ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA

SCUOLA DI INGEGNERIA E ARCHITETTURA

DIPARTIMENTO DI INFORMATICA – SCIENZA E INGEGNERIA

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

TESI DI LAUREA

IN

SISTEMI OPERATIVI T

ANALISI E SPERIMENTAZIONE DELLA PIATTAFORMA CLOUD DATAFLOW

CANDIDATO:

Francesco Pennella

RELATORE:

Chiar.ma Prof.ssa Anna Ciampolini

Anno Accademico 2015/2016

Sessione II

Desidero ringraziare la mia famiglia che mi ha permesso di intraprendere gli studi universitari, nonché amici, colleghi e le persone a me care per avermi supportato e aiutato in tutti questi anni. Sono grato alla Prof.ssa Anna Ciampolini per avermi concesso la possibilità di svolgere questo lavoro di tesi.

Indice

Indice delle figure

Indice delle tabelle

1 Introduzione

Negli ultimi anni molti fornitori di servizi internet hanno aumentato il loro budget d'investimento e le loro offerte relative al Cloud Computing, considerandolo come un realistico futuro dell'informatica. Tra questi anche Google da qualche anno a questa parte sta lavorando al progetto chiamato Google Cloud Platform, cioè un'infrastruttura altamente scalabile e affidabile che permette agli sviluppatori di costruire, testare e distribuire applicazioni.

I Prodotti offerti dalla Google Cloud Platform per permettere ai programmatori di sviluppare sono di vario tipo e spaziano tra diversi ambiti tra cui: Compute, Networking, Big Data, Machine Learning e Developer Tools. In questa trattazione, tuttavia, si è interessati a sperimentare le possibilità offerte nel campo dell'elaborazione di Big Data da parte di una piattaforma di Cloud Computing sviluppata da Google, chiamata Cloud Dataflow.

Questo progetto di tesi è stato reso possibile grazie all'accordo tra l'Ateneo e Injenia, azienda partner di Google in Italia, da cui è nato l'**Innovation development center**, un laboratorio virtuale, inaugurato all'inizio del 2016 nella Scuola di Ingegneria dell'Università di Bologna. Cuore dell'accordo è la messa a disposizione da parte di Injenia delle preziose applicazioni di Google che vengono usate per sviluppare software utili alle imprese. All'interno del laboratorio, ma anche da remoto grazie a un sistema di account accessibile da casa, studenti, docenti e ricercatori possono esercitarsi e studiare tutti gli utilizzi possibili delle tecnologie del colosso di Mountain View.

1.1 Struttura della tesi

Nel **Cap. 2** viene introdotto il tema del Cloud Computing in modo generale, a partire dalle caratteristiche principali e i vantaggi, per poi proseguire con i modelli di erogazione e di servizio.

Nel **Cap. 3** viene descritto il modello di programmazione MapReduce utilizzato per risolvere il problema dell'elaborazione di BigData su cluster di server fisici o virtuali, e in particolare vengono presentate due implementazioni delle piattaforme Apache Hadoop e Apache Spark.

Nel **Cap. 4** viene presentato il nuovo modello di programmazione Cloud Dataflow, recentemente introdotto da Google, prima descrivendo tutti i concetti principali, per poi concentrarsi sulle modalità di esecuzione di un servizio.

Nel **Cap. 5** vengono confrontate le prestazioni e l'utilizzo delle piattaforme Apache Hadoop, Apache Spark e Cloud Dataflow tramite l'esecuzione di un programma di WordCount, utilizzando un cluster di macchine virtuali sulla Google Cloud Platform. Infine vengono presentati i risultati ottenuti nei test, grazie ai quali è stato possibile analizzare in modo sperimentale il funzionamento del servizio.

Il **Cap. 6** contiene il codice completo dei programmi di WordCount scritti in java e utilizzati per il confronto tra le tre piattaforme in questione.

2 Cloud Computing

Con la crescita dei servizi Web negli anni 2000, molte importanti compagnie Internet, tra cui Amazon, eBay, Google e Microsoft hanno cominciato ad investire centinaia di milioni di dollari nella costruzione di grandi data center per diventare fornitori di servizi di Cloud Computing. Contemporaneamente, hanno dovuto sviluppare infrastrutture software scalabili, come *MapReduce* [4], il *Google File System* e *BigTable,* e le competenze operative per proteggere i propri data center contro potenziali attacchi.

Cloud Computing [1] è un nuovo termine che si riferisce al computing come un servizio e che recentemente sta emergendo come una realtà commerciale, rappresenta un modello per l'accesso tramite rete pool di risorse di calcolo configurabili che possono essere rapidamente fornite o rilasciate. In particolare quindi il Cloud Computing è un sistema che mette a disposizione degli utenti delle risorse sia hardware che software sotto forma di servizi in modo totalmente trasparente.

Il cloud ha la potenzialità di trasformare gran parte dell'industria legata all'*Information Technology,* rendendo i software sempre più attraenti come servizio e modellando il modo in cui l'hardware è progettato e acquistato. Gli sviluppatori con idee innovative per nuovi servizi Internet non hanno più bisogno di grandi spese hardware per sviluppare i loro servizi o per il loro funzionamento.

2.1 I vantaggi del Cloud

Per quanto riguarda i benefici che porta l'utilizzo di un modello cloud, le principali caratteristiche offerte sono:

- **On-demand self-service**: un utente può gestire le risorse computazionali in base alle proprie necessità, senza bisogno di interagire con il proprio service provider.
- **Ampio accesso alla rete**: le funzionalità del cloud sono rese disponibili ai clienti attraverso la rete ed è quindi garantito l'accesso ad una moltitudine di piattaforme differenti.

- **Pool di risorse**: le risorse di calcolo sono raggruppate per servire più consumatori in contemporanea, con i sistemi fisici e virtuali che sono allocati o riallocati dinamicamente in base alle richieste dei clienti.
- **Elasticità**: le risorse del sistema possono essere fornite o rilasciate in modo elastico per dimensionare rapidamente le risorse computazionali in base alle reali esigenze e il cliente può acquistare risorse in qualunque momento e in ogni quantità.
- **Servizio misurato**: i sistemi cloud controllano e ottimizzano automaticamente l'utilizzo delle risorse in modo appropriato a seconda del servizio. L'utilizzo delle risorse può essere monitorato e controllato garantendo trasparenza sia per il fornitore che per il consumatore del servizio utilizzato.

Dal punto di vista dell'hardware, sono presenti tre aspetti del Cloud Computing che risultano innovativi:

- L'illusione di infinite risorse di calcolo disponibili su richiesta, che permette agli utenti di non dover pianificare in anticipo le risorse di cui avrà bisogno.
- L'eliminazione di un impegno in anticipo da parte degli utilizzatori del Cloud, poiché il capitale da investire è estremamente ridotto permettendo così alle compagnie di utilizzare inizialmente piccole risorse hardware e ampliarle solo in caso di un reale incremento nei loro bisogni. Inoltre grazie all'alta efficienza delle reti cloud, spesso si ha una forte riduzione dei costi totali.
- La possibilità di pagare per l'utilizzo di risorse computazionali solo se necessario e rilasciare invece le macchine e la memoria quando cessa la loro utilità.

Gli utenti finali possono quindi accedere al servizio offerto dai service provider in ogni momento e dovunque, ed è possibile condividere e tenere i propri dati memorizzati al sicuro nell'infrastruttura.

2.1.1 Modelli di erogazione

Secondo la definizione proposta dal *National Institute of Standards and Technology* [2], i servizi cloud, in base ai principali modelli di erogazione, si dividono in:

- **Cloud pubblico**: disponibile per uso pubblico, è gestito da un provider che offre servizi cloud ed è accessibile tramite Internet. I servizi possono essere offerti gratuitamente, con delle risorse limitate, oppure a consumo.
- **Cloud privato**: disponibile per uso esclusivo da parte di una singola organizzazione, comprendente più consumatori. Può essere gestito dall'impresa stessa o da una di terze parti. È anche chiamato cloud interno o cloud aziendale. Offre servizi di hosting ad un numero limitato di utenti dietro ad un firewall. I progressi nella virtualizzazione e il calcolo distribuito hanno permesso agli amministratori della rete aziendale di diventare effettivamente fornitori di servizi che soddisfano le esigenze dei loro clienti all'interno della società. In particolare, l'obiettivo del cloud privato è la gestione efficiente e dinamica delle risorse hardware a disposizione dell'azienda che lo utilizza.
- **Cloud ibrido**: combina il cloud pubblico con il cloud privato, garantendo trasparenza all'utente, che vede un unico set di risorse uniformi. Generalmente un fornitore dispone di cloud privato e costituisce una partnership con un provider che gestisce un cloud pubblico, o viceversa. Il cloud ibrido è un ambiente di cloud computing in cui l'organizzazione fornisce e gestisce alcune risorse locali mentre altre vengono fornite esternamente.

2.1.2 Modelli di servizio

Tutti i modelli di servizio assumono la forma *XaaS* (Something as a Service) dove X rappresenta il tipo di infrastruttura che viene erogata.

I tre modelli di servizio sono:

- **Software as a Service** (*SaaS*): La capacità fornita al consumatore è quella di utilizzare le applicazioni del provider in esecuzione su un'infrastruttura cloud. Le applicazioni sono accessibili da vari dispositivi client. Il provider gestisce o controlla l'infrastruttura cloud di base, che include i sistemi operativi, la memoria, la rete, i server o anche le funzionalità delle singole applicazioni, con l'eccezione di limitate impostazioni di configurazione.
- **Platform as a Service** (*PaaS*): Il cliente può sviluppare le proprie applicazioni usando l'infrastruttura cloud, tramite linguaggi di programmazione, librerie, servizi e strumenti supportati dal provider. Il consumatore non gestisce o controlla l'infrastruttura cloud di

base tra cui rete, server, sistemi operativi o archiviazione, ma ha il controllo sulle applicazioni implementate e delle impostazioni di configurazione del sistema operativo.

- **Infrastructure as a Service** (*IaaS*): mette a disposizione del consumatore macchine virtuali, storage, reti e altre risorse hardware di calcolo con cui il consumatore è in grado di distribuire ed eseguire software. Il provider che fornisce il servizio IaaS gestisce l'infrastruttura cloud sottostante, mentre il consumatore ha il controllo sui sistemi operativi, le applicazioni e le iterazioni utente con il sistema.

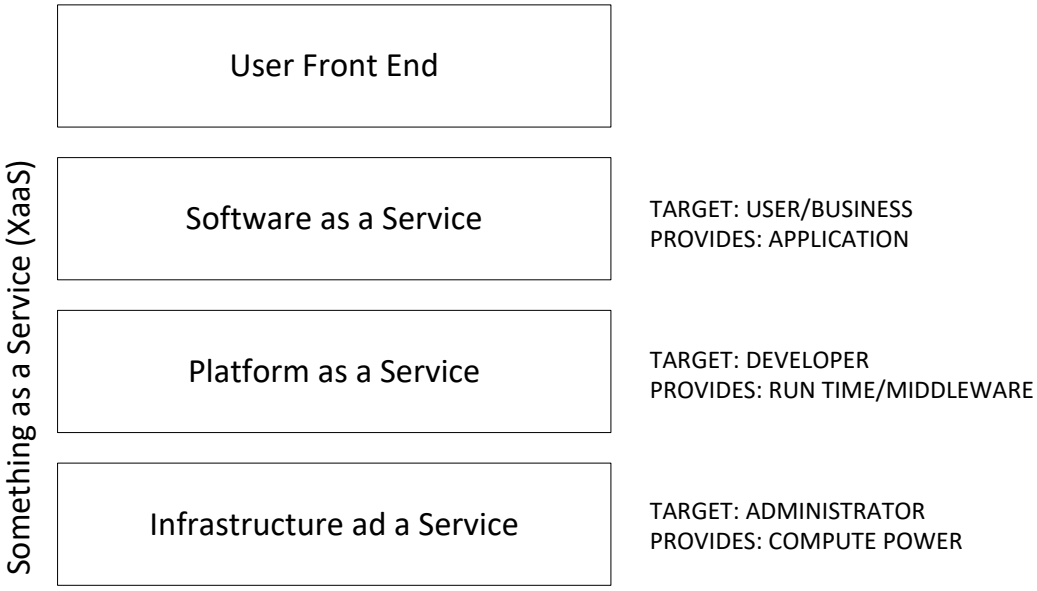

Figura 1: *Architettura di una infrastruttura cloud generica*

2.2 Nuove opportunità

Il Cloud Computing ha sicuramente introdotto nuove tipologie di applicazioni e svariati modelli d'uso, e allo stesso tempo molte classi di applicazioni già esistenti possono beneficiarne. In particolare tipi di applicazioni che rappresentano un'ottima opportunità per il Cloud Computing sono le seguenti:

- **Applicazioni mobili**: questi servizi vengono utilizzati su dispositivi mobili che spesso dispongono di scarse capacità computazionali: per questo possono beneficiare del cloud computing per riuscire a svolgere operazioni complesse o per memorizzare grandi insiemi di dati in modo completamente trasparente all'utente.
- **Elaborazioni batch in parallelo**: il Cloud Computing rappresenta un'opportunità per svolgere lavori di elaborazione batch e di analisi di grandi quantità di dati che altrimenti

richiederebbero molte ore. Uno dei grandi vantaggi del cloud è la *cost associativity*: usare centinaia di computer per un breve periodo di tempo ha lo stesso costo di utilizzare un paio di computer per molto tempo.

- **BigData Analytics**: rappresenta un caso particolare di elaborazioni batch. Mentre l'industria di database di grandi dimensioni è stata originariamente dominata dall'elaborazione delle transazioni, ora una quota crescente di risorse di calcolo è utilizzata per comprendere i clienti e le loro abitudini di acquisto per andare meglio incontro alle loro esigenze. Grazie alla varietà dei dati che è possibile analizzare, all'enorme quantità che è possibile immagazzinare con le moderne tecnologie e allo stesso tempo alla velocità che non è più un limite nell'esecuzione di grandi quantità di informazioni, si possono elaborare modelli di analisi che possono fornire predizioni e non limitarsi a semplici analisi descrittive.
- **Estensioni di applicazioni desktop ad elevata intensità di calcolo**: molte applicazioni desktop ad elevata intensità di calcolo possono beneficiare di estensioni nel cloud per svolgere elaborazioni costose.

2.3 Panoramica di alcuni servizi cloud

Ogni applicazione ha bisogno di un modello di computazione, un modello di archiviazione e, assumendo che l'applicazione sia distribuita, un modello di comunicazione. Tramite la virtualizzazione delle risorse il programmatore riesce a nascondere l'implementazione sottostante e allo stesso tempo garantire sia l'elasticità che l'illusione di capacità infinita. Differenti servizi possono essere distinti in base al livello di astrazione presentato al programmatore e al livello di gestione delle risorse.

2.3.1 Amazon EC2

Un'istanza EC2 (*Amazon Elastic Compute Cloud*) assomiglia molto di più a un hardware fisico e gli utenti posso controllare da vicino l'intera pila di software, a partire dal kernel verso l'alto. Le API sono poche e vengono utilizzate per richiedere e configurare l'hardware virtuale. Il basso livello di virtualizzazione permette agli sviluppatori di sviluppare dove vogliono. D'altra parte questo rende intrinsecamente difficile per Amazon offrire scalabilità automatica e failover, perché la semantica associata con la replicazione e altri problemi di gestione dello stato sono altamente dipendente dall'applicazione

2.3.2 Google AppEngine

Google AppEngine è una piattaforma di applicazioni domain-specific che si rivolge esclusivamente alle applicazioni web tradizionali, imponendo una struttura dell'applicazione con una chiara separazione tra un livello di computazione stateless e un livello di archiviazione stateful. Inoltre le applicazioni AppEngine dovrebbero essere basate sul modello request-reply e, come tali, sono razionate nel tempo di utilizzo della CPU per servire una particolare richiesta. I meccanismi di scaling automatico e di alta disponibilità di AppEngine e l'archivio dati proprietario MegaStore, basato su BigTable e disponibile per le applicazioni, si basano su questi vincoli.

2.3.3 Microsoft Azure

Microsoft Azure si trova in una posizione intermedia tra gli application framework completi come AppEngine e le macchine virtuali hardware come Amazon EC2. Le applicazioni di Azure sono scritte usando le librerie .NET e compilate tramite il CLR, cioè un ambiente di runtime in cui viene eseguito il codice. Il sistema supporta computazioni general-purpose piuttosto che una singola categoria di applicazioni. Gli utenti possono scegliere il linguaggio di programmazione, ma non possono controllare il sistema operativo sottostante o runtime. Le librerie forniscono un certo grado di configurazione di rete automatica e il failover/scalabilità, ma per poterlo garantire richiedono allo sviluppatore di specificare in modo dichiarativo alcune proprietà dell'applicazione.

2.4 Conclusione

Si prevede che nei prossimi anni il Cloud Computing crescerà ancora e per questo motivo gli sviluppatori dovrebbero tenerlo sempre più in considerazione. Indipendentemente dal tipo di servizio che viene erogato, ad un basso livello di astrazione come EC2 o ad un livello più alto come AppEngine, il computing, lo storage e il networking devono tutti concentrarsi sulla scalabilità delle risorse virtualizzate, piuttosto che sulla singola prestazione.

Inoltre:

- Le applicazioni software devono poter scalare rapidamente. Tale software ha anche bisogno di un modello che consenta di pagare le risorse in base all'effettivo utilizzo, in modo da soddisfare le esigenze del cloud computing.

- Il software di infrastruttura deve essere consapevole di essere in esecuzione su delle macchine virtuali.
- I sistemi hardware devono essere progettati in scala con un container, che rappresenta la dimensione minima di acquisto. Il costo di funzionamento corrisponderà alle prestazioni, mentre il costo di acquisto all'energy proportionality, impostando porzioni di memoria inattiva, disco e rete in modalità a basso consumo.

3 MapReduce

MapReduce è un modello di programmazione definito per elaborare grandi quantità di dati. I programmi scritti secondo questo modello funzionale, possono essere automaticamente eseguiti in parallelo in sistemi distribuiti costituiti ad esempio da grandi cluster di macchine.

L'utente specifica una funzione *map* che processa una coppia chiave/valore per generare un insieme di coppie chiave/valore intermedie, e una funzione *reduce* che unisce tutti i valori intermedi raggruppandoli per la stessa chiave.

Il sistema run-time si occupa del partizionamento dei dati in ingresso, dello scheduling dell'esecuzione dei programmi su un insieme di macchine, del trattamento dei fallimenti e della gestione delle comunicazioni tra le varie macchine. Questo permette anche a programmatori senza alcuna esperienza con sistemi paralleli e distribuiti, di utilizzare in modo semplice le risorse di un sistema distribuito. MapReduce è altamente scalabile e una tipica computazione processa molti terabytes di dati su alcune migliaia di macchine all'interno di un cluster.

3.1 Introduzione al modello MapReduce

La maggior parte delle computazioni che processano grandi quantità di dati è concettualmente semplice, anche se i dati in ingresso sono spesso in grande quantità e i calcoli devono essere distribuiti su centinaia o migliaia di macchine per riuscire a portare a termine l'esecuzione in un tempo ragionevole.

Il problema di come mettere in parallelo i calcoli, distribuire i dati e trattare i guasti, rischiava di oscurare la reale semplicità dei calcoli con una grande quantità di codice complesso per risolvere questi problemi. Come reazione a tale complessità, Google ha progettato una nuova astrazione che permette di esprimere i semplici calcoli che si voleva eseguire, ma nascondendo in una libreria tutto il disordine legato ai dettagli del parallelismo, della tolleranza ai guasti, della distribuzione dei dati e dell'aumento del carico. Tale astrazione è ispirata alle primitive *map* e *reduce* presenti in molti linguaggi funzionali.

I maggiori contributi di questo lavoro sono una semplice e potente interfaccia che consente un'automatica parallelizzazione e distribuzione di calcoli in larga scala, combinata con un'implementazione dell'interfaccia che raggiunge alte performance su grandi cluster.

3.2 Modello di programmazione

Ogni computazione riceve un insieme di coppie chiave/valore in ingresso, e produce in uscita un insieme di coppie chiave/valore. L'utilizzatore della libreria MapReduce esprime i calcoli come due funzioni: *Map* e *Reduce*.

La funzione *Map*, scritta dall'utente, prende in input una coppia e produce un insieme di coppie intermedie chiave/valore. La libreria MapReduce raggruppa insieme tutti i valori intermedi associati dalla stessa chiave intermedia e li passa alla funzione Reduce.

La funzione *Reduce*, scritta dall'utente, riceve i dati inviati dalla funzione Map e successivamente unisce insieme questi valori per formarne un insieme minimo possibile. Tipicamente viene prodotto al massimo un valore di uscita per ogni invocazione del metodo Reduce. I valori intermedi vengono forniti alla funzione tramite un iteratore; che ci permette di gestire elenchi di valori che sono troppo grandi per adattarsi alla memoria.

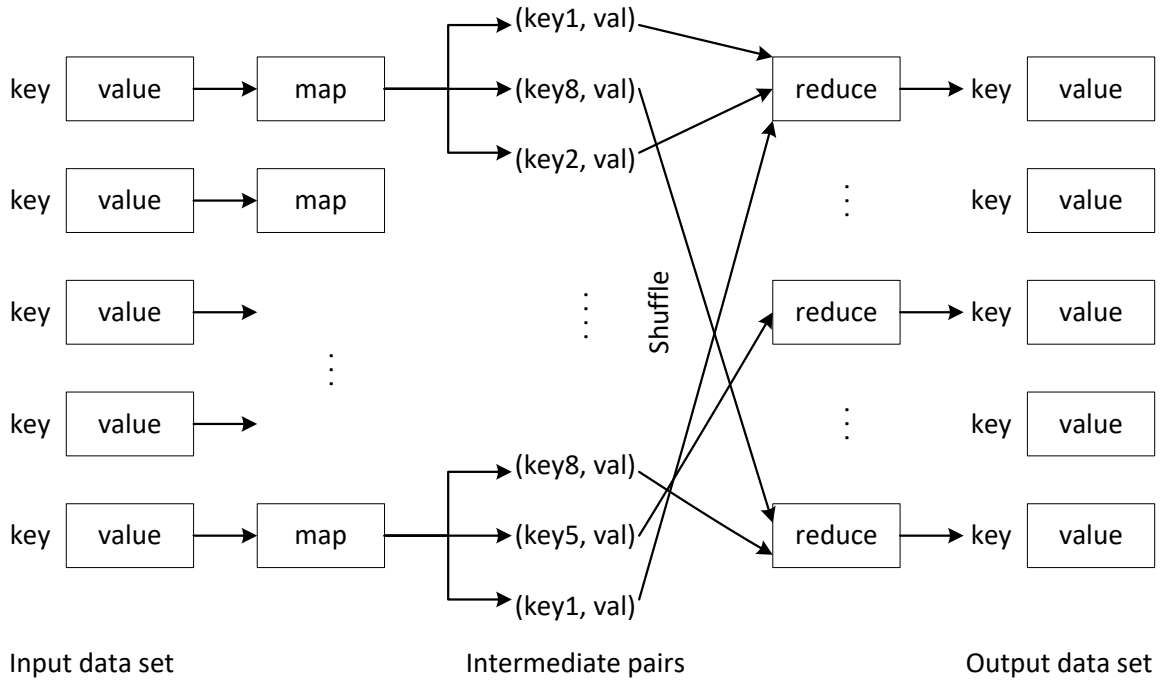

Figura 2: *Rappresentazione del modello di programmazione MapReduce*

3.2.1 WordCount

Un esempio di pseudo-codice per contare le occorrenze di ogni singola parola all'interno di una collezione di documenti:

```
// key: document name. Value: document contents
map (String key, String value) {
      for each word w in value:
      EmitIntermediate (w, "1")
}

// key: a word. Values: a list of counts
reduce (String key, Iterator values) {
      int result = 0;
      for each v in values:
            result += ParseInt(v);
            Emit(AsString(result));
      }
}
```

La funzione map genera le coppie chiave/valore intermedie, emettendo ogni parola più un numero associato di occorrenze. La funzione reduce somma insieme tutti i conteggi delle occorrenze per una specifica parola.

3.3 Implementazione

MapReduce è stato pensato per permettere a qualsiasi utente di utilizzare un sistema distribuito di larga scala, senza preoccuparsi di tutti i problemi pratici che comporta gestire un simile sistema. Infatti, il framework si occupa automaticamente dei dettagli riguardanti il partizionamento dei dati, lo scheduling dei thread sulle macchine, il controllo dei fallimenti dei nodi e la gestione della comunicazione tra le macchine e tra i nodi.

Sono possibili diverse implementazioni del modello MapReduce, ma la scelta più adatta dipende spesso dall'ambiente di esecuzione. Per esempio, un'implementazione può essere adatta ad una piccola macchina dotata di memoria condivisa, e un'altra invece per una grande collezione di macchine in rete.

Esistono diversi implementazioni per eseguire un programma MapReduce su architettura distribuita e permettono a uno sviluppatore di utilizzare facilmente le risorse di un data center per elaborare grandi moli di dati. Tra tutte le varie implementazioni di MapReduce, andremo ad analizzare in particolare le piattaforme **Apache Hadoop** [5] e **Apache Spark** [6], che

utilizzeremo più avanti anche per effettuare il confronto con la piattaforma Google Cloud Dataflow nel **Cap. 5**.

3.4 Apache Hadoop

Hadoop è un framework software open-source che consente di eseguire facilmente applicazioni che elaborano grandi quantità di dati in parallelo, su cluster di grandi dimensioni in modo affidabile e fault-tolerant. Attualmente è disponibile la versione 2 di Hadoop la quale, rispetto alla precedente, consente ad applicazioni strutturate anche in modo diverso da MapReduce di sfruttare le potenzialità del framework; inoltre rappresenta uno dei progetti top-level di Apache, usato e sviluppato da una grande community di contributori e utenti.

Il nucleo di Hadoop è costituito da due parti fondamentali: una parte di memorizzazione, nota come Hadoop File System (HDFS) e una parte di elaborazione dati chiamata MapReduce. Questi due componenti sono fatti per lavorare in coppia, infatti non si può eseguire MapReduce senza l'HDFS. Hadoop quindi divide i file in grandi blocchi e li distribuisce tra i nodi di un cluster, dove vengono processati in parallelo. Questo approccio guadagna vantaggio dalla località dei dati, poiché ogni nodo elabora solo i dati a cui ha effettivo accesso, e garantisce che siano processati in modo più efficiente e veloce che in una architettura convenzionale che si basa su un file system parallelo in cui i dati vengono distribuiti tramite la rete, con costi e tempi di esecuzione più elevati.

Il framework mette a disposizione numerosi moduli:

- **Hadoop Common**: fornisce librerie e funzionalità utilizzate da tutti gli altri moduli.
- **Hadoop MapReduce**: un modello di programmazione per l'elaborazione di BigData, altamente parallelizzabile e ispirato a Google MapReduce.
- **Hadoop Distributed File System** (HDFS): un file system distribuito che memorizza I dati su commodity hardware e risulta efficiente nel caso in cui ci sia bisogno di memorizzare una quantità di dati che deve essere partizionata su più macchine. HDFS si ispira al funzionamento di Google File System.
- **Hadoop YARN**: una piattaforma di gestione delle risorse che gestisce e assegna le risorse computazionali alle varie applicazioni.

Il sistema Hadoop è altamente affidabile, in quanto può funzionare su commodity hardware, è altamente scalabile in quanto si possono aggiungere o rimuovere nodi al cluster su necessità ed

è anche fault-tolerant, poiché è stato progettato per continuare a funzionare anche in caso di problemi su uno o più nodi.

3.4.1 Hadoop File System

In un cluster HDFS viene utilizzata un'architettura master/slave, per cui non tutti i nodi sono intercambiabili, infatti è presente un solo **NameNode** (nodo master), che gestisce il namespace del filesystem e regola l'accesso ai file, i quali sono memorizzati separatamente nei **DataNode** (nodo worker), solitamente uno per ogni nodo del cluster.

Ogni file viene partizionato in uno o più blocchi di dimensione compresa tra 64MB e 128MB, che sono memorizzati in un insieme di DataNode. Ogni blocco di file viene replicato tra più nodi del cluster per garantire anche tolleranza ai guasti e disponibilità. I DataNode eseguono quindi le operazioni di creazione, cancellazione e replicazione dei dati su richiesta del nodo master e sono responsabili di servire le richieste di lettura e scrittura da parte dei client del filesystem.

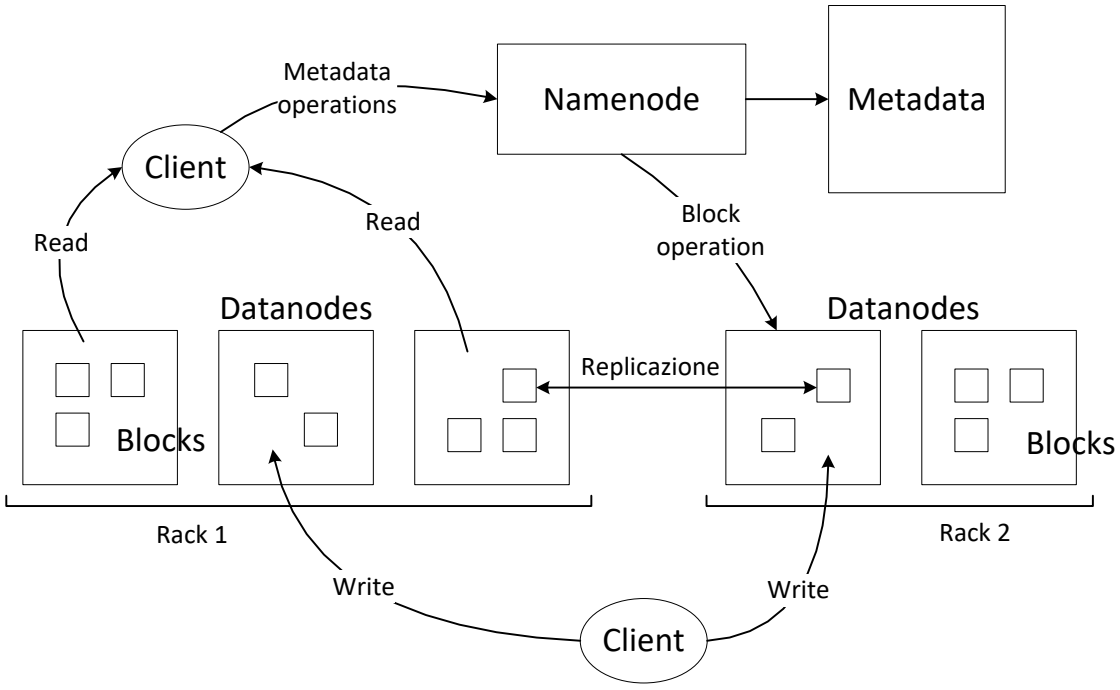

Figura 3: *Architettura Hadoop Distributed File System*

Il NameNode gestisce la struttura gerarchica di file e cartelle, e memorizza per ogni blocco l'insieme dei DataNode che lo contiene, effettuando un vero e proprio mapping tra i nodi e i vari blocchi di file. Inoltre esegue operazioni come l'apertura e la chiusura di file e cartelle.

I DataNode comunicano con il NameNode tramite heartbeats, di default uno ogni 3 secondi. Gli heartbeats hanno un duplice scopo: da un lato di segnalare l'attività e lo stato dei DataNode e di riportane le statistiche (come spazio totale e spazio disponibile), dall'altra di trasportare nelle risposte le istruzioni del NameNode riguardanti le operazioni da svolgere, come ad esempio la rimozione di blocchi o la replicazione su altri nodi.

3.4.2 Hadoop MapReduce

Tipicamente i nodi che si occupano della computazione e quelli che si occupano della memorizzazione dei dati sono gli stessi, questo vuol dire che il framework MapReduce e l'HDFS vengono eseguiti sugli stessi nodi del cluster. Questa configurazione permette al framework di pianificare in modo efficiente i tasks sui nodi in cui sono già presenti i dati che devono essere elaborati.

A livello di infrastruttura, Hadoop MapReduce è costituito da un singolo nodo master chiamato **ResourceManager** e da tanti **NodeManager** quanti sono i nodi del cluster. Inoltre le applicazioni devono specificare i parametri di input e di output e fornire le funzioni *Map* e *Reduce* tramite l'implementazione delle interfacce. Questi parametri costituiscono la configurazione del job MapReduce.

A questo punto l'*Hadoop job-client* invia il job e la sua configurazione al ResourceManager, che si assume la responsabilità di distribuire il software ai vari nodi, di mettere in esecuzione i tasks e di monitorarli, fornendo anche informazioni di stato e di diagnostica.

Il ResourceManager è l'autorità che arbitra e gestisce le risorse tra tutte le applicazioni presenti nel sistema, mentre il NodeManager si occupa di monitorare l'utilizzo delle risorse (CPU, memoria, disco e rete) e di comunicarlo al nodo master.

3.5 Apache Spark

Anche Spark è un framework open-source per l'elaborazione dati su cluster nato per essere veloce ed efficiente, infatti a differenza di Hadoop MapReduce non utilizza solo il disco fisso, ma può effettuare operazioni anche direttamente in memoria centrale, riuscendo ad offrire prestazioni migliori, mantenendo però le proprietà di scalabilità e tolleranza ai guasti tipiche del modello MapReduce.

L’utilizzo ottimale della memoria permette a Spark di essere ordini di grandezza più veloce, rispetto a MapReduce, nell’esecuzione di algoritmi iterativi, cioè quegli algoritmi che svolgono in modo iterativo le stesse istruzioni sui dati.

Entrambi i framework possono funzionare su YARN, il gestore di risorse della piattaforma Hadoop, e per entrambi si può utilizzare l’HDFS come supporto alla memorizzazione dei dati.

L’astrazione principale in Spark è chiamata Resilient Distributed Dataset (RDD), cioè una collezione read-only di oggetti partizionata tra varie macchine e che può essere ricostruito anche se una partizione vien persa. In questo modo gli utenti possono salvare un RDD nella memoria condivisa tra le macchine e sfruttarlo per eseguire multiple operazioni parallele come MapReduce.

Nel suo core, Spark mette a disposizione un motore di computazione che viene sfruttato da tutti gli altri componenti del framework. Il core si occupa delle funzionalità base, come lo scheduling, la distribuzione e il controllo dell’esecuzione dell’applicazione utente. Inoltre vengono messi a disposizione numerosi moduli come *Spark SQL*, *Spark Streaming*, *MLlib* e *GraphX*, che si integrano perfettamente tra di loro e possono essere utilizzati contemporaneamente sulla stessa applicazione.

3.5.1 Architettura di Spark

Spark può lavorare sia in un singolo nodo che in cluster di macchine. Nel caso generale vengono messi in esecuzione per ogni applicazione Spark: un driver e molteplici executor. Il driver è il programma principale di un’applicazione Spark, che assegna i compiti da far svolgere ai vari processi executor, che sono in esecuzione nella macchina client. Inoltre nel driver è presente un oggetto di tipo **Spark Context**, la cui istanza comunica con il **Cluster Manager**, cioè il gestore delle risorse del cluster, per richiedere le risorse di cui hanno bisogno gli executor.

L’architettura è di tipo master/slave, per cui per ogni applicazione Spark ci sarà in esecuzione un driver, che svolge il ruolo di master, mentre gli executor svolgono il ruolo di workers. Un’applicazione Spark si compone di jobs, uno per ogni azione. Si ha cioè un job ogni qualvolta si vuole ottenere dal sistema il risultato di una computazione. Ogni job è composto da un insieme di stage che dipendono l’un l’altro, svolti in sequenza, ognuno dei quali viene eseguito da una moltitudine di task, eseguiti in modo parallelo dagli executor.

Il **driver** è il processo principale, cioè quello in cui viene eseguito il main contenente il codice utente, e può essere eseguito sia all'interno del cluster che nella macchina client che manda in esecuzione l'applicazione. Il codice, che contiene operazioni di trasformazione e azioni sugli RDD, viene eseguito in parallelo dagli executor distribuiti nel cluster. Il compito del driver è quello di convertire il programma in un insieme di tasks, che rappresentano la più piccola unità di lavoro. Inoltre si occupa di fare lo scheduling dei tasks sui vari nodi executor, in base alla località dei file, per evitare il più possibile di trasferirli in rete. Nel caso in cui un nodo fallisce, la piattaforma esegue automaticamente lo scheduling in un altro nodo.

Gli **executor**, invece sono i processi che svolgono i tasks. Ogni applicazione ha i propri executor, ognuno dei quali può avere più thread in esecuzione. Gli executor di diverse applicazioni Spark non possono comunicare tra di loro, perciò diverse applicazioni non possono condividere i dati se non scrivendoli prima su disco.

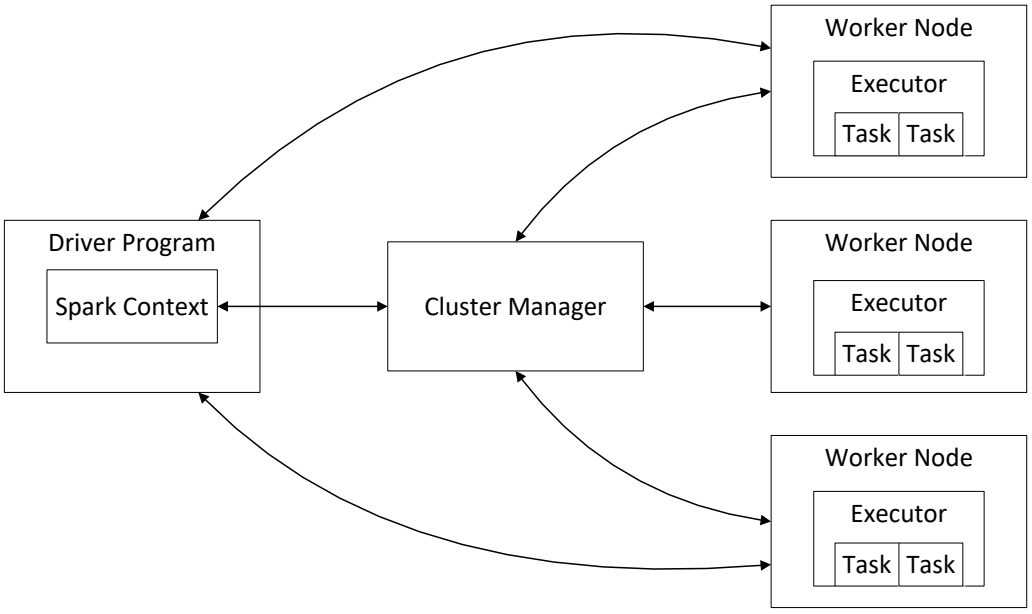

Figura 4: *Schema dell'architettura di Spark*

3.6 Panoramica di esecuzione

Le invocazioni della funzione *Map* sono distribuite su più macchine suddividendo automaticamente i dati di input in un set di *M* parti, le quali possono essere processate in parallelo da macchine differenti.

Le invocazioni della funzione *Reduce*, invece, sono distribuite suddividendo lo spazio delle chiavi intermedie in *R* partizioni, utilizzando una funzione di partizionamento. Il numero delle partizioni *R* e la funzione di partizionamento sono specificate dall'utente.

Quando il programma utente chiama la funzione MapReduce, si verificano le seguenti sequenze di azioni (le etichette numerate nella **Figura 5** corrispondono agli indici della lista):

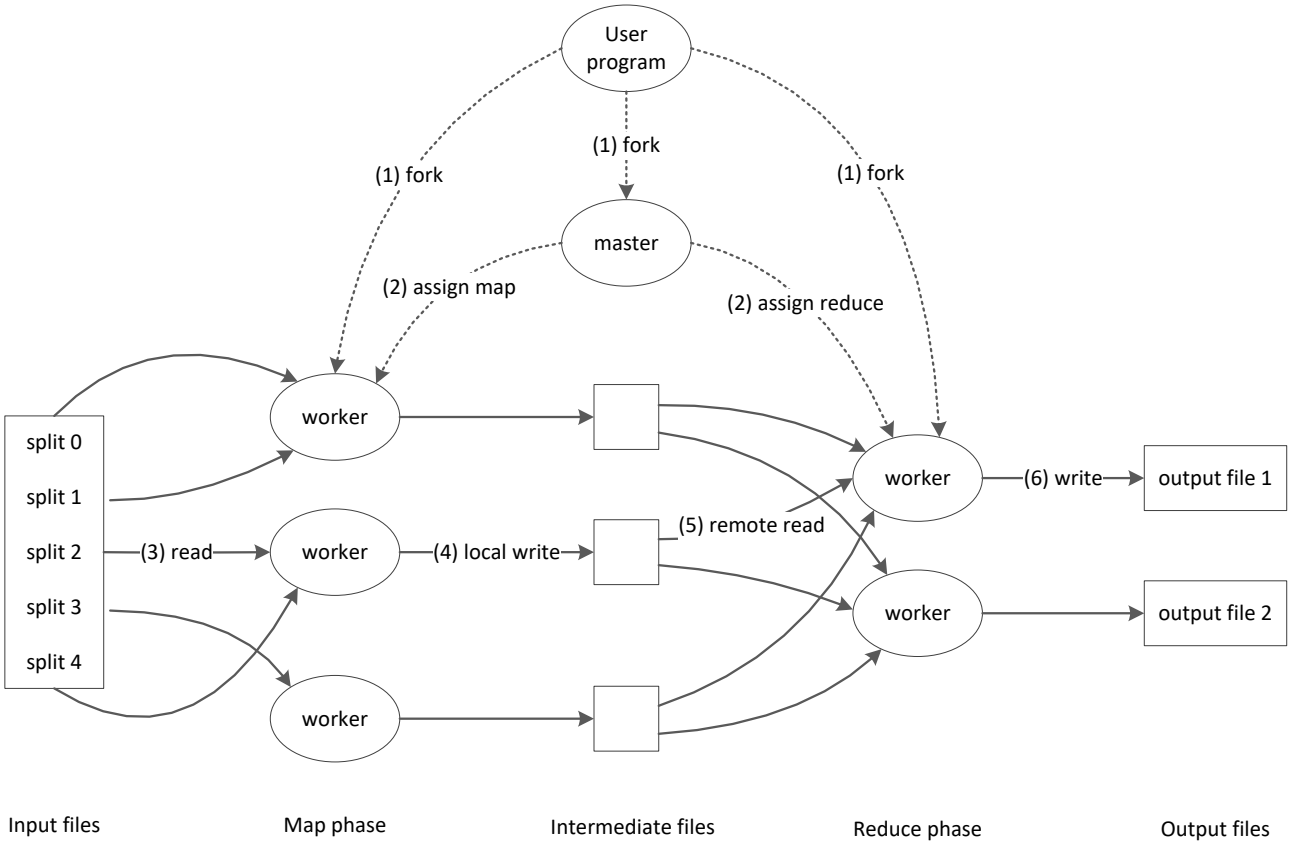

Figura 5: *Panoramica di esecuzione del modello MapReduce*

1. La libreria MapReduce nel programma utente inizialmente divide i file in ingresso in M parti ognuna di dimensione tra i 16 e i 64 MB, poi mette in esecuzione alcune copie del programma in un cluster di macchine.
2. Una delle copie del programma è speciale e viene denominata "master". Alle restanti copie chiamate "workers" viene assegnato il lavoro dal master. Ci sono *M* map tasks e *R* reduce tasks da assegnare. Il master individua i worker inattivi e assegna ad ognuno un map task o un reduce task.
3. Un worker a cui viene assegnato un map task legge il contenuto della corrispondente parte in ingresso, analizza le coppie chiave/valore dei dati che gli vengono passati e

invia ogni coppia alla funzione *Map* definita dall'utente. Le chiavi intermedie prodotte dalla funzione *Map* sono inserite in memoria.

4. Periodicamente le coppie memorizzate in memoria vengono scritte sul disco locale, partizionato in *R* regioni dalla funzione di partizionamento. La posizione di queste coppie memorizzate sul disco locale viene passata indietro al master che è responsabile dell'invio di tali posizioni ai reduce workers.
5. Quando un reduce worker viene notificato dal master di queste posizioni, utilizza chiamate a procedure remote per leggere i dati memorizzati dal disco locale dei map workers. Quando un reduce worker ha letto tutti i dati intermedi, li ordina in base alle chiavi intermedie così che tutte le occorrenze relative alla stessa chiave siano raggruppate insieme. Se la quantità di dati intermedi è troppo grande per entrare in memoria, viene usato un ordinamento esterno.
6. Ogni reduce worker itera i dati intermedi ordinati e ogni volta che incontra una chiave intermedia unica, passa la stessa e il corrispondente insieme di valori intermedi alla funzione *Reduce* definita dall'utente. Il risultato della funzione viene allegato al file finale di output per questa corrispondente partizione.
7. Quando tutti i processi di map e reduce sono completati, il master sveglia il programma utente e la chiamata alla funzione MapReduce ritorna indietro.

Dopo il positivo completamento, il risultato dell'esecuzione è disponibile negli *R* file di output, uno per ogni processo, con un nome specificato dall'utente.

3.7 Conclusione del modello MapReduce

Il modello di programmazione MapReduce è stato utilizzato ampiamente in questi anni per molti scopi diversi; e il successo può essergli attribuito per vari motivi:

- Il modello è di facile utilizzo, anche da parte di programmatori senza esperienza con sistemi distribuiti e paralleli, poiché nasconde tutti i dettagli complessi.
- Una grande varietà di problemi sono facilmente esprimibili come computazioni MapReduce.
- Le implementazioni di MapReduce sono state sviluppate per essere scalabili per grandi cluster di macchine con un uso efficiente delle risorse, e per questo risultano adatte ad essere impiegate per la risoluzione di molti dei grandi problemi computazionali.

4 Google Cloud Dataflow

Google Cloud Platform [8] è un'infrastruttura cloud di Google altamente scalabile e affidabile che mette a disposizione dello sviluppatore una serie di servizi e prodotti di elaborazione, di storage e per lo sviluppo di applicazioni, sulla stessa infrastruttura di supporto che Google utilizza internamente per i propri prodotti.

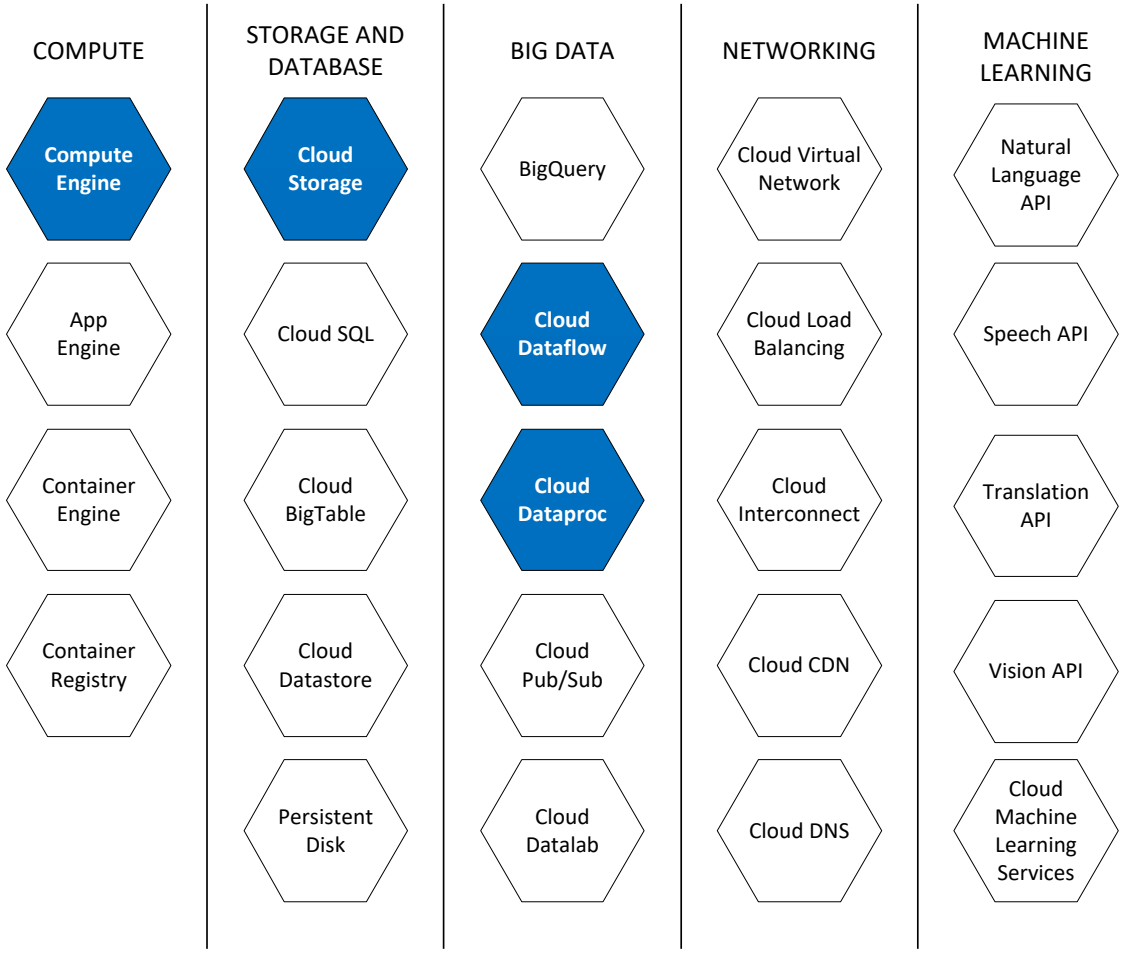

Figura 6: *Servizi offerti dalla Google Cloud Platform*

La Cloud Platform fornisce diversi servizi:

- **Compute**: mette a disposizione una serie di opzioni personalizzabili in base all'esigenze dell'utente, a partire da singole macchine virtuali (Compute Engine), piattaforme per lo sviluppo di applicazioni web o mobile completamente scalabili (App Engine) e strumenti per la gestione dei cluster (Container Engine).

- **Storage and Databases**: offre servizi di archiviazione sul cloud di vario tipo, come un servizio di storage adatto ad ogni tipo di struttura dati (Cloud Storage), database relazionali completamente gestiti (Cloud SQL) o basati su NoSQL (Cloud BigTable) e database con alta disponibilità orientati ai documenti (Cloud Datastore).
- **Big Data**: offre diverse soluzioni per l'elaborazione di BigData. Permette di eseguire query SQL-like in tempo reale (BigQuery), utilizzare un modello di programmazione parallela per esecuzioni sia batch che streaming (Cloud Dataflow) e rende disponibile anche un servizio per gestire le piattaforme Hadoop e Spark direttamente dalla Google Cloud Platform (Cloud Dataproc).
- **Networking**: permette di gestire le connessioni delle proprie risorse virtuali (Cloud Virtual Network) e di dimensionare e distribuire il carico delle risorse delle applicazioni (Cloud Load Balancing).
- **Machine Learning**: mette a disposizione numerose API come quelle per il linguaggio naturale (Natural Language API), il riconoscimento delle parole (Speech API) o effettuare traduzioni (Translation API).
- **Management and Developer Tools**: fornisce un insieme di strumenti e librerie essenziali per sviluppare e gestire le proprie applicazioni.

In particolare, tra tutti i servizi offerti dalla Google Cloud Platform, per la parte sperimentale di questa tesi abbiamo utilizzato il modello di programmazione **Cloud Dataflow** per eseguire il programma di WordCount su una collezione di file, la **Compute Engine** [9] per creare il cluster di macchine virtuali, il **Cloud Storage** [10] per la memorizzazione dei file e la piattaforma **Cloud Dataproc** per gestire le piattaforme Hadoop e Spark.

4.1 Introduzione al modello Cloud Dataflow

Cloud Dataflow [7] è un modello di programmazione unificato e di servizio cloud per lo sviluppo e l'esecuzione di una vasta gamma di elaborazioni dati.

Il servizio gestisce in modo completamente trasparente il ciclo di vita delle risorse e può fornirne dinamicamente su richiesta per ridurre al minimo la latenza e mantenere alta l'efficienza di utilizzo. A differenza di molti altri servizi libera l'utente da compiti come la gestione delle risorse e l'ottimizzazione delle prestazioni.

Le caratteristiche principali di Dataflow sono:

- **Gestione delle risorse automatizzata**: la gestione delle risorse di elaborazione necessarie è completamente automatizzata
- **Disponibilità on demand**: tutte le risorse vengono rese disponibili su richiesta, consentendo di scalare facilmente per soddisfare le proprie esigenze.
- **Programmazione intelligente del lavoro**: il lavoro di partizionamento è reso automatico e ottimizzato, in modo tale da riequilibrare in modo dinamico i ritardi di esecuzione.
- **Autoscaling**: il modello permette di scalare orizzontalmente[7] le risorse in modo automatico per soddisfare i requisiti ottimali.
- **Modello di programmazione unificato**: le API di Dataflow permettono di esprimere operazioni analoghe a MapReduce, effettuare il windowing e il controllo di correttezza dei dati a prescindere dalla sorgente.
- **Monitoring**: essendo integrato nella console *Google Cloud Platform*, Dataflow fornisce statistiche e log di ispezione dei worker quasi in tempo reale.
- **Modello integrato**: si integra facilmente con molti altri servizi offerti dalla piattaforma, come *Cloud Storage*, *Cloud Pub/sub*, *Cloud Datastore*, *Cloud BigTable* e *BigQuery* e può essere esteso per interagire con *Apache Kafka* e *HDFS* (Hadoop File System).
- **Elaborazione affidabile e consistente**: fornisce supporto integrato per la tolleranza ai guasti, che è consistente e corretta indipendentemente dalla dimensione dei dati, dimensione del cluster, modello di elaborazione o complessità della pipeline.
- **Open source**: il modello di programmazione è facilmente estensibile e questo consente agli sviluppatori di implementare Dataflow anche in ambienti di servizio alternativi.

Cloud Dataflow mette a disposizione dello sviluppatore il Dataflow SDK (recentemente sta entrando a far parte del progetto Apache Beam) [13] per i due linguaggi di programmazione Java e Phyton, che può essere utilizzato per eseguire elaborazioni sul servizio cloud di Google.

4.2 Modello di programmazione

Dataflow definisce un modello di programmazione che è stato progettato per semplificare i meccanismi di elaborazione dati su larga scala. Quando programmi utilizzando il Dataflow SDK, essenzialmente stai creando un processo di elaborazione dati che deve essere eseguito su

uno dei servizi runner (*PipelineRunner*) di Cloud Dataflow, i quali hanno il compito di analizzare il programma e costruire la pipeline.

I servizi runner possono essere di 3 tipi: *DirectPipelineRunner*, *DataflowPipelineRunner* e *BlockingDataflowPipelineRunner*. Il DirectPipelineRunner esegue la pipeline sulla macchina locale, gli altri due runner invece eseguono sul cloud; il DataflowPipelineRunner permette di eseguire la pipeline in modo asincrono e durante l'esecuzione permette di monitorare i progressi del job, mentre il BlockingDataflowPipelineRunner esegue in modo sincrono e quindi aspetta la terminazione del job.

Questo modello di programmazione permette di concentrarsi sulla composizione logica del tuo processo di elaborazione, piuttosto che sulla coordinazione fisica per l'esecuzione in parallelo. È possibile quindi concentrarsi su ciò che è necessario che faccia il processo, invece di come esattamente deve essere eseguito.

Il modello Dataflow fornisce una serie di utili astrazioni che si isolano dai dettagli di basso livello dell'elaborazione distribuita, come ad esempio il coordinamento dei singoli worker. Questi dettagli di basso livello sono completamente gestiti per voi da parte dei servizi runner di Cloud Dataflow.

I concetti fondamentali nel processo di elaborazione dati con Dataflow sono:

- **Pipeline**: rappresenta un singolo servizio (job o unità di lavoro), potenzialmente ripetibile, dall'inizio alla fine.
- **PCollection**: rappresenta un insieme di dati, in particolare l'input e l'output di ogni operazione di una pipeline.
- **Transform**: è un'operazione di elaborazione dati della pipeline, prende in ingresso una o più PCollections, esegue una funzione di elaborazione, fornita dall'utente, su gli elementi di tale collezione, e produce una PCollection in uscita.
- **I/O Source e Sink**: sorgenti di dati per gestire l'input e l'output delle pipeline.

Nei paragrafi successivi verranno approfonditi tutti questi concetti.

4.3 Pipeline

Una Pipeline rappresenta un servizio di elaborazione dati (data processing job) e consiste in un set di operazioni che può leggere i dati in input da una sorgente esterna, trasformare quei dati e

scrivere l'output risultante. I dati e le trasformazioni in una pipeline sono uniche e appartengono a quella specifica pipeline. Se un programma crea e utilizza pipeline multiple, queste non possono condividere dati o trasformazioni.

Ogni Pipeline può avere diversi gradi di complessità: può essere relativamente semplice e lineare, se le trasformazioni sono collegate in sequenza e quindi eseguite una dopo l'altra, o può risultare più complessa usando condizioni, cicli e altre strutture tipiche della programmazione.

4.3.1 Pipeline data e Pipeline transform

Una Pipeline è costituita da due parti: i dati (PCollection) e le trasformazioni (Transform) applicate ai dati. L'SDK di Dataflow fornisce le classi per rappresentare sia i dati che le trasformazioni.

Le Pipeline usano una collezione specializzata di dati chiamata PCollection per rappresentare sia i dati in input, sia quelli in output che quelli intermedi. PCollection può essere utilizzata per rappresentare insiemi di dati di qualunque dimensione. PCollection a differenza della classe java Collection è specifica per il supporto alla elaborazione parallela.

Una trasformazione (Transform) rappresenta una singola operazione della pipeline. Ogni transform prende in input una o più PCollection, cambia o altrimenti manipola gli elementi della collezione e produce una o più PCollection in output.

4.3.2 Creazione di una Pipeline

L'esempio seguente mostra la costruzione e l'esecuzione di una pipeline con tre trasformazioni: una trasformazione per leggere alcuni dati in input, una per contarli e infine una per scrivere il risultato del conteggio.

```
public static void main(String[] args) {
        // Create a pipeline parameterized by command line flags
        Pipeline p =
              Pipeline.create(PipelineOptionsFactory.fromArgs(arg));

        p.apply(TextIO.Read.from("gs://..."))     // Read input
         .apply(new CountWords())                 // Do some processing
         .apply(TextIO.Write.to("gs://..."));     // Write output

        // Run the pipeline
        p.run();
}
```

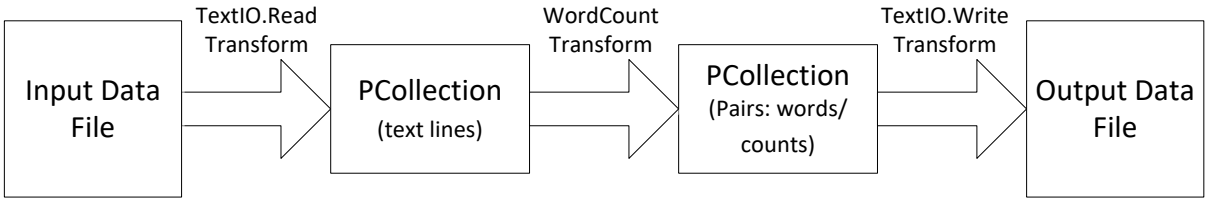

Figura 7: S*equenza di esecuzione delle trasformazioni di una pipeline*

4.4 PCollection

Una PCollection rappresenta un insieme potenzialmente numeroso e immutabile di elementi. Non esiste un limite massimo al numero di elementi che un PCollection può contenere.

Ogni PCollection ha alcuni aspetti chiave con cui si differenzia da una normale collezione:

- è immutabile, quindi una volta che viene creata non si possono più aggiungere, rimuovere o modificare i singoli elementi che ne fanno parte.
- non supporta l'accesso casuale ai singoli elementi.
- appartiene alla pipeline in cui viene creata, quindi non è possibile condividere una PCollection con le altre pipeline.

4.4.1 PCollection limitate e illimitate

La dimensione di una PCollection può essere limitata o illimitata, e questa viene determinata quando viene creata la collezione di dati. Alcune root transform producono PCollection limitate, mentre altre ne creano di illimitate; spesso dipende dalla sorgente dei dati in ingresso.

Una PCollection viene detta limitata se rappresenta un insieme di dati fisso, con una propria dimensione che non cambia. Al contrario invece una PCollection si dice illimitata se rappresenta un insieme di dati in continuo aggiornamento o più semplicemente uno streaming di dati. Alcune sorgenti, in particolare quelle che creano PCollection illimitate, aggiungono automaticamente ad ogni elemento della collezione un timestamp, che verrà utilizzato in seguito per raggruppare i dati in finestre logiche di dimensioni finite (Windowing).

Il fatto che una PCollection sia limitata o meno determina il modo in cui Dataflow elabora quei dati. Le collezioni limitate possono essere elaborate utilizzando dei *batch jobs*, i quali leggono l'intero set di dati una volta sola ed eseguono l'elaborazione. Le collezioni illimitate invece

devono essere elaborate usando degli *streaming jobs*, visto che l'intera collezione non può essere disponibile tutta in una volta.

4.4.2 Creazione di una PCollection

Dataflow fornisce due metodi per creare una PCollection iniziale: puoi leggere i dati da una sorgente esterna, come ad esempio un file o puoi creare una PCollection di dati che sono memorizzati in memoria.

```
// 1. READING EXTERNAL DATA
// Read transform read data from an external source and return a
// PCollection representation of the data for use by your pipeline.
p.apply(TextIO.Read.named("ReadFromText")
                                .from("gs://my_bucket/path/to/input/");


// 2. CREATE A PCOLLECTION FROM DATA IN LOCAL MEMORY
// Create is a root Transform, that accept a java Collection and a
// Coder object, which specifies how the elements should be encoded.
static final List<String> LINES = Arrays.asList(
                    "line one", "line two", "line three", "line four");
PipelineOptions options = PipelineOptionsFactory.create();
Pipeline p = Pipeline.create(options);
p.apply(Create.of(LINES)).setCoder(StringUtf8Coder.of())
```

Inoltre è possibile creare delle collezioni di dati dove il tipo degli elementi è un tipo di dato custom che viene fornito dall'utente. Questo può essere utile nel caso in cui l'utente abbia bisogno di creare una collezione di una propria classe o di una struttura con campi specifici. Quando si crea una PCollection di tipo custom, è necessario fornire un Coder per quel particolare tipo. Il Coder comunica al servizio Dataflow come serializzare e deserializzare gli elementi della collezione e come il set di dati è parallelizzato e partizionato tra le varie istanze di pipeline.

4.5 Transform

All'interno di una Pipeline, una trasformazione rappresenta un passo o un'operazione che processa i dati. Un oggetto Transform è in grado di eseguire quasi ogni tipo di operazione di elaborazione, come l'esecuzione di calcoli matematici, il raggruppamento, la lettura o la scrittura dei dati. Inoltre nel modello Dataflow le trasformazioni possono essere innestate, cioè possono esistere trasformazioni che invocano altre trasformazioni, formando così delle

trasformazioni composite. La classe Transform rappresenta la logica di elaborazione delle pipeline.

```
// The input PCollection of word strings.
PCollection<String> words = ...;

// The ComputeWordLengths transform takes a PCollection of Strings
// as input and returns a PCollection of Integers as output.
static class ComputeWordLength
       extends PTransform<PCollection<String>, PCollection<Integer>> {
       //Do something ...
}

// To use a transform, you apply the transform to the input
// PCollection that you want to process
PCollection<Integer> wordLengths = words.apply(
                                          new ComputeWordLength());
```

4.5.1 Tipi di trasformazioni

All'interno del Dataflow SDK sono presenti tre diversi tipi di trasformazioni: Core Transform, Composite Transform e Root Transform.

Le **Core Transforms** rappresentano la base del modello di elaborazione parallelo di Google Cloud Dataflow e quindi dell'elaborazione con le pipeline. Ogni volta che viene utilizzata una Core Transform bisogna fornire la logica di elaborazione tramite una funzione, la quale viene applicata agli elementi della PCollection in input. L'istanza della funzione viene eseguita in parallelo su differenti istanze di Google Compute Engine e produce gli elementi che vengono poi inseriti nella PCollection in output. Queste trasformazioni inoltre vengono utilizzate direttamente nella pipeline.

Alcuni esempi di Core Transform sono: *ParDo* per eseguire una generica elaborazione parallela, *GroupByKey* utilizzata per raggruppare in coppie chiave/valore, *Combine* utilizzata per combinare collezioni di valori e *Flatten* per raggruppare insieme gli elementi di più collezioni.

```
// The input PCollection of word strings.
PCollection<String> words = ... ;

// The DoFn to perform on each element in the input PCollection.
static class ComputeWordLengthFn
       extends DoFn<String, Integer> {
             //Do something ...
       }

// Apply a ParDo to the input PCollection "words" to compute
```

```
// lengths for each word.
PCollection<Integer> wordLengths = words.apply(
                        ParDo.of(new ComputeWordLengthFn()));
// ComputeWordLenghtFn is the DoFn function to perform on each
// element, which we define above.
```

Le **Composite Transforms** sono trasformazioni costituite da più sotto-trasformazioni. Il modello delle trasformazioni infatti risulta modulare, cioè si possono costruire trasformazioni che sono implementate in termini di altre trasformazioni.

Le **Root Transforms** sono trasformazioni specializzate per prelevare o inserire i dati in una pipeline e spesso vengono utilizzate come endpoint di una pipeline, anche se possono essere utilizzate in qualunque punto della pipeline. Queste trasformazioni vengono classificate in base alla loro funzione: le *read transforms* leggono i dati da una sorgente esterna e producono una PCollection contenenti gli elementi appena letti da utilizzare in una pipeline, le *write transforms* scrivono i dati contenuti all'interno di una PCollection su una fonte di dati esterna. Infine appartengono a questa categoria anche le *create transforms* che sono utilizzate per creare PCollection a partire da dati che si trovano già in memoria.

4.6 Esecuzione del servizio

Una volta che è stata costruita la pipeline, si può sfruttare il servizio di Cloud Dataflow per effettuarne il deploy e in seguito eseguirla; in questo modo la pipeline diventa un Dataflow job. Oltre a gestire le risorse, il servizio Dataflow svolge e ottimizza molti aspetti legati all'elaborazione parallela distribuita, tra cui:

- **Parallelismo e distribuzione**: partiziona automaticamente i dati e distribuisce il codice alle istanze di Compute Engine per la vera e propria esecuzione parallela.
- **Ottimizzazione**: utilizza il codice della pipeline per creare il grafico di esecuzione che rappresenta i dati e le trasformazioni della pipeline stessa e lo ottimizza per ottenere le performance più efficienti e un miglior utilizzo delle risorse.
- **Funzioni automatiche**: fornisce alcune funzioni che aggiustano l'allocazione delle risorse e il partizionamento dei dati, come l'**Autoscaling** e il **Dynamic Work Rebalancing**, che permettono di eseguire il job più in fretta e in modo più efficiente possibile.

4.6.1 Ciclo di vita di una Pipeline

La fase in cui viene costruito il grafico di esecuzione a partire dalla pipeline prende il nome di **Graph Construction Time**. Durante questo periodo viene verificata la presenza di errori e viene garantito che non contenga operazioni illegali. Il grafico viene poi tradotto in formato JSON e trasmesso all'endpoint del servizio Dataflow, il quale valida il grafico e in seguito lo trasforma in un job.

4.6.2 Grafico di esecuzione

Quando viene messa in esecuzione una pipeline, Dataflow costruisce un diagramma di passi che rappresenta la pipeline, sulla base dei dati e delle trasformazioni che sono state utilizzate per costruirla. Il grafico di esecuzione è uno strumento fondamentale per visualizzare in modo semplice l'ordine in cui le trasformazioni vengono eseguite e come queste vengano espanse in sotto-trasformazioni durante l'esecuzione.

Una volta che il grafico di esecuzione di una pipeline in formato JSON è stato validato, il servizio di gestione di Cloud Dataflow può modificarlo per effettuare alcune ottimizzazioni prima di eseguirlo sulle risorse cloud, come ad esempio unire più operazioni o più trasformazioni in un singolo passaggio. In questo modo il servizio non deve costruire ogni PCollection intermedia, che potrebbe risultare costosa in termini di memoria e di numero di elaborazioni.

Mentre tutte le trasformazioni che sono state specificate nella costruzione della pipeline vengono eseguite, queste però possono essere svolte in un ordine differente, o come parte di una più grande trasformazione ottenuta dall'unione di più operazioni, per garantire un'esecuzione più efficiente. Dataflow infatti rispetterà tutte le dipendenze dei dati tra i vari passaggi del grafico di esecuzione, ma ogni passaggio può essere eseguito in qualsiasi ordine.

La **Figura 8** mostra come il grafico di esecuzione di un programma di WordCount viene ottimizzato e come le trasformazioni sono unite insieme per garantire maggiore efficienza:

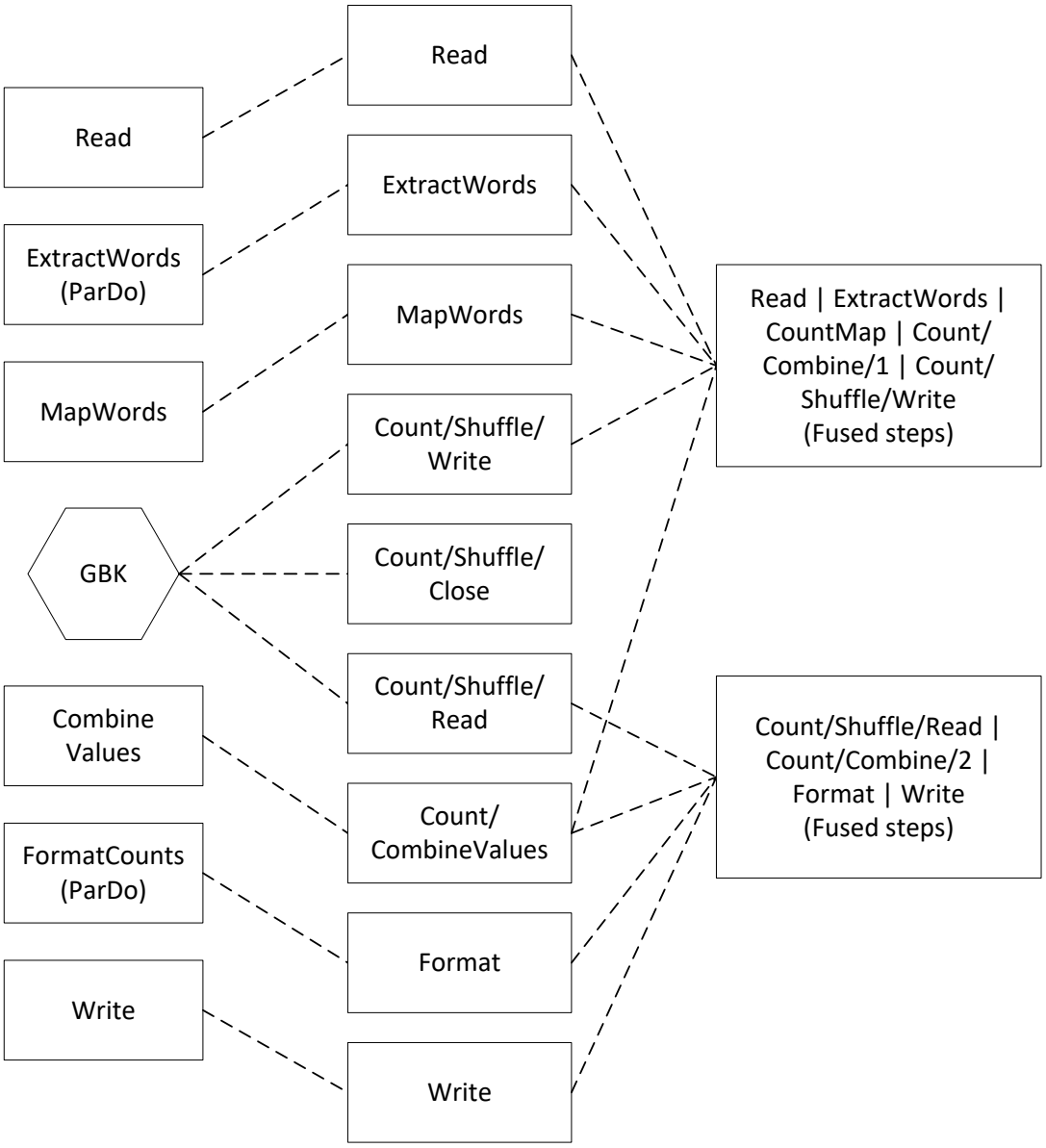

Figura 8: *Grafico di esecuzione ottimizzato del programma di WordCount*

4.6.3 Utilizzo e gestione delle risorse

Il servizio Dataflow gestisce completamente le risorse della Google Cloud Platform, come ad esempio l'accensione o lo spegnimento di istanze di **Google Compute Engine** (a volte indicate come workers o macchine virtuali) e l'accesso ai **Google Cloud Storage Buckets** relativi al tuo progetto sia per l'input e l'output sia per la memorizzazione di file temporanei.

Ogni utente può eseguire al massimo 25 Dataflow jobs concorrenti per ogni progetto Cloud Platform e il servizio Dataflow è al momento limitato a processare richieste di job in formato JSON che hanno dimensione massima di 10MB.

Inoltre Dataflow attualmente consente un massimo di 1000 istanze di Compute Engine per ogni job. Il tipo di macchina utilizzata di default è *n1-standard-1* per un processo batch, mentre per lo streaming viene utilizzata la *n1-standard-4*, che dispone di 4 core per ogni processore virtuale e quindi il servizio può allocare fino a 4000 core per ogni job.

Ogni istanza di worker è fornita di 15 dischi persistenti e questi dischi sono locali per ogni macchina virtuale di Compute Engine. Un job quindi non può avere un numero di workers più alto del numero di dischi persistenti. La dimensione di ogni disco è di 250GB in modalità batch e 400GB in modalità streaming.

4.6.4 Parallelismo e distribuzione

La logica di elaborazione delle pipeline viene automaticamente parallelizzata e distribuita ai workers che sono stati assegnati a svolgere quello specifico job. In particolare Dataflow utilizza l'astrazione del proprio modello di programmazione per rappresentare il parallelismo, in cui il codice di elaborazione (rappresentato dalle funzioni *DoFn*) viene distribuito tra multipli workers che vengono fatti eseguire in parallelo.

La trasformazione *ParDo* viene utilizzata per eseguire una generica elaborazione parallela al cui interno viene specificata una funzione *DoFn*, che rappresenta l'operazione da svolgere sui dati contenuti nella PCollection su cui sta operando la trasformazione. Si può quindi considerare la funzione *DoFn* come una entità indipendente di cui possono esistere più istanze in esecuzione su differenti macchine, in modo indipendente l'una dall'altra.

Il servizio Dataflow garantisce che ogni elemento della PCollection venga elaborato da un'istanza *DoFn* una volta sola, ma non garantisce come questi elementi vengano raggruppati per essere processati insieme. Inoltre non viene specificato il numero di istanze *DoFn* che verranno create durante l'esecuzione della pipeline.

4.6.5 Autotuning

Il servizio Dataflow gestisce in modo automatico tramite l'**Autoscaling** il numero appropriato delle istanze di workers necessari per l'esecuzione di un job. Inoltre può anche allocare o de-allocare dinamicamente uno o più workers a run-time in modo da soddisfare alcune caratteristiche di esecuzione richieste dal job. Alcune parti della pipeline possono essere computazionalmente più pesanti e complesse di altre e quindi durante queste fasi verranno utilizzati dei worker addizionali, che verranno rilasciati quando non saranno più necessari.

Per le elaborazioni batch, Dataflow sceglie automaticamente il numero dei workers basandosi sia sulla mole di lavoro da eseguire in ogni fase della pipeline sia sul throughput di quella specifica fase. Per quanto riguarda lo streaming di dati invece l'autoscaling permette di adattare il numero dei workers utilizzati in risposta ai cambiamenti di carico e utilizzo delle risorse, ed è stato progettato per ridurre i costi delle risorse utilizzate durante l'esecuzione.

Il **Dynamic Work Rebalancing** invece permette al servizio di partizionare dinamicamente il lavoro in base a determinate condizioni a run-time, come il ritardo di alcuni workers che impiegano più tempo del previsto a terminare, e può riassegnare quindi il lavoro a workers inutilizzati o utilizzati al di sotto delle loro potenzialità per ridurre il tempo di elaborazione complessivo. Questa funzione però ha una limitazione, infatti viene utilizzata sono quando devono essere processati dei dati di input in parallelo.

5 Confronto tra Cloud Dataflow e le piattaforme Spark e Hadoop

Il progetto di tesi che si è sviluppato è stato svolto con l'intento di analizzare e confrontare in modo sperimentale le caratteristiche e le performance dei sistemi Cloud Dataflow, Apache Hadoop e Apache Spark con l'esecuzione di programmi di WordCount basati sul modello MapReduce.

Un programma di WordCount conta le occorrenze di ogni singola parola all'interno di una collezione di documenti, e rappresenta un classico problema che può essere risolto mediante un modello di calcolo parallelo.

5.1 Configurazione del cluster

Per gli esperimenti è stato utilizzato un cluster di macchine appartenenti al data center globale della Google Cloud Platform, messo a completa disposizione per questo progetto di tesi. Dal punto di vista hardware, il cluster utilizzato è composta da 10 macchine virtuali identiche le cui caratteristiche sono riassunte nella **Tabella 1**.

Ogni macchina del cluster è di tipo *n1-standard-1* ed essendo una macchina standard *Compute Engine* è particolarmente adatta per attività che hanno un equilibrio di esigenze di utilizzo della CPU e della memoria. Ogni istanza è fornita di 15 dischi persistenti e questi dischi sono locali per ogni macchina virtuale, mentre la CPU virtuale è implementata come un singolo hardware hyper-thread, il che permette di utilizzare le risorse in modo più efficiente, consentendo a due thread di girare sullo stesso core.

Macchina	Descrizione	N. CPU virtuali	Memoria	GCEUs	Dischi persistenti
n1-standard-1	Macchina di tipo standard con 1 CPU virtuale e 3.75 GB di memoria	1	3.75 GB	2.75	15

Tabella 1: *Caratteristiche del tipo di macchina virtuale utilizzata nel cluster*

La Google Compute Engine Unit (GCEU) è l'unità di capacità della CPU utilizzata per descrivere la relativa potenza delle macchine nella Google Cloud Platform (2.75 GCEUs rappresenta la potenza minima di una CPU virtuale).

5.1.1 Hadoop e Spark sulla Google Cloud Platform

Cloud Dataproc [11] è un servizio basato sul cloud offerto dalla Google Cloud Platform che permette di gestire anche piattaforme come Hadoop e Spark. Dataproc sfrutta molte tecnologie messe a disposizione dalla piattaforma Google, come *Compute Engine* e *Cloud Storage*, per offrire cluster completamente gestiti, che però utilizzano alcuni framework di elaborazione dati esterni come Apache Hadoop e Apache Spark.

Cloud Dataproc separa l'elaborazione e la memorizzazione, infatti utilizza le macchine virtuali della Compute Engine per l'esecuzione e il Cloud Storage per la memorizzazione dei file. Dataproc ha un set di meccanismi di controllo e integrazione che permettono di coordinare il ciclo di vita, la gestione e il coordinamento del cluster. Inoltre è anche integrato con Hadoop YARN in modo tale da rendere la gestione delle risorse e l'utilizzo del cluster molto più semplici.

Questo servizio include alcuni pacchetti open source utilizzati nell'elaborazione dati, tra cui sono compresi anche elementi provenienti dagli ecosistemi di Apache, e strumenti per connettere facilmente questi framework con gli altri prodotti della piattaforma Google.

5.2 Eseguire il modello MapReduce su Dataflow

Nonostante il modello di esecuzione di Dataflow sia completamente diverso da quello utilizzato nel modello MapReduce è comunque possibile mettere in esecuzione delle pipeline che utilizzano particolari trasformazioni che processano i dati con uno stile di elaborazione molto simile a quello di MapReduce. In particolare mi sono concentrato sull'analisi dell'esecuzione di un programma in stile *Map/Shuffle/Reduce* sulla piattaforma Cloud Dataflow.

Dataflow costruisce un grafico di esecuzione a partire dalla pipeline che viene costruita e la trasforma in un Dataflow job, che poi viene eseguito direttamente dal servizio.

```
// Main del programma di WordCount
// Prima configura i parametri della pipeline e poi la esegue
public static void main(String[] args) {
        DataflowPipelineOptions options = PipelineOptionsFactory
```

```
                        .as(DataflowPipelineOptions.class);

        options.setProject("mapreduce-apps-in-hybrid-cloud");
        options.setStagingLocation("gs://mapreduce-apps-in-hybrid-
                                                cloud-bucket/staging");
        options.setRunner(BlockingDataflowPipelineRunner.class);
        // Eventualmente si possono aggiungere anche le seguenti righe
        // per impostare il numero di workers da utilizzare
        // options.setNumWorkers(10);
        // options.setMaxNumWorkers(15);

        Pipeline p = Pipeline.create(options);

        p.apply(TextIO.Read.named("Read").from("gs://mapreduce-apps-in-
                                    hybrid-cloud-bucket/input/*"))
         .apply(new CountWords())
         .apply(ParDo.of(new FormatAsTextFn()))
         .apply(TextIO.Write.named("Write").to("gs://mapreduce-apps-in-
                                    hybrid-cloud-bucket/output.txt"));
        p.run();
} // Main
```

Questo programma configura, costruisce e successivamente mette in esecuzione una pipeline che è composta da una sequenza di operazioni che elaborano i dati prelevati dalla sorgente esterna e produce il conteggio di ogni singola parola. Le trasformazioni che costituiscono la pipeline in ordine sono:

- **Read:** legge i dati in input prelevandoli dalla sorgente esterna indicata come argomento del metodo *from* e costruisce la PCollection iniziale della pipeline
- **CountWords**: converte una PCollection contenente un insieme di righe di testo in una PCollection di singole parole
- **ParDo**: formatta gli elementi della PCollection che riceve in ingresso, in questo caso converte le coppie chiave/valore ricevute dalla CountWords in stringhe stampabili.
- **Write**: scrive i dati contenuti in una PCollection su una fonte di dati esterna.

Di seguito viene presentato il codice della funzione *ExtractWordsFn* che viene utilizzata dalla trasformazione composta *CountWords*.

```
// ExtractWordFn: divide le linee di testo in singole parole
static class ExtractWordsFn extends DoFn<String, String> {
        private static final long serialVersionUID = 1L;
        private final Aggregator<Long, Long> emptyLines =
                createAggregator("emptyLines", new Sum.SumLongFn());

        @Override
        public void processElement(ProcessContext c) {
                if (c.element().trim().isEmpty()) {
                        emptyLines.addValue(1L);
```

```
                }
                String[] words = c.element().split("[^a-zA-Z']+");
                for (String word : words) {
                      if (!word.isEmpty()) {
                            c.output(word);
                      }
                }
          }
    } // ExtractWordFn


    // CountWords: converte una PCollection contenente linee di testo
    // in una PCollection di singole parole
    public static class CountWords extends PTransform<PCollection<String>,
                                        PCollection<KV<String, Long>>> {
          private static final long serialVersionUID = 1L;

          @Override
          public PCollection<KV<String, Long>>
                                        apply(PCollection<String> lines){
                PCollection<String> words = lines.apply(ParDo
                                              .of(new ExtractWordsFn()));
                PCollection<KV<String, Long>> wordCounts = words
                                  .apply(Count.<String>perElement());
                return wordCounts;
          }
    } // CountWords
```

La trasformazione *CountWords* è una Composite Transform costituita dalle trasformazioni *ParDo*, che sfrutta la funzione *ExtractWordsFn* per dividere tutte le righe di testo contenute nella PCollection in singole parole, e *Count.PerElement*, la quale effettua la vera e propria operazione di conteggio delle occorrenze di ogni singolo elemento.

Count è una Pre-Written Transform dell'SDK di Dataflow e più in particolare è una Composite Transform costruita utilizzando le implementazioni delle trasformazioni *ParDo* e *Combine*, che a sua volta risulta essere una Composite Transform che esegue una *GroupByKey* e successivamente raggruppa tutti i singoli valori in base alla loro chiave. La trasformazione *Count* quindi accetta una collezione di elementi e restituisce un insieme ridotto di soli elementi unici ciascuno associato al proprio contatore delle occorrenze.

La trasformazione *ParDo* è l'operazione di elaborazione parallela generica e in questo caso la sua elaborazione è molto simile a quello che avviene durante la fase Map di una funzione MapReduce, poiché crea le coppie chiave/valore per ogni elemento, ma può essere utilizzata anche in modo molto simile alla fase di Reduce. Mentre la *GroupByKey* è analoga alla fase di Shuffle, in cui tutti i valori vengono aggregati tramite una chiave in comune univoca.

```
// FormatAsTextFn: converte una parola e il suo contatore in una
// stringa stampabile
      public static class FormatAsTextFn extends
                                    DoFn<KV<String, Long>, String> {
      private static final long serialVersionUID = 1L;

      @Override
      public void processElement(ProcessContext c) {
            c.output(c.element().getKey() + ": " +
                                    c.element().getValue());
      }
} // FormatAsTextFn
```

Infine prima di eseguire la trasformazione Write, viene eseguita una trasformazione *ParDo* che utilizza la funzione *FormatAsTextFn* che formatta le coppie chiave/valore contenute all'interno della PCollection in stringhe stampabili e di più facile lettura.

Quando viene messa in esecuzione la pipeline corrispondente al programma di WordCount viene creato il grafico di esecuzione (**Figura 9**), in cui si possono vedere in ordine tutte le trasformazioni che vengono eseguite sui dati della PCollection iniziale e come queste vengano espanse in sotto-trasformazioni.

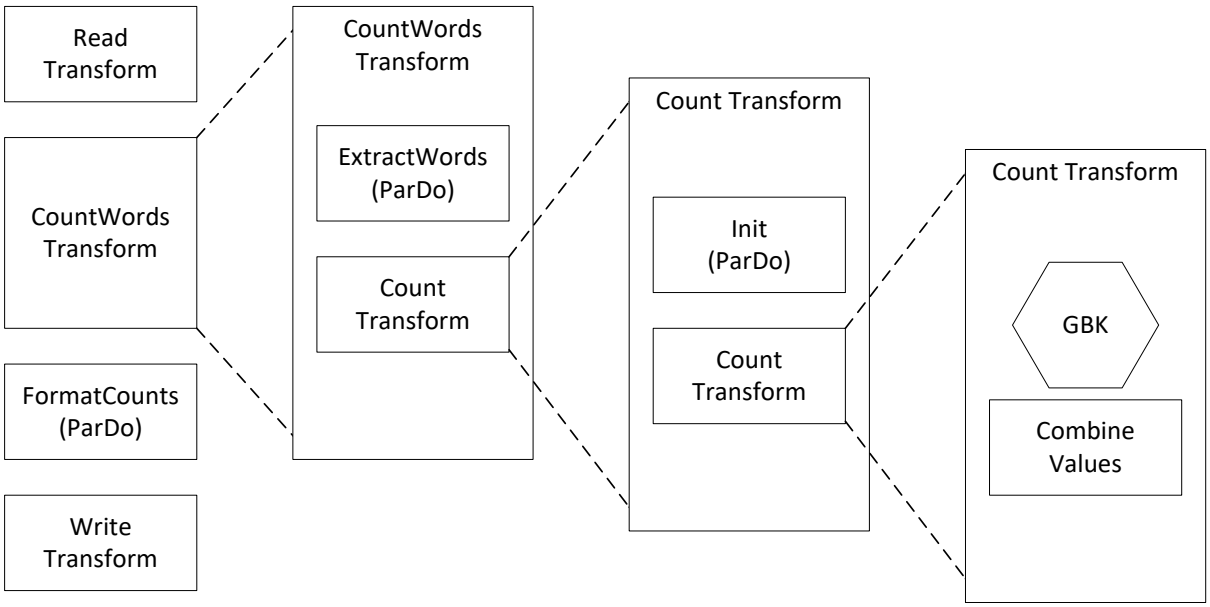

Figura 9: *Grafico di esecuzione prodotto dalla pipeline del programma WordCount*

5.3 Analisi dei risultati ottenuti

I test sulle tre differenti piattaforme sono stati effettuati sull'esempio WordCount implementato nel modello MapReduce, disponibili interamente nel **Cap 6**, e che contano le occorrenze delle

parole contenute in una collezione di file di testo di varie dimensioni, in particolare: 1GB, 2GB, 5GB, 8GB e 10GB. Abbiamo utilizzato file di dimensioni diverse in modo tale da cercare di analizzare le performance in varie condizioni.

I risultati con i vari tempi di esecuzione dei programmi di WordCount sulle diverse piattaforme e in base alla dimensione dei file in input utilizzati, sono raggruppati nella **Tabella 2**.

	1GB (s)	2GB (s)	5GB (s)	8GB (s)	10GB (s)
Dataflow	211	232	299	376	621
Hadoop	147	219	439	716	881
Spark	81	87	107	145	172

Tabella 2: *Tempi di esecuzione delle diverse piattaforme*

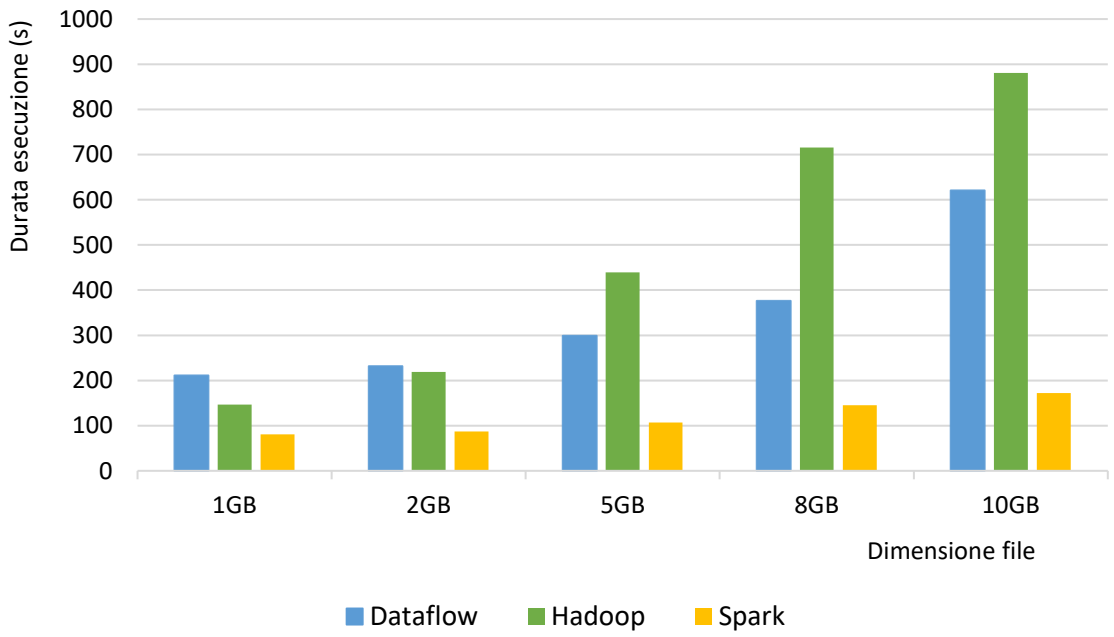

Figura 10: *Grafico che mostra il confronto tra i vari tempi di esecuzione*

Dal grafico (**Figura 10**), che confronta i risultati ottenuti, è facilmente intuibile che tra le tre piattaforme utilizzate Spark è sicuramente la più veloce indipendentemente dalla dimensione del file di testo. Questo è possibile poiché Spark a differenza delle altre piattaforme utilizza in modo efficiente la memoria centrale piuttosto che il disco per salvare i dati che vengono utilizzati durante l'esecuzione, riuscendo ad offrire prestazioni migliori, mantenendo le

proprietà di scalabilità e tolleranza ai guasti tipiche del modello MapReduce. Spark riesce quindi ad essere ordini di grandezza più veloce, rispetto ad Hadoop e Cloud Dataflow, nell'esecuzione di algoritmi iterativi.

Le altre due piattaforme invece presentano delle performance molto simili con file di testo di piccole dimensioni, 1GB o 2GB, mentre all'aumentare della dimensione Cloud Dataflow esegue il programma di WordCount in tempi più rapidi rispetto ad Hadoop.

La piattaforma Cloud Dataflow riesce a mantenersi al di sotto dei 400 secondi di esecuzione analizzando un file di 8GB, quindi riesce a completare la computazione nella metà del tempo di Hadoop, che con lo stesso file impiega invece 716s. Possiamo quindi dedurre che nei nostri test Dataflow riesce a mantenere una buona efficienza e utilizzo delle risorse fino agli 8GB, mentre con i 10GB peggiora notevolmente le performance, mantenendosi pur sempre migliore di Hadoop. Questo probabilmente è legato al fatto che tutte le esecuzioni, per essere uniformi l'una con l'altra, sono state effettuate utilizzando un cluster di 10 macchine virtuali indipendentemente dalla dimensione del file da analizzare. Per questo si può desumere che con 10GB di file la piattaforma avrebbe allocato più di 10 macchine virtuali, risultando quindi nei test più lenta del previsto a causa dei ritardi di alcune macchine in cui si è accumulato troppo lavoro da svolgere. Infatti eseguendo la stessa computazione ma attivando l'autoscaling, Dataflow impiega solo 480s (vedi **Tabella 3**) contro i 621s ottenuti nel test, ma allocando 12 workers invece che 10, quindi due workers in più rispetto a quelli che abbiamo impostato nel cluster di prova.

	5GB	10GB
Dataflow (autoscaling)	425 s (5/6 workers)	480 s (12 workers)

Tabella 3: *Tempi di esecuzione della piattaforma Cloud Dataflow con funzione di autoscaling attiva*

	5GB	10GB
Dataflow (5 workers)	408 s	721 s
Dataflow (15 workers)	262 s	345 s

Tabella 4: *Tempi di esecuzione della piattaforma Cloud Dataflow con cluster di 5 e 10 workers*

A fronte di queste considerazioni sono stati effettuati altri test su Cloud Dataflow con cluster di 5 e 10 workers, in modo tale da testare la capacità di autoscaling della piattaforma. Come è possibile vedere dalla **Figura 11**, Dataflow svolge le esecuzioni con la funzione di autoscaling attiva in maniera assolutamente conforme agli altri dati raccolti nei test. Infatti nel caso della collezione di file da 5GB, impiega quasi lo stesso tempo ottenuto nell'esecuzione con 5 workers, mentre con la collezione di file da 10GB si pone proprio a metà tra i tempi ottenuti con 10 e 15 workers, risultando più efficiente dell'esecuzione con 10 workers e meno costosa in termini di utilizzo delle risorse di quella con 15 workers, poiché ne sfrutta solo 12.

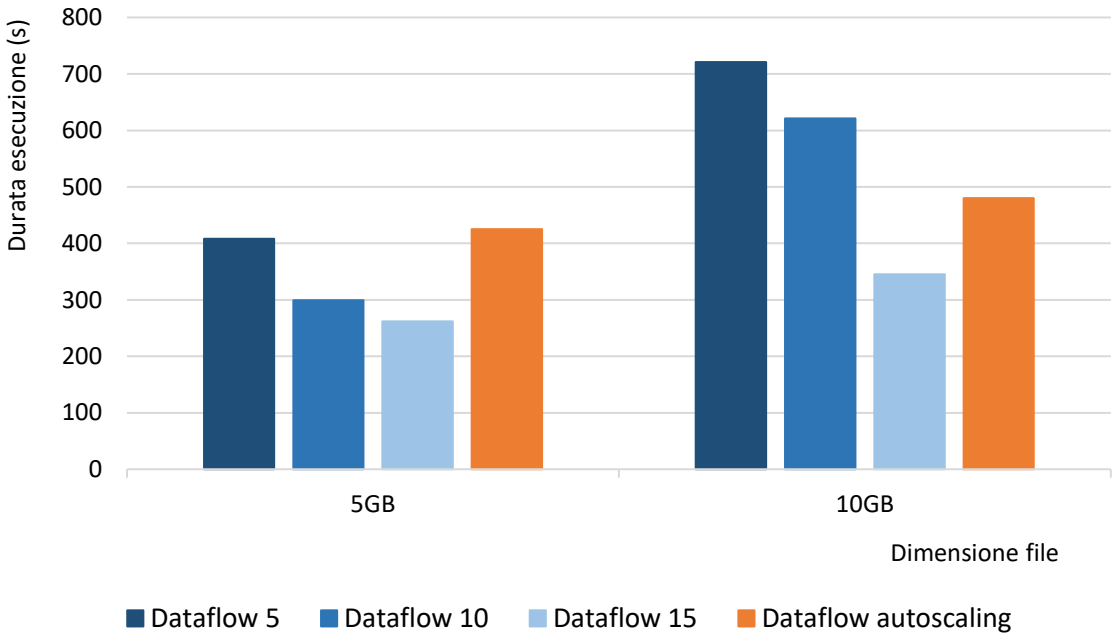

Figura 11: *Grafico che mostra il confronto tra i tempi di esecuzione di Cloud Dataflow con file di 5GB e 10GB al variare del numero dei workers utilizzati e con l'autoscaling*

5.4 Conclusioni

Il modello di programmazione MapReduce, di cui Hadoop ne è la realizzazione open source più famosa, è stato per anni lo strumento più utilizzato per eseguire computazioni sui BigData. Tuttavia, con l'introduzione di elaborazioni sempre più complesse, il modello MapReduce ha mostrato alcuni punti deboli, come la sua lentezza nell'eseguire più cicli di operazioni sugli stessi dati e un utilizzo troppo dispendioso della memoria. Questi sono solo alcuni dei motivi per cui si stanno cercando nuovi metodi di approccio al calcolo distribuito che devono riuscire a soddisfare le sempre crescenti esigenze.

Google ha già innovato la sua struttura di calcolo con il suo Google Cloud Dataflow, destinato a rimpiazzare quasi del tutto MapReduce. Sul fronte open source invece, come abbiamo visto è presente già da alcuni anni il framework Apache Spark che promette di migliorare e velocizzare enormemente le prestazioni di Hadoop.

Dataflow è l'unico tra i sistemi di elaborazione dati in parallelo che si basa su un modello di elaborazione *out-of-order*, cioè esegue istruzioni in un ordine governato dalla disponibilità dei dati in ingresso, piuttosto che dall'ordine originale del programma, ed è quindi progettato per rispondere alla sfida del trattamento dei dati in tempo reale, senza però comprometterne la correttezza. Inoltre detiene vantaggi in termini di flessibilità del modello di programmazione, di potenza ed espressività.

Come abbiamo potuto capire dai test effettuati uno dei punti di forza della piattaforma Dataflow è sicuramente l'autoscaling dinamico [12] effettuato automaticamente dal servizio senza bisogno di interventi da parte dell'utente. Utilizzando modelli di programmazione come MapReduce o sistemi come Hadoop e Spark, risulta sempre complessa e dispendiosa la messa a punto del job che si vuole eseguire, il che comprende anche il dover impostare il numero di workers e di task da utilizzare nel cluster. Inoltre una singola configurazione non è spesso la risposta corretta ad ogni situazione, poiché le risorse hanno bisogno di cambiare dinamicamente durante tutta la durata del job.

Sia Spark che Cloud Dataflow hanno i mezzi per fornire automaticamente le risorse per l'esecuzione di un job, ma si comportano in maniera diversa.

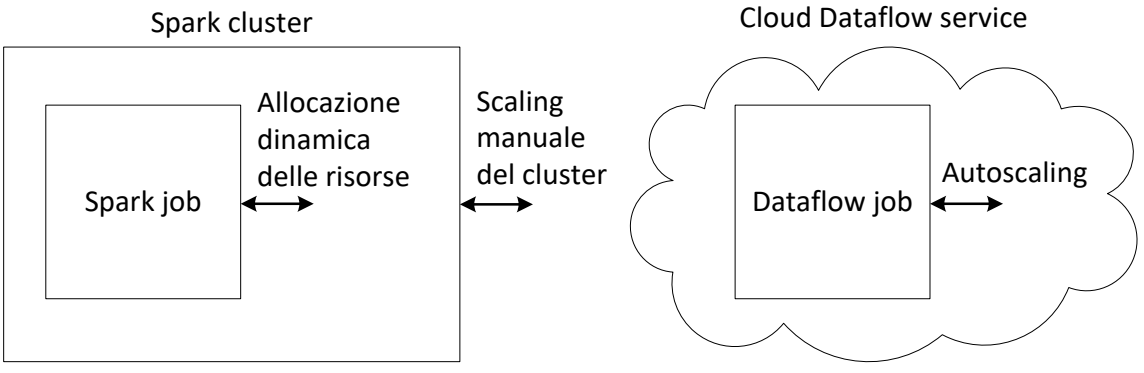

Figura 12: *Spark vs Cloud Dataflow autoscaling*

Con Spark, gli utenti devono prima implementare il cluster e poi distribuire il job che vogliono eseguire. Questo però crea limiti di scalabilità. Spark infatti offre servizi di allocazione

dinamica delle risorse che permettono di allocare per l'esecuzione di un job altri workers disponibili in un cluster (per cui viene definito un numero minimo e massimo di workers). Però non fornisce un metodo per cambiare dinamicamente anche il numero dei tasks, quindi i processi già in esecuzione potrebbero non beneficiarne.

Dall'altro lato Cloud Dataflow è stato progettato per essere eseguito nel cloud, con già in mente l'idea dell'autoscaling, che non solo aggiusta il numero dei workers, ma regola anche il numero dei tasks in modo tale da avere i workers sempre occupati.

In sintesi, con Spark gli utenti devono configurare, coordinare e monitorare sia l'allocazione delle risorse che il dimensionamento del cluster, e questo comporta alcuni limiti di scalabilità. Invece, con Cloud Dataflow, che è stato costruito appositamente per essere eseguito nel cloud, il servizio è in grado di scalare automaticamente al posto dell'utente.

Infine, un'altra questione che bisogna tenere a mente è che Dataflow è un servizio completamente gestito, che è stato progettato per semplificare i meccanismi di elaborazione dati su larga scala. Questo permette di concentrarsi sulla composizione logica del processo di elaborazione e garantisce completa trasparenza rispetto alla configurazione del cluster. È possibile quindi concentrarsi su ciò che è necessario che faccia il processo, invece di come esattamente deve essere eseguito, facilitando l'utilizzo anche agli utenti inesperti, i quali potranno lasciare alla piattaforma il compito di gestire le risorse e l'esecuzione a basso livello.

6 Codice

Di seguito ho inserito il codice completo dei programmi di WordCount utilizzati nelle due differenti piattaforme per l'esecuzione e l'analisi di grandi quantità di dati.

6.1 WordCount per la piattaforma Cloud Dataflow

```
public class WordCount {

  // ExtractWordFn: divide le linee di testo in singole parole
  static class ExtractWordsFn extends DoFn<String, String> {
       private static final long serialVersionUID = 1L;
       private final Aggregator<Long, Long> emptyLines =
                   createAggregator("emptyLines",new Sum.SumLongFn());

       @Override
       public void processElement(ProcessContext c) {
             if (c.element().trim().isEmpty()) {
                   emptyLines.addValue(1L);
             }
             String[] words = c.element().split("[^a-zA-Z']+");
             for (String word : words) {
                   if (!word.isEmpty()) {
                         c.output(word);
                   }
             }
       }
  } // ExtractsWordFn


  // FormatAsTextFn: converte una parola e il suo contatore in una
  // stringa stampabile
  public static class FormatAsTextFn extends
                                    DoFn<KV<String, Long>, String> {
       private static final long serialVersionUID = 1L;

       @Override
       public void processElement(ProcessContext c) {
             c.output(c.element().getKey() + ": " +
                                    c.element().getValue());
       }
  } // FormatAsTextFn


  // CountWords: converte una PCollection contenente linee di testo
  // in una PCollection di singole parole
  public static class CountWords extends
       PTransform<PCollection<String>, PCollection<KV<String, Long>>>{
       private static final long serialVersionUID = 1L;

       @Override
       public PCollection<KV<String, Long>>
                                    apply(PCollection<String> lines) {
```

```
            PCollection<String> words = lines.apply(ParDo
                                        .of(new ExtractWordsFn()));
            PCollection<KV<String, Long>> wordCounts = words
                                    .apply(Count.<String>perElement());
            return wordCounts;
        }
    } // CountWords


    // Main del programma di WordCount
    // Prima configura i parametri della pipeline e poi la esegue
    public static void main(String[] args) {
        DataflowPipelineOptions options = PipelineOptionsFactory
                                .as(DataflowPipelineOptions.class);

        options.setProject("mapreduce-apps-in-hybrid-cloud");
        options.setStagingLocation("gs://mapreduce-apps-in-hybrid-
                                                cloud-bucket/staging");
        options.setRunner(BlockingDataflowPipelineRunner.class);

        // Eventualmente si possono aggiungere anche le seguenti righe
        // per impostare il numero di workers da utilizzare
        // options.setNumWorkers(10);
        // options.setMaxNumWorkers(15);

        Pipeline p = Pipeline.create(options);

        p.apply(TextIO.Read.named("Read").from("gs://mapreduce-apps-in-
                                    hybrid-cloud-bucket/input/*"))
         .apply(new CountWords())
         .apply(ParDo.of(new FormatAsTextFn()))
         .apply(TextIO.Write.named("Write").to("gs://mapreduce-apps-in-
                                    hybrid-cloud-bucket/output.txt"));
        p.run();
    } // Main
} // WordCount
```

6.2 WordCount per la piattaforma Hadoop

```
public class WordCount {

    // Mapper: mappa le coppie chiave/valore ricevute in ingresso in
    // un insieme di coppie chiave/valore intermedi
    public static class TokenizerMapper
                    extends Mapper<Object, Text, Text, IntWritable>{
        private final static IntWritable one = new IntWritable(1);
        private Text word = new Text();

        // funzione map
        public void map(Object key, Text value, Context context)
                        throws IOException, InterruptedException {
            StringTokenizer itr = new
                            StringTokenizer(value.toString());
            while (itr.hasMoreTokens()) {
                word.set(itr.nextToken());
                context.write(word, one);
            }
        }
```

```
  } // Mapper

  // Reducer: riduce un insieme di valori intermedi che condividono
  // una chiave in un insieme più piccolo
  public static class IntSumReducer
           extends Reducer<Text,IntWritable,Text,IntWritable> {
     private IntWritable result = new IntWritable();

     // funzione reduce
     public void reduce(Text key, Iterable<IntWritable> values,
         Context context) throws IOException, InterruptedException {
       int sum = 0;
       for (IntWritable val : values) {
             sum += val.get();
       }
       result.set(sum);
       context.write(key, result);
     }
  } // Reducer


  // Main del programma di WordCount
  public static void main(String[] args) throws Exception {
     Configuration conf = new Configuration();
     String[] otherArgs = new GenericOptionsParser(conf, args)
                                                .getRemainingArgs();
     if (otherArgs.length < 2) {
       System.err.println("Usage:
                          wordcount <in> [<in>...] <out>");
       System.exit(2);
     }

     Job job = Job.getInstance(conf, "word count");

     job.setJarByClass(WordCount.class);
     job.setMapperClass(TokenizerMapper.class);
     job.setCombinerClass(IntSumReducer.class);
     job.setReducerClass(IntSumReducer.class);
     job.setOutputKeyClass(Text.class);
     job.setOutputValueClass(IntWritable.class);

     for (int i = 0; i < otherArgs.length - 1; ++i) {
       FileInputFormat.addInputPath(job,
                                    new Path(otherArgs[i]));
     }
     FileOutputFormat.setOutputPath(job,
                      new Path(otherArgs[otherArgs.length - 1]));
     System.exit(job.waitForCompletion(true) ? 0 : 1);
  } // Main
} // WordCount
```

6.3 WordCount per la piattaforma Spark

```
public final class JavaWordCount {
  private static final Pattern SPACE = Pattern.compile(" ");

  // Main del programma di WordCount
```

```
  public static void main(String[] args) throws Exception {
        if (args.length < 1) {
              System.err.println("Usage: JavaWordCount <file>");
              System.exit(1);
        }

        SparkSession spark = SparkSession.builder()
              .appName("JavaWordCount")
              .getOrCreate();

        // divide i file di testo passati come parametro di
        // invocazione in singole linee
        JavaRDD<String> lines = spark.read()
                                     .textFile(args[0]).javaRDD();

        // divide le linee di testo in singole parole
        JavaRDD<String> words = lines.flatMap(new
                               FlatMapFunction<String, String>() {
              @Override
              public Iterator<String> call(String s) {
                    return Arrays.asList(SPACE.split(s)).iterator();
              }
        }); // lines.flatMap

        // associa ad ogni singola parola un contatore di occorrenze
        JavaPairRDD<String, Integer> ones = words.mapToPair(
                         new PairFunction<String, String, Integer>() {
              @Override
              public Tuple2<String, Integer> call(String s) {
                    return new Tuple2<>(s, 1);
              }
        }); // words.mapToPair

        // conta le occorrenze di ogni singola parola
        JavaPairRDD<String, Integer> counts = ones.reduceByKey(
                         new Function2<Integer, Integer, Integer>() {
              @Override
              public Integer call(Integer i1, Integer i2) {
                    return i1 + i2;
              }
        }); // ones.reduceByKey

        List<Tuple2<String, Integer>> output = counts.collect();
        for (Tuple2<?,?> tuple : output) {
              System.out.println(tuple._1() + ": " + tuple._2());
        }

        spark.stop();
  } // Main
} // JavaWordCount
```

Bibliografia

[1] AA. VV, **Above the Clouds: A Berkeley View of Cloud Computing**
UC Berkeley Reliable Adaptive Distributed Systems Laboratory, febbraio 2009
http://www.eecs.berkeley.edu/Pubs/TechRpts/2009/EECS-2009-28.pdf

[2] Peter Mell e Timothy Grance, **The NIST Definition of Cloud Computing**
The National Institute of Standards and Technology, settembre 2011,
http://nvlpubs.nist.gov/nistpubs/Legacy/SP/nistspecialpublication800-145.pdf

[3] AA. VV, **A View of Cloud Computing, Communications of the ACM, Vol. 53 No. 4**
Association for Computing Machinery, aprile 2010
http://cacm.acm.org/magazines/2010/4/81493-a-view-of-cloud-computing/pdf

[4] Jeffrey Dean e Sanjay Ghemawat, **MapReduce: Simplified Data Processing on Large Clusters**, Google Inc., dicembre 2004,
http://research.google.com/archive/mapreduce-osdi04.pdf

[5] The Apache Software Foundation, **Apache Hadoop 2.7.2 Documentation**
The Apache Software Foundation, novembre 2016
http://hadoop.apache.org/docs/stable/

[6] Apache Spark, **Apache Spark 2.0.2 Documentation**
Apache Spark, novembre 2016
http://spark.apache.org/docs/latest/

[7] Google Inc., **Google Cloud Platform – Cloud Dataflow Documentation**
Google Inc., settembre 2016
https://cloud.google.com/dataflow/docs/

[8] Google Inc., **Google Cloud Platform**
Google Cloud Platform., dicembre 2016
https://cloud.google.com/

[9] Google Inc., **Google Cloud Platform – Compute Engine**
Google Cloud Platform, dicembre 2016
https://cloud.google.com/compute/

[10] Google Inc., **Google Cloud Platform – Cloud Storage**
Google Cloud Platform, dicembre 2016
https://cloud.google.com/storage/

[11] Google Inc., **Google Cloud Platform – Cloud Dataproc**
Google Cloud Platform, dicembre 2016
https://cloud.google.com/dataproc/

[12] Google Inc. e Eric Anderson, **Comparing Cloud Dataflow autoscaling to Spark and Hadoop**, Google Cloud Platform, marzo 2016
https://cloud.google.com/blog/big-data/2016/03/comparing-cloud-dataflow-autoscaling-to-spark-and-hadoop

[13] Apache Beam, **Apache beam (Cloud Dataflow) SDKs**
Apache Beam, dicembre 2016
https://github.com/apache/incubator-beam