第二章

Spark 簡介

介紹完發展史後，是時候開始使用 Spark 了。本章將簡述 Spark 的核心叢集架構、應用程式並透過 DataFrame 與 SQL 說明結構化 API。過程中也會提及 Spark 核心術語與概念讓你可以正確地使用 Spark。讓我們先從一些基礎背景知識開始談起吧。

Spark 基礎架構

一般來說提及「電腦」時，可能會想到家裡或工作場所中一台聳立在桌面的機器。這種機器在觀賞影片或執行一些試算表軟體時運作良好。然而，許多使用者可能有遇過執行某些任務時，這類電腦運算能力不足的經驗。一個特別明顯的例子便是資料處理領域。單一機器經常沒有足夠的運算能力處理大量的訊息（或使用者可能沒有足夠的時間等待運算執行完畢）。叢集，也就是一群電腦主機，匯聚許多電腦的資源，讓使用者如同操縱單一機器般地運用這些資源。然而，一群堆放聚集的電腦主機沒有什麼作用，仍須有軟體框架協調叢集主機間的任務運行。這便是 Spark 的任務，管理並協調叢集上的任務執行。

Spark 透過叢集管理器管理叢集主機，這類叢集管理器諸如 Spark 獨立叢集管理器、YARN 或 Mesos 等。Spark 應用程式遞交給叢集管理器後，它們會分配資源給應用程式執行後續任務。

Spark 應用程式

Spark 應用程式包含**驅動器**程序以及一組**執行器**程序。驅動器程序會執行程式的 `main()` 函式並運行於叢集的一個節點中，它的任務主要有三項：維護 Spark 應用程式的相關資訊、回應用戶的應用程式或輸入，以及分析、分配與排程任務到執行器（稍後立即會討論此主題）。驅動器是必備的程序，也是 Spark 應用程式的核心，維護應用程式生命週期內所有的相關資訊。

執行器負責實際執行驅動器所分派的工作。這代表每個執行器僅負責兩件任務：執行驅動器分配的程式碼與回報執行狀態給驅動器所在的節點。

圖 2-1 展示了叢集管理器如何控制實體機器以及分配資源給 Spark 應用程式。叢集管理器可以是下列三者：Spark 獨立叢集管理器、YARN 或 Mesos。多個 Spark 應用程式可以在叢集中同時運行，第四篇會討論更多關於叢集管理器的議題。

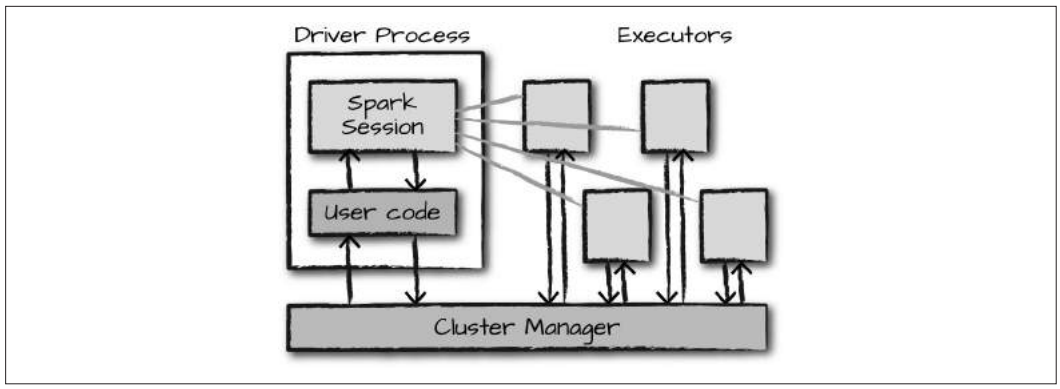

圖 2-1　Spark 應用程式架構

圖 2-1 中左側有一個驅動器而右側有四個執行器。而圖中省略了叢集節點的概念。使用者可以設定每個節點允許運行的執行器數量。

除了叢集模式外，Spark 也提供**本機模式**。驅動器與執行器實質上都是某個程序，而這些程序能運行在不同或相同的機器上。本機模式中，驅動器與執行器會以執行緒的方式運行於相同的主機中。本書撰寫時有考量到 Spark 的本機執行模式，因此本書的任何程式碼皆可在單機上運行。

```
.option("inferSchema", "true")\
.option("header", "true")\
.csv("/data/flight-data/csv/2015-summary.csv")
```

這些 DataFrame（在 Scala 與 Python 中）都有一組欄位與無特定長度的列。沒有指定列數是因為資料讀取是轉換操作，為惰性求值，Spark 僅會先讀取少部分數據作為推論每個欄位型別所用。圖 2-7 展示將 CSV 讀入 DataFrame 再轉換成一般陣列或串列的示意圖。

圖 2-7　讀取 CSV 檔案到 DataFrame，並轉換成一般的陣列或串列物件

若對 DataFrame 執行 take 行動操作，便能如同先前透過指令執行般得到相同結果：

```
flightData2015.take(3)
```

```
Array([United States,Romania,15], [United States,Croatia...
```

接著來執行更多的轉換操作！現在根據某個計數欄位（整數型別）排序資料。圖 2-8 展示了此過程。

注意，sort 並沒有修改 DataFrame。使用 sort 會如同其他轉換操作般回傳新的 DataFrame。對存放結果的 DataFrame 呼叫 take 時所觸發的一系列轉換過程如圖 2-8 所示。

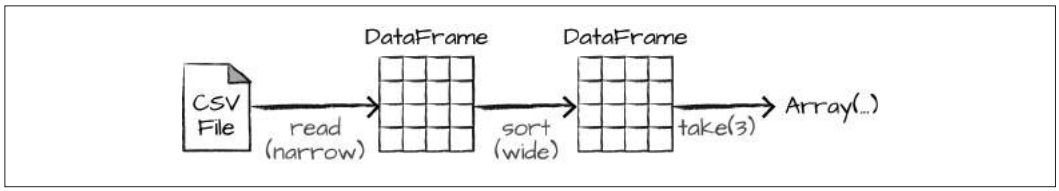

圖 2-8　讀取、排序並收集 DataFrame 資料

DataFrame 語法與 SQL 查詢在語義上非常相似，僅在實做與呼叫順序上有些微差別。如同前述，這兩者底層的執行計畫皆相同。撰寫這些查詢並觀察結果為何：

```
// 在 Scala 中
import org.apache.spark.sql.functions.desc

flightData2015
  .groupBy("DEST_COUNTRY_NAME")
  .sum("count")
  .withColumnRenamed("sum(count)", "destination_total")
  .sort(desc("destination_total"))
  .limit(5)
  .show()
```

```
# 在 Python 中
from pyspark.sql.functions import desc

flightData2015\
  .groupBy("DEST_COUNTRY_NAME")\
  .sum("count")\
  .withColumnRenamed("sum(count)", "destination_total")\
  .sort(desc("destination_total"))\
  .limit(5)\
  .show()
```

```
+-----------------+-----------------+
|DEST_COUNTRY_NAME|destination_total|
+-----------------+-----------------+
|    United States|           411352|
|           Canada|             8399|
|           Mexico|             7140|
|   United Kingdom|             2025|
|            Japan|             1548|
+-----------------+-----------------+
```

透過 explain 函式便能得知 DataFrame 的執行計畫，從原始資料到結果共計七個步驟。執行「程式」的每個對應步驟如圖 2-10 所示。執行計畫（透過 `explain` 取得）與圖 2-10 呈現的順序不同，因為物理執行計畫有經過最佳化。然而，此圖例仍是良好的解說教材。執行計畫是由轉換操作的**有向非循環圖**（directed acyclic graph, DAG）組成，每個步驟的結果都會產生新的 DataFrame 並且無法修改，執行計畫最後會呼叫一個行動操作產生結果。

值得特別注意的是，隨著 Spark SQL 不斷發展，型別可能會隨著時間改變，因此未來更新時可以參考 Spark 的文件（*http://bit.ly/2EdflXW*）。當然，這些型別都相當好用，但你幾乎不太可能僅接觸純靜態的 DataFrame。通常都會操控與轉換這些格式。因此，說明結構化 API 的執行過程相當重要。

結構化 API 執行概覽

本節將展示在叢集中這些程式將如何被執行。這有助於理解（或可能的除錯任務）在叢集上撰寫與執行程式的流程，因此我們從用戶端程式碼開始，逐步理解一個結構化 API 查詢的執行過程。以下是步驟概述：

1. 編寫 DataFrame/Dataset/SQL 程式碼。
2. 若是有效的程式碼，Spark 會將其轉換為**邏輯計劃**。
3. 接著 Spark 將**邏輯計劃**轉換為**物理計劃**，並在過程中進行優化。
4. 然後 Spark 在叢集中執行此物理計劃（RDD 操作）。

執行前必須先寫好程式，接著透過終端機或已經提交的作業將其提交給 Spark。接著程式碼會被傳送至 Catalyst Optimizer 在此將決定程式應該如何執行並製定計劃，最後運行程式並將結果返回給用戶，過程如圖 4-1 所示。

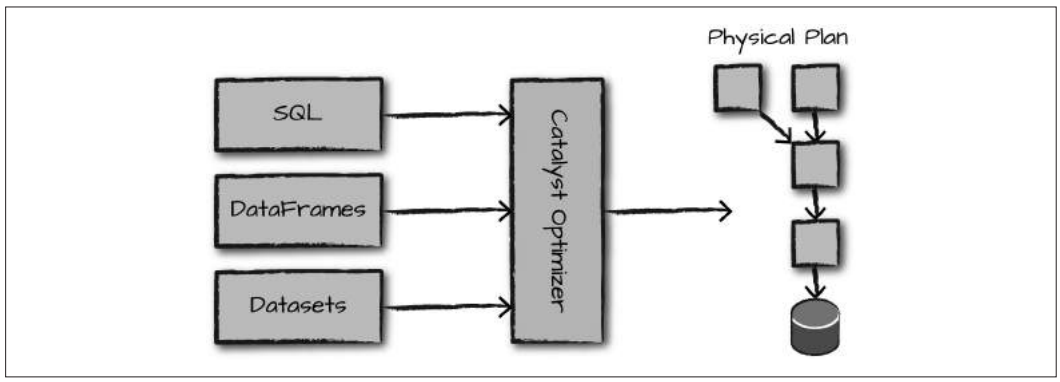

圖 4-1　Catalyst 優化器

邏輯計畫

第一階段代表獲取用戶程式碼並將其轉換為邏輯計劃。圖 4-2 說明了此過程。

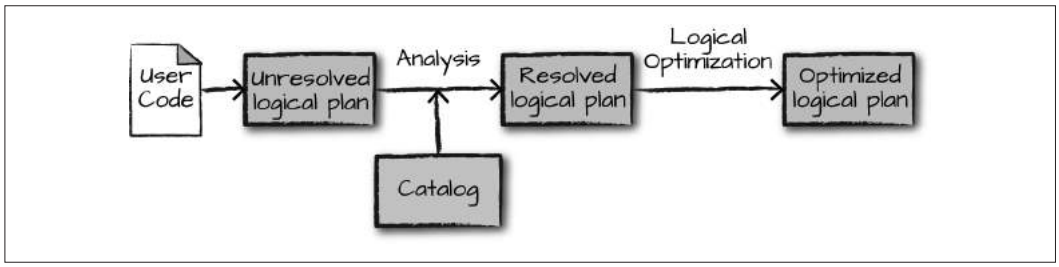

圖 4-2　結構化 API 邏輯計畫流程圖

邏輯計劃僅代表一組不參考執行器或驅動器的抽象轉換，純粹將用戶的表達式轉換為最佳化的版本。為此 Spark 會將用戶程式碼轉換為**尚未解析**的邏輯計劃。因為計畫尚未解析，儘管程式碼可能是正確的，但程式碼引用的表格或欄位仍有可能不存在。Spark 使用 *Catalog*（所有表格與 DataFrame 資訊的存儲庫）**解析分析器**中的欄位與表格。如果 Catalog 中沒有對應的表格或欄位，分析器可能會拒絕該邏輯計劃。若分析器解析成功，則將結果回傳給 Catalyst Optimizer 進行優化，Catalyst Optimizer 是一組規則，並試圖透過斷言下推或選擇器來優化邏輯計劃。可以擴展 Catalyst 套件涵蓋特定領域的優化規則。

物理計畫

成功建立優化的邏輯計畫之後，Spark 會開始執行**物理計畫**。物理計劃（通常稱為 Spark 計劃）會產生不同的物理執行策略，並透過成本模型選出邏輯計劃在叢集上的執行方式（如圖 4-3 所示）。一個成本比較的例子，選擇如何執行關聯時，可能會檢視特定表格的物理屬性（表格或分區的大小）來決定。

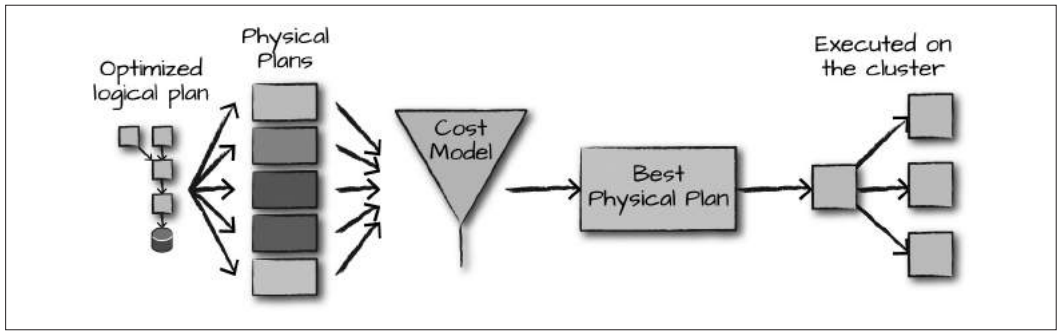

圖 4-3　物理計畫流程圖

物理計畫會產生一系列的 RDD 和轉換操作。這就是為什麼你可能聽過 Spark 被視作一種編譯器，它會處理 DataFrames、Datasets 和 SQL 中的查詢並將他們編譯成 RDD 轉換操作。

執行

決定物理計劃之後，Spark 會透過 RDD 來執行所有這些程式碼，RDD 為 Spark 的低階程式介面（第三篇將會介紹）。Spark 在執行期會進一步的優化，生成原生的 Java bytecode，也可以在執行期刪除整個任務或階段。最後將結果返回用戶端。

結論

本章介紹了 Spark 的結構化 API 以及 Spark 如何將程式碼轉換為叢集上實際運行的程式。後續章節將介紹核心概念以及如何使用結構化 API 的關鍵功能。

另一個常見的任務是計算欄位或欄位集合的概述統計量。可以使用 describe 方法來實現。這將運用所有數值型欄位來計算數量、平均值、標準偏差、最小值和最大值。使用這些函式時主要是在終端機中查看結果，因為綱要可能在未來會做修改：

```
// 在 Scala 中
df.describe().show()

# 在 Python 中
df.describe().show()
```

```
+-------+------------------+------------------+------------------+
|summary|          Quantity|         UnitPrice|        CustomerID|
+-------+------------------+------------------+------------------+
|  count|              3108|              3108|              1968|
|   mean| 8.627413127413128| 4.151946589446603|15661.388719512195|
| stddev|26.371821677029203|15.638659854603892|1854.4496996893627|
|    min|               -24|               0.0|           12431.0|
|    max|               600|            607.49|           18229.0|
+-------+------------------+------------------+------------------+
```

如果需要這些計算後的數值結果，也可以透過導入函式，並將它們應用於需要欄位上執行這些操作並聚合：

```
// 在 Scala 中
import org.apache.spark.sql.functions.{count, mean, stddev_pop, min, max}

# 在 Python 中
from pyspark.sql.functions import count, mean, stddev_pop, min, max
```

StatFunctions 套件中提供了許多統計函數（可以使用下面程式碼中看到的 stat 來進行存取）。這些都是屬於 DataFrame 方法，可以運用它們來計算各種不同的東西。例如，approxQuantile 方法可用來計算資料的精確或近似位數：

```
// 在 Scala 中
val colName = "UnitPrice"
val quantileProbs = Array(0.5)
val relError = 0.05
df.stat.approxQuantile("UnitPrice", quantileProbs, relError) // 2.51

# 在 Python 中
colName = "UnitPrice"
quantileProbs = [0.5]
relError = 0.05
df.stat.approxQuantile("UnitPrice", quantileProbs, relError) # 2.51
```

現在已經創建了函式並對它進行了測試。接著需要在 Spark 中註冊它們，以便可以在所有工作節點上使用。Spark 會將驅動器上的函式序列化，並透過網路將其傳輸到所有執行器的程序，不管是哪種程式語言皆可。

使用此函式時，基本上會發生兩種不同的情況。如果函式是用 Scala 或 Java 編寫的，可以直接在 Java 虛擬機（JVM）中執行。這代表除了不能利用 Spark 為內建函式提供的程式碼來生成函式之外，不會有其他的性能消耗。但是如果建立或使用大量的物件，可能會出現性能的問題；將在第 19 章的優化部分做詳細的介紹。

如果函式是用 Python 編寫的，會發生一些完全不同的情境。Spark 會在 worker 上啟動一個 Python 程序，將所有資料序列化成 Python 可以理解的格式（記住，它一開始是在 JVM 中），接著在 Python 程序中依照每列來執行此函式，最後回傳 JVM 和 Spark 對列的操作結果。圖 6-2 提供了整個過程的概述。

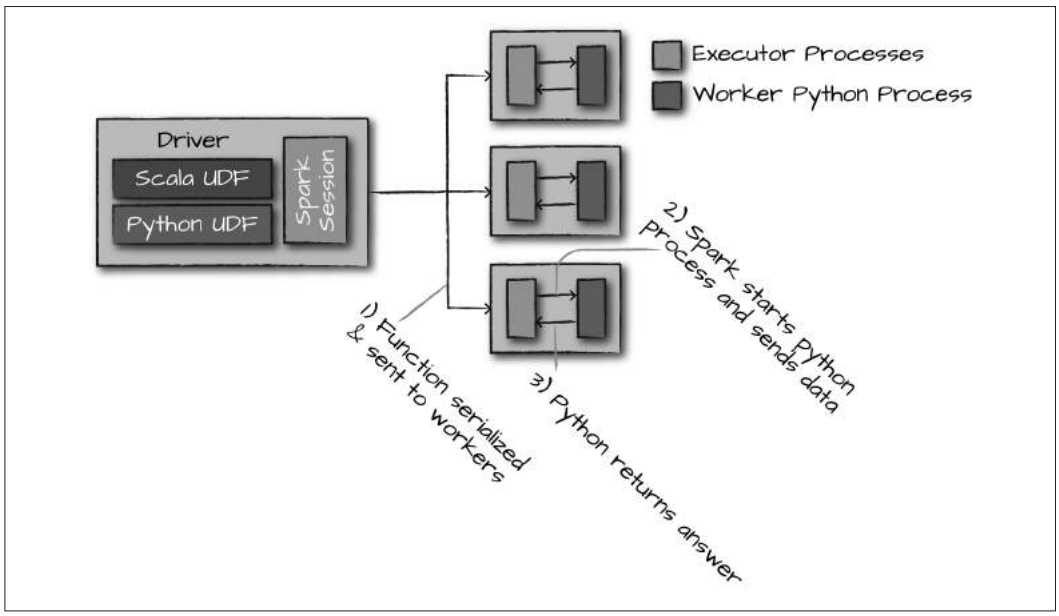

圖 6-2　圖片說明

大表格與小表格

當表格小到符合單個工作節點的記憶體時，可以試著優化關聯的方式。雖然可以使用大表格與大表格的通訊策略，但這裡使用廣播關聯的效果會更好，也就是將小的 DataFrame 複製到群集中的每個工作節點上（無論是一台機器還是多台機器）。雖然這聽起來很消耗資源，但是這種方式可以避免整個關聯過程中執行所有節點間的通訊。相反的，只會在開始時執行一次，並讓每個工作節點單獨執行工作，不必等待或與任何其他工作節點通訊，如圖 8-2 所示。

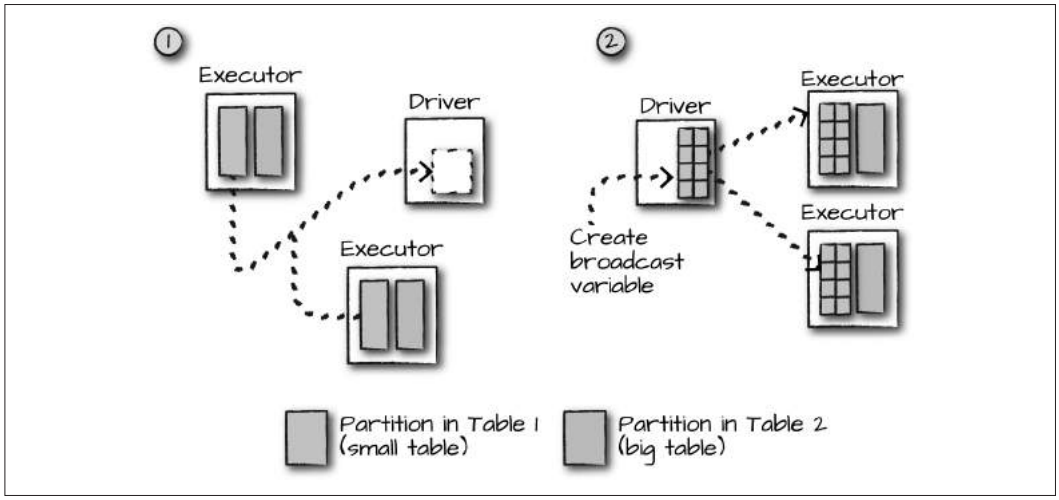

圖 8-2　廣播關聯

此種關聯方式一開始是一個大型通訊，就像以前的關聯類型中一樣。但是在第一次之後，節點之間將不再進一步做通訊。

這代表只會在每個節點上單獨執行關聯，反而使 CPU 成為最大的瓶頸。對於當前的資料集而言，可以看到 Spark 透過執行計劃自動將其設定成廣播關聯：

```
val joinExpr = person.col("graduate_program") === graduateProgram.col("id")

person.join(graduateProgram, joinExpr).explain()
```

```
== Physical Plan ==
*BroadcastHashJoin [graduate_program#40], [id#5....
:- LocalTableScan [id#38, name#39, graduate_progr...
+- BroadcastExchange HashedRelationBroadcastMode(....
   +- LocalTableScan [id#56, degree#57, departmen....
```

Read/write	Key	Potential values	Default	Description
Both	timestampFormat	任何符合 Java 的 SimpleDateFormat 的字串或字元	yyyy-MM-dd'T'HH:mm:ss.SSSZZ	指定任意時間戳型別的欄位格式
Read	primitiveAsString	true, false	false	將所有值自動推斷為字串型別
Read	allowComments	true, false	false	忽略 JSON 記錄中 Java/C++ 樣式的註解
Read	allowUnquotedFieldNames	true, false	false	允許不帶有引號的 JSON 欄位名稱
Read	allowSingleQuotes	true, false	true	允許除了雙引號外，也可使用單引號
Read	allowNumericLeadingZeros	true, false	false	允許數字前綴為零（例如，00012）
Read	allowBackslashEscapingAnyCharacter	true, false	false	允許使用反斜線來跳脫某些字元
Read	columnNameOfCorruptRecord	Any string	spark.sql.column&NameOfCorruptRecord 的值	若欄位含有因為 permissive 模式所導致的字串錯誤，允許進行重新命名，將會覆蓋原來設定的值
Read	multiLine	true, false	false	允許讀取不是用行分隔的 JSON 檔案

讀取行分隔的 JSON 檔案僅在設定 format 和 option 中有所不同：

```
spark.read.format("json")
```

讀取 JSON 檔案

接著來看一個讀取 JSON 檔案並比較不同參數的範例：

```
// 在 Scala 中
spark.read.format("json").option("mode", "FAILFAST").schema(myManualSchema)
  .load("/data/flight-data/json/2010-summary.json").show(5)

# 在 Python 中
spark.read.format("json").option("mode", "FAILFAST")\
  .option("inferSchema", "true")\
  .load("/data/flight-data/json/2010-summary.json").show(5)
```

寫入 JSON 檔案

寫入 JSON 檔案就像讀取一樣簡單，並且與資料源沒有相關性，可以使用之前創建的 CSV DataFrame 作為 JSON 檔案來源。這也遵循著之前的規則：每個分區將寫出一個檔案，整個 DataFrame 將輸出成一個資料夾。檔案裡面每行都是一個 JSON 物件：

```
// 在 Scala 中
csvFile.write.format("json").mode("overwrite").save("/tmp/my-json-file.json")
```

```
# 在 Python 中
csvFile.write.format("json").mode("overwrite").save("/tmp/my-json-file.json")
```

```
$ ls /tmp/my-json-file.json/
```

```
/tmp/my-json-file.json/part-00000-tid-543....json
```

Parquet 檔案

Parquet 是一個開源並以欄位為存儲導向的格式，提供了多種存儲的優化，適合使用於資料分析。它運用了欄位的壓縮方式，可以節省存儲空間，也可以讀取單個欄位而不用讀取整個檔案。它也是 Apache Spark 預設的檔案格式。建議可以將資料寫入 Parquet 進行長期存儲，因為從 Parquet 檔案中讀取資料會比從 JSON 或 CSV 更有效率。Parquet 的另一個優點是它支援複合型別。這代表如果欄位是一個陣列（在 CSV 檔案中會失敗）、映射或結構，仍然可以快速的進行讀取或寫入。以下是如何將 Parquet 設定為讀取格式的範例：

```
spark.read.format("parquet")
```

讀取 Parquet 檔案

Parquet 可以設定的參數很少，因為它在存儲資料時強制執行自己的綱要。因此，只需要設定 format。也可以透過強制設定綱要來限制 DataFrame 輸出，但通常不需要做此設定，因為在讀取時會自動使用**綱要**來做設定，類似使用 CSV 檔案的自動推斷模式。但是，對於 Parquet 檔案，此自動推斷的方法更強大，因為綱要就內建於檔案之中（因此不需要推斷）。

以下是一些從 parquet 讀取資料的例子：