Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

1 Introduzione

Uno delle caratteristiche chiave per operare con i Big Data è la memorizzazione delle informazioni, indipendentemente da quando queste ultime sono utilizzate. Trattandosi infatti di dati di tipologie differenti ed il cui volume aumenta molto rapidamente, si può affermare che gli archivi ed i supporti utilizzati aumentino di dimensione sino a raggiungere il proprio limite. Una soluzione tradizionale, che prevede un database relazionale per lo storage di un elevata molte di dati, può risultare dunque un vero e proprio collo di bottiglia. Motivo per il quale importanti aziende, come Google e Facebook, adottano strumenti diversi dai database tradizionali per la memorizzazione dei propri dataset. Tra le tecnologie sviluppate per questo scopo vi è sicuramente **Apache Hadoop**. Hadoop è un framework open source sviluppato per realizzare in maniera intuitiva applicazioni che elaborano grandi quantità di dati in parallelo, su cluster di grandi dimensioni composti da migliaia di nodi, assicurando elevata disponibilità dei dati e tolleranza ai guasti. Se non si ha accesso ad una rete di nodi che formano un cluster, come nel caso di test o applicazioni locali, è possibile utilizzare la piattaforma **Docker**. Essa è una popolare piattaforma di container indipendenti, i quali consentono di eseguire applicazioni, integrando al loro interno ambienti di sviluppo, librerie e dipendenze necessarie per il running del software.
Nel lavoro svolto vengono in primo luogo analizzate nel dettaglio le caratteristiche sia della piattaforma Docker che del framework Apache Hadoop. In seguito sono esaminate le potenzialità offerte dai due strumenti, sviluppando ad-hoc un caso di studio relativo al programma MapReduce per la ricerca di anagrammi. L'implementazione del software **AnagramFinder** consente di individuare quelli che sono i vantaggi che è possibile ottenere adottando dall'utilizzo combinato di tool di sviluppo, opportunamente trattati nel lavoro in esame.

2 Docker

Docker è una piattaforma open-source per lo sviluppo e l'esecuzione di applicazioni. Essa consente di separare le applicazioni dall'infrastruttura sottostante, in modo da distribuire rapidamente il software prodotto. Tra i vantaggi principali di Docker vi è infatti la capacità di ridurre notevolmente il tempo tra la scrittura del codice e l'esecuzione dello stesso.

2.1 La piattaforma Docker

Docker offre la possibilità di eseguire un'applicazione in un ambiente isolato, definito *container*. L'isolamento e la sicurezza, offerti dalla piattaforma, consentono di lanciare contemporaneamente più containers su una determinata macchina host. I contenitori sono definiti leggeri, poichè non necessitano del carico aggiuntivo di un hypervisor, come accade per le macchine virtuali, ma vengono eseguiti direttamente nel kernel del computer host. Ciò si traduce nella possibilità di eseguire più containers su un determinato hardware rispetto all'utilizzo delle virtual machines. A riguardo si può anche lanciare container Docker su host che in realtà sono macchine virtuali.
Analizzando quindi le caratteristiche di Docker si può affermare che tale piattaforma semplifica il ciclo di vita dello sviluppo software, consentendo ai programmatori di lavorare in ambienti standardizzati che adottano container locali, che a loro volta forniscono applicazioni e servizi. In dettaglio risulta interessante esaminare il seguente scenario per evidenziare gli effettivi benefici che è possibile trarre dall'utilizzo di Docker durante la realizzazione del software:

1. Gli sviluppatori scrivono codice locale e condividono il loro lavoro tramite i contenitori Docker.

2. Essi utilizzano Docker per inserire le applicazioni in un ambiente di test.

3. Quando gli sviluppatori trovano un bug possono risolverlo nell'ambiente di sviluppo, ridistribuendo poi il codice nell'ambiente di test per la convalida.

Da quanto definito dallo scenario analizzato si evince che la piattaforma Docker basata sui container facilità l'elevata portabilità del software realizzato. I contenitori Docker possono essere infatti eseguiti sul laptop locale di uno sviluppatore, su macchine fisiche o virtuali in un data center o su provider

cloud. Inoltre la natura leggera di Docker semplifica anche la gestione dinamica dei carichi di lavoro, attraverso il ridimensionamento di applicazioni o servizi realizzato quasi in tempo reale, secondo le esigenze ed i contesti di sviluppo. Da questo punto di vista infatti la piattaforma si presenta veloce e leggera, risultando un'alternativa economica alle macchine virtuali basate su hypervisor. Si può dunque concludere che essa risulta particolarmente adatta per ambienti ad alta densità di lavoro e per implementazioni software di piccole/medie dimensioni, in cui è necessario operare con minori risorse disponibili.

2.2 Docker Engine

Docker Engine è un'applicazione di tipo client-server che presenta le seguenti componenti principali:

- Un **server**, il quale è un programma ad esecuzione prolungata chiamato processo demone e che corrisponde al comando *dockerd*.
- Un'**API REST**, che specifica le interfacce che i programmi possono utilizzare per interagire con il demone.
- Un **client**, basato su interfaccia a riga di comando che è identificato nel comando *docker*.

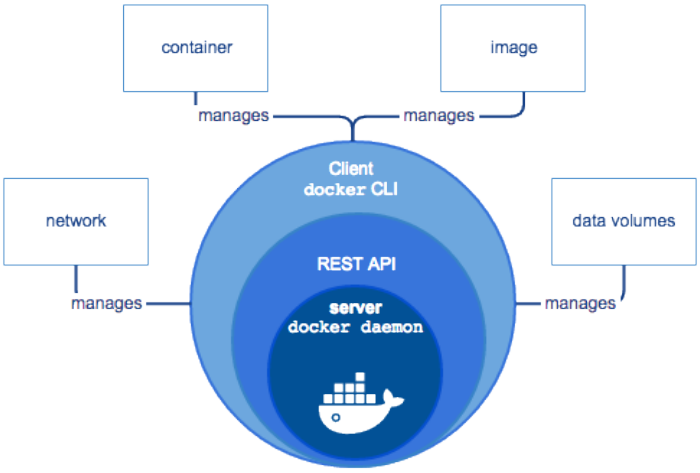


Figura 1: Docker Engine

2.3 Architettura Docker

Docker adotta un'architettura di tipo client-server. Il client interagisce con il demone Docker, che a sua volta esegue il lavoro di realizzazione, esecuzione e distribuzione dei container. Il client ed il demone possono essere eseguiti sullo stesso sistema, oppure far sì che il client sia connesso ad un demone remoto. La comunicazione, in questo caso, avviene tramite un'API REST, socket UNIX o un'interfaccia di rete.

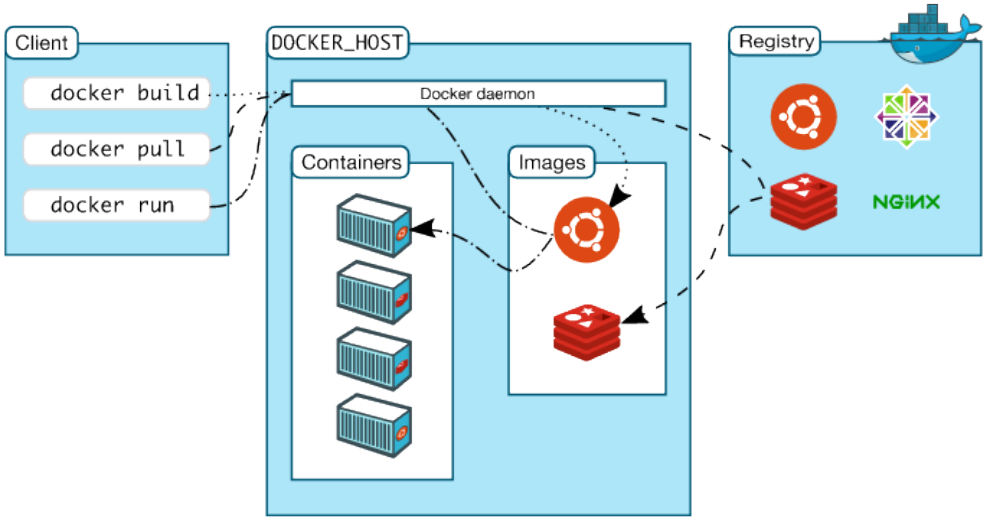


Figura 2: Archtettura Docker

Analizzando la Figura 2 possiamo individuare tre componenti fondamentali che caratterizzano l'architettura di Docker, vale a dire:

- Il **demone Docker**,ovvero *dockerd*, ascolta le richieste dell'API Docker e gestisce oggetti come immagini, contenitori, reti e volumi. Un demone può inoltre comunicare con altri demoni per gestire i servizi offerti.
- Il **client Docker** che rappresenta lo strumento attraverso il quale gli utenti interagiscono con la piattaforma. Quando si inseriscono comandi, come ad esempio *docker run*, il client li invia a *dockerd* che li esegue, dove con *dockerd* si fa riferimento al processo persistente che gestisce i container. Caratteristica fondamentale è che un client Docker può comunicare con uno o più demoni.
- I **registri Docker**, che memorizzano le immagini Docker. *Docker Hub* è un registro pubblico e Docker, una volta installato, è configurato per cercare immagini su tale registro in modo predefinito.

Inoltre, quando si lavora con Docker si creano ed utilizzano immagini, contenitori, reti, volumi, plug-in ed altri oggetti di tale piattaforma. In dettaglio possiamo individuare:

- Le **immagini**, che risultano template di tipo read-only con le istruzioni per creare i container Docker. Di solito un'immagine è basata su un'altra immagine, con alcune ulteriori personalizzazioni. Ad esempio è possibile realizzare un'immagine basata su Ubuntu, ma che a sua volta installa determinate applicazioni, oltre a presentare la configurazione dell'ambiente necessaria per far funzionare tali software. In Docker è possibile creare le proprie immagini o utilizzare quelle realizzate da altre e pubblicate in un registro. Per creare un immagine personalizzata bisogna generare e lanciare un *Dockerfile*, caratterizzato da una semplice sintassi per definire i passi necessari per ottenere l'immagine. Ogni istruzione nel *Dockerfile* determina un livello nell'immagine. Quando eventualmente si modifica il *Dockerfile* e si rigenera l'immagine, vengono ricostruiti solo quei livelli che sono stati modificati. Questa è la peculiarità che rende le immagini leggere, di ridotte dimensioni e veloci, rispetto ad altre tecnologie di virtualizzazione.

- I **contenitori**, che rappresentano istanze eseguibili di un'immagine. Si può creare, avviare, fermare, spostare, o eliminare un container attraverso l'API Docker o l'interfaccia da riga di comando. Inoltre è possibile connettere un contenitore ad una o più reti, allargare lo spazio di archiviazione disponibile o creare una nuova immagine in base al suo stato corrente. Un contenitore è definito dalla sua immagine e dalle eventuali opzioni di configurazione fornite durante la creazione o l'avvio. Quando un contenitore viene rimosso, tutte le modifiche al suo stato che non sono archiviate nella memoria permanente scompaiono.

2.4 Container Docker

Un contenitore Docker può essere considerato come un'unita standard software che impacchetta il codice e tutte le sue dipendenze, in modo che l'applicazione venga eseguita in maniera rapida ed affidabile da un ambiente di elaborazione ad un altro. L'immagine di un Docker container è un pacchetto software leggero, autonomo ed eseguibile, il quale include tutto il necessario per eseguire una determinata applicazione: codice, eseguibile, strumenti di sistema, librerie di sistema ed impostazioni di configurazione. In particolare le immagini diventano container solamente quando vengono eseguite su Docker Engine.

La tecnologia dei container Docker è disponibile sia per applicazioni basate su Linux che su Windows, il software nel container funzionerà sempre allo stesso modo, indipendentemente dall'infrastruttura. I container Docker che sono eseguiti su Docker Engine risultano:

- **Standard**: Docker infatti ha creato lo standard industriale per i containers, in modo che essi possano essere lanciati ovunque.
- **Leggeri**: i container non richiedono un sistema operativo per ogni applicazione, aumentando l'efficienza e riducendo i costi di server e licenze.
- **Sicuri**: a riguardo Docker offre le più potenti funzionalità di isolamento presenti nel settore.

Un esempio di utilizzo dei container è dato dall'esecuzione del seguente comando da shell, in quale consente di eseguire un container Ubuntu, collegarsi in modo interattivo alla sessione della riga di comando locale ed eseguire /bin/bash:

```
1 $ docker run -i -t ubuntu /bin/bash
```

Quando si esegue tale comando, si verifica quanto segue:

1. Se non si ha l'immagine *ubuntu* localmente, Docker la estrae dal registro preconfigurato.
2. Docker crea poi un nuovo container.
3. Viene in seguito allocato un file-system di tipo read-write nel container. Ciò consente, nel container in esecuzione, di creare o modificare file e directory nel file system locale.
4. Successivamente Docker crea un'interfacia di rete per connettere il contenitore alla rete predefinita. Ciò include l'assegnazione di un indirizzo IP al container. Di default i container possono connettersi a reti esterne adottando la connessione di rete della macchina host.
5. A questo punto Docker avvia il contenitore ed esegue */bin/bash*. Poiché il container è in esecuzione in modo interattivo e collegato al terminale, in base ai flag *-i* e *-t*, è possibile inserire l'input attraverso la tastiera, mentre l'output appare sul terminale.

6. Infine, quando si digita *exit* per terminare il comando */bin/bash*, il container si arresta ma non viene rimosso. Esso può essere successivamente riavviato o definitivamente eliminato.

2.5 Container e Virtual Machine

I container e le macchine virtuali hanno simili vantaggi di isolamento e allocazione delle risorse, ma operano in modo differente. I container infatti virtualizzano il sistema operativo anziché l'hardware, risultando maggiormente efficienti e portabili.
Esaminando nel dettaglio le differenze che vi si interpongo tra tali elementi possiamo affermare che, come già delineato in 2.4, i **Container** rappresentano un'astrazione a livello di app che racchiude codice e dipendenze insieme. Più contenitori possono essere eseguiti sullo stesso computer e condividere il kernel del SO con altri contenitori, ciascuno dei quali in esecuzione come processi isolati nello spazio utente. I contenitori, da questo punto di vista, occupano meno spazio delle macchine virtuali, per lo più decine di MB, possono gestire più applicazioni e richiedono meno risorse.
Invece le **Macchine Virtuali(VM)** sono un'astrazione dell'hardware fisico. Attraverso un ulteriore strato nell'architettura di virtualizzazione, detto hypervisor, è consentito eseguire più macchine virtuali su un singolo computer. Ogni VM include una copia completa di un sistema operativo, con l'aggiunta delle applicazioni e delle librerie necessarie, occupando dimensioni in termini di memoria pari a decine di GB. Ciò fa sì che esse possano risultare lente anche in fase di avvio.

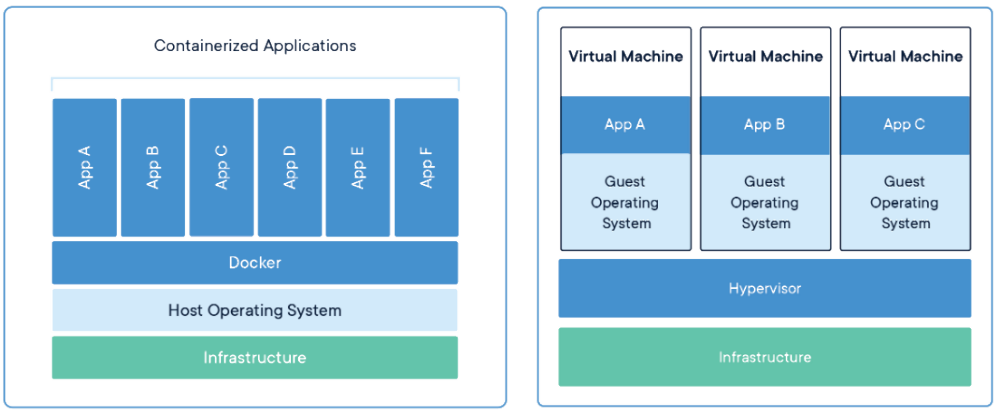


Figura 3: Container vs Virtual Machines

3 Apache Hadoop

Hadoop è un framework open-source, fornito da Apache, che consente di archiviare, elaborare ed analizzare un elevato volume di dati, con cui si opera. Scritto in Java, al giorno d'oggi risulta essere utilizzato da importanti aziende come Google, Facebook, Linkedin, Yahoo, Twitter ed altre.

3.1 Big Data

Con il termine *Big Data* si indica dati di dimensioni molto elevate, i quali non possono essere processati con tecniche di elaborazione tradizionali. In genere si lavora con dati di dimensioni quantificabili in Megabyte o al più in Gigabyte, ma quando si raggiungono size di 10^{15} byte, ovvero in termini di Petabyte, si parla appunto di Big Data. A causa dell'avvento di nuove tecnologie, dispositivi e mezzi di comunicazione come i social network, la quantità di dati prodotta dall'uomo sta crescendo rapidamente ogni anno. Per questo motivo possiamo intendere quello dei Big Data non come un singolo strumento o una semplice tecnica, ma come un argomento ampio, che comprende strumenti, tecniche e framework differenti.
I Big Data coinvolgono i dati prodotti da diversi dispositivi e applicazioni. In particolare, possiamo individuare alcune delle fonti da cui essi provengono tra le seguenti aree:

- Siti di **social network**, che generano enormi quantità di dati al giorno, essendo caratterizzati da milioni di utenti che pubblicano post ed opinioni da tutto il mondo.
- Siti di **e-commerce**, che memorizzano prodotti ed acquisti dei clienti.
- **Stazioni metereologiche**, che assieme ai satelliti forniscono enormi quantità di dati che devono essere manipolati per la previsione del tempo.
- **Aziende di telecomunicazione**, che memorizzano i dati dei propri milioni di utenti e ne analizzano le tendenze.
- **Mercato azionario**, dai quali provengono le informazioni circa le decisioni di acquisto e di vendita delle quote aziendali da parte dei clienti.
- **Motori di ricerca**, i quali sono in grado di recuperare dati da differenti database.

Le tipologie di dati con cui si opera nell'ambito dei Big Data possono essere individuate in strutturate, semi-strutturare e non strutturate. Inoltre il modello di crescita di tali dati viene inizialmente definito delle 3V, con cui si identifica:

- **Volume**, in riferimento alla quantità di dati generati.
- **Varietà**, in relazione alla differente tipologia dei dati che vengono generati, accumulati, ed utilizzati.
- **Velocità**, ad indicare la rapidità con la quale i nuovi dati sono generati.

In seguito tale modello è stato ulteriormente esteso, introducendo altre due V che rappresentano:

- **Veridicità**, ovvero assegnare un indice di affidabilità ai dati, considerando l'eterogeneità delle fonti da cui essi provengono e la velocità con cui essi sono generati.
- **Valore**, ad indicare la capacità di trasformare i dati in valore in termini di business.

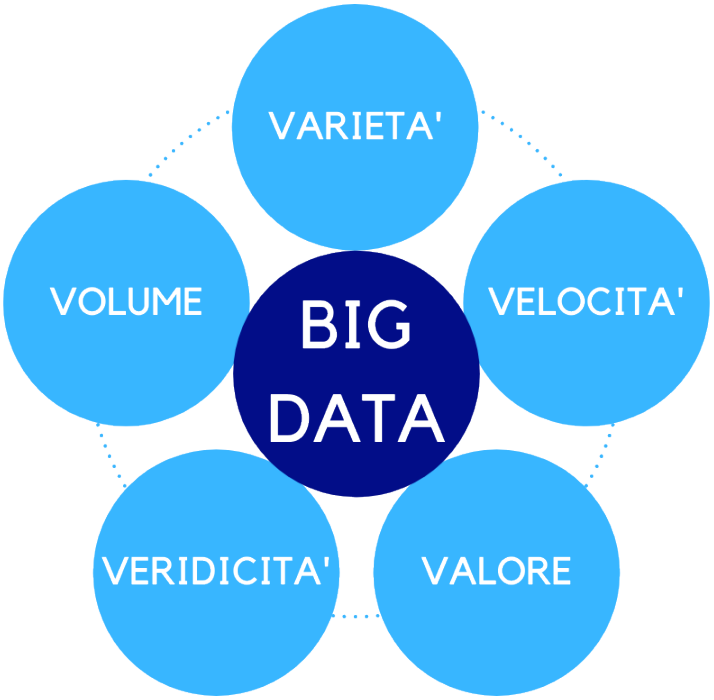


Figura 4: Big Data - Modello 5V

Le tecnologie per i Big Data risultano di particolare importanza al fine di fornire analisi più accurate, in maniera tale da avere una maggiore efficienza operativa, una riduzione dei costi ed un successivo abbassamento dei rischi per l'azienda. Per sfruttare la potenza dei big data, è necessaria però un'infrastruttura in grado di gestire ed elaborare enormi volumi di dati in tempo reale e di garantire la sicurezza dei dati stessi.
Le soluzioni per operare con i big data possono essere differenti. In particolare si può attuare un approccio di tipo **tradizionale**, in cui un'azienda presenta una singola macchina per la memorizzazione e l'elaborazione dei dati. In questo modo gli utenti interagiscono con il sistema centralizzato che si occupa di memorizzare ed elaborare la grande mole di dati aziendali.

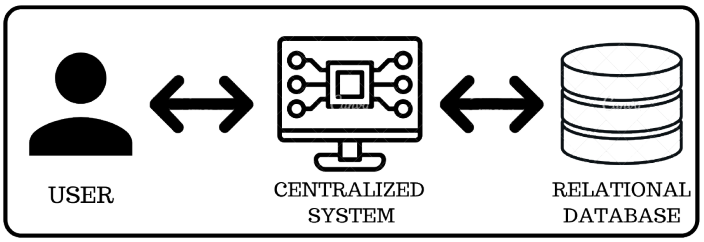


Figura 5: Big Data - Approccio Tradizionale

Questo approccio può risultare vantaggioso quando la quantità di dati da archiviare ed elaborare risulta essere di dimensione ridotta, tale da essere memorizzata in un database tradizionale. Quando invece si ha a che fare con un volume di dati elevato, esso rappresenta un vero e proprio collo di bottiglia. Per risolvere tale problematica Google ha provveduto a realizzare un algoritmo, definito **MapReduce**. Questo algoritmo divide i task in parti più piccole, assegnandole a macchine appartenenti ad un cluster. Successivamente vengono raccolti i risultati, che una volta integrati formano l'insieme dei dati risultante, ottenuto dall'elaborazione.

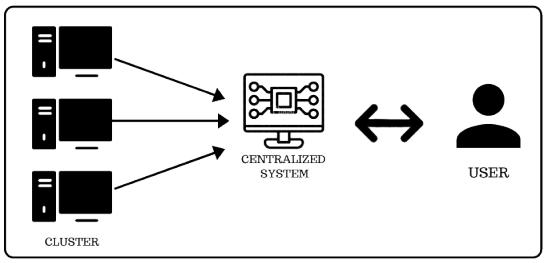


Figura 6: Big Data - MapReduce Google

Basandosi sull'algoritmo MapReduce sviluppato da Google, Doug Cutting ed il proprio team hanno realizzato un progetto open-source definito **Hadoop**. Quest'ultimo esegue le applicazioni utilizzando proprio l'algoritmo appena esposto, elaborando i dati in parallelo. Hadoop può essere dunque visto come un framework che consente di sviluppare applicazioni in grado di effettuare analisi statistiche complete su un volume di dati elevato.

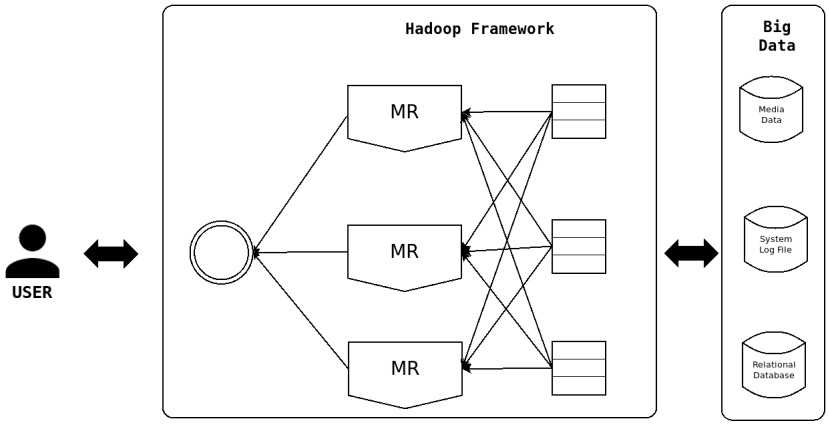


Figura 7: Big Data - Hadoop Framework

3.2 Architettura Hadoop

Le componenti principali che caratterizzano l'architettura di Hadoop risultano essere:

- **HDFS**: Hadoop Distributed File System, basato sul GFS(Google File System), fornisce un file system distribuito il quale è progettato per essere eseguito su hardware di largo consumo. Presenta come peculiarità rilevanti un'alta tolleranza ai guasti, accesso ad alta velocità ai dati delle applicazioni ed è adatto ad operare con volumi di dati elevati.

- **MapReduce**: modello di programmazione parallela per la scrittura di applicazioni distribuite. Si divide nelle fasi di *Map*, durante la quale si accettano i dati in input e convertiti in coppie $< chiave, valore >$, oltre alla fase di *Reduce*, in cui viene consumato l'output della fase precedente per restituire il risultato dell'elaborazione.

- **YARN**: Yet another Resource Negotiator è un framework utilizzato per il job scheduling e la gestione delle risorse.

- **Hadoop Common**: librerie Java adottate da altri moduli Hadoop.

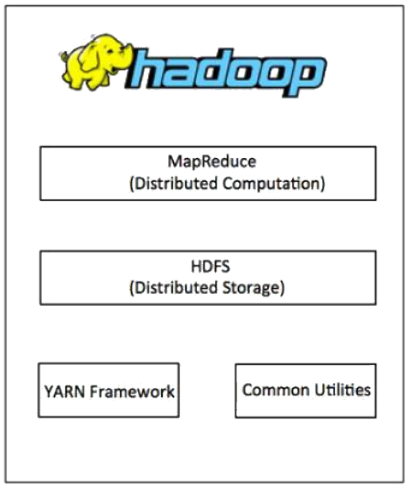


Figura 8: Architettura Hadoop

Questa tipologia di architettura, mostrata in Figura 8, è progettata per essere realizzata su un cluster di macchine a basso costo. In particolare, uno dei motivi principali per i quali è stato realizzato il framework Hadoop risulta essere legato al fatto che è molto più costoso costruire server di grandi dimensioni che gestiscono elaborazioni su larga scala, piuttosto che collegare tra loro molti computer economici come un unico sistema distribuito. Tale soluzione fa sì che le macchine possano leggere i dati in parallelo e fornire un elevato throughput.
L'idea alla base di Hadoop è dunque quella di eseguire il codice su un cluster di computer. Questo processo è caratterizzato dalle seguenti attività principali, che il framework realizza:

- I dati sono inizialmente divisi in directory e file. I file sono divisi in blocchi di dimensioni uniformi di 128M o 64M.
- I file vengono poi distribuiti su vari nodi del cluster per un'ulteriore elaborazione.
- HDFS, in cima al file system locale, supervisiona l'intera elaborazione.
- I blocchi vengono replicati per la gestione di guasti hardware.
- Verifica che il codice sia stato eseguito correttamente.
- Esegue l'ordinamento che si svolge tra le fasi Map e Reduce.
- Invia i dati ordinati ad una determinata macchina del cluster.

I vantaggi di adottare Hadoop possono essere ricondotti ai seguenti aspetti:

- **Velocità**, poiché in HDFS i dati sono distribuiti all'interno del cluster, e mappati in modo tale da facilitarne il recupero. Gli stessi strumenti per l'elaborazione delle informazioni si trovano sui server, riducendo i tempi di calcolo.
- **Scalabilità**, un cluster Hadoop può essere esteso aggiungendovi semplicemente nodi, senza interruzioni del framework stesso.
- **Convenienza**, legata al fatto che Hadoop è un framewok open-source che necessità di hardware a basso costo per archiviare ed elaborare i dati, risultando molto meno dispendioso in termini economici rispetto ai database relazionali.
- **Compatibilità**, proprietà garantita per gran parte delle piattaforme, essendo Hadoop scritto in Java.
- **Parallelismo**, attraverso cui il framework opera per l'elaborazione delle informazioni.
- **Tolleranza ai guasti**, essendo HDFS in grado di replicare i dati all'interno del cluster. Normalmente, i dati vengono replicati tre volte ma il fattore di replica è configurabile.

3.3 Cluster Hadoop

Analizzando l'architettura esposta in 3.2, possiamo affermare che Hadoop è composto da due livelli principali:

- Livello di archiviazione(HDFS)
- Livello di elaborazione(MapReduce)

Ancor più nel dettaglio un cluster Hadoop è costituito da un singolo master e più nodi slave. Il nodo master include Job Tracker, Task Tracker, NameNode e DataNode mentre il nodo slave include DataNode e TaskTracker. Per ogni nodo master è presente un singolo Job Tracker, responsabile della schedulazione dei job da inviare agli slave, del monitoraggio delle attività e della riesecuzione in caso di fallimento. Per ogni slave è presente un singolo Task Tracker, che si occupa dell'esecuzione dei task come richiesto dal master. Questa struttura è opportunamente mostrata in Figura 9.

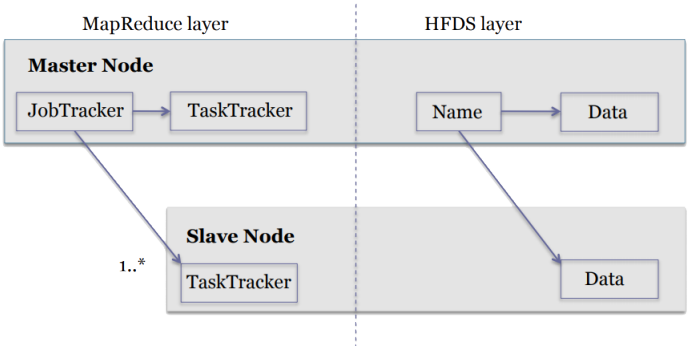


Figura 9: Livelli Hadoop

3.3.1 Livello HDFS

Il livello HDFS è basato su un'architettura di tipo master/slave. Tale architettura è costituita da un singolo *NameNode*, che svolge il ruolo di master, e più *DataNodes*, che svolgono i ruoli di slave. Sia i NameNode che i DataNode possono essere eseguiti su macchine con hardware economico. Allo stesso modo, possono essere eseguiti su qualsiasi macchina che supporta Java, il linguaggio adottato per sviluppare il software da lanciare sui NameNode e sui DataNode. In dettaglio, le componenti di tale livello sono definite come:

- **NameNode**: singolo server master esistente nel cluster HDFS. Gestisce il namespace del file system, attraverso l'esecuzione di operazioni come l'apertura, la ridenominazione e la chiusura dei file.
- **DataNode**: multipli nodi slave contenuti nel cluster HDFS. Ogni DataNode contiene più blocchi di dati, utilizzati per archiviare le informazioni. Essi infatti eseguono la creazione, l'eliminazione la replica dei blocchi, su istruzioni del NameNode. Inoltre è responsabilità dei DataNode leggere e scrivere le richiste del client nel file system.

3.3.2 Livello MapReduce

Il paradigma *MapReduce* parte quando l'applicazione client invia il job MapReduce al *Job Tracker*. In risposta, Job Tracker invia la richiesta ai *Task Tracker* appropriati. A volte, i TaskTracker restituiscono errori o timeout. In tal caso, tale parte del lavoro viene riprogrammata. Possiamo dunque definire le componenti di tale livello come:

- **Job Tracker**: Il loro compito è accettare i jobs di tipo MapReduce dal client ed elaborare i dati utilizzando il NameNode. In risposta, il NameNode fornisce metadati al Job Tracker.

- **Task Tracker**: Funziona come nodo slave per il Job Tracker. Riceve attività e codice dal Job Tracker e applica quel codice ai file. Questo processo può anche essere chiamato come *Mapper*.

3.4 Hadoop File System

Hadoop viene fornito con un file system distribuito chiamato HDFS. A differenza di altri sistemi distribuiti, HDFS è altamente tollerante ai guasti e progettato per essere eseguito su hardware a basso costo. In HDFS infatti i dati sono distribuiti su più macchine e replicati per garantirne la reperibilità in caso di guasti, oltre che un'elevata disponibilità per le applicazioni parallele. In particolare questo tipo di file system distribuito risulta essere particolarmente adatto quando si opera con:

- **File di grandi dimensioni**: ovvero il size deve essere dell'ordine dei GB o maggiore.
- **Accesso ai dati in streaming**: vale a dire quando si dà maggiore importanza al tempo di lettura dell'intero dataset, rispetto alla latenza relativa alla lettura del primo elemento. Da questo punto di vista HDFS è basato sul modello write-once e read-many-times.
- **Hardware economico**: il file system è progettato per lavorare su componenti a basso costo.

Al contrario HDFS non dovrebbe essere utilizzano quando si opera nei seguenti contesti:

- **Accesso ai dati a bassa latenza**: in quanto, in questo caso, le applicazioni richiedono accesso rapido al primo elemento di un dataset, piuttosto che alla lettura di tutti gli elementi.
- **File di piccole dimensioni**: ovvero quando si opera con un elevato numero di documenti di size ridotto. Lo svantaggio è dovuto al fatto che i NameNode memorizzano i metadata dei file e se essi risultano di piccole dimensioni ciò comporta un consumo di memoria eccessivo, per la memorizzazione di tali informazioni.
- **Scritture multiple**: vale a dire quando bisogna aggiornare diverse volte i file. HDFS infatti si basa su un modello coerente dei dati, secondo cui un file creato non può essere cambiato, ma solo aggiunto troncato.

Possiamo inoltre individuare quelli che sono gli obbiettivi principali su cui si basa Hadoop File System, vale a dire:

- **Rilevamento e recupero dei guasti**: Poiché l'HDFS comprende un gran numero di nodi, i guasti dei componenti sono frequenti. Pertanto dovrebbe disporre di meccanismi per il loro rilevamento e recupero.
- **Hardware vicino ai dati**: Un compito richiesto può essere svolto in modo efficiente, quando il calcolo avviene vicino ai dati. Soprattutto quando sono coinvolti enormi dataset, ciò riduce il traffico di rete e aumenta la velocità di trasmissione.

3.4.1 Architettura HDFS

L'architettura di Hadoop File System, mostrata in Figura 10, si basa sul modello master/slave, ed caratterizzata da blocchi, NameNode e DataNode. In dettaglio tali elementi, evidenziati in Figura 11, sono definiti come:

- Il **NameNode** è la macchina che nell'architettura distribuita funge da master. Esso è in grado di svolgere attività come: la gestione del namespace del file system, la regolazione dell'acceso da parte dei client, l'apertura, la chiusura e la ridenominazione dei file e delle directory. Analizzando le attività appena esposte possiamo definire che il NameNode può essere visto come nodo *controller* e gestore all'interno di HDFS, in quanto conosce lo stato ed i metadati di tutti i file. Le informazioni circa i metadati risultano le autorizzazioni legate ai file, i nomi e la posizione di ciascun blocco. I metadati sono di size ridotto e vengono quindi memorizzati nella memoria del NameNode, consentendo un accesso più rapido ai file.
- I **DataNode**, risultano le macchine che nell'architettura considerata rappresentano gli slave. Essi gestiscono la memorizzazione dei dati all'interno del file system, eseguendo operazioni di lettura e scrittura, secondo le richieste del client. Si occupano inoltre di operazioni come la creazione, la replicazione o l'eliminazione dei blocchi, secondo quando richiesto dal NameNode. Comunicano periodicamente con il NameNode, inviando le informazioni circa i blocchi che stanno memorizzando.
- I **blocchi**, rappresentano la quantità minima di dati che può essere letta o scritta in HDFS. La loro dimensione predefinita è di 128MB, ma tale parametro può essere configurato. Ogni blocco è memorizzato come unità indipendente.

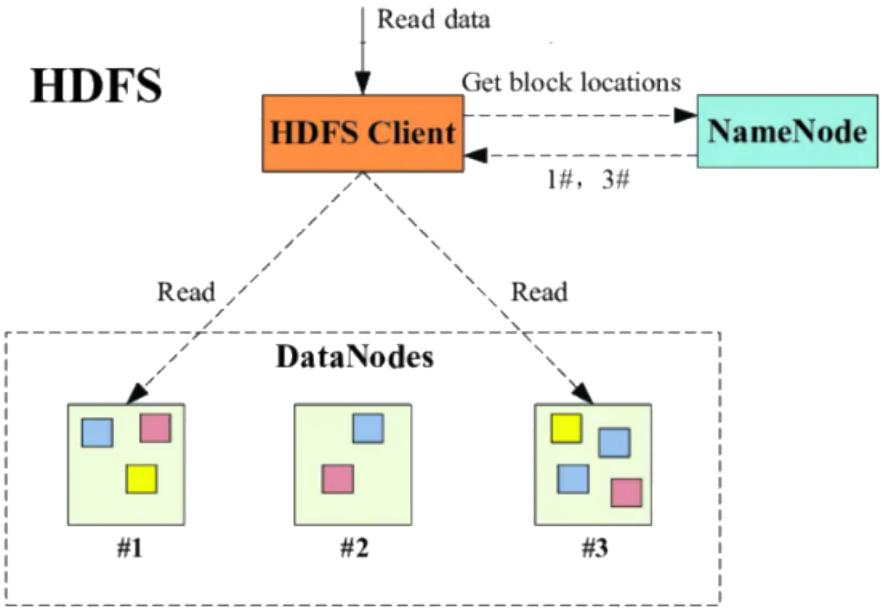


Figura 10: Architettura HDFS

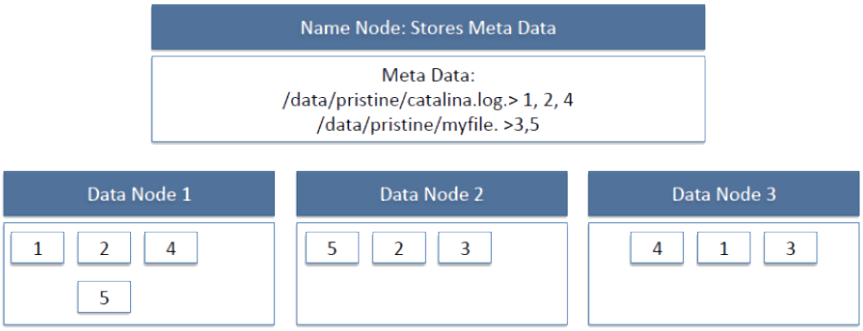


Figura 11: HDFS: Namenode e DataNode

3.5 Yet Another Resource Manager

Yet Another Resource Manager(YARN) porta la programmazione, all'interno di un cluster Hadoop, ad un livello superiore a quello di Java, consentendo l'esecuzione di applicazioni basate su tecnologie più complesse, come HBase o Spark. Differenti applicazioni YARN possono coesistere all'interno dello stesso cluster, in modo tale che sistemi basati su MapReduce, HBase, Spark o altri siano in running allo stesso tempo, apportando grandi vantaggi nell'utilizzo delle macchine. YARN è caratterizzato dalle seguenti componenti:

- **Client**: per la sottomissione dei jobs di tipo MapReduce.
- **ResourceManager**: per gestire l'uso delle risorse all'interno del cluster.
- **NodeManager**: per la gestione dei singoli nodi di elaborazione nel cluster.
- **MapReduce Application Master**: per il controllo delle attività che eseguono il processo MapReduce. Le applicazioni ed i task di MapReduce sono eseguiti in containers, a loro volta schedulati dal Resource Manager e gestiti dal Node Manager.

Jobtracker e Tasktracker, precedentemente esposti in 3.3.2, erano utilizzati nella versione precedente di Hadoop, come responsabili della gestione delle risorse e del controllo dei progressi. Tuttavia, in Hadoop 2.0 sono presenti il ResourceManager ed NodeManager per superare i limiti di Jobtracker e Tasktracker. Inoltre possiamo elencare quelli che sono i vantaggi di adottare YARN, individuati in:

- **Scalabilità**: YARN è progettato per essere eseguito su circa 10000 nodi.
- **Utilizzo**: Node Manager risulta essere in grado di gestire un pool di risorse, piuttosto che un un numero fisso di slot prestabiliti.
- **Multitenancy**: differenti versioni di MapReduce possono essere eseguite su YARN, in modo da rendere il processo di upgrade, dello stesso MapReduce, più gestibile.

3.6 MapReduce

MapReduce è una tecnica di programmazione utilizzata al fine di elaborare i dati parallelamente ed in forma distribuita. È stata progettata nel 2004, sulla base del paper "MapReduce: Simplified Data Processing on Large Clusters"[1], pubblicato da Google. Il paradigma MapReduce è composto da due fasi, vale a dire *Map* e *Reduce*. Riguardo la fase Map, l'input è fornito sotto forma di coppie $chiave - valore$ e convertito in un altro set di dati, in cui i singoli elementi sono suddivisi a loro volta in coppie $chiave - valore$. Il corrispettivo output è utilizzato dalla fase Reduce come input. Infatti quest'ultima inizia solo dopo la fase Map. L'output ottenuto dalla fase di Reduce, anch'esso in forma $chiave - valore$, risulta essere il risultato finale prodotto dal modello MapReduce. La tipologia di input ed output, che caratterizza le fasi di un'applicazione o job, è esposta in Tabella 1:

	Input	Output
Map	<k1, v1>	list (<k2, v2>)
Reduce	<k2, list(v2)>	list (<k3, v3>)

Tabella 1: MapReduce Input/Output

Durante l'esecuzione di un processo di tipo MapReduce, basato sulle fasi appena esposte, Hadoop ha il compito di inviare i task di Map e Reduce ai nodi appropriati all'interno del cluster. Il framework gestisce tutti i dettagli relativi al passaggio delle informazioni, come la verifica del completamento dei task e la copia dei dati tra i nodi del cluster. In dettaglio, gran parte dell'elaborazione avviene su quei nodi i cui dati da elaborare sono memorizzati su dischi locali, in maniera tale da ridurre il traffico di rete. Una panoramica relativa alla suddivisione dei compiti nel framework MapReduce viene mostrata in Figura12 .
Il vantaggio principale dell'utilizzo di MapReduce è legato alla facilità con cui consente di scalare il processo di elaborazione dei dati su più nodi. Infatti, sebbene la scomposizione di un processo nella fasi di Map e Reduce è talvolta non banale, una volta realizzata ciò, scalare l'applicazione su centinaia o migliaia di nodi in un cluster risulta semplicemente una modifica delle configurazioni. Dunque si può affermare che la capacità di ottenere un'elevata scalabilità dei processi su più macchine in un cluster è la peculiarità che ha attratto diversi sviluppatori ad adottare il modello.

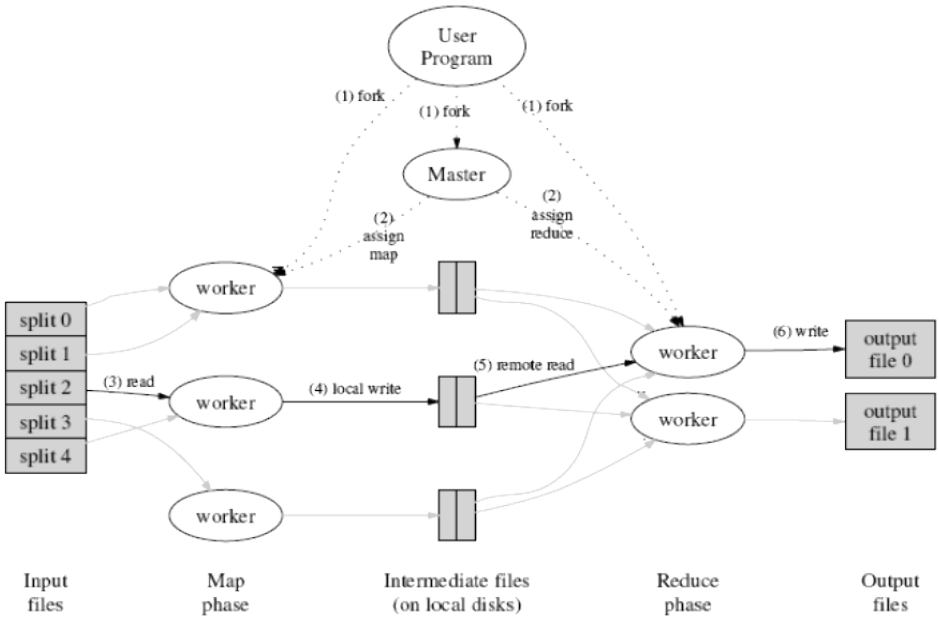


Figura 12: MapReduce Master/Worker

3.6.1 Flusso di dati

Il metodo MapReduce viene adottato per elaborare dati di elevate dimensioni. Per gestire tali quantità di informazioni in modo parallelo e distribuito, essi devono attraversare varie fasi, mostrate in Figura 13:

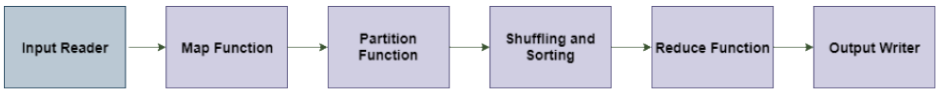


Figura 13: Flusso di dati in MapReduce

Le fasi del flusso di dati in MapReduce appena esposte, possono essere esaminate individualmente, in dettaglio:

- L'**Input Reader** legge i dati di ingresso e li divide in blocchi della dimensione appropriata, vale a dire 64 o 128 Megabyte. Ogni blocco di dati è associato ad una funzione Map. Dopo la lettura dei dati vengono generate le coppie $chiave - valore$ da utilizzare per la fase di Map, con l'utilizzo ad esempio di un *parser* o *splitter* di dati. É importante ricordare che i dati in input risiedono in HDFS e possono presentarsi in qualsiasi forma.

- La **Map Function** elabora le coppie $chiave - valore$ in ingresso e generano le corrispondenti coppie $chiave - vaore$ in uscita. Il tipo dei dati in input ed in output in questo caso potrebbe risultare differente.
- La **Partition Function** assegna l'output di ciascuna funzione Map al Reducer appropriato. Si ottiene in questo modo l'indice dei Reducer.
- Lo **Shuffling** prevede che i dati vengano mescolati tra i nodi del cluster e siano pre-elaborati per la fase di Reduce. In seguito si applica la funzione di **Sorting**, in cui i dati sono confrontati ed ordinati in base alla chiave.
- La **Reduce Function** prende in ingresso i dati intermedi ottenuti dalla fase precedente e li riduce in una soluzione di dimensione inferiore.
- L'**Output Writer** ha il compito di trascrivere nella memoria persistente il risultato ottenuto dalla funzione Reduce.

Un job di tipo MapReduce definisce la corretta esecuzione delle operazioni appena esposte. Durante l'esecuzione di tale job i dati sono suddivisi in chunk indipendenti e processati in parallelo dai task Map. L'output delle attività di Map viene dapprima combinato, poi ordinato e successivamente elaborato nella fase di Reduce per ottenere l'output finale.

4 Caso di Studio: Anagram Finder

Al fine di analizzare le caratteristiche di Docker ed Hadoop, esaminate in linea teorica, si è provveduto a realizzare un caso di studio, finalizzato alla ricerca di anagrammi. In dettaglio, l'obiettivo principale del progetto realizzato è quello di implementare un cluster Hadoop in Docker, ed eseguire il programma di testing per la ricerca degli anagrammi implementato in [5].

4.1 Analisi del problema

Un *anagramma* è una parola o frase ottenuta trasponendo le lettere di un'altra parola o frase. Ad esempio la parola "anagram" può essere riorganizzata in "nagaram". Allo stesso modo "Elvis" può essere riorganizzata in "lives", in maniera tale che "Elvis" e "lives" siano anagrammi. Inoltre possiamo definire colui che crea anagrammi come anagrammista. Altri esempi di anagrammi possono essere individuati in:

- La parola "mary" che può essere riorganizzata in "army".
- La parola "secure" che può essere riorganizzata in "rescue".
- La parola "death" che può essere riorganizzata in "hated".

Nel caso in esame, l'obiettivo del programma MapReduce realizzato è quello di trovare anagrammi a partire da un file o documento in input. In particolare esso elabora anagrammi che risultano singole parole, non frasi, e sono ignorati quei termini minori di una certa lunghezza N, dove N è un parametro da impostare da riga di comando.

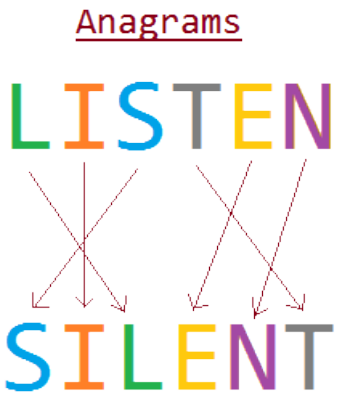


Figura 14: Esempio Anagramma

4.2 Configurazione Docker

Apache Hadoop, analizzato nel dettaglio in 3, è un popolare framework per big data particolarmente utilizzato per lo sviluppo software. Come sistema distribuito, Hadoop funziona su cluster che vanno da un singolo nodo a migliaia di nodi. Se, come nel caso in esame, si vuole testare un'applicazione Hadoop oppure non si ha accesso ad una grande rete di cluster, è possibile configurare un cluster Hadoop in locale utilizzando Docker.
Docker, esposto in 2, è una popolare piattaforma di container software indipendenti, la quale consente di creare ed eseguire applicazioni, con i loro ambienti di sviluppo, librerie e dipendenze integrati nei container. A loro volta i container sono portable, ovvero è possibile impostare lo stesso esatto sistema su un'altra macchina, eseguendo alcuni semplici comandi Docker. Attraverso Docker risulta dunque facile creare, condividere ed eseguire applicazioni dovunque, senza dover dipendere dalla configurazione attuale del sistema operativo.
Se Docker non è già installato localmente, è possibile seguire le istruzioni sulla homepage[3] ufficiale di Docker. Successivamente, per verificare le versioni di *Docker-Engine* e *Docker-Compose* installate, si adottano i seguenti comandi:

```
1 $ docker --version
2 $ docker-compose --version
```

Se è la prima volta che viene eseguito Docker, per verificare che esso funzioni correttamente, è possibile avviare un server web con il seguente comando:

```
1 docker run -d -p 80:80 --name myserver nginx
```

Dato che è la prima esecuzione di tale comando e la rispettiva immagine non è ancora disponibile offline, Docker la estrarrà dalla libreria Docker-Hub, come configurato di default. Al termine dell'esecuzione del comando, attraverso la pagina *http://localhost* è visibile la pagina iniziale del nuovo server.

4.3 Configurazione Apache Hadoop

Per installare Hadoop in un container Docker, è necessaria un'immagine Docker di Hadoop. Per generare l'immagine viene adottato, nel caso di studio, il repository Big Data Europe[4]. Dopo aver scaricato la cartella *docker-hadoop* in locale, bisogna modificare il file *docker-compose.yml*, per abilitare alcune porte in ascolto e cambiare il punto in cui Docker-compose estrae le immagini, nel caso in cui queste ultime siano già disponibili localmente. In particolare, il file *docker-compose.yml* viene impostato come segue:

Listato 1: Configurazione Docker-Compose

```
version: "2"

services:
  namenode:
    build: ./namenode
    image: bde2020/hadoop-namenode:1.1.0-hadoop2.7.1-java8
    container_name: namenode
    volumes:
      - hadoop_namenode:/hadoop/dfs/name
    environment:
      - CLUSTER_NAME=test
    env_file:
      - ./hadoop.env
    ports:
      - "9870:9870"

  resourcemanager:
    build: ./resourcemanager
    image: bde2020/hadoop-resourcemanager:1.1.0-hadoop2.7.1-java8
    container_name: resourcemanager
    depends_on:
      - namenode
      - datanode1
      - datanode2
    env_file:
      - ./hadoop.env
    ports:
      - "8089:8088"

  historyserver:
    build: ./historyserver
    image: bde2020/hadoop-historyserver:1.1.0-hadoop2.7.1-java8
    container_name: historyserver
    depends_on:
      - namenode
      - datanode1
      - datanode2
    volumes:
      - hadoop_historyserver:/hadoop/yarn/timeline
    env_file:
      - ./hadoop.env

  nodemanager1:
    build: ./nodemanager
    image: bde2020/hadoop-nodemanager:1.1.0-hadoop2.7.1-java8
    container_name: nodemanager1
    depends_on:
      - namenode
      - datanode1
      - datanode2
    env_file:
      - ./hadoop.env


  datanode1:
    build: ./datanode
    image: bde2020/hadoop-datanode:1.1.0-hadoop2.7.1-java8
    container_name: datanode1
    depends_on:
      - namenode
    volumes:
      - hadoop_datanode1:/hadoop/dfs/data
```

```
63        env_file:
64          - ./hadoop.env
65
66      datanode2:
67        build: ./datanode
68        image: bde2020/hadoop-datanode:1.1.0-hadoop2.7.1-java8
69        container_name: datanode2
70        depends_on:
71          - namenode
72        volumes:
73          - hadoop_datanode2:/hadoop/dfs/data
74        env_file:
75          - ./hadoop.env
76
77      datanode3:
78        build: ./datanode
79        image: bde2020/hadoop-datanode:1.1.0-hadoop2.7.1-java8
80        container_name: datanode3
81        depends_on:
82          - namenode
83        volumes:
84          - hadoop_datanode3:/hadoop/dfs/data
85        env_file:
86          - ./hadoop.env
87
88    volumes:
89      hadoop_namenode:
90      hadoop_datanode1:
91      hadoop_datanode2:
92      hadoop_datanode3:
93      hadoop_historyserver:
```

Per il deploy del cluster Hadoop, con l'utilizzo del file di configurazione appena esposto, deve essere lanciato da shell il seguente comando, posizionandosi nella directory che contiene il file:

```
1  $ docker-compose up -d
```

Docker-Compose è un potente tool in Docker, adottato per configurare più container contemporaneamente. Il parametro $-d$ viene utilizzato per indicare a Doker-Compose di eseguire il comando in background e restituire il prompt dei comandi, in modo tale da poter eseguire altre operazioni. In particolare, con il solo comando di cui sopra ed attraverso la definizione del rispettivo file di configurazione utilizzato ovvero *docker-compose.yml*, viene impostato un cluster Hadoop con tre slave(datanodes), un master(namenode) con il quale gestire i datanodes, uno YARN resourcemanager, un historyserver ed un nodemanager.
Docker-Compose prova inizialmente ad estrarre le immagini dalla libreria Docker-Hub, se queste ultime non sono disponibili localmente. In seguito sono create le immagini ed avviati i container. Al termine è possibile verificare la presenza di eventuali container attualmente in esecuzione attraverso il seguente comando:

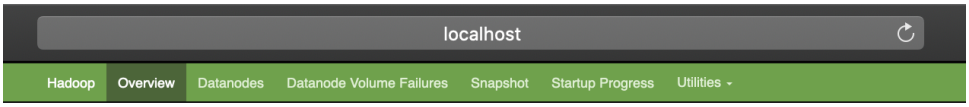
```
1  $ docker ps
```

Il comando precedente restituisce l'output mostrato in Figura 15:

```
MBPdiAntonio2:docker-hadoop-master antonioabate$ docker ps
CONTAINER ID        IMAGE                                          COMMAND                  CREATED             STATUS                            PORTS                              NAMES
86cb712a3563        bde2020/hadoop-historyserver:1.1.0-hadoop2.7.1-java8   "/entrypoint.sh /run…"   2 weeks ago         Up 5 seconds (health: starting)   8188/tcp                           historyserver
cf205bd81b67        bde2020/hadoop-nodemanager:1.1.0-hadoop2.7.1-java8     "/entrypoint.sh /run…"   2 weeks ago         Up 5 seconds (health: starting)   8042/tcp                           nodemanager1
4b7903540a26        bde2020/hadoop-resourcemanager:1.1.0-hadoop2.7.1-java8 "/entrypoint.sh /run…"   2 weeks ago         Up 5 seconds (health: starting)   0.0.0.0:8089->8088/tcp             resourcemanager
c85a782b3158        bde2020/hadoop-datanode:1.1.0-hadoop2.7.1-java8        "/entrypoint.sh /run…"   2 weeks ago         Up 7 seconds (health: starting)   9864/tcp                           datanode3
92d83c43df60        bde2020/hadoop-datanode:1.1.0-hadoop2.7.1-java8        "/entrypoint.sh /run…"   2 weeks ago         Up 7 seconds (health: starting)   9864/tcp                           datanode1
169a2130a217        bde2020/hadoop-datanode:1.1.0-hadoop2.7.1-java8        "/entrypoint.sh /run…"   2 weeks ago         Up 7 seconds (health: starting)   9864/tcp                           datanode2
e65d8a96c2f0        bde2020/hadoop-namenode:1.1.0-hadoop2.7.1-java8        "/entrypoint.sh /run…"   2 weeks ago         Up 8 seconds (health: starting)   0.0.0.0:9870->9870/tcp             namenode
```

Figura 15: Docker ps - Output

Visitando poi la pagina *http://localhost:9870*, è possibile visualizzare lo stato corrente del sistema dal namenode, come mostrato in Figura 16.



Overview 'namenode:9000' (active)

Started:	Mon Jan 06 14:38:41 +0100 2020
Version:	3.1.3, rba631c436b806728f8ec2f54ab1e289526c90579
Compiled:	Thu Sep 12 04:47:00 +0200 2019 by ztang from branch-3.1.3
Cluster ID:	CID-8467a126-23a3-4f9c-9cfe-873b15f8ff65
Block Pool ID:	BP-1891000386-172.18.0.2-1576661886770

Summary

Security is off.

Safemode is off.

142 files and directories, 94 blocks (94 replicated blocks, 0 erasure coded block groups) = 236 total filesystem object(s).

Heap Memory used 65.83 MB of 148.5 MB Heap Memory. Max Heap Memory is 444.5 MB.

Non Heap Memory used 49.14 MB of 50.19 MB Commited Non Heap Memory. Max Non Heap Memory is <unbounded>.

Configured Capacity:	175.25 GB
Configured Remote Capacity:	0 B
DFS Used:	18.49 MB (0.01%)
Non DFS Used:	7.86 GB
DFS Remaining:	158.39 GB (90.38%)
Block Pool Used:	18.49 MB (0.01%)
DataNodes usages% (Min/Median/Max/stdDev):	0.01% / 0.01% / 0.01% / 0.00%
Live Nodes	3 (Decommissioned: 0, In Maintenance: 0)
Dead Nodes	0 (Decommissioned: 0, In Maintenance: 0)
Decommissioning Nodes	0
Entering Maintenance Nodes	0
Total Datanode Volume Failures	0 (0 B)
Number of Under-Replicated Blocks	2
Number of Blocks Pending Deletion (including replicas)	0
Block Deletion Start Time	Mon Jan 06 14:38:41 +0100 2020
Last Checkpoint Time	Mon Jan 06 14:38:45 +0100 2020

Figura 16: Stato Cluster Hadoop

4.4 Implementazione

Affinché possa essere correttamente eseguito in Hadoop, il programma Anagram Finder realizzato segue il modello MapReduce, come esposto in 3.6. In dettaglio, le classi Java sviluppate per la realizzazione del software risultano essere:

- **AnagramDriver**, la classe principale che sottomette il job.
- **AnagramMapper**, la classe che implementa la funzione map().
- **AnagramReducer** la classe che implementa la funzione reduce().

4.4.1 AnagramDriver

AnagramDriver è la classe Java che sottomette il job ad Hadoop, al fine di individuare e contare gli anagrammi. Il metodo più importante della classe risulta il metodo *run()*, il quale riprende i seguenti passi definiti in pseudocodice:

Algorithm 1 AnagramDriver: run()

```
1: Creare un nuovo job con la configurazione data
2: Impostare il tipo dell'input del job
3: Impostare il tipo dell'output del job
4: Comunicare al job di utilizzare AnagramMapper come classe Mapper
5: Comunicare al job di utilizzare AnagramReducer come classe Reducer
6: Impostare il path del file di input
7: Impostare il path del file di output
8: Attendere fino al completamento del job
9: Restituire lo stato SUCCESS se tutto è andato correttamente
```

I passi descritti in precedenza sono stati implementati ed analizzati nel seguente codice Java:

Listato 2: AnagramDriver

```
1  package org.dataalgorithms.chapB05.anagram.mapreduce;
2
3  import org.apache.log4j.Logger;
4  import org.apache.hadoop.util.Tool;
5  import org.apache.hadoop.util.ToolRunner;
6  import org.apache.hadoop.conf.Configured;
7  import org.apache.hadoop.fs.Path;
8  import org.apache.hadoop.io.Text;
9  import org.apache.hadoop.mapreduce.Job;
10 import org.apache.hadoop.mapreduce.lib.output.TextOutputFormat;
11 import org.apache.hadoop.mapreduce.lib.output.FileOutputFormat;
12 import org.apache.hadoop.mapreduce.lib.input.TextInputFormat;
13 import org.apache.hadoop.mapreduce.lib.input.FileInputFormat;
14
15
```

```java
public class AnagramDriver extends Configured implements Tool {

    private static Logger theLogger = Logger.getLogger(AnagramDriver.class);

    @Override
    public int run(String[] args) throws Exception {

        //Creazione di un nuovo Job con rlativa configurazione
        Job job = new Job(getConf());
        job.setJarByClass(AnagramDriver.class);
        job.setJobName("AnagramDriver");

        //Nella configurazione del Job setta il valore della proprieta' word.count.ignored.length all'intero N
        int N = Integer.parseInt(args[0]);
        job.getConfiguration().setInt("word.count.ignored.length", N);

        //Impostare il formato dell'input come TextInputFormat, un formato semplice per i file testuali.
        //In questo modo il file di input e' diviso in linee.
        //Sia il linefeed che il ritorno a capo sono utilizati per indicare la fine della liea.
        //Viene impostato quindi <Key: posizione nel file, Value: riga di testo>
        job.setInputFormatClass(TextInputFormat.class);

        //Impostare il formato dell'output
        job.setOutputFormatClass(TextOutputFormat.class);

        //Impostare la classe chiave per i dati di output del Job
        job.setOutputKeyClass(Text.class);

        //Impostare la classe valore per i dati di output del Job
        job.setOutputValueClass(Text.class);

        //Impostare la classe Mapper
        job.setMapperClass(AnagramMapper.class);

        //Impostare la classe Reducer
        job.setReducerClass(AnagramReducer.class);

        //Impostare il path dei dati di input, args[1] = input directory
        FileInputFormat.setInputPaths(job, new Path(args[1]));

        //Impostare il path dei dati di output, args[2] = output directory
        FileOutputFormat.setOutputPath(job, new Path(args[2]));

        //Attendere il completamento del job
        boolean status = job.waitForCompletion(true);

        //Restituire lo stato ottenuto dopo il completamento del job
        theLogger.info("run(): status="+status);
        return status ? 0 : 1;
    }


    //Il metodo main per il programma anagram finder in map/reduce.
    //@throws Exception Quando vi e' un problema do comunicazione con il JobTracker.
    public static void main(String[] args) throws Exception {
        // Verificare che siano impostati esattamente 3 parametri da riga di comando
        if (args.length != 3) {
            throw new IllegalArgumentException("usage: <N> <input> <output>");
        }

        //String N = args[0];
        theLogger.info("N="+args[0]);

        //String inputDir = args[1];
        theLogger.info("inputDir="+args[1]);

        //String outputDir = args[2];
        theLogger.info("outputDir="+args[2]);

        int returnStatus = submitJob(args);
        theLogger.info("returnStatus="+returnStatus);

        System.exit(returnStatus);
    }

    //Questo metodo sottomette il job map/reduce al JobTracker.
    //@throws Exception Quando vi e' un problema do comunicazione con il JobTracker.
    public static int submitJob(String[] args) throws Exception {
        int returnStatus = ToolRunner.run(new AnagramDriver(), args);
        return returnStatus;
    }
}
```

4.4.2 AnagramMapper

La classe AnagramMapper legge un record del file di input, dove per record vi si intende un insieme di parole corrispondente ad una linea del documento. Per ogni parola del record ne ordina le lettere e restituisce all'OutputCollector di Hadoop l'output nella forma:

- Chiave: lettere ordinate della parola
- Valore: parola stessa

Possiamo sintetizzare i passi seguiti da tale classe attraverso lo pseudocodice seguente:

Algorithm 2 AnagramMapper

1: Leggere un record in input(linea di testo del fine)
2: Per ogni linea di testo: estrarre le parole che la compongono
3: Per ogni parola individuata: ordinare le lettere che la compongono
4: Per ogni parola ordinata: impostare la coppia <chiave, valore>
5: Scrivere la coppia <chiave,valore> nel contesto Hadoop

Le fasi appena mostrate sono reimplementate ed esaminate dettagliatamente nel codice Java successivo:

Listato 3: AnagramMapper

```
1  package org.dataalgorithms.chapB05.anagram.mapreduce;
2
3  import java.io.IOException;
4  import org.apache.hadoop.io.Text;
5  import org.apache.hadoop.io.LongWritable;
6  import org.apache.hadoop.mapreduce.Mapper;
7  import org.apache.hadoop.mapreduce.Mapper.Context;
8  import org.apache.commons.lang.StringUtils;
9  import java.util.Arrays;
10
11
12 //La classe AnagramMapper estende, secondo il concetto di ereditarieta', la classe Mapper.
13 //public class Mapper<KEYIN,VALUEIN,KEYOUT,VALUEOUT>
14 //Il framwork Hadoop MapReduce genera un task di tipo map per ogni split dell'input, creata
15 //attraverso l'InputFormat per il job. Le implementazioni di Mapper possono accedere
16 //alla configurazione per il Job tramite JobContext.getConfiguration().
17 //Tutti i valori intermedi associati a una determinata chiave di output vengono
18 //successivamente raggruppati dal framework e passati a un Reducer,
19 //per determinare l'output finale.
20 public class AnagramMapper
21         extends Mapper<LongWritable, Text, Text, Text> {
22
23     //Impostare i place holders per l'output key/value
24     private final Text keyAsSortedText = new Text();
25     private final Text valueAsOrginalText = new Text();
26
27     private static final int DEFAULT_IGNORED_LENGTH = 3; // default
28     private int N = DEFAULT_IGNORED_LENGTH;
29
30     // chiamato una volta all'inizio del task, imposta il valore di N.
31     @Override
32     protected void setup(Context context)
33             throws IOException, InterruptedException {
34         this.N = context.getConfiguration().getInt("word.count.ignored.length", DEFAULT_IGNORED_LENGTH);
35     }
```

```
36
37      //Chiamata una volta per ogni coppia chiave valore nella suddivisione dell'input(una volta per ogni riga)
38      //In input si ha la coppia chiave/valore oltre che context, ovvero il contesto durante il run in Hadoop.
39      @Override
40      public void map(LongWritable key, Text value, Context context)
41              throws IOException, InterruptedException {
42          //Se a riga e' nulla, ritorna.
43          if (value == null) {
44              return;
45          }
46          //Convertire la riga in String
47          String valueAsString = value.toString();
48          if (valueAsString == null) {
49              return;
50          }
51          //Convertire la riga in minuscolo e senza spazi bianchi iniziali e finali
52          String line = valueAsString.trim().toLowerCase();
53          if ((line == null) || (line.length() < this.N)) {
54              return;
55          }
56          //Estrarre le parole che compongono la riga
57          String[] words = StringUtils.split(line);
58          if (words == null) {
59              return;
60          }
61          //Per ogni parola della riga
62          for (String word : words) {
63              if (word.length() < this.N) {
64                  //Ignorare la parola se il suo size e' minore di N
65                  continue;
66              }
67              if (word.matches(".*[,.;]$")) {
68                  // Rimuovere i caratteri speciali
69                  word = word.substring(0, word.length() - 1);
70              }
71              if (word.length() < this.N) {
72                  // Ignorare la parola se il suo size e' minore di N
73                  continue;
74              }
75              //Ordinare i caratteri che compongono la singola parola
76              String sortedWord = sort(word);
77              //Impostare i caratteri ordinati della parola come chiave
78              keyAsSortedText.set(sortedWord);
79              //Impostare la parola come valore
80              valueAsOrginalText.set(word);
81              //Scrivere la coppia <chiave,valore> nel contesto
82              context.write(keyAsSortedText, valueAsOrginalText);
83          }
84      }
85      //Funzione per l'ordinamento della parola
86      static String sort(final String word) {
87          char[] chars = word.toCharArray();
88          Arrays.sort(chars);
89          String sortedWord = String.valueOf(chars);
90          return sortedWord;
91      }
92  }
```

4.4.3 AnagramReducer

La classe AnagramReducer raggruppa i valori delle chiavi ordinate in input, ovvero i valori intermedi ottenuti dalla funzione map. Dopo il raggruppamento di tali valori verifica se, per ogni chiave, vi sono due o più valori. In questo caso si è individuato un anagramma. Possiamo sintetizzare il procedimento appena descritto nel seguente pseudocodice:

Algorithm 3 AnagramReducer

1: Per ogni parola ordinata(chiave): leggere i valori associati
2: Per ogni valore associato ad una chiave: inserire il valore in un insieme
3: Se l'insieme ha cardinalità maggiore di 1: si ha un anagramma

I passi evidenziati sono implementati e commentati nel seguente codice Java:

Listato 4: AnagramReducer

```
1  package org.dataalgorithms.chapB05.anagram.mapreduce;
2
3  import java.util.Set;
4  import java.util.HashSet;
5  import java.io.IOException;
6  //
7  import org.apache.hadoop.io.Text;
8  import org.apache.hadoop.mapreduce.Reducer;
9  import org.apache.hadoop.mapreduce.Reducer.Context;
10
11
12 public class AnagramReducer
13         extends Reducer<Text, Text, Text, Text> {
14
15     // Questo metoto e' chiamato una volta per ogni chiave
16     @Override
17     public void reduce(Text key, Iterable<Text> values, Context context)
18             throws IOException, InterruptedException {
19         //Creare un nuovo HashSet
20         Set<String> set = new HashSet<>();
21         //Per ogni parola associata alla chiave in input
22         for (Text value : values) {
23             //Convertire la parola in String
24             String word = value.toString();
25             //Inserire la parola nell'HashSet
26             set.add(word);
27         }
28         //Se vi e' piu' di una parola per ogni chiave(parola ordinata)
29         //allora si ha un anagramma
30         if (set.size() > 1) {
31             context.write(key, new Text(set.toString()));
32         }
33     }
34 }
```

4.5 Esecuzione

Per testare il cluster Hadoop realizzato e l'esecuzione del programma preso in esame, bisogna innanzitutto posizionarsi con una shell all'interno del namenode attivo, con il seguente comando lanciato da terminale locale:

```
$ docker exec -it namenode bash
```

In seguito devono essere create le directory che contengono i file da utilizzare, con il comando seguente lanciato dal terminale del namenode:

```
$ mkdir -p /home/mp/data-algorithms-book/
```

Successivamente, con il comando *docker container ls* dal terminale locale, viene salvato l'ID che identifica univocamente il container del namenode. Quest'ultimo è utilizzato per la copia dei file necessari dal pc locale al container. In particolare viene copiata la cartella *src* che contiene i file Java sorgenti, la cartella *lib* che contiene le librerie necessarie all'esecuzione e la cartella *anagram* che contiene l'input. I comandi utilizzati, da terminale locale, sono i seguenti:

```
docker cp src e65d8a96c2f0:/home/mp/data-algorithms-book/src
docker cp lib e65d8a96c2f0:/home/mp/data-algorithms-book/lib
docker cp anagram e65d8a96c2f0:anagram
```

In seguito vengono compilati i file Java tramite il comando *javac*, ed il file eseguibile ottenuto è posizionato nella directory */home/mp/data-algorithms-book/dist/*. Vengono poi settate tutte le variabili d'ambiente necessarie per la corretta esecuzione del programma, con i seguenti comandi lanciati dalla shell del namenode:

```
#Cartella home dei file utilizzati
export BOOK_HOME=/home/mp/data-algorithms-book

#Path del file eseguibile
export APP_JAR=$BOOK_HOME/dist/data_algorithms_book.jar

#Path di Hadoop
export HADOOP_HOME=/opt/hadoop-3.1.3

#Path della directory input
export INPUT=anagram/input/anagram.txt


```

```
14 #Path della directory output
15 export OUTPUT=anagram/output
16
17 #Definizione classe driver
18 export
    DRIVER=org.dataalgorithms.chapB05.anagram.mapreduce.AnagramDriver
```

Infine, i file che Hadoop utilizza per l'esecuzione di AnagramFinder sono copiati in HDFS, inserendo dal terminale del namenode i comandi:

```
1  #Creazione in HDFS directory lib
2  hadoop fs -mkdir /lib
3
4  #Copia da locale del file eseguibile nella cartella lib
5  hadoop fs -copyFromLocal $APP_JAR /lib/
6
7  #Copia da locale delle librerie necessarie nella cartella lib
8  hadoop fs -copyFromLocal $BOOK_HOME/lib/*.jar /lib/
9
10 #Creazione in HDFS directory anagram
11 hadoop fs -mkdir anagram
12
13 #Copia della directory di input nella cartella anagram
14 hadoop fs -put anagram/ anagram
15
16 #Lettura del file input
17 hadoop fs -cat anagram/input/anagram.txt
```

Al termine delle operazioni preliminari appena esposte, il programma può essere correttamente lanciato in esecuzione con i seguenti comandi dalla shell del namenode:

```
1 hadoop jar $APP_JAR $DRIVER 2 $INPUT $OUTPUT
2 hadoop fs -cat anagram/output/part-r-00000
```

4.6 Test

L'input utilizzato per testare il programma è caratterizzato da un file testuale suddiviso per righe, al cui interno sono presenti parole che tra loro costituiscono anagrammi. Il file adottato è mostrato nel codice seguente:

```
1 Mary and Elvis lives in Detroit army Easter Listen
2 a silent eaters Death Hated elvis Mary easter Silent
3 Mary and Elvis are in a army Listen Silent detroit
```

Durante l'esecuzione il programma restituisce a video, come mostrato in Figura 17, l'avanzamento delle fasi map-reduce. Al termine sono mostrate le informazioni circa la realizzazione del framework MapReduce, come il numero di record letti in input o restituiti in output, oltre allo stato di uscita.

Figura 17: AnagramFinder - Esecuzione

L'output restituito dall'esecuzione del programma AnagramFinder è costituito, come si evince dalla Figura 18, dalla lista di chiavi, ovvero le lettere delle parole ordinate, cui è associato un elenco di valori, ovvero le parole individuate come anagramma.

```
root@e65d8a96c2f0:/# hadoop fs -cat anagram/output/part-r-00000
WARNING: HADOOP_PREFIX has been replaced by HADOOP_HOME. Using value of HADOOP_PREFIX.
2020-01-07 09:50:57,252 INFO sasl.SaslDataTransferClient: SASL encryption trust check: localHostTrusted = false, remoteHostTrusted = false
adeht   [death, hated]
aeerst  [eaters, easter]
amry    [army, mary]
eilnst  [silent, listen]
eilsv   [lives, elvis]
```

Figura 18: AnagramFinder - Output

Riferimenti bibliografici

[1] Dean, J., and Ghemawat, S. Mapreduce: Simplified data processing on large clusters. *Commun. ACM 51*, 1 (Jan. 2008), 107–113.

[2] Docker. Docker docs. `https://docs.docker.com/engine/docker-overview/`.

[3] Docker. Docker homepage. `https://www.docker.com/`.

[4] Europe, B. D. Docker Hadoop. `https://github.com/big-data-europe/docker-hadoop`.

[5] Parsian, M. Data Algorithms book. `https://github.com/mahmoudparsian/data-algorithms-book`.

[6] Parsian, M. *Data Algorithms: Recipes for Scaling Up with Hadoop and Spark*. O'Reilly Media, 2015.

[7] White, T. *Hadoop: The Definitive Guide*, 4th ed. O'Reilly Media, Inc., 2015.