**Progetto di Scalable and Cloud Programming (Co-purchase Analysis)**
**Università di Bologna**
**anno 2024-2025**
**Daniele D'Ugo - 0001186581**

Il progetto consiste in un'analisi di co-acquisto, implementata tramite Spark e Scala ed eseguita su Google Cloud – Dataproc, su un dataset fornito. Il codice è disponibile al seguente link:
https://github.com/papum20/unibo__projects__ScalableAndCloudProgramming

Si premette che l'infrastruttura di testing ha dato risultati molto incostanti e lenti (decisamente peggio che eseguendo in locale) e, unito al limite dei crediti per il servizio e al fatto che molti siano stati consumati per imparare a far funzionare al meglio l'infrastruttura stessa, è stato possibile effettuare solo poche misurazioni – come si vedrà nella relazione.

## Implementazione finale

Durante lo svolgimento, sono stati effettuati diversi tentativi e modifiche sull'algoritmo, prima per farlo funzionare al meglio e successivamente per ottimizzarne l'esecuzione; nel codice, ogni cambiamento nell'algoritmo è stato riscritto come una nuova funzione, in modo da poter confrontare le prestazioni con la versione vecchia – attualmente, dunque, sono presenti diverse versioni, mentre altre sono state rimosse.
Tutte le implementazioni, comunque, hanno fatto uso di *RDD*, visti a lezione.

La versione attuale dell'algoritmo usa la seguente serie di operazioni su RDD:

1. *textFile()* : legge il dataset come RDD;
2. *map()* : trasforma ogni riga del file (stringa) in una coppia di numeri interi (rappresentanti i codici di, rispettivamente, scontrino e prodotto);
3. *groupByKey()* : raggruppa gli elementi in base allo scontrino (che ora è la chiave, essendo il primo elemento di ogni coppia);
4. *flatMap(getPairs)* : la *getPairs()*, applicata a ogni gruppo, ritorna tutte le possibili coppie di prodotti acquistati nello stesso scontrino (salvandoli con la coppia e un numero *1*, utile nella *reduce* successiva); successivamente, la *flatMap()* fa in modo che si ottenga un'unica, grande collezione con tutte le coppie di prodotti per tutti gli scontrini;
5. *reduceByKey()* : raggruppa tutte le occorrenze di coppie degli stessi prodotti, contandone le ripetizioni; ritorna dunque il risultato finale della co-purchase analysis, ovvero una mappatura da ogni possibile coppia di prodotti al numero di volte che compaiono nello stesso scontrino;
6. *map()* : converte solo in una stringa da stampare, in formato *.csv*;
7. *saveAsTextFile()* : salva l'RDD su file (già in formato *.csv)*.

L'implementazione della *getPairs()* esegue due cicli annidati, solo per coppie di prodotti *a, b* con *a<b:* in questo modo, già si prende ogni coppia una sola volta (come ci si aspetta nel risultato finale), oltre a generare meno coppie (grazie al minore stretto) – nonostante questo non cambi la complessità asintotica.

Nel repository del progetto sono anche presenti diversi script, spiegati nel suo *README*. Questi si occupano anche di produrre un unico file *.csv* di output, per convenienza, dato che la *saveAsTextFile()* ne produce diversi, per ragioni implementative di Spark e per sfruttarne il parallelismo.

## Implementazioni intermedie

Di seguito, si riporta un'analisi di piccoli e grandi cambiamenti apportati durante l'implementazione, a fronte dei tempi di esecuzione di test (un solo test per quasi ogni tentativo).

Le funzioni con nome del tipo "MapPairsReduce" usano lo stesso approccio descritto, con cambiamenti minori ognuna rispetto alla versione precedente; sono rimaste nel codice e state analizzate anche per poter osservare quali cambiamenti effettivamente migliorino l'efficienza dell'algoritmo. Si riportano anche i tempi, ma si rimanda a dopo l'analisi:

1. versione 1: (873.535ms) invece di fare una *reduce*, usava una *groupBy* e contava gli elementi di ogni gruppo; dunque, usava una prima versione di *getPairs()* (che non aggiunge l'1);
2. versione 2: (947.174ms) cambiamento minore, converte subito le stringhe in interi (come la versione 10);
3. versione 3: (1.416.774ms) semplifica alcune *narrow transformation* unendole (come la 10);
4. versione 4: (1.410.627ms) evita di salvare dati ridondanti;
5. versione 5: (1.317.293ms) inizia ad usare la nuova *getPairs()* (quella presentata sopra);
6. versione 6: (571.140ms) usa *groupByKey()* invece di una *groupBy()* banale;
7. versione 7: (2 test da 553.044ms e 935.760ms; media 654.530,5ms) introduce la *reduceByKey()*;
8. versione 8: (636.282ms) non converte in stringhe, ma stampa gli elementi così come sono;
9. versione 9: (2 test da 673.260ms e 756.017ms; media 714.638,5ms) converte un'altra *groupBy()* banale in *groupByKey()*;
10. versione 10: (3 test da 728.371ms, 55.1890ms e 536.158ms; media 605.473ms) versione finale;

Come già menzionato, i risultati sono molto variabili, e non c'è stata la possibilità di eseguire più test (per poterne fare una media).
Comunque, si può osservare un notevole miglioramento (da tempi superiori ai 1.000.000ms a circa 600.000ms) già dalla versione 6, che non usava la *reduceByKey()*, ma introduceva la nuova *getPairs()* (*getPairs2()*) - che, però, non dovrebbe cambiare l'efficienza. Stranamente, infatti, questo balzo lo si ha solo dalla 5 alla 6, in cui la *groupByKey()* sostituiva solo una *groupBy()* banale (che quindi, a meno di ottimizzazioni nell'implementazione, non cambia la logica del programma) – e non dalla 4 alla 5 (che introduce la *getPairs2()*) o dalla 6 alla 7 (che introduce la *reduceByKey*): potrebbe essersi trattato di un errore di misurazione, o potrebbe rientrare nella (ampia) inconsistenza dei test - oppure potrebbero effettivamente esserci differenze nell'implementazione delle funzioni.
Come ci si poteva aspettare, invece, aggiungere, rimuovere o modificare *narrow transformation* successive non cambia la complessità (essendo sequenze di operazioni elementari, e non cambiando il numero di *shuffle*, che invece chiaramente richiedono uno sforzo computazionale in più), così come altre modifiche minori che non fanno una differenza dal punto di vista asintotico.
In luce di queste osservazioni, la scelta della 10 come versione finale, al di là dell'essere tra le implementazioni con i tempi minori, è dovuta solo a una maggiore pulizia del codice.

Dai test, risulta anche che l'uso di RDD è necessario in ogni fase dell'algoritmo, poiché solo questi possono garantire scalabilità, con grandi input. Infatti, provando ad ottenere i risultati finali sotto forma di *Map* o *Map* concorrenti – attraverso *reduce()* o *aggregate()*, invece che *reduceByKey()* -, si incorreva in errori di out-of-memory; provando a scrivere i risultati su un unico file attraverso funzioni standard (e non la *saveAsTextFile()*), oppure eseguendo prima una *coalesce()* per unire i dati di tutti i nodi di Spark su uno solo, portava a prestazioni inferiori o errori.

Avendo avuto più tempo, crediti di Google Cloud e risorse, si sarebbero potute fare altre prove, per esempio con diversi *partitioner.*

## Analisi delle prestazioni

La *work complexity* dell'algoritmo è stata stimata a *W(n)=O(n*m)*, su un dataset di input formato da *n* elementi (ovvero *n* righe del file), con *m* la massima lunghezza di uno scontrino (*m<=n*). Seguono i passaggi del calcolo*:*

1. *textFile()* : *O(n)*, leggendo ogni riga del file;
2. *map()* : *O(n)*, eseguendo una conversione di ogni riga del file;
3. *groupByKey()* : *O(n)*. Richiede uno shuffle, ma, assumendo che – come dice la documentazione – si usino delle *hashtable*, la *groupByKey()* richiede, per ogni elemento, di calcolarne l'*hash* e inserirlo nella cella giusta dell'*hashtable* (che, tipicamente, ha costo ammortizzato di un inserimento *O(1)*);
4. *getPairs()* : *O(k^2)*, essendo quadratica nelle dimensioni della *collection* in input (di k elementi*)*;
5. *flatMap(getPairs)* : *O(n*m)*. Infatti, nel caso peggiore, in cui gli *n* elementi sono suddivisi tra *n/m* scontrini da *m* elementi ciascuno (cioè tutti hanno dimensione massima, *m*), si dovrebbe applicare la *getPairs() n/m* volte, su *m* elementi alla volta: dunque, *(n/m)*(m^2)=n*m*. Quindi, per esempio, nel caso limite peggiore, si avrebbe un dataset costituito da un solo scontrino (*m=n* e *n/m=1)*, con complessità *O(n^2*);
6. *reduceByKey()* : O(*n*m*). Analogamente a *groupByKey()*, visita ogni elemento una sola volta (ma ora la collezione in input ha lunghezza *O(n*m)*);
7. *map()* : *O(n*m)*. Sarebbe difficile calcolare quante delle coppie possano essere unite dalla precedente *reduce* ma, nel caso pessimo, si avrebbe il suo stesso costo (*O(nm)*), cioè nel caso in cui tutte le coppie siano diverse e la *reduce* non riesca a ridurne il numero (per esempio, con un unico scontrino, e quindi il massimo numero di coppie diverse possibile);
8. *saveAsTextFile()* : *O(nm*), eseguendo la scrittura di ogni elemento - come la precedente *map()*.

Dunque, la complessità finale, per tenere conto di tutte le fasi, sarebbe *W(n)=O(n*m)*.

Da tale analisi, si possono dedurre le parti più impegnative computazionalmente:

- sono presenti due *shuffle (*in *groupByKey()* e *reduceByKey()*);
- la funzione più complessa è la *flatMap(getPairs)*, che costituisce il passaggio da una *collection* di *n* elementi a una di *n*m*, per ogni coppia di prodotti che condividono uno scontrino (con ripetizioni, che verranno poi rimosse dalla *reduce()*);

Si possono però anche determinare dei lower-bound:

- processando l'input (di dimensione *n*), non si può fare meglio di *O(n)*, almeno per lettura ed eventuali conversioni;
- processando l'output (di dimensione al più *p^2*), non si può fare meglio di *O(p^2)*, almeno per la scrittura ed eventuali conversioni.

Alla luce di ciò, questa è stata l'implementazione ritenuta migliore in fase di sviluppo: i due *shuffle* sembrano fondamentali - il primo per raggruppare elementi con lo stesso scontrino, il secondo per contare le coppie di prodotti -, mentre provare a migliorare la *flatMap(getParts)* significherebbe cercare un approccio ben diverso.

Per quanto riguarda la *depth complexity*, la maggior parte degli step sono *embarrassingly parallel* (quindi *O(1)*): le *map*, la lettura e scrittura di file, ma anche *flatMap(getPairs)*, essendo parallelizzabili sia la (*flat*) *map*, sia la *getPairs* (che genera tutte le possibili coppie di prodotti in uno scontrino, in maniera indipendente l'una dall'altra). Ci sono però delle eccezioni, tra cui, soprattutto, gli shuffling, il cui costo denoteremo s*(n)* (costo dello shuffling su dataset di dimensione *n*). D'altra parte, non è stato possibile trovare, tra ricerche su internet e documentazioni, gli esatti costi o implementazioni di varie funzioni di Spark, per cui possiamo solo generalizzare o fare delle supposizioni. In particolare:

- *groupByKey()* : *O(s(n)). A*ssumendo l'implementazione *hash*-based, dovrebbe essere *embarrassingly parallel* (aggiungere tutti gli elementi a un'hashtable in maniera indipendente), se non per lo shuffling di *n* entry del dataset;
- *reduceByKey() : O(log(n)+s(nm))*. Oltre allo shuffling (*s(nm)*, presente qui su dataset grande *O(nm)*), si ha il costo della *reduce* applicata a ogni chiave. Poiché la *reduce* parallela viene tipicamente implementata con un albero (avendo dunque *work complexity* pari al logaritmo della lunghezza della collezione in input, implicando una *depth complexity* della *reduceByKey* pari alla massima complessità tra tutte le *reduce* applicate), il caso peggiore di *O(log(n))*, si ottiene con tutte le entry associate a un'unica chiave (facendo in modo che la *reduceByKey* applichi un'unica, grande *reduce*), cioè, per esempio, quando esistono solo due prodotti diversi, e ogni scontrino presenta esattamente quei due prodotti – risultando, quindi, in *n/2* coppie su cui effettuare un'unica *reduce*.

Mettendo tutto insieme, quindi, si avrebbe una *depth complexity* di *O(log(n)+s(n)+s(nm))*.

Come già detto, non è stato possibile trovare il costo *s(n)*, ma, solo per provare ad avere dati più concreti, un'intelligenza artificiale (chatGPT) ha suggerito che anche lo shuffle abbia un'implementazione basata su alberi, e quindi una *depth complexity* logaritmica nella dimensione del dataset – purtroppo, senza essere in grado di citare fonti che lo supportassero.
Con questa assunzione, la *depth complexity* sarebbe dunque *O(log(nm))*:

- la *O(s(n))* della *groupByKey* diventerebbe *O(log(n))*
- la *O(s(nm))* della *reduceByKey* diventerebbe *O(log(nm))*
- in conclusione, *O(log(n)+log(n)+log(nm))=O(log(nm))*

## Analisi di scalabilità

Calcoliamo adesso le metriche di scalabilità.
Intanto, queste sono le misurazioni sulle esecuzioni dell'algoritmo:

- 1 worker: 1.505.099ms
- 2 worker: 651.140ms
- 3 worker: 728.371ms, 551.890ms, 536.158ms – media 605.473ms

Gli *speedup* (calcolati come rapporto tra il tempo di esecuzione su un nodo e quello sul numero di nodi considerati) sono i seguenti:

- 2 worker: *S(2)=2.31* ( *1.505.099ms / 651.140ms* )
- 3 worker: *S(3)=2.49* ( *1.505.099ms / 605.473ms* )

Le *strong scaling efficiency* (calcolate come rapporto tra *speedup* e numero di nodi considerati) sono:

- 2 worker: *E(2)=1.16* ( *S(2) / 2* )
- 3 worker: *E(3)=0.83* ( *(S(3) / 3* )

Le *weak scaling efficiency* non sono state calcolate, essendosi esauriti i crediti per Goggle Cloud.
Per farlo, sarebbe stato necessario eseguire l'algoritmo su frazioni del dataset – quindi, metà per 2 worker e un terzo per 3 worker -, in modo da poter verificare che ci fosse una sufficiente riduzione del tempo impiegato, all'aumentare dei nodi.
Nel progetto, è anche presente uno script per sezionare il dataset originale (*scripts/4.1.split-dataset.sh*), facendo scegliere tra il prendere le prime righe o delle righe casuali (in numero pari alla frazione della lunghezza desiderata); ogni approccio ha dei pro e contro: considerando che l'originale è ordinato in base allo scontrino, prendere delle righe consecutive assicura che abbastanza prodotti siano presenti in uno stesso acquisto, presumibilmente richiedendo più lavoro da parte dell'algoritmo (in base all'analisi della *work complexity* effettuata). Dunque, con pochi worker (e quindi un fattore di frazionamento del

dataset contenuto) – come nel caso dei nostri 2 o 3 -, prendere delle righe consecutive potrebbe essere una buona idea; tuttavia, andando a scalare da un lato e ridurre il dataset dall'altro, avere degli scontrini non altrettanto ridotti di dimensioni avrebbe un impatto più significativo – per cui, potrebbe essere meglio scegliere le righe casualmente, per assicurare degli scontrini opportunamente ridimensionati (casualmente).
Ovviamente, dei test aiuterebbero ad identificare l'opzione migliore.

In conclusione, analizziamo le metriche calcolate.
Anche qui, le performance - a dir poco altalenanti – e l'impossibilità di effettuare più test - per averne una media - restituiscono dei risultati di scalabilità particolari.
Da *S(2)*, infatti, ci si aspetterebbe un valore inferiore a *2* – con 2 worker -, ma si è invece ottenuto uno *speedup* ben superiore. *S(3)*, invece, sembra più ragionevole, essendo maggiore di *S(2)* ma comunque inferiore a 3. Stona solo il fatto che sia molto vicino a *S(2)*, che però, come appena detto, è probabilmente troppo alto per errori di misurazione.

Anche i risultati calcolati per la *strong scaling efficiency* si allineano a quanto detto per lo *speedup*: *E(2*) vale più di 1 – mentre ci si aspetterebbe un valore inferiore, per quelle parti non parallelizzabili del codice -, mentre *E(2)* rientra nei valori attesi.

## Testing

I test sono stati effettuati, su Google Cloud, con macchine *n2-highmem-2*, cioè ognuna con 2 virtual CPU e 16GB di RAM; la scelta era molto limitata, ma sono state fatte le seguenti considerazioni:

- essendo il limite massimo di 8 vCPU, 2 core per ogni macchina hanno permesso di scalare fino a 4 esecutori;
- RAM inferiori portavano a errori di out-of-memory e, in particolare per il test con single-worker, è stato necessario usare 8GB di JVM.

Inoltre, sempre per i test con un solo nodo, è stato necessario eseguire in *local mode*, indicando di usare tutti i thread disponibili (con *setMaster(local[*])* ), per evitare errori.

Per ottenere i tempi di esecuzione, il codice scala li misura e poi stampa al lancio dell'algoritmo, con metodi analoghi a quelli visti a lezione: in particolare, riporta sia il tempo di creazione dell'RDD, sia quello di esecuzione della scrittura finale su file; ovviamente, solo la *saveAsTextFile()* richiede un tempo significativo (che è quello preso in analisi), essendo l'unica *action* presente nel codice a "triggerare" una computazione di Spark.