
 MEMORY_ONLY_SER nivel de almacenamiento, 184
 merge(), alterando los datos de cambio en las tablas usando, 281

Mesos (see Apache Mesos)
 metadata, 93, 326 metrics, 224,
 326

nota

Algoritmo Naive Bayes, 287
 dependencias estrechas, 30
 Netflix, 272
 modelos que no son MLlib, aprovechando Spark para, 336-340
 análisis no basados en SQL, para bases de datos, 268
 agregaciones de transmisión no basadas en el tiempo, 238 verificación nula, 115 columna numInputRows (StreamingQuery), 224 propiedad numPartitions, 130

Numpy, 14

oh objetos, 69

memoria de montón fuera de Java, 168
 Nivel de almacenamiento OFF_HEAP, 184
 OHE (codificación one-hot), 297
 OLAP (procesamiento analítico en línea), 267
 OLTP (procesamiento de transacciones en línea), 267
 Modo una vez (streaming estructurado), 217
 Codificador OneHot, 298
 ONNX (intercambio de redes neuronales abiertas), 334
 Excepciones OOM (memoria insuficiente), 67 método open(), 233 apertura, de soluciones de almacenamiento, 265 historial de operaciones, auditoría de cambios de datos con, 282
 optimización de aplicaciones Spark, 173-205 método option(), 94, 96, 100 acciones opcionales, 281
 ORC, como fuente de datos para DataFrames y tablas SQL, 106 orden de ejecución, 323 método orderBy(), 30, 66 uniones externas, 252 modos de salida (Structured Streaming), 212, 215,
 245
 sumidero de salida, 215 sobrescribir(), 306

PAG
 Algoritmo PageRank, 9
 Pandas sobre, 340 API de función, 354, 356 funciones definidas por el usuario (UDF), 115-117, 336, 354, 355 operaciones paralelas, 323 parámetro de parallelismo, 320 paralelismo, maximización, 180 parallelizar(), 119 parámetros, 326
 ParamGridBuilder, 317
 Parquet about, 60, 96 como fuente de datos para DataFrames y tablas SQL, 97-100 lectura de archivos en DataFrames, 97 lectura de archivos en tablas Spark SQL, 97 escritura de DataFrames en, 99 propiedad de la columna de partición, 130 particiones sobre, 44
 creación, 181
 datos distribuidos y, 12
 importancia de, 130 número de para barajar, 262 barajar, 182 rendimiento, 262, 265 persistir(), 183-187 persistencia, almacenar datos en caché y, 183-187
 Fase de planificación física (Spark SQL), 81 biblioteca de serialización pickle (Python), 115 Pipeline API, 289 creación de pipelines, 296-302 definido, 290 diseño, 289-307 optimización, 320-321 pivot, 153
 PMML (lenguaje de marcado de modelo predictivo), 335 puerto 4040, 197 Base de datos PostgreSQL, 86, 132 método process(), 233 columnaprocessedRowsPerSecond (Streaming) Consulta, 224 detalles de procesamiento, especificación, 216 motor de procesamiento, lagos de datos y, 269 tiempos de espera de tiempo de procesamiento, 257 Desencadenador ProcessingTime, 217 Proyecto Hidrógeno, 15, 351 Project Tungsten, 4, 16, 82, 119, 167 proyecciones, DataFrames y 61 componentes de proyectos (MLflow), 324 métricas de publicación con Dropwizard Metrics, 224 repositorio PyPI, 20 PySpark, 115-117, 354 Carcasa PySpark, 22-25 Columnas de Python y, 55 tipos de datos, 49 sugerencias de tipos, 354
 Monitoreo de consultas Q activo, 223-225 instantáneas de tablas, 283 comenzando, 218 Transmisión estructurada, 213-225 con Spark SQL Shell, 119

R

biblioteca R, 21
R2, 302-306
R2D3, 309
bosques aleatorios, 313-316
randomSplit(), 293 funciones de clasificación, 149
RDBS (sistemas de gestión de bases de datos relacionales), 1
RDD (Conjunto de datos distribuido resistente), 5, 16, 43, 75
rdd.getNumPartitions(), 13 read(), 29 leyendo archivos Avro en DataFrames, 104 archivos Avro en tablas Spark SQL, 105 archivos binarios en DataFrames, 110 archivos CSV en DataFrames, 102 archivos CSV en tablas Spark SQL, 102 de lagos de datos , 269 de bases de datos, 267 de archivos, 226 de Kafka, 228 archivos de imagen en DataFrames, 108 archivos JSON en DataFrames, 100 archivos JSON en tablas Spark SQL, 100 archivos ORC en DataFrames, 107 archivos ORC en tablas Spark SQL, 107 archivos Parquet en DataFrames, 97 archivos Parquet en tablas SQL, 97 tablas en DataFrames, 93 archivo README.md, 21 inferencia en tiempo real, patrones de exportación para, 334 receptores, 169 propiedad de nombre de registro, 106 modelo de procesamiento de registro a la vez, 207 propiedad recordNamespace, 106 rediseño de Pandas UDF, 354 función reduce(), 144, 162 reduceByKey(), 187 componente de registro (MLflow), 324 árboles de decisión de regresión, 308-313 lineal, 294 logístico, 287 bosques aleatorios, 313-316 cambio de nombre () método, 153 columnas de cambio de nombre, 63, 153 REST API, 324

función inversa(), 139 RFórmula, 300 RISELab, 3 RMSE (error cuadrático medio), 302 rollup(), 235 raíz, de árboles de decisión, 308 Objetos de fila, 57 filas genérico, 69 en DataFrames, 57 selección aleatoria de características por, 313 Ruiz, Edgar, Mastering Spark with R, xvi, 285 columna runID (StreamingQuery), 224 ejecución de consultas Spark SQL, 120 ejecuciones, 325 arquitectura de tiempo de ejecución (Spark), 11, 74, 96, 170, 203, 234, 345

S

datos de muestra, 160, 162-167 método sampleBy(), 68 método save(), 96 método saveAsTable(), 96 guardar modelos, 306 sbt (herramienta de compilación Scala), 40 Scala creación de aplicaciones independientes en, 40 clases de casos en, 71 columnas y, 55 API única para, 157-160 usando, 22-25

Scala shell, 23, 274 escalabilidad de bases de datos, 268 de soluciones de almacenamiento, 265 Spark, 177-182 UDF escalares de Pandas, 116 SCD (dimensión que cambia lentamente), 271 programadores, 351 cumplimiento/gobernanza del esquema, 271, 278-279 método schema(), 94

SchemaRDDs, 43 esquemas, 50-54, 279 scikit-learn, 289, 310, 312, 336, 339 motor Tungsten de segunda generación, 167 método select(), 28, 61, 73, 162, 215, 235 función selectExpr() , 138 garantías semánticas, con marcas de agua, 245 propiedad sep, 103

función secuencia(), 139
 SerDe (serialización y deserialización), 169
 tiburón, 113
 conchas
 PySpark, 22-25
 Escala, 23, 274
 Chispa, 25, 85, 119
 Chispa SQL, 119
 SHJ (combinación hash aleatoria), 349 particiones aleatorias, 182
 servicio aleatorio, 179 función aleatoria(), 139 aleatorias , 187, 262

SIMD (instrucción única, datos múltiples), 167 API singulares, 157-160 sumideros
 personalizado,
 230-234 opciones para, 222 streaming, 226-234 sesgar, 199 sklearn, 332 función slice(), 139

SMJ (combinación de combinación de clasificación aleatoria), 189-197, 349 instantáneas, consulta, 283
 SNLJ (unión de bucle anidado de reproducción aleatoria y replicación), 349
 Snowflake, 137
 software, usuario en este libro, xviii
 sortBy(), 187 opciones de fuente, 222
 límites de tasa de fuente, configuración, 262
 Chispa
 acerca de, xvi,
 4 conceptos de aplicación, 25-28
 construcción de casas junto al lago con Delta Lake y, 274-283
 adopción/expansión de la comunidad de, 16
 características de diseño de, 4 desarrolladores y, 14-17 directorios, 21 ejecución distribuida, 10-14 descargas, 19-22 primeros años de, 3 facilidad de uso de, 5 evolución de, 1 extensibilidad de, 5 archivos, 21 formato interno en comparación con Java Object
 formato, 168

aprovechando para modelos que no son MLlib, 336-340 modularidad de, 5 escalado, 177 velocidad de, 4 estructuración, 44-47 usos para, 14-17 Spark + AI Summit, 16 Spark API, 16 Spark Cassandra Connector, 231 Spark Core, 343 Spark shell, 25, 85, 119 Spark SQL about, 6, 7, 76, 343 Apache Hive y, 113-155 ejemplos de consultas básicas, 84-89 Optimizador Catalyst, 77-82 Conjuntos de datos y, 157-172 orden de evaluación, 115 funciones de orden superior en, 138-144 base de datos JDBC, 89, 129-131 uniones, 348 comprobación nula, 115 operaciones, 144-155 tablas, 89-94 uso e interfaz, 83 funciones definidas por el usuario (UDF), 114 uso en Aplicaciones Spark, 84-89 vistas, 89-94 Servidor Spark Thrift JDBC/ODBC, 121 Guía de ajuste de Spark, 320 Interfaz de usuario de Spark, 31-33, 197, 197-205 Interfaz de usuario web de Spark (consulte la interfaz de usuario web) spark-shell, 31 chispa -enviar script, 21, 174, 179, 197, 352 spark.executor.memory, 178 opción spark.local.dir, 182 paquete spark.ml, 8, 312 paquete spark.mllib, 8 spark.read(), 247 chispa función .read.csv(), 60 spark.readStream(), 247, 333 interfaz programática spark.sql e, 85 Objeto SparkConf, 11 Objeto SparkContext, 11, 263 Agua con gas, 335 Sparklyr, 21 Proyecto SparkR, 21 SparkSession, 10, 26, 85, 92 SparseVector, 298

velocidad, de Spark, 4
 Bases de datos SQL, 90, 129-131
 Pestaña SQL (interfaz de usuario de Spark), 202
Tablas SQL
 sobre, 60
 almacenamiento
 en caché, 93
 creación, 119 fuentes de datos
 para, 94-112 inserción de datos,
 120 lectura de archivos Avro, 105
 lectura de archivos CSV, 102 lectura
 de archivos JSON, 100 lectura de
 archivos ORC, 107 lectura de archivos
 Parquet en, 97 escribiendo DataFrames
 en, 99 método sql(), 84

objeto SQLContext, 11
 SQLLine CLI, 120
 etapas, 26, 28
 Pestaña Etapas (interfaz de usuario de
 Spark), 198 creación de aplicaciones
 independientes en Scala, 40
 Ejemplo de contar M&Ms, 35-39
 Proyecto Scala/Java usando coordenadas Maven,
 274

Administrador de clúster independiente,
 sintaxis de 12 estrellas, 281 método
 start(), 218, 221 start-thriftserver.sh, 121
 método stat(), 68 agregaciones de
 transmisión con estado, 238-246
 transformaciones con estado, 215, 235 transformaciones
 sin estado, 215, 235 estático asignación de recursos,
 177 stddev() , 239 lagos de datos de almacenamiento
 y, 269 importancia de lo óptimo, 265 niveles de, 184
 pestaña AlmaSpark, 200 (interfaz de usuario de dujo),
 246 uniones de flujo a flujo, 248-252 transmisión
 (consulte Streaming) streaming DataFrame, 214
 uniones de streaming, 246, 248-252 consultas de
 streaming, múltiples, 263 fuentes de streaming,
 personalizado, 230-234 estado de streaming, 234
 objeto StreamingContext, 11

StreamingQuery, 223
 StreamingQueryListener, 225
 transmisiones, cargando en tablas de Delta Lake, 277
 Indizador de cadenas, 298, 320
API estructuradas
 sobre, 43
 API de marco de datos, 47-69
 API de conjunto de datos, 69-74
 Conjunto de datos distribuido resistente (RDD), 43
 Spark SQL, 76-82
 estructurando Spark, 44-47
datos estructurados, 49
 Streaming estructurado, 207-264 sobre,
 6, 8, 352
API, 8
 cálculos arbitrarios con estado, 253-261 puntos de
 control, 217 ingenieros de datos y 15 fuentes de
 datos, 226-234 transformaciones de datos, 234-237
 carga de flujos de datos en tablas de Delta Lake,
 277
 procesamiento de flujo de microlotes, 208
 MLflow y, 333 ajuste
 de rendimiento, 262 filosofía de,
 210 modelo de programación de,
 211-213 consultas, 213-225 sumideros,
 226-234 agregaciones de transmisión con
 estado, 238-246 motor de procesamiento
 de transmisión, 207-211 uniones de transmisión,
 246-252

STS (Spark Thrift Server), 119 método
 sum(), 67, 239 aprendizaje automático
 supervisado, 286
 Admite máquinas de vectores, 287
 idiomas admitidos, 357

T

Tableau, consultas con, 122-129 tablas

about, 89
 (ver también tablas SQL)
 creación de bases de datos SQL y, 90
 Delta Lake, 275, 281
 lectura en DataFrames, 93 no
 administrado en comparación con administrado, 89
 pestanas (interfaz de usuario de Spark), 197-205 método
 take(), 73

- tareas, 26, 28
 vistas temporales, 92
 TensorFlow, 332, 336
 conjuntos de datos de prueba, creación, 291-293 paquetes Spark de terceros, 5
 Servidor Thrift JDBC/ODBC, 119
 rendimiento, 330
 Tibshirani, Robert, Los elementos de la estadística Aprendizaje, 309
 tiempos de espera, gestión de grupos interactivos mediante, 257-261 función `to_date()`, 64 función `to_json()`, 138 función `to_timestamp()`, 64 componente de seguimiento (MLflow), 325-330 datos de entrenamiento, 291-293 compatibilidad con transacciones, 265, 271 función `transform()`, 142, 293 transformaciones sobre, 28-30 datos, 279-282
- DataFrames y, 61, 222 estrecho, 30 datos de muestra, 162-167 con estado, 215, 235 sin estado, 215, 235 Streaming estructurado y, 234-237 de ancho, 30 transformadores, 290, 293 modelos basados en árboles, 307-316 detalles de activación, 217 tuning Spark, 173-205 tuplas, 50 API tipificados, 69 objetos tipificados, 69
- EN**
- Uber Ingeniería, 272 UDF (ver funciones definidas por el usuario (UDF)) método `union()`, 147, 357 método `unionAll()`, 357 uniones, 147 transformaciones con estado no administradas, 237 tablas no administradas, 89 API sin tipo, 69 objetos sin tipo, 69 Modo de actualización (Transmisión estructurada), 212, 215, 245 actualizar datos, 280 propiedad `upperBound`, 130 alteración, 271, 281 funciones definidas por el usuario (UDF), 114-117, 239, 336 datos relacionados con el usuario, eliminación, 280 funciones de utilidad, 138 Conjunto de datos de validación **V**, 316 variables, compartidas (ver variables compartidas) Transformador `VectorAssembler`, 293 `VectorIndexer`, 300 lector ORC vectorizado, 106 lector vectorizado, 106 UDF vectorizados (consulte Pandas, funciones definidas por el usuario (UDF)) vistas sobre, 89 creación, 91 metadatos, 93 Retraso de marca de agua **W**, 243 marcas de agua manejo de datos tardíos con, 243 uniones internas con opcional, 248-252 uniones externas con, 252 garantías semánticas con, 245 interfaz de usuario web, 31, 197 Wendell, Patrick, xv método `where()`, 61, 235 generación de código de etapa completa, 119 amplias dependencias, 30 amplias transformaciones (ver particiones aleatorias) función `window()`, 149, 240 transformaciones en ventana, 210 ventanas, 149-151 `withColumn()` método, 64, 152 `withWatermark()`, 244 cargas de trabajo, 265, 269, 271 escritura
- DataFrames en archivos Avro, 105
 DataFrames en archivos CSV, 103
 DataFrames en archivos JSON, 101
 DataFrames a archivos ORC, 108
 DataFrames a archivos Parquet, 99
 DataFrames a tablas Spark SQL, 99 desde bases de datos, 267 a lagos de datos, 269

a archivos, [227](#)
a Kafka, [229](#) a
sistemas de almacenamiento, [230](#)

X

XGBoost, [323](#), [335](#)
Biblioteca XGBoost4J-Spark, [335](#)

Y

Yahoo!, [2](#)
YARN (ver Hadoop YARN)
función year(), [65](#)

Zacarías, Mateo, [xv](#)

Sobre los autores

Jules S. Damji es defensor senior de desarrolladores en Databricks y colaborador de MLflow. Es un desarrollador práctico con más de 20 años de experiencia y ha trabajado como ingeniero de software en empresas líderes como Sun Microsystems, Netscape, @Home, Loudcloud/Opsware, Verisign, ProQuest y Hortonworks, construyendo sistemas distribuidos a gran escala. Tiene un B.Sc. y un M.Sc. en informática y una maestría en defensa política y comunicación de la Universidad Estatal de Oregón, Cal State y la Universidad Johns Hopkins, respectivamente.

Brooke Wenig es líder de prácticas de aprendizaje automático en Databricks. Dirige un equipo de científicos de datos que desarrollan canalizaciones de aprendizaje automático a gran escala para los clientes, además de impartir cursos sobre las mejores prácticas de aprendizaje automático distribuido. Anteriormente, fue consultora principal de ciencia de datos en Databricks. Tiene una maestría en ciencias de la computación de la UCLA con un enfoque en el aprendizaje automático distribuido.

Tathagata Das es ingeniero de software del personal de Databricks, un committer de Apache Spark y miembro del Comité de gestión de proyectos (PMC) de Apache Spark. Es uno de los desarrolladores originales de Apache Spark, el desarrollador líder de Spark Streaming (DStreams), y actualmente es uno de los principales desarrolladores de Structured Streaming y Delta Lake. Tathagata tiene una maestría en informática de UC Berkeley.

Denny Lee es un defensor del personal de desarrollo en Databricks que ha estado trabajando con Apache Spark desde 0.6. Es un ingeniero práctico de sistemas distribuidos y ciencias de datos con amplia experiencia en el desarrollo de infraestructura a escala de Internet, plataformas de datos y sistemas de análisis predictivo para entornos locales y en la nube. También tiene una maestría en informática biomédica de la Universidad de Ciencias y Salud de Oregón y ha diseñado e implementado potentes soluciones de datos para clientes empresariales de atención médica.

Colofón

El animal de la portada de Learning Spark, segunda edición, es el tiburón gato de manchas pequeñas (*Scyliorhinus canicula*), una especie abundante en las aguas poco profundas del mar Mediterráneo y en el Atlántico, frente a las costas de Europa y el norte África. Es un tiburón pequeño y esbelto de cabeza chata, ojos ovalados y hocico redondeado. La superficie dorsal es de color marrón grisáceo y está estampada con muchos pequeños puntos oscuros y, a veces, más claros. Al igual que otros tiburones, la textura de su piel está formada por “dentículos dérmicos”, pequeños “dientes” que crecen en una sola dirección (como escamas de pescado), formando una superficie que es tanto hidrodinámica como resistente a lesiones y parásitos.

Este tiburón que se alimenta de noche crece hasta aproximadamente 3 pies de largo, pesa un promedio de 3 libras en la madurez y en la naturaleza puede vivir hasta 12 años. Se alimenta principalmente de moluscos, crustáceos, cefalópodos y gusanos poliquetos, aunque también come otros peces. Esta especie

exhibe algunos comportamientos sociales, especialmente cuando es joven, y un estudio de 2014 realizado por la Universidad de Exeter encontró que las personas mostraban diferentes personalidades sociales. A pesar de los cambios en el hábitat, algunos tiburones prefirieron permanecer en grupos llamativos, mientras que otros permanecieron solos, camuflados en el fondo del hábitat. Estos comportamientos de socialización también reflejan una variabilidad en las estrategias de seguridad, ya sea a través de números o de camuflaje.

Este tiburón gato es ovíparo (pone huevos) y las hembras depositan de 18 a 20 cajas de huevos pequeños cada año. Estos estuches de caparazón duro tienen zarcillos que se enganchan en las algas en el fondo del océano; cada caja contiene un tiburón joven. Los jóvenes eclosionan después de unos nueve meses.

Debido a que el tiburón gato de manchas pequeñas no es deseable para las pesquerías comerciales, las poblaciones actualmente son estables y la especie está catalogada por la UICN como de Preocupación Menor. Muchos de los animales de las portadas de O'Reilly están en peligro de extinción; todos ellos son importantes para el mundo.

La ilustración de la portada es de Karen Montgomery, basada en un grabado en blanco y negro de Animate Creation (1885) de JG Wood. Las fuentes de la portada son Gilroy Semibold y Guardian Sans. La fuente del texto es Adobe Minion Pro; la fuente del encabezado es Adobe Myriad Condensed; y la fuente del código es Ubuntu Mono de Dalton Maag.