MINING OF MASSIVE DATASETS

NGUYEN HAI LONG

TON DUC THANG UNIVERSITY

Mục lục

Chương II: Tìm hiểu về Spark properties, Spark RDD và Spark DataFrame

1. Spark properties

1.1 Giới thiệu

Là phương tiện điều chỉnh môi trường thực thi cho các ứng dụng Spark. Spark properties điều khiển hầu hết các cài đặt ứng dụng và được cấu hình riêng cho từng ứng dụng. Các thuộc tính này có thể được đặt trực tiếp trên SparkConf được chuyển đến SparkContext của bạn. SparkConf cho phép bạn định cấu hình một số thuộc tính phổ biến (ví dụ: URL chính và tên ứng dụng), cũng như các cặp khóa-giá trị tùy ý thông qua phương thức set().

Ví dụ: chúng ta có thể khởi tạo một ứng dụng với hai luồng như sau:[1]

```
val conf = new SparkConf()
        .setMaster("local[2]")
        .setAppName("CountingSheep")
val sc = new SparkContext(conf)
```

Khi chúng ta chạy với local[2], nghĩa là hai luồng - đại diện cho sự song song "tối thiểu", có thể giúp phát hiện lỗi chỉ tồn tại khi chúng ta chạy trong ngữ cảnh phân tán. Ngoài ra chúng ta có thể có nhiều hơn 1 luồng ở chế độ cục bộ và trong những trường hợp như Spark Streaming, chúng ta thực sự có thể yêu cầu nhiều hơn 1 luồng để ngăn chặn bất kỳ loại vấn đề chết đói nào.[1]

Spark properties chủ yếu có thể được chia thành hai loại: một là liên quan đến triển khai, như "spark.driver.memory", "spark.executor.instances", loại thuộc tính này có thể không bị ảnh hưởng khi thiết lập lập trình thông qua SparkConf trong thời gian chạy, hoặc hành vi phụ thuộc vào trình quản lý cụm và chế độ triển khai bạn chọn, vì vậy bạn nên đặt thông qua tệp cấu hình hoặc tùy chọn dòng lệnh spark-submit; một cái khác chủ yếu liên quan đến kiểm soát thời gian chạy Spark, như "spark.task.maxFailures", loại thuộc tính này có thể được đặt theo một trong hai cách.[1]

1.2 Tải động các thuộc tính Spark (Dynamically Loading Spark Properties)

Trong một số trường hợp, bạn có thể muốn tránh mã hóa cứng các cấu hình nhất định trong a SparkConf. Ví dụ: nếu bạn muốn chạy cùng một ứng dụng với các bản gốc khác nhau hoặc số lượng bộ nhớ khác nhau. Spark cho phép bạn chỉ cần tạo một conf trống:

```
val sc = new SparkContext(new SparkConf())
```

Mọi giá trị được chỉ định dưới dạng cờ hoặc trong tệp thuộc tính sẽ được chuyển đến ứng dụng và được hợp nhất với những giá trị được chỉ định thông qua SparkConf. Các thuộc tính được đặt trực tiếp trên SparkConf được ưu tiên cao nhất, sau đó các cờ được chuyển đến spark-submithoặc spark-shell, sau đó là các tùy chọn trong tệp spark-defaults.conf. Một vài khóa cấu hình đã được đổi tên kể từ các phiên bản Spark trước đó; trong những trường hợp như vậy, các tên khóa cũ hơn vẫn được chấp nhận, nhưng được ưu tiên thấp hơn bất kỳ trường hợp nào của khóa mới hơn.

Các thuộc tính của Spark chủ yếu có thể được chia thành hai loại: một là liên quan đến triển khai, như “spark.driver.memory”, “spark.executor.instances”, loại thuộc tính này có thể không bị ảnh hưởng khi thiết lập theo chương trình SparkConftrong thời gian chạy, hoặc hành vi phụ thuộc vào trình quản lý cụm và chế độ triển khai bạn chọn, vì vậy bạn nên đặt thông qua tệp cấu hình hoặc spark-submittùy chọn dòng lệnh; một loại khác chủ yếu liên quan đến kiểm soát thời gian chạy Spark, như “spark.task.maxFailures”, loại thuộc tính này có thể được đặt theo một trong hai cách.

1.3 Xem thuộc tính Spark (Viewing Spark Properties)

Giao diện người dùng web ứng dụng tại http://<driver>:4040 liệt kê các thuộc tính Spark trong tab "Môi trường". Đây là một nơi hữu ích để kiểm tra để đảm bảo rằng các thuộc tính của bạn đã được đặt chính xác. Lưu ý rằng chỉ có giá trị xác định một cách rõ ràng thông qua spark-defaults.conf, SparkConf hoặc dòng lệnh sẽ xuất hiện. Đối với tất cả các thuộc tính cấu hình khác, bạn có thể giả sử giá trị mặc định được sử dụng.

1.4 Thuộc tính có sẵn (Available Properties)

Hầu hết các thuộc tính kiểm soát cài đặt nội bộ đều có giá trị mặc định hợp lý. Một số tùy chọn phổ biến nhất để đặt là:

1.4.1 Thuộc tính ứng dụng (Application Properties)

Đây là một số thuộc tính ứng dụng:

Property Name	Meaning
spark.app.name	Tên ứng dụng của bạn. Điều này sẽ xuất hiện trong giao diện người dùng và trong dữ liệu nhật ký.
spark.driver.cores	Số lõi để sử dụng cho quy trình trình điều khiển, chỉ ở chế độ cụm.
spark.driver.maxResultSize	Giới hạn tổng kích thước của các kết quả được tuần tự hóa của tất cả các phân vùng cho mỗi hành động Spark (ví dụ: thu thập) tính bằng byte. Tối thiểu phải là 1M hoặc 0 cho không giới hạn. Công việc sẽ bị hủy bỏ nếu tổng kích thước vượt quá giới hạn này. Có giới hạn cao có thể gây ra lỗi hết bộ nhớ trong trình điều khiển (phụ thuộc vào spark.driver.memory và chi phí bộ nhớ của các đối tượng trong JVM). Đặt giới hạn thích hợp có thể bảo vệ trình điều khiển khỏi lỗi hết bộ nhớ.
spark.driver.memory	Dung lượng bộ nhớ sử dụng cho quá trình lái xe, tức là nơi SparkContext được khởi tạo, trong định dạng giống như chuỗi ký ức JVM với một đơn vị kích thước hậu tố ("k", "m", "g" hoặc "t") (ví dụ 512m, 2g) .
spark.driver.memoryOverhead	Số lượng bộ nhớ không phải heap sẽ được cấp phát cho mỗi quá trình trình điều khiển ở chế độ cụm, trong MiB trừ khi được chỉ định khác. Đây là bộ nhớ chiếm những thứ như tổng chi phí VM, chuỗi được thực hiện, các chi phí chung khác, v.v. Điều này có xu hướng phát triển theo kích thước vùng chứa (thường là 6-10%). Tùy chọn này hiện được hỗ trợ trên YARN, Mesos và Kubernetes.

1.4.2 Môi trường thực thi (Runtime Environment)

Đây là một số môi trường thực thi:

Property Name	Meaning
spark.driver.extraClassPath	Các mục nhập classpath bổ sung để thêm trước vào classpath của trình điều khiển.
spark.driver.defaultJavaOptions	Một chuỗi các tùy chọn JVM mặc định để thêm vào spark.driver.extraJavaOptions. Điều này được thiết lập bởi các quản trị viên. Ví dụ: cài đặt GC hoặc ghi nhật ký khác. Lưu ý rằng việc đặt cài đặt kích thước đống tối đa (-Xmx) với tùy chọn này là bất hợp pháp. Có thể đặt cài đặt kích thước đống tối đa spark.driver.memorytrong chế độ cụm và thông qua --driver-memorytùy chọn dòng lệnh trong chế độ khách.
spark.driver.extraJavaOptions	Giới hạn tổng kích thước của các kết quả được tuần tự hóa của tất cả các phân vùng cho mỗi hành động Spark (ví dụ: thu thập) tính bằng byte. Tối thiểu phải là 1M hoặc 0 cho không giới hạn. Công việc sẽ bị hủy bỏ nếu tổng kích thước vượt quá giới hạn này. Có giới hạn cao có thể gây ra lỗi hết bộ nhớ trong trình điều khiển (phụ thuộc vào spark.driver.memory và chi phí bộ nhớ của các đối tượng trong JVM). Đặt giới hạn thích hợp có thể bảo vệ trình điều khiển khỏi lỗi hết bộ nhớ.
spark.driver.extraLibraryPath	Đặt một đường dẫn thư viện đặc biệt để sử dụng khi khởi chạy trình điều khiển JVM.
spark.driver.userClassPathFirst	(Thử nghiệm) Có ưu tiên các lọ do người dùng thêm vào các lọ của chính Spark khi tải các lớp trong trình điều khiển hay không. Tính năng này có thể được sử dụng để giảm thiểu xung đột giữa phụ thuộc của Spark và phụ thuộc của người dùng. Nó hiện là một tính năng thử nghiệm. Điều này chỉ được sử dụng trong chế độ cụm.

1.4.3 Hành vi xáo trộn (Shuffle Behavior)

Đây là một số hành vi xáo trộn:

Property Name	Meaning
spark.reducer.maxSizeInFlight	Kích thước tối đa của đầu ra bản đồ để tìm nạp đồng thời từ mỗi tác vụ giảm, trong MiB trừ khi được chỉ định khác. Vì mỗi đầu ra yêu cầu chúng ta tạo một bộ đệm để nhận nó, điều này đại diện cho chi phí bộ nhớ cố định cho mỗi tác vụ giảm, vì vậy hãy giữ nó nhỏ trừ khi bạn có một lượng lớn bộ nhớ.
spark.reducer.maxReqsInFlight	Cấu hình này giới hạn số lượng yêu cầu từ xa để tìm nạp các khối tại bất kỳ điểm nhất định nào. Khi số lượng máy chủ trong cụm tăng lên, nó có thể dẫn đến số lượng rất lớn các kết nối gửi đến một hoặc nhiều nút, khiến các công nhân bị lỗi khi tải. Bằng cách cho phép nó giới hạn số lượng yêu cầu tìm nạp, tình huống này có thể được giảm thiểu.
spark.reducer.maxBlocksInFlightPerAddress	Cấu hình này giới hạn số lượng khối từ xa được tìm nạp cho mỗi tác vụ giảm từ một cổng máy chủ nhất định. Khi một số lượng lớn các khối đang được yêu cầu từ một địa chỉ nhất định trong một lần tìm nạp hoặc đồng thời, điều này có thể làm hỏng trình thực thi phục vụ hoặc Trình quản lý nút. Điều này đặc biệt hữu ích để giảm tải trên Node Manager khi bật chế độ trộn bên ngoài. Bạn có thể giảm thiểu vấn đề này bằng cách đặt nó thành một giá trị thấp hơn.
spark.shuffle.compress	Có nén các tệp đầu ra bản đồ hay không. Nói chung là một ý kiến hay. Nén sẽ sử dụng spark.io.compression.codec.
spark.shuffle.file.buffer	Kích thước của bộ đệm trong bộ nhớ cho mỗi luồng đầu ra tệp trộn, trong KiB trừ khi được chỉ định khác. Các bộ đệm này làm giảm số lần tìm đĩa và các lệnh gọi hệ

	thống được thực hiện trong việc tạo các tệp ngẫu nhiên trung gian.

1.4.4 Giao diện người dùng Spark (Spark IU)

Đây là một số giao diện người dùng Spark:

Property Name	Meaning
spark.eventLog.logBlockUpdates.enabled	Có ghi lại các sự kiện cho mỗi lần cập nhật khối hay không, nếu spark.eventLog.enabledđúng. * Cảnh báo *: Điều này sẽ làm tăng đáng kể kích thước của nhật ký sự kiện.
spark.eventLog.longForm.enabled	Nếu đúng, hãy sử dụng biểu mẫu dài của các trang web cuộc gọi trong nhật ký sự kiện. Nếu không, hãy sử dụng mẫu ngắn.
spark.eventLog.compress	Có nén các sự kiện đã ghi, nếu spark.eventLog.enabledđúng.
spark.eventLog.compression.codec	Codec để nén các sự kiện đã ghi. Nếu điều này không được đưa ra, spark.io.compression.codecsẽ được sử dụng.
spark.eventLog.erasureCoding.enabled	Cho phép nhật ký sự kiện sử dụng mã hóa xóa hay tắt mã hóa xóa, bất kể giá trị mặc định của hệ thống tệp. Trên HDFS, các tệp được mã hóa xóa sẽ không cập nhật nhanh như các tệp sao chép thông thường, do đó, các bản cập nhật ứng dụng sẽ mất nhiều thời gian hơn để xuất hiện trong Máy chủ Lịch sử.

Ngoài ra còn rất nhiều thuộc tính khác bạn có thể tham khảo tại đây: https://spark.apache.org/docs/latest/configuration.html

2. Spark RDD

2.1 Giới thiệu

Resilient Distributed Datasets - Tập dữ liệu phân tán có khả năng phục hồi (RDD) là một cấu trúc dữ liệu cơ bản của Spark. Nó là một tập hợp các đối tượng được phân phối bất biến. Mỗi tập dữ liệu trong RDD được chia thành các phân vùng logic, có thể được tính toán trên các nút khác nhau của cụm. RDD có thể chứa bất kỳ loại đối tượng Python, Java hoặc Scala nào, bao gồm các lớp do người dùng định nghĩa. Về mặt hình thức, RDD là một tập hợp các bản ghi được phân vùng, chỉ đọc.[3]

Về mặt hình thức, RDD là một tập hợp các bản ghi được phân vùng, chỉ đọc. RDD có thể được tạo thông qua các hoạt động xác định trên dữ liệu trên bộ lưu trữ ổn định hoặc các RDD khác. RDD là một tập hợp các phần tử có khả năng chịu được lỗi và có thể hoạt động song song.[3]

Có hai cách để tạo RDD - parallelizing hiện có trong chương trình trình điều khiển của bạn hoặc referencing a dataset trong hệ thống lưu trữ bên ngoài, chẳng hạn như hệ thống tệp chia sẻ, HDFS, HBase hoặc bất kỳ nguồn dữ liệu nào cung cấp Hadoop InputFormat.[3]

2.2 Một số tính năng của Spark RDD

Spark RDD có một số tính năng như:

2.2.1 Tính toán trong bộ nhớ (In-memory Computation)

Spark RDD có cung cấp khả năng tính toán trong bộ nhớ . Nó lưu trữ các kết quả trung gian trong bộ nhớ phân tán (RAM) thay vì lưu trữ ổn định (ổ đĩa).[4]

2.2.2 Đánh giá lười biếng (Lazy Evaluations)

Tất cả các phép biến đổi trong Apache Spark đều lười biếng ở chỗ nó không tính toán ngay kết quả của nó. Thay vào đó, nó chỉ nhớ các phép biến đổi được áp dụng cho một số tập dữ liệu cơ sở.[4]

2.2.3 Khả năng chịu lỗi (Fault Tolerance)

Spark RDD có khả năng chịu lỗi vì nó theo dõi thông tin dòng dữ liệu để tự động xây dựng lại dữ liệu bị mất khi bị lỗi. Nó xây dựng lại dữ liệu bị mất khi thất bại bằng

cách sử dụng dòng, mỗi RDD nhớ cách nó được tạo từ các bộ dữ liệu khác (bằng các phép biến đổi như bản đồ, tham gia hoặc nhómBy) để tạo lại chính nó.[4]

2.2.4 Bất biến (Immutability)

Dữ liệu an toàn để chia sẻ trên các quy trình. Nó cũng có thể được tạo hoặc truy xuất bất cứ lúc nào giúp dễ dàng lưu vào bộ nhớ đệm, chia sẻ và nhân rộng. Vì vậy, nó là một cách để đạt được sự nhất quán trong tính toán.[4]

2.2.5 Phân vùng (Partitioning)

Phân vùng là đơn vị cơ bản của tính song song trong Spark RDD. Mỗi phân vùng là một phân chia dữ liệu hợp lý có thể thay đổi được. Người ta có thể tạo một phân vùng thông qua một số biến đổi trên các phân vùng hiện có.[4]

2.2.6 Sự bền bỉ (Persistence)

Người dùng có thể nêu những RDD nào họ sẽ sử dụng lại và chọn chiến lược lưu trữ cho chúng (ví dụ: lưu trữ trong bộ nhớ hoặc trên Đĩa).[4]

2.2.7 Hoạt động chi tiết thô (Coarse-grained Operations)

Nó áp dụng cho tất cả các phần tử trong bộ dữ liệu thông qua bản đồ hoặc bộ lọc hoặc nhóm theo hoạt động.[4]

2.2.8 Vị trí – Độ dính (Location-Stickiness)

RDD có khả năng xác định ưu tiên vị trí để tính toán các phân vùng. Tùy chọn vị trí đề cập đến thông tin về vị trí của RDD. Các DAGScheduler đặt phân vùng theo cách như vậy mà nhiệm vụ gần dữ liệu càng nhiều càng tốt. Do đó tăng tốc độ tính toán.[4]

2.3 Một số ví dụ

Spark sử dụng khái niệm RDD để đạt được các hoạt động MapReduce nhanh hơn và hiệu quả hơn. Trước tiên, chúng ta hãy thảo luận về cách các hoạt động MapReduce diễn ra và tại sao chúng không hiệu quả như vậy.

Ví dụ về RDD, hãy xem xét chương trình đơn giản dưới đây:

```
lines = sc.textFile("data.txt")

lineLengths = lines.map(lambda s: len(s))
```

```
totalLength = lineLengths.reduce(lambda a, b: a + b)
```

Dòng đầu tiên xác định một RDD cơ sở từ một tệp bên ngoài. Tập dữ liệu này không được tải trong bộ nhớ hoặc không được tác động trên: các dòng chỉ là một con trỏ đến tệp. Dòng thứ hai xác định lineLengths là kết quả của việc chuyển đổi bản đồ.

Một lần nữa, lineLengths không được tính toán ngay lập tức, do sự lười biếng. Cuối cùng, chúng ta chạy giảm, đó là một hành động. Tại thời điểm này, Spark chia nhỏ tính toán thành các tác vụ để chạy trên các máy riêng biệt và mỗi máy chạy cả phần bản đồ và phần giảm cục bộ, chỉ trả lại câu trả lời cho chương trình điều khiển.

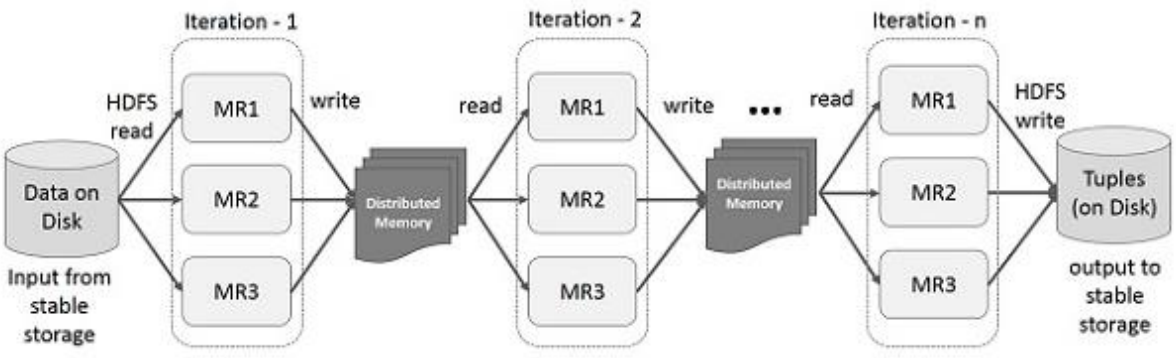
Chia sẻ dữ liệu chạy chậm trong MapReduce do sao chép, tuần tự hóa. Hầu hết các ứng dụng Hadoop dành hơn 90% thời gian để thực hiện các thao tác đọc-ghi HDFS. Do đó các nhà nghiên cứu đã phát triển một framework chuyên biệt có tên là Apache Spark để khắc phục vấn đề này. Ý tưởng chính của Spark là Resilient Distributed Datasets (RDD); nó hỗ trợ tính toán xử lý trong bộ nhớ. Điều này có nghĩa là, nó lưu trữ trạng thái bộ nhớ như một đối tượng trên các công việc và đối tượng có thể chia sẻ giữa các công việc đó. Chia sẻ dữ liệu trong bộ nhớ nhanh hơn mạng và đĩa từ 10 đến 100 lần.

2.4 Hoạt động của Spark RDD

Spark có hai kiểu hoạt động: hoạt động lặp lại (iterative operations) và hoạt động tương tác (interactive operations).

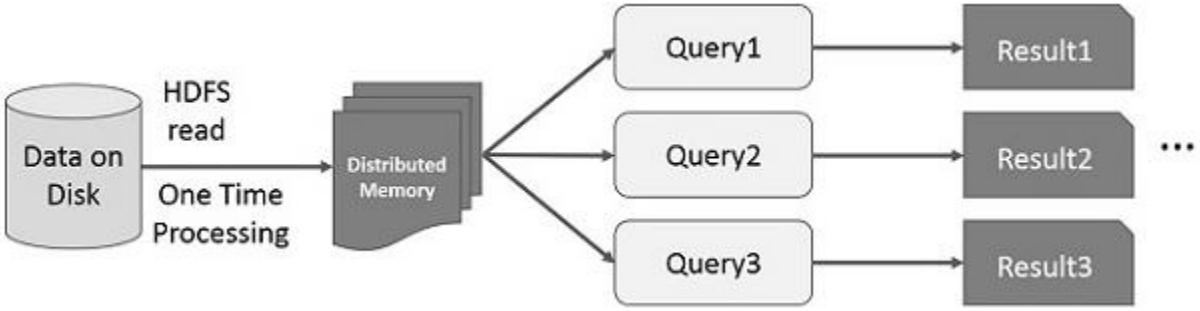
2.4.1 Hoạt động lặp lại (iterative operations)

Hình minh họa dưới đây cho thấy các hoạt động lặp lại trên Spark RDD. Nó sẽ lưu trữ các kết quả trung gian trong một bộ nhớ phân tán thay vì Ổ lưu trữ ổn định (Đĩa) và làm cho hệ thống nhanh hơn. Lưu ý - Nếu bộ nhớ Phân tán (RAM) không đủ để lưu trữ các kết quả trung gian (Trạng thái công việc), thì nó sẽ lưu các kết quả đó trên đĩa.[3]



2.4.2 Hoạt động tương tác (interactive operations)

Hình minh họa dưới đây cho thấy các hoạt động tương tác trên Spark RDD. Nếu các truy vấn khác nhau được chạy lặp lại trên cùng một tập dữ liệu, thì dữ liệu cụ thể này có thể được lưu trong bộ nhớ để có thời gian thực thi tốt hơn.[3]



Theo mặc định, mỗi RDD đã chuyển đổi có thể được tính toán lại mỗi khi bạn chạy một hành động trên đó. Tuy nhiên, bạn cũng có thể duy trì một RDD trong bộ nhớ, trong trường hợp đó Spark sẽ giữ các phần tử xung quanh trên cụm để truy cập nhanh hơn nhiều, vào lần tiếp theo bạn truy vấn nó. Ngoài ra còn có hỗ trợ cho các RDD lâu dài trên đĩa hoặc được sao chép qua nhiều nút.[3]

Ngoài ra RDD trong Apache Spark còn hỗ trợ 2 loại hoạt động sau:

- Chuyển đổi (Transformation)

- Hành động (Actions)

2.4.3 Chuyển đổi (Transformation)

Spark RDD Transformations là các hàm sử dụng một RDD làm đầu vào và tạo ra một hoặc nhiều RDD làm đầu ra. Nó không thay đổi RDD đầu vào (vì RDD là bất biến và do đó người ta không thể thay đổi nó), nhưng luôn tạo ra một hoặc nhiều RDD mới bằng cách áp dụng các tính toán mà nó đại diện, ví dụ như Map(), filter(), ReduceByKey(),…[4]

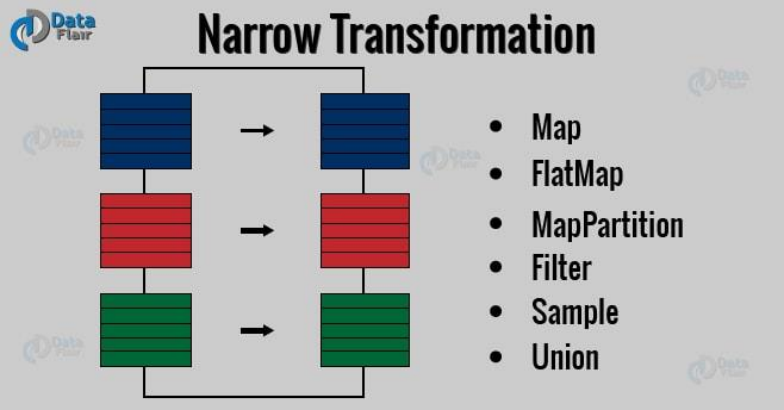
Các phép biến đổi là các hoạt động lười biếng trên RDD trong Apache Spark. Nó tạo ra một hoặc nhiều RDD mới, thực thi khi một Actions xảy ra. Do đó, Transformation tạo ra một tập dữ liệu mới từ tập dữ liệu hiện có.[4]

Một số biến đổi nhất định có thể được pipelined, đây là một phương pháp tối ưu hóa mà Spark sử dụng để cải thiện hiệu suất của các phép tính. Có hai loại Transformations: biến đổi hẹp (narrow transformation), biến đổi rộng (wide transformation).[4]

Biến đổi hẹp

Nó là kết quả của ánh xạ, bộ lọc và sao cho dữ liệu chỉ từ một phân vùng duy nhất, tức là nó tự cung cấp. Một RDD đầu ra có các phân vùng với các bản ghi bắt nguồn từ một phân vùng duy nhất trong RDD mẹ. Chỉ một tập hợp con giới hạn của các phân vùng được sử dụng để tính toán kết quả.[4]

Spark nhóm các phép biến hình thu hẹp dưới dạng một giai đoạn được gọi là pipelining.



Biến đổi rộng

Nó là kết quả của các hàm like groupByKey() và ReduceByKey(). Dữ liệu cần thiết để tính toán các bản ghi trong một phân vùng duy nhất có thể nằm trong nhiều phân vùng của RDD mẹ. Các biến đổi rộng còn được gọi là biến đổi trộn (shuffle transformation) vì chúng có thể có hoặc không phụ thuộc vào quá trình xáo trộn.[4]

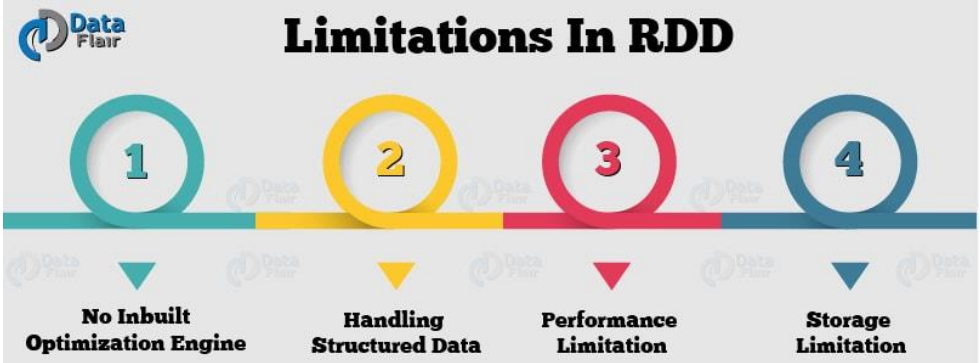
2.4.4 Hành động (Actions)

Một Action trong Spark trả về kết quả cuối cùng của các tính toán RDD. Nó kích hoạt thực thi bằng cách sử dụng đồ thị dòng để tải dữ liệu vào RDD gốc, thực hiện tất cả các phép biến đổi trung gian và trả về kết quả cuối cùng cho chương trình Driver hoặc ghi nó ra hệ thống tệp. Đồ thị tuyến tính là đồ thị phụ thuộc của tất cả các RDD song song của RDD.[4]

Các Action là các hoạt động RDD tạo ra các giá trị không phải RDD. Chúng hiện thực hóa một giá trị trong chương trình Spark. Action là một trong những cách để gửi kết quả từ người thực thi đến trình điều khiển. First(), take(), Reduce(), collect(), count() là một số Action trong spark.[4]

2.5 Hạn chế của Spark RDD

Apache Spark RDD cũng có một số hạn chế như:



2.5.1 Không có công cụ tối ưu hóa sẵn có

Khi làm việc với dữ liệu có cấu trúc, RDD không thể tận dụng lợi thế của các trình tối ưu hóa nâng cao của Spark bao gồm trình tối ưu hóa chất xúc tác và công cụ thực thi Tungsten . Các nhà phát triển cần tối ưu hóa từng RDD dựa trên các thuộc tính của nó.[4]

2.5.2 Xử lý dữ liệu có cấu trúc

Không giống như Dataframe và bộ dữ liệu, RDD không suy ra lược đồ của dữ liệu đã nhập và yêu cầu người dùng chỉ định nó.[4]

2.5.3 Giới hạn hiệu suất

Là đối tượng JVM trong bộ nhớ, RDD liên quan đến chi phí của Bộ sưu tập rác và Tuần tự hóa Java, những thứ này rất tốn kém khi dữ liệu phát triển.[4]

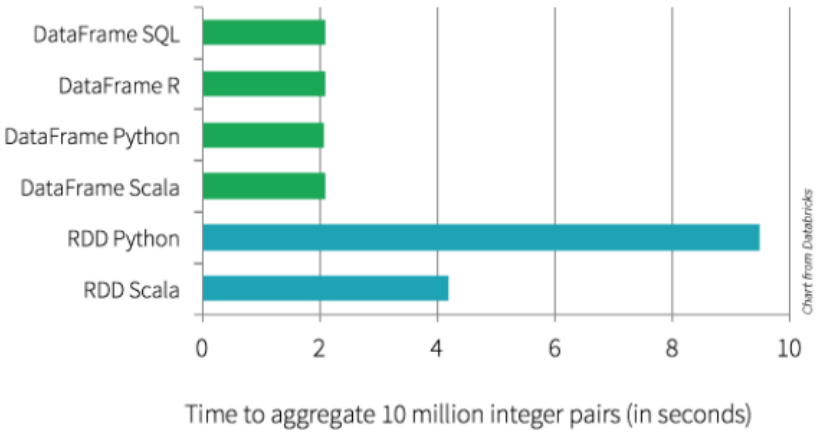
2.5.4 Giới hạn lưu trữ

RDDs suy giảm khi không có đủ bộ nhớ để lưu trữ chúng. Người ta cũng có thể lưu trữ phân vùng RDD đó trên đĩa không vừa với RAM. Do đó, nó sẽ cung cấp hiệu suất tương tự như các hệ thống song song dữ liệu hiện tại.[4]

3. Spark DataFrame

3.1 Giới thiệu

Trong Spark, DataFrame là một tập hợp dữ liệu phân tán được tổ chức thành các cột được đặt tên. Về mặt khái niệm, nó tương đương với một bảng trong cơ sở dữ liệu quan hệ hoặc một khung dữ liệu trong R/Python có kỹ thuật tối ưu hóa tốt hơn. DataFrames có thể được xây dựng từ nhiều nguồn như: tệp dữ liệu có cấu trúc, bảng trong Hive, cơ sở dữ liệu bên ngoài hoặc RDD hiện có. DataFrame là một API cấp cao của RDD được giới thiệu vào năm 2013 (kể từ Apache Spark 1.3) để giúp người dùng dễ dàng thực hiện tác vụ xử lý dữ liệu và cải thiện đáng kể hiệu suất của hệ thống. Tương tự như RDD, DataFrame cũng lưu trữ dữ liệu theo cách phân tán và bất biến, nhưng ở dạng cột, tương tự như Cơ sở dữ liệu quan hệ. API này được thiết kế cho các ứng dụng Khoa học dữ liệu và Dữ liệu lớn hiện đại lấy cảm hứng từ DataFrame trong Lập trình R và Pandas trong Python.[6]



3.2 Một số tính năng của Spark Dataframe

DataFrame có một số tính năng đặc trưng:[5]

- Khả năng xử lý dữ liệu có kích thước từ Kilobyte đến Petabyte trên một cụm nút đơn đến cụm lớn.

- Hỗ trợ các định dạng dữ liệu khác nhau (Avro, csv, tìm kiếm đàn hồi và Cassandra) và hệ thống lưu trữ (HDFS, bảng HIVE, mysql, v.v.).

- Tối ưu hóa hiện đại và tạo mã thông qua trình tối ưu hóa Spark SQL Catalyst (khung chuyển đổi cây).

- Có thể dễ dàng tích hợp với tất cả các công cụ và khuôn khổ Dữ liệu lớn thông qua Spark-Core.

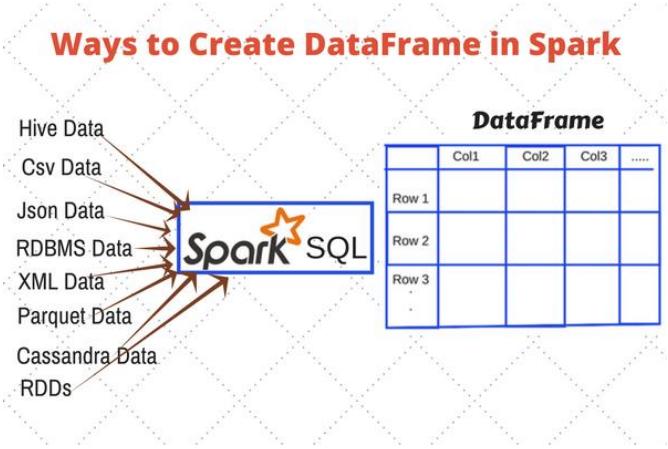
- Cung cấp API cho Lập trình Python, Java, Scala và R.

3.3 Tạo DataFrame

Có rất nhiều cách để chúng ta tạo ra Dataframe:

- Chúng tôi có thể tạo nó bằng cách sử dụng các định dạng dữ liệu khác nhau, chẳng hạn như tải dữ liệu từ JSON, CSV.

- Nó cũng có thể bằng cách tải dữ liệu từ RDD hiện có.



Ví dụ về cách tạo DataFrames trong Python:

```
# Tạo DataFrame từ bảng người dùng trong Hive.

users = context.table("users")

# Từ các tệp JSON trong S3

logs = context.load("s3n://path/to/data.json", "json")
```

3.4 Một số ví dụ

Sau khi được xây dựng, DataFrames cung cấp một ngôn ngữ dành riêng cho miền để thao tác dữ liệu phân tán. Dưới đây là một ví dụ về việc sử dụng DataFrames để xử lý dữ liệu có cấu trúc của một lượng lớn người dùng:

3.4.1 Đọc tài liệu JSON:

Đầu tiên, chúng ta phải đọc tài liệu JSON. Dựa trên điều này, tạo một DataFrame có tên (df). Sử dụng lệnh sau để đọc tài liệu JSON có tên là worker.json. Dữ liệu được hiển thị dưới dạng bảng với các trường - id, tên và tuổi.[5]

```
val df = sqlContext.read.json("worker.json")

# Để xem dữ liệu trong DataFrame, hãy sử dụng lệnh sau.

df.show()
```

File JSON:

worker.json

```
[
   {"id" : "1201", "name" : "satish", "age" : "25"},
   {"id" : "1202", "name" : "krishna", "age" : "28"},
   {"id" : "1203", "name" : "amith", "age" : "39"},
   {"id" : "1204", "name" : "javed", "age" : "23"},
   {"id" : "1205", "name" : "prudvi", "age" : "23"}
]
```

Output:

```
from pyspark.sql import SQLContext
sqlContext = SQLContext(sc)
from pyspark  import SparkConf, SparkContext

my_JSON = (sc.textFile('/content/sample_data/worker.json'))
df = sqlContext.read.json(my_JSON)
df.show()
```

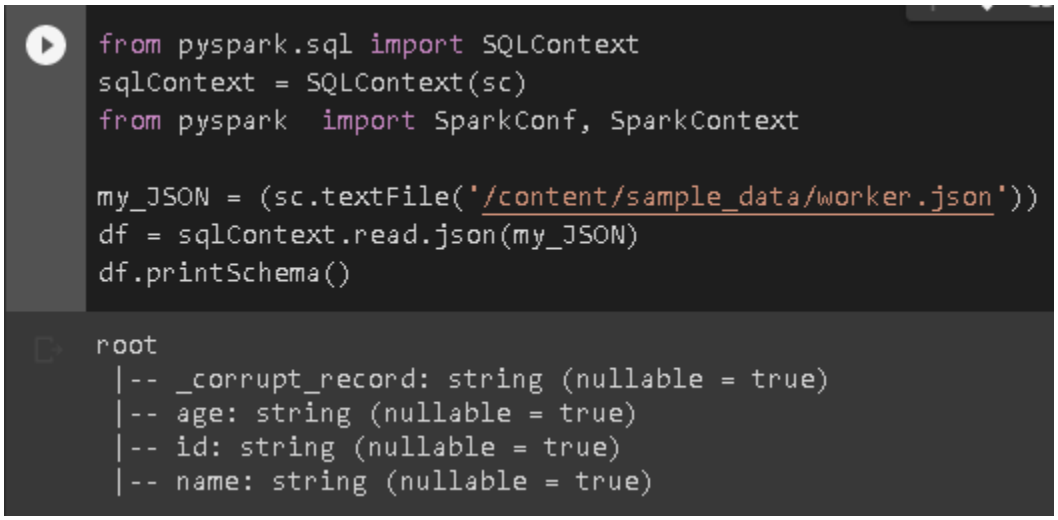
```
+---------------+----+----+-------+
|_corrupt_record| age|  id|   name|
+---------------+----+----+-------+
|              [|null|null|   null|
|           null|  25|1201| satish|
|           null|  28|1202|krishna|
|           null|  39|1203|  amith|
|           null|  23|1204|  javed|
|           null|  23|1205| prudvi|
|              ]|null|null|   null|
+---------------+----+----+-------+
```

3.4.2 Sử dụng phương pháp printSchema

Nếu bạn muốn xem Cấu trúc (Lược đồ) của DataFrame, hãy sử dụng lệnh sau:[5]

```
df.printSchema()
```

Output:

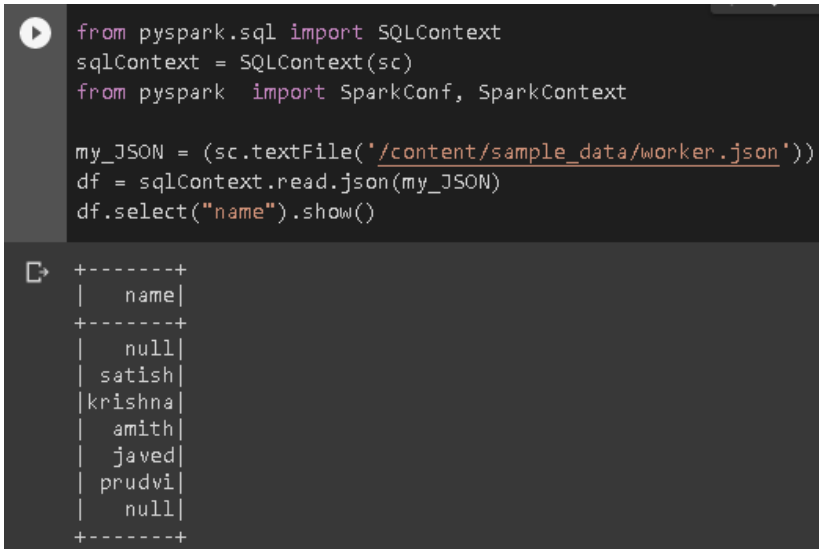


3.4.3 Sử dụng phương pháp Select

Sử dụng lệnh sau để tìm nạp tên-cột trong số ba cột từ DataFrame.[5]

```
df.select("name").show()
```

Output:

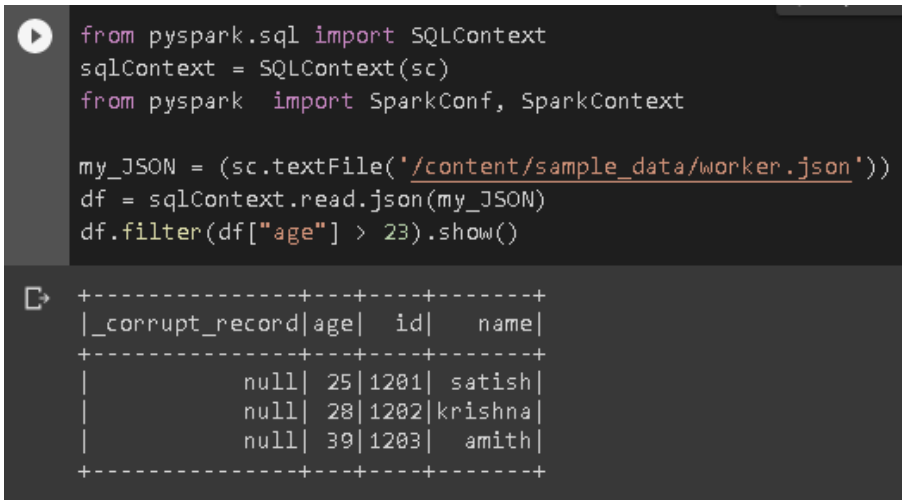


3.4.4 Sử dụng filter độ tuổi

Sử dụng lệnh sau để tìm nhân viên có tuổi lớn hơn 23 (tuổi> 23).[5]

```
df.filter(df["age"] > 23).show()
```

Output:

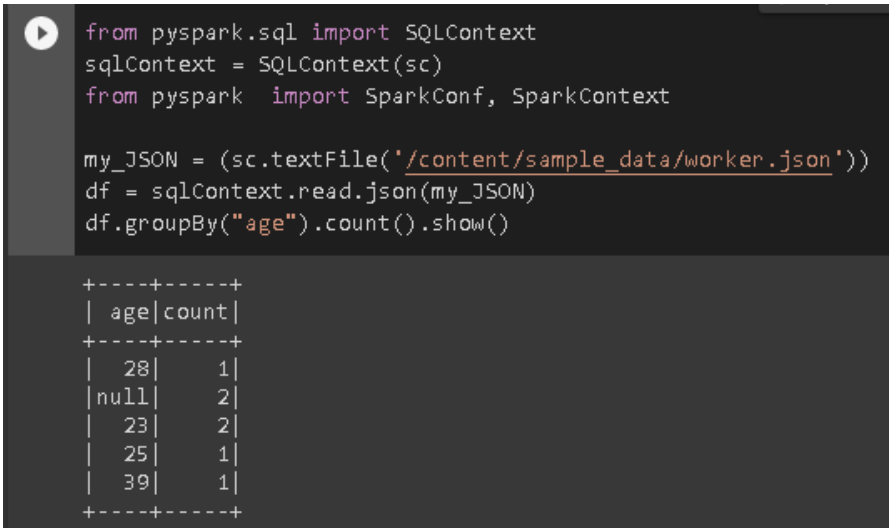


3.4.5 Sử dụng phương pháp groupBy

Sử dụng lệnh sau để đếm số lượng nhân viên ở cùng độ tuổi.[5]

```
df.groupBy("age").count().show()
```

Output:



Ngoài ra còn một số ví dụ cơ bản khác:

```
# Tạo DataFrame mới chỉ chứa "người dùng trẻ"

young = users.filter(users.age < 21)

# Ngoài ra, sử dụng cú pháp Pandas-like

young = users[users.age < 21]

# Tăng tuổi của mọi người thêm 1

young.select(young.name, young.age + 1)

# Đếm số lượng người dùng trẻ theo giới tính

young.groupBy("gender").count()

# Tham gia với người dùng trẻ bằng một DataFrame khác được gọi là nhật ký

young.join(logs, logs.userId == users.userId, "left_outer")
```

Ngoài ra bạn cũng có thể kết hợp SQL trong khi làm việc với DataFrames bằng Spark SQL. Ví dụ này đếm số lượng người dùng trẻ trong DataFrame.

```
young.registerTempTable("young")

context.sql("SELECT count(*) FROM young")
```

Trong Python, bạn cũng có thể chuyển đổi tự do giữa Pandas DataFrame và Spark DataFrame:

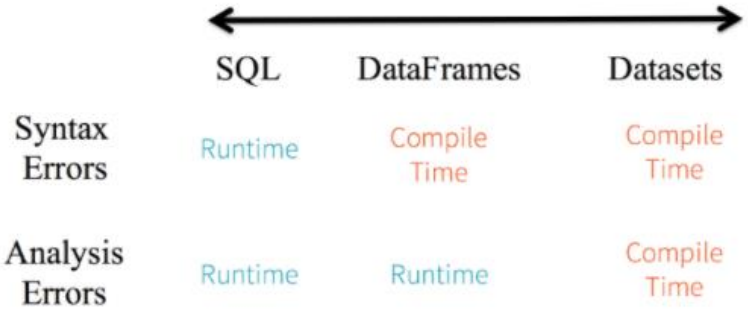
```
# Chuyển đổi Spark DataFrame sang Pandas

pandas_df = young.toPandas()

# Tạo một Dữ liệu Spark từ Pandas

spark_df = context.createDataFrame(pandas_df)
```

3.5 Hạn chế của SparkSQL DataFrames

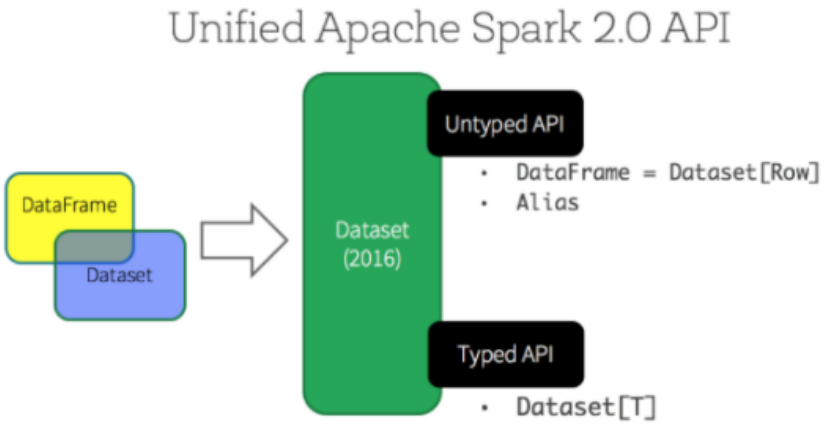
Có một số hạn chế của Dataframes trong Spark SQL, như:

- Trong Dataframes SQL, không có an toàn kiểu thời gian biên dịch. Do đó, vì cấu trúc không xác định nên không thể thao tác dữ liệu.[6]

- Chúng ta có thể chuyển đổi đối tượng miền thành DataFrame. Nhưng một khi chúng ta làm điều đó, thì chúng ta không thể tạo lại đối tượng miền.[6]



Để giải quyết nhược điểm này của DataFrame, Spark đã giới thiệu Datasets API từ năm 2015 (Apache Spark 1.6) bằng cách kết hợp những ưu điểm của cả RDD và DataFram. Không giống như RDD sử dụng tuần tự hóa Java, Dataset sử dụng Bộ mã hóa để tuần tự hóa các đối tượng (Tuần tự hóa là quá trình chuyển đổi một Đối tượng thành một chuỗi byte để thao tác hoặc truyền đối tượng qua mạng. De-serialization là quá trình ngược lại của Serialization chuyển đổi một chuỗi của byte trở lại một Đối tượng). Cơ chế tuần tự hóa mã hóa cho phép Spark thực hiện một số hoạt động như lọc, sắp xếp, băm,… mà không cần De-serialization như trong cơ chế Serialization thông thường. DataFrame và Dataset đã được hợp nhất thành một API kể từ Spark 2.0,trong đó DataFrame là một bí danh kiểu của DataSet [Row] (Row là một đối tượng chưa được định kiểu) và Dataset [T] là một tập hợp các đối tượng được gõ mạnh với 'T' là một đối tượng được định kiểu.[6]



Tài liệu tham khảo

1. <URL: https://spark.apache.org/docs/2.4.0/configuration.html>

2. <URL: https://www.tutorialspoint.com/apache_spark/apache_spark_rdd.htm>

3. <URL: https://data-flair.training/blogs/spark-rdd-tutorial/>

4. <URL: https://databricks.com/blog/2015/02/17/introducing-dataframes-in-spark-for-large-scale-data-science.html>

5. <URL: https://www.tutorialspoint.com/spark_sql/spark_sql_dataframes.htm>

6. <URL: http://itechseeker.com/en/tutorials-2/apache-spark-3/apache-spark-with-scala/spark-sql-dataset-and-dataframes/>