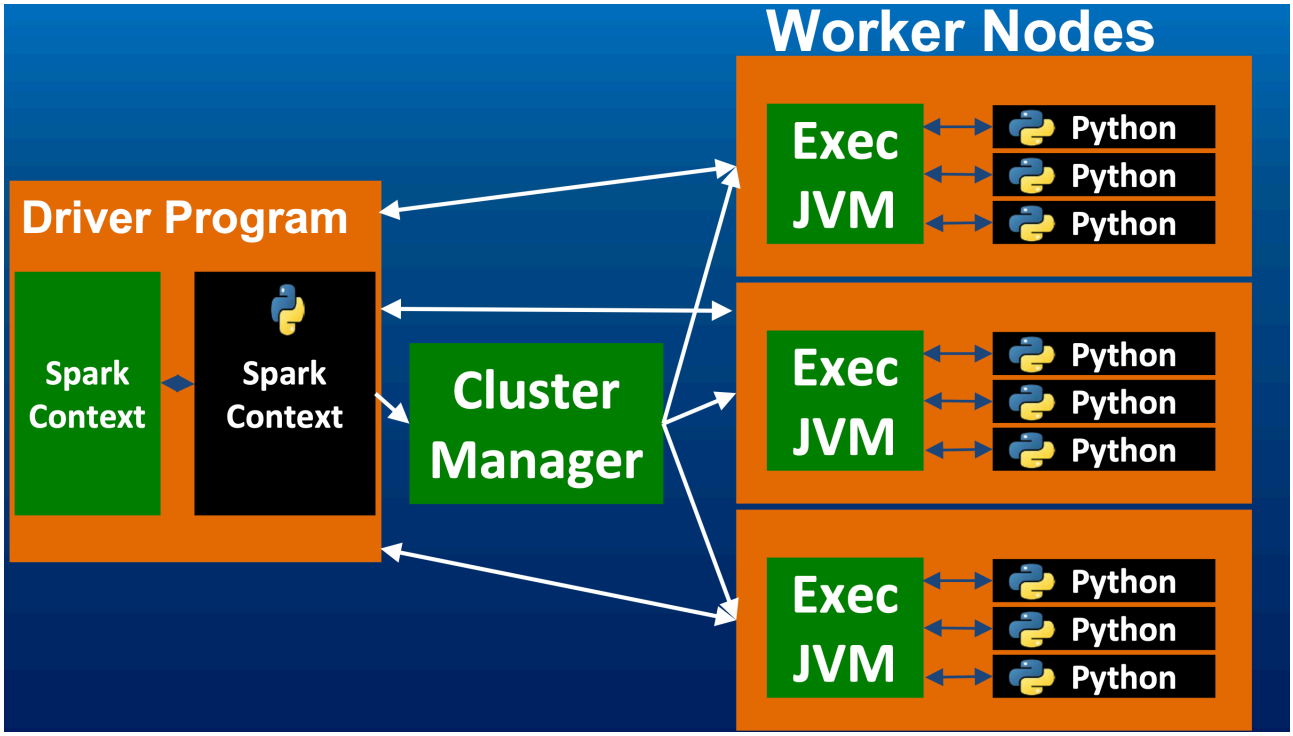
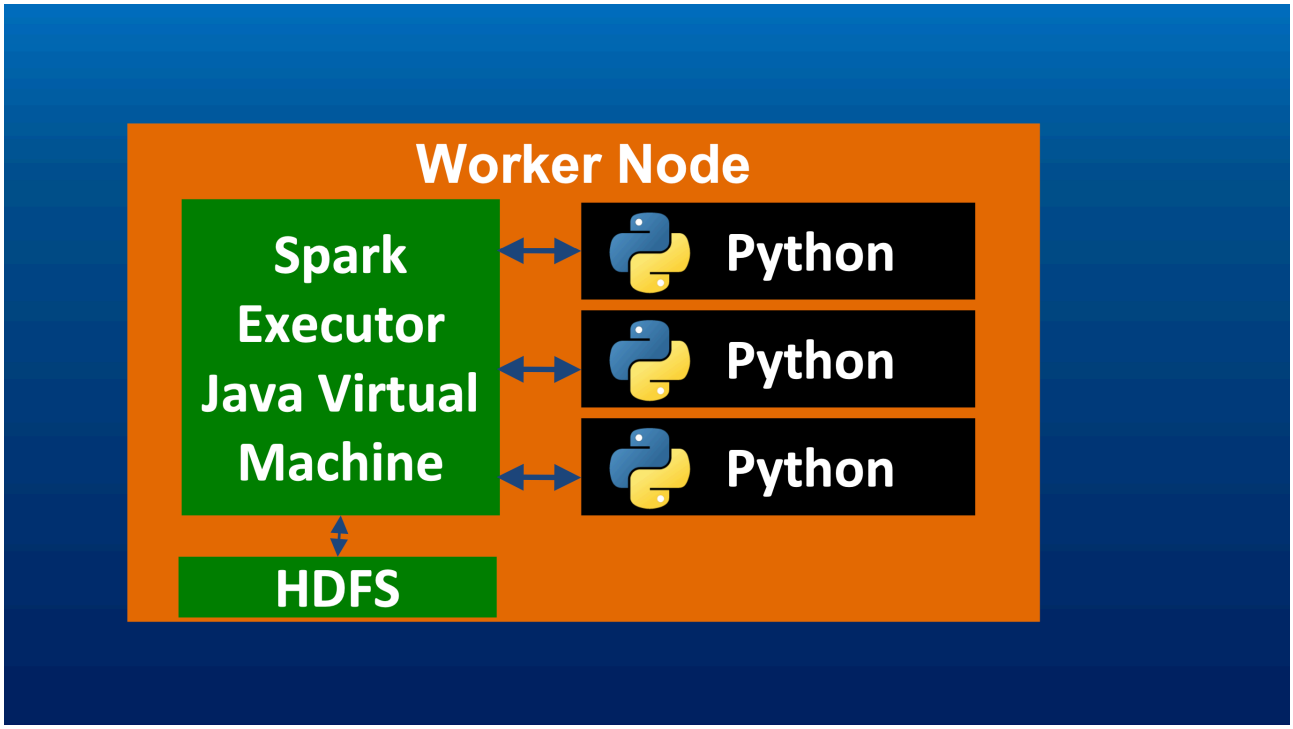
Apache Spark Notes

Spark Architecture

Il Driver Program chiede a YARN (Cluster Manager) di distribuire copie del container originale con il nostro eseguibile sui Worker Nodes. Ogni container si occupera' di fare mapreduce sulla frazione di dati di sua competenza (Big Data = porto l'elaborazione verso i dati, e non viceversa come avviene in cluster non scalabili orizzontalmente es. Oracle RAC)

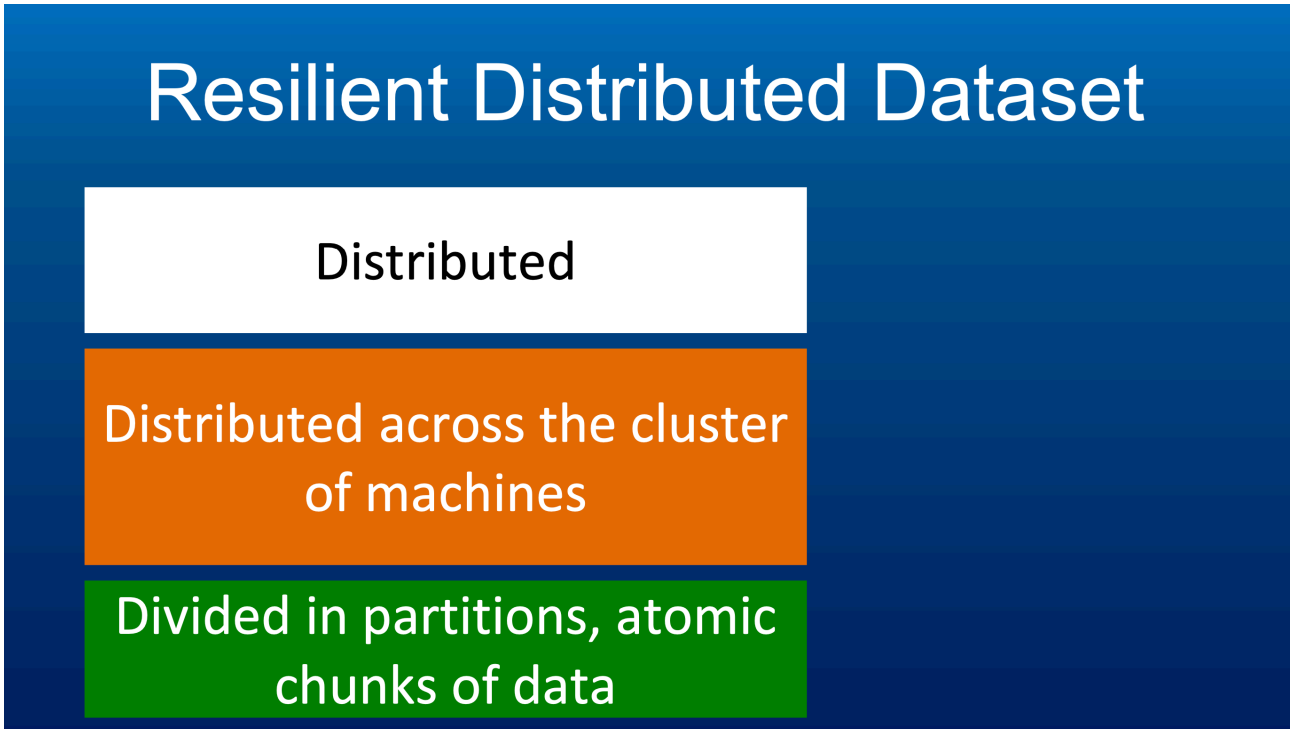


Una JVM con una copia del programma viene dislocata sui nodi con sufficienti risorse. Vengono creati tanti thread quanti sono i Virtual Cores che abbiamo richiesto. La JVM fa il lavoro pesante di I/O da e verso HDFS, ogni thread fa una parte del lavoro di Transformation

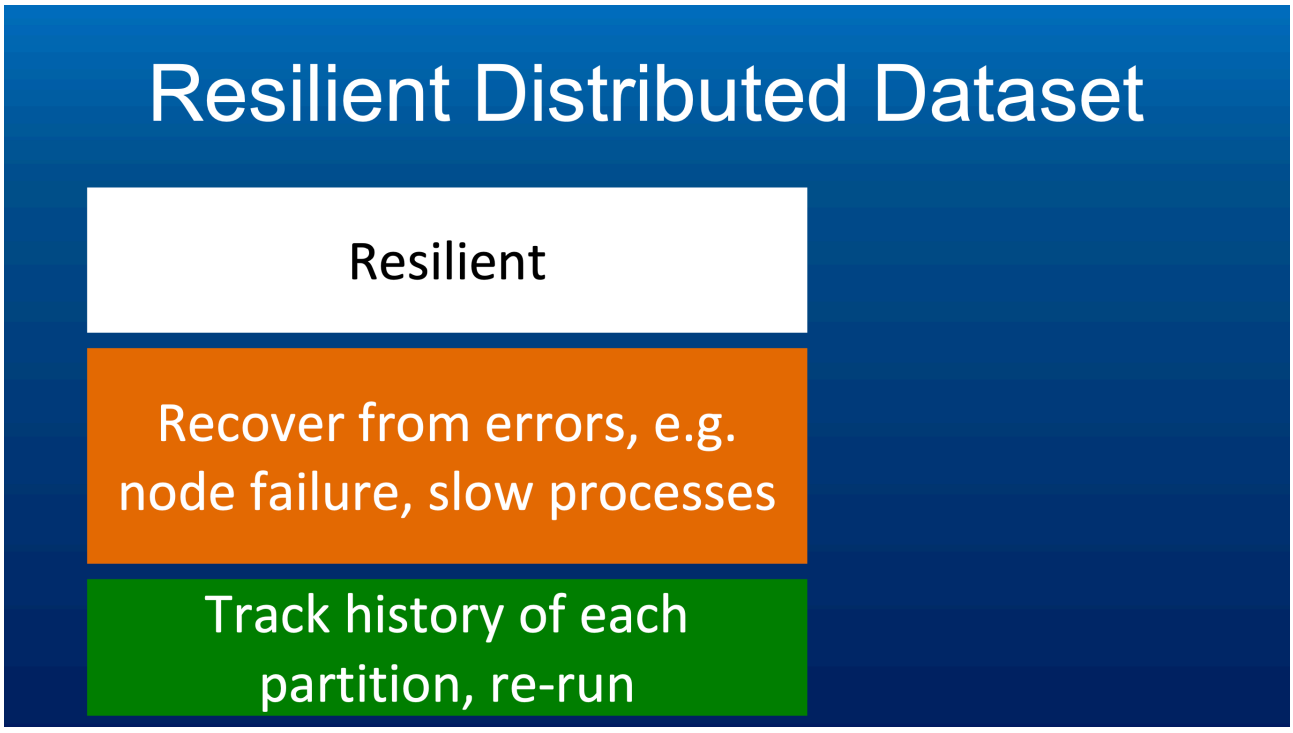


Resilient Distributed Dataset (RDD)

Gli RDD (Resilient Distributed Dataset) sono la struttura dati su cui si basa il funzionamento di Spark



Le caratteristiche e i punti di forza degli RDD sono pochi, ma di fondamentale importanza in un ambiente di computing distribuito



Spark Dataframes

Gli Spark Dataframes sono un'astrazione degli RDD che permette di operare facilmente sui dati, particolarmente su dataset Tabellari (molti campi e valori invece che singole key/value pairs come in definitiva si possono riassumere gli RDD). Questo rende trasparente allo sviluppatore i dettagli del paradigma mapreduce che governano il flusso di dati "under the hood", e che permette di concentrarsi sullo scrivere poche istruzioni molto potenti senza "perdere tempo" in dettagli sistemistici

```
students_df = sqlCtx.createDataFrame(students,
    ["id", "name", "grade", "degree"])

students_df.printSchema()
root
 |-- id: long (nullable = true)
 |-- name: string (nullable = true)
 |-- grade: double (nullable = true)
 |-- degree: string (nullable = true)
```

I Dataframes si creano e operano all'interno dello Spark SqlContext, che e' un'estensione dello SparkContext in cui di solito operiamo con gli RDD "nudi e crudi"

```
sqlCtx.createDataFrame?
Create a DataFrame from an RDD of tuple/list, list or pandas.DataFrame.

`schema` could be :class:`StructType` or a list of column names.

When `schema` is a list of column names, the type of each column
will be inferred from `rdd`.

When `schema` is None, it will try to infer the column name and type
from `rdd`, which should be an RDD of :class:`Row`, or namedtuple,
or dict.

If referring needed, `samplingRatio` is used to determined how many
rows will be used to do referring. The first row will be used if
`samplingRatio` is None.

:param data: an RDD of Row/tuple/list/dict, list, or pandas.DataFrame
:param schema: a StructType or list of names of columns
:param samplingRatio: the sample ratio of rows used for inferring
:return: a DataFrame

>>> l = [('Alice', 1)]
>>> sqlCtx.createDataFrame(l).collect()
[Row(_1=u'Alice', _2=1)]
>>> sqlCtx.createDataFrame(l, ['name', 'age']).collect()
[Row(name=u'Alice', age=1)]
```

Spark SQL

Una delle possibilita' offerte dagli Spark Dataframes e' di usare direttamente codice SQL su un Dataframe, comunque esso sia stato definito

DataFrames & SQL

- Query existing Spark DataFrames (however created) with SQL
  - Same functionality, different interface
  - Legacy SQL

Ad esempio, se avessimo creato un Dataframe "yelp" a partire da un RDD, oppure registrando qualsiasi tipo di Tabella Esterna, noi potremmo eseguire una query come la seguente

Run SQL statements

```
filtered_yelp = sqlCtx.sql("SELECT * FROM yelp WHERE
useful >= 1")
filtered_yelp

Out[]: DataFrame[business_id: string, cool: int, date: string,
funny: int, id: string, stars: int, text: string, type: string,
useful: int, user_id: string, name: string, full_address:
string, latitude: double, longitude: double, neighborhoods:
string, open: string, review_count: int, state: string]
```

L'approccio descritto sopra non e' dei piu' performanti, in quanto si basa su ottimizzatori che non danno il pieno controllo, soprattutto se si fanno operazioni complesse. Ma sicuramente il grosso vantaggio e' che stiamo gia' lavorando sfruttando pienamente un motore di computing parallelo, il tutto scrivendo del classico codice SQL. Siamo gia' un gradino sopra HiveQL, perche' stiamo gia' sfruttando il motore Spark (descritto nelle slide precedenti) che ha performance molto superiori a Hive in tutti i benchmark

Performance

- Either DataFrame calls or SQL
- Same under-the-hood optimizer (Catalyst)
- Creates DAG
- Parallel execution
- Creates bytecode

L'accesso diretto ad una Tabella Hive e' assolutamente identico alla Query vista poc'anzi per un Dataframe generico che sia stato preventivamente registrato come Temp Table. Un esempio infatti potrebbe essere il seguente

Hive table to DataFrame

- sqlCtx.sql has access to Hive tables
- Load data uploaded during the Hive class
- Result is a DataFrame

```
customers_df = sqlCtx.sql("SELECT * FROM customers")
customers_df.show()
```

Allegati e proseguimento della discussione

Nella cartella “docs” si trovano versioni piu’ estese dei doc da cui sono state tratte le immagini precedenti, che danno piu’ dettagli sui concetti presentati

Nella cartella “code” si trovano invece diversi snippet di codice per fare prove pratiche con particolare enfasi su query e trasformazioni usando le funzioni dedicate di Spark, avendo piu’ controllo sul flusso delle aggregazioni/trasformazioni e quindi sulle performance, ma soprattutto che mettono in grado di integrare il tutto con codice python che permette di eseguire operazioni non possibili con SparkSQL oppure con HiveQL, ad es. UDF (User Defined Functions), Windows Functions, loop sui Dataframes evitando la crescita incontrollata dell’Execution Plan DAG ad ogni ciclo, caching misto memoria/disco ecc

iPython PySpark CLI vs. “plain” Python Spark CLI

Nei nostri esempi useremo il driver iPython quando opereremo con la CLI di Spark

iPython permette operazioni aggiuntive che possono venire utili:

- Aprire finestre di output grafico che appaiono direttamente sul nostro Client
- Incollare snippet di codice, correttamente indentato, direttamente nella CLI senza incorrere in fastidiosi problemi di formattazione che causano errori

Per lanciare la CLI PySpark usando il driver iPython, come utente “hdfs” il comando da eseguire potrebbe essere simile al seguente:

```
PYSPARK_DRIVER_PYTHON=ipython pyspark \
--executor-memory 2G \
--executor-cores 2 \
--num-executors 4 \
--driver-memory 4G \
--master=yarn
```