POLITECNICO DI TORINO

Corso di Laurea in Ingegneria Informatica

Tesi Magistrale

Progettazione e sviluppo di applicazioni per la gestione di flussi di dati IoT tramite un'architettura Big Data nel Cloud

Relatori:
Prof. Paolo Garza

Candidata:
Mariagrazia Cardile

Tutor Aziendale:
Ing. Marco Gatta

Anno Accademico 2017/2018

Indice

Capitolo 1

INTRODUZIONE

Questa Tesi si basa sul lavoro svolto durante un periodo semestrale di Stage, presso l'azienda *Data Reply*, facente parte del gruppo Reply, che pone il suo focus su Big Data e Data Analytics.

Il caso documentato è quello di uno dei principali clienti dell'azienda, che opera in ambito automotive: la richiesta era quella di ricevere, processare, analizzare, flussi di dati provenienti dalle box telematiche installate sui veicoli. Al mio arrivo era presente già una prima soluzione, sviluppata su infrastruttura On-Premise, che gestiva i dati in arrivo da veicoli pesanti. Il cliente, tuttavia, ha realizzato nuove board che saranno presenti a partire dal prossimo anno su un numero di veicoli (pesanti e leggeri) almeno 10 volte maggiore rispetto al passato e che sono anche in grado di inviare un numero superiore di informazioni. La soluzione sviluppata, stavolta basata sul Cloud per garantire scalabilità e superare i problemi dell'On-Premise, mira quindi sia a gestire interamente su Cloud i nuovi dati per mezzo di applicazioni real-time e batch a seconda del caso d'uso, sia a portare sul Cloud le informazioni provenienti dalle vecchie box, che col tempo tenderanno a scemare naturalmente.

In queste pagine vedremo quindi come si è riusciti ad innovare soluzioni e architetture precedenti, in parte adattandole a nuovi ambienti che rispondono meglio alle esigenze del cliente, e in parte studiando, progettando e implementando nuovi sistemi, con l'ausilio delle tecnologie più adatte. Il mondo dei Big Data, infatti, offre un continuo rinnovarsi di piattaforme, componenti e

in generale, strumenti per gestire, utilizzando approcci differenti, l'enorme quantità di informazioni a disposizione: negli ultimi due anni, i dati prodotti nel mondo sono quasi raddoppiati e le sorgenti di informazioni sono in continuo aumento ed evoluzione. Tutto ciò porta all'urgenza di avere a disposizione soluzioni in grado di gestire queste enormi moli di dati, che riescano anche a garantire alte prestazioni anche in caso di variazioni di carico.
In ambito tecnologico, e nella fattispecie nell'ambiente Big Data, è fondamentale avere un *know-how* in costante crescita, cioè scoprire, studiare e saper utilizzare i nuovi strumenti messi a disposizione dal mercato, in modo da poter progettare e sviluppare soluzioni che siano all'avanguardia e che possano soddisfare pienamente le richieste dei clienti.

Durante la fase di studio delle soluzioni che verranno descritte, sono stati analizzati e presi in considerazione molti software disponibili sul mercato dei Big Data, cercando quelli che rispondessero meglio ai requisiti dell'utente.
Come accade molto spesso nell'ambito della gestione e dell'analisi dei Big Data, è stato sfruttato il framework Apache Hadoop, con una parte degli strumenti basati su esso, ma anche numerosi servizi forniti dalla piattaforma cloud Microsoft Azure, grazie ai quali è stato possibile gestire flussi di dati real-time e batch con l'ausilio degli strumenti più appropriati.

In questo documento quindi verrà descritto il processo complessivo di transizione da cluster On-Premise a Cloud, scendendo nel dettaglio implementativo di alcune soluzioni per la gestione di nuove macro-categorie di flussi di dati. Ogni implementazione è stata prima testata su ambienti di test, al fine di assicurarsi del corretto funzionamento, eventualmente corretta e poi rilasciata in ambiente di certificazione.

1.1 Struttura della Tesi

Questa Tesi è strutturata in sei capitoli, incluso quello introduttivo. Il secondo capitolo si focalizza sull'analisi dell'ambiente di partenza, cioè l'On-Premise, e quello di arrivo, il Cloud, dell'intero

processo. In particolare verranno messe in evidenza le caratteristiche principali dei due, per capirne al meglio l'utilizzo. A questo segue un focus sull'infrastruttura Hadoop, un approfondimento per fornire al lettore informazioni sugli strumenti utilizzati durante l'intero processo. Secondo e terzo capitolo sono quindi preupedetici al quarto, in cui è mostrato l'*as-is*, cioè la situazione presente prima del lavoro qui descritto, e in cui vengono spiegate le ragioni che hanno reso necessario il cambio di ambiente. Il capitolo cinque costituisce il cuore dell'intero lavoro: in esso vengono descritti i processi sviluppati sia da un punto di vista globale, sia entrando nel dettaglio delle singole applicazioni. Infine, il sesto capitolo tira le somme e introduce nuovi possibili scenari di sviluppo.

1.2 I Big Data

Negli ultimi anni, il volume e la complessità delle informazioni ha subito un incremento esponenziale e questo trend viene confermato costantemente. Uno studio fornito dall'International Data Corporation (IDC), condotto dagli studiosi John Gantz e David Reinsel [1], dimostra proprio come la quantità di dati, in particolar modo di tipo non strutturato, emessi tenda a raddoppiare ogni due anni, per cui è previsto che entro il 2020 questa raggiunga e superi quota 40000 Exabytes (1 ZB = 2^{18} Byte), grazie al continuo aumento delle sorgenti in grado di emetterli .

Insieme alla crescita complessiva dei dati, anche le varie tecnologie hanno fatto grossi passi in avanti in termini di storage e di capacità computazionale; tuttavia, i sistemi di gestione dei dati tradizionali non sono in grado di supportare i flussi di informazioni che vengono da tutte le moderne sorgenti (transazioni online, social network, portali, motori di ricerca, reti di sensori,...). Proprio questo aspetto costituisce uno dei rischi più concreti del mondo Big Data, cioè la possibilità che questi sistemi non riescano a supportare le necessità di un mondo sempre più "connesso", in cui chi ha le maggiori disponibilità di dati, corredata dalla maggiore velocità di accesso alle informazioni, ha maggiori probabilità di successo nei relativi settori.

Chi vuole descrivere i Big Data, tipicamente non si appella ad una definizione formale, ma

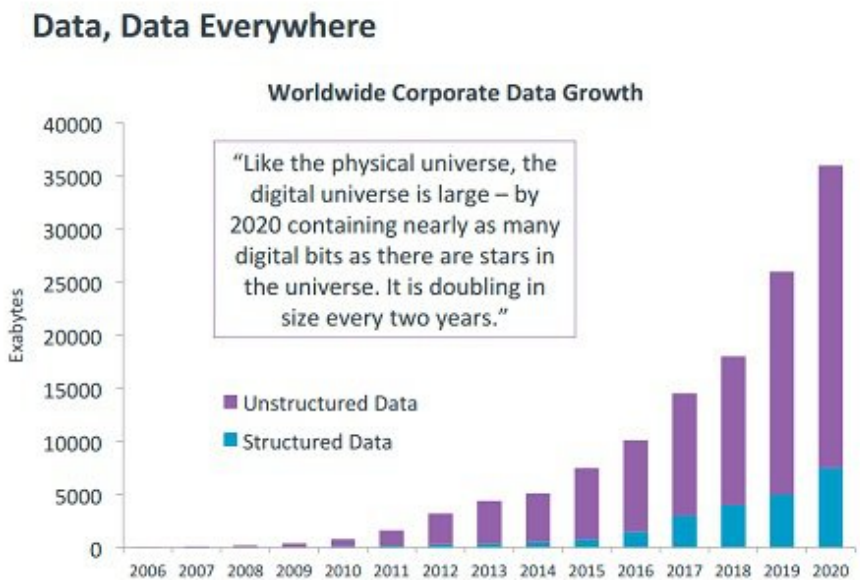

Figura 1.1: Crescita del volume di dati fra il 2010 e il 2020

lo fa attraverso un modello chiamato *Big Data 3 Vs*, da poco diventato delle *5V*, che è sintetizzato nella figura 1.2. Vediamo quali sono le caratteristiche fondamentali che determinano questa nomenclatura:

Volume: questo parametro vuole sottolineare l'ingente quantità di dati prodotta in ogni istante, dell'ordine non di Tera ma di ZettaByte o addirittura BrontoByte. Per avere un'idea, è sufficiente pensare a quante foto, video, email, post su Social Network o anche segnali provenienti dai sensori IoT vengano prodotti e condivisi in ogni secondo: negli ultimi due anni siamo arrivati a produrre il 90% della quantità totale di dati dall'inizio dei tempi. Una delle conseguenze di questo incremento è che i database tradizionali non riescono a soddisfare le esigenze di memorizzazione e analisi di data set sempre più grandi. Le nuove tecnologie Big Data dunque, permettono di archiviare e poi utilizzare questi set, tramite l'ausilio di sistemi distribuiti, in cui i dati sono tipicamente splittati, posti in sedi diverse e poi rimessi insieme via software.

Velocità: nel mondo Big Data, uno dei punti focali è la rapidità con cui le informazioni vengono prodotte, mosse e analizzate. Anche per chi non opera in questo campo, ciò risulta evidente:

si pensi all'importanza dei controlli sulle transazioni finanziarie, atti a rilevare in tempo reale eventuali frodi, o alla velocità con cui un messaggio pubblicato su un social network possa diventare virale. Gli strumenti messi a disposizione dal mondo Big Data ci permettono ormai di processare e analizzare i dati non appena questi vengono prodotti, in alcuni casi anche senza doverli memorizzare su un database. É evidente dunque quanto sia importante tenere tempi di risposta bassi, cioè reagire tempestivamente agli input che provengono dalle varie sorgenti di dati.

Varietà: come abbiamo visto nella figura 1.1, la maggior parte dei dati a disposizione è di tipo non strutturato. Una delle caratteristiche peculiari dei Big Data è, appunto, la varietà dei formati con cui possiamo avere a che fare. In passato, l'unico modello di dati archiviabili e memorizzabili su tabelle, era quello strutturato, che prevedeva tipicamente l'inserimento in Database relazionali. Oggi si stima che il 90% dei dati sia per sua natura non strutturato, essendo inclusi anche documenti di vario genere, come PDF, csv, file testuali, per cui occorre svincolarsi dagli strumenti tradizionali se si ha l'obiettivo di analizzarli, aggregarli e rappresentarli. La varietà dei Big Data si spiega anche nelle fonti: le informazioni possono provenire da miriadi di sorgenti diversi, da sensori o direttamente dagli utenti, dai log dei sistemi di rilevamento delle intrusioni a quelli relativi al traffico che attraversa i router.

Negli ultimi anni, al modello delle tre *V*, sono stati aggiunti altri due elementi:

Veridicità: l'aumento costante di sorgenti e della mole di dati è direttamente proporzionale alla crescita della difficoltà di garantirne accuratezza e in generale qualità. Per veridicità infatti, si intende proprio l'affidabilità delle informazioni a nostra disposizione. Ad esempio, non è compito facile provare a ricavare informazioni utili dai post su social network come Twitter, che prevede hashtag costituiti da parole concatenate, o interpretare le svariate sfumature del linguaggio naturale, o ancora discriminare l'affidabilità dei contenuti. Gli alti volumi di dati in qualche modo bilanciano possibili problemi di scarsa affidabilità delle informazioni, ma è chiaro come la veridicità dei dati incida sulle analisi fatte basandosi su questi.

Valore: è probabilmente una delle più importanti caratteristiche. Avere accesso alle tecnologie Big Data è un ottimo punto di partenza, ma diventa inutile se non siamo effettivamente in grado di ottenere un valore dai dati.

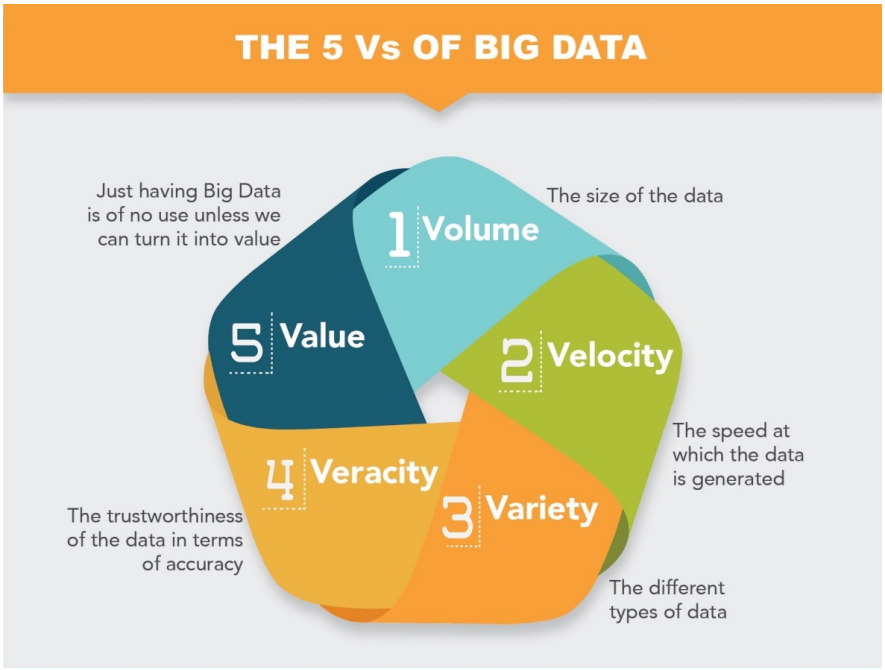

Figura 1.2: Le 5V dei Big Data

In ambito industriale, la definizione più comune di Big Data, è quella fornita proprio dalla Gartner qualche anno fa, società leader nel campo dell'Information Technology:

I Big Data sono risorse informative caratterizzate da alto volume, alta velocità e/o alta varietà, che richiedono forme efficaci e innovative di elaborazione delle informazioni. Esse permettono di ottenere informazioni approfondite, di prendere decisioni e di automatizzare i processi.

Anche per i non addetti ai lavori è facile comprendere dunque come il mondo dei Big Data sia destinato a destare interesse in un numero sempre maggiore di settori e ad espandersi ancora, così come crescerà la necessità di avere a disposizione sempre più dati, di saperli ricevere da fonti

disparate in modo veloce, di processarli ed emettere risultati in tempi ridotti.

Capitolo 2

ANALISI DELLE INFRASTRUTTURE

Lo scopo di questo secondo capitolo è quello di introdurre le principali infrastrutture esistenti in ambito Big Data, l'On-Premise e il Cloud, per poi evidenziarne similitudini e differenze, nonché i principali vantaggi e svantaggi. Le principali feature delle due architetture renderanno in seguito più chiare le scelte infrastrutturali decise dall'Azienda in cui ho svolto il mio stage.

2.1 On-Premise

La prima tipologia di architettura che viene analizzata in questo paragrafo è anche quella maggiormente utilizzata fino alla prima metà degli anni 2000. Il termine "On-Premise", letteralmente "nello stabile", indica un tipo di infrastruttura in cui l'hardware e/o il software sono gestiti internamente dall'azienda che li utilizza. Chi sceglie questa soluzione deve quindi avere il proprio server, ospitato in una sede aziendale, o in un data center, in cui è possibile affittare macchine per i propri scopi.

2.1.1 Caratteristiche principali

Per valutare le due soluzioni contrapposte, tipicamente si prendono in considerazione quattro fattori, cioè costi, sicurezza dei dati, gestione e manutenzione dei sistemi e scalabilità. Analizziamo

dunque il comportamento dell'On-Premise in base ad essi:

Costi: in ambito aziendale è uno dei fattori più importanti. La gestione dei Big Data su cluster On-Premise richiede alle aziende l'installazione di una costosa infrastruttura, in grado di ricevere, immagazzinare e analizzare i dati. Trattandosi di una piattaforma fisica che richiede un numero importante di server, è richiesto di conseguenza innanzitutto spazio per farli funzionare, ma anche una grande quantità di elettricità. É per questa ragione che in passato molte piccole aziende non erano in grado di implementare soluzioni Big Data, proprio per gli eccessivi costi iniziali.

Sicurezza dei dati: costituisce uno dei maggiori vantaggi di una soluzione On-Premise. Avere dati immagazzinati localmente permette un accesso più immediato e un monitoraggio diretto.

Gestione e manutenzione dei sistemi: si tratta del punto debole più evidente dell'On-Premise. Adottare una soluzione del genere significa dover affrontare direttamente il tema, sempre molto delicato, della manutenzione. É necessario dunque, per le aziende, avere un team dedicato che assicuri costantemente il corretto funzionamento dei sistemi.

Scalabilità: è uno dei nodi cruciali in ambito Big Data. Si tratta sostanzialmente della flessibilità nell'aumentare o diminuire la capacità di raccolta di dati in base alle necessità. In ambito On-Premise, non è banale scalare ed è piuttosto comune trovarsi di fronte a situazioni di *under-provisioning*, cioè casi in cui si avrebbe bisogno di installare infrastrutture maggiormente performanti per gestire grosse quantità di dati, e di *over-provisioning*, cioè situazioni in cui non vi sono grosse moli di dati da gestire e di conseguenza l'infrastruttura viene sotto-utilizzata.

Focalizzandosi sulla parte sinistra dell'immagine 2.1, è evidente come in ambito On-Premise i costi maggiori siano dovuti ai punti appena elencati.

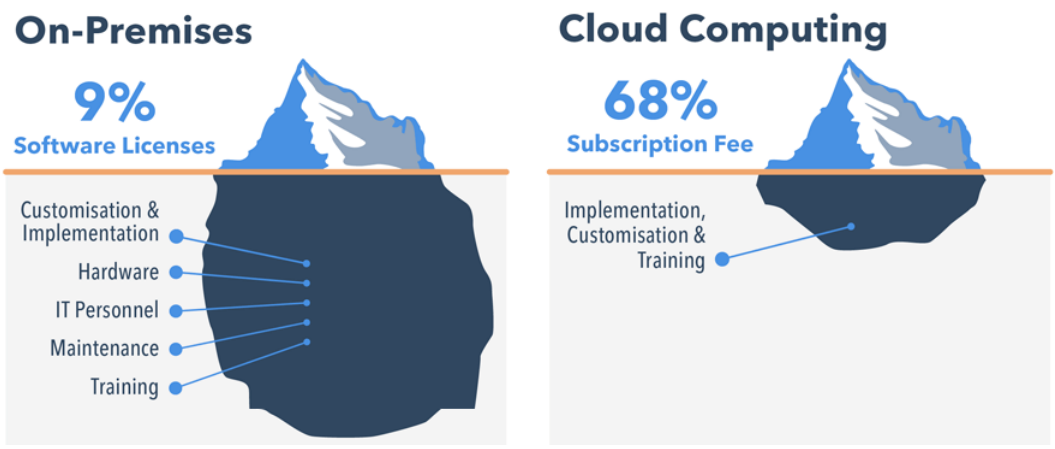

Figura 2.1: Analisi dei costi fra On-Premise e Cloud

2.2 Cloud Computing

Per Cloud Computing si intende la distribuzione di servizi di calcolo, come server, risorse di archiviazione, database, rete, software, analisi e molto altro tramite Internet (”il cloud”). Si parla cioè di una serie di tecnologie che permettono di elaborare i dati sfruttando risorse distribuite sulla rete. I fornitori di tali servizi sono detti *provider* di servizi cloud, tipicamente addebitano all’utente un costo proporzionale all’utilizzo dei servizi stessi e gestiscono in maniera trasparente all’utente, che non deve così preoccuparsi di installare fisicamente qualcosa, tutte le infrastrutture offerte. In ambito aziendale la scelta di servizi Cloud è sempre più popolare e adottata: un sondaggio del 2017 mostra come le aziende interrogate abbiano scelto di gestire il 79% del loro carico di lavoro proprio sul cloud [2].

2.2.1 Caratteristiche Principali

Scendiamo ora maggiormente nel dettaglio, analizzando le principali motivazioni che spingono le aziende ad adottare soluzioni basate sul Cloud:

Costi: si tratta di uno dei maggiori benefici portati da questo tipo di soluzione. I costi, infatti,

sono in continuo abbattimento, grazie al fenomeno dell' *economies of scale* [1]. Il cloud offre risparmi sulle infrastrutture (server, apparati, energia,...) e sulla modalità di gestione e aggregazione della domanda, in quanto, aumentando la percentuale di utilizzo dei server si attenua la variabilità globale dovuta a picchi di lavoro. Inoltre porta ad una maggiore efficienza del *multi-tenancy*, poiché riduce i costi di gestione di applicazione e server per ogni tentant. Il risultato è che più aziende adotteranno soluzioni cloud, meno esso costerà, mentre il costo dell'On-Premise rimarrà pressocché invariato.

Oltre a beneficiare dell'economies of scale, chi sceglie il cloud lo fa anche perché non ci sono solitamente pagamenti in anticipo, ma si paga solamente ciò che si usa. Questo concetto è radicalmente diverso da ciò che accade On-Premise, in cui si investe inizialmente nella creazione del cluster e sono necessari costi successivi per il suo mantenimento e/o potenziamento, anche se poi non viene sfruttato completamente.

Scalabilità: sotto questo aspetto, il cloud riduce in maniera sostanziale il problema del dimensionamento dell'ambiente Big Data. Sfruttando l'elasticità del cloud, non è più necessario l'overprovisioning per poter far fronte a picchi di carico sulle macchine. La possibilità di scalare le risorse permette di avere componenti su misura, in base ai task da eseguire, senza rischiare di non averne a sufficienza, oppure di non sfruttarle pienamente. É l'ambiente che dunque si adatta al task, e non più il task che si adatta all'ambiente.

Velocità: un altro aspetto cruciale da considerare è la proprio la rapidità con cui è possibile creare un'infrastruttura Big Data. Se On-Premise le macchine vanno configurate, i servizi installati, in Cloud è possibile rilasciare un nuovo ambiente in pochi minuti. Il pattern *Infrastructure As Code* viene incontro a questa esigenza: la creazione di un'infrastruttura viene definita come se fosse un codice software.

Produttività: gli aspetti sopra elencati si tramutano in un risparmio per le aziende, che non hanno più bisogno di caricarsi di spese di creazione e mantenimento, potenziamento e otti-

[1] Le economie di scala si riferiscono a costi ridotti per unità derivanti dall'aumento della produzione totale di un prodotto. Ad esempio, uno stabilimento più grande produrrà utensili elettrici a prezzo unitario più basso.

mizzazione di datacenter privati. Ciò consente loro di poter reinvestire il budget risparmiato in nuove attività Big Data, che possono così generare guadagni.

Affidabilità: il Cloud riduce i costi di backup dei dati e aumenta la facilità di ripristino in caso di emergenza o guasto, grazie alla possibilità di replicare i dati all'interno della rete del provider.

2.3 Tipologie di servizi Cloud

Il Cloud Computing è un settore in grossa espansione e, per comprenderlo meglio, è utile distinguere le tipologie principali di servizi:

IaaS (Infrastructure as a Service): è la tipologia basilare dei servizi cloud. Essa prevede che venga affittata l'infrastruttura IT, cioè server, macchine virtuali, risorse di archiviazioni, reti e sistemi da un provider, con la logica del pagamento in base al consumo.

PaaS (Platform as a Service) : si tratta di una vera e propria piattaforma che costituisce un ambiente on demand per sviluppare, testare, distribuire e gestire software. L'utente, in questo caso, dovrà solo preoccuparsi di sviluppare le applicazioni, in quanto i problemi legati alla creazione della piattaforma sono demandati al provider.

SaaS (Software as a Service): è il servizio più ad alto livello offerto all'utente, grazie al quale i provider ospitano e gestiscono l'applicativo software e l'infrastruttura, occupandosi del rilascio di patch e aggiornamenti. All'utente finale basta connettersi all'applicazione tramite Internet per usufruire del servizio.

L'immagine in figura 2.2 riassume le diverse responsabilità che l'utente deve assumersi nel caso in cui scelga una soluzione On-Premise o una delle soluzioni Cloud fin qui analizzate.

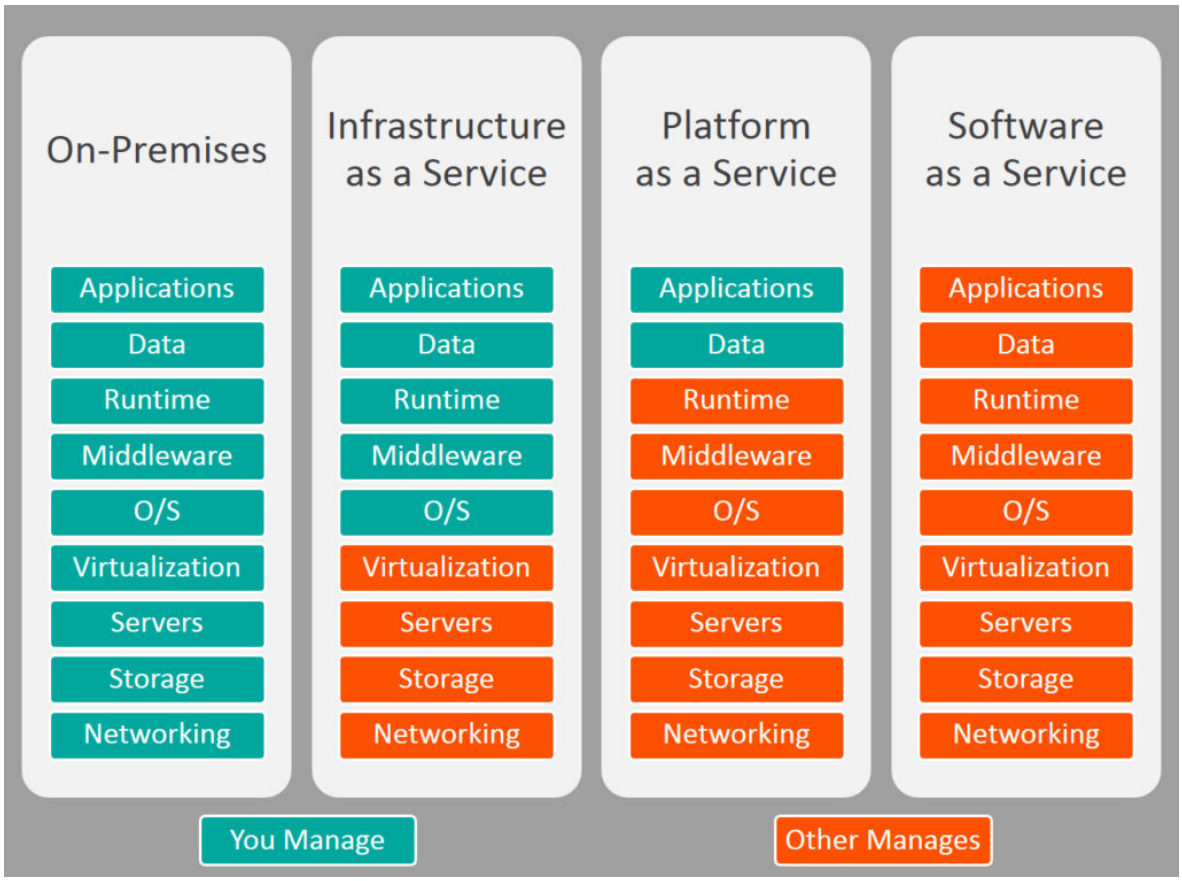

Figura 2.2: Confronto fra modelli On-Premise e Cloud

Oltre alla distinzione fra IaaS, PaaS e SaaS, occorre menzionare anche le differenti modalità con cui le risorse cloud possono essere fornite:

Cloud Pubblico: in questo caso le risorse fanno parte di un'infrastruttura condivisa e appartengono al provider che le distribuisce tramite Internet. Questo modello permette quindi la possibilità di scalare le risorse e di pagare soltanto quelle effettivamente consumate. Inoltre risulta particolarmente semplice eseguire la fase di deploy.

Cloud Privato: con questo termine ci si riferisce invece alle risorse possedute ed utilizzate esclusivamente da un privato, azienda o organizzazione. Queste possono scegliere se ospitare fisicamente il data center, o affittarne uno presso un provider. Il vantaggio principale è

che è possibile avere un maggiore controllo sui dati, ma, d'altro canto, scalare diventa più complicato e i costi sono più alti.

Cloud Ibrido: questa soluzione cerca di sfruttare i vantaggi di entrambe le soluzioni, prendendo dal cloud privato la possibilità di controllare i dati più critici, ma offrendo al contempo la possibilità di scalare in modo più flessibile.

Capitolo 3

L'ECOSISTEMA HADOOP

Lo scopo di questa appendice è quello di introdurre gli aspetti principali su cui si basano gli strumenti utilizzati durante l'intero processo di sviluppo, con particolare attenzione al framework Hadoop, al fine di fornire al lettore le conoscenze necessarie per capirne pienamente il funzionamento.

3.1 Il Framework Hadoop

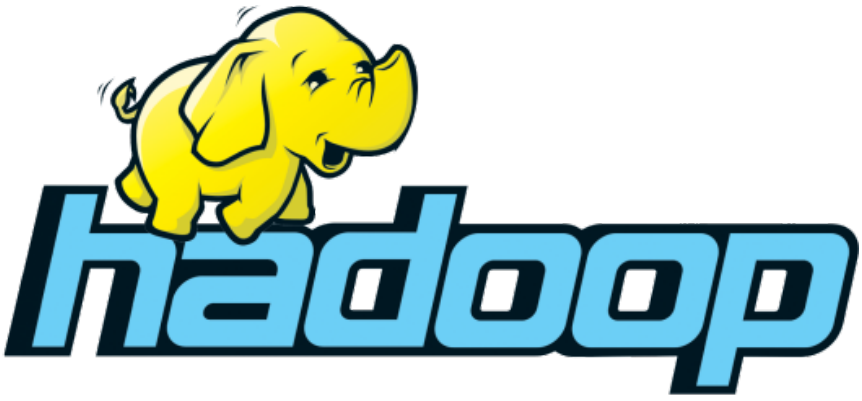

Figura 3.1: Logo Hadoop

Apache Hadoop è un framework open source, che permette di elaborare grandi set di dati, grazie alla capacità di sfruttare la potenza di calcolo e lo spazio di archiviazione di computer facenti parte di cluster, proponendo un modello di programmazione molto semplice. Il framework è progettato per scalare da un singolo server a migliaia di macchine, ognuna delle quali mette a disposizione le proprie risorse [3] .
Hadoop ormai si può considerare a tutti gli effetti una pietra miliare nel mondo Big Data, uno strumento di riferimento che presenta le seguenti caratteristiche [4]:

Scalabilità: il framework permette di gestire ed elaborare ed archiviare grandi quantità di dati, grazie al fatto che l'elaborazione è distribuita su più host, detti nodi. É possibile inoltre integrare in maniera semplice nuovi nodi, in modo da aggiungere nuove risorse ai cluster.

Affidabilità: i grandi data center per loro natura sono soggetti ad un numero di guasti considerevole, a causa dei malfunzionamenti che possono occorrere ai singoli nodi. Hadoop è stato sviluppato con l'idea che i guasti siano non l'eccezione, ma eventi comuni e pertanto tutti i componenti prevedono una gestione automatica interna di eventuali problemi. Una delle tipiche contromisure ai guasti, ad esempio è la replica automatica dei dati, che permette di reindirizzare l'elaborazione ad altri nodi, qualora uno cadesse. Questo approccio si sposa perfettamente con l'utilizzo di commodity hardware, cioè di componenti relativamente economici e molto diffusi e ciò porta anche all'abbattimento dei costi complessivi.

Flessibilità: come abbiamo visto, la maggior parte dei dati elaborati in ambito aziendale è di tipo non strutturato. A differenza dei sistemi tradizionali di gestione delle basi dati, non è più sempre necessario definire la struttura delle informazioni prima di archiviarle: è possibile infatti salvare i dati in qualsiasi formato, strutturato e non, per poi gestirli nella maniera più opportuna durante successive elaborazioni.

Costi: Hadoop è stato rilasciato sotto licenza Apache [5], ma viene offerta dal mercato la possibilità di usufruire di implementazioni commerciali a pagamento, come Cloudera e Hortonworks, in base alle proprie esigenze.

L'ecosistema Hadoop è costituito da diversi componenti che compongono un'infrastruttura di elaborazione basata su un file system distribuito, l'Hadoop Distributed File System (HDFS) e del paradigma di programmazione MapReduce. Vediamo ora nel dettaglio i principali componenti:

Hadoop Common: è un layer di livello software che fornisce accesso al file system di Hadoop e in generale supporta tutti gli altri moduli Hadoop.

Hadoop Distributed File System: è un file system distribuito tra i nodi all'interno di un cluster. Al fine di fronteggiare eventuali guasti dei nodi e quindi la possibile indisponibilità di alcuni dati, permette di replicarli in modo da massimizzarne la disponibilità. Inoltre è studiato per fornire throughput elevati e basse latenze durante operazioni di IO.

Hadoop YARN: è il framework che consente di schedulare job e di gestire le risorse presenti nel cluster.

Hadoop MapReduce: è un framework prodotto da Google che permette di elaborare in maniera distribuita e parallela sui nodi del cluster grandi set di dati.

Quando si parla di Hadoop, si fa riferimento tipicamente all'intero ecosistema che è nato grazie al grande successo del framework (figura 3.2). Questo, infatti, si basa sui quattro componenti appena descritti, ma è stato via via arricchito da diversi moduli che offrissero diverse funzionalità all'utente, come Apache Flume, Apache ZooKeeper, Apache Hive e così via.

3.1.1 Architettura Hadoop

Un *cluster* è un insieme di due o più nodi connessi fra loro tramite reti ad alta velocità, dove per nodo si intende un host, cioè un computer. La dimensione dei cluster può variare, a seconda della necessità, da pochi nodi o da migliaia. Questi sono organizzati all'interno dei data center in *rack*, cioè scaffali, connessi fra loro tramite switch, che possono contenere un numero variabile di server (tipicamente fra 16 e 24). La figura 3.3 mostra un esempio di come possa essere organizzato un cluster Hadoop.

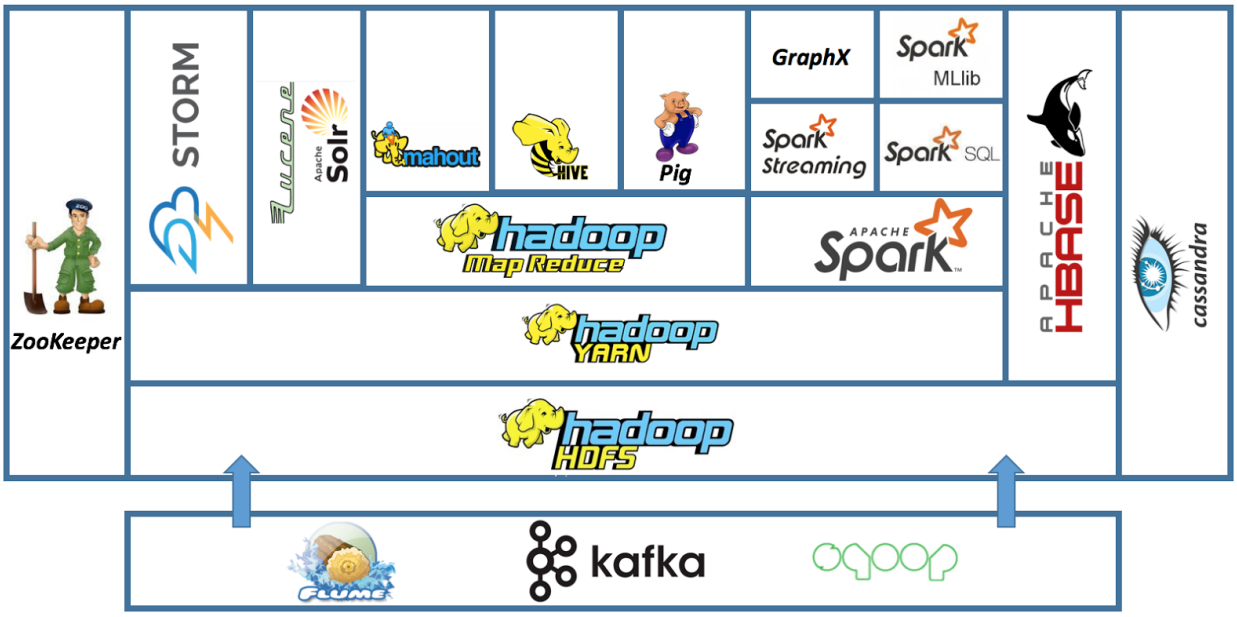

Figura 3.2: Ecosistema Hadoop

In un cluster Hadoop possiamo trovare due diverse categorie di nodi: *master node* e *worker node*.
I nodi master svolgono una funzione di supervisione e coordinamento delle attività principali. Fra questi troviamo il *JobTracker*, che ha il compito di gestire l'esecuzione distribuita delle applicazioni Map Reduce, ma anche il *NameNode*, che garantisce la gestione degli accessi al file system distribuito.

In ambiente distribuito è comune avere la necessità di ottenere maggiori risorse in caso di aumenti sul carico di lavoro e da qui nasce la necessità di affrontare il problema della scalabilità. Da questo punto di vista, si può scegliere di adottare uno di questi due approcci:

Scalabilità Orizzontale (scale out): consiste nell'aggiungere più nodi al cluster. Questo tipo di approccio è adottato generalmente quando si hanno grosse disponibilità di commodity hardware, cioè di dispositivi molto diffusi e dal costo limitato, che sono potenzialmente rimpiazzabili da altri in caso di necessità. Concettualmente, è l'opposto dell'*hardware dedicato*,

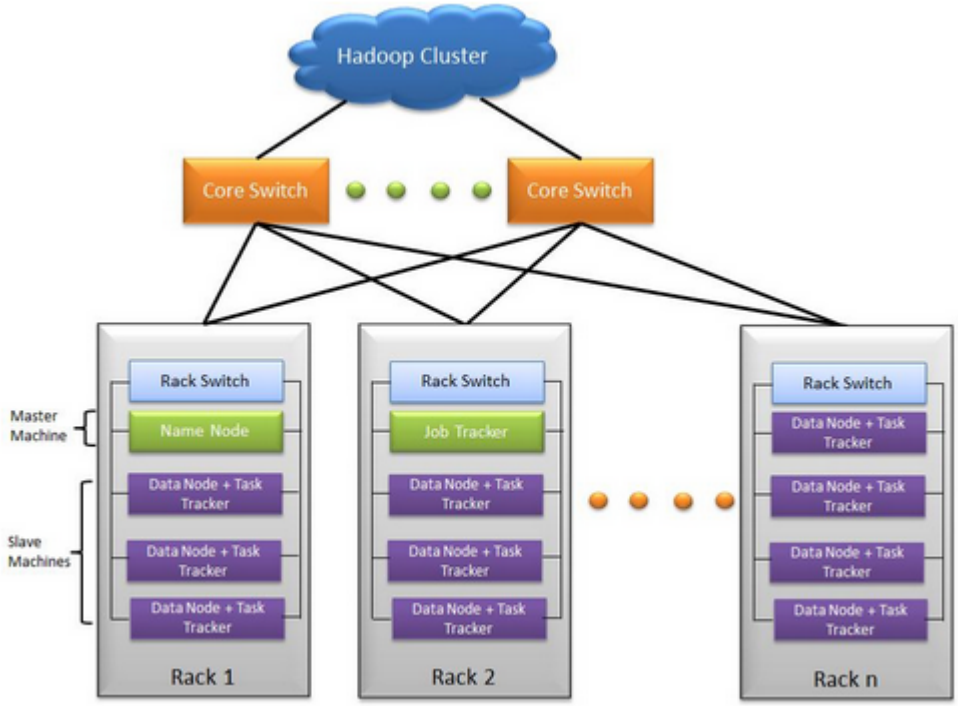

Figura 3.3: Esempio di configurazione di un cluster Hadoop

cioè di prodotti progettati ad hoc, che prevedono costi più alti, sono più difficili da reperire e per i quali è complesso trovare dispositivi compatibili in grado di sostituirli in caso di necessità.

Scalabilità Verticale (scale up): questo secondo approccio consiste nel potenziare i singoli nodi con ulteriori risorse, migliorandone la memoria, lo spazio su disco, o il numero di CPU. Solitamente si adotta questa strategia nel caso in cui si decida di usare un numero limitato di host, ma altamente performanti.

Come scritto in precedenza, Hadoop rende facile l'aggiunta di nuovi nodi al cluster, proprio grazie alla scelta di sposare la strategia della scalabilità orizzontale. L'utente che utilizza Hadoop inoltre, non deve preoccuparsi di gestire lo scheduling e la sincronizzazione dei task nell'ambiente distribuito, che gli risulta trasparente e viene gestito internamente dal framework, potendo concentrarsi così direttamente sull'implementazione delle varie soluzioni software.

3.2 Hadoop Distributed File System (HDFS)

L'Hadoop Distributed File System è il file system, scritto in Java, su cui si basa l'architettura Hadoop, costruito in modo da garantire affidabilità e scalabilità. Come accade anche per gli altri componenti, l'idea su cui si basa è il considerare guasti hardware come eventi usuali e non eccezioni. A seconda delle esigenze, infatti, un'istanza HDFS può essere costituita anche da migliaia di nodi, per cui è impensabile pensare di trascurare la possibilità che in un dato istante uno di questi abbia un guasto.
L'HDFS si ispira al file system proprietario di Google, che venne descritto per la prima volta in un documento rilasciato proprio da Google nel 2003 [6]. Si tratta di un file system pensato proprio per l'utilizzo su Hadoop, quindi non di tipo *general purpose*, progettato in modo da rilassare alcune delle specifiche dello standard POSIX, per ottimizzare le performance degli applicativi [7].
Le applicazioni Big Data che usano HDFS hanno solitamente a che fare con file di dimensioni molto alte, spesso dell'ordine dei GB o dei TB. Ogni file, all'interno del file system viene diviso in *chunk*, blocchi più piccoli tutti della stessa dimensione, fatta eccezione per l'ultimo. La loro grandezza è configurabile per ogni file, ma tipicamente si usano valori compresi fra 16 e 128 MB. Per garantire affidabilità, ogni blocco viene replicato su diversi host, in modo da reagire prontamente a parti di file corrotte o irragiungibilità di uno dei nodi che li ospita. Si parla di *replication factor (fattore di replica)* per indicare il numero di copie.

La figura 3.4 mostra una possibile configurazione di un cluster con un fattore di replica pari a tre. Un eventuale guasto può occorrere a diversi livelli, e la posizione delle repliche è pensata proprio per reagire in ogni caso: una copia si trova in un secondo nodo all'interno dello stesso rack, come contromisura ad un guasto sul primo nodo; per non perdere dati nel caso in cui il problema coinvolga l'intero rack, un'altra replica viene inserita all'interno di un rack differente.

La topologia dei nodi all'interno del cluster, come si può immaginare, ha un ruolo fondamentale in termini di performance nel corso delle operazioni di IO. Il framework è stato concepito

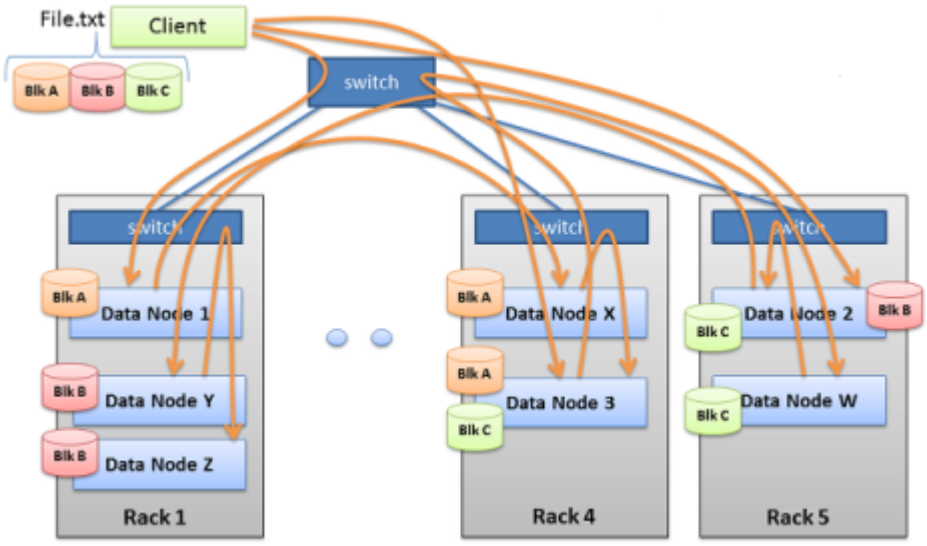

Figura 3.4: Configurazione con fattore di replica uguale a tre

per minimizzare il consumo di banda e la latenza durante queste attività, garantendo che i dati vengano letti sempre dal nodo più vicino a chi ha lanciato il comando di read. Nel caso migliore, questo è presente sulla stessa macchina, altrimenti si cercano copie all'interno dello stesso rack, o, in ultima istanza, viene selezionata una copia dal rack più vicino.

3.2.1 Architettura HDFS

L'HDFS è scritto interamente in Java e, nonostante non ci siano limiti sul numero di nodi che possono essere attivi su una singola macchina, tipicamente viene istanziato un singolo nodo HDFS su ogni host.

Il file system distribuito di Hadoop è basato su un'architettura di tipo master-slave: in ogni cluster è sempre presente un singolo master, chiamato *NameNode* e un numero variabile di slave, i *DataNode*.

NameNode: le operazioni di apertura, chiusura, rinominazione di file e cartelle è demandata a questo nodo, che si occupa in generale di gestire le richieste di accesso dei vari nodi ai file. Il NameNode determina inoltre il mapping dei blocchi che compongono i file e gestisce le loro repliche sui vari DataNode. Si occupa infine dei metadati del file system, ma non entra mai direttamente nell'ambito dei dati applicativi.

DataNode: coordinati dal NameNode, i DataNode gestiscono le operazioni di creazione, cancellazione e replica dei dati. Rispondono quindi alle richieste di lettura e scrittura dei singoli blocchi e in generale sono adibiti alla gestione dei dati presenti sui nodi in cui risiedono.

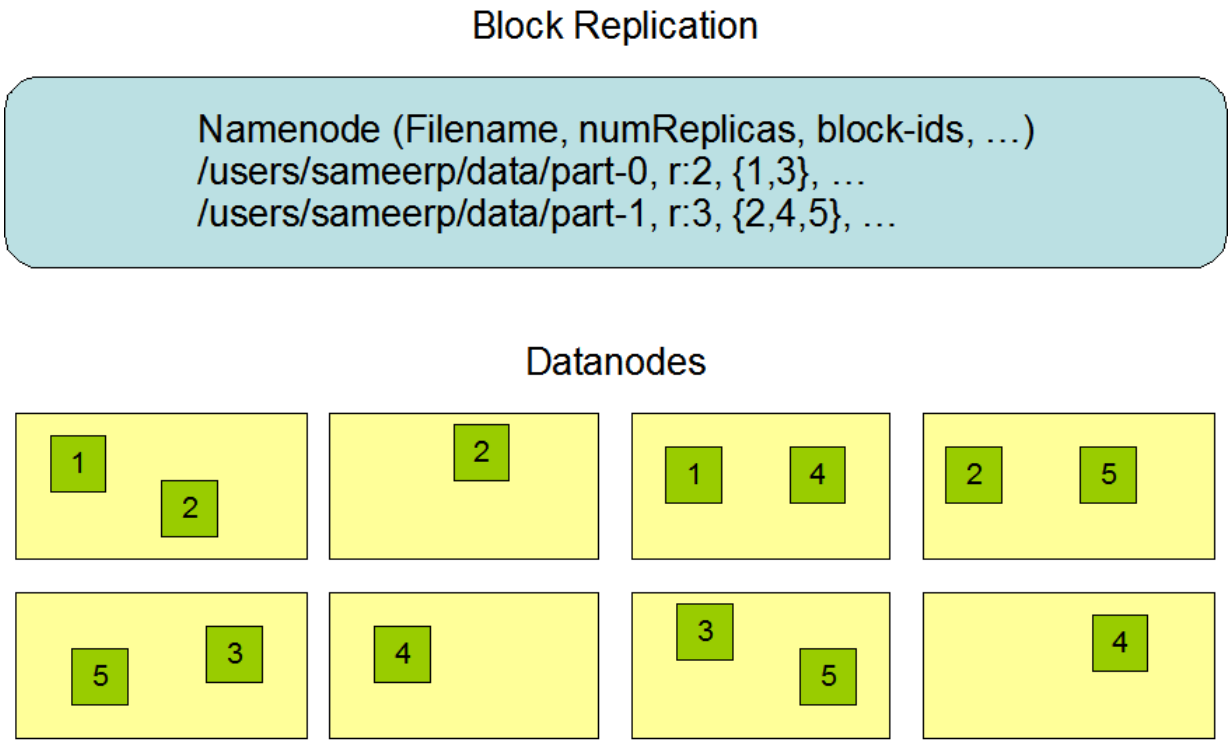

Figura 3.5: NameNode e DataNode nell'HDFS

3.2.2 HDFS High Availability

La prima versione di Hadoop prevedeva che ogni cluster avesse un singolo NameNode, per avere un'architettura il più possibile semplificata. Questo però rendeva il NameNode un *single point*

of failure, in quanto, essendo responsabile dell'accesso ai dati archiviati sui DataNode, bastava un guasto sul suo host per rendere inutilizzabile il cluster o comunque inaccessibili i dati fino al riavvio.

Per fronteggiare questo problema, nella versione 2.0 è stata introdotta la funzionalità dell'*HDF High Availability*, grazie alla quale è possibile creare due NameNode nel cluster: uno dei due NameNode così funge da nodo attivo, mentre l'altro è in uno stato di *hot standby*, cioè è pronto a subentrare al principale in caso di guasti, in maniera rapida.

Chiaramente, il tutto funziona a patto che il NameNode attivo e quello in standby siano sempre sincronizzati, per conoscere la posizione dei blocchi e il namespace di riferimento. Per far sì che si raggiunga questa condizione, viene utilizzato un log di edit situato in una cartella condivisa, in cui sono registrati tutti gli eventuali cambiamenti. In particolare sono i DataNode che inviano in maniera automatica queste informazioni. Il NameNode in standby periodicamente legge i log e applica le modifiche scritte al suo interno, in modo da essere sempre pronto al risveglio.

I messaggi di stato inviati dai DataNode sono di due tipi: *Heartbeat* e *Block Report*:

HeartBeat: è un segnale che il DataNode invia ai NameNode periodicamente per indicare la sua presenza, cioè per segnalare che è vivo e funziona correttamente. Di default l'intervallo di trasmissione è settato a tre secondi, ma si tratta di un parametro configurabile. Se un NameNode non riceve heartbeat da una certa quantità di tempo, configurabile, quel DataNode viene considerato come fuori servizio, per cui le repliche che ospita non sono più disponibili. Se questo porta ad un numero di repliche di un blocco inferiore al valore impostato come fattore di replica, il NameNode pianifica ed avvia il processo di creazione di nuove copie di quel blocco.

Block Report: si tratta di un report prodotto dai DataNode ed inviato al NameNode ad intervalli regolari, anche questi configurabili. I DataNode, infatti, oltre a salvare i file HDFS sul proprio file system locale, tengono anche traccia di alcuni metadati, come i checksum. I Block Report vengono quindi generati proprio in seguito alla scansione dei file system, in modo che il NameNode sia sempre aggiornato sullo stato delle repliche sui vari nodi.

3.2.3 Persistenza dei metadati dell'HDFS

La gestione del namespace HDFS è demandata al NameNode, che la effettua tramite l'ausilio di due file, il *FsImage* e l'*EditLog*, che conserva nel suo file system locale. Il primo, l'FsImage, costituisce un'istantanea dei metadati sul file system, cioè contiene lo stato completo del file system in un momento specifico. Ad ogni modifica del file system viene assegnato un ID univoco e monotonicamente crescente, pertanto un file FsImage rappresenta lo stato del file system dopo l'applicazione di queste transizioni. L'Edit Log invece registra ogni modifica a partire dall'ultima istantanea effettivamente memorizzata sull'FSImage, per evitare che venga prodotto un nuovo FSImage ad ogni modifica, anche piccola, come la rinominazione di un file. All'avvio, il NameNode esegue un'operazione chiamata *checkpoint*, cioè applica le modifiche scritte sull'EditLog sull'immagine contenuta nel file FSImage. Un checkpoint quindi occorre solo all'avvio di un nodo e consiste nelle seguenti operazioni:

1. Lettura da parte del NameNode dei file di Edit Log e FsImage.
2. Applicazione delle modifiche scritte sull'Edit Log al FsImage.
3. Sostituzione della vecchia versione del FsImage su disco, con l'immagine appena aggiornata.
4. Cancellazione del contenuto del file dell'Edit Log, in quanto le modifiche sono state ormai applicate all'immagine del file system sia in memoria, sia su disco.

Vista l'importanza dell'EditLog e del FsImage nell'ambito HDFS, per assicurarsi che un loro eventuale danneggiamento non renda l'intero sistema inutilizzabile, è possibile configurare il NameNode in modo da crearne copie multiple, che comunque vanno tenute aggiornate. La sincronizzazione può degradare le performance del NameNode, ma è un trade-off necessario per garantire la stabilità del sistema.
Una delle novità presenti nella versione 2.0 di Hadoop, è stata l'introduzione di un nodo chiamato *Secondary NameNode*. Nella vecchia versione, veniva eseguito il mapping delle modifiche dell'EditLog sul FsImage solo durante l'avvio del NameNode. Di conseguenza, era possibile che l'EditLog raggiungesse dimensioni considerevoli e che quindi venisse rallentata in maniera decisa la fase di

partenza di un nodo. Il Secondary NameNode, a dispetto del nome, non è una replica, ma ha semplicemente la funzione di eseguire periodicamente il merge dei due file, per evitare che l'EditLog raggiunga dimensioni eccessive. Nella figura 3.6 viene spiegato nel dettaglio il funzionamento di questo sistema:

1. Ad intervalli regolari, il NameNode Master recupera i file EditLog e FsImage.
2. Viene prodotto un nuovo file FsImage, tramite l'applicazione delle modifiche, e viene inviato al NameNode Master.
3. Al successivo avvio, il NameNode Master adopererà la nuova versione del file FsImage ricevuta.

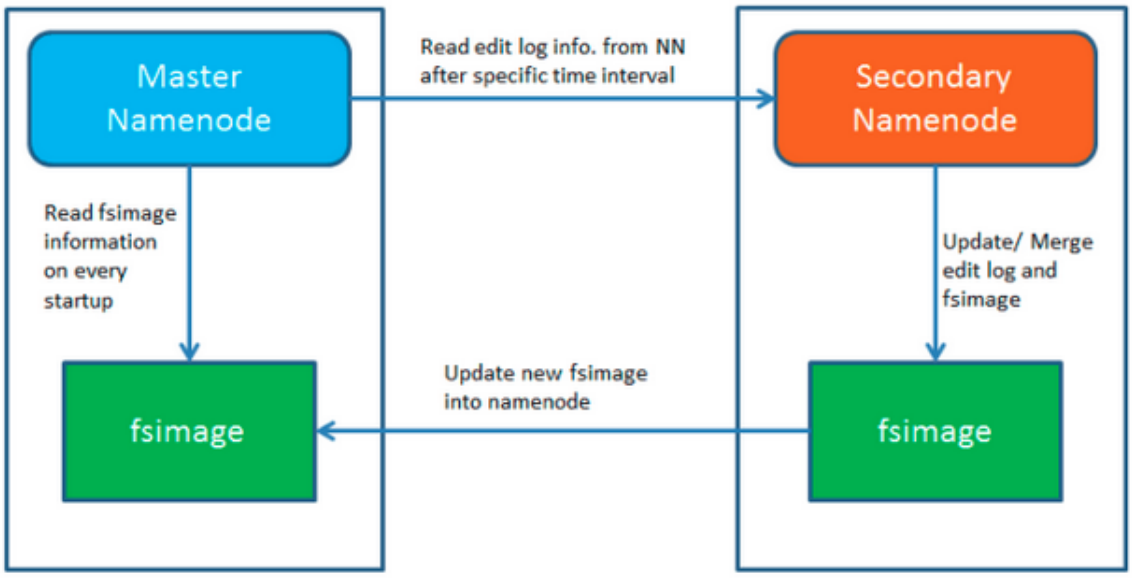

Figura 3.6: Il NameNode Secondario

3.2.4 Fase di Staging

La richiesta di creazione di un nuovo file da parte di un client non viene soddisfatta in maniera immediata, inoltrandola al NameNode, ma i dati vengono spostati e bufferizzati su una porzione

temporanea del file system del client. Questo contatta effettivamente il NameNode solo quando il file viene chiuso o quando viene raggiunta una quantità di dati sufficiente a riempire un blocco. A quel punto i dati vengono scritti su HDFS, seguendo questi passi:

1. Il NameNode inserisce un nuovo elemento nel file system e alloca un blocco in uno dei DataNode.

2. Il NameNode risponde alla richiesta del client indicando quale DataNode sta per ricevere i dati.

3. Il client invia i dati dal file system locale al DataNode e comunica al NameNode che può chiudere il file.

4. Il NameNode memorizza l'operazione inserendola nell'EditLog. Qualora il NameNode dovesse cadere prima che le modifiche contenute nell'EditLog vengano scritte, il riferimento al file verrebbe perso.

In Hadoop è configurabile il fattore di replica per i vari file: se è maggiore di uno, la fase di staging è seguita da quella di *replication pipelining*. In questo caso, cioè, il client richiede la creazione di un file e il NameNode risponde con una lista contenente i DataNode che riceveranno i blocchi. Il client, quindi, oltre a trasmettere il dato al primo DataNode, invia anche la lista degli altri DataNode, ricevuta dal NameNode. Il primo DataNode, ricevuta la sua porzione di file, la salva sul proprio file system locale e contestualmente la invia agli altri DataNode della lista.
La fase di staging rappresenta uno dei rilassamenti introdotti rispetto alle specifiche POSIX durante la progettazione del file system, principalmente allo scopo di evitare che la rete venisse congestionata dalla scrittura dei file eseguita direttamente dai client.

3.3 MapReduce

L'architettura Hadoop si basa su un paradigma di programmazione distribuito chiamato MapReduce, creato da Google all'inizio degli anni 2000, pubblicato ufficialmente in un documento del

2004 [8].
L'idea alla base del modello di MapReduce è quella di permettere di creare applicazioni distribuite, in grado di processare grandi quantità di dati, garantendo affidabilità e resistenza ad eventuali guasti.
Fra le varie implementazioni, quella presa in esame è la versione open source di Apache Hadoop.
MapReduce si adatta a svariati tipi di use-case e in particolare, fra le sue applicazioni si possono menzionare:

- Le analisi di testi, con funzioni di indicizzazione e di ricerca.
- L'analisi di file di Log.
- La ricerca basata su strutture dati complesse, come i grafici.
- Data mining
- Algoritmi di Machine Learning
- Analisi numeriche e attività di computazione matematica complessa, tramite task distribuiti.

Lo stesso Google ha implementato applicativi in grado di generare indici di ricerca nelle pagine web, tramite l'uso della propria versione di MapReduce.

3.3.1 Principi operativi

L'idea alla base del modello MapReduce è quella di mappare i dati su coppie *key-Value (chiave-valore)*: degli esempi possono essere l'URL di una pagina web come chiave e il contenuto HTML come valore, o il numero di riga di un testo (chiave) e la frase corrispondente.
Un programma basato su MapReduce è composto da due step fondamentali, che danno il nome al framework:

Map: in questa fase, i dati di input, che possono essere di diversi formati, vengono inviati al nodo master, che li divide in blocchi e fa in modo che i job vengano distribuiti ai nodi che fungono

da slaves. Un nodo mapper applica trasformazioni ai dati e genera l'output della funzione map() sottoforma di coppia (chiave, valore), memorizzandolo sul file system distribuito. Si tratta di un risultato intermedio, la cui posizione viene indicata al master, al termine della fase di map.

Reduce: si tratta della seconda fase dell'elaborazione, in cui il nodo master raccoglie tutti gli output della fase di map e trasforma le coppie (chiave, valore) in una lista di valori che hanno la chiave comune, ordinandoli proprio in base a questa, in una fase chiamata *Shuffle*. L'ordine può essere lessicografico, crescente o customizzabile da parte dell'utente. Le coppie prodotte sono del tipo chiave, Lista(valore, valore, ...)) e vengono inviate ai nodi ai quali è demandata l'esecuzione delle funzione reduce().

Riassumendo, la funzione di Map può essere così definita: Map(k1,v1) ->list(k2,v2)
La funzione Reduce invece è la seguente: Reduce(k2, list (v2)) ->list(v3)

Durante l'esecuzione di un'applicazione MapReduce, il programmatore non deve preoccuparsi di alcune operazioni intermedie, che vengono gestite internamente dal framework in modo trasparente, come ad esempio l'aggregazione per chiave dei valori prodotti dal Mapper.

Al momento del submit di un Job MapReduce, è prevista la copia dell'applicazione sui nodi del cluster che devono eseguire l'elaborazione distribuita. Un programma MapReduce è costituito da tre diverse parti, ognuna implementata da una classe:

Driver: è un'istanza (l'unica) della classe Driver, che viene eseguita sul client. Essa contiene il codice di configurazione del job e coordina il flusso di esecuzione sul cluster.

Mapper: è un'istanza della classe mapper che implementa la fase di mapping. Questa è completamente parallelizzabile, per cui stavolta ne vengono eseguite istanze multiple. Il numero di mapper, al fine di sfruttare al meglio il parallelismo, è pari al numero di blocchi in input. Le istanze vengono eseguite dai nodi del cluster, ed in base alla capacità degli host, possono anche essere presenti istanze multiple sulla stessa macchina. Per minimizzare l'utilizzo

della rete, che può essere molto costoso lavorando con grandi quantità di dati, il framework garantisce che le istanze, ove possibile, siano eseguite sui nodi che contengono i dati stessi.

Reducer: è un'istanza della classe reducer, che implementa la fase di reduce. Anche queste istanze sono eseguite sui nodi del cluster, ma stavolta il loro numero viene settato dall'utente all'interno della classe driver, e varia a seconda del tipo di programma. In output, ogni reducer produce un file e lo salva su HDFS.

L'esempio più semplice e famoso di programma MapReduce è il problema del Word Count, che si pone l'obiettivo di contare le occorrenze delle parole in un file di testo. Nella figura 3.7 viene mostrato il suo funzionamento.

In questo caso, il file testuale viene diviso in quattro blocchi su HDFS, in ognuno dei quali è memorizzato un certo numero di record, che non sono altro che una sequenza di parole.
Per ogni blocco viene istanziato un Mapper, ognuno dei quali riceve come input una riga di testo alla volta, sotto forma di coppia *(NULL), riga di testo*. La chiave, in questo caso, è irrilevante, per cui viene settata a NULL per ridurre al minimo le informazioni che andranno inviate sulla rete.
Ogni mapper processa le righe di input, divide le parole e produce per ognuna di esse, una coppia *(parola, 1)*, dove il numero 1 del valore sta ad indicare che quel termine appare una volta sola.
Queste coppie vengono memorizzate temporaneamente sul file system locale dei nodi Mapper. A questo punto vi è una fase di ordinamento chiamata Shuffle, in cui tipicamente viene applicata una funzione hash alla chiave delle coppie. Segue quindi l'invio delle coppie ai nodi a cui è demandata la fase di Reduce. In particolare, le coppie che condividono la chiave sono raggruppate e inviate allo stesso Reducer, in base al risultato dell'hash precedente.
I Reducer quindi ricevono, una alla volta, coppie del tipo (chiave, list(1,1,1,...), con tanti elementi nella lista quante sono le occorrenze della chiave. Pertanto, per ogni coppia, i valori unitari vengono sommati, ottenendo il conteggio finale per ogni termine all'interno del testo. La coppia *parola, occorrenze* viene emessa e il risultato finale scritto sull'HDFS.
Ogni Reducer nella fattispecie, genererà un file di output. Questi, insieme ai file di input del programma, sono generalmente salvati su HDFS. I prodotti intermedi del processo invece, come gli

output dei Mapper, non stanno sul file system distribuito, ma su quello locale dei nodi che li hanno eseguiti.

Per valutare la bontà delle performance di un programma di Map Reduce, è cruciale considerare la quantità dei dati che vengono inviati attraverso la rete durante il transito dai nodi Mapper ai nodi Reducer, in quanto, vista l'alta quantità di dati, è facile avere un degrado delle prestazioni causato dal congestionamento proprio sulla rete.

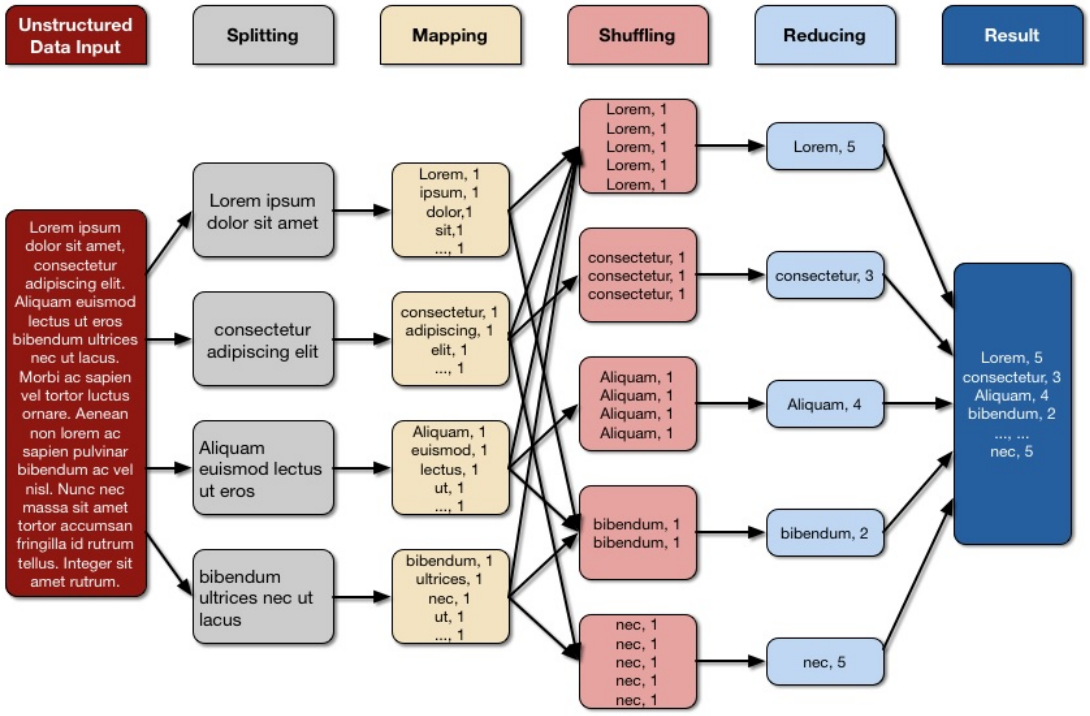

Figura 3.7: Word Count - Map Reduce

Una delle possibili ottimizzazioni del Map Reduce viene dall'utilizzo di una fase chiamata Combiner, che opera in modo analogo al Reducer, ma agendo sui dati locali, prima dell'invio sulla rete. Nell'esempio, infatti, è possibile notare come la coppia *parola, 1* venga emessa dal Mapper tante volte quante sono le occorrenze della parola all'interno del blocco, portando a possibili degradi delle performance di rete. Tramite il combiner si può eseguire una precomputazione in locale,

riducendo la quantità di dati complessivi da inviare al Reducer.

3.3.2 Limiti del Map Reduce

Il paradigma MapReduce è uno dei capostipiti della programmazione in ambito Big Data e, anche se è ancora largamente utilizzato in svariati campi, ci sono delle situazioni in cui è difficilmente applicabile. Esso ha infatti limiti dovuti a scarsa flessibilità e alla gestione dei risultati intermedi:

- Si tratta di un modello semplice, che non si adatta pienamente alla risoluzione di problemi complessi.
- Essendo stato progettato per applicazioni batch, non si adatta a quelle streaming o interattive.
- Non performa bene in caso di problemi iterativi, in quanto ad ogni ciclo sarebbe necessario leggere dati dal disco.

Questi limiti hanno portato, negli ultimi anni, allo sviluppo di strade alternative al MapReduce, che, grazie all'introduzione di Apache YARN in Hadoop, sono state integrate all'interno del framework.
Sono stati inoltre sviluppati prodotti che mascherano la complessità dei programmi Map Reduce, come *Apache Hive* e *Apache Pig*, che traducono automaticamente delle query scritte in un linguaggio sql-like in una sequenza di operazioni Map Reduce.

3.4 Apache YARN

Fra i quattro componenti principali di Hadoop 2.0, si distingue Apache YARN (Yet Another Resource Negotiator), che ha in carico la gestione delle risorse complessive del sistema, e la schedulazione dei processi all'interno del cluster.

YARN è stato progettato con l'idea di separare le funzionalità di gestione delle risorse dalle attività di scheduling e monitoraggio dei job, assegnando ognuna di esse a demoni diversi. In particolare, vi è un *ResourceManager* globale e un *ApplicationMaster* per ogni applicazione. I servizi offerti da YARN vengono gestiti per mezzo di due demoni:

ResourceManager: è il demone master, presente in un'unica istanza all'interno del cluster, che si occupa della ricezione delle richieste e la successiva assegnazione delle risorse da parte dei client. Ha inoltre il compito di tener traccia di quanti siano i nodi attivi e di assegnare i task ai singoli worker.

NodeManager: è il demone slave di YARN, di cui è presente un'istanza per ogni nodo. Il suo compito è quello di mantenersi aggiornato con il ResourceManager, monitorando l'utilizzo delle risorse, come CPU, memoria, spazio di archiviazione e in generale lo stato del nodo su cui è in esecuzione. Col termine NodeManager tipicamente si indica sia il processo, sia il nodo host che lo ospita.

La figura 3.8 mostra lo schema architetturale di YARN.

Ogni applicazione è divisa in *task* e uno di quelli definiti dal framework è il cosiddetto *ApplicationMaster*, che ha il compito di negoziare le risorse richieste dai client con il ResourceManager e di lanciare e monitorare i task sul NodeManager.
Il ResourceManager, a sua volta, presenta due componenti principali, un *Resource Scheduler* e un *ApplicationManager*:

Resource Scheduler: è il componente che gestisce l'allocazione delle risorse richieste dalle applicazioni. In generale, lancia i suoi task in relazione alle richieste e alla quantità di risorse disponibili su tutti i nodi.

ApplicationManager: si occupa della gestione della lista delle applicazioni che vengono inoltrate. Per ogni applicazione, controlla che non siano state fatte richieste di risorse non soddisfacibili dall'ApplicationMaster e controlla anche che non ci siano altre applicazioni

con lo stesso ID, per evitare eventuali attacchi di un malintenzionato. Alla fine, inoltra l'applicazione allo scheduler. Permette inoltre di restartare l'ApplicationMaster nel caso in cui un task fallisca.

Il ResourceManager è il componente che possiede la visione d'insieme di tutte le risorse del cluster e di dove siano quelle disponibili, in quanto ogni NodeManager gli comunica lo stato del nodo che lo ospita.
Ogni applicazione può richiedere come risorse ulteriore memoria o maggiore potenza computazionale, ma presto sarà possibile chiedere anche più GPU, o maggiore utilizzo di banda o porzioni di disco.
Le risorse che vengono assegnate dal Resource Scheduler sono raggruppate in un blocco logico, chiamato *Container*.

Ogni applicazione quindi viene eseguita attraversando i seguenti step:

1. L'applicazione contatta il ResourceManager.
2. Viene assegnato un container di risorse all'applicazione dal ResourceManager.
3. Assegnato il primo container, parte l'esecuzione dell'ApplicationMaster.
4. L'ApplicationMaster richiede nuovi container che servono all'applicazione e negozia con il ResourceManager. L'applicazione viene divisa in processi chiamati task e inizia così il processamento.
5. Al termine dei task, i Container vengono deallocati. Alla fine, anche l'ApplicationMaster termina e con lui, viene rilasciato l'ultimo container.
6. L'applicazione esce.

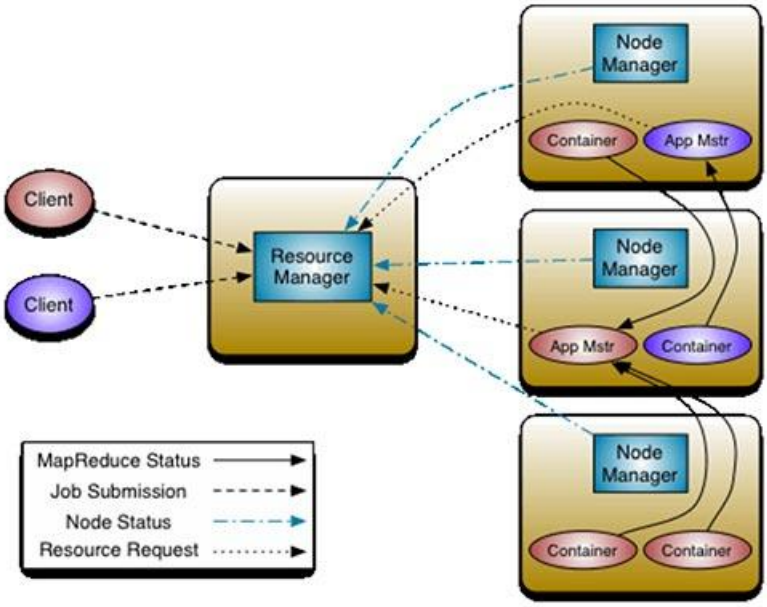

Figura 3.8: Architettura YARN

3.5 Apache Kafka

Figura 3.9: Logo Kafka

Apache Kafka è una piattaforma di stream processing di tipo open source [9], scritta in Java e basata sul pattern *publisher/subscriber*. Si tratta quindi di un sistema capace di gestire flussi in tempo reale, in grado di offrire ottime performance in termini di throughput e latenza, e persistenza dei dati. L'architettura Kafka consta di quattro componenti fondamentali:

Topic: è la categoria a cui appartiene un certo flusso di messaggi, intesi come sequenze di byte.

Producer: è il soggetto che produce i messaggi, pubblicandoli nei topic.

Consumer: è il soggetto che consuma i messaggi, registrandosi ad uno o più topic.

Brokers o Kafka Cluster : rappresenta il set di macchine su cui risiedono i topic e i dati che vengono pubblicati su essi.

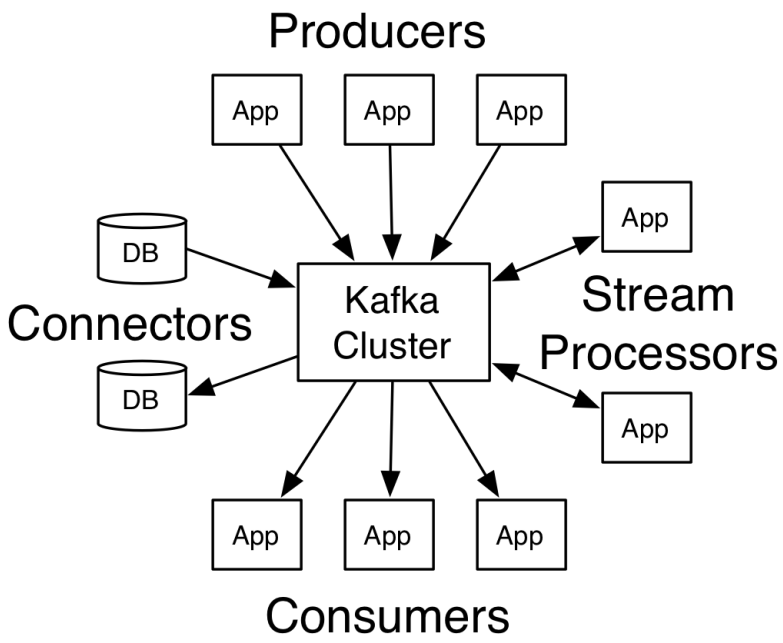

Figura 3.10: Architettura Kafka

Le funzionalità di Kafka sono fruibili attraverso delle API:

Producer API: serve a pubblicare messaggi su uno o più topic.

Consumer API: viene usata dall'applicazione per registrarsi ai topic e processare i messaggi.

Streams API: consente all'applicativo di agire come un processore stream, consumando i messaggi dai topic, trasformandoli e ripubblicandoli su un nuovo topic.

Connect API: è un'interfaccia che semplifica e automatizza l'integrazione di una nuova sorgente di dati o il collegamento fra i topic Kafka e le applicazioni.

Il cuore di Kafka dunque è rappresentato dai topic e ognuno di essi è memorizzato in un log. Questo è diviso in partizioni, come indicato in figura 3.11.
Ogni partizione non è altro che una sequenza ordinata e immutabile di record, che vengono accodati ad esse, rendendole un vero e proprio registro di commit. Ad ogni elemento delle partizioni infatti viene assegnato un id numerico incrementale, chiamato *offset*, univoco all'interno della partizione.

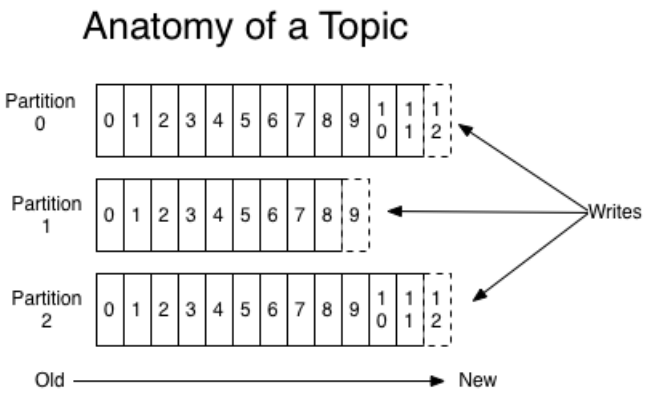

Figura 3.11: Struttura di un topic Kafka

Esiste la possibilità di settare il tempo massimo di conservazione dei record pubblicati, che possono essere mantenuti nel cluster Kafka, che siano stati consumati o meno. Le performance di Kafka infatti non sono inficiate dalla dimensione dei dati, per cui conservarli anche per lunghi periodi non è un problema [9].

A livello di metadati, vengono mantenuti per ogni consumer, gli offset, cioè le loro posizioni all'interno del log. É il consumer che ha il controllo degli offset: può scegliere di far avanzare il suo offset in modo lineare, conseguentemente alla lettura dei record, così come consumare i record in qualsiasi ordine voglia. Ad esempio, un consumer potrebbe decidere di saltare al record più recente, iniziando a consumare a partire dal timestamp attuale, o, viceversa, potrebbe voler riprocessare dei dati del passato, ripristinando un offset più vecchio. Uno dei vantaggi di questa organizzazione è che i consumer Kafka possono muoversi a loro piacimento, senza impattare sul

cluster o su altri consumer.

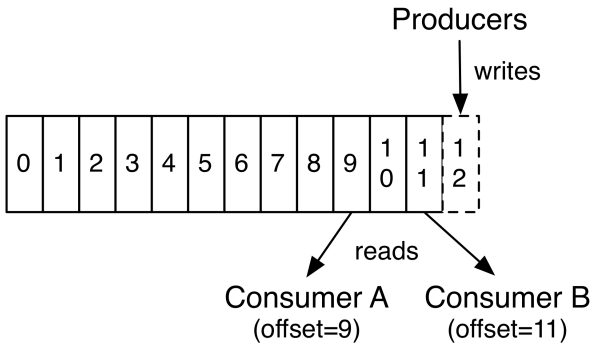

Figura 3.12: Kafka: Producer e Consumer

Le partizioni dei log vengono distribuite sui nodi del cluster Kafka, ognuno dei quali è responsabile della gestione delle richieste di condivisione delle sue partizioni. Fra le sue funzionalità, Kafka offre la possibilità di replicare le partizioni su altri nodi, secondo un numero configurabile, per fronteggiare eventuali guasti. Per ogni partizione vi è un server che funge da leader, che gestisce le richieste di lettura e scrittura, e zero o più server followers, che replicano in maniera passiva il leader. Il sistema complessivo risulta ben bilanciato per ciò che concerne il carico di lavoro, in quanto ogni nodo si comporta da leader per alcune delle sue partizioni e da follower per altre.

Fra le sue applicazioni, Kafka performa molto bene nel caso in cui si debbano gestire flussi streaming di dati, in quanto offre:

Scalabilità: è un sistema distribuito, pertanto è facile da scalare in caso di necessità senza dover fermare i flussi.

Durabilità: è possibile configurare il tempo di conservazione dei file sui dischi; anche scegliendo tempi lunghi, non vi sono grossi impatti in termini di performance.

Affidabilità: offre la possibilità di replicare i dati per fronteggiare eventuali malfunzionamenti. Supporta sottoscrizioni multiple ed è in grado di bilanciare automaticamente i consumer in caso di fallimento.

Performance: offre alte performance anche in caso di grossi volumi di dati, con throughput alti sia per i producer che per i consumer.

3.6 Apache Flume

Figura 3.13: Logo Flume

Apache Flume è un servizio distribuito, efficiente ed affidabile, di data ingestion, che permette la raccolta, l'aggregazione e il trasporto di grandi dati streaming, come file di log, da varie sorgenti a HDFS. Tale acquisizione tipicamente è coordinata da Apache YARN.
Con il termine *Event* viene indicata la singola unità di dati che viene trasportata da sorgente a destinazione. Il cuore di Flume è costituito da un processo Java, che prende il nome di *agent* e ha il compito di coordinare tre componenti:

Source: riceve in input dati dalla sorgente e li inoltra su uno o più canali sottoforma di event.

Channel: si tratta di uno storage temporaneo, su cui sono salvati gli event inviati dalla sorgente fino a che non vengono consumati dal sink.

Sink: consuma i dati recuperandoli dal canale e li porta a destinazione, ad esempio scrivendoli su HDFS

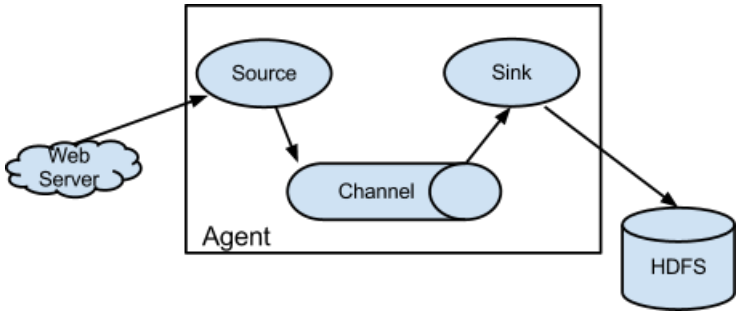

Figura 3.14: Agent Flume

Il successo di Flume in ambito Big Data è in costante aumento, grazie alla sua semplicità di uso e alla facile integrazione con altri componenti. Ad esempio, negli ultimi tempi è stata realizzata l'integrazione con Apache Kafka e Flume, chiamata *Flafka*.

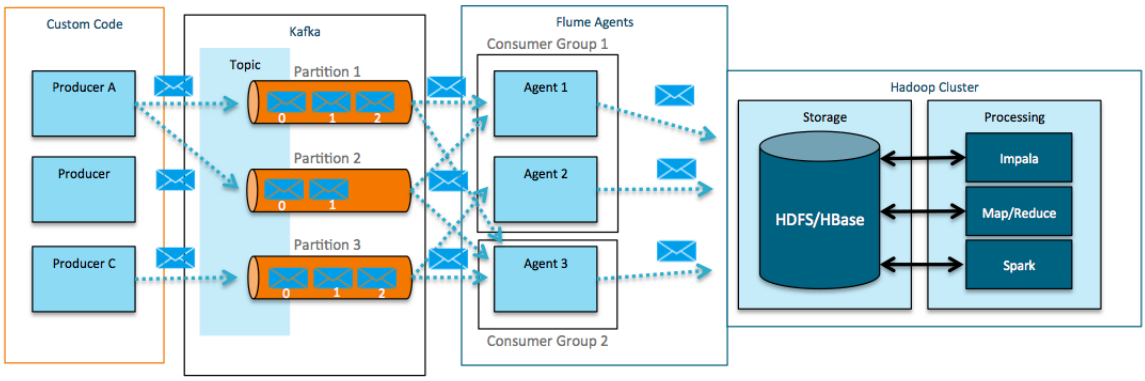

Figura 3.15: Esempio di un'architettura Flafka

Grazie alla collaborazione fra i due componenti, si riescono a creare producer e consumer Kafka senza aver bisogno di scrivere codice, ed è stata aggiunta inoltre la possibilità di processare o trasformare al volo i messaggi provenienti dai topic Kafka.

3.7 Apache Hive

Figura 3.16: Logo Hive

Apache Hive è un sistema di data warehouse per Hadoop che consente di analizzare in maniera semplice ed intuitiva i dati, interrogandoli tramite query scritte in un linguaggio simile a SQL.
Il sistema è stato creato nel 2007 dagli sviluppatori di Facebook, che avevano bisogno di gestire e analizzare i petabyte di informazioni prodotti tramite il social network, mentre attualmente è usato e sviluppato da altre aziende come Netflix e Amazon. Hive offre diverse funzionalità:

- Permette di leggere i dati tramite un linguaggio basato su SQL, *HiveQL*, che consente di eseguire interrogazioni sui dati ed operazioni ETL.
- Flessibilità sui formati di input e output.
- Possibilità di rendere strutturati i dati presenti su HDFS.
- Capacità di far girare query tramite programmi Spark e/o MapReduce.

Da un punto di vista strutturale, Hive è composto da diversi elementi:

Interfaccia Utente: permette agli utenti di scrivere, eseguire query e ottenerne il risultato.

Metastore: è uno dei componenti più delicati, in quanto conserva le informazioni sulle tabelle e le partizioni, fra cui struttura e nomi o tipi delle varie colonne. Oltre a questo, registra la

posizione dei dati sul file system e contiene indicazioni su come eseguire la lettura o scrittura dei dati.

Compilatore: genera un piano di esecuzione, dopo aver preso in consegna le query, averne verificato la correttezza semantica e aver recuperato le informazioni necessarie dal Metastore. Il piano corrisponde ad un grafo aciclico, i cui step sono operazioni di I/O su HDFS o job di MapReduce.

Motore di esecuzione: è il componente che esegue effettivamente le operazioni, riportate sul piano di esecuzione, assegnandole ai singoli componenti Hadoop, coordinandoli.

Driver: riceve le query dagli utenti, crea le sessioni e funge da ponte fra compilatore e motore di esecuzione, gestendo la comunicazione fra essi.

La figura 3.17 mostra lo schema architetturale di Hive.

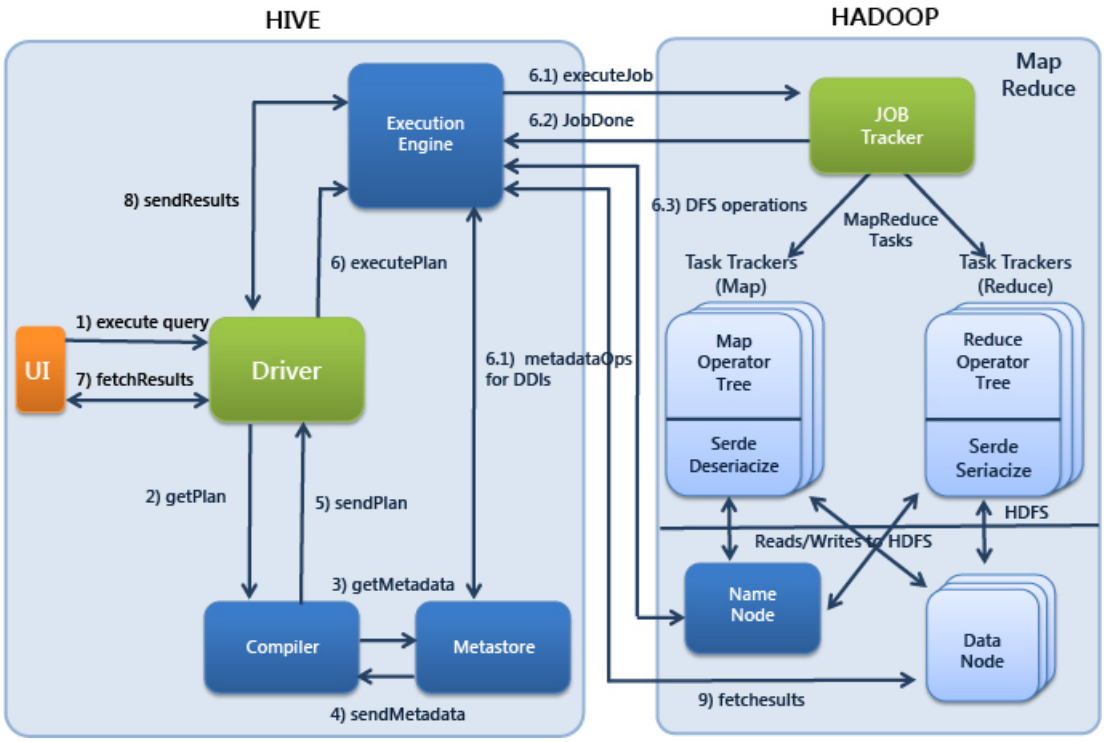

Figura 3.17: Architettura Hive

Hive separa la memorizzazione dello schema delle tabelle dalla quella dei dati veri e propri, ponendoli in blocchi diversi. L'Hive Metastore invece si preoccupa di salvare tutti i metadati di una tabella in un database relazionale, che viene letto ad ogni accesso alla tabella stessa.
Una delle possibilità offerte da Hive è quella di creare tabelle a partire da file già esistenti su HDFS. Visto il disaccoppiamento descritto in precedenza, infatti, si può raggiungere questo obiettivo senza modificare i blocchi sul file system, ma semplicemente il sistema creerà i metadati su indicazione dell'utente e successivamente sarà possibile accedere ai dati attraverso HiveQL.

Il servizio Hive Metastore, come accade per altri componenti Hadoop per evitare di avere un single point of failure, è spesso settato in modalità *High Availability mode*. In altre parole, esiste un secondo nodo che contiene una copia del metastore, che viene tenuto in standby, pronto a svegliarsi in caso di guasti.

3.8 Cloudera Impala

Figura 3.18: Logo Impala

Cloudera Impala è un motore per interrogazioni SQL open source ad elaborazione parallela di massa (MPP) di dati archiviati in host su cui viene eseguito Apache Hadoop. Il suo scopo è quello di unire i pregi di SQL e Hadoop, cioè la facilità di eseguire query con le performance e la flessibilità di Hadoop, arrivando a fornire alte prestazioni e bassa latenza. Impala è scritto in Java e C++ ed è stato pensato per integrarsi con altri componenti dell'ecosistema Hadoop, come YARN e HDFS. Inoltre presenta la caratteristica peculiare di indipendenza dal livello di archiviazione sottostante. Impala si basa su tre demoni, come descritto in figura 3.19:

Demone Impala (impalad): gestisce la ricezione e l'esecuzione delle query sul cluster. Viene eseguito su ogni nodo del cluster su cui sta girando il DataNode che contiene i dati, in modo da sfruttare la località dei dati, potendo leggerli direttamente dal file system, senza passare dalla rete. Nel momento in cui esegue la query, assume il ruolo di *coordinator*, anche se il sistema è pensato affinché tutti i demoni possano svolgere tutti i ruoli, con il conseguente vantaggio di un efficiente bilanciamento del carico e di una semplice sostituzione in caso di fallimento.

Demone Statestore (statestored): gestisce i metadati e del loro invio ai processi interessati dall'esecuzione, oltre a verificare lo stato dei Demoni Impala. Controlla infatti la presenza di eventuali guasti, nel qual caso si occupa di informare gli altri Demoni Impala, in modo che considerino il nodo offline e quindi non disponibile per l'esecuzione di query.

Demone Catalog (catalogd): risiede nella stessa macchina del Demone Statestore, che gli richiede di inviare i metadati ai Demoni Impala, nel caso in cui questi vengano modificati da una query.

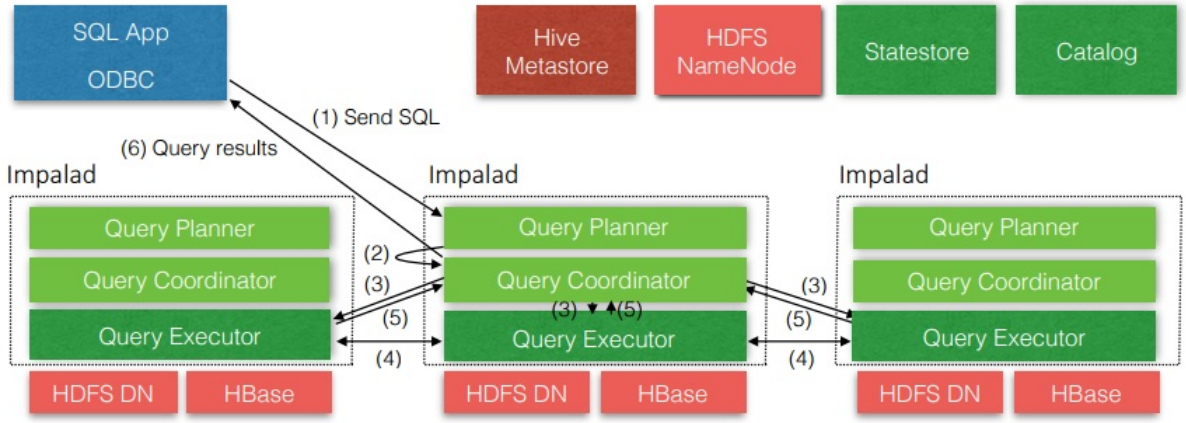

Figura 3.19: Architettura Impala

L'idea alla base di Impala è quella di ottimizzare le query SQL su Hadoop, in modo da renderle rapide.

Impala sfrutta l'infrastruttura creata da Hive ed usa il suo stesso Metastore, ma a differenza di quest'ultimo, tenta di ridurne gli accessi, eseguendo la cache dei metadati sui Demoni Impala distribuiti sul cluster. Le query vengono convertite in job MapReduce, esattamente come in Hive, ma questo lavoro è eseguito in memoria, portando tempi di risposta più bassi.

3.9 Apache Spark

Figura 3.20: Logo Spark

Apache Spark è un framework distribuito di tipo opensource general-purpose sviluppato dall'UC Berkeley nel 2009 e successivamente donato all'Apache Software Foundation nel 2013. Viene definito dagli sviluppatori stessi come un motore veloce e generico per l'elaborazione di dati su larga scala [10].

Spark dunque, così come Hadoop MapReduce, si pone l'obiettivo di processare dati, ma adottando un approccio sostanzialmente diverso. Come abbiamo visto, il limite principale di MapReduce sono infatti le frequenti operazioni di lettura e scrittura su disco, che ne abbassano le performance, specie in caso di job iterativi o multipli.
L'idea su cui si basa Spark, dunque, è quella di sfruttare i dischi solo quando necessario, e contestualmente, di tenere i dati sulla memoria principale, di sua natura più veloce e il cui costo negli ultimi anni ha subito un decremento. Di conseguenza, la velocità di elaborazione fra i due fra-

mework differisce in modo significativo: Spark può essere fino a 100 volte più veloce. D'altro canto, è diverso anche il volume di dati processabili, in quanto MapReduce è in grado di lavorare con serie di dati molto più grandi di Spark. Spark, dunque è stato progettato come un sistema di elaborazione distributita che:

- Fosse in grado di eseguire sequenze complete di operazioni, anche iterative, in un singolo job, con conseguente riduzione della complessità del codice applicativo.
- Permettesse anche analisi avanzate. Spark infatti non supporta solo MapReduce, ma anche query SQL, data streaming, machine learning e algoritmi basati sui grafi.
- Desse la possibilità di tenere i dati in memoria fino al termine del processo, evitando così costose operazioni di I/O.
- Supportasse diversi linguaggi, come Java, Python, R e Scala.
- Offrisse la stessa capacità di Hadoop di resistere ad eventuali guasti dei nodi del cluster.
- Sfruttasse la località dei dati, assegnando l'elaborazione di un blocco di informazioni allo stesso nodo in cui erano immagazzinati.

3.9.1 Principi Operativi

Un'applicazione Spark viene eseguita come un insieme di processi indipendenti sul cluster, coordinati dall'oggetto *SparkContext* contenuto nel programma principale, chiamato *Driver Program*. Lo SparkContext ha il compito di connettersi al *Cluster Manager*, a cui spetta l'allocazione delle risorse.
Una volta connesso, lo SparkContext avvia gli *Executor*, cioè processi responsabili dello svolgimento delle operazioni, all'interno dei *Worker Node*. Ogni processo Executor, uno per ogni core del processore, è in realtà una Java Virtual Machine, a cui viene inviato il codice del programma (contenuto in un JAR) e i task che dovrà eseguire.

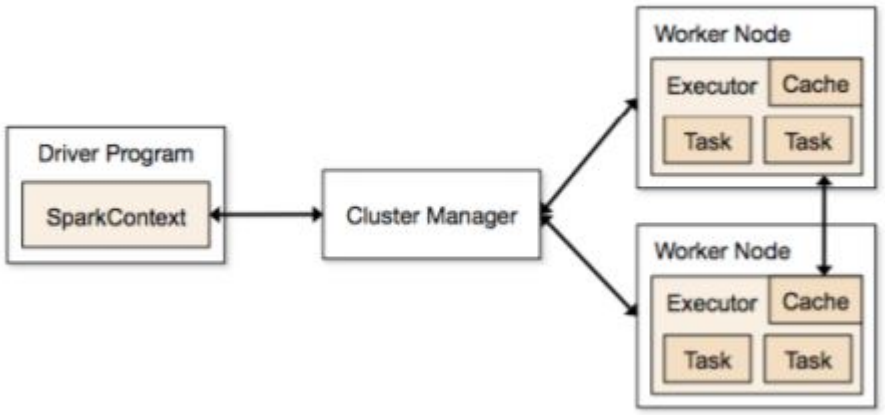

Figura 3.21: Interazioni fra Driver, Cluster Manager e Worker Node

3.9.2 Architettura

Da un punto di vista architetturale, Spark è formato da più componenti, come mostrato in figura 3.22:

Spark Core: è il cuore di Spark e contiene componenti per il task scheduling, per la gestione della memoria, per il recupero in caso di fallimenti e le principali strutture di astrazione dei dati *(RDD)*.

Spark SQL: permette di eseguire query sui dati in linguaggio SQL. Offre inoltre la possibilità di lettura da sorgenti di dati diversificate (JSON, tabelle Hive, Parquet, ...) e di combinare le query SQL con le operazioni di manipolazione degli RDD, eseguibile in diversi linguaggi (Scala, Python, Java).

Spark Streaming: è il componente per il processamento real-time di flussi di dati. Fornisce API per gestirli, usando le componenti base di Spark Core (RDD). É inoltre progettato per offrire la stessa fault tolerance, lo stesso throughput e scalabilità di Spark Core.

GraphX: è la libreria per la gestione dei grafi. Permette computazioni altamente parallelizzabile, di creare grafi a partire dagli RDD e una serie di algoritmi specifici per gestirli.

MLib: Offre le funzionalità più comuni di machine learning e diversi algoritmi di apprendimento automatico (classificazione, regressione, clustering, ...). Tutte le funzioni sono state progettate per distribuire il calcolo sui nodi del cluster.

Standalone Spark Scheduler: si tratta di uno scheduler locale, usato per eseguire le applicazioni in locale o su un cluster.

YARN: è il Resource Manager contenuto in Hadoop 2.0.

Mesos: è Cluster Manager generico che può anche eseguire MapReduce e applicazioni di servizio.

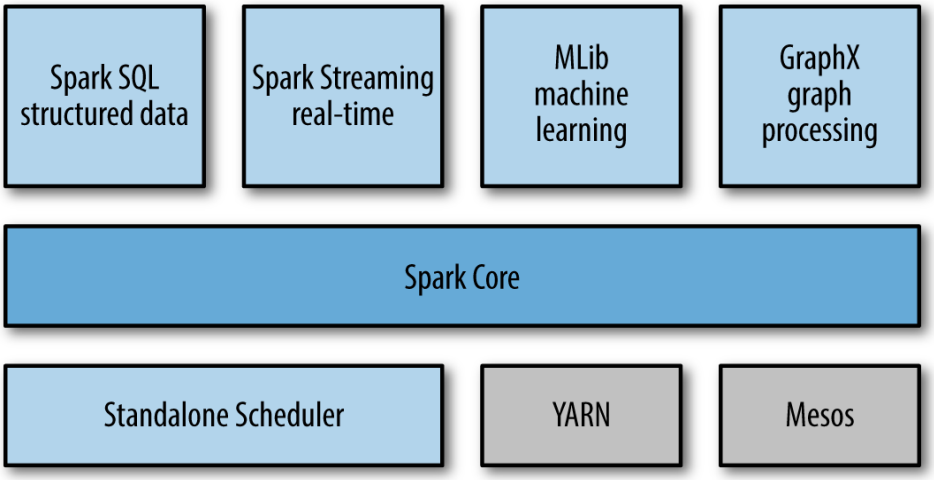

Figura 3.22: Architettura Spark

3.9.3 Resilient Distributed Datasets

Gli RDD, Resilient Distributed Datasets, costituiscono la struttura dati fondamentale di Spark. Ogni RDD è una collezione immutabile distribuita di oggetti di diverso tipo, anche di classi definite dall'utente. Un RDD può essere creato a partire da un dataset esterno (JSON, HDFS, MySQL, CSV,...) o come risultato di manipolazioni di un'applicazione Spark. Ogni dataset in RDD è diviso in partizioni logiche, che possono essere poi computate su diversi nodi del cluster.
Una volta che il dato è stato memorizzato su un RDD, su di esso saranno possibile due operazioni:

Trasformazioni: permettono la creazione di un nuovo RDD da un RDD precedente, attraverso operazioni come mapping, filtering e altre. Per le trasformazioni si parla di *Lazy Evaluation*: per questioni di performance, non vengono eseguite fino a che non viene eseguita un'azione.

Azioni: restituiscono al Driver un risultato calcolato a partire da un RDD, o lo scrivono su disco.

Capitolo 4

STATO DELL'ARTE

In questo capitolo verrà riportata una panoramica dell'architettura già presente all'inizio del mio lavoro. Questa è stata pensata innanzitutto per ricevere i dati provenienti dalle box telematiche montate sui veicoli. Si procede quindi con la pulizia degli stessi, attraverso la verifica che rispondano agli standard prefissati a monte, alla fase di analisi e aggregazione, per arrivare ad esempio a produrre report settimanali sui singoli veicoli o ad effettuare analisi sull'efficienza dei consumi di carburante, utilizzando un'infrastruttura Hadoop On-Premise.

4.1 Architettura Logica

Il progetto fu commissionato nel 2015 da un cliente di *Data Reply*, leader in ambito automotive, e prevedeva la ricezione e successiva gestione di diversi flussi di dati, in costante aumento nel tempo. I messaggi, serializzati in formato XML, venivano inviati periodicamente dalle board e potevano contenere informazioni di vario genere: dalle posizioni gps dei veicoli durante il percorso effettuato fra Key-On e Key-Off, a dati anagrafici dei driver, a indicazioni su velocità e consumi di carburante.

L'immagine 4.1 mostra la prima implementazione della soluzione e, oltre a descrivere l'architettura logica, permette di capirne i principi operativi fondamentali: le sorgenti, simili a dispositivi

IoT, prevedono al loro interno un client Web Service, capace di comunicare col punto di accesso del layer di ingestion. I dispositivi, periodicamente, inviano messaggi al WebServer che, dopo averli processati, decomprimendoli, validandoli tramite wsdl, ed eseguendone il parsing, smista le informazioni su diversi topic Kafka. Questi topic sono collegati ad un agente Flume che consuma i dati e li scrive su HDFS. Delle tabelle inoltre vengono create sulle cartelle di destinazione e possono essere interrogate attraverso Hive ed Impala.

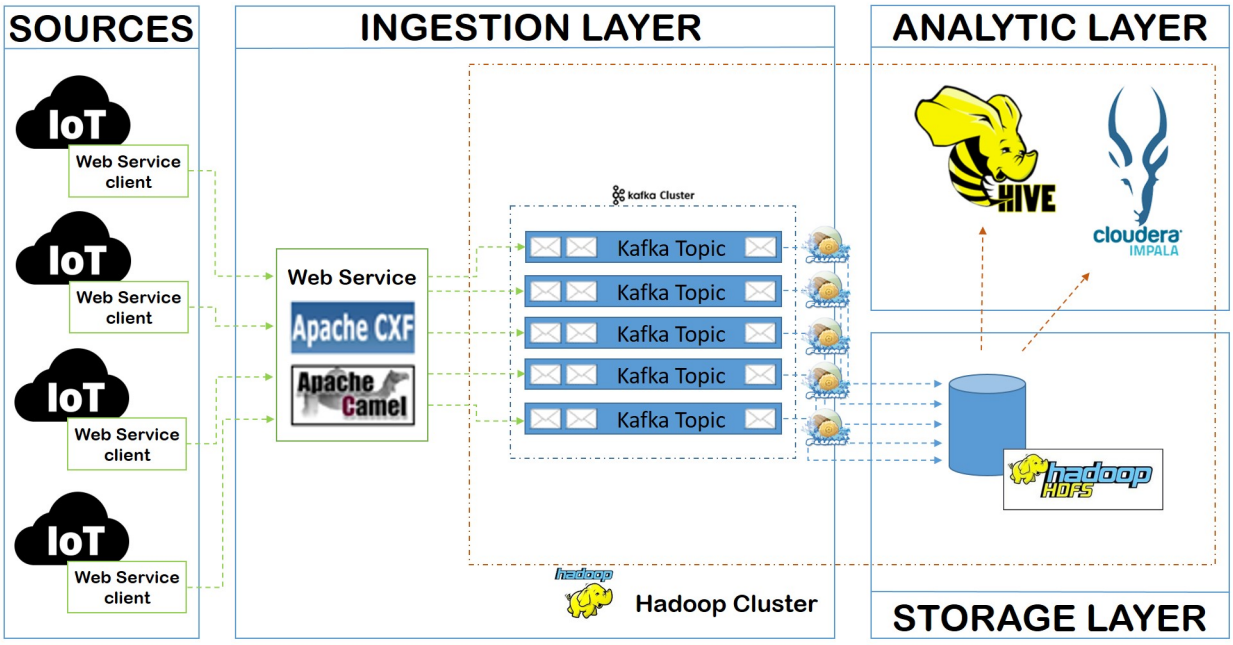

Figura 4.1: Prima Implementazione

Al fine di migliorare le performance dell'intero sistema, ed eliminare il *single point of failure* rappresentato dal Web Service, a partire dalla seconda metà del 2017 sono state effettuate delle modifiche, sia architetturali che logiche. Lo spazio sulle macchine fisiche era terminato e di conseguenza sono state aggiunte delle macchine virtuali. Su di esse sono stati spostati i Web Services di ingestion. Inoltre non è più il Web Service che si occupa di processare i messaggi, ma si limita a decomprimerli, per poi inviare il risultato a dei topic Kafka. A quel punto, delle applicazioni basate sulle API Kafka Streams hanno il compito di processare i payload e ridirigere i record risultanti sui

topic Kafka appropriati. Infine, grazie all'utilizzo del tool *Streamsets*, i record vengono consumati dalle code Kafka e scritti su HDFS e su *Kudu*, un archivio dati column-oriented appartenente all'ecosistema Apache che si è dimostrato particolarmente adatto alla situazione in termini di tempi di risposta.

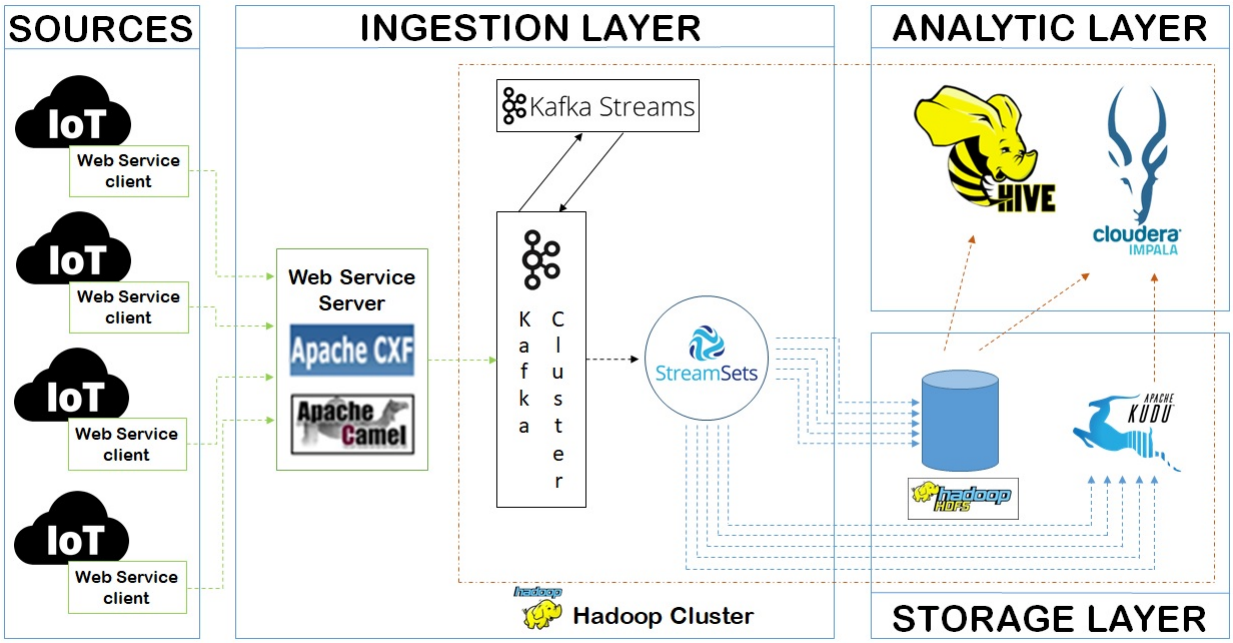

Figura 4.2: Seconda Implementazione

4.1.1 StreamSets

Figura 4.3: Logo StreamSets

Uno dei componenti introdotti nella seconda implementazione, che rivedremo anche nel passaggio dall'architettura On-Premise a quella Cloud, è StreamSets.

Si tratta di una raccolta di prodotti atti a controllare flussi di dati. StreamSet offre essenzialmente due prodotti: il *Data Collector* e il *Dataflow Performance Manager*. Nella soluzione qui riportata è stato utilizzato soltanto il primo dei due.
Il Data Collector è un'applicazione open source che consente agli utenti di creare pipeline di dati indipendenti dalla piattaforma. Il sistema è pensato per eseguire in maniera ottimizzata l'ingestione dei continua di dati senza latenza.
Una pipeline descrive il flusso di dati dal sistema di origine a quello di destinazione e definisce le trasformazioni dei dati lungo il percorso. Una pipeline è composta da fasi, che possono essere di tre tipi differenti:

Origine: rappresenta la sorgente. Per ogni pipeline è concessa una sola origine.

Processore: si tratta del tipo di elaborazione dei dati che si desidera eseguire. É possibile inserire all'interno della pipeline un numero variabile di elementi di questo tipo, in base alle esigenze.

Destinazione: rappresenta il target della pipeline. Anche in questo caso si può inserire una o più destinazioni nella stessa pipeline.

Costruire una pipeline è semplice ed intuitivo: basta trascinare e rilasciare i componenti necessari nell'area di lavoro, collegarli tramite frecce ed impostare pochi parametri. Il Data Collector permette inoltre, tramite interfaccia grafica, di creare e monitorare flussi di dati batch e streaming.

4.2 Ambiente

La soluzione venne sviluppata su un cluster formato da tre macchine fisiche e due macchine virtuali, con le caratteristiche mostrate in tabella 4.1.

La configurazione dei nodi segue il modello master-slave di Hadoop. L'HDFS è configurato in modalità *High Availability* con l'host01 come istanza attiva di NameNode e il nodo host02 in

HOST	RUOLO	TIPO	CPU	RAM	STORAGE
host01	Management	Virtual	8	16 GB	300 GB
host02	Management	Virtual	8	16 GB	300 GB
host03	Worker	Physical	40	189 GB	1 TB + 10 x 3,6 TB
host04	Worker	Physical	40	189 GB	1 TB + 10 x 3,6 TB
host05	Worker	Physical	40	189 GB	1 TB + 10 x 3,6 TB

Tabella 4.1: Configurazione del Cluster

standby.

La capacità totale del file system è di circa 110 TB, distribuita su tre DataNode. Ognuno di essi ha 1TB di hard disk progettato per ospitare il file system locale del nodo e 3.7TB di disco dedicati all'HDFS.

La comunicazione verso l'esterno avviene tramite interfacce a 1Gbps, mentre la rete interna supporta fino a 10Gbps.

L'interfaccia di cluster management è disponibile, tramite il *Cloudera Manager* e permette di monitorare lo stato dei nodi.

Infine, un'altra macchina, situata fuori dal cluster Hadoop, viene usata come web server ed è una piccola macchina virtuale Linux di 8 core e 8Gb di memoria.

4.3 Perché passare al Cloud?

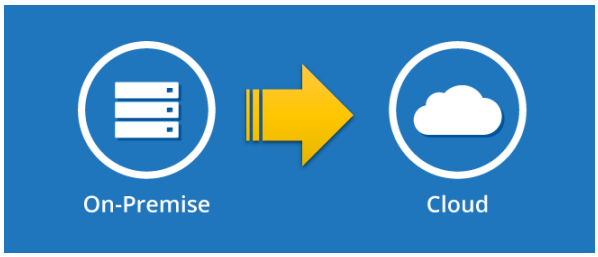

Figura 4.4: Dall'On-Premise al Cloud

Le ragioni che hanno portato l'azienda, su commissione del cliente, a passare da una soluzione On-Premise al Cloud ed in particolare al Cloud Azure Microsoft sono state molteplici. Il tutto è nato dall'evoluzione dell'intera piattaforma del cliente, che comprendeva cambiamenti radicali atti a svecchiare il sistema, e dalla necessità di dover gestire una nuova box telematica che verrà montata sui nuovi veicoli in vendita nel 2019. La nuova piattaforma, oltre a gestire i veicoli dotati della box ad oggi in commercio, deve quindi prevedere la gestione dei nuovi dispositivi di telematica avanzata e la trasmissione di una mole di dati superiore rispetto al passato. Inoltre, i problemi derivanti dalla gestione di un'architettura On-Premise erano i seguenti:

Infrastruttura Monolitica: la soluzione pregressa era basata su un Web Service monolitico e ciò rendeva lo scenario non scalabile e difficilmente gestibile. Avendo inoltre in carico la gestione di decine di flussi, il codice era complesso ed intricato e di conseguenza anche aggiungere la gestione di nuovi flussi costituiva un'operazione molto delicata e non facile da implementare. Inoltre, avere un unico Web Service significava avere un unico *point of failure*: un malfunzionamento del sistema legato ad un singolo flusso poteva avere ripercussioni sulla ricezione e la gestione di tutti gli altri.

Performance: l'aspetto prestazionale era strettamente vincolato dalla dimensione della macchina. Il Cluster era allo stesso tempo *under provisioned* per task molto onerosi, e *over provisioned* nel resto del giorno, nei momenti in cui era relativamente più scarico e quindi non sfruttato completamente. In conclusione, l'ambiente doveva essere tarato in base ai picchi, per essere sicuri di avere risorse sufficienti a gestire tutti i flussi, ma in questo modo inevitabilmente si aveva uno spreco in alcune fasi delle giornate.

Manutenzione Software: installare nuovi componenti richiedeva sforzi eccessivi, competenze tecniche e molta attenzione, in modo da non andare ad intaccare il funzionamento di altre funzionalità preesistenti. Anche l'aggiornamento dei sistemi era un'operazione delicata: per eseguirli, era necessario spegnere il cluster preventivamente, per poi riaccenderli dopo l'upgrade, con conseguente spreco di tempo durante tutto il processo. Inoltre, nel caso di

aggiunte di nuove macchine, spesso era necessario installare software datato, per questioni di compatibilità con gli altri nodi.

Manutenzione Hardware: i nodi che componevano il cluster cominciavano a presentare forti problemi di usura. Si rendeva quindi spesso necessario richiedere sostituzioni di alcuni componenti, ma questo implicava dover attendere anche una/due settimane, con conseguenti forti disservizi.

Costi: il costo del mantenimento dell'architettura, dovuto, oltre che al consumo ininterrotto di energia, anche al monitoraggio dei sistemi, era troppo alto e vi era la necessità di abbatterlo.

Capitolo 5

SOLUZIONI SCALABILI PER LA GESTIONE DEI DATI SU CLOUD

In questo capitolo verranno illustrate le fasi di redirezione dei flussi di dati da on-premise a cloud e l'implementazione di nuove soluzioni atte a gestire l'arrivo di nuovi dati provenienti dalle nuove box telematiche direttamente sul cloud.
Durante la descrizione, verranno via via introdotti gli strumenti principali utilizzati nelle varie fasi, con un focus particolare sui componenti di *Microsoft Azure* adoperati.

5.1 Da On-Premise a Cloud

La fase iniziale della transizione da ambiente On-Premise a Cloud ha seguito il modello del *Lift-And-Shift*, letteralmente "solleva e sposta", che consiste nello sfruttare i vecchi sistemi, integrandoli ad una strategia che permetta di portare i dati anche su Cloud. In particolare, è stata implementata un'architettura che alimentasse simultaneamente e in tempo reale sia la componente On-Prem, sia quella Cloud, senza impatti sulla prima, in modo da poter continuare a memorizzati i dati su HDFS e, contestualmente, iniziare a testare i servizi di archiviazione del Cloud.

Nel prossimo paragrafo verranno introdotti i componenti fin qui mai menzionati che sono coinvolti nel processo di biforcazione dei dati. Successivamente verrà approfondita l'architettura e descritto il dettaglio realizzativo.

5.1.1 Componenti Utilizzati

5.1.1.1 Azure Data Lake

Azure Data Lake è un servizio scalabile di archiviazione e analisi dei dati. In esso è possibile archiviare dati strutturati, semi-strutturati o non strutturati prodotti da applicazioni quali social network, sensori, video, app Web, dispositivi mobili o desktop. Un singolo account Azure Data Lake Store ha una capacità di archiviazione potenzialmente illimitata e ogni singolo file può avere dimensione maggiore di un petabyte.
Il Data Lake propaga parti di file su singoli server di archiviazione, rendendo alta la velocità effettiva durante la lettura in parallelo dei file. Ciò rende lo strumento particolarmente adatto alle analisi di grosse moli di dati.

5.1.1.2 Azure SQL

Microsoft Azure SQL Database è un servizio SaaS offerto dalla piattaforma cloud Azure. É quindi un servizio gestito che offre scalabilità, backup e disponibilità dei dati. Include inoltre servizi di intelligenza integrata per riconoscere pattern applicativi e massimizzare le performance, l'affidabilità e la protezione dei dati.
Per gestire il database, all'utente viene data la possibilità di usare tool come cheetah, sql-cli, VS Code e strumenti Microsoft come Visual Studio, SQL Server Management Studio, PowerShell, API REST o direttamente il portale Azure.

5.1.1.3 Azure DataBricks

DataBricks è una piattaforma gestita pensata per eseguire applicazioni Apache spark. Ciò significa che non è necessario dover apprendere concetti complessi riguardante la gestione dei cluster, né

eseguire attività di manutenzione per sfruttare Spark.
La piattaforma, per soddisfare le esigenze delle aziende, implementa la funzionalità del controllo degli accessi basasto sui ruoli e altre ottimizzazioni che migliorano l'usabilità per gli utenti e riducono anche costi e complessità per gli amministratori.
Per il programmatore è possibile scrivere il proprio codice in Scala, Python, R e SQL.
In figura 5.1 è riassunto ciò che costituisce il vero e proprio core di Databricks:

Cluster: sono un set di macchine virtuali Linux che ospitano il Driver Spark e i nodi worker, su cui gireranno le applicazioni. Attraverso un'interfaccia intuitiva, è possibile creare cluster di tipo *Standard* e *Serverless Pool*, specificando il numero di nodi, le politiche di autoscaling e di autoterminazione, i tipi di istanze di macchina virtuale per i nodi e altre configurazioni relative a Spark. Nel caso in cui si scelga l'opzione Serverless Pool, è possibile specificare il massimo e il minimo numero di nodi nel cluster e sarà la piattaforma stessa a scalare in quel range in base all'utilizzo effettivo.

Notebook: attraverso essi le applicazioni Spark possono essere eseguite direttamente sul cluster. É possibile configurare gli accessi, per condividere o meno il codice con colleghi, e all'interno di essi è possibile mixare più linguaggi di programmazione.

Librerie: sono containers atti ad ospitare le librerie Python, R, Java/Scala dell'utente. Dopo aver importato il codice sorgente, risiedono all'interno del workspace e possono essere cancellate o sovrascritte in qualsiasi momento.

Job: il codice Spark viene inviato come "Job" per essere eseguito dai cluster Databricks. La piattaforma offre strumenti grafici per creare, gestire e monitorare i Job che sono in esecuzione. Creando un nuovo job è necessario specificare il Notebook o il Jar da eseguire, il cluster e la politica di scheduling; sono inoltre settabili altre configurazioni opzionali come il timeout o la politica di retry.

Workspace: ogni utente ha a disposizione uno spazio in cui organizzare i propri notebook, le proprie librerie e dashboard. L'organizzazione è simile a quella delle cartelle, gerarchica, con la possibilità di settarne la visibilità agli altri utenti.

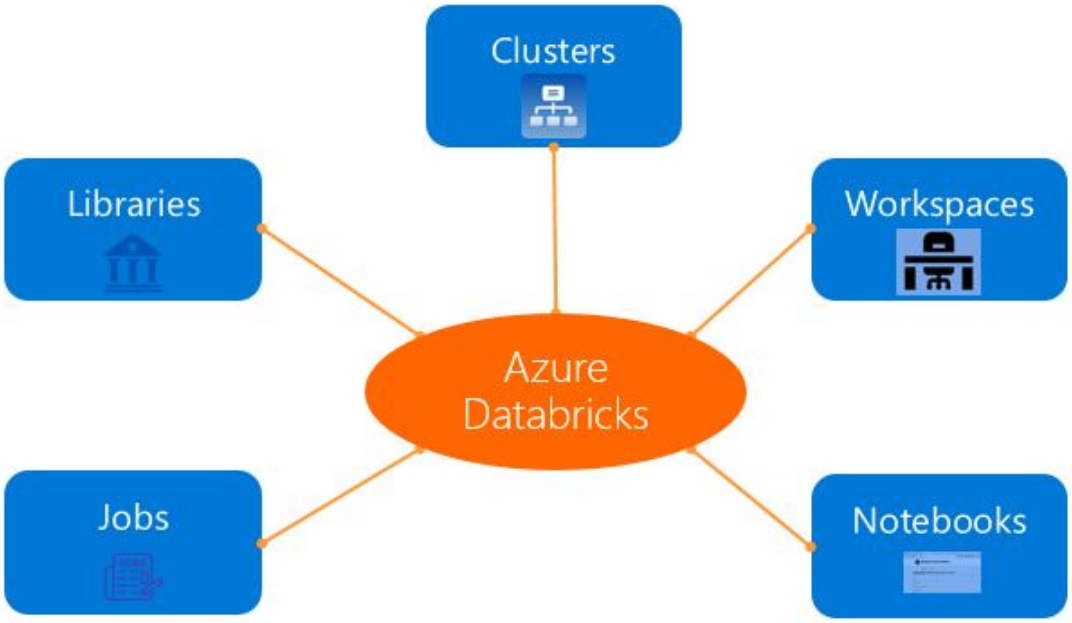

Figura 5.1: Core di Databricks

Fra le caratteristiche risultate più utili a questo lavoro di tesi, è bene menzionare anche l'integrazione con i servizi Azure sopra elencati. Oltre a poter accedere ai dati in modo semplice, infatti, è possibile agganciare gli Azure Storage al file system Databricks (DBFS) tramite la semplice operazione di *dbutils.fs.mount()*. L'utente in questo modo può quindi vedere i dati proprio come se fossero sottocartelle del file system.

5.1.2 Architettura Logica

La figura 5.2 mostra lo schema logico della soluzione.

Nella sezione inferiore dell'immagine è presente la vecchia architettura, con i dati che partono dalle varie sorgenti, vengono raccolti dal Web Service e smistati su diversi topic Kafka. Da lì

vengono infine scritti su HDFS e da quel momento in poi sono pronti per essere analizzati tramite query Hive o Impala.

La sezione superiore, delimitata dal tratteggio verde, indica invece l'implementazione delle nuove funzionalità on-cloud. Avendo strutturato il lato On-Premise usufruendo di tecnologie come Kafka, di tipo *publisher/subscriber*, è stato aggiunto un nuovo consumer, cioè *StreamSets*, che permettesse di alimentare la componente Cloud, senza impatti sulla struttura pre-esistente. I dati presenti sui topic Kafka vengono consumati da Streamsets, che li redirige su un'istanza di Azure Data Lake facente parte della sottoscrizione, sotto un path partizionato per anno, mese e giorno in cui si è svolto il processo di ingestion.

Una volta memorizzati sul Data Lake sotto forma di CSV, i dati vengono riprocessati: viene effettuato il parsing dei payload, sulla base del formato definito nelle specifiche. I dati vengono quindi trasformati, riscritti su Data Lake sempre in formato CSV e su un'istanza di Database SQL, tramite applicazioni Spark sviluppate sulla piattaforma Databricks. In particolare, dopo il parsing, i record ricevuti vengono sottoposti a processi di Data Quality, con lo scopo di verificare che i valori dei campi siano semanticamente corretti e conformi alle specifiche. Ad esempio, si considerino dei messaggi relativi al percorso svolto da un veicolo: occorre verificare che il timestamp finale sia superiore a quello iniziale, o che il consumo di carburante ragionevolmente proporzionale alla lunghezza della tratta.

All'utente infine viene data la possibilità di analizzare in maniera semplice ed efficace i dati tramite un layer di visualizzazione, che comprende dashboard e report sviluppati in Power BI, e un'interfaccia web che permette di eseguire interrogazioni più complesse.

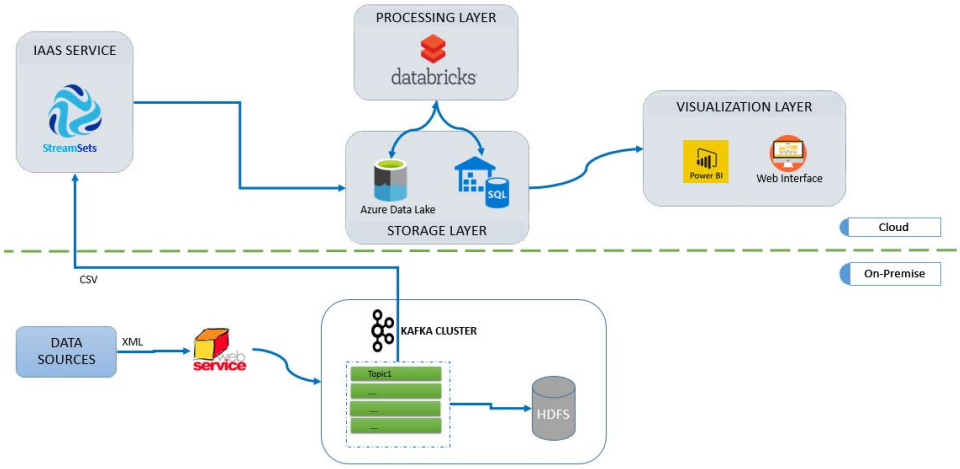

Figura 5.2: Redirezione dei flussi

5.1.3 Implementazione

Nella figura 5.3 mostra il modo in cui sono composte le Pipeline StreamSets che hanno il compito di portare i dati sul Cloud.

Il primo blocco è un Consumer Kafka, che, connesso al Cluster Kafka presente On-Premise, legge e consuma i topic corrispondenti ai flussi di interesse. Di seguito troviamo uno Stream Selector, che è configurato in modo da discernere, in base ai metadati ricevuti, il flusso sotto analisi e quindi, di conseguenza, capace di individuare il path di uscita sul Data Lake. I dati non riconosciuti, o comunque non appartenenti ai flussi gestiti, vengono scartati (blocco Trash).

La difficoltà maggiore avuta nell'implementare fisicamente questa architettura è stata l'incomunicabilità dell'ambiente On-Premise e quello Cloud.

Per ragioni di sicurezza, infatti, la sottoscrizione Azure creata dal cliente per gestire tutti i progetti commissionati, era protetta da una Virtual Network che poneva vincoli sulle comunicazioni con l'esterno.

La soluzione trovata per fronteggiare questo problema è stata quella di porre il servizio IaaS StreamSets, cruciale per poter agganciarsi ai flussi Kafka, su una macchina virtuale Linux posta all'interno del Cloud.

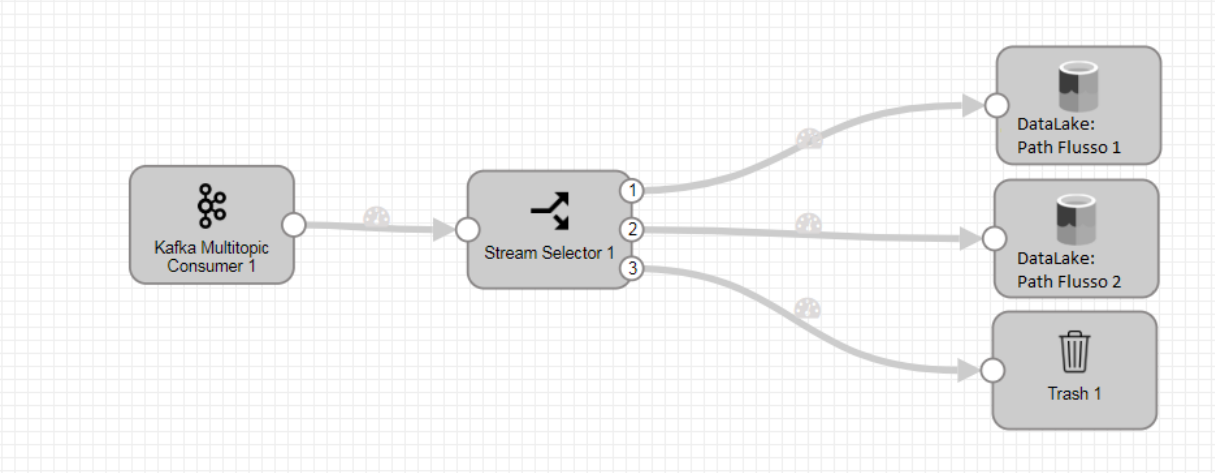

Figura 5.3: Esempio di Pipeline StreamSets

La macchina, opportunamente settata, ha svolto la funzione di ”ponte” fra i due mondi, permettendo l'atterraggio dei dati sul Data Lake.

5.1.3.1 Future Implementazioni

L'utilizzo della *macchina-ponte* costituisce di fatto una soluzione tampone, un *workaround* che ci ha permesso di implementare la connessione di servizi On-Premise e PaaS, in attesa che venga attivato un altro componente messo a dispozione da Azure, l'*ExpressRoute*.

Questo permette infatti di stabilire connessioni private tra i data center di Azure e l'infrastruttura presente in locale o in un ambiente con più sedi.
Non sfruttando la rete Internet pubblica, le connessioni ExpressRoute inoltre garantiscono un livello di sicurezza molto alto, affidabilità, velocità più elevate e minori latenze rispetto alle connessioni Internet tradizionali.
Di fatto quindi nei prossimi mesi saremo in grado di passare dalla soluzione Streamset IaaS a quella PaaS, per sfruttare così i servizi esposti come piattaforma direttamente da Azure, senza dover più gestire ulteriori infrastrutture.

5.2 Gestione dei nuovi dati

Oltre a continuare a gestire i vecchi messaggi, facendoli atterrare sul Cloud, è stato necessario iniziare a gestire, utilizzando diverse tecnologie in base alla natura dei dati stessi, le informazioni emesse dalle nuove box telematiche del cliente.
In questo paragrafo scenderemo nel dettaglio implementativo della gestione di questi nuovi flussi, discriminando un modello di analisi in micro-batching, e uno invece real-time.

5.2.1 Applicazioni Micro-Batch

Alcuni tipi di informazioni inviati dalle box sono stati pensati a monte come messaggi: si tratta di record di dimensione contenuta, organizzati in piccoli pacchetti avro, caratterizzati da un header e un payload, che atterrano su un Azure Blob Storage.

La natura di queste informazioni, che non sono pensate per dare in futuro risposte immediate all'utente, ma serviranno perlopiù per report settimanali/mensili, ha portato alla decisione di processarli ogni 10 minuti tramite un applicativo basato su Spark e alla fine di scriverli su un Azure Data Lake. Come prima cosa, introduciamo i componenti Azure fino ad ora non menzionati che sono coinvolti nel processo, al fine di comprendere al meglio il suo funzionamento.

5.2.1.1 Componenti Utilizzati

Azure IoT Hub: si tratta di un servizio cloud pensato per gestire la comunicazione bidirezionale fra applicazioni IoT e dispositivi. Supporta più modelli di messaggistica, dai dati di telemetria al caricamento di file dai dispositivi. Il servizio offre inoltre la possibilità di monitorale la soluzione tramite il rilevamento di eventi, come la creazione di un dispositivo, eventuali errori o nuove connessioni.
L'IoT Hub consente quindi di creare un canale di comunicazione sicuro per i dispositivi che vogliono inviare dati, offrendo diversi tipi di autenticazione.

Dal punto di vista della programmazione, supporta linguaggi diversi (C#, C, Java, Python, Node.js) e sistemi operativi diversi, da Windows a più distribuzioni Linux.

Azure Blob Storage: fra i suoi servizi, Azure offre la possibilità di archiviare dati in modi differenti. L'Azure Blob Storage ha proprio questo obiettivo: si tratta di un archivio generico di oggetti, utilizzabile per vari scopi, fra cui le analisi Big Data.

I dati che possono essere archiviati spaziano da formati testuali a binari, da dati di backup ad altri di utilizzo generico.

Dal punto di vista strutturale, ogni account di archiviazione include *containers*, che a loro volta contengono i dati sotto forma di BLOB. Anche in questo caso, dal punto di vista della programmazione, sono previste SDK per diversi linguaggi, da .NET a Java, da Python a C++, Android e iOS.

A differenza di quanto accade con i Data Lake, tuttavia, la piattaforma prevede per i Blob Storage alcuni limiti di default [11].

Il costo per transazioni suggerisce il Blob Storage come soluzione adatta se si pensa di dover solo accumulare dati non strutturati con il requisito di un recupero frequente e veloce. Qualora invece si debbano eseguire analisi sui dati, risulta più conveniente utilizzare l'Azure Data Lake Store.

Azure CosmosDB: si tratta del database multimodello distribuito a livello globale di Microsoft, lanciato nel 2017.

Può supportare più modelli di dati utilizzando un unico back-end. Ciò significa che può essere utilizzato per diversi modelli, da documenti, a coppie chiave-valore, a relazionali e grafici. É considerabile alla stregua di un database NoSQL perché non si basa su alcuno schema. Tuttavia, poiché utilizza un linguaggio di query simile a SQL e può facilmente supportare le transazioni ACID, alcune persone lo hanno classificato come un tipo di database *NewSQL*.

Date le sue caratteristiche, è particolarmente indicato per le aziende che hanno bisogno di un database scalabile e distribuito globalmente. Ciò significa infatti che tutte le risorse sono

suddivise orizzontalmente in ogni regione del mondo e replicate in diverse aree geografiche. Questo implica una latenza minima e un'esperienza veloce e fluida per gli utenti che lo utilizzano.

Infine, una della caratteristiche più interessanti del servizio è che è Multi-API: poiché i dati vengono indicizzati automaticamente, l'utente può accedervi usando l'API che preferisce. I dati infatti sono visibili usando SQL, MongoDB, Cassandra, Gremlin, Javascript, Azure Table Storage.

Azure Data Factory: è un servizio pensato per consentire agli sviluppatori di integrare fonti di dati diverse. Si tratta di una piattaforma che permette di creare pipeline di attività, grazie ad una semplice interfaccia, trascinando i componenti di interesse.

In tal modo si può arrivare a creare complessi progetti di estrazione, trasformazione e caricamento (ETL) e di integrazione dei dati.

Per creare il proprio flusso di esecuzione all'utente basta selezionare i vari componenti a disposizione, settando poche configurazioni, per poi schedularne l'esecuzione.

Data Factory offre inoltre una dashboard di monitoraggio aggiornata, il che vuol dire che è possibile implementare le proprie pipeline di dati e iniziare immediatamente a visualizzarle come parte della dashboard di monitoraggio.

Microsoft Power BI: è un servizio di analisi fornito da Microsoft. Fornisce visualizzazioni interattive con funzionalità di business intelligence, per cui anche utenti inesperti possono creare dashboard e report da soli. Grazie a diversi connettori, in costante aumento, è possibile accedere a sorgenti di dati disparate, dai Data Lake a database SQL, utilizzando diversi formati di dati, dai json ai csv a tabelle.

L'utente può dunque creare grafici, tabelle derivate, e altri elementi di proprio interesse ed ha anche la possibilità di condividere i propri prodotti con altri utenti o in maniera pubblica, su Internet.

5.2.1.2 Architettura Logica

La figura 5.4 mostra la struttura logica della soluzione: a monte del processo troviamo un IoT Hub che raccoglie i messaggi ricevuti dai dispositivi e li scrive su un Blob Storage sotto forma di file *avro*. Ogni record è contiene un header e un Body, che è in formato binario. L'intestazione invece è composta da tre campi:

EnqueuedTimeUTC: è il timestamp in cui è partito il messaggio dalla board.

Properties: è un campo di tipo *map(string,string)*, che contiene diverse indicazioni, come il tipo di flusso, alcuni flag che discriminano il comportamento dell'applicazione, un identificatore del veicolo e così via.

SystemProperties: anche in questo caso si tratta di un campo di tipo *map(string,string)*, che contiene un ID del messaggio e altre informazioni tecniche relative alla connettività.

L'IoT Hub scrive i file in sottocartelle diverse a seconda dell'istante in cui gli arrivano i messaggi, sotto il path */anno/mese/giorno/ora/minuto*.

L'applicazione sviluppata in questo lavoro di tesi, scritta in PySpark, ogni dieci minuti raccoglie i nuovi file scritti nell'intervallo precedente, analizza l'header dei singoli messaggi contenuti nei pacchetti avro e controlla la valorizzazione di un flag settato nelle Properties: nel caso in cui sia settato, procede, altrimenti interroga un repository ospitato su un Cosmos DB per sapere se quel singolo record debba essere scartato.
Una volta ottenuto il DataFrame finale con i record, questi vengono scritti in base al tipo di flusso a cui appartengono, sopra un Azure Data Lake, stavolta in modo partizionato per */anno/mese/giorno*. In caso di esecuzione senza errori di questa prima parte, il processo termina cancellando i file .avro analizzati. Il processo di raccolta di dati viene schedulato tramite Data Factory: un trigger infatti, è configurato per far girare ogni dieci minuti il Notebook Databricks che contiene la soluzione, passandogli come parametri il Timestamp corrente e la durata dell'intervallo da analizzare.

Un'altra applicazione scritta in Databricks e schedulata sempre tramite Data Factory, ha il compito di riprocessare i file scritti nello step precedente e in particolare di eseguirne il parsing del payload, cioè l'estrazione dei vari campi, in accordo alle specifiche tecniche fornite dai progettisti del flusso, e la loro scrittura. Questa avviene sia nuovamente su Data Lake, secondo un path opportuno, partizionato per anno e mese, sia su una o più tabelle appartenenti ad un Database SQL.

Proprio queste tabelle fungono da sorgenti per i report sviluppati in Microsoft Power BI, che contengono grafici e tabelle e il cui scopo è quello di permettere al cliente di visualizzare in modo trasparente ed intuitivo i dati.

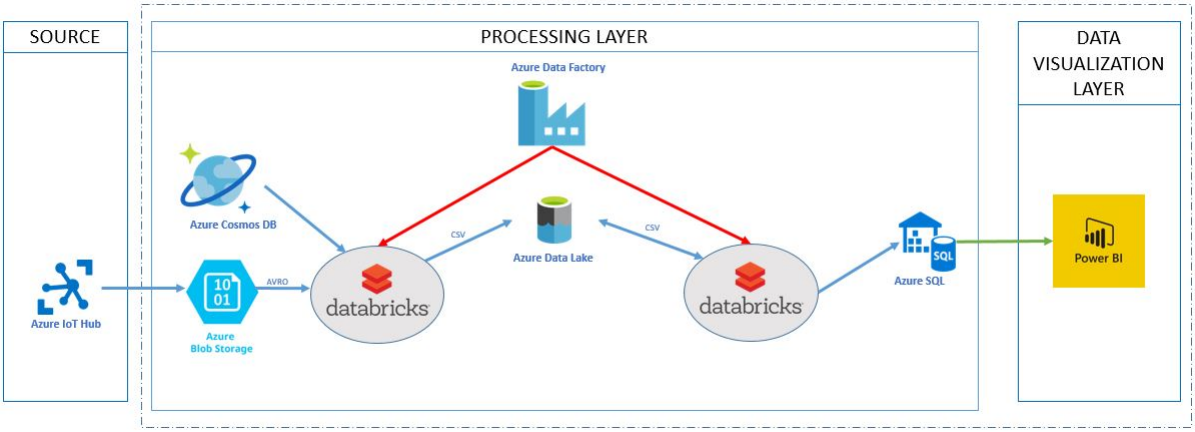

Figura 5.4: Flussi Near Realtime: Architettura Logica

5.2.1.3 Implementazione

In figura 5.5 è riportata la Pipeline creata su Data Factory per eseguire l'intero processo di ingestion.

A monte è stata inserita una operazione di Wait di cinque minuti: studiando il processo di scrittura da parte dell'IotHub infatti ci si è resi conto che a volte succedeva che la scrittura dei file iniziasse sotto un certo path, ma che questa terminasse qualche minuto dopo l'istante individuato dal path stesso. Ad esempio, un file avro poteva trovarsi nella cartella */2018/06/01/10/00* e contemporaneamente riportare come timestamp di ultima modifica 2018-06-01 10:03:00. Essendo l'intero

processo batch basato sui tempi, c'era quindi il rischio che qualche file potesse essere escluso dall'analisi.
Introducendo una wait di cinque minuti, valore basato sui settaggi dell'IoTHub, abbiamo risolto a monte il problema.
Il secondo blocco della Pipeline è rappresentato dal Notebook su cui è scritto il cuore del processo, a cui il Data Factory passa come parametri il timestamp in cui il trigger viene eseguito e il timespan che definisce l'intervallo. In caso di successo, si passa al Notebook in cui è scritta l'operazione di Delete dei file analizzati. Perché i file da cancellare corrispondano esattamente a quelli processati, a questo blocco vengono passati gli stessi parametri della fase precedente.
Infine, per far fronte ad eventuali problemi temporanei, è stata impostata una politica di retry, che fa sì che in caso di errore, venga fatto un nuovo tentativo di esecuzione dopo 30 minuti.

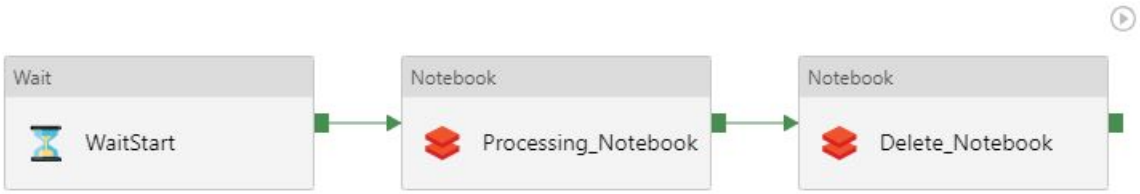

Figura 5.5: La Pipeline di ingestion su Data Factory

Vediamo ora lo scheletro del blocco centrale, riportato qui di seguito in pseudo-codice:

```
import sys
from datetime import datetime, timedelta
import time

dbutils.widgets.text("start_ts", "","")
dbutils.widgets.get("start_ts")
ts = getArgument("start_ts")
dbutils.widgets.text("timespan", "","")
dbutils.widgets.get("timespan")
diff = getArgument("timespan")
```

```
#converte in datetime
start_ts_datetime = datetime.strptime(ts[:19], "%Y-%m-%dT%H:%M:%S")

#Conversione timespan in timedelta
diff_fields = diff.split(':')
d = timedelta(days=int(diff_fields[0]), hours=int(diff_fields[1]), minutes=int
    (diff_fields[2]))

#estremo inferiore dell'intervallo (formato datetime)
end_ts_datetime = start_ts_datetime - d

#memorizza nel dataframe_avro i record dell'intervallo
dataframe_avro = read_BS(start_ts_datetime, end_ts_datetime)
NumRowFromBS = dataframe_avro.count()

#nel caso in cui ci siano record, si filtra leggendo informazioni aggiuntive
    dal CosmosDB
if NumRowFromBS > 0:
  dataframe_info_from_cosmos = read_cosmos()
  dataframe_filtered = filter_avro(dataframe_avro, dataframe_info_from_cosmos,
       start_ts_datetime)
  write_df_filtered = write_df(dataframe_filtered, start_ts_datetime)
```

Listing 5.1: Pseudocodice del Main

Il frammento di codice crea di fatto una tabella sul path, accessibile tramite query SQL e Spark SQL. Per fare inoltre in modo che la tabella si rendesse conto di nuovi inserimenti, occorreva che andasse a leggere ad ogni esecuzione del Notebook, i metadati relativi al path. Per farlo, quindi, era stata inserita l'operazione seguente all'inizio del main:

```
MSCK REPAIR TABLE Tabella;
```

Listing 5.2: MSCK Repair Table

Utilizzando questo tipo di implementazione, l'operazione di lettura dei file dal Blob Storage era dunque di questo tipo:

```
def read_BS(start_ts, end_ts):
        df_avro = sqlContext.sql("SELECT CONCAT(year,'-',month,'-',day,' ',
            hour,' :',minute,' :00') as TimeUtc FROM Tabella WHERE cast(CONCAT
            (year,'-',month,'-',day,' ',hour,' :',minute,' :00') as timestamp
            ) between cast('{0}' as timestamp) and cast('{1}' as timestamp)".
            format(end_ts,start_ts))
```

Listing 5.3: Pseudocodice della Read Blob Storage

Concatenando le parti del path costituenti anno, mese, giorno, ora e minuto, ormai divenute colonne della tabella, si poteva facilmente individuare l'intervallo di interesse, con una semplice operazione di BETWEEN. Il blocco Databricks di Delete era implementato in maniera del tutto analoga, utilizzando lo stesso sistema per individurare i file da cancellare.

Durante la fase di test dell'applicazione, grazie all'interfaccia di monitoraggio offerta da Data Factory ho notato prima un graduale degrado delle prestazioni (inizialmente l'esecuzione completa durava circa 2 minuti, mentre col passare del tempo si arrivava anche a 6/7) e poi una serie di fallimenti in catena.
Dopo una minuziosa Log Analysis, il problema è stato individuato proprio nell'operazione di MSCK REPAIR TABLE: le cancellazioni progressive effettuate durante i blocchi Delete, eliminavano i file, ma lasciavano tracce di metadati quando le cartelle che li ospitavano divenivano vuote. Lo scorrere del tempo, quindi, rendeva la MSCK via via sempre più onerosa e addirittura dannosa. Con il dilatarsi dei tempi infatti poteva accadere che due MSCK andassero in conflitto, causando *Dead Lock* e blocco del cluster.

Per aggirare il problema è stata studiata una seconda implementazione, in cui lo scheletro del main è rimasto di fatto invariato, mentre è stato modificato il codice della lettura da Blob Storage. In particolare, è stata introdotta la libreria *BlockBlobService*, che permette di accedere direttamente

al Blob senza passare dal file system di Databricks. Il listing dei file tramite essa è risultato subito molto efficiente, così come la cancellazione dei file. Utilizzando le funzioni preposte, infatti, si è risolto il problema dei metadati relativi alle cartelle vuote, che con la nuova implementazione vengono cancellate automaticamente.

La soluzione è riportata in 5.4.

```
from datetime import *
from pyspark.sql.functions import *
from pyspark.sql.types import *
from pyspark.sql import Row
from azure.storage.blob import BlockBlobService

def read_BS(start_date, end_date):
        #Fase di aggancio al servizio
        blob_service = BlockBlobService(account_name='[...]', account_key='
            [...]')
        bloblistingresult = blob_service.list_blobs(container_name='[...]',
            prefix='path_base')

        #Inserimento in lista di tutti i file a partire dal path_base
        LSName = [i.name for i in bloblistingresult]

        if LSName:
                df = spark.createDataFrame(LSName, StringType())

                #Ricavo la data, immettendola in una nuova colonna
                df = df.withColumn("data", gdata(col("value")))

                #Filtro sull'intervallo
                df = df.filter(df["data"] >= lit(end_date)).filter(df["data"]
                    < lit(start_date))
```

```
        #Ricreo il path completo per ogni file
                df = df.withColumn("fullPath", gPath(col("value"))).drop("
                    value").drop("data")

        #Trasformo la colonna in una lista
    ListaFile = df.select("fullPath").rdd \
        .flatMap(lambda x: x).collect()

    #Leggo puntualmente i file avro interessati, partendo dalla lista
    if ListaFile:
        dataframe = spark.read.format("com.databricks.spark.avro").load(
            ListaFile)
```

Listing 5.4: Pseudocodice della Read Blob Storage - V2

L'idea è quella di recuperare la lista di tutti i file e riversarla su una colonna di un Dataframe Spark. Tramite *udf* custom, ricavare una colonna che corrisponde al timestamp a partire dal path dei singoli record per applicare infine un filtro.
Alla fine dell'elaborazione, il Dataframe conterrà solo i record appartenenti all'intervallo di tempo desiderato. Basterà dunque convertirlo in una lista, tramite l'operazione Spark di *collect()* e passare tale lista al metodo *read()*.

Anche questa seconda implementazione è stata sottoposta a test scrupolosi, che stavolta ne hanno evidenziato la robustezza.
Le performance del processo sono risultate nettamente migliori della versione precedente: mediamente l'intera esecuzione, se escludiamo l'attesa programmata di 5 minuti, ha una durata complessiva di 1/2 minuti, che stavolta però si mantiene costante col passare del tempo.
Il fattore che tuttavia discrimina la bontà della soluzione è l'assenza di stalli all'interno del cluster: aggirando il problema della MSCK REPAIR Table, il pericolo di dead lock è stato scongiurato in maniera definitiva.
Una volta che i dati atterrano sul Data Lake, sotto forma di CSV, vengono riprocessati e trasfor-

mati in base alla tipologia di dato sotto analisi. In particolare, su Databricks vengono eseguite le seguenti operazioni:

- Parsing dei payload, sulla base dei formati descritti nella documentazione ufficiale.
- Scrittura sul datalake, in formato CSV, sotto un opportuno path.
- Scrittura su una o più tabelle SQL, a seconda della tipologia di dato.

Infine, per mettere gli utenti nella condizione di dover interagire con i dati ad un livello più alto possibile, sono state sviluppate sulla piattaforma Microsoft Power BI e pubblicate online su un workspace riservato, delle dashboard con grafici e tabelle, a cui è possibile applicare svariati filtraggi.

5.2.2 Applicazioni Real-Time

Insieme ai messaggi definiti nel paragrafo precedente, vi sono altre tipologie di dati che debbono essere gestite: si tratta di pacchetti di dimensione più grande pensati per essere inviati dalle box con alta frequenza e da gestire stavolta in real-time, che verranno utilizzati in futuro per effettuare computazioni, che permetteranno all'utente di visualizzare l'evoluzione in tempo reale, attraverso interfacce grafiche.
Proprio la natura dei messaggi non rendeva consigliabile l'uso dell'architettura basata su Databricks (Spark) descritta prima. Occorreva infatti un sistema che riuscisse a scalare prontamente, abile a gestire flussi più corposi di informazioni ed eventuali picchi di lavoro. Da qui la scelta di basare l'architettura dei nuovi applicativi su tecnologia Docker e *Kubernetes*, scrivendoli in C# su ambiente *.NET Core*.
Anche stavolta, verrà fatto un focus preliminare sulle tecnologie utilizzate per lo sviluppo.

5.2.2.1 Componenti utilizzati

Rispetto ai componenti visti prima, occorre aggiungere degli elementi fondamentali allo scenario: l'Azure Service Bus, Docker e la piattaforma Kubernetes.

Azure Service Bus: questo componente offre la possibilità di interazione fra applicazioni e servizi. L'Azure Service Bus permette infatti la gestione di messaggi tramite code semplici, o la gestione degli scambi tramite meccanismi di pubblicazione/sottoscrizione, o ancora, la connessione diretta fra applicativi.

Si tratta di un servizio multi-tenant, cioè condiviso fra più utenti, che hanno la possibilità di creare un namespace e definire i meccanismi di comunicazioni necessari al suo interno.

All'interno del tipo di applicazioni descritto in questo paragrafo, è stato sfruttato il meccanismo della coda semplice: un mittente invia un messaggio e un destinatario lo utilizza in un momento successivo.

Ogni messaggio è formato da due parti: delle proprietà, intese come coppie chiave-valore e un payload, che può essere binario, testuale o anche XML.

Docker: Il software Docker è una tecnologia che consente la creazione e l'utilizzo di container Linux.

Un container è un insieme di processi isolati dal resto del sistema, che eseguono un'immagine contenente i file necessari per tali processi. Tale immagine contiene quindi tutte le dipendenze dell'applicazione e ciò garantisce portabilità e facile controllo delle versioni.

A differenza di quello che accade con le macchine virtuali, come mostrato in figura 5.6, i container condividono il kernel della macchina host, isolano i processi applicativi dal resto del sistema e non hanno uno strato di virtualizzazione tramite hypervisor, che nelle macchine virtuali permette di eseguire più sistemi operativi contemporaneamente in un singolo sistema, ma rende l'utilizzo delle risorse complessivamente più intensivo.

Le caratteristiche peculiari dei container Docker sono:

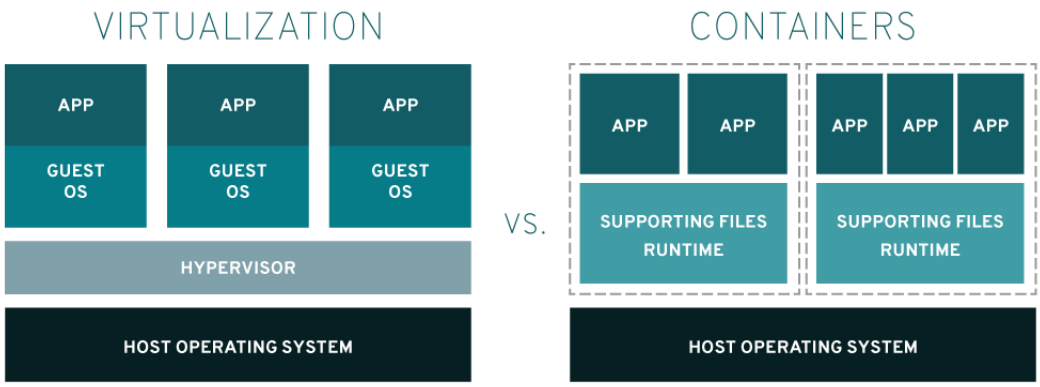

Figura 5.6: Virtualizzazione vs Containers

Modularità: Docker utilizza un approccio granulare, basato sui microservizi, che permette di estrarre i singoli componenti di un'applicazione, da aggiornare o riparare. Permette inoltre di condividere processi fra più applicazioni.

Strati e Versioning: i file immagine sono costituiti da più strati, che corrispondono alle modifiche apportate all'immagine stessa. Ogni modifica è condivisa tra le immagini e ciò porta benefici in termini di velocità, dimensione ed efficienza. Per ogni modifica, inoltre, un registro offre il controllo sulle immagini containerizzate. La stratificazione infine, rende possibili eventuali operazioni di rollback.

Deployment rapido: si tratta di una delle caratteristiche più importanti di Docker. I container, infatti, possono essere deployati in pochi secondi e non è necessario riavviare il sistema per aggiungerne o spostarne uno.

Kubernetes: è una piattaforma open source che consente di gestire cluster di host su cui sono eseguiti container Linux.

Alcune applicazioni possono trovarsi su più container, da distribuire su più host. Kubernetes ha proprio lo scopo di orchestrare e gestire la distribuzione dei container in maniera scalabile, per ottimizzare la gestione dei carichi di lavoro.
Sfruttando Kubernetes è possibile:

- Orchestrare container su host multipli.
- Ottimizzare l'utilizzo dell'hardware e gestire la scalabilità di risorse e applicazioni.
- Controllare e automatizzare i deployment, con la possibilità di customizzarne le modalità, e gli aggiornamenti delle applicazioni.
- Monitorare l'integrità delle applicazioni e gestire eventuali correzioni.
- Montare e aggiungere storage per applicazioni stateful.

Come la maggior parte delle piattaforme distribuite, un cluster Kubernetes è composto da almeno un master e più nodi di computazione. Lo schema architetturale è mostrato in figura 5.7. Analizziamo nel dettaglio i componenti e i concetti principali:

Master: è la macchina che controlla i nodi, costituisce il punto di origine di tutte le attività, schedula i deployment e in generale gestisce l'intero cluster.

Nodi: sono macchine, fisiche o virtuali, che eseguono i task richiesti e vengono controllate dal Master. Ogni nodo esegue un container runtime, oltre ad un agente che comunica col master. I nodi eseguono anche componenti aggiuntivi per il logging, il monitoraggio e l'individuazione di servizi aggiuntivi.

Pod: costituiscono gli oggetti base di Kubernetes che possono essere creati o gestiti e che vengono eseguiti dai nodi. Ogni pod rappresenta una singola istanza di un'applicazione o di un processo in esecuzione ed è composta da uno o più container.
Un pod, inteso quindi come gruppo di container, può essere avviato, arrestato e replicato dalla piattaforma. In fase di runtime, i pod replicati, al fine di garantire **affidabilità**, rimpiazzando prontamente container con eventuali guasti, **load-balancing**, permettendo la distribuzione del traffico su differenze istanze, e **scalabilità**, aggiungendo ulteriori risorse in caso di necessità.

Controller di Replica: ha il compito di controllare il numero di copie di un pod da eseguire sul cluster.

Servizio: è un'astrazione che definisce un set logico di Pod e una politica con cui accedervi. In altre parole, disaccoppia le definizioni dei task dai pod. Anche nel caso in cui i pod vengano riposizionati nel cluster, i proxy di servizio di Kubernetes reindirizzano le richieste al pod corretto in maniera automatica.

Kubelet: è il servizio eseguito dai nodi che legge i manifest dei container e garantisce il corretto avvio ed esecuzione dei conteiner definiti.

kubectl: è lo strumento di configurazione da riga di comando di Kubernetes.

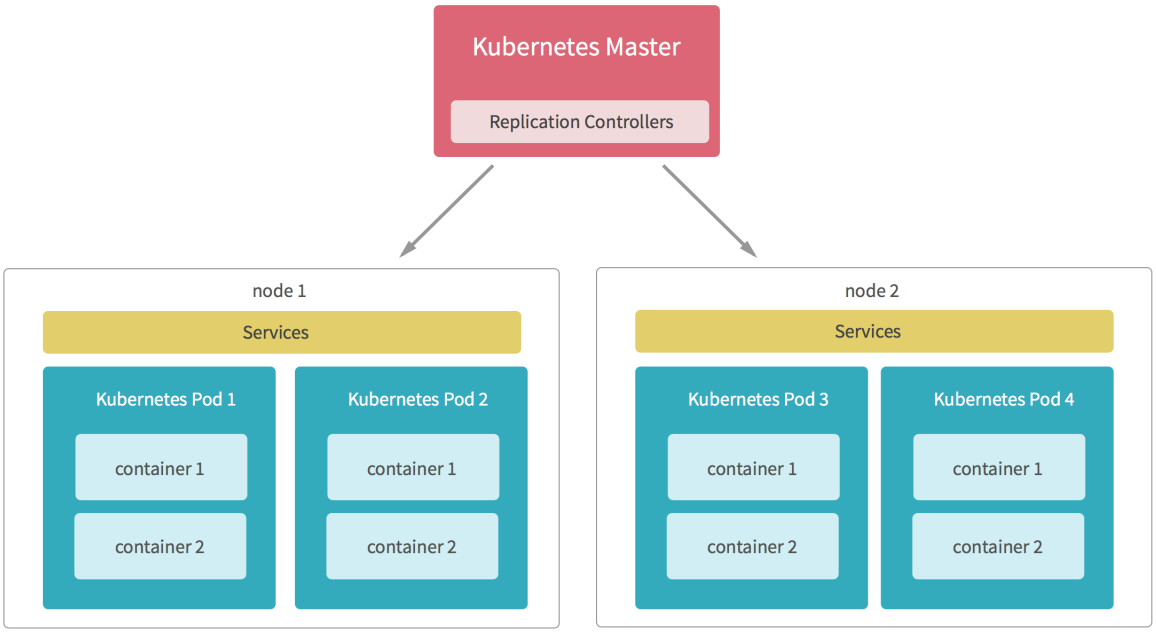

Figura 5.7: Architettura Kubernetes

Per quanto concerne l'interazione Docker-Kubernetes, quando viene assegnato un pod ad un nodo, il kubelet su quel nodo chiede a Docker di lanciare i container necessari. Le informazioni riguardanti lo stato dei container vengono aggregate nel nodo master dal kubelet. Docker dunque invia i container sul nodo e si occupa di arrestarli e avviarli. É Kubernetes però, a chiedere a docker di eseguire queste operazioni in maniera automatizzata.

5.2.2.2 Architettura Logica

In figura 5.8 è rappresentata l'architettura logica della soluzione: a monte dell'applicazione sviluppata, un altro servizio il cui deploy è stato eseguito ancora su Kubernetes, ha il compito di ricevere dei file e di memorizzarli all'interno di un container presente in un Blob Storage.
Contestualmente al trasferimento del file, manda un messaggio su una coda di un Service Bus. Esso è composto ancora una volta da un Header e un Payload. L'intestazione è composta da:

EnqueuedTimeUTC: anche stavolta è il Timestamp di invio del messaggio.

Action: si tratta di un campo che stabilisce se il messaggio è stato mandato in modo automatico, periodicamente, o su richiesta del driver.

Il payload è un file JSON da deserializzare. Fra i suoi campi occorre menzionare l'URL che indica dove recuperare il file, alcuni flag che suggeriscono il comportamento all'applicazione, un ID del veicolo e la tipologia di messaggio.

L'applicazione sviluppata per questo use-case, dunque è perennemente in ascolto sulla coda del service bus. Ogni volta che riceve un messaggio, lo deserializza e fra i suoi campi controlla in particolare un flag dell'header, che, come nel caso del flusso near real-time precedentemente descritto, indica se si debba interrogare o meno il Cosmos DB prima di proseguire, e un URL relativo al Blob Storage, che indica la posizione del file da spostare sul Data Lake.
Come prima, infatti, qualora le informazioni ricavate dal database Cosmos indichino un risultato negativo, occorre scartare il messaggio. In caso contrario, il processo termina scrivendo il file sul Data Lake, rimettendosi in attesa di ulteriori arrivi.

La figura 5.9, che riproduce la macchina a stati, riassume gli step fondamentali dell'applicazione.

5.2.2.3 Implementazione

La soluzione è composta dalla classe *Program.cs*, il punto di partenza, da un file di configurazione che contiene le credenziali di accesso ai vari componenti, e da più classi che implementano i

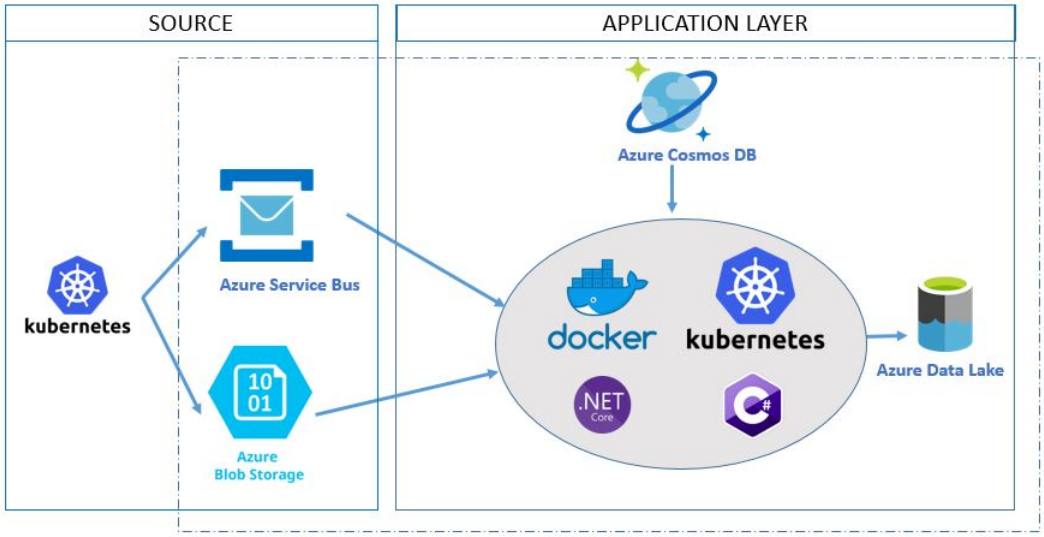

Figura 5.8: Flussi Realtime: Architettura Logica

servizi e che sfruttano un insieme di librerie presenti nel progetto complessivo sviluppato da altri colleghi di Reply.

Le classi principali sono:

Program.cs: è l'entry point del programma, contiene il recupero delle informazioni relative alle credenziali di accesso ai singoli componenti interessati, l'istanziazione dei servizi corrispondenti e delle classi *Processor.cs* e *DeadLetterQueueProcessor* che dipendono da essi.

DeadLetterQueueProcessor.cs: è la classe che prende in consegna eventuali messaggi corrotti o che sollevino eccezioni nel codice.
In questa fase la classe è stata creata ed istanziata, ma si limita a buttar via i messaggi non conformi. In futuro, tuttavia, se dovesse essere stabilita una politica di gestione diversa, andrebbe implementata in questa classe.

Processor.cs: è la classe che prende in gestione i messaggi che arrivano sulla coda del Service Bus. Nel suo costruttore vengono inseriti i servizi che saranno usati nei vari step.

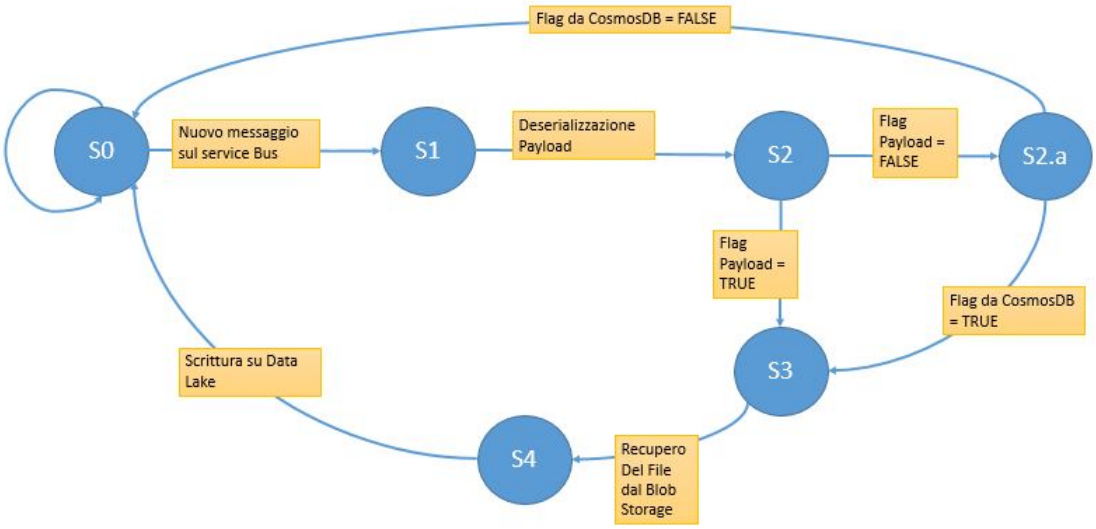

Figura 5.9: Applicazione RealTime: Macchina a stati

L'esecuzione del suo metodo principale crea e lancia consecutivamente i metodi delle classi *FirstStep.cs* e *SecondStep.cs*.

FirstStep.cs: Questo primo step ha il compito di analizzare il messaggio. Il primo passo è quello di deserializzarne il payload, andando ad analizzare i vari campi. Fra questi, vi è il flag che indica se sia necessario o meno eseguire un'interrogazione supplementare sul Cosmos DB: in caso negativo si procede, altrimenti è necessario valutare la risposta della query per capire se il messaggio vada scartato o meno.
In seguito il processo si occupa di recuperare il file indicato in uno dei campi del payload dalla locazione del Blob Storage corrispondente.

SecondStep.cs: questa classe si occupa di eseguire lo step finale. In particolare carica il file recuperato nel passaggio precedente su una cartella del Data Lake in base alla tipologia del messaggio e all'istante in cui è stato ricevuto sulla coda del Service Bus.

L'applicazione, a partire dalla classe principale *Program.cs*, sfrutta il pattern di programmazione conosciuto come *Dependency Injection*, che costituisce una tecnica per ottenere l'inversione di controllo (IoC) fra classi e relative dipendenze.

Una dipendenza è un qualsiasi oggetto richiesto da un altro oggetto e a livello di codice, specie in progetti di grandi dimensioni, realizzarle tramite l'operatore *new()* può causare problemi. Vediamolo in un esempio:

```
public class A
{
    public A()
    {
    }
    public void Write(string message)
    {
        Console.WriteLine(message);
    }
}

public class B
{
        A _dipendenza = new A();
        public void InviaMessaggio(){
        _dipendenza.Write("Inviato Messaggio");
    }
}
```

Listing 5.5: Esempio di dipendenza

La classe B crea e ha una dipendenza dall'istanza A. Questo può causare problemi:

- Per sostituire la classe A con un'implementazione diversa, la classe B andrebbe modificata.
- Se a sua volta la classe A avesse altre dipendenze, queste andrebbero configurate dalla classe. In caso di progetti di grandi dimensioni, la configurazione risulterebbe sparsa e difficile da tenere sotto controllo.
- É complicato eseguire unit test di questa implementazione.

Il pattern Dependency Injection risolve i problemi con il seguente approccio:

- Uso di un'interfaccia per astrarre l'implementazione dalle dipendenze.
- La registrazione della dipendenza nel contenitore di servizi *IServiceProvider* fornito dalla piattaforma.
- Inserimento del servizio nel costruttore della classe in cui viene usato. Sarà il framework a creare l'istanza della dipendenza e ad eliminarla quando non più necessaria.

Nel riquadro 5.6 viene mostrato un esempio di come sia stato sfruttato il pattern all'interno del progetto.

```
class Program {
        private static ServiceProvider RegisterServices(IConfigurationBuilder
            cb){
        IConfiguration Configuration = cb.Build();

        var serviceCollection = new ServiceCollection();

        serviceCollection.AddOptions();

        serviceCollection.AddSingleton(Configuration);

        var localConfig = new ServiceConfiguration(Configuration);

        serviceCollection.AddSingleton<IServiceConfiguration>(localConfig);

        serviceCollection.Configure<DataLakeConfig>("datalake", Configuration.
            GetSection("datalake"));

        serviceCollection.AddSingleton<IDataLakeService, DataLakeService>();

                return serviceCollection.BuildServiceProvider();
    }
```

```
}
```

Listing 5.6: Pattern Dependency Injection

L'esempio mostra come sia stata creata l'istanza ServiceCollection e, dopo aver ricavato dal file json di configurazione la sezione relativa al Data Lake, come sia stato agganciato il servizio corrispondente.

Anche questa soluzione, così come le altre, è stata prima valutata in un ambiente di test, per assicurarsi che nessun file venisse perso o mandato erroneamente in Dead-Letter e in generale che i file venissero memorizzati correttamente sul Data Lake, e poi è stato effettuata la conteinerizzazione Docker e il deploy su Kubernetes in ambiente di certificazione.

Capitolo 6

CONCLUSIONE

Il lavoro presentato in questa Tesi ha mostrato una delle possibili soluzioni al problema della migrazione fra gli ambienti On-Premise e Cloud e lo sviluppo di ulteriori componenti per la gestione di nuovi flussi. Il processo di progettazione e sviluppo dell'architettura ci ha dato modo di capire come e quando usare e, in generale, di esplorare nuove tecnologie.

Una menzione speciale in questo senso va a Databricks, probabilmente il tool più sorprendente fra quelli usati. I suoi punto di forza si sono rivelati essere la semplicità con cui configurare i cluster, la versatilità delle applicazioni implementabili tramite esso e la facilità con cui è possibile interagire con gli altri componenti del mondo Azure, in particolare con quelli di archiviazione. Abbiamo visto come siano stati gestiti i vecchi file, passando dalla vecchia infrastruttura, e insieme i nuovi flussi.

Ecco i principali vantaggi portati dalla nuova infrastruttura:

- Non è più necessario sovradimensionare le macchine per essere sicuri di essere in grado di gestire i carichi di lavoro più ingenti. Sfruttando la dockerizzazione, l'autoscaling di Kubernetes, abbiamo la possibilità di scalare potenzialmente in maniera infinita, dimensionando man mano l'infrastruttura in base al carico del momento.

- Per l'ingestion non si usa più un unico entry point centralizzato, ed è quindi possibile gestire i flussi di dati in maniera divisa, con la possibilità di introdurne dei nuovi senza impatti sugli

altri.

- Vi è stato un abbattimento dei costi complessivi, grazie alla possibilità di sfruttare il concetto Cloud del pay-per-use, cioè pagare soltanto le risorse che vengono effettivamente sfruttate.

6.1 Sviluppi Futuri

Tutte le soluzioni implementate lasciano spazi a scenari futuri di diversa natura. Come anticipato nel capitolo 4, nei prossimi mesi verrà messo a disposizione il servizio Express Route di Azure, che porterà alla dismissione della soluzione tampone della macchina-ponte.

Per quanto riguarda la vecchia architettura, questa rimarrà ancora in piedi. Si prevede infatti che col passare del tempo le vecchie box telematiche verranno usate sempre meno, e di conseguenza anche i vecchi formati di dati spariranno gradualmente in modo naturale.

In quanto al lato relativo ai nuovi flussi, tutte le applicazioni implementate fanno sì che i dati atterrino sulla stessa istanza di Data Lake, sotto percorsi differenziati in base al tipo di informazione che si sta trattando. Grazie all'alta capacità di interazione fra i servizi Azure, sarà possibile, utilizzando come sorgente proprio il Data Lake, effettuare nuove analisi, trasformazioni e riscritture dei dati, in modo da renderli conformi alle esigente dell'utente. Altresì, essendo le applicazioni strutturate in maniera modulare e flessibile, sarà possibile inserire al loro interno nuove funzionalità senza intaccare le sezioni di codice preesistenti. Un'altra funzionalità che verrà implementata nel prossimo futuro nell'ambito dei flussi real-time, sarà il parsing dei dati, in maniera del tutto speculare a quanto avviene per gli applicativi batch.

Questo tipo di processi fanno tutti parte di una prima macro-fase di ingestion e archiviazione dei dati. Oltre ad essa, l'obiettivo futuro sarà quello di riuscire a creare soluzioni veramente IoT, che permettano comunicazioni bilaterali fra i dispositivi ed i sistemi, in modo da essere in grado di fornire all'utente una risposta pronta ad eventi di vario tipo. In questo senso, due processi che verranno implementati sono:

IoT Predictive Maintenance: il nome, manutenzione predittiva, indica già lo scopo dei processi di questo tipo. Essi infatti si basano su algoritmi di apprendimento automatico con capacità predittive, al fine di prevedere futuri fallimenti delle macchine. Infatti, grazie ai dispositivi IoT, è possibile monitorare in tempo reale le apparecchiature, inviando continuamente informazioni all'algoritmo di machine learning, che, grazie all'apprendimento sui dati di training, sarà in grado di rilevare e segnalare prontamente anomalie.

IoT Geofencing: si tratta di una tecnologia emergente basata su GPS, che nel caso del nostro cliente permetterà di tener traccia delle posizioni dei veicoli, di ottimizzare percorsi in tempo reale, o l'efficienza nell'uso del carburante, prendendo decisioni basate sui dati inviati dai dispositivi IoT.

6.2 Apprendimento personale

Da un punto di vista personale, non posso che essere grata per l'opportunità di crescita avuta: in pochi mesi ho potuto apprendere e soprattutto mettere in pratica linguaggi di programmazione che non conoscevo, come Python, di approfondire la conoscenza di Spark e dell'ambiente .NET, imparando nuovi pattern di programmazione.
Ho inoltre potuto esplorare il mondo Hadoop, accessibile grazie alla distribuzione Cloudera, e l'insieme vastissimo di servizi e strumenti che compongono il Cloud di Azure.
Oltre all'acquisizione delle pure conoscenze tecniche, ho potuto inoltre affinare le capacità di team-working, oltre ad avere un primo approccio con le interazioni col cliente, aspetto fondamentale in questo settore.

Listings

Elenco delle figure

Elenco delle tabelle

Bibliografia

[1] John Gantz e David Reinsel. *THE DIGITAL UNIVERSE IN 2020: Big Data, Bigger Digital Shadows, and Biggest Growth in the Far East*. 2012.

[2] Kim Weins. *Cloud Computing Trends: 2017 State of the Cloud Survey*. URL: https://www.rightscale.com/blog/cloud-industry-insights/cloud-computing-trends-2017-state-cloud-survey.

[3] Apache Software Foundation. *Apache Hadoop*. 2018. URL: http://hadoop.apache.org.

[4] Apache Software Foundation. *Apache Hadoop powered by*. 2018. URL: https://wiki.apache.org/hadoop/PoweredBy.

[5] Apache Software Foundation. *Apache license-2.0*. 2004. URL: http://www.apache.org/licenses/LICENSE-2.0.

[6] Sanjay Ghemawat, Howard Gobioff e Shun-Tak Leung. *The Google File System*. 2003.

[7] Tom White. *Hadoop: The Definitive Guide, 4th Edition*. O'Reilly Media, 2004.

[8] Jeffrey Dean e Sanjay Ghemawat. *MapReduce: Simplified Data Processing on Large Clusters*. 2004.

[9] Apache Software Foundation. *ApacheKafka*. 2018. URL: https://kafka.apache.org/intro.html.

[10] Apache Software Foundation. *Apache Spark*. 2018. URL: https://spark.apache.org/.

[11] Microsoft Azure. *Obiettivi di scalabilità e prestazioni per Archiviazione di Azure*. URL: https://docs.microsoft.com/it-it/azure/storage/common/storage-scalability-targets#.

RINGRAZIAMENTI

Questa Tesi rappresenta l'anello di congiunzione fra il lungo, faticoso, ma gratificante percorso di studi, e l'inizio di una nuova strada. Prima di intraprenderla, è giusto fermarsi a ricordare chi ha fatto parte di questo cammino.

Prima di tutto vorrei dire grazie a Marco, che mi ha dato l'opportunità di entrare nel bellissimo gruppo di Data Reply. Grazie a tutti i ragazzi e le ragazze che ne fanno parte e che da marzo mi hanno accolta fra di loro, sempre disponibili a darmi una mano in caso di necessità.

Una menzione particolare va al mio Relatore, il Professor Paolo Garza, che mi ha fatta appassionare al mondo dei Big Data e poi, con gentilezza e professionalità, mi ha fornito gli strumenti necessari per terminare questo lavoro.

Ringrazio la mia famiglia, dislocata a varie latitudini, che non mi ha fatto mai mancare il suo sostegno. Mamma, papà, spero che leggendo questa tesi, possiate finalmente realizzare che il mio lavoro non consiste nell'aggiustare computer sui camion (‼!).

Grazie a Cecilia, presenza fondamentale di questi anni, e a Giulia, con cui ho condiviso il periodo toscano dal vivo, e quello torinese a distanza.

Grazie a tutti,
Mariagrazia