By Sai Kumar

SPARK DIAGNOSING AND TUNING

RUNNING ON YARN MODE

Advantages: Scheduler assignment, MPP, Fault tolerant, better executor management

SPARK UIs

The web UI is run by the Spark driver

– When running locally: **http://localhost:4040**

– When running in client mode: **http://gateway:4040**

– When running in cluster mode, access via the **YARN UI**

The Spark UI is only available while the application is running

Use the Spark application **history server** to view metrics for a completed application.

Note: In YARN mode Spark UI is accessed from Application master

Spark Application Configuration Properties

Spark provides numerous properties to configure your application

Some example properties

– **spark.master**: Cluster type or URI to submit application to

– **spark.app.name**: Application name displayed in the Spark UI

– **spark.submit.deployMode**: Whether to run application in client or cluster mode (default: client)

– **spark.ui.port:** Port to run the Spark Application UI (default 4040)

– **spark.executor.memory**: How much memory to allocate to each Executor (default 1g)

– **spark.pyspark.python**: Which Python executable to use for Pyspark Applications

For all configurations:
https://spark.apache.org/docs/latest/configuration.html

Setting Configuration Properties

Most properties are set by system administrators

Managed manually or using Cloudera Manager

Stored in a properties file

Developers can override system settings when submitting applications by

- ✓ **Using submit script flags**
- ✓ **Loading settings from a custom properties file instead of the system file**
- ✓ **Setting properties programmatically in the application**

Properties that are not set explicitly use Spark default values

Overriding Properties Using Submit Script

Some Spark submit script flags set application properties

For example

Use **--master** to set spark.master

Use **--name** to set spark.app.name

Use **--deploy-mode** to set spark.submit.deployMode

Not every property has a corresponding script flag

Use **--conf** to set any property

**spark-submit **

--conf spark.pyspark.python=/alt/path/to/python

Setting Properties in a Properties File

System administrators set system properties in properties files

You can use your own custom properties file instead

spark.master   local[*]

spark.executor.memory   512k

spark.pyspark.python   /alt/path/to/python

Specify your properties file using the properties-file option

```
spark-submit \
--properties-file=dir/my-properties.conf
```

▪Note that Spark will load only your custom properties file

System properties file is ignored

Copy important system settings into your custom properties file

Custom file will not reflect future changes to system settings

Setting Configuration Properties Programmatically

Spark configuration settings are part of the Spark session or Spark context

Set using the Spark session builder functions

appName sets spark.app.name

master sets spark.master

config can set any property

```
import org.apache.spark.sql.SparkSession
...
val spark = SparkSession.builder.
appName("my-spark-app").
config("spark.ui.port","5050").
getOrCreate()
```

Priority of Spark Property Settings

Properties set with higher priority methods override lower priority methods

1. Programmatic settings

2. Submit script (command line) settings

3. Properties file settings

– Either administrator site-wide file or custom properties file

4. Spark default settings

Viewing Spark Properties

You can view the Spark property settings two ways

Using --verbose with the submit script

In the Spark Application UI Environment tab

Partitioning from Data in Files

Partitions are determined when files are read

Core Spark determines RDD partitioning based on location, number, and size of files

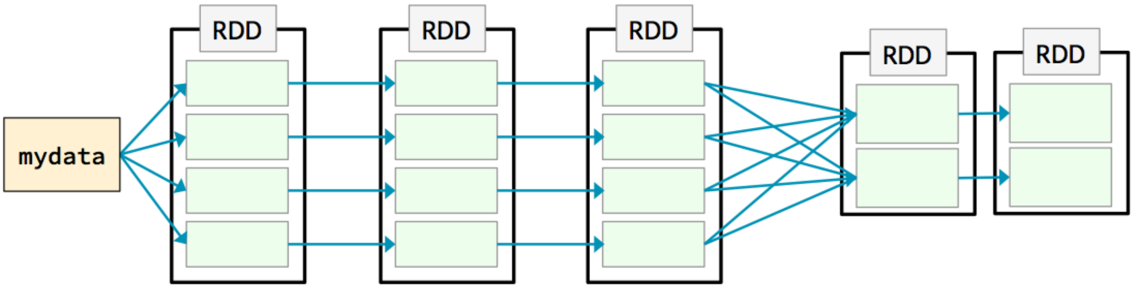
Usually, each file is loaded into a single partition

Very large files are split across multiple partitions

Catalyst optimizer manages partitioning of RDDs that implement DataFrames and Datasets

Example: Average Word Length by Letter

```
avglens = sc.textFile(mydata) \

.flatMap(lambda line: line.split(' ')) \

.map(lambda word: (word[0],len(word))) \

.groupByKey() \

.map(lambda (k, values): \

(k, sum(values)/len(values)))
```



Stages and Tasks

A **task** is a series of operations that work on the same partition and are pipelined together

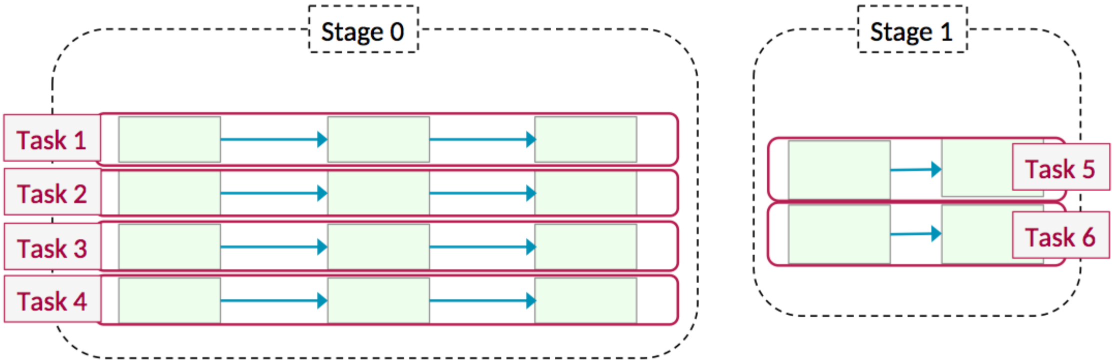
Stages group together tasks that can run in parallel on different partitions of the same RDD

Jobs consist of all the stages that make up a query

Catalyst optimizes partitions and stages when using DataFrames and Datasets — Core Spark provides limited optimizations when you work directly with RDDs

You need to code most RDD optimizations manually

To improve performance, be aware of how tasks and stages are executed when working with RDDs



Summary of Spark Terminology

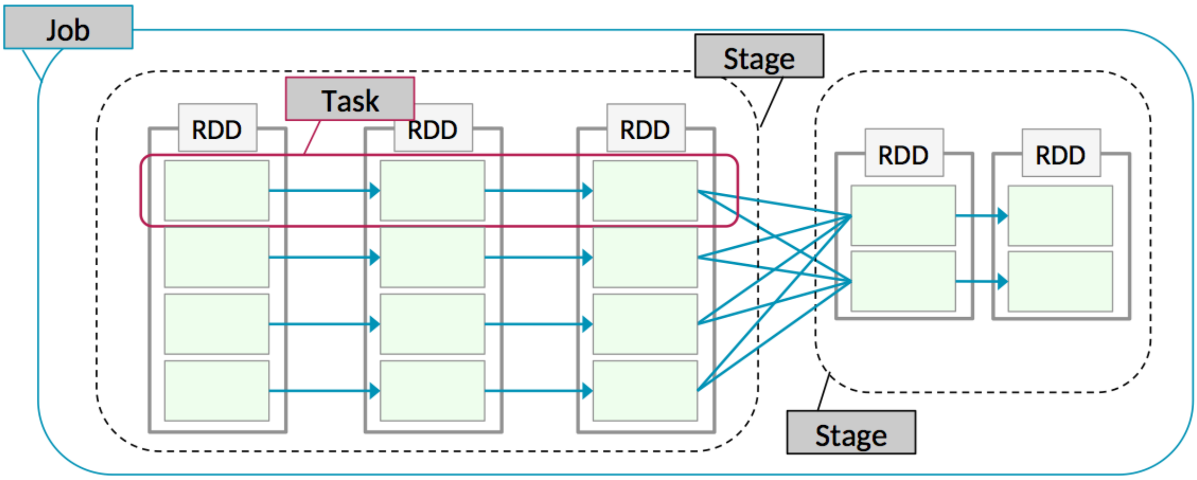
Job—a set of tasks executed as a result of an action

Stage—a set of tasks in a job that can be executed in parallel

Task—an individual unit of work sent to one executor

By Sai Kumar

Application—the set of jobs managed by a single driver



Execution Plans

Spark creates an execution plan for each job in an application

Catalyst creates SQL, Dataset, and DataFrame execution plans

– **Highly optimized**

Core Spark creates execution plans for RDDs

– **Based on RDD lineage**

– **Limited optimization**

How Execution Plans are Created

Spark constructs a DAG (Directed Acyclic Graph) based on RDD dependencies

Narrow dependencies

– Each partition in the child RDD depends on just one partition of the parent RDD

– No shuffle required between executors

– Can be pipelined into a single stage

– Examples: map, filter, and union

Wide (or shuffle) dependencies

– Child partitions depend on multiple partitions in the parent RDD

– Defines a new stage

– Examples: reduceByKey, join, and groupByKey

Controlling the Number of Partitions in RDDs

Partitioning determines how queries execute on a cluster

– More partitions = more parallel tasks

– Cluster will be under-utilized if there are too few partitions

– But too many partitions will increase overhead without an offsetting increase in performance

Catalyst controls partitioning for SQL, DataFrame, and Dataset queries

You can control how many partitions are created for RDD queries
Specify the number of partitions when data is read

– Default partitioning is based on size and number of the files (minimum is two)

– Specify a different minimum number when reading a file

myRDD = sc.textFile(myfile,5)

Manually repartition

– Create a new RDD with a specified number of partitions using repartition or coalesce

– coalesce reduces the number of partitions without requiring a shuffle

– repartition shuffles the data into more or fewer partitions

newRDD = myRDD.repartition(15)

Specify the number of partitions created by transformations

– Wide (shuffle) operations such as reduceByKey and join repartition data

– By default, the number of partitions created is based on the number of partitions of the parent RDD(s)

– Choose a different default by configuring the

spark.default.parallelism property

spark.default.parallelism 15

– Override the default with the optional numPartitions operation parameter
**countRDD = wordsRDD. **

reduceByKey(lambda v1, v2: v1 + v2, 15)

Catalyst Optimizer
Catalyst can improve SQL, DataFrame, and Dataset query performance by optimizing the DAG to
Minimize data transfer between executors

Such as broadcast joins—small data sets are pushed to the executors where the larger data sets reside

Minimize wide (shuffle) operations
Such as unioning two RDDs—grouping, sorting, and joining do not require Shuffling
– Pipeline as many operations into a single stage as possible

– Generate code for a whole stage at run time

– Break a query job into multiple jobs, executed in a series

Catalyst Execution Plans
Execution plans for DataFrame, Dataset, and SQL queries include the following phases

Parsed logical plan—calculated directly from the sequence of operations specified in the query

Analyzed logical plan—resolves relationships between data sources and columns

Optimized logical plan—applies rule-based optimizations

Physical plan—describes the actual sequence of operations

Code generation—generates bytecode to run on each node, based on a cost model.

By Sai Kumar

Viewing Catalyst Execution Plans

You can view SQL, DataFrame, and Dataset (Catalyst) execution plans

Use DataFrame/Dataset explain

Shows only the physical execution plan by default, pass true to see the full execution plan

Use SQL tab in the Spark UI or history server

Shows details of execution after job runs

empDF=spark.read.option("header","true").option("inferschema","true").csv("/user/root/EMP.csv")

deptDF=spark.read.option("header","true").option("inferschema","true").csv("/user/root/dept.csv")

empdeptDF=empDF.join(deptDF,"deptno")

empdeptDF.explain(True)

WITH JOIN HINT

sortmergeJoin = empDF.hint("merge").join(deptDF,"deptno")

Viewing RDD Execution Plans

You can view RDD (lineage-based) execution plans

Use the RDD **toDebugString** function

Use Jobs and Stages tabs in the Spark UI or history server

Shows details of execution after job runs

Note that plans may be different depending on programming language

Plan optimization rules vary

val peopleRDD = sc.textFile("people2.csv").keyBy(s => s.split(',')(0))

val pcodesRDD = sc.textFile("pcodes2.csv").keyBy(s => s.split(',')(0))

val joinedRDD = peopleRDD.join(pcodesRDD)

joinedRDD.toDebugString

Persistence

You can persist a DataFrame, Dataset, or RDD

Also called caching

Data is temporarily saved to memory and/or disk

Persistence can improve performance and fault-tolerance

Use persistence when

- ✓ Query results will be used repeatedly
- ✓ Executing the query again in case of failure would be very expensive.

Persisted data cannot be shared between applications

Table and View Persistence

Tables and views can be persisted in memory using CACHE TABLE

spark.sql("CACHE TABLE people")

CACHE TABLE can create a view based on a SQL query and cache it at the same time
**spark.sql("CACHE TABLE over_20 AS SELECT *
FROM people WHERE age > 20")**

Queries on cached tables work the same as on persisted DataFrames, Datasets, and RDDs
The first query caches the data

Subsequent queries use the cached data

Storage Levels

Storage levels provide several options to manage how data is persisted

Storage location (memory and/or disk)

Serialization of data in memory

Replication

Specify storage level when persisting a DataFrame, Dataset, or RDD

Tables and views do not use storage levels

Always persisted in memory

Data is persisted based on partitions of the underlying RDDs

Executors persist partitions in JVM memory or temporary local files

The application driver keeps track of the location of each persisted partition's data

Storage Levels: Location

Storage location—where is the data stored?

MEMORY_ONLY: Store data in memory if it fits

DISK_ONLY: Store all partitions on disk

MEMORY_AND_DISK: Store any partition that does not fit in memory on disk called spilling

Language: Python

from pyspark import StorageLevel

myDF.persist(StorageLevel.DISK_ONLY)

Language: Scala

import org.apache.spark.storage.StorageLevel

myDF.persist(StorageLevel.DISK_ONLY)

Storage Levels: Partition Replication

Replication—store partitions on two nodes

DISK_ONLY_2

MEMORY_AND_DISK_2

MEMORY_ONLY_2

MEMORY_AND_DISK_SER_2 (Scala and Java only)

MEMORY_ONLY_SER_2 (Scala and Java only)

You can also define custom storage levels for additional replication

By Sai Kumar

Default Storage Levels

The storageLevel parameter for the DataFrame, Dataset, or RDD persist operation is optional

The default for DataFrames and Datasets is MEMORY_AND_DISK

The default for RDDs is MEMORY_ONLY

persist with no storage level specified is a synonym for cache

myDF.persist() is equivalent to myDF.cache()

Table and view storage level is always MEMORY_ONLY

When and Where to Persist

- ✓ When should you persist a DataFrame, Dataset, or RDD?
- ✓ When the data is likely to be reused
- ✓ Such as in iterative algorithms and machine learning
- ✓ When it would be very expensive to recreate the data if a job or node fails.

How to choose a storage level

Memory—use when possible for best performance

Save space by serializing the data if necessary

Disk—use when re-executing the query is more expensive than disk read

Such as expensive functions or filtering large datasets

Replication—use when re-execution is more expensive than bandwidth

Changing Storage Levels

You can remove persisted data from memory and disk

Use **unpersist** for Datasets, DataFrames, and RDDs

Use **Catalog.uncacheTable(table-name)** for tables and views

Call with no parameter to uncache all tables and views

Unpersist before changing to a different storage level

Re-persisting already-persisted data results in an exception

myDF.unpersist()

myDF.persist(new-level)