**Tạ Văn Trọng – 51702048**

**TÌM HIỂU VỀ SPARK**

**Spark properties**

Thuộc tính Spark kiểm soát 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](https://spark.apache.org/docs/latest/api/scala/org/apache/spark/SparkConf.html) được chuyển đến SparkContext. SparkConf cho phép bạn định cấu hình một số thuộc tính chung (ví dụ: URL chính và tên ứng dụng), cũng như các cặp khóa-giá trị tùy ý thông qua set() phương thức. Ví dụ: chúng ta có thể khởi tạo một ứng dụng với hai luồng như sau:

Lưu ý rằng khi chạy với local [2], nghĩa là hai luồng - thể hiện sự song song “tối thiểu”, có thể giúp phát hiện lỗi chỉ tồn tại khi chạy trong bối cảnh phân tán.



Lưu ý rằng chúng ta có thể có nhiều hơn 1 luồng ở chế độ cục bộ và trong những trường hợp như Spark Streaming, chúng tôi thực sự có thể yêu cầu nhiều hơn 1 luồng để ngăn chặn bất kỳ loại vấn đề rắc rối nào.

Các thuộc tính chỉ định một số khoảng thời gian nên được cấu hình với một đơn vị thời gian. Ví dụ:

* 25ms (milliseconds)
* 5s (seconds)
* 10m or 10min (minutes)
* 3h (hours)
* 5d (days)
* 1y (years)

Thuộc tính chỉ định kích thước byte phải được cấu hình với đơn vị kích thước. Ví dụ:

* 1b (bytes)
* 1k or 1kb (kibibytes = 1024 bytes)
* 1m or 1mb (mebibytes = 1024 kibibytes)
* 1g or 1gb (gibibytes = 1024 mebibytes)
* 1t or 1tb (tebibytes = 1024 gibibytes)
* 1p or 1pb (pebibytes = 1024 tebibytes)

Thuộc tính có sẵn:

|  |  |  |  |
| --- | --- | --- | --- |
| **Property Name** | **Default** | **Meaning** | **Since Version** |
| spark.app.name | (none) | Tên ứng dụng của bạn. Điều này sẽ xuất hiện trong giao diện người dùng và trong dữ liệu nhật ký. | 0.9.0 |
| spark.driver.cores | 1 | Số lõi để sử dụng cho quy trình trình điều khiển, chỉ ở chế độ cụm. | 1.3.0 |
| spark.driver.maxResultSize | 1g | 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. | 1.2.0 |
| spark.driver.memory | 1g | 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) .  Lưu ý: Trong chế độ máy khách, cấu hình này không được đặt thông qua SparkConf trực tiếp trong ứng dụng của bạn, vì trình điều khiển JVM đã bắt đầu tại thời điểm đó. Thay vào đó, hãy đặt điều này thông qua --driver-memorytùy chọn dòng lệnh hoặc trong tệp thuộc tính mặc định của bạn. | 1.1.1 |
| spark.driver.memoryOverhead | driverMemory \* 0.10, with minimum là 384 | Số lượng bộ nhớ không heap sẽ được phân bổ 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. Lưu ý: Bộ nhớ không heap bao gồm bộ nhớ off-heap (khi nào spark.memory.offHeap.enabled=true) và bộ nhớ được sử dụng bởi các quy trình trình điều khiển khác (ví dụ: quy trình python đi với trình điều khiển PySpark) và bộ nhớ được sử dụng bởi các quy trình không phải trình điều khiển khác đang chạy trong cùng một vùng chứa. Kích thước bộ nhớ tối đa của vùng chứa đến trình điều khiển đang chạy được xác định bằng tổng của spark.driver.memoryOverhead và spark.driver.memory. | 2.3.0 |
| spark.driver.resource.{resourceName}.amount | 0 | Số lượng của một loại tài nguyên cụ thể để sử dụng trên trình điều khiển. Nếu điều này được sử dụng, bạn cũng phải chỉ định spark.driver.resource.{resourceName}.discoveryScript trình điều khiển để tìm tài nguyên khi khởi động. | 3.0.0 |
| spark.driver.resource.{resourceName}.discoveryScript | none | Tập lệnh cho trình điều khiển chạy để khám phá một loại tài nguyên cụ thể. Điều này sẽ ghi vào STDOUT một chuỗi JSON ở định dạng của lớp ResourceInformation. Điều này có một tên và một mảng địa chỉ. Đối với trình điều khiển do khách hàng gửi, tập lệnh khám phá phải gán các địa chỉ tài nguyên khác nhau cho trình điều khiển này so với các trình điều khiển khác trên cùng một máy chủ. | 3.0.0 |
| spark.driver.resource.{resourceName}.vendor | none | Nhà cung cấp tài nguyên để sử dụng cho trình điều khiển. Tùy chọn này hiện chỉ được hỗ trợ trên Kubernetes và thực tế là cả nhà cung cấp và miền tuân theo quy ước đặt tên plugin thiết bị Kubernetes. (ví dụ: Đối với GPU trên Kubernetes, cấu hình này sẽ được đặt thành nvidia.com hoặc amd.com) | 3.0.0 |
| spark.resources.discoveryPlugin | org.apache.spark.resource.ResourceDiscoveryScriptPlugin | Danh sách tên lớp được phân tách bằng dấu phẩy triển khai org.apache.spark.api.resource.ResourceDiscoveryPlugin để tải vào ứng dụng. Điều này dành cho người dùng nâng cao để thay thế lớp khám phá tài nguyên bằng một triển khai tùy chỉnh. Spark sẽ thử từng lớp được chỉ định cho đến khi một trong số chúng trả về thông tin tài nguyên cho tài nguyên đó. Nó sẽ thử tập lệnh khám phá lần cuối nếu không có plugin nào trả lại thông tin cho tài nguyên đó. | 3.0.0 |
| spark.executor.memory | 1g | Dung lượng bộ nhớ sử dụng cho mỗi quá trình thi hành di chúc, 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). | 0.7.0 |
| spark.executor.pyspark.memory | No set | Lượng bộ nhớ được cấp phát cho PySpark trong mỗi trình thực thi, trong MiB trừ khi được chỉ định khác. Nếu được đặt, bộ nhớ PySpark cho một trình thực thi sẽ bị giới hạn ở số lượng này. Nếu không được thiết lập, Spark sẽ không giới hạn việc sử dụng bộ nhớ của Python và tùy thuộc vào ứng dụng để tránh vượt quá không gian bộ nhớ trên cao được chia sẻ với các quy trình không phải JVM khác. Khi PySpark được chạy trong YARN hoặc Kubernetes, bộ nhớ này được thêm vào các yêu cầu tài nguyên của trình thực thi. Lưu ý: Tính năng này phụ thuộc vào mô-đun `tài nguyên` của Python; do đó, các hành vi và hạn chế được kế thừa. Ví dụ: Windows không hỗ trợ giới hạn tài nguyên và tài nguyên thực tế không bị giới hạn trên MacOS. | 2.4.0 |
| spark.executor.memoryOverhead | ExecutiveMemory \* 0,10, với tối thiểu là 384 | Lượng bộ nhớ bổ sung sẽ được cấp phát cho mỗi quá trình thực thi ở chế độ cụm, trong MiB trừ khi được chỉ định khác. Đây là bộ nhớ chiếm những thứ như tổng chi phí VM, các chuỗi được thực hiện, các chi phí gốc khác, v.v. Điều này có xu hướng phát triển theo kích thước trình thực thi (thường là 6-10%). Tùy chọn này hiện được hỗ trợ trên YARN và Kubernetes. Lưu ý: Bộ nhớ bổ sung bao gồm bộ nhớ người thực thi PySpark (khi spark.executor.pyspark.memorykhông được định cấu hình) và bộ nhớ được sử dụng bởi các quá trình không phải người thực thi khác đang chạy trong cùng một vùng chứa. Kích thước bộ nhớ tối đa của container để chạy thi hành di chúc được xác định bởi tổng của spark.executor.memoryOverhead, spark.executor.memory, spark.memory.offHeap.sizevà spark.executor.pyspark.memory. | 2.3.0 |
| spark.executor.resource.{resourceName}.amount | 0 | Số lượng của một loại tài nguyên cụ thể để sử dụng cho mỗi quá trình thực thi. Nếu điều này được sử dụng, bạn cũng phải chỉ định spark.executor.resource.{resourceName}.discoveryScript trình thực thi để tìm tài nguyên khi khởi động. | 3.0.0 |
| spark.executor.resource.{resourceName}.discoveryScript | none | Một tập lệnh để trình thực thi chạy để khám phá một loại tài nguyên cụ thể. Điều này sẽ ghi vào STDOUT một chuỗi JSON ở định dạng của lớp ResourceInformation. Điều này có một tên và một mảng địa chỉ. | 3.0.0 |
| spark.executor.resource.{resourceName}.vendor | none | Nhà cung cấp tài nguyên để sử dụng cho những người thực thi. Tùy chọn này hiện chỉ được hỗ trợ trên Kubernetes và thực tế là cả nhà cung cấp và miền tuân theo quy ước đặt tên plugin thiết bị Kubernetes. (ví dụ: Đối với GPU trên Kubernetes, cấu hình này sẽ được đặt thành nvidia.com hoặc amd.com) | 3.0.0 |
| spark.extraListeners | (none) | Danh sách các lớp được phân tách bằng dấu phẩy triển khai SparkListener; khi khởi tạo SparkContext, các cá thể của các lớp này sẽ được tạo và đăng ký với bus lắng nghe của Spark. Nếu một lớp có một phương thức khởi tạo một đối số chấp nhận một SparkConf, thì phương thức khởi tạo đó sẽ được gọi; nếu không, một hàm tạo không đối số sẽ được gọi. Nếu không tìm thấy hàm tạo hợp lệ nào, quá trình tạo SparkContext sẽ không thành công với một ngoại lệ. | 1.3.0 |
| spark.local.dir | / tmp | Thư mục để sử dụng cho không gian "đầu" trong Spark, bao gồm các tệp đầu ra bản đồ và RDD được lưu trữ trên đĩa. Điều này phải nằm trên một đĩa cục bộ, nhanh trong hệ thống của bạn. Nó cũng có thể là một danh sách được phân tách bằng dấu phẩy gồm nhiều thư mục trên các đĩa khác nhau. Lưu ý: Điều này sẽ bị ghi đè bởi các biến môi trường SPARK\_LOCAL\_DIRS (Độc lập), MESOS\_SANDBOX (Mesos) hoặc LOCAL\_DIRS (YARN) do người quản lý cụm thiết lập. | 0.5.0 |
| spark.logConf | false | Ghi lại SparkConf hiệu quả dưới dạng INFO khi SparkContext được khởi động. | 0.9.0 |
| spark.master | (none) | Người quản lý cụm để kết nối. Xem danh sách URL chính được phép. | 0.9.0 |
| spark.submit.deployMode | (none) | Chế độ triển khai của chương trình trình điều khiển Spark, "máy khách" hoặc "cụm", có nghĩa là khởi chạy chương trình trình điều khiển cục bộ ("máy khách") hoặc từ xa ("cụm") trên một trong các nút bên trong cụm. | 1.5.0 |
| spark.log.callerContext | (none) | Thông tin ứng dụng sẽ được ghi vào nhật ký Yarn RM / nhật ký kiểm tra HDFS khi chạy trên Yarn / HDFS. Chiều dài của nó phụ thuộc vào cấu hình Hadoop hadoop.caller.context.max.size. Nó phải ngắn gọn và thường có thể có tối đa 50 ký tự. | 2.2.0 |
| spark.driver.supervise | false | Nếu đúng, hãy tự động khởi động lại trình điều khiển nếu nó không thành công với trạng thái thoát khác không. Chỉ có hiệu lực trong chế độ độc lập Spark hoặc chế độ triển khai cụm Mesos. | 1.3.0 |
| spark.driver.log.dfsDir | (none) | Thư mục cơ sở trong đó nhật ký trình điều khiển Spark được đồng bộ hóa, nếu spark.driver.log.persistToDfs.enabled đúng. Trong thư mục cơ sở này, mỗi ứng dụng ghi nhật ký trình điều khiển vào một tệp ứng dụng cụ thể. Người dùng có thể muốn đặt vị trí này thành một vị trí thống nhất như thư mục HDFS để các tệp nhật ký trình điều khiển có thể được duy trì để sử dụng sau này. Thư mục này sẽ cho phép mọi người dùng Spark đọc / ghi tệp và người dùng Spark History Server xóa tệp. Ngoài ra, các nhật ký cũ hơn từ thư mục này sẽ được máy chủ Spark làm sạch nếu spark.history.fs.driverlog.cleaner.enabledđúng và nếu chúng cũ hơn tuổi tối đa được định cấu hình bằng cách cài đặt spark.history.fs.driverlog.cleaner.maxAge. | 3.0.0 |
| spark.driver.log.persistToDfs.enabled | false | Nếu đúng, ứng dụng spark đang chạy ở chế độ máy khách sẽ ghi nhật ký trình điều khiển vào một bộ lưu trữ liên tục, được định cấu hình trong spark.driver.log.dfsDir. Nếu spark.driver.log.dfsDirkhông được định cấu hình, nhật ký trình điều khiển sẽ không được lưu giữ. Ngoài ra, hãy bật trình dọn dẹp bằng cách đặt spark.history.fs.driverlog.cleaner.enabled  thành true trong Spark History Server . | 3.0.0 |
| spark.driver.log.layout | % d {yy / MM / dd HH: mm: ss.SSS}% t% p% c {1}:% m% n | Bố cục cho nhật ký trình điều khiển được đồng bộ hóa với spark.driver.log.dfsDir. Nếu điều này không được định cấu hình, nó sẽ sử dụng bố cục cho appender đầu tiên được xác định trong log4j.properties. Nếu điều đó cũng không được định cấu hình, nhật ký trình điều khiển sử dụng bố cục mặc định. | 3.0.0 |
| spark.driver.log.allowErasureCoding | flase | Có cho phép nhật ký trình điều khiển sử dụng mã xóa hay không. Trên HDFS, các tệp được mã hóa xóa sẽ không cập nhật nhanh như các tệp được sao chép thông thường, do đó, chúng mất nhiều thời gian hơn để phản ánh các thay đổi do ứng dụng viết. Lưu ý rằng ngay cả khi điều này là đúng, Spark vẫn sẽ không buộc tệp sử dụng mã hóa xóa, nó sẽ chỉ sử dụng mặc định của hệ thống tệp. | 3.0.0 |

**Spark RDD**

1. **Resilient Distributed Datasets**

Resilient Distributed Datasets (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 cả các lớp do người dùng định nghĩa.

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 chịu được lỗi có thể hoạt động song song.

Có hai cách để tạo RDD - **song song** một tập hợp hiện có trong chương trình trình điều khiển của bạn hoặc **tham chiếu tập dữ liệu** 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 Định dạng đầu vào Hadoop.

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.

1. **Data Sharing is Slow in MapReduce**

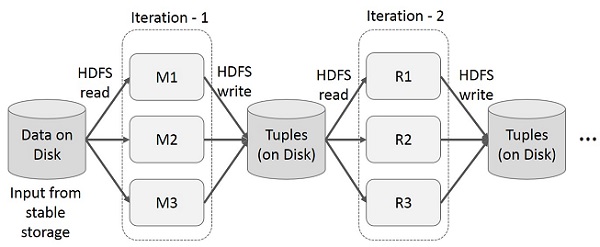
MapReduce được sử dụng rộng rãi để xử lý và tạo các bộ dữ liệu lớn với một thuật toán phân tán, song song trên một cụm. Nó cho phép người dùng viết các phép tính song song, sử dụng một tập hợp các toán tử cấp cao, mà không phải lo lắng về việc phân phối công việc và khả năng chịu lỗi.

Thật không may, trong hầu hết các khuôn khổ hiện tại, cách duy nhất để sử dụng lại dữ liệu giữa các lần tính toán (Ví dụ - giữa hai công việc MapReduce) là ghi dữ liệu đó vào hệ thống lưu trữ ổn định bên ngoài (Ví dụ - HDFS). Mặc dù khung công tác này cung cấp nhiều nội dung trừu tượng để truy cập tài nguyên tính toán của cụm, người dùng vẫn muốn nhiều hơn thế.

Cả hai ứng dụng **lặp lại** và **tương tác đều** yêu cầu chia sẻ dữ liệu nhanh hơn trên các công việc song song. Chia sẻ dữ liệu chậm trong MapReduce do **sao chép, tuần tự hóa** và **IO đĩa**. Về hệ thống lưu trữ, hầu hết các ứng dụng Hadoop, chúng dành hơn 90% thời gian để thực hiện các thao tác đọc-ghi HDFS.

1. **Iterative Operations on MapReduce**

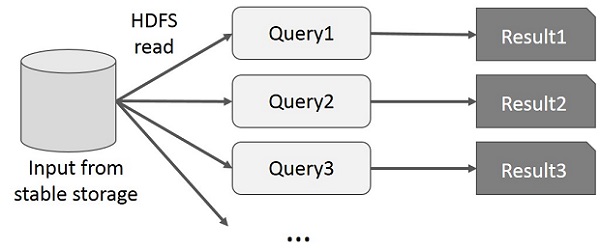
Sử dụng lại các kết quả trung gian qua nhiều lần tính toán trong các ứng dụng nhiều giai đoạn. Hình minh họa sau đây giải thích cách hoạt động của khung hiện tại trong khi thực hiện các hoạt động lặp lại trên MapReduce. Điều này phát sinh chi phí đáng kể do sao chép dữ liệu, I / O đĩa và tuần tự hóa, khiến hệ thống chậm.



1. **Interactive Operations on MapReduce**

Người dùng chạy các truy vấn đặc biệt trên cùng một tập con dữ liệu. Mỗi truy vấn sẽ thực hiện I/O đĩa trên bộ nhớ ổn định, có thể chi phối thời gian thực thi ứng dụng.

Hình minh họa sau giải thích cách hoạt động của khung hiện tại khi thực hiện các truy vấn tương tác trên MapReduce.



1. **Data Sharing using Spark RDD**

Chia sẻ dữ liệu trong MapReduce do **sao chép, tuần tự hóa** và **IO đĩa**. Hầu hết các ứng dụng Hadoop, chúng dành hơn 90% thời gian để thực hiện các thao tác đọc-ghi HDFS.

Nhận thức được vấn đề này, các nhà nghiên cứu đã phát triển một framework chuyên biệt có tên là Apache Spark.

Ý tưởng chính của spark là **R**esilient **D**istributed **D**atasets (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 ổ đĩa từ 10 đến 100 lần.

Bây giờ chúng ta hãy thử tìm hiểu cách các hoạt động lặp lại và tương tác diễn ra trong Spark RDD.

1. **Iterative Operations on Spark RDD**

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.



1. **Interactive Operations on Spark RDD**

Hình minh họa này 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.



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.

1. **Các loại RDD**



* Các RDD biểu diễn một tập hợp cố định, đã được phân vùng các record để có thể xử lý song song.
* Các record trong RDD có thể là đối tượng Java, Scale hay Python tùy lập trình viên chọn. Không giống như DataFrame, mỗi record của DataFrame phải là một dòng có cấu trúc chứa các field đã được định nghĩa sẵn.
* RDD đã từng là API chính được sử dụng trong series Spark 1.x và vẫn có thể sử dụng trong version 2.X nhưng không còn được dùng thường xuyên nữa.
* RDD API có thể được sử dụng trong Python, Scala hay Java:
  + Scala và Java: Perfomance tương đương trên hầu hết mọi phần. (Chi phí lớn nhất là khi xử lý các raw object)
  + Python: Mất một lượng performance, chủ yếu là cho việc serialization giữa tiến trình Python và JVM

**Spark DataFrame**

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 các bảng quan hệ có kỹ thuật tối ưu hóa tốt.

Một DataFrame có thể được xây dựng từ một loạt các nguồn khác nhau như bảng Hive, tệp Dữ liệu có cấu trúc, cơ sở dữ liệu bên ngoài hoặc RDD hiện có. API này được thiết kế cho các ứng dụng Khoa học dữ liệu và Dữ liệu lớn hiện đại lấy cảm hứng từ **DataFrame in R Programming** and **Pandas in Python**.

1. **Các tính năng của DataFrame**

Dưới đây là một số tính năng đặc trưng của DataFrame:

* 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.

1. **Hoạt động DataFrame**

DataFrame cung cấp một ngôn ngữ dành riêng cho miền để thao tác dữ liệu có cấu trúc. Ở đây, chúng tôi bao gồm một số ví dụ cơ bản về xử lý dữ liệu có cấu trúc bằng DataFrames.

**Tệp 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"}  } |

**Đọc tài liệu JSON**

Đầu tiên, chúng ta phải đọc tài liệu JSON. Dựa trên điều này, tạo một DataFrame có tên (dfs).

Sử dụng lệnh sau để đọc tài liệu JSON có tên là **employee.json**. Dữ liệu được hiển thị dưới dạng bảng với các trường - id, tên và tuổi.

|  |
| --- |
| val dfs = sqlContext.read.json("employee.json") |

**Đầu ra** - Tên trường được lấy tự động từ **employee.json**.

|  |
| --- |
| dfs: org.apache.spark.sql.DataFrame = [age: string, id: string, name: string] |

**Hiển thị dữ liệu**

Nếu bạn muốn xem dữ liệu trong DataFrame, hãy sử dụng lệnh sau.

|  |
| --- |
| dfs.show() |

**Đầu ra** - Bạn có thể xem dữ liệu nhân viên ở định dạng bảng.

|  |
| --- |
| <console>:22, took 0.052610 s  +----+------+--------+  |age | id | name |  +----+------+--------+  | 25 | 1201 | satish |  | 28 | 1202 | krishna|  | 39 | 1203 | amith |  | 23 | 1204 | javed |  | 23 | 1205 | prudvi |  +----+------+--------+ |

Nguồn:

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

<https://laptrinh.vn/books/apache-spark/page/apache-spark-rdd>

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