

Parallel Programming

Per parallel programming si intende l'andare a implementare degli algoritmi che possono essere eseguiti in parallelo su più macchine.

Design:

- 1) Identificare quali pezzi di lavoro possono essere performati contemporaneamente
- 2) Partitionare il lavoro in processioni indipendenti
- 3) Distribuire un input/output/dati interni del programma \rightarrow Non necessario se la mem. è condivisa
- 4) Coordinare l'accesso ai dati per evitare conflitti \rightarrow Non necessario se vengono usati i messaggi
- 5) Assicurare la sincronizzazione tra i job

Task Dependency Graph

Un problema diviso in task può essere descritto da un Task dependency GRAPH (TDG), il quale è un Directed Acyclic Graph (DAG) dove:

- 1) Nodi = Tasks
- 2) Archi diretti = Ordine di esecuzione (control dependency)
- 3) Node labels = Identifica il task (grandezza computazionale / peso del task)

Osservazioni:

- 1) L'esecuzione di una op. può essere divisa in task in diversi modi
- 2) lo stesso problema può essere diviso in diversi modi allo stesso modo

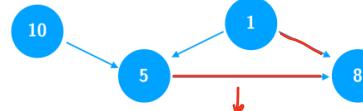
Task Granularity

Il numero dei task determina la "granularity" di un problema

- 1) Se il numero di task è grande, abbiamo la Fine-grained Decomposition
- 2) Se il numero di task è piccolo, abbiamo la coarse grained decomposition

Degree of Concurrency

Indica il numero di task che possono essere eseguiti in parallelo.



Indica che il task 8 può essere eseguito solo dopo i task 1 e 5

Questo valore può cambiare nel tempo, infatti viene solitamente misurato:

- 1) Maximum degree of Concurrency
- 2) Avg Degree of Concurrency

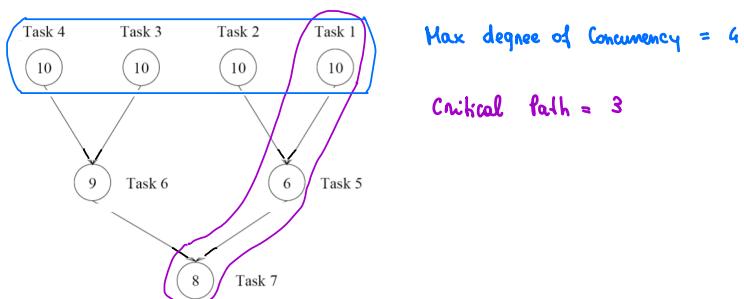
Osservazioni:

- 1) Se avg degree è simile al massimo, abbiamo che il parallel system è usato molto efficientemente.
- 2) Ye degree of concurrency aumenta se aumenta la granularità (+ task, + concorrente) e viceversa.

Critical Path

Ye critical Path in un TDG rappresenta un sequenza di task che devono essere eseguiti uno dopo l'altro (senza aver la possibilità di parallelizzare)

Ye path più lungo determina il minimum execution time di un parallel program, chiamato anche Critical Path length.



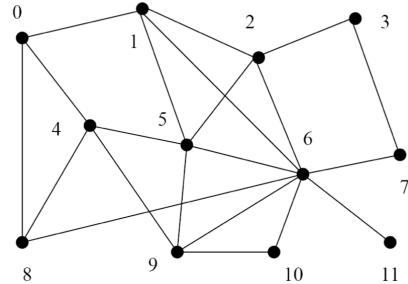
Limitazioni delle performance del Parallel Computing

- 1) C'è un limite intrinseco in quanto la granularity di una computazione può essere fine.
Non possiamo dividere il problema in infiniti piccoli task.
- 2) Ye task devono scambiarsi i dati tra loro \Rightarrow Communication overhead
Ye tradeoff tra la granularità e l'overhead della comunicazione associata, spesso determinano i limiti delle performance

Task interaction graph

Se i task si scambiano i dati, abbiamo un TIG, il quale è un grafo pesato senza direzionalità.

- 1) Nodi = Tasks
 - 2) Archi indiretti = Scambio dei dati
 - 3) Node labels = Identifica il task (Peso computazionale)
 - 4) Edge labels = Quantità di dati Scambiati



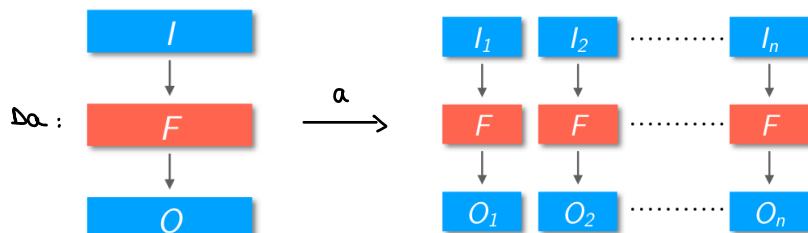
Granularity

In genere, se la granularità è più fine, l'overhead della comunicazione aumenta.

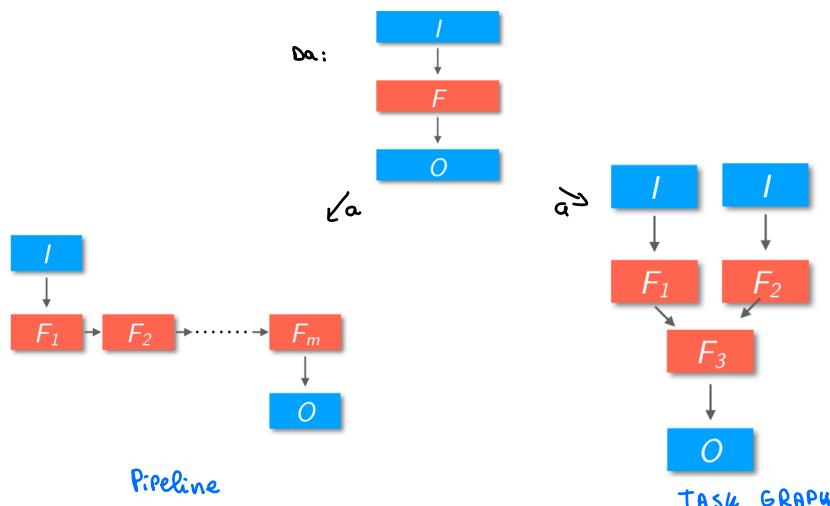
Come ottenerci : parallel task?

Possiamo decomporre un problema sia in base ai dati che al task:

-) Data: Esegue la stessa operazione su dati diversi in parallelo



- 2) Task: Esegue computazioni diverse allo stesso tempo

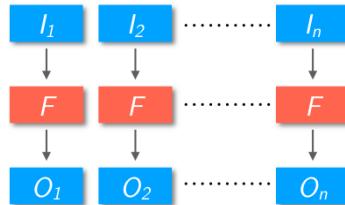


Parallel Patterns

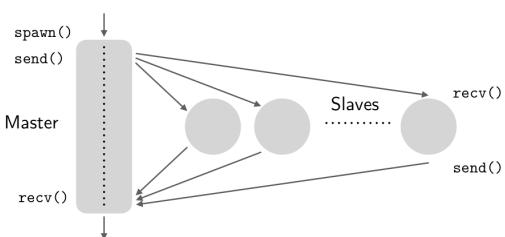
Ci sono diversi pattern possibili nel parallel computing:

- 1) **Embarrassing Parallel Pattern** : Gnosco insieme di task completamente separati

a) Solitamente dataparallel



b) TDG



Master-slave Implementation

a) I dati sono partitionati staticamente

b) N slaves creati dinamicamente

c) A ogni slave viene assegnata una partizione dell'input

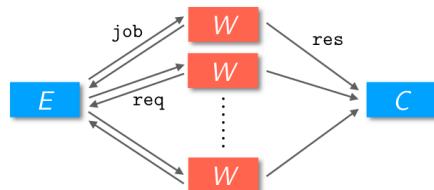
Questo pattern non performa bene quando i dati partitionati non sono bilanciati.

- 2) **Farm Pattern** : È composto da tre entità logiche

a) Emitters: Crea Job staticamente o dinamicamente

b) Workers: Esegue un job da eseguire

c) Collections: Unisce i risultati della computazione



Quanti Workers?

Dati:

a) T_{work} = tempo per completare un task \rightarrow Bandwidth_{computation} = n / T_{work}

b) T_{comm} = tempo di comunicazione (inviare task + ricevere risultati) \rightarrow Bandwidth_{comm} = $1 / T_{comm}$

Il numero ottimale si ottiene quando $B_{comm} = B_{comp}$ (perché se non sto mandando troppi task)

quindi quando: $1 / T_{comm} = n / T_{work}$

e i worker non riescono a processarli, o
viceversa.)



$$n = T_{work} / T_{comm}$$

3) Divide and conquer Pattern

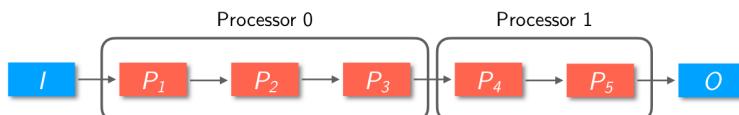
- Decomporre in piccoli problemi indipendenti
- Usare la ricorsione

Problemi:

- Lo stage iniziale e finale hanno un basso grado di parallelismo
- I subtask potrebbero non essere bilanciati

4) Pipeline Pattern

Computazione partitionata in stage, eseguiti sequenzialmente.



Problemi: Il throughput della pipeline dipende dallo slowest stage

↳ Soluzioni possibili:

- Assegnare multi task veloci o pochi task lenti (coarse grain)
- Ridurre il grado di parallelismo

Functional Programming

Le functional programming è un approccio matematico per risolvere i problemi

Caratteristiche

- Purity: le funzioni non modificano nulla all'esterno dei loro parametri.

Ritornano sempre lo stesso output per gli stessi parametri in input (idempotency).

Non hanno side effects

Pure Function

```
int pure(int a, int b)
{
    return a + b;
}
```

Not Pure Function

```
void notpure(int &a, int &b)
{
    a = b;
}
```

2) Higher order functions: le funzioni fanno almeno una di queste due funzioni:

a) Prendono uno o più funzioni come argomenti

b) Ritornano una funzione come risultato

3) Composition: è possibile usare funzioni composite $h = f(g(x))$

4) Currying: Processo con il quale una funzione con più argomenti viene trasformata in una funzione con meno argomenti (numero di argomenti).

$$f(x,y) \text{ return } z \xrightarrow{\text{currying}} f(x) \text{ return } y,z$$

Big Data Applications

Sono applicazioni che solitamente lavorano con grandi quantità di dati e cercano di estrarre info interessanti da essi.

Design

- Scale "out", not "up" (horizontal scaling)
 - Avoid supercomputer, too costly
 - Use commodity machines, low costs
 - Drawback: many failures
- Move processing to the data (Muove il codice verso i dati e non viceversa)
 - Same code, runs everywhere
 - Reduce data over the network
 - Drawback: code must be portable
- Process data sequentially, avoid random access
 - Huge data files (terabytes), not small files (megabytes)
 - Write once, read many
 - Drawback: poor support for standard file APIs
- Right level of abstraction
 - Hide implementation details from applications development
 - Write very few lines of code
 - Drawback: everything needs to fit into the abstraction

Map Reduce

Map Reduce è un programming model derivato dal functional programming, nel quale il programmatore definisce due funzioni:

i) Map function: Prende in input una tupla (chiave, valore) e produce zero o più tuple (chiave, valore)

- z) Reduce function: Prende in input una tuple (chiave, [lista di valori]) e produce zero o più tuple (chiave, valore)
- Entrambe le funzioni sono stateless!

Happens

- Y mapper leggono una pair (chiave, valore) dal HDFS.
- 1) YP mapper può ignorare la chiave o usarla
 - 2) l'output del mapper deve essere nella forma (key,value)
 - 3) Y mappers funzionano in parallelo ed ognuno processa una porzione dei dati.

Reducers

- Y reducer ricevono una tuple (key, [list of values]).
- 1) I valori intermedi (Mapper out) vengono raggruppati in base alla chiave e presentati in una lista al reducer.
 - a) Tutti i valori associati ad una chiave vanno allo stesso reducer
 - b) I valori sono passati al reducer in un ordine basato sulla chiave
 - 2) L'output produce zero (key,value) pairs finali, che vengono scritte nel HDFS.

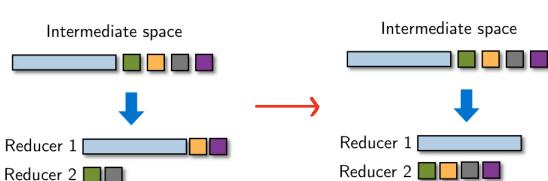
Ottimizzazioni

Y più importante bottleneck nelle map reduce applications è lo scambio dei dati tra map e reduce,
perché:

- 1) È una All-to-All communication
- 2) Dipende dai dati intermedi
- 3) Le proprietà dei dati intermedi dipendono dal codice dell'applicazione e l'input data.

Partitioners

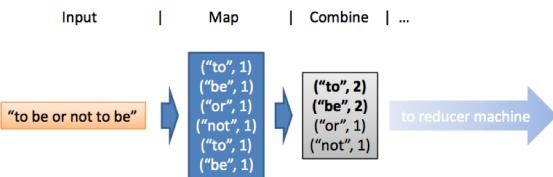
Bilanciano l'assegnamento delle chiavi ai reducers.



Combiners

Fare una local aggregation prima dello shuffle.

Questo permette di ridurre la quantità di dati che devono essere inviati sulla rete.



Hadoop

Hadoop è un MapReduce Framework.

Hadoop terminology

1) MapReduce Job = Unità di lavoro che il client vuole che venga eseguita.

È composta da:

- a) Input Data
- b) MapReduce Program
- c) Configuration information

MapReduce divide un Job in tasks.

2) MapReduce task = Un task può essere di tipo map task o reduce task.

Un task è scheduled da YARN (Yet Another Resource Negotiation) e viene eseguito nel cluster.

Se il task faila, viene automaticamente rischedulato su un nodo differente.

3) Input splits = Parti in cui hadoop ha diviso l'input.

Hadoop crea una map task per ogni split e la map function definita dall'utente viene eseguita per ogni record dello split.

Hadoop Partition e Combiner

- 1) Se un combiner è usato, esso ha la stessa forma della reduce function.
È una implementazione del Reducer ad eccezione che i suoi output type sono le intermediate key e values.
- 2) La partition function opera sulle intermediate key e value types e ritorna un partition index.
(il partition dipende dalla chiave, il valore viene ignorato)

Hadoop Job Configuration

Ci sono diverse configurazioni da fare sul job.

Type Configuration

La type info non è sempre presente a runtime (Java generics type erasure),

quindi deve essere data ad Hadoop in modo esplicito.

Input type viene settato tramite l' "input format".

Gli altri tipi possono essere settati esplicitamente chiamando gli altri metodi sul job.

Attenzione: Dovuto a questo type erasure, è possibile configurare gli hadoop jobs con tipi incompatibili, perché la config non è checkata a runtime.

Default configurations

Le config di default per i vari componenti sono:

- 1) Default input format = TextInputFormat → output = Key (longWritable), value (text)
- 2) Default mapper = Identità
- 3) Default reducer = Identità
- 4) Default partitioner = HashPartitioner → fa l'hash delle chiavi intermedie per determinare la partizione.

Ogni partizione è processata da un reduce task, quindi il numero di esse è uguale al numero di reduce tasks del Job.

Di default c'è un Reducer, quindi una partizione.

- 5) Numero di Map tasks = Numero degli split \rightarrow Dipende dalla grandezza dell' input e il file's block size.

Hadoop Serialization

Hadoop usa la sua serializzazione chiamata Writables e per effettuare la comunicazione tra i nodi usa la Remote Procedure Calls (RPC).

Java primitive	Writable implementation	Serialized size (bytes)
boolean	BooleanWritable	1
byte	ByteWritable	1
short	ShortWritable	2
int	IntWritable	4
	VIntWritable	1-5
float	FloatWritable	4
long	LongWritable	8
	VLongWritable	1-9
double	DoubleWritable	8

Permettono di usare meno bytes di quelli di default

Utile quando sappiamo che una variabile occupa meno di 8 bytes.

Hadoop Input

Gli input split e records sono logici e sono rappresentati dalla Java class InputSplit.

Un InputSplit ha una lunghezza in bytes e un insieme di storage locations.

Uno split è una REFERENZA ai dati, non li contiene direttamente.

Le storage locations vengono usate da Hadoop per piattare i map tasks il più vicino possibile ai data splits.

La lunghezza viene usata per ordinare gli split in modo che il più grande viene processato per primo, per minimizzare il job runtime.

- 1) Il client che punta il job chiama la `getSplits()` per calcolare gli split.
- 2) Li invia all'application master, il quale usa le loro storage locations per schedulare i map tasks che processeranno gli split nel cluster.
- 3) La map task passa lo split al `createRecordReader()` method sull' `InputFormat` per ottenere un `RecordReader` per quello split.
- 4) Un `RecordReader` è un iterator sui records, e il map task ne usa uno per generare le pairs key-value, le quali li passa alla map function.

Setup e cleanup Methods

Sono metodi usati dal Mapper o Reducer per eseguire del codice prima/dopo la map/reduce function.

Ad esempio:

- 1) Inizializzare le data structures
- 2) Leggere i dati da un file esterno
- 3) Settare i parametri

`Setup()` = Metodo eseguito prima del map/reduce method

`Cleanup()` = Metodo chiamato dopo del map/reduce method

Hadoop Tricks

- 1) Limitare l'uso di memoria che il programma utilizza durante l'esecuzione
 - a) Evitare il salvataggio dei reduce values in liste locali se possibile
 - b) Usare oggetti static final
 - c) Riusare gli oggetti Writable
- 2) Gli oggetti vengono condivisi tra le varie reduce invocations \rightarrow Fare deep copies se necessario
- 3) Usare la job config o dati esterni (HDFS, cache) per passare i parametri
- 4) Attenzione nell'allocare le liste per salvare gli Herable obj \rightarrow Memory!

Hadoop Distributed File System

Hadoop DFS è un file system distribuito con le seguenti caratteristiche:

- 1) Fault tolerance
- 2) High Throughput
- 3) Suitable per large data sets, il tempo per leggere l'intero file è più importante del leggere il primo record.
- 4) Streaming Access ai dati del FS
- 5) Può essere costruito su hardware "scarsa"

Organizzazione del FS

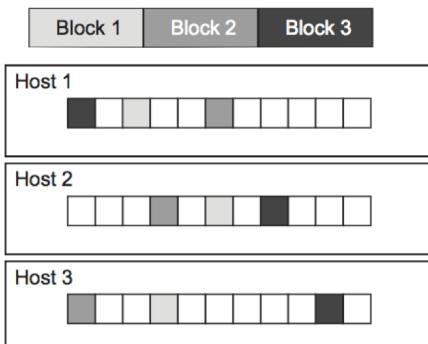
- 1) I file sono divisi in blocchi (chunks) da 64/128 mbytes di grandezza.
- 2) Ogni blocco è replicato in nodi differenti (solitamente ≥ 3) che sono su racks differenti.
- 3) Il master node salva, per ogni file, la posizione del suo blocco ed è anch'esso replicato.
- 4) Una directory (tree) per il file system conosce dove trovare il master node e può essere anch'essa replicata.
- 5) Tutti i partecipanti al DFS sanno dove le copie della directory sono.

Blocchi

I blocchi indicano la quantità di dati che possono essere scritti / letti, 64 mbyte di default per minimizzare i costi di nascita.

La block abstraction permette:

- 1) File più grandi dei blocchi
- 2) Non necessitano di essere salvati sullo stesso disco
- 3) Semplificano lo storage subsystem
- 4) Facile fare la replicazione



Name Nodes e DataNodes

L'architettura usata da Hadoop è di tipo Master/slave.

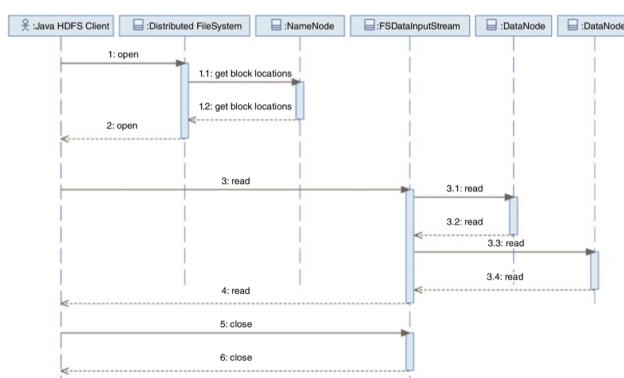
NameNode: Il namenode è un Master Server che gestisce il FS namespace e nega l'accesso ai file dei clients.

Data Nodes: I data nodes sono gli slave e sono solitamente uno per nodo del cluster.

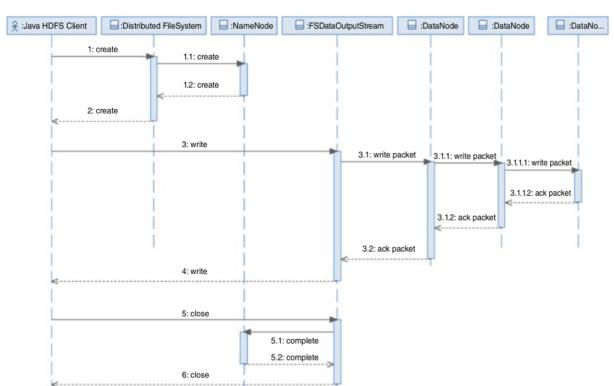
Un file viene spartito in uno o più blocchi e l'insieme dei blocchi sono salvati nei data nodes.

I data node inviano dei keep-alive messages per informare il namenode del loro stato.

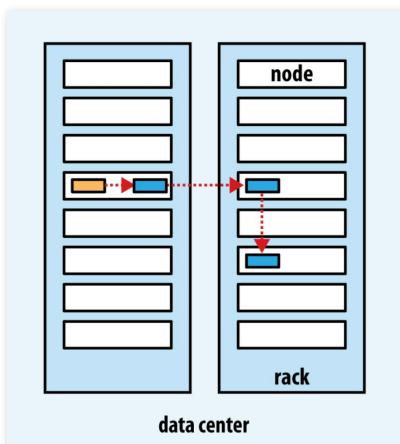
Anatomy of a read



Anatomy of a write



Sistema di Replica



- 1° replica: Stesso nodo del client, per client al di fuori del cluster, viene scelto un nodo random
 - 2° replica: Rack differente dalla prima replica, scelto random
 - 3° replica: Stesso rack della seconda replica, differente nodo scelto random
 - Altre repliche: Piancate random tra i nodi del cluster
- Oss: Il sistema cerca sempre di evitare di piazzare troppo repliche sullo stesso rack/node.

Hadoop Distributed Resource Management

Unitalmente, con Hadoop 1.0, c'era un Job tracker il quale accettava i jobs degli utenti, creava i task, assegnava map e reduce task ai task trackers e monitorava i task trackers e il loro stato.

Questo approccio però aveva dei limiti:

- 1) Scalabilità: Il Job tracker era una bottleneck
- 2) Disponibilità: Il Job tracker era un singolo punto di fallimento e se esso falliva, i jobs dovevano essere resubmitted dagli users e ripartite da 0.
- 3) Uso delle risorse: Dovuto al numero predefinito di map e reduce
- 4) Problemi nell'eseguire le appl. non-hadoop essendo che hadoop è batch-driven

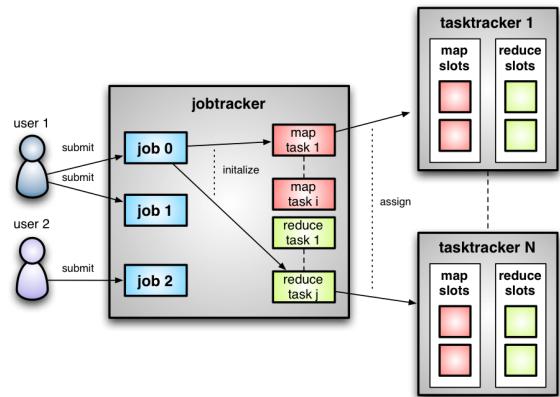
Hadoop 2.0 and YARN

Con l'uso di YARN (Yet another resource Negotiator), le limitazioni sopra descritte vengono risolte.

YARN Components

YARN fornisce i suoi servizi principali tramite due tipi di long-running daemon processes:

- 1) Resource Manager (uno per cluster): Per gestire l'uso delle risorse tra i vari clusters
- 2) Node Managers (uno per nodo del cluster): Per eseguire e monitorare i containers



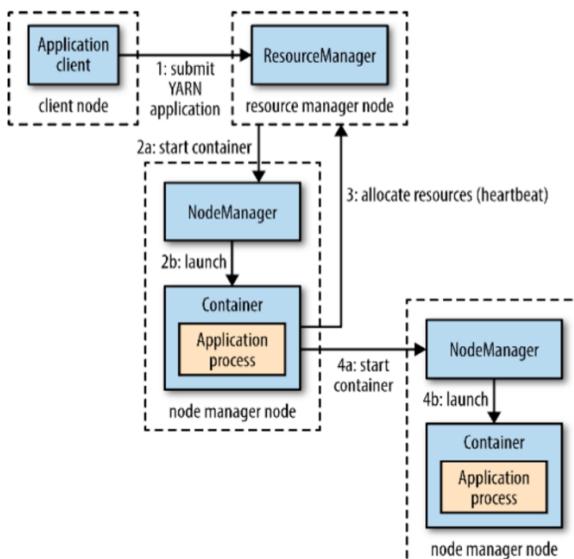
Un container esegue un application-specific process (Process master) con un insieme di risorse vincolato.

Una resource request per un insieme di container può esprimere:

- 1) l'ammontare delle risorse computazionali richieste da ogni container
- 2) g vincoli locali per i container in quella richiesta.

↳ Se non possono essere soddisfatti:

- a) Non viene fatta l'allocazione
- b) Il vincolo può essere non considerato



YARN Applications

La vita di una YARN application può essere categorizzata in funzione di come si mappano ai Jobs che l'utente esegue:

- 1) First: Un'applicazione per user job → Es: MapReduce Jobs
- 2) Second: Un'applicazione per workflow o user session di jobs → Es: Spark Jobs

↳ Può essere più efficiente rispetto al primo, essendo che i container possono essere riutilizzati
- 3) Third: long-running applications che sono condivise tra gli utenti → Es: App coordination

YARN scheduling

YARN fornisce una scelta di scheduler configurabili

1) F:FO scheduler: Piazza le applicationi in una singola queue e le esegue nell'ordine di submission.

Non è adatta per gli shared cluster.

2) Capacity scheduler: Ha diverse code separate (dedicate), ognuna configurata per usare una frazione della cluster capacity.

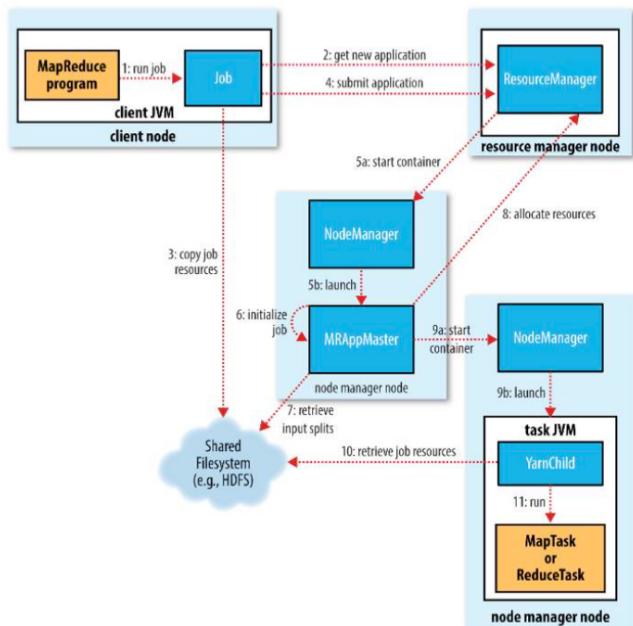
le code sono schedulate in modo F:FO

3) Fair Scheduler: Dinamicamente bilancia le risorse tra tutti i running jobs.

4) Delay scheduling: Aspettare poco tempo può aumentare drasticamente la probabilità che un container venga allocato sul request node, e quindi aumentare l'efficienza del cluster

Ogni node Manager in un Yarn cluster invia periodicamente una heartbeat request al resource manager.

Anatomy of a MapReduce application run



Fault Tolerance

1) Task Failure

a) Lo user code lancia un runtime exception

- Task JVM riporta l'errore al suo application master generatore prima di uscire
- L'app. master dichiara il task come fallito
- L'application master libera il container

b) Sudden exit delle task JVM

- Il node manager informa l'application master
- L'application master dichiara il task come fallito
- L'application master libera il container

c) Hanging Tasks

- L'application master nota che non ha ricevuto un progress update per un po'
- Il TASK JVM process sarà killato automaticamente dopo questo periodo
- L'application master dichiara il task come fallito

OSSERVAZIONI: 1) Quando l'application master notifica un task come failed, rischedule la sua esec.
2) Il rescheduling viene evitato su un node manager dove è fallito precedentemente.

2) Application Master failure

a) Il resource manager rileva la failure e initializza una nuova istanza del master e seguito in un nuovo container (gestito da un node manager)

b) Il client necessita di tornare indietro al resource manager per chiedere il nuovo application master's address

c) Se un MapReduce application master faila due volte non verrà tentato nuovamente e il job fallisce

3) Node Manager Failure

- a) Il resource manager noterà che un node manager ha smesso di inviare heartbeats se non ne riceve uno per 10 min
- b) Il Resource Manager lo rimuoverà dal suo pool di nodi dove schedolare i containers
- c) Qualsiasi task o application master in esecuzione sul failed node sarà considerato failed
- d) I node managers possono essere blacklisted se il numero di failures dell'appl. è alto.
- e) Il blacklisting è effettuato dall'application master

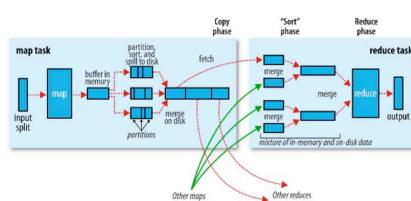
4) Resource Manager Failure

- a) Per ottenere high availability, è necessario eseguire una pair di resource managers in una active-standby configuration
- b) Le info riguardanti tutte le running applications sono salvate in un highly available state store.
- c) Lo standby resource manager può recuperare il core state del failed active resource manager
- d) La transizione di un resource manager da standby a attivo è gestita da un failover controller
- e) Il failover controller di default usa la leader selection per assicurare che ci sia un solo resource manager attivo per volta.

Speculative Execution

- Problem:** Stragglers (i.e., slow workers) significantly lengthen the completion time:
 - Other jobs may be consuming resources on machine
 - Bad disks with soft (i.e., correctable) errors transfer data very slowly
 - Other weird things: processor caches disabled at machine init
- Solution:** Close to completion, spawn backup copies of the remaining in-progress tasks.
 - Whichever one finishes first, "wins"
- Additional cost: a few percent more resource usage
- Example: A sort program without backup = 44% longer

Shuffle and Sort



Map Reduce Design Patterns

MapReduce ha diversi design patterns, utilizzabili in base all'applicazione.

Intermediate Data

Gli intermediate data, come abbiamo visto, sono i dati intermedi prodotti dal mapper e in input al reducer.

Questa operazione di scambio di dati map->reduce è una bottleneck.

Per risolvere questo problema abbiamo già visto: Reduce Data e i Combiners.

Un'altra possibile soluzione è usare un In-Mapping combining.

In-Mapping Combining

In-mapping combining ci permette "sacrificare" un po' di memoria per ridurre la quantità di dati inviati sulle reti.

Infatti, viene fatto un combining dei dati all'interno del mapper andando a costituire una data struttura interna, popolandola e fare l'emit dell'intera struttura invece che per ogni dato.

```
1: class MAPPER
2:   method INITIALIZE
3:     H ← new ASSOCIATIVEARRAY
4:   method MAP(docid a, doc d)
5:     for all term t ∈ doc d do
6:       H{t} ← H{t} + 1
7:   method CLOSE = cleanup method
8:     for all term t ∈ H do
9:       EMIT(term t, count H{t})
```

} creatione della struttura

} singolo emit per ogni termine

Vantaggi:

- Complete local aggregation control (how and when)
- Guaranteed to execute { nei casi precedente non è detto che sia usato il combiner}
- Direct efficiency control on intermediate data creation
- Avoid unnecessary objects creation and destruction (before combiners)

Svantaggi:

- Breaks the functional programming background (state)
- Potential ordering-dependent bugs
- Memory scalability bottleneck (solved by memory foot-printing and flushing)

Matrix Generation

Dato un input di dimensione N , genera un output $N \times N$.

es: Dato una collezione di documenti, emit dei bigrammi

Due soluzioni:

- 1) Pairs = generare $O(N^2)$ dati in $O(1)$ space
- 2) Stripes = generare $O(N)$ dati in $O(N)$ space

Pairs

```

1: class MAPPER
2:   method MAP(docid a, doc d)
3:     for all term w ∈ doc d do
4:       for all term u ∈ NEIGHBORS(w) do
5:         EMIT(pair (w, u), count 1)           ▷ Emit count for each co-occurrence
6:
7: class REDUCER
8:   method REDUCE(pair p, counts [c1, c2, ...])
9:     s ← 0
10:    for all count c ∈ counts [c1, c2, ...] do
11:      s ← s + c                         ▷ Sum co-occurrence counts
12:    EMIT(pair p, count s)

```



Sente l'uso dell'in-mappe combining sto generando una quantità di dati enorme perché per ogni termine "vicino" a w sto facendo un emit delle coppie (w, u)

Stripes

```

1: class MAPPER
2:   method MAP(docid a, doc d)
3:     for all term w ∈ doc d do
4:       H ← new ASSOCIATIVEARRAY            ▷ Per ogni termine
5:       for all term u ∈ NEIGHBORS(w) do
6:         H[u] ← H[u] + 1                   ▷ entra un associative array
7:       EMIT(Term w, Stripe H)             ▷ Popola l'associative array
8:
9: class REDUCER
10:  method REDUCE(term w, stripes [H1, H2, H3, ...])
11:    Hf ← new ASSOCIATIVEARRAY          ▷ Tally words co-occurring with w
12:    for all stripe H ∈ stripes [H1, H2, H3, ...] do
13:      SUM(Hf, H)                      ▷ faccio l'emit del termine e l'associative array
14:    EMIT(term w, stripe Hf)            ▷ Element-wise sum

```

Con l'uso dell'in-mappe combining riduco la quantità di dati generati andando a popolare un associative array e facendo l'emit solo una volta per fermare.

In più considero anche l'unicità delle copie in questo modo e non faccio l'emit più volte perché avranno lo stesso posto nell'associative array.

Relation Algebra Operations

A_1	A_2	A_3	A_4	A_5

Schema

Tuple t

Relation R

SELECTION: Selezionare le tuple che soddisfano una certa condizione c

PROJECTION: Per ogni tupla, selezionare un sottoinsieme di attributi

UNION, INTERSECTION, DIFFERENCE: Operazioni intrecciate tra due relazioni con lo stesso schema.

NATURAL JOIN

Grouping e Aggregation

Selection

Map: Per ogni tupla t in R , se la condizione è soddisfatta, in output una (t, \perp) pair

Tupla come chiave
empty value

Reduce: Per ogni $(t, \perp, \perp, \perp, \dots)$ pair in input, in output produce (t, \perp)

Projection

Map: Per ogni tupla t in R , crea una nuova tupla t' che contiene solo gli attributi progettati e in output una (t', \perp) pair

Reduce: Per ogni $(t', \perp, \perp, \perp, \dots)$ pair, in output produce (t', \perp)

Unione

Map: Per ogni tupla t in R , in output produce una (t, \perp) pair

Reduce: Per ogni input key t , ci saranno 1 o 2 valori uguali a \perp .

Li unisce in un singolo output (t, \perp)

Intersezione

Map: Per ogni tupla t in R , in output produce una (t, \perp) pair

Reduce: Per ogni input key t , ci saranno 1 o 2 valori uguali a \perp .

Se ci sono 2 valori, li unisce in un singolo output (t, \perp) .

Se invece è unico il valore, non fa nulla.

Difference

Map: Per ogni tupla t in R , produce in output (t, R) e per ogni tupla t in S , produce in output (t, S)

Reduce: Per ogni input key t , ci saranno 1 o 2 valori uguali a \perp .

Se c'è 1 valore uguale a (t, R) allora produce in output (t, \perp) .

Se ci sono 2 valori, non fa nulla.

Natural Join

Assumiamo per semplicità di avere due relazioni: $R(A, B)$ e $S(B, C)$.

Vogliamo trovare le tuple che sono in accordo con i B attribute values e fare l'output di esse.

Map: Per ogni tupla (a,b) da R, produce in output $(b, (R,a))$ e per ogni tupla (b,c) da S, produce in output $(b, (S,c))$

Reduce: Per ogni key b, ci sarà una lista di valori della forma $(R,a) \circ (S,c)$.

Costituisce tutte le coppie tra questi valori e li produce in output insieme a b.

Grouping e Aggregation

Per semplificare, assumiamo che abbiamo una relazione $R(A,B,C)$ e vogliamo raggruppare per A ed aggregare su B, scartando C.

Map: Per ogni tupla (a,b,c) da R, produce in output (a,b) .

Ogni chiave rappresenta un gruppo.

Reduce: Applica l'operazione di aggregazione alla lista di b values associati con la group key a, producendo x.

Produce poi in output (a,x)

Matrix Vector Multiplication

Quando consideriamo la moltiplicazione tra vettore e matrice,

dobbiamo considerare due casi:

1) Il vettore v è fissa nella machine's memory

2) Il vettore v non è fissa nella machine's memory

$$\begin{array}{c} \boxed{} \\ \times \\ \boxed{} \end{array} = \boxed{} \\ \text{Matrix } M \\ \text{size } m \times n \\ \text{Vector } v \\ \text{size } n \times 1 \\ m \times 1 \end{matrix}$$

Il vettore fissa

La matrice è salvata nel HDFS come una lista di (i,j, m_{ij}) tuples e l'elemento v_j di v è disponibile a tutti i mappers.

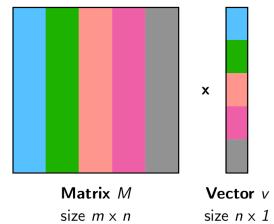
Map: Prende in input $((i,j), m_{ij})$ e produce in output $(i, m_{ij}v_j)$

Reduce: Prende in input $(i, [m_{i1}v_1, \dots, m_{in}v_n])$ e produce in output $(i, m_{i1}v_1 + \dots + m_{in}v_n)$

Yl vettore non fitta

Yl vettore viene diviso in sub-vectors di ugual dimensione che possono fittare in memoria.

Coh lo stesso concetto, anche la matrice viene divisa in strisce.



le strisce i e il subvector i sono indipendenti dalle altre strisce / subvectors

Viene usato l'algoritmo precedente per ogni coppia (stripe, subvector)

Matrix Matrix Multiplication

Una matrice può essere vista come una 3 attributes relation: (row-index, column-index, value_{row,col})

Quindi: M e N possono essere descritte come:

$$M \rightarrow (i, j, m_{ij})$$

$$N \rightarrow (j, k, n_{jk})$$

Yl prodotto MN può essere visto come un natural join sull'attributo j , seguito dal calcolo del prodotto, seguito dal grouping e aggregation.

OSS: Essendo le matrici grandi composte spesso da molti 0, queste tuple vengono omesse

1° stage

Map: Dato (i, j, m_{ij}) produce $(j, (M, i, m_{ij}))$

Dato (j, k, n_{jk}) produce $(j, (N, k, n_{jk}))$

Reduce: Dato $(j, [(M, i, m_{ij}), (N, k, n_{jk})])$ produce $((i, k), m_{ij} \times n_{jk})$

altrimenti nulla

2° stage

Map: Identity

Reduce: Produce la somma della lista di valori associati con la chiave

Graphs

Un grafo è $G = (V, E)$ dove V rappresenta i vertici ed E gli archi.

Rappresentazione: Grafi

Per rappresentare un grafo possiamo usare due metodi:

1) Adjacency Matrix

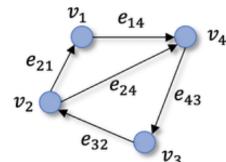
Rappresentare un grafo come una matrice $n \times n$, dove:

$$a) n = |V|$$

$$b) m_{ij} = 1 \text{ se esiste un arco DIREZIONALE da } i \text{ a } j$$

Vantaggi:

- Facile da manipolare con la matematica
- le iterazioni sulle righe e colonne corrispondono alle computazioni su outlinks e inlinks



	v_1	v_2	v_3	v_4
v_1	0	0	0	1
v_2	1	0	0	1
v_3	0	1	0	0
v_4	0	0	1	0

Svantaggi:

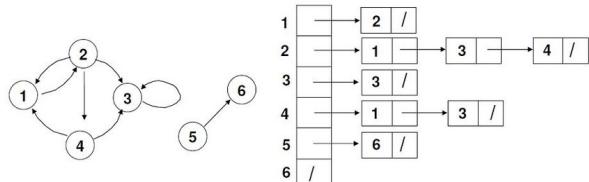
- Molte zeri → matrici sparse
- Molto spazio perso

2) Adjacency List

Permettono di prendere la adjacency matrix e buttare via tutti gli zero

Vantaggi:

- Rappresentazione più compatto
- Facile da computare sugli outlinks



Svantaggi:

- Molto più difficile da computare sugli inlinks

Shortest Path Algorithm

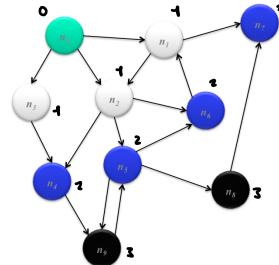
Lo shortest Path algorithm consiste nel determinare il percorso più breve tra due nodi.

Definizione: b è raggiungibile da a se b è nella adjacency list di a

Dato un source node S, l'intuizione è la seguente:

- 1) Per il nodo S, la $\text{DISTANCE}(S) = 0$
- 2) Per tutti i nodi p raggiungibili da S, la $\text{DISTANCE}(p) = 1$
- 3) Per tutti i nodi n raggiungibili da altri insieme di nodi M, la $\text{DISTANCE}(n) = 1 + \min(\text{DISTANCE}(m), m \in M)$

$$\text{DISTANCE}(n) = 1 + \min(\text{DISTANCE}(m), m \in M)$$



Data representation

Key = Node ID

value = (distanza dalla source, adjacent list del nodo)
↓
inizializzata a ∞

Hopper

- 1) riceve in input (node ID, (d, adjacent list))
- 2) per ogni nodo m nella adjacency list, produce in output (m, $d+1$)
- 3) Bookkeeping Addizionale deve tenere traccia del path attuale

Sort/shuffle

Raggruppa le distanze by node ID

Reducer

- 1) Seleziona la distanza minima tra i nodi ricevuti
- 2) Bookkeeping Addizionale deve tenere traccia del path attuale

Dettagli:

Sono necessarie più iterazioni per esplorare l'intero graph

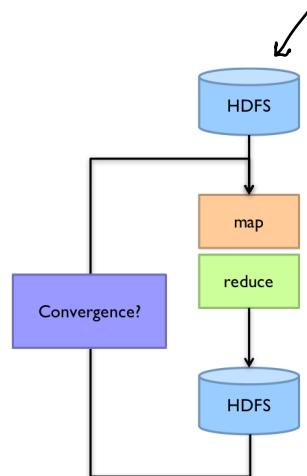
- 1) Ogni iterazione della MapReduce aumenta "la frontiera" di 1-hop (> numero di reachable nodes)
- 2) Il mapper emette anche (n, adj list) per mantenere la graph structure

Pseudocodice

```

1: class MAPPER
2:   method MAP(nid n, node N)
3:     d ← N.DISTANCE
4:     EMIT(nid n, N)
5:     for all nodeid m ∈ N.ADJACENCYLIST do
6:       EMIT(nid m, d + 1)
7:
8: class REDUCER
9:   method REDUCE(nid m, [d1, d2, ...])
10:    dmin ← ∞
11:    M ← ∅
12:    for all d ∈ counts [d1, d2, ...] do
13:      if IsNODE(d) then
14:        M ← d
15:      else if d < dmin then
16:        dmin ← d
17:    M.DISTANCE ← dmin
18:    EMIT(nid m, node M)
  
```

▷ Pass along graph structure
▷ Emit distances to reachable nodes
▷ Recover graph structure
▷ Look for shorter distance
▷ Update shortest distance



Graph Algorithm General Approach

- 1) Rappresentare i grafici come adjacent lists
- 2) Performare alcune local computations nel mapper
- 3) Trasmettere i risultati parziali tramite gli outlinks, con chiave i destination node
- 4) Iterare fino a convergenza, controllato da un "driver" esterno
- 5) Passare la graph structure tra le varie iterazioni

MapReduce Complexity

le fonti di costo a livello computazionale sono:

- 1) Map cost: Costo dell'esecuzione di ogni mapper
 - È un costo fisso (costante), viene incluso solitamente nel transmission cost
- 2) Transmission cost (costo di comunicazione): È il costo della shuffle and sort phase, dove:
 - dati vengono trasferiti dai mapper ai reducer
 - è proporzionale al numero totale di pair (key,value) intermedi generati dai mappers
- 3) Reduce cost (computation cost): Costo di esecuzione dei reducer
 - somma del costo computazionale per ogni reducer, solitamente non grande in confronto con l'input / dati intermedi prodotti dall'algoritmo

Computation - communication Trade off

Definiamo due metriche:

- 1) **Reducer site (q)**: limite superiore sul numero di intermediate values che possono apparire nella lista associata ad una singola chiave.

Cioè è un valore stimato t.c tutte le liste date ai reducers hanno un numero di valori inferiore o uguale al limite

a) $q = \text{Average computation cost}$

b) Con una reducer site piccola possiamo fare che ci siano molti reducers e quindi:

→ High parallelism → low overall computation time

c) Con una reducer site piccola possiamo performare le computazioni associate ad un singolo

reducer nella main memory (RAM) del compute node (essendo pochi valori posso salvare nella main memory)

→ low synchronization → low overall computation time

- 2) **Replication rate (r)**: Numero di pairs (key,value) intermedie prodotte da ogni mapper su tutti i suoi input, diviso il numero degli input

a) $r = \text{Average communication cost}$

b) Con un replication rate piccolo, riduciamo la quantità di dati inviati sulla rete

Osservazione: Noteremo che il replication rate (r) aumenta la reducer site (q), questo perché sono inversamente proporzionali in un rapporto: $qr = 2n$ dove n è l'input size

1) 4 casi estremi sono i seguenti:

a) $r=1, q=2n \Rightarrow$ c'è un solo reducer che fa tutte le computazioni

b) $r=n, q=2 \Rightarrow$ c'è un reducer per ogni coppia di input

2) Bisogna scegliere un r abbastanza piccolo t.c i dati fitano nella local DRAM e c'è abbastanza parallelismo.

Example: Natural Join

Map

- For each input tuple $R(a, b)$:
 - Generate key = b , value = (R, a)
- For each input tuple $S(b, c)$:
 - Generate key = b , value = (S, c)

Replication rate
 $r = 1$

Reduce

- Input: $(b, \text{list of values})$
- In the list of values pair each entry of the form (R, a) with each entry (S, c) , and output (a, b, c)

Reducer size
(worst case)
 $q = |R| + |S|$

① Example: 2 steps Mat Mul

First stage

- Map:** given (i, j, m_{ij}) produce $(j, (M, i, m_{ij}))$
given (j, k, n_{jk}) produce $(j, (N, k, n_{jk}))$

Replication rate
 $r = 1$

- Reduce:** given $(j, [(M, i, m_{ij}), (N, k, n_{jk})])$
produce $((i, k), m_{ij} \times n_{jk})$
otherwise do nothing

Reducer size
 $q = 2n$
caso peggiore (entrambi le matrici sono sparse)

Second stage

- Map:** identity
- Reduce:** produce the sum of the list of values associated with the key

Replication rate
 $r = 1$
Reducer size
 $q = n$
caso peggiore (entrambi le matrici non sparse)

② Example: 1 step Mat Mul

Algorithm 1: The Map Function

```

1 for each element  $m_{ij}$  of  $M$  do
2   produce  $(key, value)$  pairs as  $((i, k), (M, j, m_{ij}))$ , for  $k = 1, 2, 3, \dots$  up
   to the number of columns of  $N$ 
3 for each element  $n_{jk}$  of  $N$  do
4   produce  $(key, value)$  pairs as  $((i, k), (N, j, n_{jk}))$ , for  $i = 1, 2, 3, \dots$  up
   to the number of rows of  $M$ 
5 return Set of  $(key, value)$  pairs that each key,  $(i, k)$ , has a list with
   values  $(M, j, m_{ij})$  and  $(N, j, n_{jk})$  for all possible values of  $j$ 

```

Replication rate

$r = n$

Algorithm 2: The Reduce Function

```

1 for each key  $(i, k)$  do
2   sort values begin with  $M$  by  $j$  in  $list_M$ 
3   sort values begin with  $N$  by  $j$  in  $list_N$ 
4   multiply  $m_{ij}$  and  $n_{jk}$  for  $j$ th value of each list
5   sum up  $m_{ij} * n_{jk}$ 
6 return  $(i, k), \sum_{j=1}^n m_{ij} * n_{jk}$ 

```

Reducer size

$q = 2n$

④ e ⑦ : Nessuno dei due è meglio perché se riduci la reduzione si è nel ④
aumento di tempo: neanche nel ⑦

Example: Similarity Join (I)

- Given a large set of elements X and a similarity measure $s(x, y)$
- Output pairs whose similarity exceeds a given threshold t
- Example: given a database of 10^6 images of size 1 MB each, find pairs of images that are similar
 - Input: (i, P_i) where i is an ID for the picture and P_i is the image
 - Output: (P_i, P_j) or simply (i, j) for those pairs where $s(P_i, P_j) > t$

Second Approach

- Group images into g groups with $10^6/g$ images each
- Map**
 - For each picture P_i
 - Generate $g - 1$ keys (u, v) where u is the group id of P_i and v is the id of any other group
 - Associated value is (i, P_i)
 - Problema:** P_i sarà inviata a $(g_1, g_2) \in P_1$ e P_2 sarà inviata a $(g_2, g_1) \in P_2$ \Rightarrow Soluzione: Ordinare & intercalare keys $(u, v) = (\min(u, v), \max(u, v))$
- Reduce**
 - Consider reducer with key (u, v) and value list of pictures
 - For each P_i that belongs to group u in list of pictures
 - For each P_j that belongs to group v in list of pictures
 - Compute $s(P_i, P_j)$, and output (i, j) if it is above threshold
- Input size = n^2 1 MB images = 10^{12} MB
- Replication rate $r = g - 1 = 10^6 - 1 \Rightarrow$ Per ogni img genera $n-1$ copie
- Reducer size $q = 2 = 2 \text{ MB}$
- Communication cost = $n(n - 1) \text{ MB} = 10^{12} \text{ MB} = 10^{18} \text{ bytes}$
- Communication time over 1Gbit ethernet = $8 \times 10^{18}/10^9 \text{ s} = 10^{10} \text{ s} = 300 \text{ years}$

↓ Reduce replication
rate aumentando reduce
size

↓ computazione
abbiamo spostato
del lavoro dal mappe
al reducer!

First Approach

Map

- For each picture P_i
 - For each picture P_j not equal to P_i
 - Generate pair $(i, j), (P_i, P_j)$

Reduce

- Compute $s(P_i, P_j)$
- Output (i, j) if similarity $s(P_i, P_j)$ above threshold t

- Input size = n^2 1 MB images = 10^{12} MB
- Replication rate $r = n - 1 = 10^6 - 1 \Rightarrow$ Per ogni img genera $n-1$ copie
- Reducer size $q = 2 = 2 \text{ MB}$
- Communication cost = $n(n - 1) \text{ MB} = 10^{12} \text{ MB} = 10^{18} \text{ bytes}$
- Communication time over 1Gbit ethernet = $8 \times 10^{18}/10^9 \text{ s} = 10^{10} \text{ s} = 300 \text{ years}$

↓ communication cost
très coûteux

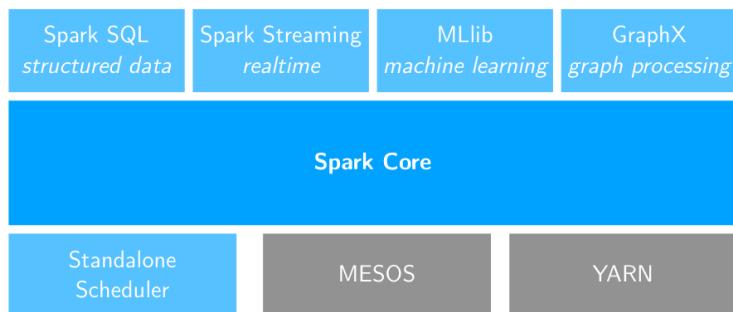
MAPReduce Limitations

- 1) Inefficiente per algoritmi con più passi
- 2) Non ha primitive efficienti per il data sharing
 - a) lo stato tra gli steps vanno all' HDFS
 - b) lento dovuto alle replication e disk storage
- 3) Può richiedere asintoticamente molta comunicazione o I/O

SPARK

Spark è un Framework per il distributed programming che usa la ram invece che l'HDFS per salvare i valori intermedi.

SPARK Stack



Spark Core

Fornisce le funzionalità di base, includendo:

- 1) task scheduling
- 2) Memory Management
- 3) Fault Recovery
- 4) Interacting with storage systems

Fornisce una data abstraction chiamata Resilient Distributed Dataset (RDD), la quale è una collectione di items distribuita su più compute nodes, che può essere manipolata in parallelo.

Fornisce più API per costruire e manipolare gli RDD

SPARK Modules

- 1) SPARK SQL = Usata per lavorare con i dati strutturati
 - a) Permette l'uso delle query via SQL
 - b) Estende la SPARK RDD API;
- 2) SPARK Streaming = Usata per processare le live streams di dati
 - a) Estende la SPARK RDD API;
- 3) MLlib = Usata per il ML
 - a) GraphX = API per manipolare grafici e eseguire graph-parallel computations
 - a) Include graph algo comuni
 - b) Estende la SPARK RDD API;

RDD (Resilient Distributed Dataset)

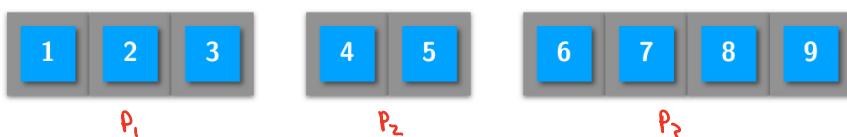
Non modificabile



Un RDD è una distributed memory abstraction, cioè una collezione IMMUTABILE di oggetti distribuita sul cluster.



Una RDD è divisa in un numero di partizioni, le quali sono Atomic Pieces di informazione, cioè non possono essere divise in sub-partitions



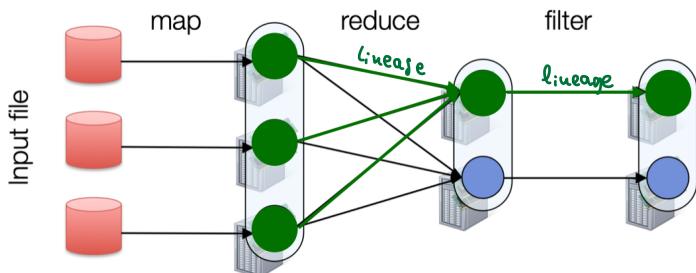
Un RDD può essere mutato andando ad usare una trasformazione che crea un nuovo RDD.

Gli RDD possono essere automaticamente ricostituiti in failure.

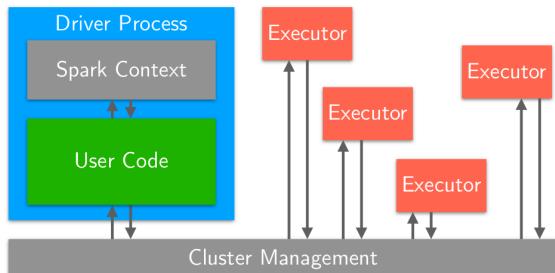
RDD lineage

Quando una trasformazione viene chiamata, non viene fatta immediatamente, ma viene creata una lineage per ogni trasformazione.

La lineage tiene traccia di tutte le trasformazioni dell'RDD e la location da dove legge i dati. In questo modo un RDD può essere ricostruito automaticamente on failure.



Spark Applications Architecture



Una spark application è composta da:

- 1) Driver Process
- 2) Insieme di executor processes

Spark Driver

Il processo driver è:

- 1) Il cuore di una spark application
- 2) Eseguito su un nodo del cluster
- 3) Esegue la main() function

È responsabile di:

- 1) Mantenere le info relative alla spark application
- 2) Interagire con l'utente
- 3) Analizzare, distribuire e schedolare il lavoro tra i vari executors

È composto da:

1) Spark Context

È un oggetto che rappresenta una connessione con il cluster system.

Esso viene creato automaticamente all'avvio della `pySpark shell` ed è accessibile attraverso la variabile `sc`.

```
sc = SparkContext(appName="MY-APP-NAME", master="local[*]")
```

Il `master` argument può assumere diversi valori, ad esempio:

- `local`: run in local mode with a single core
- `local[N]`: run in local mode with N cores
- `local[*]`: run in local mode and use as many cores as the machine has
- `yarn`: connect to a YARN cluster

Per sottimizzare i job fra tutti i cluster managers viene usata la `spark-submit`

2) User Code

Spark Executors

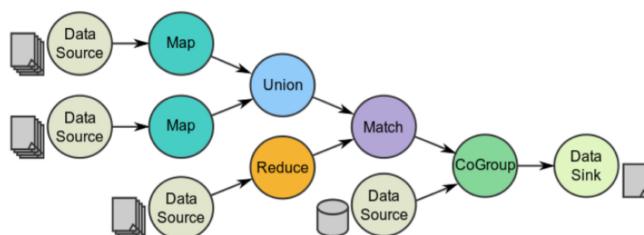
Sono responsabili di:

- 1) Esecuire il codice assegnatoli dal driver
- 2) Reportare lo stato della computazione al driver

Spark Programming Language Model

Esso è basato su operatori parallelizzabili e il dataflow è composto da un qualsiasi numero di data sources, operations e data sinks connettendo il loro input e output.

La job description è basata su un directed acyclic graph (DAG)



RDD operations

Gli RDDs supportano due tipi di operazioni:

- 1) Trasformazioni: Permettono di costruire il piano logico
 - a) Non computano il loro risultato subito (Lazy)
 - b) Ricorda le trasformazioni applicate ad dataset iniziale
 - c) Sono computate solo quando un'azione richiede che il risultato venga restituito al driver program.

- `distinct` removes duplicates from the RDD
- `filter` returns the RDD records that match some **predicate function**
- `sample` draws a random sample of the data, with or without replacement
- `map` transforms an **RDD** of **length n** into another **RDD** of **length n**.
- `flatMap` allows returning **0, 1 or more elements** from map function.
- `sortBy` sorts an RDD
- `union` performs the **merging** of RDDs
- `intersection` performs the **set intersection** of RDD
- To create a key-value RDD:
 - `map` over your current RDD to a basic key-value structure.
 - Use the `keyBy` to create a key from the current value. \Rightarrow
 - `keys` and `values` extract keys and values from the RDD, respectively
 - `lookup` looks up the **list of values** for a particular key in an RDD
- `reduceByKey` combines values with the same key
 - Takes a **function** as input and uses it to **combine values** of the same key
- `sortByKey` returns an RDD sorted by the key
- `join` performs an inner-join on the key
- Other types of join:
 - `?fullOuterJoin`
 - `leftOuterJoin, rightOuterJoin`
 - `cartesian`

- 2) Azioni: Permette di triggerare la computazione

- a) Utilizza Spark per computare un risultato da una serie di trasformazioni
- b) Ci sono tre tipi di azioni:
 - i) Vedere dati nella console
 - ii) Collectionare i dati negli oggetti nativi del linguaggio

III) Scrivere l'output ai data sinks

- `collect` returns all the elements of the RDD as an **array** at the driver
- `first` returns the first **value** in the RDD
- `take` returns an **array** with the first **n** elements of the RDD
 - Variations on this function: `takeOrdered` and `takeSample`
- `count` returns the **number** of elements in the dataset
- `max` and `min` return the **maximum** and **minimum** values, respectively.
- `reduce` aggregates the elements of the dataset using a given **function**.
 - The given function should be **commutative** and **associative** so that it can be computed correctly in parallel.
- `saveAsTextFile` writes the elements of an RDD as a **text file**.
 - Local filesystem, HDFS or any other Hadoop-supported file system.

RDD Persistence

Di Default, ogni RDD trasformato può essere ric算calcolato ogni volta che un'azione viene eseguita su di esso.

Spark supporta anche la persistenza (caching) degli RDD in memoria tra le varie operazioni per un rapido rechiamo.

Quando l'RDD è persistente, ogni nodo salva ogni partizione dell'RDD che computa in memoria e la riusa in altre azioni sul dataset.

Questo permette alle azioni future di essere più veloci.

Per persistere un RDD, bisogna usare il metodo `persist()` o `cache()` su di esso.

Tramite il metodo `Persist` puoi specificare il livello della persistenza:

- 1) Memory - Only
- 2) Memory - and - disk
- 3) Disk - Only
- :

Il miglior livello di storage è il `Memory-only` se il disco non è richiesto (la funzione non è costosa).

Inoltre, conviene usare la replicated storage solo se vogliamo una fast fault recovery.

Alcuni problemi:

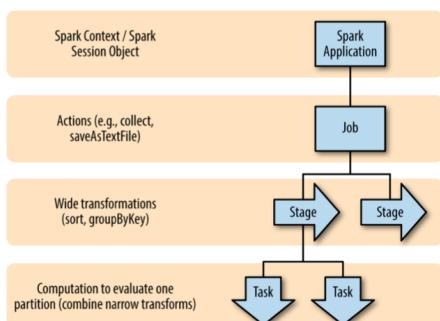
- 1) Job iterativi o singoli con grandi variabili globali
- 2) Tenere in considerazione gli eventi che accadono durante la job execution
- 3) le closures sono rinviate con ogni job
- 4) È inefficiente inviare grandi quantità di dati al worker
- 5) le closures sono one-way, driver to workers

Distributed Shared Variables

In aggiunta agli RDD, Spark ha due tipi di Distributed Shared Variables:

- 1) Broadcast Variables: Permettono di salvare un valore su tutti i worker nodes e riutilizzarlo su varie Spark actions senza doverla rinviare al cluster.
È un modo efficiente per inviare grandi read-only values ai worker.
- 2) Accumulators: Permettono di raggruppare i dati da tutti i task in un unico risultato condiviso.
È un modo efficiente e fault tolerant per aggiornare un valore all'interno di varie trasformazioni e propagarlo al driver node.
Sono variabili che vengono aggiornate solo attraverso operazioni associative e commutative, in modo che sia efficientemente supportate in parallelo.
Le trasformazioni vengono effettuate solo alla chiamata di un'azione (lazy ev.) e vengono viste dai workers come variabili write-only.

Anatomia di una Spark Application

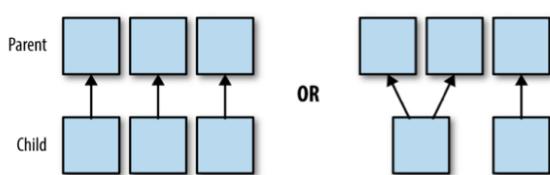


- 1) Spark Application: Non fa nulla fin quando il driver program non chiama un'azione (lazy ev.)
- 2) Spark Job: È il più alto elemento della gerarchia dell'esecuzione Spark
 - a) Ogni Job corrisponde a un'azione
 - b) Ogni azione è chiamata dal Driver program di una Spark application
- 3) Spark stage: Ogni Job è suddiviso in una serie di stages
 - a) Gli stages in Spark rappresentano un gruppo di task che possono essere eseguiti insieme
 - b) le wide transformations definiscono il breakdown dei job negli stages
- 4) Spark Tasks: Uno stage consiste in tasks, i quali sono le più piccole unità di esecuzione
 - a) Ogni Task rappresenta una local computation
 - b) Tutti i task in uno stage eseguono lo stesso codice su un differente pezzo di dati.

Wide e Narrow Transformations

Le trasformazioni ricadono in due categorie:

- 1) Trasformazioni con Narrow Dependencies: le dipendenze sono determinate a design time e



ogni genitore ha al massimo una child partition. Possono essere eseguite su un subset arbitrario senza nessuna info sulle altre partitioni.

- 2) Trasformazioni con Wide Dependencies: Non possono essere eseguite su niste arbitrarie e richiedono che i dati siano partitionati in un modo particolare, ad esempio seguendo il valore della chiave

