Lan man, Hadoop and Spark

Performance

- **Về tốc độ xử lý thì Spark nhanh hơn Hadoop**. Spark được cho là nhanh hơn Hadoop gấp `100 lần` khi chạy trên **`RAM`**, và gấp `10 lần` khi chạy trên **ổ cứng**.
- Hơn nữa, người ta cho rằng `Spark` sắp xếp (sort) `100TB` dữ liệu nhanh gấp 3 lần Hadoop trong khi sử dụng ít hơn 10 lần số lượng hệ thống máy tính.
- Sở dĩ `Spark` nhanh là vì nó xử lý mọi thứ ở `RAM`. Nhờ xử lý ở bộ nhớ nên `Spark` cung cấp các phân tích dữ liệu thời gian thực cho các chiến dịch quảng cáo, `machine learning`, hay các trang web mạng xã hội.
- Tuy nhiên, khi `Spark` làm việc cùng các dịch vụ chia sẻ khác chạy trên `YARN` thì hiệu năng có thể giảm xuống. Điều đó có thể dẫn đến rò rỉ bộ nhớ trên `RAM`. `Hadoop` thì khác, nó dễ dàng xử lý vấn đề này. **Nếu người dùng có khuynh hướng xử lý hàng loạt (`batch process`) thì `Hadoop` lại hiệu quả hơn `Spark`**.

Security

- Bảo mật của `Spark` đang được phát triển, hiện tại nó chỉ hỗ trợ xác thực mật khẩu (`password authentication`). Ngay cả trang web chính thức của `Apache Spark` cũng tuyên bố rằng, "Có rất nhiều loại mối quan tâm bảo mật khác nhau. Spark không nhất thiết phải bảo vệ chống lại tất cả mọi thứ".
- Mặt khác, `Hadoop` trang bị toàn bộ các mức độ bảo mật như `Hadoop Authentication`, `Hadoop Authorization`, `Hadoop Auditing`, and `Hadoop Encryption`. Tất cả các tính năng này liên kết với các dự án Hadoop bảo mật như `Knox Gateway` và `Sentry`.
- Vậy là ở **mặt bảo mật thì Spark kém bảo mật hơn Hadoop**.

=> **Nếu có thể tích hợp Spark với Hadoop thì Spark có thể "mượn" các tính năng bảo mật của Hadoop**.

Fee

Trước tiên, cả `Spark` và `Hadoop` đều là các `framework` mã nguồn mở (open source), nghĩa là nó miễn phí. Cả hai đều sử dụng các `server` chung, chạy trên `cloud`, và dường như chúng sử dụng các cầu hình phần cứng tương tự nhau.

	Apache Hadoop	Apache Spark
CORE	4	8-16
Memmory	24 GB	8 GB to hundreds of GB
DISKS	4-6 one TB disk	4-8
Network	1 GB Ethernet all-to-all	10 GB or more

- `Spark` cần một lượng lớn `RAM` vì nó xử lý mọi thứ ở bộ nhớ. Đây là yếu tố ảnh hưởng đến chi phí vì giá thành của `RAM` cao hơn ổ đĩa.

- Trong khi đó `Hadoop` bị ràng buộc bởi ổ đĩa, ổ đĩa thì rẻ hơn `RAM`. Tuy nhiên Hadoop thì cần nhiều hệ thống hơn để phân bổ ổ đĩa `I/O`.

Do vậy, khi nhu cầu của bạn là xử lý lượng lớn dữ liệu dạng lịch sử thì `Hadoop` là lựa chọn tốt vì dữ liệu dạng này cần lưu và có thể được xử lý trên ổ đĩa. Còn khi yêu cầu là xử lý dữ liệu thời gian thực thì `Spark` là lựa chọn tối ưu vì ta chỉ cần ít hệ thống cho sử lý cùng một lượng lớn dữ liệu với thời gian giảm nhiều lần hơn khi sử dụng `Hadoop`.

Easy to use

Một trong những ưu điểm lớn nhất của Spark là tính dễ sử dụng. `Spark` có giao diện người dùng thân thiện. `Spark` cung cấp các `API` thân thiện cho `Scala` `Java`, `Python`, và `Spark SQL` (hay còn gọi là Shark). Việc `Spark` được xây dựng từ các khối đơn giản nó giúp ta tạo các hàm do người dùng xác định một cách dễ dàng.

Trong khi đó `Hadoop` được viết bằng `Java` và gay ra cho sự khó khăn trong việc viết chương trình không có chế độ tương tác. Mặc dù `Pig` (một công cụ bổ trợ) giúp lập trình dễ dàng hơn, nhưng nó cũng tốn thời gian để học cú pháp.

Việc só sánh trên mang đến trong ta cảm giác `Spark` và `Hadoop` là "kẻ thù" của nhau. Vậy liệu rằng chúng có mối liên hệ kiểu hiệp lực nào không?

Câu trả lời là có. Hệ sinh thái `Apache Hadoop` bao gồm `HDFS (Hadoop Distributed File System)`, `Apache Query`, và `HIVE`.

Hãy xem `Apache Spark` có thể sử dụng gì từ chúng?

Integration Apache Spark vs HDFS

- Mục đích của `Apache Spark` là xử lý dữ liệu. Tuy nhiên, để xử lý dữ liệu, hệ thống cần dữ liệu đầu vào từ thiết bị lưu trữ. Và với mục đích này, `Spark` sử dụng `HDFS`. Đây không phải là lựa chọn duy nhất, nhưng là lựa chọn phổ biến nhất vì `Apache` là bộ não đằng sau cả hai.

Mixture Apache Hive and Apache Spark

- `Apache Spark` và `Apache Hive` có tính tương thích cao, vì cùng nhau, chúng có thể giải quyết nhiều vấn đề của nghiệp vụ.
- Chẳng hạn, giả sử rằng một doanh nghiệp đang phân tích hành vi của người tiêu dùng. Giờ đây, công ty sẽ cần thu thập dữ liệu từ nhiều nguồn khác nhau như mạng xã hội, bình luận, dữ liệu nhấp chuột, ứng dụng di động của khách hàng và nhiều hơn nữa. Tổ chức của bạn có thể sử dụng `HDFS` để lưu trữ dữ liệu và tổ `Apache Hive` làm cầu nối giữa `HDFS` và `Spark`.

Installation

```
# innstall java
!apt-get install openjdk-8-jdk-headless -qq > /dev/null
```

```
# install spark (change the version number if needed)
!wget -q https://archive.apache.org/dist/spark/spark-3.0.0/spark-
3.0.0-bin-hadoop3.2.tgz

# unzip the spark file to the current folder
!tar xf spark-3.0.0-bin-hadoop3.2.tgz

# install findspark and pyspark using pip
!pip install -q findspark
!pip install -q pyspark
```

1. Launch a Spark app

```
from pyspark.sql import functions as sf
from pyspark.sql import SparkSession
from pyspark import SparkConf, SparkContext
from pyspark.sql.window import Window
```

1.1. Use `SparkSession`

```
import findspark
findspark.init()

spark = SparkSession.builder.master("local[*]").getOrCreate()
spark.conf.set("spark.sql.repl.eagerEval.enabled", True) # Property
used to format output tables better
spark
```

```
24/03/14 13:13:39 WARN SparkSession: Using an existing Spark session;
only runtime SQL configurations will take effect.

<pyspark.sql.session.SparkSession at 0x7f69574081f0>
```

- **Change AppName**

```
app_name = "nhan"
spark.sparkContext.appName = app_name
spark
```

```
<pyspark.sql.session.SparkSession at 0x7f69574081f0>
```

1.2. Use `SparkConf`

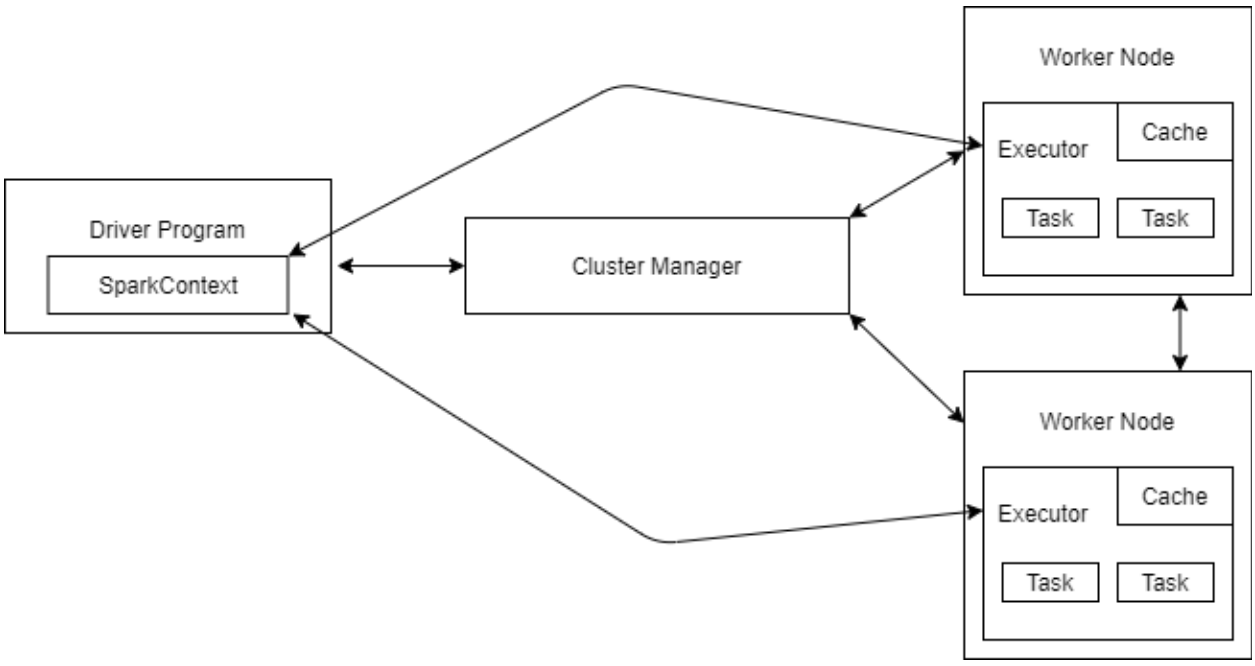
```
conf = SparkConf().setAppName('TA').set("spark.driver.memory", "1g")\
                                   .set("spark.executor.memory",
"12g")\
                                   .set('spark.driver.cores', '2')\
                                   .set('spark.driver.cores', '2')\
                                   .set('spark.dynamicAllocation.enabl
```

```
ed', 'true')\
                                    .set('spark.executor.instances',
'0')\
                                    .set('spark.dynamicAllocation.initi
alExecutors', '0')\
                                     .set('spark.jars', 'gs://spark-
lib/bigquery/spark-bigquery-latest_2.12.jar')
spark_new = SparkSession.builder.config(conf=conf).getOrCreate()
spark_new
```

```
24/03/14 14:19:35 WARN SparkSession: Using an existing Spark session;
only runtime SQL configurations will take effect.
```

```
<pyspark.sql.session.SparkSession at 0x7f69574081f0>
```

2. How to optimize executor on sparks



There are three main aspects to look out for to configure your Spark Jobs on the cluster – `number of executors`, `executor memory`, and `number of cores`.

- An `executor` is a single `JVM (Java Virtual Machine)` process that is launched for a spark application on a node while a core is a basic computation unit of `CPU` or concurrent tasks that an executor can run.
- A `node` can have multiple `executors` and `cores`. All the computation requires a certain amount of memory to accomplish these tasks.

You might think more about the number of cores you have more concurrent tasks you can perform at a given time. While this ideology works but there is a limitation to it.

- It is observed that many spark applications with **more than 5 concurrent tasks are sub-optimal and perform badly**. This number came from the ability of the executor and not from how many cores a system has.
- So the number 5 stays the same even if you have more cores in your machine. So setting this to 5 for good `HDFS` throughput (by setting `executor-cores` as 5 while submitting `Spark application`) is a good idea.

  ◇ When you run spark applications using a `Cluster Manager`, there will be several `Hadoop` daemons that will run in the background like `name node`, `data node`, `job tracker`, and `task tracker` (they all have a particular job to perform which you should read). So, while specifying `num-executors`, you need to make sure that you leave aside enough cores (`~1` core per `node`) for these daemons to run **smoothly**.

  ◇ Also, you will have to leave at least 1 executor for the Application Manager to negotiate resources from the Resource Manager. You will also have to assign some executor memory to compensate for the overhead memory for some other miscellaneous tasks. Literature shows assigning it to about 7-10% of executor memory is a good choice however it shouldn't be too low.

- For example, suppose you are working on a `10 nodes cluster` with `16 cores per node` and `64 GB RAM per node`. You can assign `5 cores per executor` and leave `1 core per node` for Hadoop daemons. So now you have `15 as the number of cores available per node`. Since you have `10 nodes`, the `total number of cores` available will be `10×15 = 150`.

  Now

```
                the number of available executors = total cores/cores
per executor = 150/5 = 30,
```

  but you will have to leave `at least 1 executor` for Application Manager hence the `number of executors will be 29`.

  Since you have `10 nodes`, you will have `3 (30/10) executors per node`.

  The memory per executor will be `memory per node/executors per node = 64/2 = 21GB`.

  Leaving aside `7% (~3 GB) as memory overhead`, you will have `18 (21-3) GB per executor as memory`. Hence finally your parameters will be:

```
                    executor-cores 5, –num-executors 29, –executor-
memory 18 GB.
```

Like this, you can work out the math for assigning these parameters. Although do note that this is just one of the ways to assign these parameters, it may happen that your job may get tuned at different values but the important point to note here is to have a structured way to think about tuning these values rather than shooting in the dark.

Avoid Long Lineage

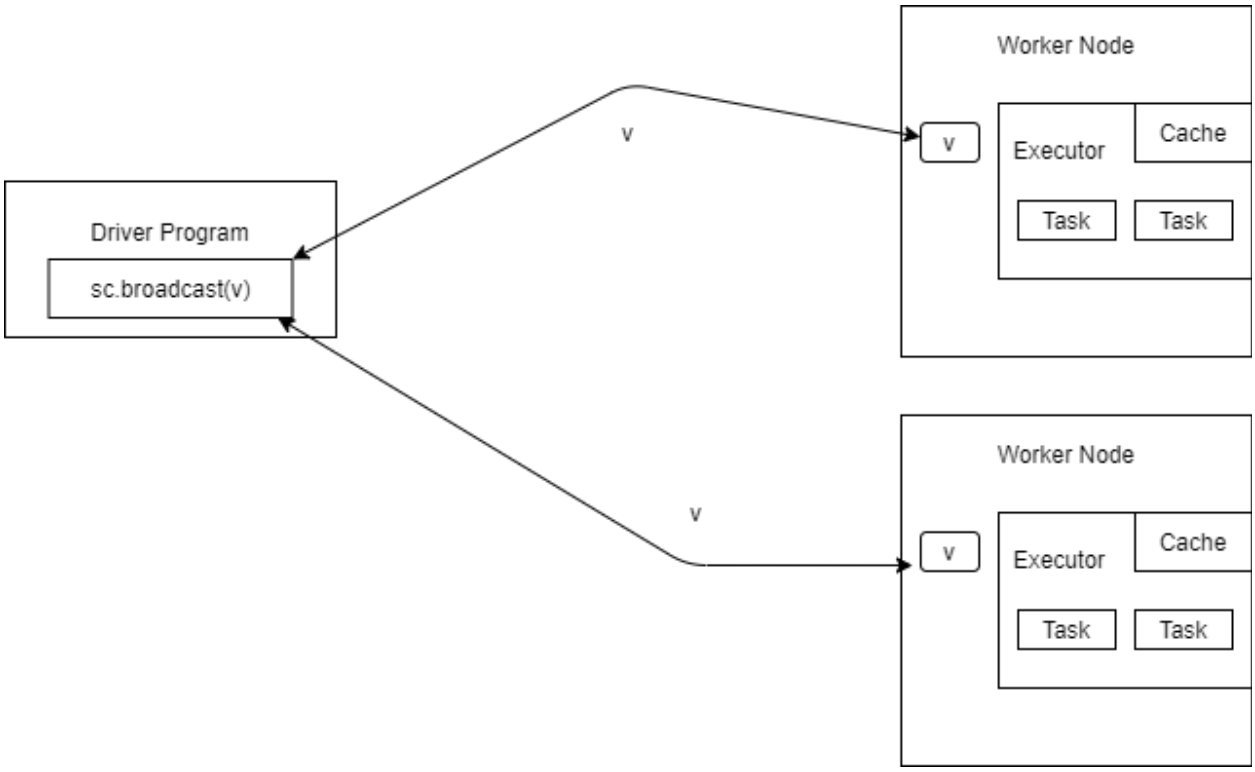
Spark offers two types of operations: Actions and Transformations.

Transformations (eg. map, filter,groupBy, etc.) construct a new RDD/DataFrame from a previous one, while Actions (e.g. head, show, write, etc.) compute a result based on an RDD/DataFrame, and either return it to the driver program or save it to the external storage system. Spark does all these operations lazily. Lazy evaluation in spark means that the actual execution does not happen until an action is triggered. Every transformation command run on spark DataFrame or RDD gets stored to a lineage graph.

It is not advised to chain a lot of transformations in a lineage, especially when you would like to process huge volumes of data with minimum resources. Rather, break the lineage by writing intermediate results into HDFS (preferably in HDFS and not in external storage like S3 as writing on external storage could be slower).

Broadcasting

When a variable needs to be shared across executors in Spark, it can be declared as a broadcast variable. Note the broadcast variables are read-only in nature. Broadcast variables are particularly useful in case of skewed joins. For example, if you are trying to join two tables one of which is very small and the other very large, then it makes sense to broadcast the smaller table across worker nodes' executors to avoid the network overhead.

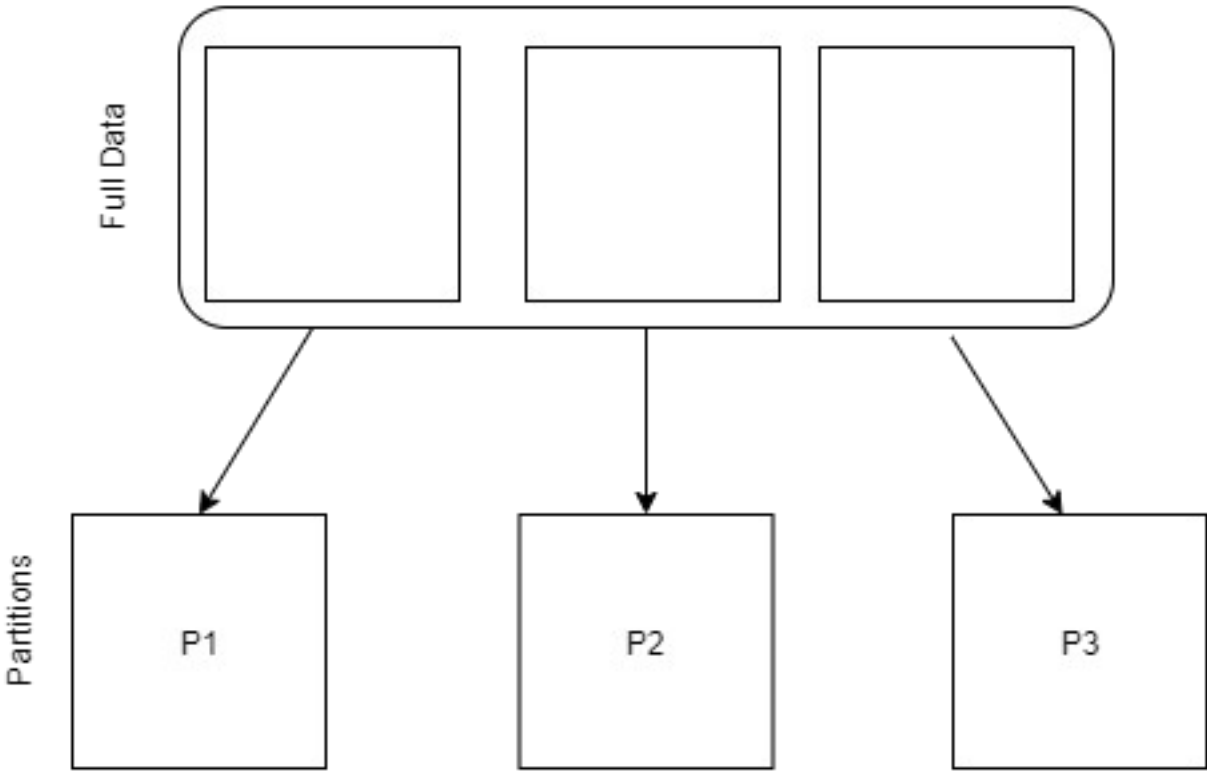


Partitioning your DataSet

While Spark chooses good reasonable defaults for your data, if your Spark job runs out of memory or runs slowly, bad partitioning could be at fault.

If your dataset is large, you can try repartitioning (using the repartition method) to a larger number to allow more parallelism on your job. A good indication of this is if in the Spark UI you

don't have a lot of tasks, but each task is very slow to complete. On the other hand, if you don't have that much data and you have a ton of partitions, the overhead of having too many partitions can also cause your job to be slow. You can repartition to a smaller number using the coalesce method rather than the repartition method as it is faster and will try to combine partitions on the same machines rather than shuffle your data around again.



Columnar File Formats

Spark utilizes the concept of Predicate Push Down to optimize your execution plan. For example, if you build a large Spark job but specify a filter at the end that only requires us to fetch one row from our source data, the most efficient way to execute this is to access the single record that you need. Spark will actually optimize this for you by pushing the filter down automatically.

Columnar file formats store the data partitioned both across rows and columns. This makes accessing the data much faster. They are much more compatible in efficiently using the power of Predicate Push Down and are designed to work with the MapReduce framework. Some of the examples of Columnar file formats are Parquet, ORC, or Optimized Row-Column, etc. Use

Parquet format wherever possible for reading and writing files into HDFS or S3, as it performs well with Spark.

Use DataFrames/Datasets instead of RDDs :

Resilient Distributed Dataset or RDD is the basic abstraction in Spark. RDD is a fault-tolerant way of storing unstructured data and processing it in the spark in a distributed manner. In older versions of Spark, the data had to be necessarily stored as RDDs and then manipulated, however, newer versions of Spark utilizes DataFrame API where data is stored as DataFrames or Datasets. DataFrame is a distributed collection of data organized into named columns, very much like DataFrames in R/Python. Dataframe is much faster than RDD because it has metadata (some information about data) associated with it, which allows Spark to optimize its query plan. Since the creators of Spark encourage to use DataFrames because of the internal optimization you should try to use that instead of RDDs.