Relazione finale primo progetto Big Data

di Del Nero Claudio e Goggia Alessio - "Sparkiamoci in due"

Introduzione

In questo primo progetto ci è stato chiesto di sperimentare diverse tecnologie di Big Data, in vari scenari e con file di input di dimensione variabile, sia in locale sia su cluster.
In locale i diversi job sono stati eseguiti su una macchina Ubuntu 16.04 64 bit reale, con 8 Gb di RAM, 80 Gb di hard disk, processore i7 quad-core a 2,2 GHz; mentre i job Hive sono stati eseguiti su una macchina Ubuntu 16.04 32 bit virtuale (perdendo, come si potrà vedere dai risultati finali, in efficienza).
Come linguaggio di programmazione abbiamo scelto Java (versione 1.8 potendo così sfruttare le espressioni lambda che abbiamo trovato molto comode ed efficienti per le varie esecuzioni), e come IDE Eclipse neon.3; per quanto riguarda le versioni delle varie tecnologie, abbiamo usato Hadoop 2.7.3, Hive 2.1.1 e Spark 2.1.0.
Su cluster abbiamo scelto di utilizzare il nuovo cluster dell'Università, mettendo da parte, almeno per ora, Amazon Educate AWS.
Nel corso della relazione andremo a descrivere in modo strutturato il codice scritto per i vari job per ogni tecnologia (MapReduce, Hive e Spark), fino ad arrivare infine ai risultati finali descritti con grafici e tabelle, che mostreranno i tempi impiegati per l'esecuzione, sia in locale sia su cluster.

1. Sviluppo primo job

Il primo job da sviluppare deve essere in grado di generare per ogni mese i 5 prodotti che hanno ricevuto lo score più alto, ordinando il tutto temporalmente.

1.1. MapReduce

Per quanto riguarda MapReduce abbiamo scelto di definire due task MapReduce consecutivi:

- nella prima Map abbiamo preso come chiavi le date insieme ai prodotti, in modo tale da far raccogliere al sistema, grazie al seguente Shuffle and Sort, tutti gli score relativi ad un prodotto in un determinato mese dell'anno. Quindi la Reduce del primo task ha l'unico obiettivo di calcolare la media degli score per ogni mese dell'anno per ogni prodotto;
- la Map del secondo task, invece, raccoglie tutti gli "average score", in ogni singolo mese dell'anno, di tutti i prodotti, in modo tale da trovare i prodotti che hanno ricevuto lo score medio migliore. Per questo motivo abbiamo dovuto dividere l'id del prodotto dalla singola data, la quale è diventata la nuova chiave della Map. Come valore abbiamo usato un oggetto, creato da noi *ad hoc*, che accorpa prodotto e score (d'ora in poi ProdScore), istanziato come Writable. Dopo il lavoro dello Shuffle and Sort, che ha raggruppato tutti i ProdScore in una determinata data, li abbiamo ordinati

in maniera decrescente (usando la funzione sort di Collections) e infine filtrato prendendo i primi 5 per ogni mese.

1.2. Hive

Una volta capito il funzionamento di questa tecnologia, l'implementazione è risultata abbastanza semplice: abbiamo definito un file job1.hql, in cui, distinguiamo quattro passaggi principali. I primi due sono comuni per ogni job di cui parleremo in seguito (e per questo motivo non ripeteremo):

- creazione di una tabella su cui poi importare il file .csv di input
- importazione della funzione di conversione da UNIX time in formato "yyyy-mm" tramite JAR
- creazione di una vista che calcola la media di ogni prodotto in ogni singolo mese dell'anno
- creazione della tabella finale che, tramite la funzione ROW_NUMBER(), e in particolare a ciò che viene specificato all'interno della sua OVER, itera i vari mesi dell'anno (grazie alla PARTITION BY) e ordina gli "average score" per ogni mese (grazie alla ORDER BY). Nella WHERE si specifica quanti "record" vogliamo raccogliere, al massimo, per ogni mese dell'anno.

```
CREATE TABLE IF NOT EXISTS foodReview (
      id STRING,
      productId STRING,
      userId STRING,
      profileName STRING,
      helpNum STRING,
      helpDen STRING,
      score INT,
      provisioned STRING,
      summary STRING,
      text STRING)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

LOAD DATA LOCAL INPATH '/home/claudio/Downloads/amazon/1999_
2006.csv' INTO TABLE foodReview;
      add jar /home/claudio/Desktop/unix_date.jar;

CREATE TEMPORARY FUNCTION unix_date AS
'com.example.hive.udf.Unix2Date';

CREATE VIEW average_product AS SELECT dateFormat.year_month,
dateFormat.productId, AVG(dateFormat.score) AS averageProd
FROM ( SELECT unix_date(provisioned) AS year_month, productId,
score FROM foodReview) dateFormat GROUP BY
dateFormat.year_month, dateFormat.productId;

CREATE TABLE top_5_final AS SELECT top_5.year_month, top_
5.productId, top_5.averageProd FROM ( SELECT year_month,
productId, ROW_NUMBER() OVER (PARTITION BY
average_product.year_month ORDER BY averageProd DESC) AS rank,
averageProd FROM average_product) top_5 WHERE rank<6;
```

1.3. Spark

In Spark abbiamo definito più metodi in modo tale da separare le responsabilità in maniera abbastanza delineata:

1. Il primo metodo è *loadData()*, che legge il file .csv dato in input, restituendo un RDD composto da una stringa e da un oggetto ProdScore (questa volta istanziato come Serializable, dovendo scrivere su disco reale e non in hdfs), ove la stringa rappresenta il mese dell'anno, mentre il ProdScore è composto dall'id del prodotto e dallo score relativo al prodotto. Per creare l'RDD viene usato il metodo *mapToPair,* che legge ogni riga del file .csv e la splitta in vari campi, separati da \t, inserendoli in un array. In questo modo per accedere ad ogni campo del file, basterà passare l'indice relativo alla posizione del campo: in questo caso abbiamo il campo 8 per il mese dell'anno, il campo 2 per l'id prodotto ed il campo 7 per lo score. Questi ultimi due valori verranno utilizzati per creare l'oggetto ProdScore. Il formato dell' RDD di output di questo metodo è "data-ProdScore", cioè *JavaPairRDD<String,ProdScore>*.
2. Il secondo è *getMapOfProductsForEachMonthOrdered()*, che chiaramente sfrutta il metodo appena descritto per raccogliere i dati in input ed esegue i seguenti passaggi:
   - fa una *groupby* sull'input in base alla data generando un RDD che per ogni mese dell'anno prevede un Set *Iterable* di ProdScore relativi a quel mese specifico;
   - il secondo passaggio è una *mapToPair* dell' RDD generato in precedenza, che, per ogni mese di ogni anno, crea una mappa con *chiave* = id del prodotto e con *valore* = lista di score relativi al prodotto specificato come chiave. Quindi scorre tutti i ProdScore di ogni singolo mese di ogni anno e popola le mappe associate ad ogni mese.
   - infine si effettua una *sortByKey* per ordinare il tutto in base alla data.
3. Il terzo metodo si chiama *getProductsForEachMonthAVGScore()*, che prende in input l'output del metodo 2 ed effettua una *mapToPair* su di esso, la quale ha l'obiettivo di calcolare la media degli score di ogni prodotto per ogni mese dell'anno, salvandoli in una lista di ProdScore relativi ad un singolo mese dell'anno:
   si scorrono le chiavi di ogni mappa in modo tale da prendere l'id del prodotto e la lista di score ad esso associati, su cui viene calcolata la media. Viene creato un ProdScore, su cui vengono settati questi due parametri. Il ProdScore viene aggiunto in una lista di appoggio (una per ogni mese dell'anno).
   L'output della *mapToPair* è perciò un RDD composto da mese dell'anno e lista di "average score" relativi a quel mese.
4. Il quarto metodo denominato *getProductsForEachMonthOrdered()* utilizza una lista di appoggio per ordinare gli "average score" di ogni mese dell'anno in maniera decrescente
5. Il quinto, come dice il nome *getTop5()*, sempre tramite *mapToPair*, filtra per ogni mese dell'anno i primi 5 risultati, provenienti dalla lista ordinata di average score, ottenuta grazie al metodo precedentemente descritto

6. Il passo finale dovrà soltanto prendere il risultato delle elaborazioni precedenti, e stampare il risultato in un file di output su disco. La stampa, con il metodo *map*, si va a ciclare tutto l'RDD ottenuto dal metodo precedente, e salvare il suo contenuto in una stringa di appoggio. Questa stringa avrà 3 campi separati da \t, dove il primo rappresenta il mese dell'anno, il secondo l'id del prodotto ed il terzo lo score del prodotto. Il metodo *map* ritornerà un RDD *String* composto da tutti i campi descritti prima, e con il metodo *saveAsTextFile*, salviamo tutte le stringhe nel file di output passato come parametro (andrà quindi passato anche il path di output oltre quello di input al momento del lancio del job).
7. Il main quindi non dovrà fare altro che controllare se sono stati forniti tutti gli argomenti necessari per l'esecuzione del job, inizializzare il tempo di inizio del job, creare il job e fargli eseguire il metodo di stampa, che mano mano andrà a richiamare i vari metodi a ritroso, a partire dal caricamento del file csv in input, fino ad arrivare al salvataggio dei top 5 e alla stampa finale su file, incluso il tempo necessario per l'esecuzione del job.

1.4. Esempio di output

Di seguito verrà mostrato un esempio di output, per la precisione le prime 10 righe di uno dei file, generato dall'esecuzione del job Spark.

```
1999-10    0006641040    5.0
1999-12    B00004CXX9    5.0
1999-12    B00004CI84    5.0
1999-12    B00004RYGX    5.0
2000-01    B00002N8SM    5.0
2000-01    B00004CXX9    3.6666666666666665
2000-01    B00004CI84    3.0
2000-01    B00004RYGX    3.0
2000-02    B00004CXX9    4.0
2000-02    B00004CI84    4.0
```

2. Sviluppo secondo job

Il secondo job deve generare i 10 prodotti preferiti da ogni utente, ordinando il tutto in base all'id dell'utente.

2.1. MapReduce

L'approccio pensato per sviluppare questo job in MapReduce è più semplice di quello utilizzato per il primo e prevede un singolo task MapReduce .

- La map legge il file in input e prende come chiave l'id dell'utente, ossia il terzo campo del file, e come valore il ProdScore opportunamente settato con l'id del prodotto (secondo campo) e lo score ricevuto dal relativo utente (settimo campo).

- A questo punto il reducer crea prima un *ArrayList* vuoto in cui si inseriscono man mano i vari ProdScore per ogni utente (dato che ogni utente può aver recensito più prodotti). Si iterano con un forEach i ProdScore, che verranno aggiunti all' *ArrayList* creato inizialmente, settando sia l'id che lo score assegnato. Dopo aver popolato la lista, viene ordinata in base allo score decrescente, mediante il metodo *sort* di *Collections*, per ogni utente, per poi infine andare a memorizzare nel context soltanto i top k, che nel nostro caso sono i top 10.
- Il main crea la Configuration, il job, e setta le diverse classi per jar, map, reduce ed output, poi esegue la map e la reduce, a partire dal file di input, scrivendo il risultato di output in hdfs, ordinato per id utente. Il risultato finale vedrà appunto una serie di id utenti ordinati in modo crescente, ove per ognuno di essi saranno elencati i primi 10 prodotti con relativo score.

2.2. Hive

La query hive per il secondo job è decisamente più semplice da realizzare rispetto alle altre tecnologie.

- Dopo l'importazione del file csv e della funzione di conversione, andremo a creare una tabella che avrà come colonne l'id dell'utente, l'id del prodotto e lo score. Per definire i vari campi della tabella, si seleziona l'id dell'utente e del prodotto. Per generare i primi 10 score, viene richiamata la funzione ROW_NUMBER(), utilizzata e descritta già nel job 1, che ci consente con un solo comando di partizionare le varie righe per id utente, ed ordinarle per score decrescente. In questo modo avremo una tabella ordinata per id utente crescente, in cui per ogni utente viene specificato l'id prodotto e lo score relativo (ordinato per score decrescente). Dovendo il job restituire la top 10, per tagliare la lista di prodotti, basta specificare nella clausola WHERE il limite a 11 (con 11 escluso), in questo modo appena si legge il decimo valore relativo all'utente, si passerà al successivo.

```
CREATE TABLE IF NOT EXISTS foodReview (
      id STRING,
      productId STRING,
      userId STRING,
      profileName STRING,
      helpNum STRING,
      helpDen STRING,
      score INT,
      provisioned STRING,
      summary STRING,
      text STRING)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

LOAD DATA LOCAL INPATH '/home/claudio/Downloads/amazon/1999_
2006.csv' INTO TABLE foodReview;
      add jar /home/claudio/Desktop/unix_date.jar;

CREATE TEMPORARY FUNCTION unix_date AS
'com.example.hive.udf.Unix2Date';


CREATE TABLE top_10_products AS SELECT top_10.userId, top_
10.productId, top_10.score FROM ( SELECT userId, productId,
ROW_NUMBER() OVER (PARTITION BY userId ORDER BY score DESC) AS
rank, score FROM foodReview) top_10 WHERE rank<11;
```

2.3. Spark

L'approccio seguito in Spark è molto simile a quello adottato con MapReduce.

- Il primo passo, rappresentato dal metodo *loadData(),* è identico al primo job con l'unica differenza che, al posto del mese dell'anno, nel primo campo dell'RDD ritornato dal metodo, ci sarà l'id dell'utente
- Il secondo, a grandi linee, ci consente di avere, per ogni utente, una lista ordinata di ProdScore associati ad esso. Più nel dettaglio, si va a richiamare il metodo iniziale, che legge l'input e genera l'RDD con String-ProdScore, che verrà preso in carico dal metodo groupByKey(), ritornando un RDD ordinato per id utente crescenti con i ProdScore ad esso associati. Questo RDD viene ulteriormente processato dal metodo mapToPair(), che banalmente si crea un *ArrayList* di ProdScore, preso come appoggio per aggiungere i vari ProdScore per ogni utente; tramite un forEach si leggono tutti i ProdScore caricati in precedenza e vengono aggiunti alla lista definita in precedenza, settando opportunamente sia l'id sia lo score del prodotto. Dopo aver definito la lista di ProdScore per ogni utente, si ordinano queste liste in base allo score decrescente (usando sempre il metodo *sort* di *Collections*), per poi ritornare una nuova *Tuple2,* avente come tipi *String* e *List<ProdScore>*, cioè l'id dell'utente e la lista di ProdScore per ognuno di essi. Il ritorno del metodo avrà quindi un RDD con tutti gli utenti (ordinati per id crescente), e per ognuno di essi una lista di ProdScore (ordinati per score decrescente).
- A questo punto non ci resta che "tagliare" la lista di ProdScore per ogni utente, prendendo solo i primi 10. Si fa allo stesso modo descritto per il job 1 con unica differenza: si prendono i top10 e non i top5.
- La stampa su file e il main saranno identici al job 1.

2.4. Esempio di output

Di seguito verrà mostrato un esempio di output, per la precisione le prime 10 righe del file più grande, generato dall'esecuzione del job Spark.

A100CY9WRC18I2 B000CQG84Y 1
A101CCC619GN4S B00017L1UK 5
A101VS17YZ5ZEJ B0004LW990 5
A103OZ75AVET1Y B000CBOR60 5
A1048CYU0OV4O8 B00004RYGX 5
A1048CYU0OV4O8 B00004CI84 5
A1048CYU0OV4O8 B00004CXX9 5
A105981PIJDJUU B000FFLHSY 4
A106E0DP6X12NW B0001ES9F8 1
A106E0DP6X12NW B0007NOWMM 1

3. Sviluppo terzo job

Il terzo job ha la finalità di raccogliere tutte le coppie di utenti con gusti affini, cioè che hanno recensito con score superiore o uguale a 4 almeno tre prodotti in comune, ordinando il tutto in base all'id del primo utente della coppia.

3.1. MapReduce

Come per il primo job, abbiamo definito due task MapReduce:

- la prima Map raccoglie come chiave l'id del prodotto e come valore l'id dell'utente con la condizione che lo score debba essere strettamente superiore a 3. In questo modo viene permesso al sistema, tramite lo Shuffle and sort, di raggruppare gli utenti che hanno dato uno score maggiore uguale a 3 ad uno specifico prodotto
- la Reduce quindi, tramite due for annidati, prende e scrive nel context tutte le possibili coppie di utenti che hanno recensito il prodotto: ogni coppia viene inserita all'interno di una stringa ed usata come chiave per la futura Map. Il valore sarà l'id del prodotto
- la seconda Map non fa assolutamente nulla
- dopo lo Shuffle and sort, che raccoglie tutti i prodotti condivisi da ogni coppia, la Reduce prende le istanze in cui la lista di prodotti è maggiore strettamente di 2.

3.2. Hive

La query, utilizzata per realizzare il job3, ha una grande novità: l'utilizzo del JOIN.

- Dopo l'importazione del file csv e della funzione di conversione, si crea una vista, che ha l'obiettivo di raccogliere, grazie ad un JOIN della tabella in input con se stessa, le coppie di utenti che hanno in comune gli stessi prodotti (il JOIN viene fatto sui productId delle due tabelle), il numero di prodotti che condividono

(COUNT(DISTINCT(*f1.productId*)), e la lista di prodotti, veri e propri, condivisi.
La clausola WHERE permette di specificare che i prodotti in comune tra la coppia di utenti devono avere uno score maggiore di 3 e la condizione f1.userId>f2.userId permette di non avere duplicati.
Infine la GROUP BY permette di raggruppare il tutto secondo la coppia di utenti da valutare.

- Si crea la tabella finale specificando che il numero di prodotti in comune tra i due utenti deve essere maggiore strettamente di 2 e ordinando il tutto in base allo userId di entrambi gli utenti.

```
CREATE TABLE IF NOT EXISTS foodReview (
      id STRING,
      productId STRING,
      userId STRING,
      profileName STRING,
      helpNum STRING,
      helpDen STRING,
      score INT,
      provisioned STRING,
      summary STRING,
      text STRING)

ROW FORMAT DELIMITED FIELDS TERMINATED BY '\t';

LOAD DATA LOCAL INPATH '/home/claudio/Downloads/amazon/2011_
2012.csv' INTO TABLE foodReview;
      add jar /home/claudio/Desktop/unix_date.jar;

CREATE TEMPORARY FUNCTION unix_date AS
'com.example.hive.udf.Unix2Date';

CREATE VIEW same_users AS
SELECT f1.userId AS user1, f2.userId AS user2, COUNT(DISTINCT
f1.productId) AS same_preferences, collect_set(f1.productId)
AS products_in_common
FROM foodReview f1
JOIN foodReview f2 ON f1.productId = f2.productId
WHERE f1.score>3 AND f2.score>3 AND f1.userId>f2.userId
GROUP BY f1.userId,f2.userId;

CREATE TABLE couple_of_users_same_preferences AS
SELECT su.user1, su.user2, su.same_preferences,
su.products_in_common
FROM same_users su
WHERE su.same_preferences>2
ORDER BY su.user1, su.user2;
```

3.3. Spark

L'approccio eseguito con Spark è praticamente l'unione degli approcci utilizzati per la MapReduce e per Hive: c'è una gestione del primo campo di un RDD simile a quella utilizzata nella Map, in cui gli userId sono stati uniti in una stringa separata dal carattere '|' (lo vediamo nel dettaglio tra poco). Una similitudine c'è anche con l'approccio di Hive, poiché anche in questo caso si esegue un JOIN (che però nello specifico è diverso: il perché verrà spiegato a breve).

- si procede, come al solito, con il metodo *loadData()*, che ha due grandi differenze con lo stesso metodo utilizzato negli altri due job:

1. viene fatta una *.filter* in base allo score: si prendono solo le istanze, in cui lo score è maggiore strettamente di 3;
2. viene ritornata dalla *.mapToPair* una *newTuple2<>* in cui il primo campo è necessariamente l'id del prodotto, mentre il secondo è l'id dell'utente (il motivo è spiegato tra poco)

- il metodo *getProductWithCoupleUsers()* prende in input l'RDD, ritornato dal metodo precedente, e fa il *.join* con lo stesso RDD: questo join è un po' differente da quello fatto con Hive, in quanto non viene specificata la clausola ON, in cui si suggerisce al JOIN il campo su cui deve agire. Nel nostro caso viene preso come chiave il primo campo dell'RDD, l'id del prodotto (ecco spiegato il motivo per cui doveva essere inserito come primo capo nell'RDD), e come valori le varie coppie appartenenti al secondo campo, facenti parte dei due RDD coinvolti nel join. Il formato di output del .join è quindi: *JavaPairRDD<String,Tuple2<String,String>>*.
  Dopo il join, si esegue una *.filter* per eliminare le coppie con stesso utente e i duplicati.
  Infine con la *.mapToPair* si forma la stringa, che rappresenta la coppia di utenti separati dal carattere '|', e si inserisce come primo campo del nuovo RDD, che avrà come secondo campo l'id del prodotto condiviso dalla coppia
- il metodo finale è *getCouples()*, che raggruppa i prodotti condivisi da una coppia con una *.groupby* (ecco spiegata la nuova stringa con il separatore, definita nel metodo precedente). In particolare la *.groupby* genera per ogni coppia di utenti l'insieme di prodotti in comune.
  Quindi si tratta a questo punto solo di effettuare una *.mapToPair*, in cui si utilizza una lista di appoggio, su cui aggiungere l'insieme di prodotti per ogni coppia, e farle restituire la coppia di utenti con la lista di prodotti condivisi annessa.
  Infine si escludono le coppie con meno di 3 prodotti in comune con una *.filter* e si ordina il tutto con una *.sortByKey()*.
- vengono definiti la solita stampa e il solito main.

3.4. Esempio di output

Di seguito verrà mostrato un esempio di output, per la precisione le prime 10 righe del secondo file più grande (2009_2010.csv), generato dall'esecuzione del job Spark.

```
A1048CYU0OV4O8  A157XTSMJH9XA4  B00004RYGX  B00004CXX9  B00004CI84
A1048CYU0OV4O8  A19JYLHD94K94D  B00004RYGX  B00004CXX9  B00004CI84
A1048CYU0OV4O8  A1BZEGSNBB7DVS  B00004RYGX  B00004CXX9  B00004CI84
A1048CYU0OV4O8  A1CAA94EOP0J2S  B00004RYGX  B00004CXX9  B00004CI84
A1048CYU0OV4O8  A1CZICCYP2M5PX  B00004RYGX  B00004CXX9  B00004CI84
A1048CYU0OV4O8  A1DU58OZJNPUHV  B00004RYGX  B00004CXX9  B00004CI84
A1048CYU0OV4O8  A1E5AVR7QJN8HF  B00004RYGX  B00004CXX9  B00004CI84
A1048CYU0OV4O8  A1FJOY14X3MUHE  B00004RYGX  B00004CXX9  B00004CI84
A1048CYU0OV4O8  A1GB1Q193DNFGR  B00004RYGX  B00004CXX9  B00004CI84
A1048CYU0OV4O8  A1HWMNSQF14MP8  B00004RYGX  B00004CXX9  B00004CI84
```

4. Conclusioni e risultati finali

In conclusione, dopo l'esecuzione dei vari job su dimensioni crescenti di input, utilizzando le varie tecnologie, si è notata ovviamente una certa differenza di prestazioni. In linea generale abbiamo notato che la tecnologia con più scarse prestazioni è Hive, probabilmente perché il sistema deve tradurre la query SQL, da noi fornita, in uno o più job MapReduce (ognuno con uno o più task). I tempi di esecuzione di Hive sono anche influenzati dal fatto che i job sono stati eseguiti su una macchina virtuale, meno efficiente della macchina reale utilizzata per le altre esecuzioni.

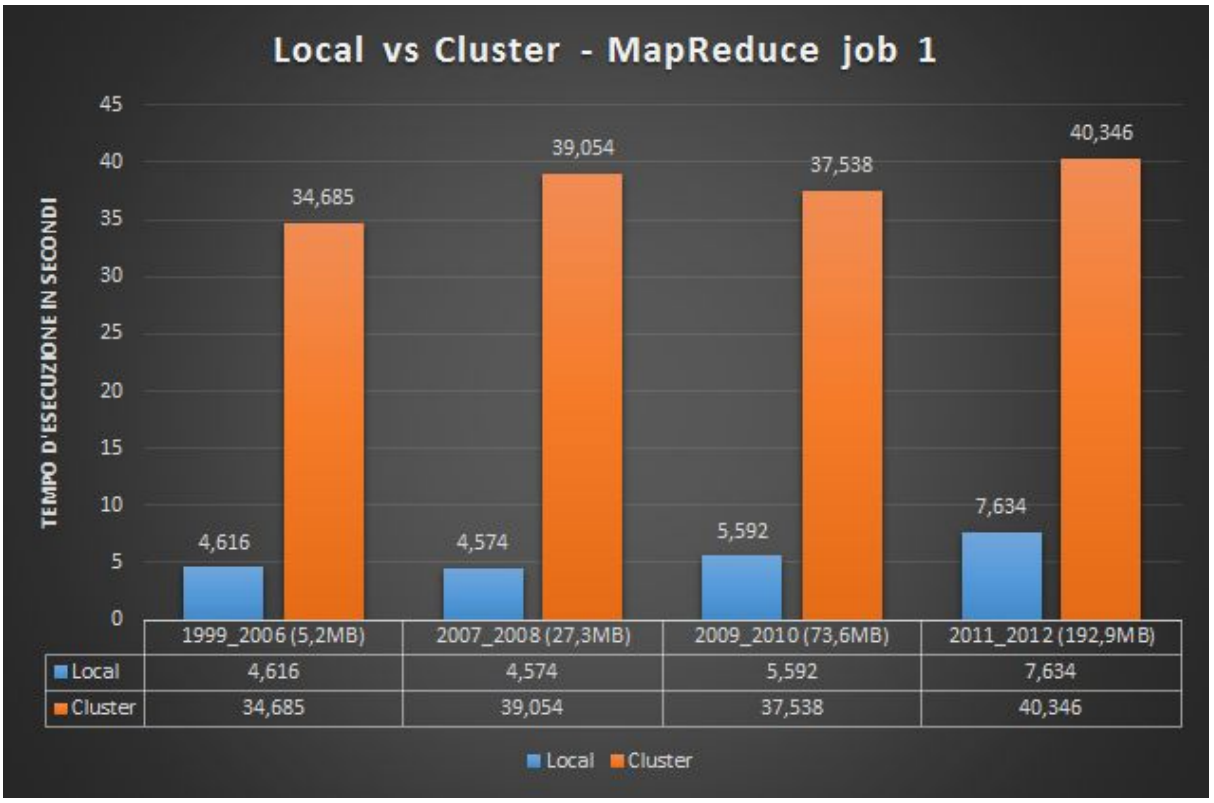
Un altro appunto da fare è la maggior efficienza di MapReduce rispetto a Spark, cosa inaspettata, dato che ci aspettavamo il contrario (tempi comunque molto simili fra loro), dovuto probabilmente all'esecuzione dei job su file relativamente piccoli (il file più grande non arriva a 200 MB).

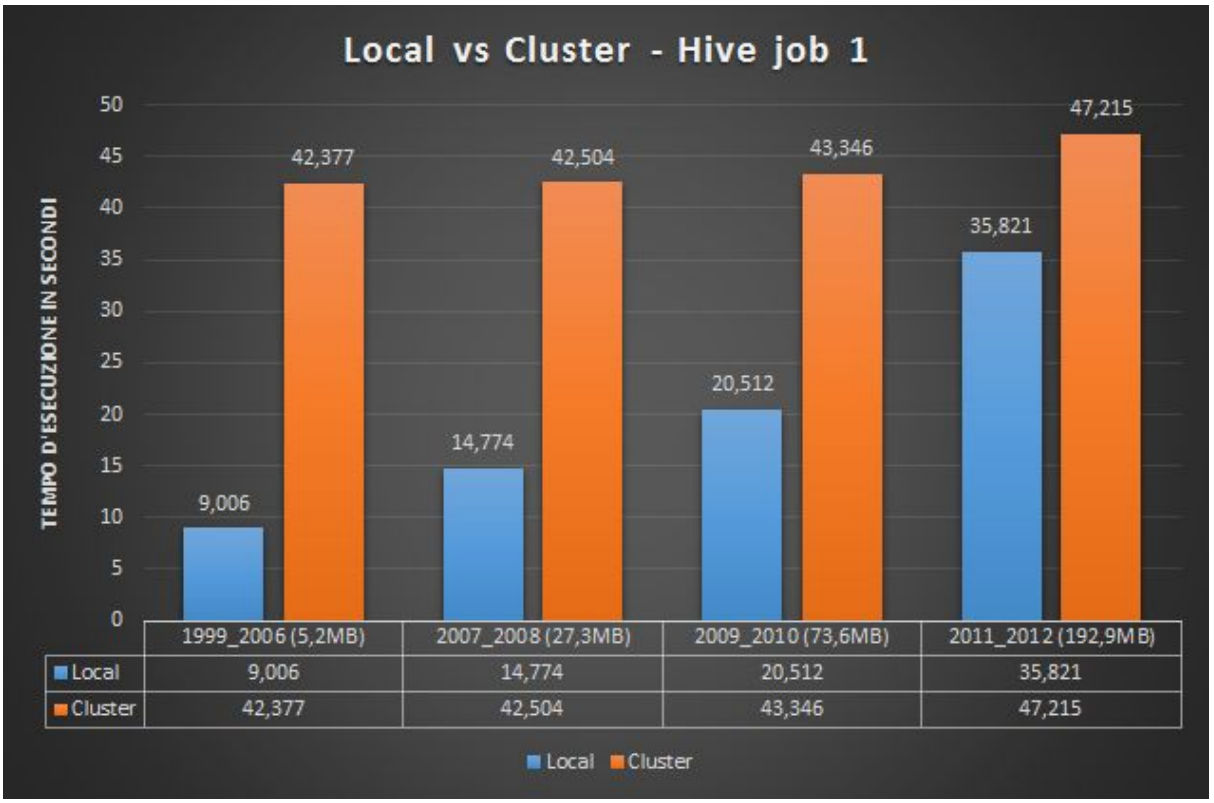
Ultimo appunto è la netta differenza dei tempi di esecuzione fra locale e cluster. Anche in questo caso la motivazione, secondo noi, è data dalla dimensione dei file non eccessiva. Eseguendo i job su file di decine di GB, sicuramente questo divario verrebbe colmato e il cluster impiegherebbe meno tempo rispetto al locale, poiché sfrutta risorse decine di volte più potenti.

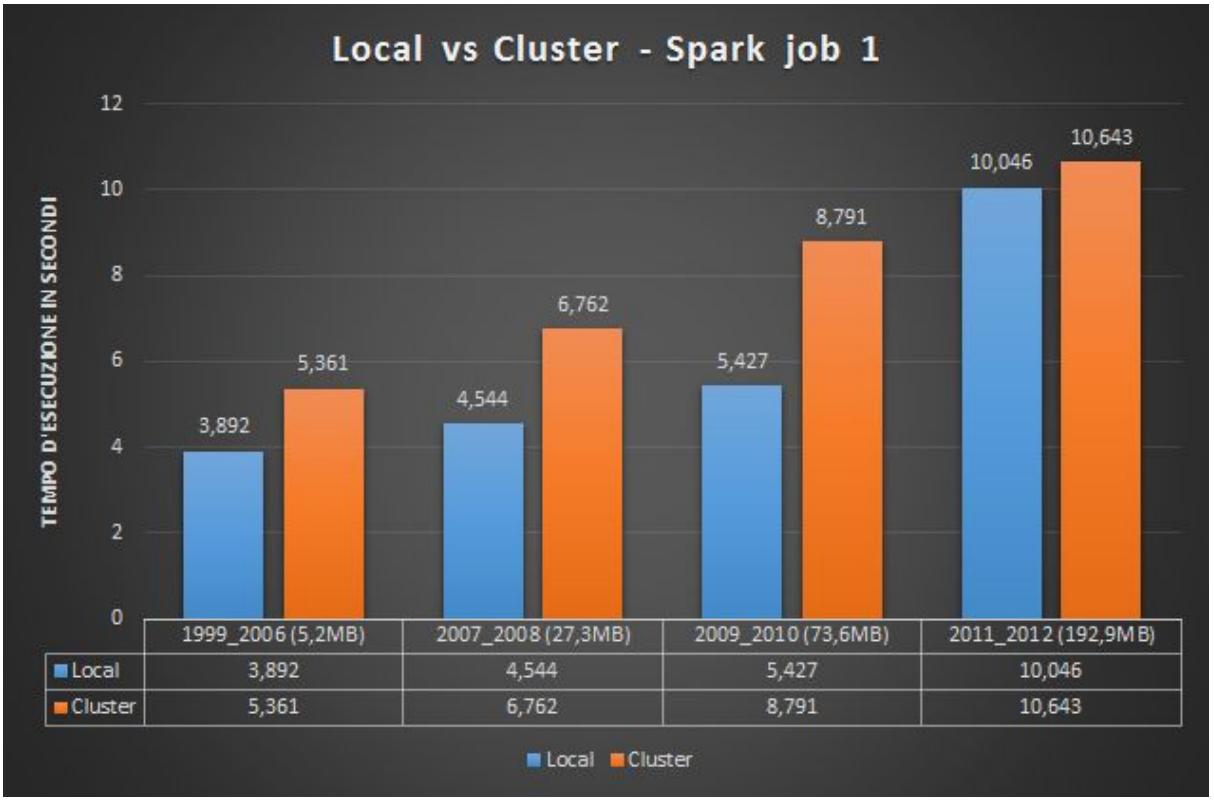
Nella consegna finale, sono stati aggiunti tutti gli output e i logs delle varie esecuzioni in locale. Per quanto riguarda il cluster invece, sono stati salvati soltanto i log del terzo job per semplicità.

A questo punto non ci resta che mostrare tabelle e grafici relative all'esecuzione dei job. Abbiamo deciso di mettere a paragone i tempi in locale e su cluster di ogni tecnologia (MapReduce, Hive e Spark) per ogni job. Ogni grafico ha sulle ascisse nome e dimensione dei 4 file di input e sulle ordinate i tempi di esecuzione dei job in secondi (comprendono sia l'esecuzione vera e propria dell'algoritmo, sia la stampa su file).

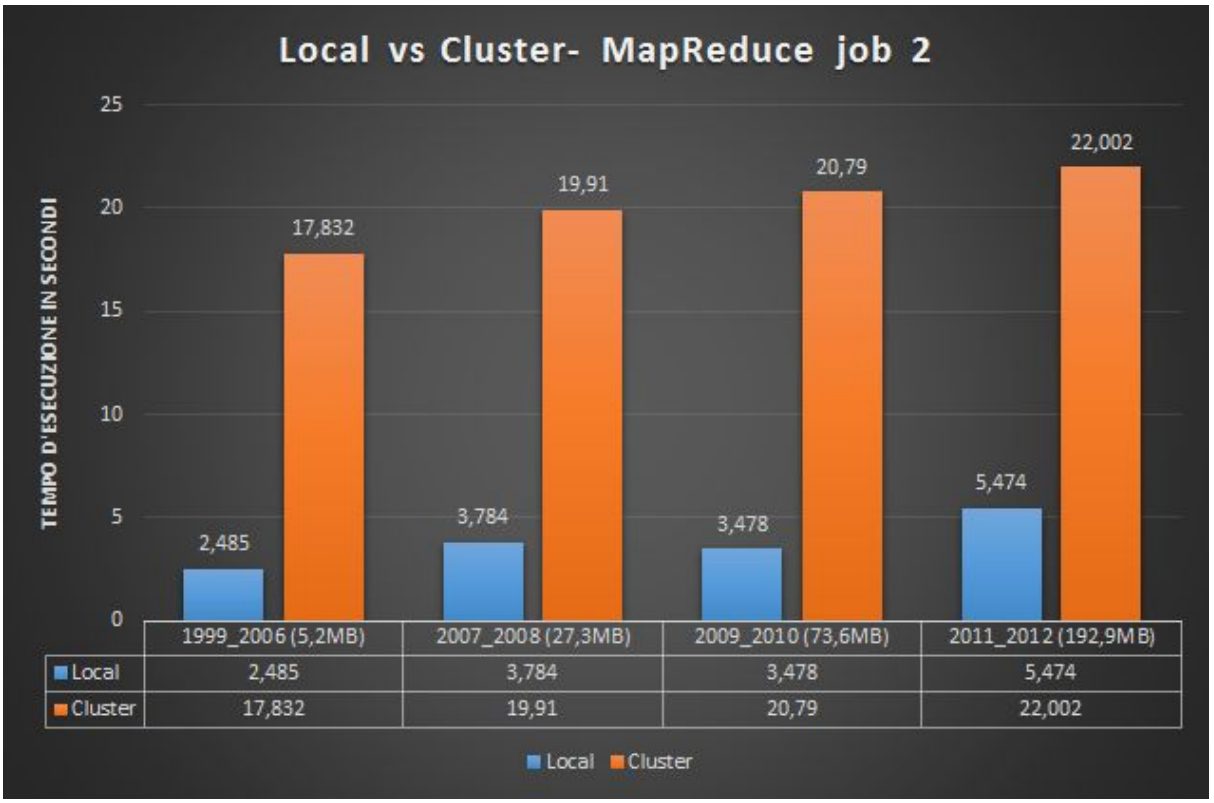
Questi sono i grafici delle tecnologie utilizzate per il job 1, confrontando fra di loro i tempi in locale e su cluster per ogni tecnologia:

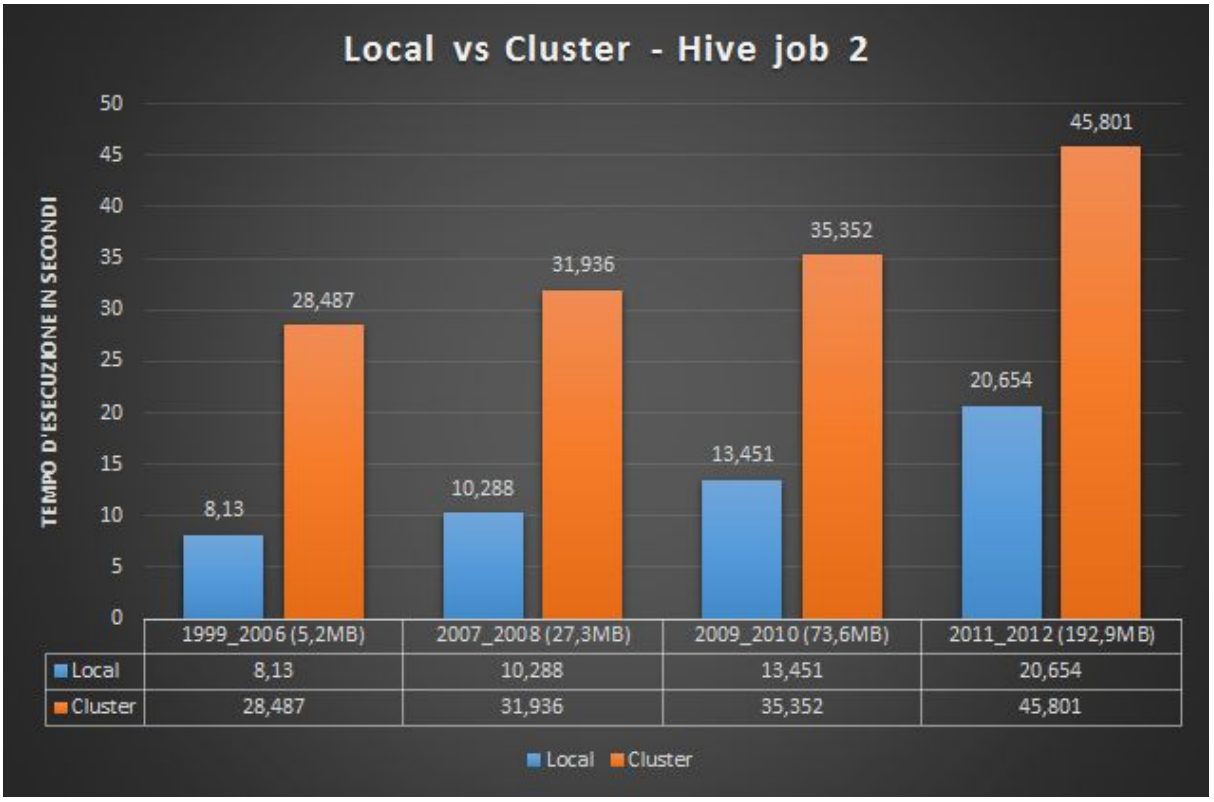


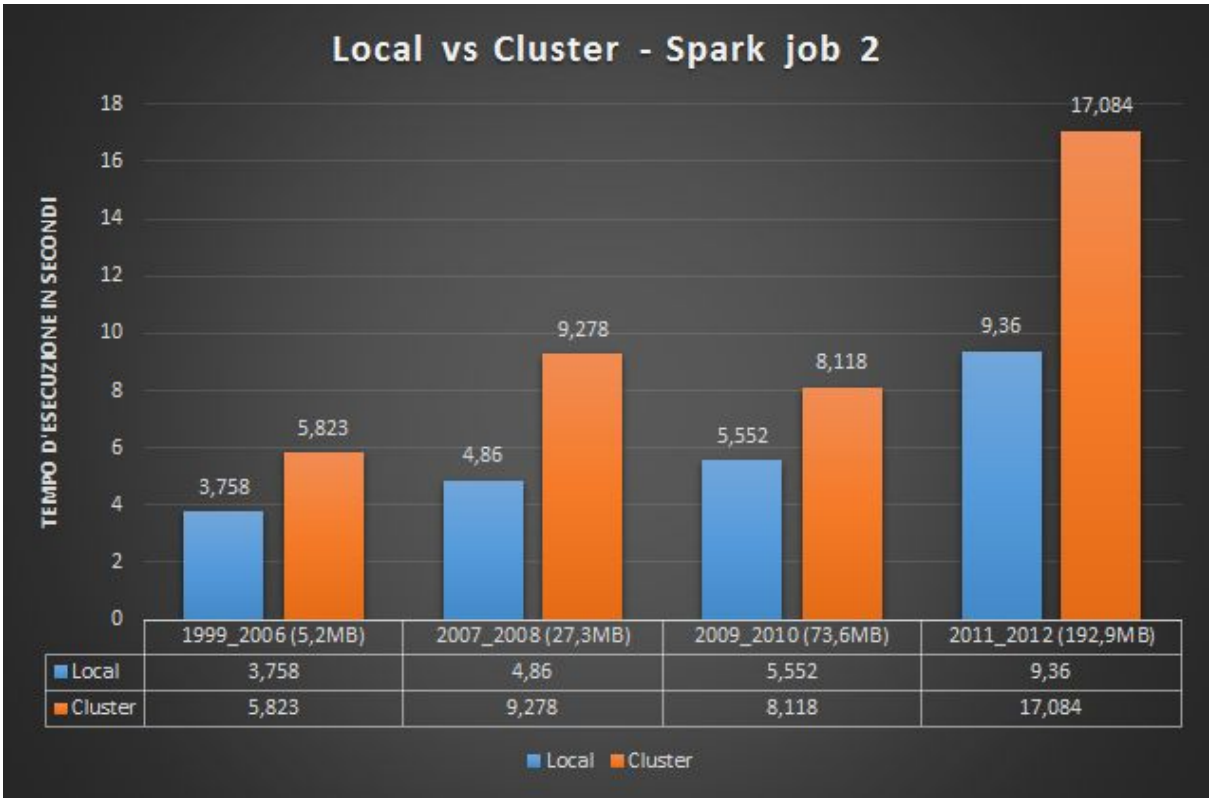




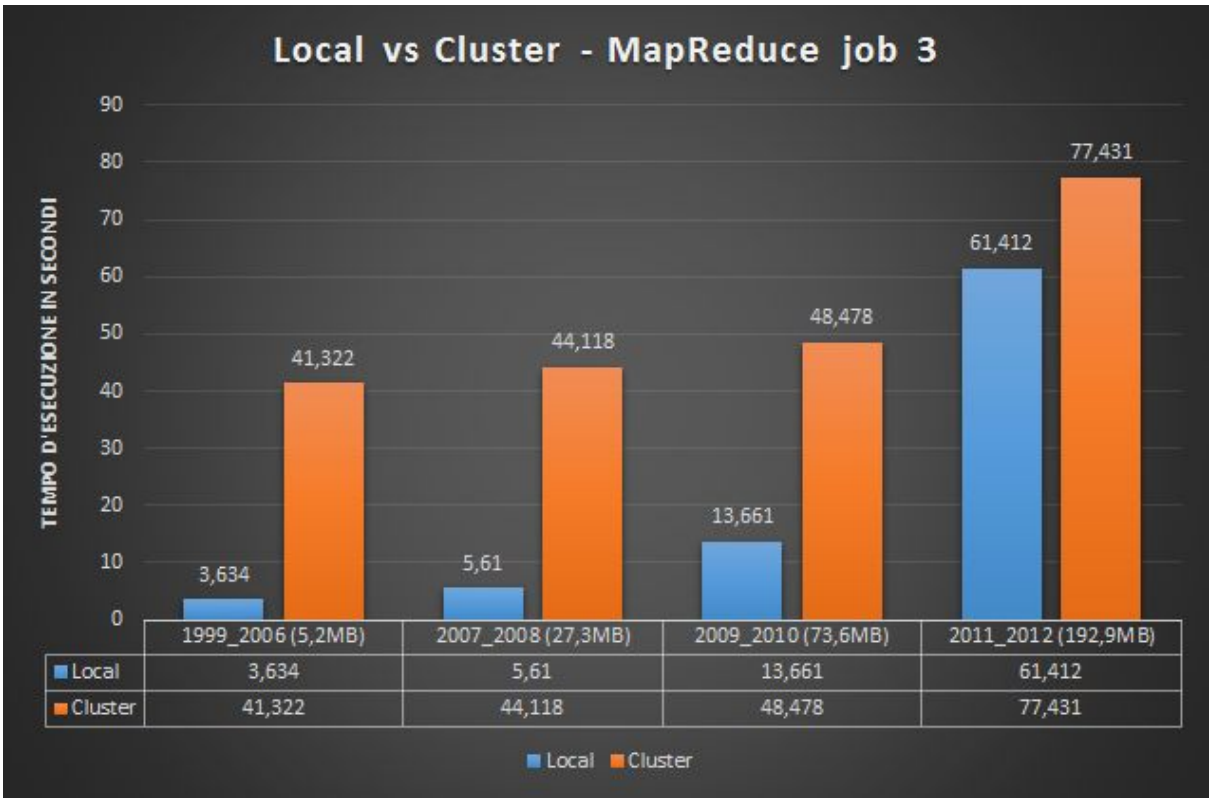
Di seguito vengono presentati i 3 grafici relativi al job2, sempre locale vs cluster per ogni tecnologia:

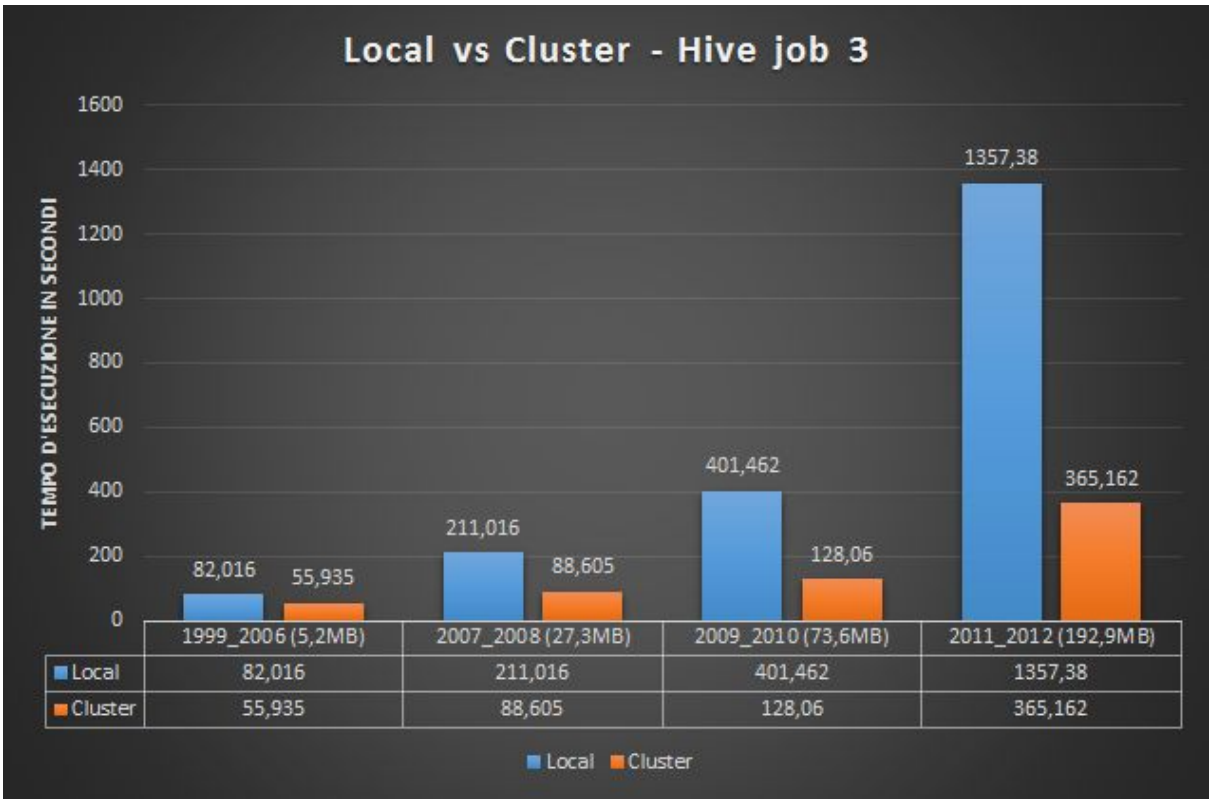


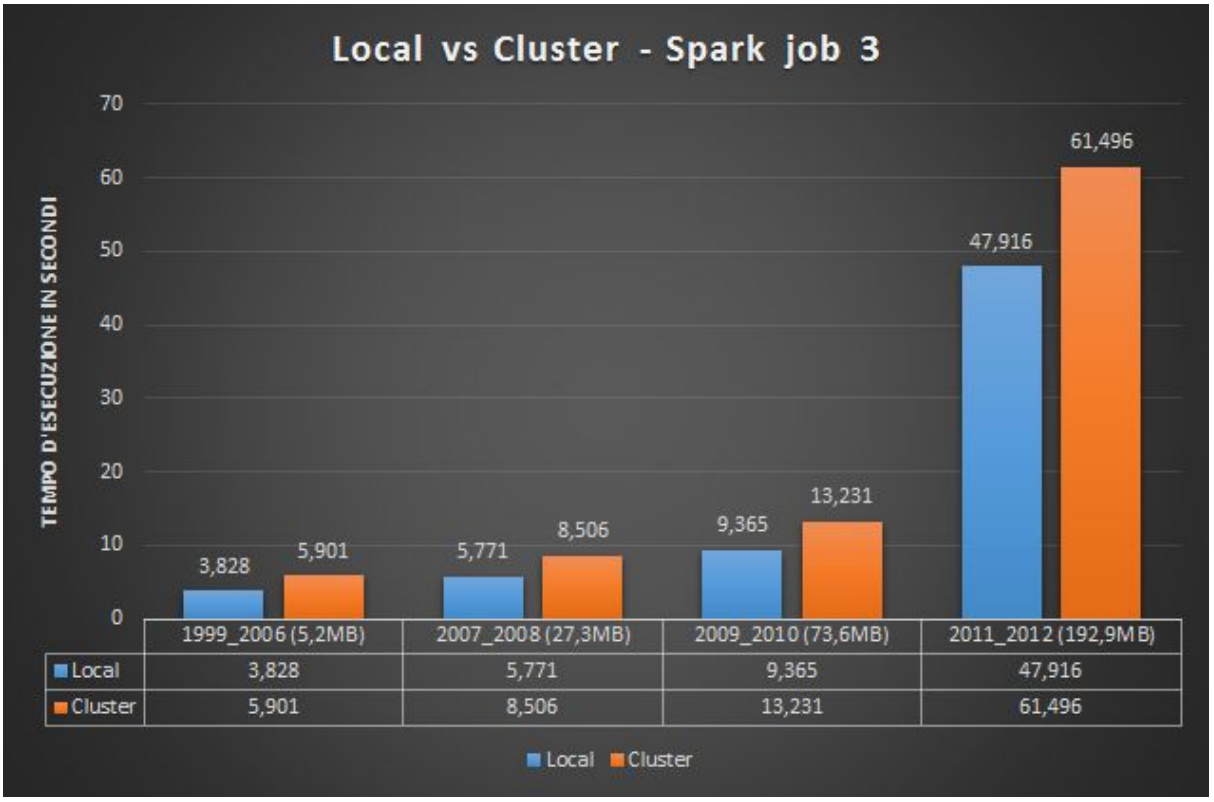

Local vs Cluster - Hive job 2
TEMPO D'ESECUZIONE IN SECONDI
50
45
40
35
30
25
20
15
10
5
0
8,13
28,487
10,288
31,936
13,451
35,352
20,654
45,801
1999_2006 (5,2MB)
2007_2008 (27,3MB)
2009_2010 (73,6MB)
2011_2012 (192,9MB)
Local 8,13 10,288 13,451 20,654
Cluster 28,487 31,936 35,352 45,801
Local
Cluster


Local vs Cluster - Spark job 2
TEMPO D'ESECUZIONE IN SECONDI
18
16
14
12
10
8
6
4
2
0
3,758
5,823
4,86
9,278
5,552
8,118
9,36
17,084
1999_2006 (5,2MB)
2007_2008 (27,3MB)
2009_2010 (73,6MB)
2011_2012 (192,9MB)
Local 3,758 4,86 5,552 9,36
Cluster 5,823 9,278 8,118 17,084
Local
Cluster

Di seguito vengono presentati i 3 grafici relativi al job3, sempre locale vs cluster per ogni tecnologia:






Local vs Cluster - Spark job 3
TEMPO D'ESECUZIONE IN SECONDI
70
60
50
40
30
20
10
0
3,828
5,901
5,771
8,506
9,365
13,231
47,916
61,496
1999_2006 (5,2MB)
2007_2008 (27,3MB)
2009_2010 (73,6MB)
2011_2012 (192,9MB)
Local
3,828
5,771
9,365
47,916
Cluster
5,901
8,506
13,231
61,496
Local
Cluster

I successivi grafici, invece, riassumono il tutto, confrontando questa volta non locale-cluster, ma le singole tecnologie all'accrescere delle dimensioni di input. In particolare abbiamo diversificato i grafici in base al job e in base all'ambiente in cui il job è stato eseguito (locale o cluster):


Quadro riassuntivo prestazioni in locale - job 2
TEMPO D'ESECUZIONE IN SECONDI
25
20
15
10
5
0
2,485
8,13
3,758
3,784
10,288
4,86
3,478
13,451
5,552
5,474
20,654
9,36
1999_2006 (5,2MB)
2007_2008 (27,3MB)
2009_2010 (73,6MB)
2011_2012 (192,9MB)
MapReduce 2,485 3,784 3,478 5,474
Hive 8,13 10,288 13,451 20,654
Spark 3,758 4,86 5,552 9,36
MapReduce Hive Spark


Quadro riassuntivo prestazioni su cluster - job 2
TEMPO D'ESECUZIONE IN SECONDI
50
45
40
35
30
25
20
15
10
5
0
17,832
28,487
5,823
19,91
31,936
9,278
20,79
35,352
8,118
22,002
45,801
17,084
1999_2006 (5,2MB)
2007_2008 (27,3MB)
2009_2010 (73,6MB)
2011_2012 (192,9MB)
MapReduce 17,832 19,91 20,79 22,002
Hive 28,487 31,936 35,352 45,801
Spark 5,823 9,278 8,118 17,084
MapReduce Hive Spark


Quadro riassuntivo prestazioni in locale - job 3
TEMPO D'ESECUZIONE IN SECONDI
1600
1400
1200
1000
800
600
400
200
0
3,634
82,016
3,828
5,61
211,016
5,771
13,661
401,462
9,365
61,412
1357,38
47,916
1999_2006 (5,2MB)
2007_2008 (27,3MB)
2009_2010 (73,6MB)
2011_2012 (192,9MB)
MapReduce 3,634 5,61 13,661 61,412
Hive 82,016 211,016 401,462 1357,38
Spark 3,828 5,771 9,365 47,916
MapReduce Hive Spark


Quadro riassuntivo prestazioni su cluster - job 3
TEMPO D'ESECUZIONE IN SECONDI
400
350
300
250
200
150
100
50
0
41,322
55,935
5,901
44,118
88,605
8,506
48,478
128,06
13,231
77,431
365,162
61,496
1999_2006 (5,2MB)
2007_2008 (27,3MB)
2009_2010 (73,6MB)
2011_2012 (192,9MB)
MapReduce 41,322 44,118 48,478 77,431
Hive 55,935 88,605 128,06 365,162
Spark 5,901 8,506 13,231 61,496
MapReduce Hive Spark