TÌM HIỂU SPARK

1. Spark properties.

Thuộc tính Spark, hay Spark properties, được dùng để 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 và được chuyển tới SparkContext của bạn.
SparkConf cho phép ta định dạng cấu hình của một số thuộc tinh phổ biến (ví dụ: URL chính, tên ứng dụng), cũng như các cặp key-value tùy ý thông qua phương thức set().
Ví dụ: khởi tạo một ứng dụng với hai luồng như sau:

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

Lưu ý rằng, ta chạy ứng dụng với local[2], nghĩa là hai luồng – thể hiện sự song song “tối thiểu”, có thể giúp phát hiện lỗi chỉ tồn tại khi chúng tôi chạy trong ngữ cảnh phân tán.

Các thuộc tính chỉ định một số khoảng thời gian nên được cấu hình với một đơn vị thời gian. Các định dạng sau được chấp nhận:

```
10ms (milliseconds)
1s (seconds)
10m or 10min (minutes)
10h (hours)
10d (days)
1y (years)
```

Các thuộc tính chỉ định kích thước byte nên được cấu hình với đơn vị kích thước. Định dạng sau được chấp nhận:

```
1b (bytes)
1k or 1kb
1m or 1mb
1g or 1gb
1t or 1tb
```

1p or 1pb

1.1 Các thuộc tính có sẵn

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

1.1.1 Application Properties (Thuộc tính ứng dụng)

Tên thuộc tính	Tác dụng
spark.app.name	Tên ứng dụng. Nó sẽ xuất hiện trong giao diện người dùng và log data
spark.driver.cores	Số lõi (cores) để sử dụng cho quy trình điều khiển (chỉ ở chế độ Cluster)
spark.driver.maxResultSize	Giới hạn tổng kích thước của các kết quả được tuần tự hóa của tất cả các phân vùng cho mỗi hàm trong Spark (ví dụ: collect), được tính bằng bytes
spark.driver.memory	Dung lượng bộ nhớ sẽ sử dụng cho quá trình điều khiển (nơi SparkContext được khởi tạo)
spark.executor.pyspark.memory	Lượng bộ nhớ được cấp phát cho PySpark trong mỗi trình thực thi
...	

1.1.2 Runtime Enviroment (Môi trường thực thi)

Tên thuộc tính	Tác dụng
spark.python.profile	Bật cấu hình trong Python worker, kết quả cấu hình sẽ hiển thị bằng sc.show_profiles () hoặc nó sẽ được hiển thị trước khi trình điều khiển thoát
spark.python.worker.memory	Dung lượng bộ nhớ sẽ sử dụng cho mỗi quá trình python worker trong quá trình tổng hợp,
spark.pyspark.python	Thực thi nhị phân Python để sử dụng cho PySpark trong trình điều khiển và trình thực thi.
...	

2. Spark RDD Resilient Distributed Datasets (RDD) là một cấu trúc dữ liệu cơ bản

của Spark.

Nó là một tập hợp bất biến phân tán của một đối tượng. Mỗi dataset trong RDD được chia ra nhiều phần vùng logical. Có thể được tính toán trên các node khác nhau của một cụm máy chủ (cluster).

RDD có thể chứa bất kỳ kiểu dữ liệu nào của Python, Java, hoặc đối tượng Scala, bao gồm các kiểu dữ liệu do người dùng định nghĩa. Thông thường, RDD chỉ cho phép đọc, phân mục tập hợp của các bản ghi. RDDs có thể được tạo ra qua điều

khiển xác định trên dữ liệu trong bộ nhớ hoặc RDDs, RDD là một tập hợp có khả năng chịu lỗi mỗi thành phần có thể được tính toán song song.

Có hai cách để tạo RDDs:

- Tạo từ một tập hợp dữ liệu có sẵn trong ngôn ngữ sử dụng như Java, Python, Scala.
- Lấy từ dataset hệ thống lưu trữ bên ngoài như HDFS, Hbase hoặc các cơ sở dữ liệu quan hệ.

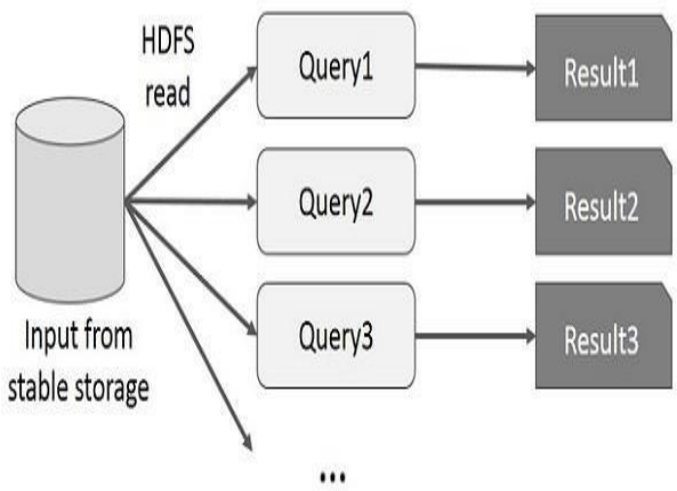
2.1 Thực thi RDD

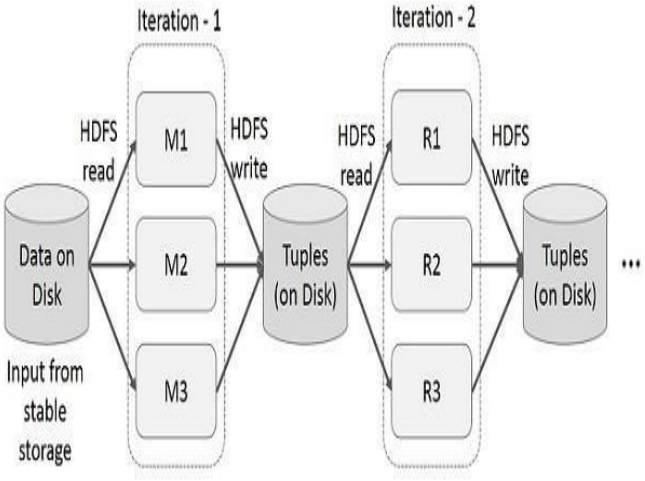
2.1.1 Thực thi trên MapReduce

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

Tuy nhiên, trong hầu hết các framework hiện tại, cách duy nhất để sử dụng lại dữ liệu giữa các tính toán (Ví dụ: giữa hai công việc MapReduce) là ghi nó vào storage (Ví dụ: HDFS). Mặc dù framework này cung cấp nhiều hàm thư viện để truy cập vào tài nguyên tính toán của cụm Cluster, điều đó vẫn là chưa đủ.

Cả hai ứng dụng Lặp (Iterative) và Tương tác (Interactive) đều yêu cầu chia sẻ truy





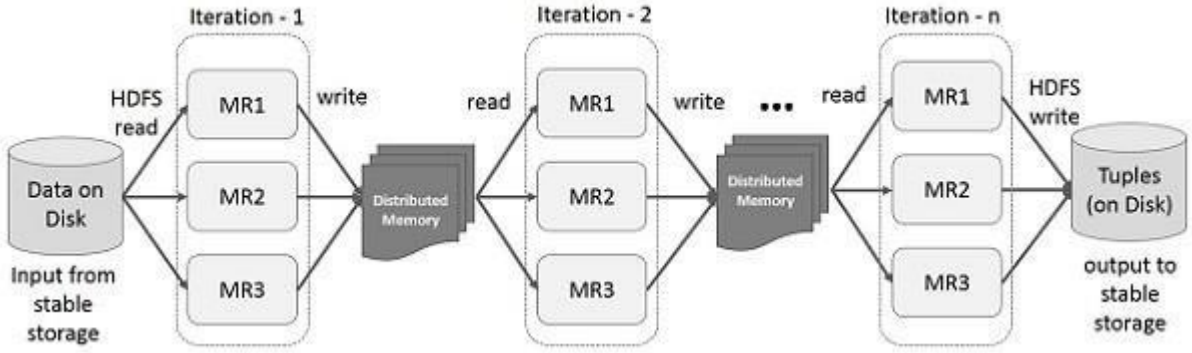
cập và xử lý dữ liệu nhanh hơn trên các công việc song song. Chia sẻ dữ liệu chậm trong MapReduce do sao chép tuần tự và tốc độ I/O của ổ đĩa. Về hệ thống lưu trữ, hầu hết các ứng dụng Hadoop, cần dành hơn 90% thời gian để thực hiện các thao tác đọc-ghi HDFS. Dưới đây là hình minh họa cho hai ứng dụng này:

Hình 2. Iterative Operation trên MapReduce

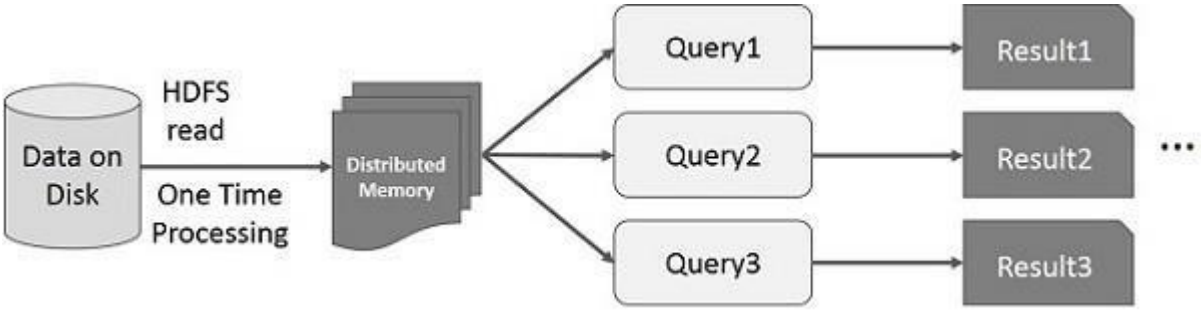
Hình 1. Iterative Operation trên MapReduce

2.1.2 Thực thi trên Spark RDD

Để khắc phục được vấn đề về MapRedure, các nhà nghiên cứu đã phát triển một framework chuyên biệt gọi là Apache Spark. Ý tưởng chính của Spark là Resilient Distributed Datasets (RDD); nó hỗ trợ tính toán xử lý trong bộ nhớ. Điều này có nghĩa, nó lưu trữ trạng thái của bộ nhớ dưới dạng một đối tượng trên các công việc và đối tượng có thể chia sẻ giữa các công việc đó. Việc xử lý dữ liệu trong bộ nhớ nhanh hơn 10 đến 100 lần so với network và disk.



Hình 3. Iterative Operation trên RDD



Hình 4. Interative Operation trên RDD

2.2 Các transformation và action với RDD

RDD cung cấp các transformation và action hoạt động giống như DataFrame lẫn DataSets. Transformation xử lý các thao tác lazily và Action xử lý thao tác cần xử lý tức thời.

2.2.1 Một số Transformation

- **distinct**: loại bỏ trùng lắp trong RDD
- **filter**: tương đương với việc sử dụng where trong SQL – tìm các record trong RDD xem những phần tử nào thỏa điều kiện. Có thể cung cấp một hàm phức tạp sử dụng để filter các record cần thiết – Như trong Python, ta có thể sử dụng hàm lambda để truyền vào filter
- **map**: thực hiện một công việc nào đó trên toàn bộ RDD. Trong Python sử dụng lambda với từng phần tử để truyền vào map
- **flatMap**: cung cấp một hàm đơn giản hơn hàm map. Yêu cầu output của map phải là một structure có thể lặp và mở rộng được.

- **sortBy**: mô tả một hàm để trích xuất dữ liệu từ các object của RDD và thực hiện sort được từ đó.
- **randomSplit**: nhận một mảng trọng số và tạo một random seed, tách các RDD thành một mảng các RDD có số lượng chia theo trọng số.

2.2.2 Một số Action: được thực thi ngay khi transformation được thiết lập

- **reduce**: thực hiện hàm reduce trên RDD để thu về 1 giá trị duy nhất
- **count**: đếm số dòng trong RDD
- **countApprox:** phiên bản đếm xấp xỉ của count, nhưng phải cung cấp timeout vì có thể không nhận được kết quả.
- **countByValue**: đếm số giá trị của RDD
- **first**: lấy giá trị đầu tiên của dataset
- **max và min:** lần lượt lấy giá trị lớn nhất và nhỏ nhất của dataset

2.3. Một số kỹ thuật đối với RDD

- Lưu trữ file:
    - Thực hiện ghi vào các file plain-text. ○ Có thể sử dụng các codec nén từ thư viện của Hadoop. ○ Lưu trữ vào các database bên ngoài yêu cầu ta phải lặp qua tất cả partition của RDD – Công việc được thực hiện ngầm trong các highlevel API.
    - sequenceFile là một flat file chứa các cặp key-value, thường được sử dụng làm định dạng input/output của MapReduce. Spark có thể ghi các sequenceFile bằng các ghi lại các cặp key-value. ○ Đồng thời, Spark cũng hỗ trợ ghi nhiều định dạng file khác nhau, cho phép define các class, định dạng output, config và compression scheme của Hadoop.
- Caching: Tăng tốc xử lý bằng cache:
    - Caching với RDD, Dataset hay DataFrame có nguyên lý như nhau.
    - Chúng ta có thể lựa chọn cache hay persist một RDD, và mặc định, chỉ xử lý dữ liệu trong bộ nhớ.
- Checkpointing: Lưu trữ lại các bước xử lý để phục hồi:
    - Checkpointing lưu RDD vào đĩa cứng để các tiến trình khác để thể sử dụng lại RDD point này làm partition trung gian thay vì tính toán lại RDD từ các nguồn dữ liệu gốc. ○ Checkpointing cũng tương tự như cache, chỉ khác nhau là lưu trữ vào đĩa cứng và không dùng được trong API của DataFrame. ○ Cần sử dụng nhiều để tối ưu tính toán.

2.4 Code minh họa

```
files = (sc.textFile('/content/text.txt').map(lambda line:
line.split(' '))) fileContent = files.reduce(lambda x: 1)
```

```
rdd = sc.parallelize(fileContent) wrds = rdd.map(lambda
word: (word, 1)) red = wrds.reduceByKey(lambda x,y: x+y)
```

Phía trên là đoạn code minh họa dùng để đếm số lượng từ trong một file và đếm số lần xuất hiện của từng từ.

3. Spark DataFrame

3.1 Khái niệm

DataFrame là một kiểu dữ liệu collection phân tán, được tổ chức thành các cột được đặt tên. Về mặt khái niệm, nó tương đương với các bảng quan hệ (relational tables) đi kèm với các kỹ thuật tối ưu tính toán.

Hình 5. Một DataFrame tổng quát

DataFrame có thể được xây dựng từ nhiều nguồn dữ liệu khác nhau như Hive table, các file dữ liệu có cấu trúc hay bán cấu trúc (csv, json), các hệ cơ sở dữ liệu phổ biến (MySQL, MongoDB, Cassandra), hoặc RDDs hiện hành. API này được thiết kế cho các ứng dụng Big Data và Data Science hiện đại. Spark DataFrame là phiên bản Spark 1.3, có thể khắc phục được những hạn chế của RDD. Chúng ta có thể tạo DataFrame bằng cách sử dụng:

 Tệp dữ liệu có cấu trúc.

 Bàn trong tổ ong. Cơ sở dữ liệu bên ngoài. Sử dụng RDD hiện có.

3.2 Đặc điểm của DataFrame

3.2.1 Lợi thế

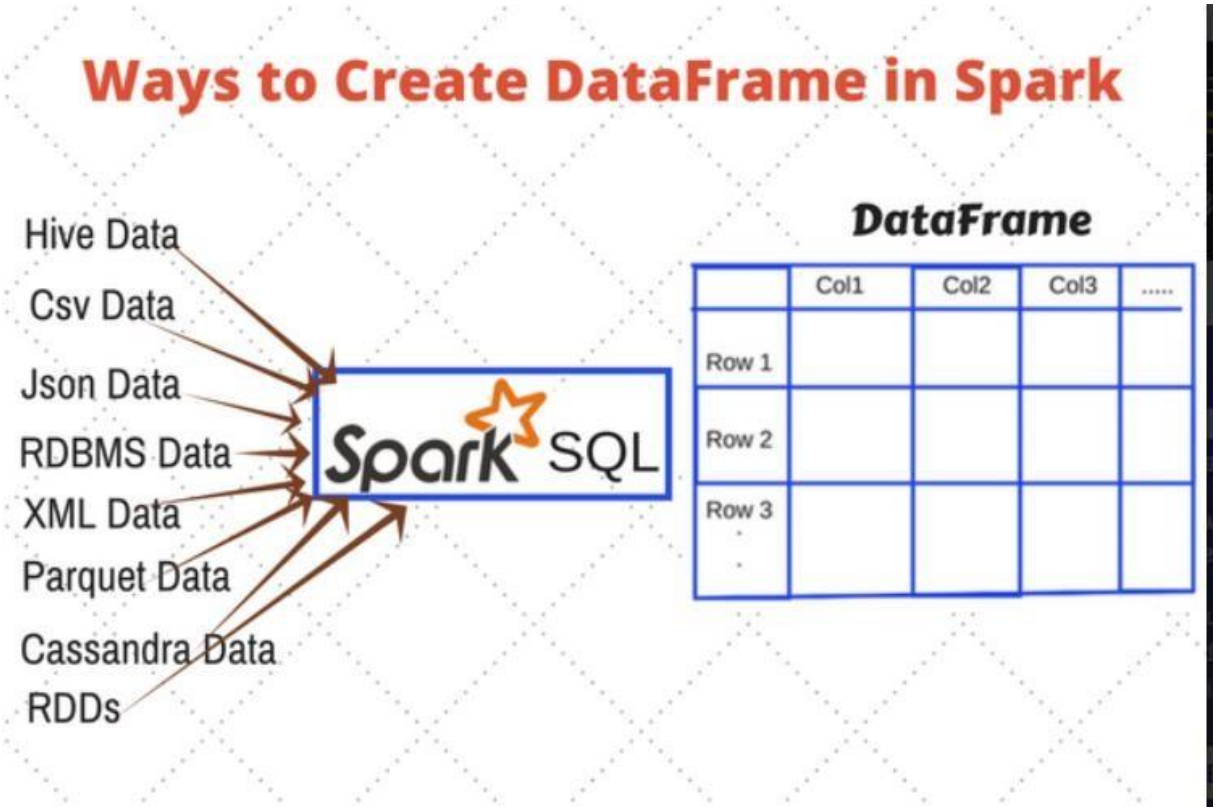
- DataFrames được thiết kế để xử lý tập hợp lớn dữ liệu có cấu trúc hoặc bán cấu trúc.
- Các quan sát trong Spark DataFrame được tổ chức dưới các cột được đặt tên, giúp Apache Spark hiểu được lược đồ của DataFrame. Điều này giúp Spark tối ưu hóa kế hoạch thực thi trên các truy vấn này.
- DataFrame trong Apache Spark có khả năng xử lý hàng petabyte dữ liệu.
- DataFrame hỗ trợ nhiều định dạng và nguồn dữ liệu.

- Nó có hỗ trợ API cho các ngôn ngữ khác nhau như Python, R, Scala, Java.

3.2.2 Khuyết điểm

Do kiểu dữ liệu được fix là row và truy cập dữ liệu trong DataFrame thông qua tên của cột (row) nên nếu có sai sót trong việc truyền tên cột, trình biên dịch sẽ không thể phát hiện ra lỗi mà khi thực thi mới có ngoại lệ.

3.3 Một số cách tạo DataFrame



Hình 6. Các cách tạo DataFrame trong Spark Có

rất nhiều cách để tạo DataFrame, dưới đây là một số cách phổ biến:

Cách 1: Tạo từ RDD

Nếu bạn đã có RDD với tên column và type tương ứng (TimestampType, IntegerType, StringType)thì bạn có thể dễ dàng tạo DataFrame bằng

```
sqlContext. createDataFrame(my_rdd, my_schema)
```

Tiếp theo, để in thông tin schema, ta dùng các hàm $printSchema()$, $dtypes$, và để trả về số record thì dùng hàm $count()$

```
fields = [StructField("access_time", TimestampType(), True),
StructField("userID", IntegerType(), True),
StructField("campaignID", StringType(), True)] schema
= StructType(fields)
whole_log_df = sqlContext.createDataFrame(whole_log,
schema) print whole_log_df.count() //đếm số hàng print
whole_log_df.printSchema() //xem các loại cột print
whole_log_df.dtypes //xem type print whole_log_df.show(5)
//in dữ liệu các cột với 5 dòng
```

Cách 2: Tạo trực tiếp từ file CSV

```
from pyspark.shell import spark from
pyspark.sql.types import *
# Định nghĩa Schema struct = StructType([ StructField('a',
StringType(), False),
     StructField('b', StringType(), False),
     StructField('c', StringType(), False) ])
# Tạo DataFrame từ file CSV df_data =
spark.read.csv('click_data_sample', struct)
```

Cách 3: Giao lưu trực tiếp từ file json

Bằng cách sử dụng $sqlContext.read.json$. Mỗi dòng của file json sẽ được coi là 1 object. Trong trường hợp object thiếu data thì sẽ null tại đó.

```
df_json = sqlContext.read.json("test_json.json")
df_json.printSchema() df_json.show(5)
```

Ngoài ra, còn có một số hàm cơ bản để thao tác với DataFrame. Cụ thể:

- $description()$: Hàm $description$ được sử dụng để tính toán thống kê tóm tắt của (các) cột số trong DataFrame. Nếu chúng tôi không chỉ định tên của các cột, nó sẽ tính toán thống kê tóm tắt cho tất cả các cột số có trong DataFrame: $data.describe().show()$

- $select()$: Để chọn ra tập con của các cột, chúng ta cần sử dụng thao tác chọn trên DataFrame và chúng ta cần chuyển các tên cột được phân tách bằng dấu phẩy bên trong Thao tác chọn:
  $data.select('id','name').show(3)$
- $dropDuplicates()$: Chúng ta có thể sử dụng thao tác $dropDuplicates$ để loại bỏ các hàng trùng lặp của DataFrame và lấy DataFrame sẽ không có các hàng trùng lặp. Ví dụ, nếu gọi hàm
  $data.select('id','name').dropDuplicates().show()$ thì ta sẽ loại bỏ các hàng trùng lặp ở hai cột trên và in ra phần tử là tất cả các hàng duy nhất của hai cột này.
- $filter()$: Chúng ta có thể áp dụng thao tác lọc ($filter$) trên một cột để lọc ra các giá trị tương ứng. Ví dụ: $data.filter(data.age > 20).count()$ tức là lọc các data có cột tuổi (age) lớn hơn 20 và in ra màn hình.
- $groupBy()$: Thao tác $groupBy$ có thể được sử dụng để nhóm các giá trị theo cột được gọi trong hàm. Ví dụ: $data.groupBy('age').count().show()$, tức là nhóm các giá trị theo cột tuổi và tiến hành count() để đếm số lượng, sau đó in ra màn hình.

Link colab: https://colab.research.google.com/drive/1U-r03c-kKRihWwo_CoyIc6CdhRpNg8wo?usp=sharing

TÀI LIỆU THAM KHẢO

[1] https://laptrinh.vn/books/apache-spark/page/apache-spark-rdd

[2] https://helpex.vn/article/spark-sql-huong-dan-gioi-thieu-5c6b25a6ae03f628d053c3e8#:~:text=Spark%20DataFrame%20l%C3%A0%20phi%C3%AAn%20b%E1%BA%A3n,d%E1%BB%AF%20li%E1%BB%87u%20trong%20R%20%2F%20Python.

[3] https://spark.apache.org/docs/latest/configuration.html#spark-properties