Università degli Studi di Verona

DIPARTIMENTO DI INFORMATICA

Corso di Laurea Magistrale in Ingegneria e scienze informatiche

Sistemi di elaborazione di grandi quantità di dati

Confronto prestazioni tra Pig Latin e MapReduce

Autori:

Marco Colognese VR423791

Mattia Rossini VR423614

Anno Accademico 2017/2018

Indice

Introduzione

Lo scopo principale di questo progetto è quello di mettere a confronto Pig Latin e MapReduce.

Per fare ciò è stato deciso di svolgere quattro esercizi basati su un dataset contenente informazioni sugli arrivi e sulle partenze negli aeroporti situati negli Stati Uniti eseguendo quindi delle specifiche analisi su tali dati. Ognuna di queste è stata implementata sia in Pig Latin che in MapReduce e su questa relazione ne sono stati riportati rispettivamente il codice e lo pseudocodice entrambi preceduti da una spiegazione.
Oltre a questo, alla fine di ogni esercizio vengono svolte delle analisi e delle statistiche riguardanti le due run, riportando dunque informazioni relative alle fasi di Map e di Reduce ed allo scambio di dati intermedi tra le due. In particolare il numero di reducer può essere indicato nel codice di default è uno.
Sia nel caso di Pig Latin che di MapReduce, il codice dei quattro esercizi è stato prima testato nel container presente nel proprio calcolatore e successivamente in un cluster vero e proprio messo a disposizione dal docente. Questo è stato fatto per analizzare le eventuali differenze che ci possono essere tra l'utilizzo di un ambiente di simulazione ed uno reale.
Infine, nell'ultimo capitolo, si parla delle competenze acquisite da parte degli studenti grazie allo sviluppo di questo progetto, delle difficoltà incontrate, delle soluzioni adottate e dei vantaggi e svantaggi relativi a Pig Latin e MapReduce.

Nell'implementazione del codice dei quattro esercizi, nella parte riguardante Pig Latin, è stato deciso di utilizzare come nome utente *"ross"*. Dunque, nel caso in cui si volessero eseguire tali codici sul proprio calcolatore, occorre modificarli sostituendo la stringa *"ross"* con il proprio nome utente.
Tale problema è invece assente per quanto riguarda MapReduce, dove il nome utente va specificato scrivendolo a linea di comando solamente quando si esegue il codice.

Dataset

I dati utilizzati come input per i quattro esercizi sono stati recuperati dalla seguente pagina web: `http://stat-computing.org/dataexpo/2009/the-data.html`.
I file scaricabili contengono informazioni strutturate riguardanti i voli aerei negli Stati Uniti e sono composti da 29 campi, in particolare quelli utilizzati sono:

- *Year* (campo numero 1): anno in cui è stato effettuato il volo;
- *DayOfWeek* (campo numero 4): giorno della settimana in cui è stato effettuato il volo, dove 1 corrisponde a lunedì e 7 a domenica;
- *FlightNum* (campo numero 10): numero del volo;
- *ArrDelay* (campo numero 15): differenza in minuti tra l'orario di arrivo previsto e quello effettivo, gli arrivi in anticipo sono indicati con numeri negativi;
- *WeatherDelay* (campo numero 26): ritardo in minuti del volo dovuto al tempo meteorologico.

I file a disposizione sul sito web sono suddivisi per anno, in particolare dal 1987 al 2008. Quelli utilizzati in questo progetto sono quelli compresi tra:

- il 1987 ed il 1990;
- il 2005 ed il 2008.

Infine, per quanto riguarda il problema dei quattro esercizi svolti e di seguito esposti, è stato deciso di prendere spunto da quelli descritti nella pagina `http://stat-computing.org/dataexpo/2009/` sotto la voce *"The challenge"*.

Cluster

Un cluster è un insieme di computer connessi tra loro tramite una rete. Lo scopo principale è quello di distribuire un'elaborazione molto complessa tra i vari calcolatori a disposizione.
Ovvero, un problema che richiede molte elaborazioni, per essere risolto viene scomposto in sotto-problemi più piccoli i quali vengono risolti in parallelo. Questo ovviamente aumenta la potenza di calcolo del sistema.
I cluster hanno le seguenti caratteristiche:

- i vari calcolatori vengono visti come una singola risorsa computazionale;
- il server cluster possiede alte prestazioni poiché, invece di gravare su un'unica macchina *standalone*, suddivide il carico di lavoro su più macchine.

Per l'utente, il cluster è assolutamente trasparente, cioè tutta la notevole complessità, sia hardware che software, viene mascherata.

Il cluster utilizzato è stato messo a disposizione dal docente e, per accedere, occorre essere connessi alla rete dell'Ateneo di Verona.
Il login tramite *ssh* alla macchina master viene effettuato utilizzando come nome utente "*student*", ovvero attraverso il comando `ssh student@hadoop-compute0.di.univr.it`.
È anche possibile controllare come procede il processing sul cluster accedendo da interfaccia web attraverso l'url `http://hadoop-compute0.di.univr.it:50030`. Vengono cioè mostrate varie informazioni relative ai Map, ai Reduce ed ai job in esecuzione, completati e falliti.
Questo cluster è composto da quattro nodi, ovvero quattro calcolatori. Ognuno di questi può eseguire al massimo due map task e due reduce task, dunque la capacità massima del cluster utilizzato corrisponde a:

- 8 map task;
- 8 reduce task.

Pig Latin

Pig Latin è un linguaggio di programmazione ad alto livello che permette di manipolare con relativa facilità grandi quantità di dati ed ottenere come output il risultato delle elaborazioni fatte mediante il layer sottostante (per esempio MapReduce).

Esso produce codice Java per conto del programmatore (a cui può risultare difficoltoso), il quale potrà dunque concentrarsi solamente sulla trasformazione dei dati.
Pig Latin offre un approccio multi-query: ciò sta a significare che, mentre in Java alcune funzioni necessitano di parecchie linee di codice per essere definite, in Pig Latin è sufficiente scrivere molte meno righe per effettuare le stesse operazioni. Questo riduce notevolmente il tempo di sviluppo di software.
È inoltre un linguaggio SQL-like, così da risultare di facile utilizzo a chi ha già conosciuto il mondo relazionale, ed offre di base molti operatori e funzioni per le query, come filtri, join ed ordinamenti.

Pig Latin offre strutture dati particolari (alcune assenti in MapReduce) quali:

- *Atom*: è un semplice valore atomico, tra cui int, float, chararray;
- *Tuple*: è una sequenza ordinata di campi, ciascuno dei quali può essere di qualsiasi tipo;
- *Bag*: è una collezione di tuple con possibili duplicati e lo schema è flessibile, cioè non è necessario avere lo stesso numero e tipo di campi;
- *Map*: è una collezione di elementi in cui ognuno di questi ha una chiave associata e lo schema è flessibile.

Per eseguire il codice è necessario aprire due terminali ed eseguire i seguenti passaggi:

- sul primo terminale si avvia *Docker*, il container desiderato (specificandone l'ID), la bash del container e si attende qualche minuto affinché tutti i servizi siano disponibili;
- il secondo terminale viene utilizzato per copiare il file *.pig* relativo all'esercizio interessato nel container attraverso la stringa:

  ```
  docker cp --archive -L ./<filename>.pig cloudera:/<PATH>/<filename>.pig
  ```

- tornando al primo terminale, si avvia l'esercizio attraverso la bash del container, scrivendo la seguente stringa:

  ```
  pig -x mapreduce <filename>.pig
  ```

  In particolare, attraverso `-x mapreduce` viene specificato di eseguire Pig Latin in modalità mapreduce (che corrisponde a quella predefinita).

MapReduce

MapReduce è un modello di programmazione ispirato dalla programmazione funzionale e permette di svolgere operazioni su grandi quantità di dati.
La sua peculiarità è la capacità di parallelizzare i calcoli ed il lavoro sulle macchine a disposizione, nonché quella di adattarsi in modo autonomo all'ampliamento del sistema e quindi all'aggiunta di nuovi elaboratori.

Le funzioni principali che esso esegue sono:

- *Map*: essa prende in input l'intero dataset oppure una sua parte, in base a quello di cui necessita l'elaborazione. Lo script di Map viene distribuito sui vari calcolatori della rete, i quali lo eseguiranno sulla porzione di dataset che ospitano producendo dei risultati intermedi. La struttura dati alla base dei mapper è la coppia chiave-valore che possono essere interi, float, stringhe, semplici byte o strutture dati arbitrarie.
- *Shuffle and Sort*: rappresenta un "group by" distribuito e si occupa di raggruppare tutte le coppie con la stessa chiave e restituisce in output una lista di valori per ogni chiave. Se una stessa coppia occorre più di una volta, il valore associato può comparire più volte nell'output dello shuffle per quella chiave. Viene inoltre imposto un ordinamento rispetto alle chiavi.
- *Reduce*: riceve in input ciò che viene raggruppato e ordinato nello stage di Shuffle. Questa funzione ha il compito di combinare i risultati intermedi per produrre il risultato finale e lo scrive sul filesystem distribuito.

Per eseguire il codice è necessario aprire due terminali ed eseguire i seguenti passaggi:

- sul primo terminale si avvia *Docker*, il container desiderato (specificandone l'ID), la bash del container e si attende qualche minuto affinché tutti i servizi siano disponibili;
- il secondo terminale viene utilizzato per copiare il file *.jar* relativo all'esercizio interessato nel container attraverso la stringa:

  ```
  docker cp --archive -L ./<filename>.jar cloudera:/<PATH>/<filename>.jar
  ```

- tornando al primo terminale, si avvia l'esercizio attraverso la bash del container, scrivendo la seguente stringa:

  ```
  hadoop jar <filename>.jar progetto.<ClassName> /user/<username>/input
  /user/<username>/mapreduce/output
  ```

Esercizio 1

Per ogni numero di volo, calcolare il giorno della settimana avente la media dei ritardi di arrivo minore.

Input

L'input è il file *2008.txt* situato all'interno della directory */input*.

Implementazione in Pig Latin

Per prima cosa è stata calcolata la media dei ritardi di arrivo per ogni numero di volo utilizzando l'operatore *AVG* e successivamente il relativo minimo.
In particolare è stato utilizzato l'operatore *JOIN* per ricavare il giorno della settimana riportato nell'output.

```
record = LOAD '/user/ross/input/2008.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

-- Average
averageGroup = GROUP record BY (FlightNum, DayOfWeek);
average = FOREACH averageGroup GENERATE group, AVG(record.ArrDelay);
flattenAverage = FOREACH average GENERATE FLATTEN(group.FlightNum) AS FlightNum, FLATTEN(group.DayOfWeek) AS DayOfWeek,
$1 AS Average;

-- Minimum
minimumGroup = GROUP flattenAverage BY FlightNum;
minimum = FOREACH minimumGroup GENERATE group, MIN(flattenAverage.Average) AS Minimo;

bestDay = JOIN minimum BY (group, Minimo), flattenAverage BY (FlightNum, Average);
bestDay1 = FOREACH bestDay GENERATE FlightNum, DayOfWeek, Average;

STORE bestDay1 INTO '/user/ross/outputPig/BestDay1';
```

Implementazione in MapReduce

Il metodo della classe *Map* riceve in input il file da analizzare, ne esegue lo split per il carattere $\backslash n$ per ottenere i vari record dei voli. Viene poi eseguito lo split sulle virgole per individuare i singoli campi di ogni record. Per ogni riga avente il campo *ArrDelay* non vuoto viene generata una chiave corrispondente a *FlightNum* ed un valore dato da *DayofWeek* e *ArrDelay*.
Nello stage di Shuffle and Sort, questi dati vengono raggruppati e ordinati rispetto alle chiavi. Infine il risultato prodotto viene passato in input ai reducer.
Nel metodo della classe *Reduce*, con l'aiuto di un HashMap e un array di contatori, si calcola la media dei ritardi per ogni giorno della settimana per un determinato *FlightNum* (che corrisponde alla chiave). Dai valori ottenuti si estrae il ritardo minimo ed il giorno corrispondente.
L'output corrisponderà dunque alla coppia chiave-valore del tipo:

- chiave: numero identificativo del volo (*FlightNum*);
- valore: giorno con ritardo minimo (*dayMin*) e ritardo(*min*).

```
class Mapper
    method Map (docid key, doc lineText, context Context)
        line <- lineText.split("\n")
        for all line l
            words <- l.split(",")
            if (ArrDelay exists)
                EMIT (term FlightNum, value (DayOfWeek, ArrDelay))

class Reducer
    method Reduce (term t, counts [c1, c2, ...])
        sum <- 0
        dayMin <- 0
        min <- 9999
        average <- 0

        for all value c
            sum <- sum + c
            hashMap(DayOfWeek) <- sum
            ct[DayOfWeek]++

        for all DayOfWeek in hashMap
            average <- hashMap(DayOfWeek) / ct[DayOfWeek]
            if average < min
                min <- average
                dayMin <- DayOfWeek

        EMIT (term t, value (dayMin, min))
```

Output, osservazioni e statistiche

Eseguendo i file **sul container** in locale è possibile notare che tra i primi output del terminale compare:

```
INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat: Total input paths to process: 1
INFO org.apache.hadoop.mapreduce.JobSubmitter: number of splits: 6
```

Da questi si nota che il numero di mapper creati, per il file in input, è pari a sei.
Non è così invece per l'**esecuzione sul cluster**, poiché i *map task* generati sono undici.

La media dei tempi di esecuzione dei job (variando il numero di reducer) sono:

	Container		**Cluster**	
♯ *reducer*	*Pig Latin*	*MapReduce*	*Pig Latin*	*MapReduce*
1	$205s$	$75s$	$208s$	$69s$
2	$225s$	$75s$	$198s$	$67s$
4	$270s$	$77s$	$207s$	$59s$

Eseguendo lo script Pig vengono creati 3 job MapReduce in corrispondenza di precise istruzioni:

- il *LOAD* del file di input 2008.txt ed il *GROUP BY* usato nella variabile *averageGroup*;
- il *GROUP BY* usato nella variabile *minimumGroup*;
- il *JOIN* tra minimo e media.

L'output si trova in */outputPig/BestDay1* ed ogni record è strutturato come segue:

- *FlightNum*: numero del volo;
- *DayOfWeek*: giorno della settimana (1 = lunedì, 7 = domenica);
- *Average*: media dei ritardi di arrivo *ArrDelay* (in minuti).

Per ogni *FlightNum* presente nel file di input viene riportato solamente il giorno della settimana e la rispettiva media aventi il campo *Average* più basso.

Esercizio 2

Calcolare il giorno della settimana avente la media dei ritardi di arrivo minore dei voli.

Input

L'input è il file *2008.txt* situato all'interno della directory */input*.

Implementazione in Pig Latin

Per prima cosa è stata calcolata la media totale di tutti i ritardi di arrivo utilizzando l'operatore *AVG* e successivamente il relativo minimo.
Anche in questo esercizio è stato utilizzato l'operatore *JOIN* per ricavare il giorno della settimana riportato nell'output.

```
record = LOAD '/user/ross/input/2008.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

-- Average
averageGroup = GROUP record BY (DayOfWeek);
average = FOREACH averageGroup GENERATE group, AVG(record.ArrDelay);
flattenAverage = FOREACH average GENERATE FLATTEN(group) AS DayOfWeek, $1 AS Average;

-- Minimum
minimumGroup = GROUP flattenAverage ALL;
minimum = FOREACH minimumGroup GENERATE group, MIN(flattenAverage.Average) AS Minimo;

bestDay = JOIN minimum BY Minimo, flattenAverage BY Average;
bestDay1 = FOREACH bestDay GENERATE DayOfWeek, Average;

STORE bestDay1 INTO '/user/ross/outputPig/BestDay2';
```

Implementazione in MapReduce

Il metodo della classe *Map* è implementato esattamente come per l'esercizio precedente. L'unica variante è la chiave che qui è rappresentata dal campo *Year*. Per ogni riga avente il campo *ArrDelay* non vuoto viene generata una chiave (*Year*) ed un valore dato da *DayOfWeek* e *ArrDelay*.
Nello stage di Shuffle and Sort, questi dati vengono raggruppati e ordinati rispetto alle chiavi. Infine il risultato prodotto viene passato in input ai reducer.
Nel metodo della classe *Reduce*, con l'aiuto di un HashMap e un array di contatori, si calcola la media dei ritardi per ogni giorno della settimana in un determinato anno (che corrisponde alla chiave). Dai valori ottenuti si estrae il ritardo minimo e il giorno corrispondente.
L'output corrisponderà dunque alla coppia chiave-valore del tipo:

- chiave: anno in cui è stato effettuato il volo (*Year*);
- valore: giorno con ritardo minimo (*dayMin*) e ritardo (*min*).

```
class Mapper
    method Map (docid key, doc lineText, context Context)
        line <- lineText.split("\n")

        for all line l
            words <- l.split(",")
            if (ArrDelay exists)
                EMIT (term Year, value (DayOfWeek, ArrDelay))



class Reducer
    method Reduce (term t, counts [c1, c2, ...])
        sum <- 0
        dayMin <- 0
        min <- 9999
        average <- 0

        for all value c
            sum <- sum + c
            hashMap(DayOfWeek) <- sum
            ct[DayOfWeek]++

        for all DayOfWeek in hashMap
            average <- hashMap(DayOfWeek) / ct[DayOfWeek]
            if average < min
                min <- average
                dayMin <- DayOfWeek

        EMIT (term t, value (DayMin, min))
```

Output, osservazioni e statistiche

Eseguendo i file **sul container** è possibile notare che tra i primi output del terminale compare:

```
INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat: Total input paths to process: 1
INFO org.apache.hadoop.mapreduce.JobSubmitter: number of splits: 6
```

Da questi si nota che il numero di mapper creati, per il file in input, è pari a sei.
Non è così invece per l'**esecuzione sul cluster**, poiché i *map task* generati sono undici.

La media dei tempi di esecuzione dei job (variando il numero di reducer) sono:

	Container		**Cluster**	
♯ *reducer*	*Pig Latin*	*MapReduce*	*Pig Latin*	*MapReduce*
1	$180s$	$83s$	$168s$	$62s$
2	$188s$	$83s$	$182s$	$63s$
4	$209s$	$84s$	$173s$	$66s$

Per questo esercizio è stata creata una variante del codice in Pig Latin. Ovvero si è deciso di evitare l'utilizzo dell'operatore *JOIN* che, talvolta, crea un numero elevato (e non necessario) di job MapReduce.
La variante è stata pensata sostituendo le ultime 5 righe del codice con le seguenti:

```
minimum = ORDER flattenAverage BY Average ASC;
bestDay = LIMIT minimum 1;
STORE bestDay INTO '/user/ross/outputPig/BestDay2_1';
```

Ovvero viene fatto un ordinamento crescente sulla media dei ritardi (*Average*) e quindi tenendone solamente il primo record.
Tuttavia la sua esecuzione media nel container è pari a $204s$, ovvero vi è un aumento di $24s$.
Il tempo di esecuzione è aumentato principalmente per due motivi:

- i record su cui si opera sono solamente sette (corrispondono ai giorni della settimana);
- la condizione del *JOIN* è molto semplice.

Eseguendo lo script Pig vengono creati 3 job MapReduce in corrispondenza di precise istruzioni:

- il *LOAD* del file di input 2008.txt ed il *GROUP BY* usato nella variabile *averageGroup*;
- il *GROUP BY* usato nella variabile *minimumGroup*;
- il *JOIN* tra minimo e media.

L'output si trova in */outputPig/BestDay2* e contiene un solo record strutturato come segue:

- *DayOfWeek*: giorno della settimana (1 = lunedì, 7 = domenica);
- *Average*: media dei ritardi di arrivo *ArrDelay* (in minuti).

Tale record indica il giorno della settimana avente la media dei ritardi più basso, ovvero in questo caso il giorno è sabato con una media di quasi 6 minuti.

Esercizio 3

Stabilire se, in generale, i voli più vecchi subiscono maggiori ritardi rispetto a quelli più recenti.

Input

I file di input scelti sono otto e si dividono in:

- recenti: *2008.txt, 2007.txt, 2006.txt, 2005.txt*;
- vecchi: *1990.txt, 1989.txt, 1988.txt, 1987.txt.*

Questi file sono tutti situati all'interno della directory */input.*

Implementazione in Pig Latin

Per prima cosa, per ognuno dei quattro anni considerati recenti, è stata calcolata la media annua di tutti i ritardi di arrivo utilizzando l'operatore *AVG* e successivamente, facendo uso sempre di tale operatore, ne è stata calcolata la media ottenendo quindi il ritardo medio di arrivo per gli anni compresi tra il 2005 ed il 2008.
Lo stesso procedimento è stato fatto per gli anni considerati vecchi.

```
year2008 = LOAD '/user/ross/input/2008.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

year2007 = LOAD '/user/ross/input/2007.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

year2006 = LOAD '/user/ross/input/2006.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

year2005 = LOAD '/user/ross/input/2005.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

year1987 = LOAD '/user/ross/input/1987.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

year1988 = LOAD '/user/ross/input/1988.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
```

```
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

year1989 = LOAD '/user/ross/input/1989.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

year1990 = LOAD '/user/ross/input/1990.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int, Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

-- New
group2008 = GROUP year2008 BY Year;
average2008 = FOREACH group2008 GENERATE AVG(year2008.ArrDelay) AS Average;
group2007 = GROUP year2007 BY Year;
average2007 = FOREACH group2007 GENERATE AVG(year2007.ArrDelay) AS Average;
group2006 = GROUP year2006 BY Year;
average2006 = FOREACH group2006 GENERATE AVG(year2006.ArrDelay) AS Average;
group2005 = GROUP year2005 BY Year;
average2005 = FOREACH group2005 GENERATE AVG(year2005.ArrDelay) AS Average;
new = UNION average2008, average2007, average2006, average2005;
newgroup = GROUP new ALL;
newAverage = FOREACH newgroup GENERATE 'New planes delay:' AS String, AVG(new.Average) AS Average;

-- Old
group1987 = GROUP year1987 BY Year;
average1987 = FOREACH group1987 GENERATE AVG(year1987.ArrDelay) AS Average;
group1988 = GROUP year1988 BY Year;
average1988 = FOREACH group1988 GENERATE AVG(year1988.ArrDelay) AS Average;
group1989 = GROUP year1989 BY Year;
average1989 = FOREACH group1989 GENERATE AVG(year1989.ArrDelay) AS Average;
group1990 = GROUP year1990 BY Year;
average1990 = FOREACH group1990 GENERATE AVG(year1990.ArrDelay) AS Average;
old = UNION average1987, average1988, average1989, average1990;
oldgroup = GROUP old ALL;
oldAverage = FOREACH oldgroup GENERATE 'Old planes delay:' AS String, AVG(old.Average) AS Average;

result = UNION newAverage, oldAverage;
STORE result INTO '/user/ross/outputPig/OlderPlanes';
```

Implementazione in MapReduce

Il metodo della classe *Map* riceve in input il file da analizzare, ne esegue lo split per il carattere $\backslash n$ e sulle virgole per individuare i campi di ogni record. Si analizza il campo *Year* e se i voli sono stati fatti in un anno successivo al 2004 si genera una chiave per identificarli come *New Planes* (per anni precedenti al 2005 vengono identificati come *Old Planes*). Come valore verrà considerato il campo *ArrDelay*.
Nello stage di Shuffle and Sort, questi dati vengono raggruppati e ordinati rispetto alle chiavi. Infine il risultato prodotto viene passato in input ai reducer.
Nel metodo della classe *Reduce* si calcola la media dei ritardi per tutti gli *Old Planes* e i *New Planes*. L'output corrisponderà dunque alla coppia chiave-valore del tipo:

- chiave: tipo di aereo (*New/Old plane*);
- valore: media dei ritardi per quel tipo di aerei (*average*).

```
class Mapper
    method Map (docid key, doc lineText, context Context)
        line <- lineText.split("\n")
        for all line l
           words <- l.split(",")
            if Year > 2004
                plane <- "New Planes"
            else
                plane <- "Old Planes"
            if (ArrDelay exists)
                EMIT (term plane, value ArrDelay)
```

```
class Reducer
    method Reduce (term t, counts [c1, c2, ...])
        ct <- 0
        average <- 0
        for all value c
            average <- average + c
            ct++
        average <- average / ct
        EMIT (term t, value (average))
```

Output, osservazioni e statistiche

Eseguendo i file **sul container** è possibile notare che tra i primi output del terminale compare:

```
INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat: Total input paths to process: 8
INFO org.apache.hadoop.mapreduce.JobSubmitter: number of splits: 35
```

Da questi si nota che il numero di mapper creati, per il file in input, è pari a trentacinque.
Non è così invece per l'**esecuzione sul cluster**, poiché i *map task* generati sono settanta.
Questa differenza è dovuta alla dimensione dei blocchi in cui i file vengono suddivisi (64MB per il Cluster, 128MB per il container).

La media dei tempi di esecuzione dei job (variando il numero di reducer) sono:

	Container		**Cluster**	
♯ *reducer*	*Pig Latin*	*MapReduce*	*Pig Latin*	*MapReduce*
1	$858s$	$348s$	$624s$	$227s$
4	$1155s$	$349s$	$639s$	$218s$
8	$1378s$	$353s$	$659s$	$216s$

Eseguendo lo script Pig Latin vengono creati 11 job:

- i primi 8 corrispondono ciascuno al *LOAD* di un file di input ed il rispettivo *GROUP BY*;
- il *GROUP BY* usato nella variabile *newGroup*;
- il *GROUP BY* usato nella variabile *oldGroup*;
- la *UNION* usata nella variabile *result*.

L'output si trova in */outputPig/OldPlanes* e contiene due informazioni:

- *New planes delay*: media totale dei ritardi di arrivo dei voli recenti (in minuti), ovvero degli anni compresi tra il 2005 ed il 2008;
- *Old planes delay*: media totale dei ritardi di arrivo dei voli vecchi (in minuti), ovvero degli anni compresi tra il 1987 ed il 1990.

La media dei ritardi di arrivo dei voli recenti (*New planes delay*) è di 8 minuti e mezzo circa e quella relativa ai voli vecchi (*Old planes delay*) è di quasi 8 minuti.
Quindi è possibile affermare che, mediamente, i voli più vecchi non subiscono maggiori ritardi rispetto a quelli più recenti in quanto c'è una sottile differenza tra le due medie ottenute dall'analisi.

Esercizio 4

Stabilire quanto bene il tempo meteorologico prevede i ritardi di arrivo dei voli.

Input

L'input è il file *2008.txt* situato all'interno della directory */input*.

Implementazione in Pig Latin

Per prima cosa sono stati filtrati tutti i record del file di input tenendo solamente quelli aventi il campo *WeatherDelay* non nullo e diverso dalla stringa "*NA*" attraverso l'operatore *FILTER*.
È stata poi calcolata la media totale di tutti i ritardi di arrivo dei voli utilizzando l'operatore *AVG*.
Successivamente, facendo sempre uso dello stesso operatore, è stata calcolata la media di tutti i ritardi causati dal meteo.

```
record = LOAD '/user/ross/input/2008.txt' using PigStorage(',') AS (
Year:int, Month:int, DayofMonth:int, DayOfWeek:int, DepTime:chararray, CRSDepTime:int, ArrTime:int, CRSArrTime:int,
UniqueCarrier:chararray, FlightNum:int, TailNum:chararray, ActualElapsedTime:int, CRSElapsedTime:int, AirTime:int,
ArrDelay:int, DepDelay:int, Origin:chararray, Dest:chararray, Distance:int, TaxiIn:int, TaxiOut:int,Cancelled:int,
CancellationCode:chararray, Diverted:chararray, CarrierDelay:chararray, WeatherDelay:chararray, NASDelay:chararray,
SecurityDelay:chararray, LateAircraftDelay:chararray);

-- Arrival delay
recordOk = FILTER record BY (WeatherDelay != 'NA' AND WeatherDelay != '' AND WeatherDelay IS NOT NULL);
newRecord = FOREACH recordOk GENERATE Year, Month, DayofMonth, DayOfWeek, DepTime, CRSDepTime, ArrTime, CRSArrTime,
UniqueCarrier, FlightNum, TailNum, ActualElapsedTime, CRSElapsedTime, AirTime, ArrDelay, DepDelay, Origin, Dest,
Distance, TaxiIn, TaxiOut, Cancelled, CancellationCode, Diverted, CarrierDelay, (int)WeatherDelay, NASDelay,
SecurityDelay, LateAircraftDelay;
averageGroup = GROUP newRecord ALL;
arrivalAverage = FOREACH averageGroup GENERATE 'Arrival delay:' AS String, AVG(newRecord.ArrDelay) AS Average;

-- Weather delay
weatherAverage = FOREACH averageGroup GENERATE 'Weather delay:' AS String, AVG(newRecord.WeatherDelay) AS Average;

result = UNION arrivalAverage, weatherAverage;
STORE result INTO '/user/ross/outputPig/WeatherDelay';
```

Implementazione in MapReduce

Il metodo della classe *Map* riceve in input il file da analizzare, ne esegue lo split per il carattere $\backslash n$ e sulle virgole per individuare i campi di ogni record. Per ogni riga avente i campi *ArrDelay* e *WeatherDelay* non vuoti viene generata una chiave corrispondente a *Year* ed un valore dato da *ArrDelay* e *WeatherDelay*.
Nello stage di Shuffle and Sort, questi dati vengono raggruppati e ordinati rispetto alle chiavi. Infine il risultato prodotto viene passato in input ai reducer.
Nel metodo della classe *Reduce* vengono calcolate le medie di *ArrDelay* e *WeatherDelay*. L'output corrisponderà dunque alla coppia chiave-valore del tipo:

- chiave: anno in cui è stato effettuato il volo (*Year*);
- valore: media dei ritardi all'arrivo (*avgArrDelay*) e media dei ritardi dovuti al tempo meteorologico (*avgWeatherDelay*).

```
class Mapper
    method Map (docid key, doc lineText, context Context)
        line <- lineText.split("\n")
        for all line l
            words <- l.split(",")

        if (ArrDelay exists)
            EMIT (term Year value (ArrDelay, WeatherDelay))

class Reducer
    method Reduce (term t, counts [c1, c2, ...])
        ct <- 0
        avgArrDelay <- 0
        avgWeatherDelay <- 0
        for all value
            avgArrDelay <- avgArrDelay + ArrDelay
            avgWeatherDelay <- avgWeatherDelay + WeatherDelay
            ct++
        average <- average / ct
        EMIT (term t, value (avgArrDelay, avgWeatherDelay))
```

Output, osservazioni e statistiche

Eseguendo i file **sul container** è possibile notare che tra i primi output del terminale compare:

```
INFO org.apache.hadoop.mapreduce.lib.input.FileInputFormat: Total input paths to process: 1
INFO org.apache.hadoop.mapreduce.JobSubmitter: number of splits: 6
```

Da questi si nota che il numero di mapper creati, per il file in input, è pari a sei.
Non è così invece per l'**esecuzione sul cluster**, poiché i *map task* generati sono undici.
Questa differenza è dovuta alla dimensione dei blocchi in cui i file vengono suddivisi (64MB per il Cluster, 128MB per il container).

La media dei tempi di esecuzione dei job (variando il numero di reducer) sono:

	Container		**Cluster**	
♯ *reducer*	*Pig Latin*	*MapReduce*	*Pig Latin*	*MapReduce*
1	$266s$	$67s$	$319s$	$66s$
2	$289s$	$67s$	$310s$	$67s$
4	$354s$	$68s$	$264s$	$68s$

Eseguendo lo script Pig vengono creati 2 job MapReduce in corrispondenza di precise istruzioni:

- il *LOAD* del file di input 2008.txt ed il *GROUP BY* usato nella variabile *averageGroup*;
- il *FOREACH* usato nella variabile *arrivalAverage*.

L'output si trova in */outputPig/WeatherDelay* e contiene due informazioni:

- *Arrival delay*: media dei ritardi di arrivo dei voli (in minuti);
- *Weather delay*: media dei ritardi relativi al meteo (in minuti).

La media dei ritardi di arrivo dei voli (*Arrival delay*) è di 57 minuti circa e quella relativa ai ritardi causati dal tempo meteorologico (*Weather delay*) è di 3 minuti circa. Quindi è possibile stabilire che, mediamente, il meteo influisce per il 5,3% sul ritardo di arrivo dei voli.
Inoltre si può notare che il campo *ArrDelay* non è mai nullo quando è presente il campo *WeatherDelay*. Dunque il ritardo meteorologico è sempre collegato al ritardo di arrivo.

Conclusioni

Per lo sviluppo di questo progetto è stato necessario dedicare inizialmente del tempo per l'apprendimento e la familiarizzazione con il nuovo linguaggio Pig Latin. Ciò è stato possibile attraverso il materiale didattico fornito dal docente (in laboratorio e durante le lezioni frontali) ed alcune guide online:

- documentazione di *Pig 0.17.0*: `http://pig.apache.org/docs/r0.17.0/`;
- guida di *TutorialsPoint*: `https://www.tutorialspoint.com/apache_pig/index.htm`;
- slide del docente: *03b-Pig/Hive.pdf*, *LabPIG-instructions.pdf* ed esercizi svolti in laboratorio.

Per quanto riguarda MapReduce è stato necessario un ripasso attraverso il materiale del docente e qualche approfondimento mediante guide e documentazioni online:

- documentazione *hadoop*: `https://hadoop.apache.org/docs/r1.2.1/mapred_tutorial.html`
- guida di *TutorialsPoint*: `https://www.tutorialspoint.com/map_reduce/map_reduce_quick_guide.htm`;
- slide del docente: *01b-MapReduce.pdf* ed esercizi svolti in laboratorio.

È stato inoltre utile il pdf MapReduce-Laboratory.pdf messo a disposizione dal docente per capire come interfacciarsi con il container ed eseguire azioni su di esso.
Qui però è stato incontrato un ostacolo che diverse volte ci impediva di testare il nostro codice, ovvero i servizi non disponibili del container utilizzato (come riscontrato anche durante le lezioni). In più occasioni è stato necessario:

- attendere diversi minuti fino all'attivazione dei servizi necessari;
- riavviare più volte Docker quando, dopo diversi minuti, i servizi non risultavano ancora accessibili;
- lasciare il terminale acceso diversi giorni per evitare di dover riavviare *Docker* e di conseguenza rischiare di incontrare nuovamente i problemi indicati;
- durante l'esecuzione dei test sono stati riscontrati riavvii improvvisi del calcolatore in uso.

Dopo ciascuna esecuzione è possibile notare che incrementando il numero di reducer può non aumentare in alcun modo le prestazioni (anzi, può causarne un notevole peggioramento).
Ciò va a confutare una credenza presente tra gli utenti: un maggiore parallelismo implica migliori prestazioni. Infatti, aumentare i reducer per carichi di lavoro non eccessivi, porterà ciascun mapper a partizionare il suo output in base ai reducer presenti. Il risultato finale, in questi casi, consiste in una quantità eccessiva di piccoli file (anche vuoti).

Per modificare il numero di reducer è sufficiente inserire le seguenti righe di codice all'interno dei file Pig/MapReduce:

- Pig: `SET default_parallel <#reducers>;`
- MapReduce: `job.setNumReduceTasks(int tasks);`

Confronto Container - Cluster

Tentando di testare i file di MapReduce è stato riscontrato un problema nella funzione *main()* di ogni programma. Ciò è dovuto alla chiamata al metodo *run* per la generazione del Job.
Questo errore è emerso a causa delle differenti versioni di Hadoop presenti nel Cluster (*hadoop 0.20.2*) e nel Container (*hadoop 2.6.0*) che richiedono istruzioni diverse per poter eseguire la stessa funzione (il comando per la versione meno recente risulta deprecato per quelle nuove):

- Cluster (hadoop 0.20.2): `Job job = new Job (getConf());`
- Container (hadoop 2.6.0): `Job job = Job.getInstance(getConf(), "exercise");`

Osservando i tempi di esecuzione dei test di Pig Latin e MapReduce si può notare un generale miglioramento delle prestazioni, soprattutto durante la fase di map.
Ciò è dovuto alla suddivisione dei task, i quali sono distribuiti sui quattro calcolatori presenti nel cluster. Al contrario, lavorando sul container in locale, essi vengono suddivisi sui *core* del processore della macchina utilizzata.
Per quanto riguarda invece la variazione del numero di reduce, si è notato un miglioramento dei tempi solamente per MapReduce; per gli script in Pig Latin si sono registrati dei graduali peggioramenti (proporzionali all'aumento del numero dei task di reduce).
La principale causa di quanto appena osservato è la differenza tra il codice MapReduce implementato e quello prodotto automaticamente durante la conversione degli script Pig in MapReduce.
Possiamo quindi dedurre che aumentare il numero dei reducer non è sempre necessario e conveniente. In particolare negli esercizi 2 e 4, aventi file di output di piccole dimensioni, il peggioramento si evidenzia anche per i file MapReduce.

Confronto Pig Latin - MapReduce

Dopo aver implementato e testato gli esercizi nelle versioni Pig Latin e MapReduce, si possono dedurre dei pro e contro per ciascun approccio.

Vantaggi e svantaggi di Pig Latin:

- Non è necessario produrre codice Java ma ci si concentra unicamente sull'approccio multi-query che richiede molte meno righe di codice per svolgere determinate funzioni.
- L'utente, grazie all'approccio multi-query, potrà concentrarsi soprattutto su *cosa* ottenere (a livello informativo) da un dataset e non a *come* ottenerlo (con l'approccio MapReduce è necessario occuparsi di entrambi gli aspetti).
- È necessario apprendere un nuovo linguaggio meno conosciuto che non ha una grande community di supporto per eventuali problemi.
- Le prestazioni di elaborazione dei dati sono generalmente inferiori poiché il codice Pig deve essere convertito in MapReduce.
- I problemi difficili possono richiedere implementazioni complesse che fanno uso, ad esempio, di raggruppamenti o join molto complessi; in questi casi, verrebbero generati un numero elevato (e non necessario) di job MapReduce.
- È facile da scrivere, leggere e manipolare perché ha un più alto livello di astrazione.
- Scrivere uno script Pig Latin richiede un tempo relativamente basso, mentre non è così per eseguirlo.
- Per l'elaborazione di dati non strutturati è richiesta una tecnica differente, chiamata UDF (*User Defined Functions* - funzioni definite dall'utente).
- È abbastanza improbabile produrre bug a livello Java scrivendo del codice in Pig.

Vantaggi e svantaggi di MapReduce:

- Le prestazioni in termini di tempo sono migliori rispetto a Pig perché non è necessaria nessuna conversione del codice.
- La community di supporto è migliore rispetto a quella di Pig poiché vi è un maggior numero di utilizzatori e di forum dedicati.
- Alcuni problemi non possono essere risolti attraverso Pig, altri invece possono essere difficili da pensare ed implementare e diventa conveniente l'approccio MapReduce.
- La scrittura delle funzioni di interesse richiede parecchie righe di codice rispetto a Pig.
- È necessaria una buona conoscenza del linguaggio di programmazione Java per sviluppare un codice performante, saper fare debugging e manutenzione del codice.
- Scrivere codice MapReduce richiede un tempo maggiore rispetto ad uno script in Pig; si ha però un vantaggio al momento dell'esecuzione.
- La presenza di un numero elevato di righe comporta una maggiore probabilità di produrre bug ed errori che sono più difficili da individuare e risolvere.