

Wes Bethel

SPARK

Notes



Big Data → Grande Volume / Grande Varietà / Grande Velocità

3V

MapReduce → modello di programmazione che permette di processare parallelamente ed in modo distribuito un'ormai grande量ità di dati.

Spark → open source → distribuito → Memory cache
Minimize le query.

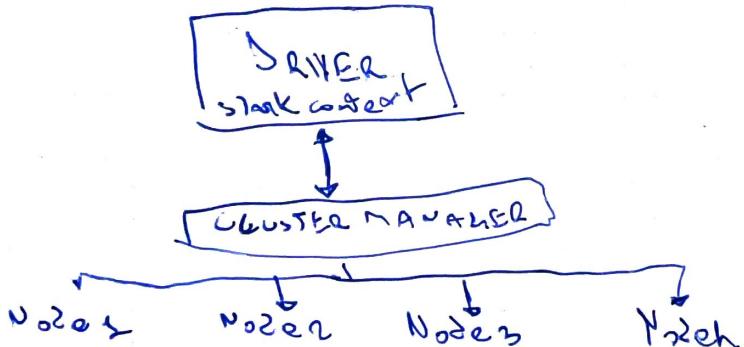
Execution plan in memory

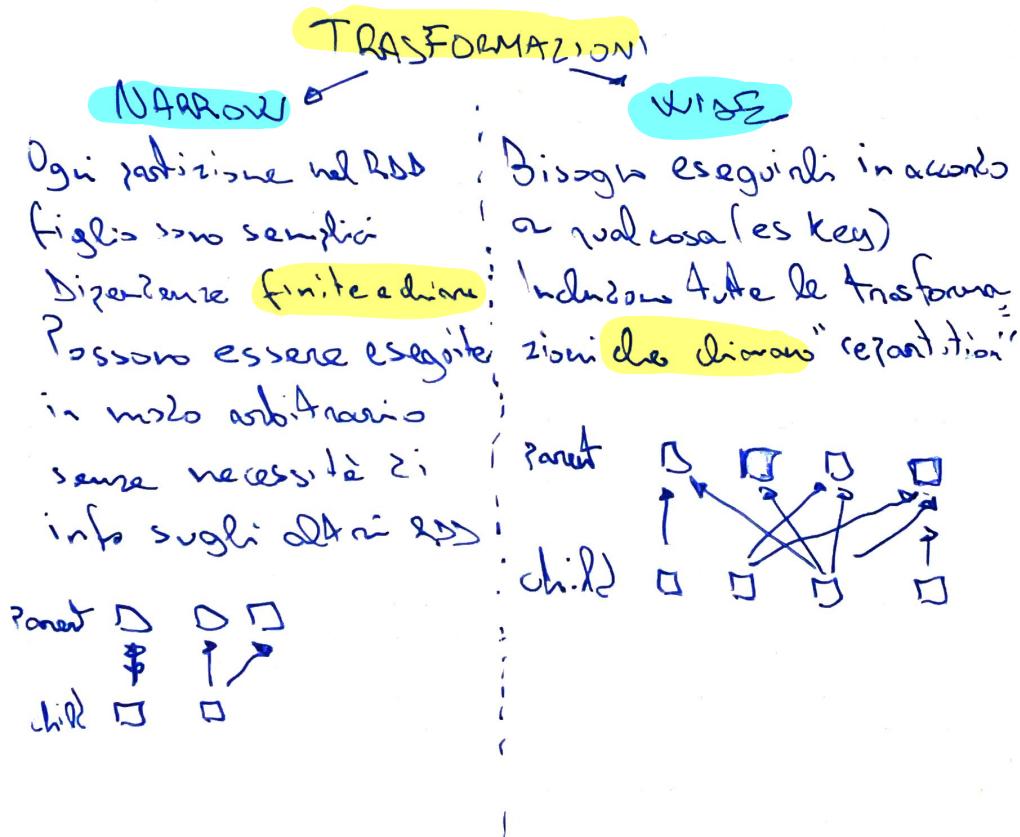
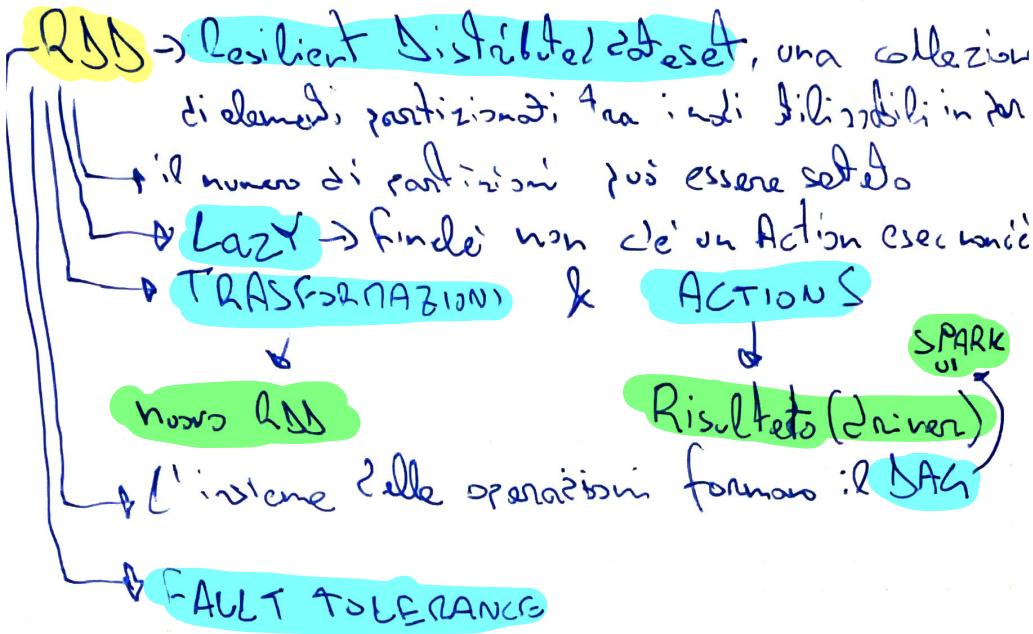
Python, Scala, Java, R

Batch, Real Time, ML, Graph

4 moduli: MLlib, Streaming, SQL, GraphX

Un'applicazione spark running in parallelizzata su un cluster coordinato dal SPARK CONTEXT





TRANSFORMAZIONI GRUNDI

MAP → Nuovo RDD con mapping

FILTER → Nuovo RDD con elementi che rispettano cond.

FLATMAP → Igualde a MAP, ma ogni input → N output

MAPPARTITIONS → Map che ruota sulle diverse partizioni

REPARTITIONNWITHINDEX → divide + indice partizione

SAMPLE → Prende un sample dall'RDD di input

UNION → elementi nel RDD più quelli nell'altro

INTERSECTION → Sceglie in comune input da entrambi

DISTINCT → Distinct delle parti

GROUPBYKEY → $(k, v) \rightarrow (k, \text{iterable}(v))$

REDUCEBYKEY → Per ogni k ritorna un solo valore

AGGREGATEBYKEY → Per ogni k si riporta un solo valore ^{inclusivo}

SORTBYKEYS → $(k, v) \rightarrow (k, v)$ ordinando chiave

SQRT → Classico SQL $\sqrt{k(v)} \rightarrow (k, v)$ $\sqrt{(k, v)} \rightarrow (k, v)$

GROUP → $(k, v) \rightarrow (k, \text{iterable}(v)) \rightarrow (k, \text{iterable}(v), \text{iterable}(f(v)))$

CARTESIAN → Prodotti cartesiani fra 2 RDD

PIPE → esegue comando shell su RDD (output stringa)

COALESCE → Dividere numero di partizioni

REPARTITION → Bilancia le partizioni, crea nuove

REPARTITIONANDSORTWITHPARTITIONS → v.p. 207va

ACTIONS COMUNI

REDUCE → Aggiunge elementi usando una funzione

COLLECT → Ritorna tutto il dataset al driver sotto forma di array

COUNT → Numero degli elementi

FIRST → Primo elemento

TAKE → Primi N elementi di un RDD

TAKE SAMPLE → Array di sample del dataset

TAKE ORDERED → I primi N elementi ordinati per quale

SAVEAS TEXT FILE → quello decide

SAVE AS SEQUENCE FILE → ^{solo k,v} scrive soltanto i valori segnati fil

SAVE AS OBJECT FILE → Scrive solo una JAVA SERIALIZED

COUNT BY KEY → Per ogni chiave conta

FOR EACH → Per ogni elemento, runna una funzione

SHUFFLE

Avviene quando bisogna redistribuire i dati per reggrupparli intorno alle partizioni. Tipicamente indossa opie di dati attraverso gli executor e rappresentano un'operazione costosa e complessa.

Questi dati vengono distribuiti tra gli executor sotto forma di partizioni. Ignorare queste viene gestito da un executor.

Spesso i dati vengono divisi in partizioni in base ad una **key**, se queste non è bilanciate ci sarà un sovraccarico su una partizione riducendo i tempi di esecuzione di tutto il programma.

Lo shuffle interviene quando i dati vengono riorganizzati tra le partizioni. Spesso quando una trasformazione necessita di informazioni in altre parti delle partizioni. Spark si occupa di raccogliere i dati ridistribuirli a diverse partizioni e combinarli in una nuova.

Solo più lento ed usa più memoria rispetto MapReduce, ma c'è quello di Reducer.

Si trova con: Spill, Partition, Compress, Parallelize, Partitioner.

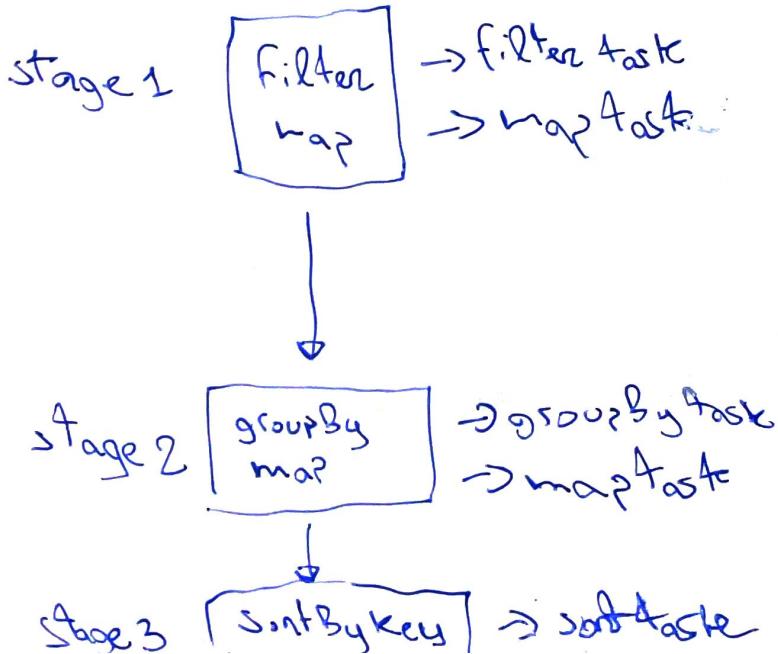
Scheduling

DAG \rightarrow i stages per ogni Spark Job

Job \rightarrow corrisponde ad una azione, ed ogni azione è chiamata dal DRIVER di un'applicazione SPARK.

STAGE \rightarrow Ogni stage corrisponde ad una dipendenza di shuffle o ad una WISE transformation.
Un stage è l'insieme delle operazioni che un'esecuzione può effettuare senza comunicare all'estero.

TASK \rightarrow Compongono un stage. È l'insieme delle operazioni effettuate su una singola partizione singolo record.



SHARED VARIABLE

Broadcast variable permettono di creare una variabile READ-ONLY e condividerla a tutti gli esecutori.

Accumulator sono variabili che permettono solo aggiunte di valori per somme e moltiplicazioni. Sono numerici ma si possono definire nuovi.

SPARK SQL

con l'arrivo di Spark SQL sposta i due mondi
d'interno dati per velocizzare le esecuzioni: DFeS

I Dataframes permettono di fare tutte le possibili
operazioni normalmente fattibili in SQL

Si possono creare tabella temporanee tenendo
in un df per poter fare query sopra.
o la global variable

Stream Processing

Structured Streaming → Based on SparkSQL

Spark Streaming → Based on SparkCore

Problemi con streaming

- Consistenza record possono essere elaborati
in una parte prima che in un'altra fornendo a
risultati senza senso

- Fault Tolerance → Non gestisce l'errore

- Data-afka-like → I dati possono arrivare in
qualsiasi ordine

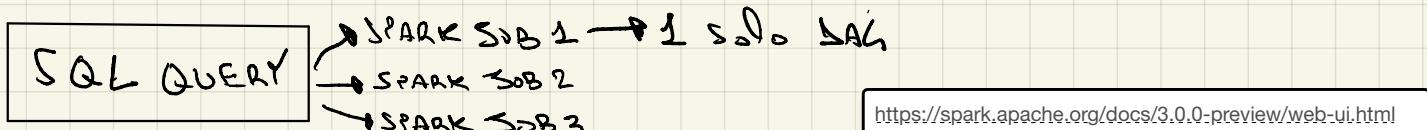
SPARK PLAN

E' possibile leggere il piano di esecuzione SPARK da SPARK UI o da SPARK HISTORY SERVER.

Durante l'esecuzione di un Plan vengono fatti CAST IMPLICITI, è sempre consigliato esplicitarli.

Le Native READER/WRITER performano molto meglio che le SERDE (Velocità e stabilità).

Se una query è corretta tutte le query seguenti saranno impostate.



E' possibile visualizzare negli specifici TAB: JOBS, STAGES, TASKS. In essi si possono visualizzare info

Possibili problemi di performance per il piano di SPARK possono essere:
- Tabelle con migliaia di partizioni (NIVEAU metadati numero)
- 100s di milioni di file (FILE SYSTEM overhead)
- Nuovi Letti non sono visibili, bisogna usare REFRESH

Delta Lake permette di risolvere tutti questi problemi.

Delta Lake + SPARK benefici:

- Completamente ACID
- Schema Management
- Data Versioning e Tracing
- Batch / Streaming unified
- Metadati scalabili
- Record aggiornati a canelli
- Late Expectation

QUESTION

• Nel TASK LIST del STAGE TAB non è possibile vedere:

GC Time ExecutionTime Input Size ShuffleSize CPU Load

• You can read the plan from the [] in the [] tab. - SPARK UI - SQL

• Pushed filter are visible in:

LOGICAL PLAN PHYSICAL PLAN OPTIMIZED PLAN

• In SPARK SQL the physical plan that will be executed is:

the best physical plan chosen by the planner The logical plan selected by optimizer The optimized plan in output from

• From which TAB is possible to execute a Thread Dump? EXECUTION TAB

• A SQL query is corresponding to: MULTIPLE SPARK SDB

• The best way to understand where a stage is spending time is the [stage tab]. There there is the first Timeline.

SPARK TUNING

Quando si crea un'applicazione SPARK bisogna pensare a quali configuration usare:

- executor.memory → esse + l'overhead deve essere minore alla dimensione del container
- driver.memory → più piccolo possibile senza la fallisca.
- executor.vcores
- enable dynamic allocation

Buoni suggerimenti sono:
- ripartire la memoria del nodi tra i vari executors

- preferire pochi grandi executors per nodo
- se hai più piccoli executors ci sono più ritardi di comunicazione
- decidere quanti usare in base al problema.
- 1 core = 1 task, il numero dei task è limitato agli executors, troppi però possono creare problemi.
thread safe
- i solo core per executor fa perdere benefici quali BROADCAST

DYNAMIC ALLOCATION

Percorre a SPARK di aggiungere e sottrarre esecutori ai vari job dell'applicazione.

Si sette Areeute:

```
- To configure
  - spark.dynamicAllocation.enabled=true
  - spark.shuffle.service.enabled=true (you have to configure external shuffle service on each worker)
  - spark.dynamicAllocation.minExecutors
  - spark.dynamicAllocation.maxExecutors
  - spark.dynamicAllocation.initialExecutors

- To Adjust
  - Spark will add executors when there are pending tasks (spark.dynamicAllocation.schedulerBacklogTimeout)
  - and exponentially increase them as long as tasks in the backlog persist (spark...sustainedSchedulerBacklogTimeout)
  - Executors are decommissioned when they have been idle for spark.dynamicAllocation.executorIdleTimeout
```

When

- Most important for shared or cost sensitive environments
- Good when an application contains several jobs of differing sizes
- The only real way to adjust resources throughout an application

Improvements

- If jobs are very short adjust the timeouts to be shorter
- For jobs that you know are large start with a higher number of initial executors to avoid slow spin up
- If you are sharing a cluster, setting max executors can prevent you from hogging it

Quando ci sono problemi di memoria si può provare:

- Se si ha più memoria provare a concederla
- Provare ad incrementare il numero delle partizioni (in una funzione o spark.default.parallelism)
- Se ci sono executors in idle probabilmente debba abbatterli.
- Se tanti sono in idle c'è un'operazione onerosa nel driver.
- Se un'operazione complesse ha errori in IDE aumentare partizioni

Prevenire shuffle spill to disk: -large executors -Configure off heap space -più partition possono aiutare -Settare shuffle settings

Tuning Can Help With

- Low cluster utilization (\$\$\$\$)
- Out of memory errors
- Spill to Disk / Slow shuffles
- GC errors / High GC overhead
- Driver / Executor overhead exceptions
- Reliable deployment

Things you can't "tune away"

- Silly Shuffles
 - You can make each shuffle faster through tuning but you cannot change the fundamental size or number of shuffles without adjusting code
 - Think about how much you are moving data, and how to do it more efficiently
 - Be very careful with column based operations on Spark SQL
- Unbalanced shuffles (caused by unbalanced keys)
 - if one or two tasks are much larger than the others
 - #Partitions is bounded by #distinct keys
- Bad Object management / Data structures choices
 - Care about memory exceptions / memory overhead exceptions / gc errors
 - Serialization exceptions*
 - Python
 - This makes Holden sad, bummer. Arrow?

Il tuning può dipendere da 6 fattori: 1- Execution Environment

- 2- Size dell'input
- 3- Tipo di computazione
- 4- Historical runs & job

EXECUTION ENVIRONMENT

What we need to know about where the job will run

- How much memory do I have available to me: on a single node/ on the cluster
 - In my queue
- How much CPU: on a single node, on the cluster → corresponds to total number of concurrent tasks
- We can get all this information from the yarn api (<https://dzone.com/articles/how-to-use-the-yarn-api-to-determine-resources-available>)
- Can I configure dynamic allocation?
- Cluster Health

SIZE OF THE INPUT DATA

- How big is the input data on Disk
- How big will it be in memory?
 - Coefficient of In Memory Expansion: shuffle Spill Memory / shuffle Spill disk
- Can get historically or guess
- How many part files? (the default number of partitions at read)
- Cardinality and type?

- La prima prova è quella di usare le partitioni

How much memory should each task use?

```
def availableTaskMemoryMB(executorMemory : Long): Double = {  
    val memFraction = sparkConf.getDouble("spark.memory.fraction", 0.6)  
    val storageFraction = sparkConf.getDouble("spark.memory.storageFraction", 0.5)  
    val nonStorage = 1 - storageFraction  
    val cores = sparkConf.getInt("spark.executor.cores", 1)  
    Math.ceil((executorMemory * memFraction * nonStorage) / cores)  
}
```

Partitions could be $\text{inputDataSize} / \text{memory per task}$

```
def determinePartitionsFromInputDataSize(inputDataSize: Double): Int = {  
    Math.round(inputDataSize/availableTaskMemoryMB()).toInt  
}
```

Historical Data

Si possono usare le info storiche Job duration, Stage Metrics, Task Metrics, Type of completion, Execution