Technological Infrastructures

Part I - Prof Ciavotta

1 Componenti di NIST

NIST sviluppa standard di riferimento per il pubblico. Software Architecture è un organizzazione globale di sistemi software, e consiste in:

- Divisione della componenti software in sottosistemi;
- Definizione delle politiche con cui questi sistemi interagiscono;
- Definizione delle interfacce tra le varie componenti.

Un'architettura di riferimento è essenzialmente un template (scatola vuota con elementi prefissati), fornisce solo il vocabolario usato comunemente per discutere le implementazioni di un dato software.
Un'architettura di riferimento per il software non è altro che architettura software dove le strutture e i vari elementi e relazioni sono forniti dai template.
L'architettura di riferimento, fornita da NIST, per i Big Data:

- Fornisce un linguaggio comune per i stakeholders;
- Incoraggia aderenza ai standard comuni;
- Permette di implementare le architetture con una certa consistenza;
- Illustra e migliora la comprensione delle componenti, processi e sistemi di Big Data;

1.1 I 5 ruoli principali per Big Data

L'architettura concettuale dei Big Data è un architettura a croce con due assi: Information value (IV) e Information Technology (IT) (Fig 1).
I 5 ruoli principali sui 2 assi dei Big Data sono:

1.1.1 System Orchestrator

Il system orchestrator coinvolge spesso anche Information Value chain, poiché si preoccupa di implementare e monitorare i processi business a livelli enterprise e le varie politiche sui dati: Redere i dati accessibili per un tempo limitato oppure fornire i dati a velocità diversa(passando il dato in memoria al disco). Può assegnare/fornire componenti framework fisici o virtuale al sistema, questa assegnazione può essere molto spesso elastica ed indipendente. Può fornire supporto GUI e collegare i varie applicazioni a un livello alto, e attraverso il management fabbric monitorare i carichi e il sistema per garantire/specificare la qualità del servizio necessaria per i vari carichi. E' molto spesso centralizzato. Es. Ambari/Cloudera

1.1.2 Data Provider

Può essere sia un software (ad es. in una pipeline più grande) che una persona. Se è una persona metterà i suoi dati e userà gli strumenti di collection e curation/preparation per caricare

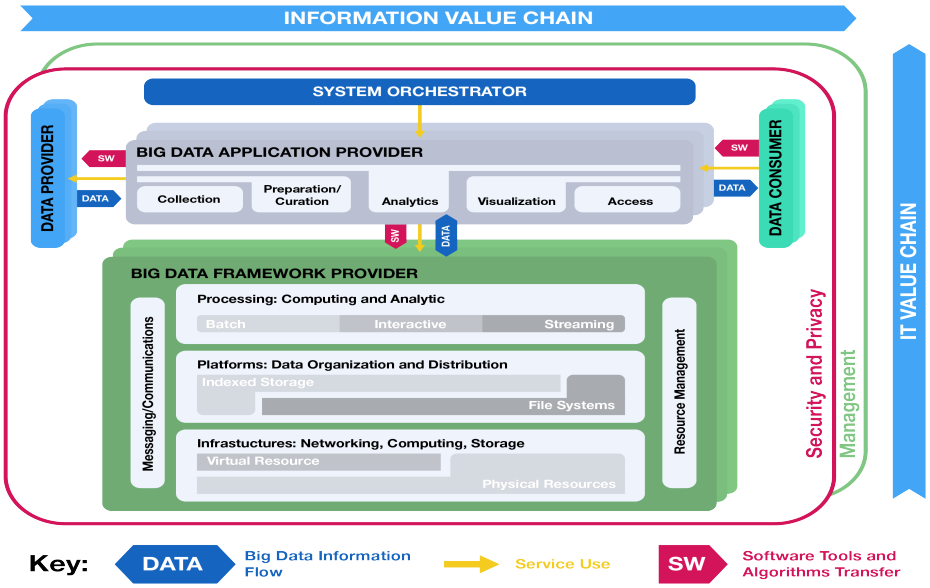

Figure 1: NBDRA Conceptual Model

i dati sul sistemi e migliorarne la qualità, Se un software metterà is suoi dati a disposizione attraverso delle interfacce come Apache Scoop. Il data provider può essere interno o esterno alla piattaforma, deve fornire i diritti di accesso ai dati, ed è obbligato a seguire le policy di privacy e security fabbric. I dati possono essere inseriti in pull o push. es. Flume per caricare i dati da MySQL a HDFS.

1.1.3 Data Consumer

Riceve i output dei sistemi BigData, può anche lui fare pull e push dei dati, può usare le informazioni per data reporting, retrival/search e visualization. Ci deve essere l'autenticazione ed autorizzazione da parte della privacy and security fabbric per la comunicazione tra l'architettura e il Data Consumer.

1.1.4 Big Data Application Provider

Corrisponde alle attività tipiche di un Data Scientist, che esistono anche nei sistema tradizionali ma hanno delle trasformazioni nella implementazioni con i Big Data. Queste attività sono:

- Collection: Si occupa di gestire l'interfaccia fornita dal Data Provider, salva/gestisce questi dati in una certa zona affinché questi non vengano persititi, inoltre implementa funzionalità di estrazione dati dal data provider;
- Preparation: Effettua Data Validation, rimozione outlier, standardizzazione, formattazione e arricchimento. Cerca di promuovere dati di alta qualità;
- Analytics: Estrazione conoscenza dai dati, sfruttando il software sottostante del Big Data Framework Provider;
- Visualization: Presentazione dati in

maniera visuale;
- Access: E' l'opposto della collection, si preoccupa di esporre i dati verso l'esterno.

1.1.5 Big Data Framework Provider

Fornisce le infrastrutture per supportare i Big Data Application Provider. Si occupa in particolare di:

- Processing dei dati - ha una dualità: da una parte è un framework che mette a disposizione delle interfacce per la programmazione per fare certe cose (es. MapReduce) e dall'altra parte definisce come l'implementazione viene effettuata. I framework possono variare tra un proccessamento batch e in streaming.
- Platforms per l'organizzazione e immagazzinamento dei dati - può contenere i metadati insieme alle descrizione semantiche dei dati. Può sia essere relazionale distribuito che non relazionale.
- Infrastructures per l'esecuzione fisica del nostro software, è l'insieme delle risorse computazionali fisiche o virtuali sulle quali il nostro sistema Big Data gira, può essere costituito da server di grandi o piccole dimensione. Queste componenti forniscono:
  - Networking - Possono essere definiti attraverso software e possono essere reti fisiche che può essere a sua volta partizionato in reti virtuali. Possono essere reti puramente virtualizzate cioè tutto quanto (firewall, router, load balancing) è realizzato in maniera virtuale (es. VM dentro la nostra macchine);
  - Computing - Hardware, Software, OS, memoria per il computing;
  - Storage - dischi per storare (in locale), RAID, in rete ecc.;
  - e altri servizi come il Raffrendamento, l'apparto elettrico e la sicurezza

  Può essere deployato su ambienti fisici che virtualizzati (nativi, hostati o contenerizzati).

Inoltre ha 2 ruoli diffusi nelle 3 componenti sopraindicate:

- Comunicazione e messaggistica tra le componenti;
- Gestione delle risorse per l'integrazione delle componenti.

1.2 I 2 ruoli diffusi per Big Data

Questi 2 ruoli prendono il nome di Fabric, il termine Fabric(tessuto) viene usato perché questi due ruoli sono cross-cutting, cioè sono presenti un pò ovunque nell'architettura.

1.2.1 Management fabbric

Le due attività principali associate sono:

- Gestione del sistema per provvedere le risorse, gestione dei software e pacchetti e infine la gestione delle configurazioni e performance delle varie pipeline;
- Ciclo di vita dei Big Data, BDLM (Big Data Life Cycle Management), contiene l'enforcing delle Policy (es. encoding/decoding), gestione dei meta data (data governance), accessibilità dei dati, data recovery e la loro preservazione.

1.2.2 Security and Privacy Fabric

Si occupa delle tre caratteristiche tipiche delle security:

- Autenticazione: Indica tutte le attività che validano l'utente;
- Autorizzazione: Una volta autenticato l'utente verifica i suoi permessi, es. alcuni

dati potrebbero non essere accessibili a certi utenti;
- Auditing: riguarda la registrazione degli eventi che accadono nel sistema, può far partire un allarme in caso di evento anomalo o a posteriori analizzare la sequenza di eventi (con i file log).

2 Virtualizzazione

Per i computer sono stati definiti con le 5 componenti classiche:

1. Input Devices: Tastiera ecc.
2. Output Devices: Display ecc.
3. Storage Devices: Volatile(RAM), Permanente(HD, SSD)
4. Processore:
    - Datapath
    - Control
5. Network

La virtualizzazione permette l'esecuzione di più sistemi operativi simultaneamente sulla macchina in maniera totalmente isolata. Può essere visto come una emulazione di un software o hardware su cui altri software possono eseguirsi, questo ambiente emulato è detto virtual machine. Il concetto di VM è stato sviluppato negli anni 60 da IBM sui mainframes. Viene abbandonato con la nascita di PC moderni e ripreso con la crescita recente di cloud. La virtual machine viene ottenuto attraverso un Virtual Machine Monitor (VMM) detto anche ipervisore.
L'**Hypervisor**(Ipervisore) è un software che giace sotto gli OS virtualizzati per offrire le funzionalità di condivisione delle risorse disponibili in modo tale che il programma o OS in esecuzione veda queste risorse come se fosse a lui dedicate. Le risorse sono CPU, memoria, storage e la rete.

Vi sono diversi benefici della virtualizzazione:

- Visione unificata delle risorse es. vedo tanti dischi come un unico disco;
- Consolidazione delle risorse virtualizzate, in modo da avere utilizzo ottimale delle risorse;
- Facilità nell'implementazione della ridondanza per copiare gli ambienti virtualizzati;
- Facilità la migrazioni di sistema su un altro, inoltre se non cambio ipervisore la macchina(virtuale) funzionerà identicamente a prima;
- Gestione centralizzata del hardware e software.

Altri benefici/proprietà della virtualizzazione sono:
Workload Isolation: attraverso la virtualizzazione è possibile isolare completamente i programmi, che ha miglioramenti anche nella sicurezza, inoltre aumenta affidabilità poiché il fallimento di un programma non comporta fallimento dei programmi poiché sono isolati, inoltre si risolvono anche i problemi riguardanti i conflitti di librerie in questo modo. Infine si ottiene un controllo sulle performance poiché l'esecuzione di una VM non affligge il performance dell'altra;
Workload Migration: ciò aiuta in:

- Mantenimento di Hardware;
- Load Balancing;
- Fault Tolerance;
- Disaster Recovery.

Poiché possiamo spostare tutto l'ambiente virtualizzato su una nuova macchina in maniera abbastanza trasparente, per fare ciò la macchina dovrebbe essere sospesa, totalmente serializzata per essere inviata nella rete, migrata su una nuova macchina e fatta ripartire immediatamente o solo salvata senza esecuzione.
Consolidazione: Sfruttando il Workload Migration è possibile consolidare macchine separate

su un unica piattaforma riducendo i costi (usata molto spesso nei Datacenter in orari non di picchi).

I diversi tipi di HyperVisor sono (Fig 2):

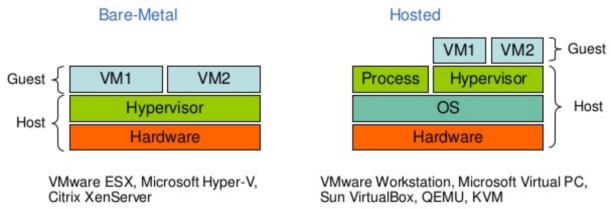

Figure 2: I tipi di Hypervisor

- **Hosted**: in questo caso l'ipervisore è un processo che gira al di sopra del sistema operativo e permette l'esecuzione di più macchine Guest. In questo caso quindi bisogna installare prima un OS su cui verrà installata VMM (ipervisore) e a questo punto l'host potrà eseguire le applicazioni all'interno della sua finestra. Il vantaggio qua è la facilità di installazione e configurazione, inoltre la HostOS e GuestOS rimangono non modificati e non dipendono dal particolare hardware, ma gli svantaggi sono la degradazione delle performance e la mancanza di supporto real time OS poiché vi sono varie entità/software in mezzo;
- **Bare-Metal**: in questo caso l'ipervisore funziona direttamente al di sopra del Hardware. In questo caso l'ipervisore è un OS molto leggero e comunica direttamente con hardware al posto di dipendere su un altro OS. I vantaggi sono miglioramento in I/O e supporto real time, mentre gli svantaggi sono la difficoltà di installazione e configurazione e la dipendenza dal tipo di hardware specifico.

Ci sono principalmente 2 tecniche di virtualizzazione: **Software Virtualization** (di cui abbiamo parlato fino ad adesso) e **Hardware Assisted Virtualization**.

Nella **Virtualizzazione Totale** VMM si preoccupa di emulare in maniera completa tutto l'hardware, quindi avremo un processore, memoria, disco e network virtuale. In questo caso OS ospite non è consapevole dell'esistenza dell'ambiente virtuale e ogni macchina è del tutto indipendente. A livello CPU, avviene la traduzione binaria: Questo avviene in diversi anelli di sicurezza, in particolare gli anelli sono 4, dove l'anello 0 è quello più privilegiato e permette l'esecuzione del codice direttamente sul hardware e qua viene eseguito il kernel dell'OS. Le applicazione dell'utente vengono eseguite sull'ultimo anello (anello 3).

L'ipervisore gira sull'anello 0 mentre le GuestOS gira sull'anello 1, quindi hanno più permessi delle normali applicazioni. VMM ha accesso sul anello 0 per avere accesso diretto sulla CPU piuttosto che virtualizzarla.

La **Para-Virtualizzazione** ha un approccio diverso rispetto alla virtualizzazione totale (è una via di mezzo tra total virtualization e bare-metal), in questo caso il Guest sono consapevoli della VMM e usa chiamate speciali in alcuni casi per essere eseguite direttamente sul hardware e ciò comporta un miglioramento nella performance ma ciò lo rende meno flessibile.

La **OS-level Virtualization (Containerization)** non usa la VMM, la virtualizzazione è fornita direttamente dal HostOS che esegue tutte le funzioni di un ipervisore totalmente virtualizzato, quindi ha una partizione puramente virtuale delle risorse, e ciò comporta un'assegnazione flessibile delle risorse alle varie applicazioni. Es. Docker.

Ci sono 3 modelli di servizio:

1. **Server Virtualization**: supponiamo di

avere diversi server su macchine diverse, in caso di un problema (crash di un nodo) o vi è un bisogno di upgrade, in caso di macchine fisiche dovrò prendere lo stesso modello o lo stesso vendore, la soluzione è quella di sfruttare una medesima macchina con un virtualizzatore e mettere i diversi server insieme. In questo modo ho una consolidazione, risorse condivise, una gestione centralizzata, facilità di migrazione, maggiore ROI e meno spazio occupato. La Disaster Ricovery e scalabilità viene facilitata, diversi modelli (hardware) a scelta (basta che sia uguale l'ipervisore) e ho maggiore disponibilità.

2. **Desktop Virtualization**: è la tecnologia che separa l'ambiente desktop e le applicazione software dal cliente fisico che lo usa. Il nostro desktop diventa un thin client che usa PC privo della memoria RAM e potenza calcolo necessaria per far funzionare una macchina reale, la macchina gira su un pool di VM su un server su un data center.
   I benefici sono molteplici: Upgrade di software e OS facilitato, alta disponibilità, fault tolerance, accessibile da LAN, WAN, Internet, inoltre vi è la possibilità di far eseguire una piccola parte della computazione dal dispositivo locale.
3. **Application Virtualization**: Molto simile alla Desktop Virtualization, solo che al posto di tutto il desktop, sono le applicazione ad essere in virtualizzate, quindi un applicazione che gira sul desktop, fa la computazione sul cloud, una piccola parte della computazione può essere effettuata anche in locale. I vantaggi sono molto simili a quelli del Desktop Virtualization, un vantaggio aggiuntivo può essere che pago solo quello che uso, riducendo i costi delle licenze.

3 Cloud

Definizione "vera" del cloud da ricordare: è un tipo di servizio distribuito di sistemi interconnessi e computer virtualizzati dinamicamente provvisionati e presentati come un unica risorsa computazionale basato su service-level agreement.
Sostanzialmente il cloud è la possibilità di avere accesso alle risorse computazionale attraverso la rete on-demand ed è composto da 5 punti chiave, 3 modelli di servizio e 4 modelli di deployement.

3.1 The 5 Properties of cloud

Le 5 proprietà aggirano attorno ad una idea centrale del cloud: Utility Computing, SOA (Service Oriented Architecture) + SLA(Service Level Agrement).
Utility Computing nel senso che il service provider fornisce le risorse computazionali e le infrastrutture che servono al cliente e li fa pagare in base all'utilizzo piuttosto che con un costo fisso, come i servizi on-demand, per massimizzare l'uso efficiente, minimizzando i costi nella maniera meno trasparente possibile(senza mostrare al cliente tutto ciò che si fa).
SOA (Service Oriented Architecture): è un insieme di servizio che comunicano tra di loro poiché non basta di solito un solo servizio per implementare un servizio web completo per utente.
SLA (Service Level Agreement): è un contratto tra il service provider e il cliente che specifica il livello di servizio che il service provider deve fornire in termini di QoS. Il **QoS**(Quality of Service) è un set di tecnologie per gestire il traffico della rete in modo da migliorare la user experience, ormai è associato a un significato più

generico riguardante le valutazioni di tecnologie non funzionali, cioè non mi interessa solo la funzionalità (il risultato della mia richiesta) ma anche la sua efficienza.
Le metriche più comuni delle SLA sono up-time e down-time, Response time. Se le metriche garantite non vengono rispettato vi sono delle penalità sui service provider.
La tecnologia grazie al quale cloud funziona sono:

- Hardware Virtualization;
- Computing distribuito e parallelo;
- Service-oriented Computing;
- Autonomic Computing (reazione a cambiamenti del sistema).

Le 5 **proprietà/caratteristiche** che identificano il cloud sono:

1. **Scalabilità, Elasticità**:
   - Scalabilità è una proprietà del sistema di crescere in maniera graduale col crescere di richieste, può essere orizzontale (Scale In & Scale Out) o verticale (Scale Up & Scale Down).
   - Elasticità è l'abilità di adattarsi automaticamente per inizializzare la scalabilità.

   Ciò si può ottenere attraverso provisioning dinamico, che fa riferimento è un ambiente complesso che permette la allocazione e deallocazione on-demand delle istanze/risorse attraverso applicazione o un console amministrativo. Di solito vengono messe delle regole per automatizzare il provisioning, ottenendo così un abbassamento dei costi e performance migliorata.
2. **Disponibilità, Affidabilità**:
   - Availibility è il rapporto tra il up-time e tempo totale di esecuzione;
   - Affidabilità è la capacità di funzionare in situazioni particolari (rottura di un nodo, attacco di hacker) per un certo tempo.

   Questi punti possono essere ottenuti attraverso sistemi di:
   - Fault Tolerance è la capacità del sistema di continuare ad funzionare anche dopo un fallimento di uno dei suoi componenti, spesso il servizio viene degradato proporzionalmente alla gravità del fallimento ma il servizio continua ad funzionare.
     Le caratteristiche principali sono che non ha un singolo punto di fallimento, la componente fallita viene individuata ed isolata per prevenire la propagazione del fallimento;
   - Resilience è l'abilità di offrire e mantenere un livello di servizio accettabile anche dopo un fallimento, essenzialmente ritornare allo stato di funzionamento dopo un caso di fallimento del sistema (ad es. viene a mancare l'elettricità). Quindi i sistemi devono avere le policy e procedure per il recupero, ad es. Backup (dei dati off-site oppure di tutto il sistema) oppure preparazione contro fault come un una corrente non-interrompibile (UPS) oppure sbalzi di corrente/tensione.
   - Sicurezza, che riguarda l'impiego di politiche e tecnologie e sistemi di controllo per protegge applicazioni e infrastrutture da accessi malevoli, quindi abbiamo il suo impiego in:
     - la protezione i dati, mantenendo l'accesso ai dati riservato solo in base ai privilegi;
     - Gestione dell'identità per garantire l'accesso alle risorse in base ai loro privilegi (protezione dei dati);
     - Sicurezza dell'applicazione riguarda la possibilità di blindare le nostre applicazioni per accessi malintenzionati;
     - Privacy per oscurare i dati in base alle leggi privacy e gestiti solo da utenti com-

petenti.

3. **Gestibilità, Interoperabilità**: La Gestbilità riguarda la gestione di questi sistemi attraverso un singolo punto di accesso mentre la interoperabilità è una proprietà di un prodotto o sistema per lavorare con altri prodotti/sistemi. Lo si ottiene attraverso **System control automation** e **System State Monitoring** che è un processo che monitora lo stati dei hardware, metriche dei performance, i log dei sistemi, il pattern dei accessi alla rete ecc. E' anche collegato al sistema di Billing, poiché gli utenti pagano ciò che usano e il cloud provider deve registrare le risorse/servizio usate da ciascun utente.

4. **Accessibilità, Portabilità**
5. **Ottimizzazione, Performance** Il **Load Balancing** è una tecnica per la distribuzione del workload su un sistema di più computer, CPU, HD ecc. per ottenere l'utilizzo ottimale, migliorando così: l'utilizzo del sistema, la performance e l'efficienza energetica riducendo overload.

   Il **Job Scheduler** è un applicazione che ha il compito di eseguire i processi in background (chiamati anche processi batch). Nel cloud i task intensivi computazionalmente o task che crescono dinamicamente vanno pianificati con Job Scheduler.

Da un punto di vista dell'utente, lui non vuole sapere come e cosa viene fatto né chi gestisce il servizio, ma vuole solo un servizio funzionale.

3.2 The 3 Service Model of cloud

Iniziamo con i tre modelli del servizio del cloud sono:

1. **IaaS (Infrastructure as a Service)**: es. le macchine virtuali, in questo caso il provider/vendor gestisce la virtualizzazione, il cliente ottiene le risorse virtuali, di solito da un catalogo che contiene le specifiche hardware (virtualizzato) pre-selezionate dal provider. Nella maggior parte dei casi gli OS sono anche prefissate per questioni di prestazioni/stabilità e compatibilità con hypervisor da loro usato. Gestito di solito da un system administrator.

2. **PaaS (Platform as a Service)**: ci viene fornito una scatoletta dove scrivere ed eseguire/testare le applicazioni, il provider sarà il garante del fatto che il sistema sia abbastanza responsive e abbia un runtime sufficiente. E' meno flessibile rispetto allo IaaS ma allo stesso momento lo sviluppatore perde meno tempo nella configurazione delle macchine. Gestito di solito da sviluppatori.

3. **SaaS (Software as a Service)**: es. GMail quindi utente non può fare niente tranne usare il software. Usato da utenti o business analyst.

In realtà il cloud vero è un po' più offuscato di quanto detto sopra, spesso non è facile collocare il modello di cloud di un'azienda esattamente in uno dei 3 modelli.

Un aspetto molto importante da considerare è quello economico, cioè grazie al modello pay-as-you-go cloud il costo capitale (CAPEX) si trasforma in costo operazionale (OPEX), inoltre il rischio di errore nella scegliere le macchine sparisce, se ho bisogno di più potenza richiedo una macchina più forte e se ho bisogno di meno potenza abbasso la potenza e risparmio. Allo stesso momento i cloud service provider hanno diversi benefici: profitto sfruttando l'economia di scala (comprare un servo costa X, ma comprare N server non costa N*X), possono capitalizzare sui loro investimenti (amazon che vende la potenza residua) o possono sfruttarlo per promuovere un loro prodotto (es. per far usare

.NET ai sviluppatori).

3.2.1 IaaS

IaaS fornisce automaticamente il processing, storage, network e altre risorse fondamentale per il computing, e l'utente può installare i software che sono per lui necessari.
Lo IaaS supporta queste 3 caratteristiche del cloud:

- Gestibilità e Interoperabilità: Attraverso un interfaccia può gestire tutte le VM che vuole, le può pagare come vuole;
- Disponibilità ed Affidabilità: gestita dal cloud provider attraverso le zone di disponibilità;
- Scalabilità ed elasticità: Proprietà del sistema ma deve essere implementato dall'utente.

L'architettura è composta dall'hardware seguita da un layer di virtualizzazione e l'utente ha accesso a due interfacce:

- Interfaccia per gestione delle risorse, queste risorse sono:
  - VM: creazione, cancellazione, gestione, sospensione delle VM
  - Virtual Storage: Allocare spazio, ridurre/aumentare del DB, scegliere servizi con velocità di lettura/scrittura diversi (per prezzo);
  - Virtual Network: gestione delle IP associate alle VM, registrazioni al dominio delle IP, scegliere la larghezza di banda.
- Interfaccia del il monitoring del sistema: viene messa a disposizione da VIM: L'orchestrazione delle macchine/risorse, in maniera rapida e dinamica, su un server viene gestita dalla Virtual Infrastructure Mangaser(VIM), diverse metriche vengono usate per il monitoraggio es:
  - VM: CPU usage, memory usage;
  - Virtual Storage: utilizzo del disco, livello di duplicazione dei dati e velocità di accesso al disco;
  - Virtual Network: L'uso della banda, lo stato di connettività e il bilanciamento sulla rete.

3.2.2 PaaS

PaaS mette a disposizione dell'utente i linguaggi di programmazione e i tools (come IDE) supportati dal provider, quindi il controllo sul deployement dell'applicazione è nelle mani del consumer ma le infrastrutture sottostanti sono gestite dal cloud provider.
Alla base vi è una archittettura come quella di IaaS, vengono aggiunti i Runtime Environment e sopra questi ci sono i Programming IDE e le API/tools supportati dall'ambiente, spesso hanno in comune le funzioni come: computation e lo storage.
Il Runtime Environment si riferisce a una collezione di software, implementati con una collezione di librerie, le proprietà fornite dal Runtime Environment sono:

- Gestibilità ed Interoperabilità
- Performance ed Ottimizzazione
- Disponibilità ed Affidabilità
- Scalabilità ed Elasticità

L'interfaccia messa a disposizione per il controllo del sistema mette a disposizione un set di azioni in base a:

- Policy based control: le azioni seguono delle regole per prendere delle decisioni quindi azioni seguite con if <> then <>
- Controllo del workflow: descrizione del flusso di installazione e configurazione delle

risorse oppure dei daemon.

3.2.3 SaaS

SaaS: vengono fornite al cliente applicazione del provider in cloud, l'utente non può gestire o sviluppare l'applicazione o gestire l'architetture nel cloud ma può interagire con esso attraverso interfacce ad es. portali/applicazioni web, che con introduzione del Web 2.0 ha potenziato l'iterabilità con l'applicazione in cloud, es: Facebook(Messenger), Office, Skype, DropBox, sistema CRM, sistema medico nazionale, sistema di trasporto pubblico.
Il punto fondamentale di questo tipo di cloud è l'Accessibilità e la portabilità.
Oss: La differenza tra il portale web e una pagina web è che il portale permette l'integrazione di diverse pagine attraverso dei login.

3.3 The 4 Cloud Deployement Models

I 4 modelli primari che differenziano un cloud, basati sulle possibilità di accesso, la dimensione e la proprietà dei servizi, sono:

- **Public Cloud** (o multi-tenant o external cloud): L'infrastruttura viene messa a disposizione del pubblico generale o almeno alle organizzazioni grandi attraverso pagamento per il suo uso, le caratteristiche sono:
  - Infrastruttura omogenea per garantire le medesime prestazioni a due utenti che pur trovandosi in posti diversi usano lo stesso servizio;
  - Policy comuni;
  - Risorse condivise;
  - Infrastrutture viene affittata;
  - Economia di Scala.
- **Private Cloud:** (o on-premise cloud o internal cloud) Viene operato da un'unica organizzazione e può essere gestita dalla stessa o un'altra organizzazione, cioè la categoria di utenti è limitata. A differenza del public cloud le caratteristiche sono:
  - Infrastruttura eterogenea per via del costo e dal fatto che una azienda privata non butta via un hardware solo per averli omogeneità;
  - Policy customizzate ad-hoc per gli utenti;
  - Risorse dedicate;
  - Infrastrutture gestita da house;
  - End-to-end control sui processi.
- **Community Cloud:** è una struttura gestita da diverse entità, in pratica più entità mettono insieme i loro cloud privati per generare un super cloud, ad esempio alcune università americane possono unire i loro cloud per qualche ricerca specifica ecc.
- **Hybrid Cloud:** è la comoposizioni di più tipi di cloud, ad esempio un azienda privata crea il suo cloud privato per la gestione di informazioni sensibili e usa il cloud pubblico per il resto, oppure lo usa per la scalabilità in caso di traffico aumentato.

4 Amazon ECOSYSTEM-IaaS

amazon web services sono creati per lavorare indipendentemente ma possono interagire tra di loro, condividendo tra di loro le stesse convenzioni di nomi e autenticazione e minimizzando le connessioni interne.
Amazon mette a disposizione il concetto di regione e di zona di disponibilità:

- Ciascuna **Availibility Zone** è un data center indipendente con la propria griglia di potenza e connessione della rete, le zone all'interno di una regione sono collegate tra di loro attraverso con-

nessione a latenza bassa. Il nome deriva dal fatto che sono zone ad alta disponibilità, quindi se una zona fallisce il suo effetto non viene notato dalle altre zone creando così una stabilità ed alta fault tolerance.

• **Regione** è un cluster di Availibility Zone localizzati in una area geografica. Quando si crea una istanza Amazon ci dà la possibilità di scegliere una availability zone (o di default lo si lascia far scegliere ad Amazon) e all'interno della stesso regione è possibile distribuire la propria istanza su più availability zone, quindi se una zona fallisce l'altra (molto probabilmente) continua a funzionare. Amazon non fa pagare i trasferimenti di dati all'interno della stessa regione, ma tra regioni le comunicazioni vengono fatte pagare.

Esiste AWS GovCloud che è una regione AWS isolata e permette di gestire dati estremamente sensibili e può essere acceduta solo da cittadini Americani verificate dal governo Americano e sul suolo Americano.

Uno dei servizi principali di AWS è **Simple Storage Service (S3)**, un storage object che è un sistema di archiviazione piatto cioè non è possibile creare cartelle al suo interno, i dati vengono salvati in formato binario e vengono distribuite automaticamente. L'accesso può essere effettuato da chiamate web (REST, Soap, BitTorrent) o query SQL e gli oggetti possono anche essere molto grandi (fino a 5TB ciascuno).

S3 sta alla base di tutta l'infrastruttura che sta alla base di Amazon e-commerce. Oltre a S3 Amazon fornisce anche un **EBS (Elastic Block Store)**, questo disco può essere formattato, montato e usato come hard disk locale, è un volume che persiste indipendentemente dal resto. Mentre le VM possono morire, i volumi rimangono sempre, e la garanzia di durabilità viene fornita da Amazon.

EBS è categorizzato in SSD e HDD, a seconda del tipo di macchine/dischi si paga in maniera diversa.

I **Security Group** definiscono l'insieme delle connessioni possibili per una certa istanza, e questi insiemi possono essere protocolli, porte, range delle IP.

4.1 EC2 - Elastic Cloud Computing

EC2(Elastic Cloud Computing) è uno dei servizi offerti per la creazione/gestione delle macchine virtuali on-demand. EC2 si basa sul concetto di data center programmabile, la possibilità di creare una propria infrastrutture (macchine in più availability zone o regioni) tramite linguaggi di programmazione. I concetti chiave che constituiscono EC2:

- **Amazon Machine Image** (AMI): è un file contenente una descrizione di una VM, cioè eventuali software e le configurazioni, possono essere pre-built, create, modificate e vendute. Ciascun AMI ha un ID unico;
- **Istanza**: rappresenta una copia di AMI in esecuzione, multiple copie di un unica AMI può essere messa in esecuzione;
- **Elastic IP address**: allocazione di un IP statico e collegarle alla propria istanza, ciascuna istanza può avere al più un IP statico.

I core messi a disposizione dal server (di cui non sappiamo praticamente nulla) vengono virtualizzati, dividendo prima in core poi dividendo ulteriormente il core-time. L'unità di misura è Elastic Compute units che corrisponde a Intel Xeon (o AMD Opteron) da 1.0-1.2 GHz del 2007.

Le risorse possono essere:

• Persistenti: anche in caso di fallimento del hardware Amazon ci garantisce la sua persistenza attraverso ridondanza, recupero automatico e failover automatizzato. Le componenti

persistenti sono:
  - Elastic IP Address
  - EBS
  - Elastic Load Balancer
  - Security Groups
  - AMI slavate in S3 o EBS

• Effimere: l'utente deve garantire la sua ridondanza e mantenimento attraverso atri servizi di EC2, in caso di fallimento i dati salvati vengono in generale persi. Le istanze sono effimere. I modelli del prezzo sono:

• **On demand**: Il pagamento avviene per le capacità (tempo) usate con nessun obbligo a lungo termine.

• **Reservati**: Si fa una sorta di abbonamento con un pagamento di una somma prima e poi le macchine si possono usare per la durata dell'abbonamento a un prezzo scontato, è disponibile in 3 tipi Light, Medium e Heavy e può essere per 1 o 3 anni e può far risparmiare fino al 71% rispetto a On-Demand. Attenzione non ci riserva delle risorse per la durata dell'abbonamento ma solo il prezzo.

• **Spot**: Si fa asta per la potenza computazionale non usata da Amazon EC2, sono quelle che costano di meno, perché non mi viene garantita la durata della macchina nel tempo, nel caso il prezzo offerto è maggiore del costo necessario per eseguire quella macchina Amazon darà la sua macchina, ma siccome il prezzo di elettricità è variabile e se dovesse superare il prezzo offerto Amazon può spegnere tale macchina senza avvisare.

Un altro indice di costo è **Data Transfer**, il trasferimento dei dati dentro o fuori dalla istanza EC2 vengono pagati (in base alla quantità) se non il trasferimento avviene al di fuori della regione. Il trasferimento dentro la regione è completamente gratuita.

4.2 AutoScaling

Uno servizio molto importanti di Amazon è **AutoScaling**(Fig 3). Per far funzionare AutoScaling è importante anche il **Elastic Load Balancer**, grazie al quale l'utente finale vedere solo un collegamento e Elastic Load Balancer pensa a distribuire le richieste alle nostre macchine. Le istanze EC2 possono essere categorizzate in auto scaling groups, di solito con un range tra un minimo e un massimo, se una nuova macchina entra in esecuzione oppure viene de-allocata il Load Balancer viene avvertito della modifica in modo che possa funzionare bene. Gli Auto Scaling group sono istanze EC2 che condividono caratteristiche simili e devono essere fatte per scalare, e aggiunge automaticamente un nuovo nodo in base a policy definite dall'utente, programmazione oppure health checks e workload e la nuova istanza può venire anche da availability zone diverse della stessa regione.

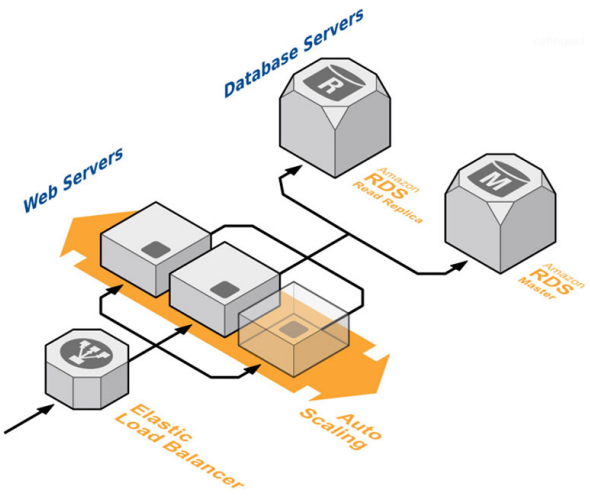

Figure 3: Esempio di Auto Scaling for AWS

Auto Scaling group contiene solo un launch configuration che descrive le istanze che devono far partire (contiene i parametri delle AMI), se si aggiorna il launch configuration questo affliggerà solo le nuove istanze, le vecchie istanze rimangono tali ma esse vengono terminate prima durante un scale in.
Attenzione una macchina può essere o spenta o terminata, nel caso viene terminata viene eliminata completamente e AutoScaling group termina una macchina.
L'ecosistema di Auto Scaling è composta da:

- Cloud Watch: Per monitorare le istanze EC2, le metriche più usate per il CloudWatch sono:
  - CPU usage
  - Latenza
  - Numero di Richieste
  - Il numero di Host(macchine) in buona salute
  - Il numero di macchine non in buona salute
- Elastic Load Balancer: Per distribuire il lavoro su un numero qualunque di istanze EC2 ed effettuare controlli periodici sulla salute del nodo (basandosi sul tempo di risposta) e in caso non siano in buona salute il load balancer smette di inviare a lui le richieste.
- Auto Scaling: Utilizza i dati collezionati dal CloudWatch per costruire sistemi che possono scalare (dentro o fuori) all'intero del range

Il **Trigger** è un meccanismo per l'attivazione di una policy, in questo caso di aumentare o diminuire il numero dei nodi. Il trigger può essere attivato da un allarme cloud watch (configurato per guardare una metrica di cloud watch) oppure un auto scaling policy che descrive cosa fare in caso di un allarme. Quando si attiva il trigger lancia un processo chiamato Scaling activity che esegue la auto scaling policy. Osservazione Auto Scaling supporta ma non necessita Elastic Load Balancer. In generale almeno due trigger sono necessari, uno per Scale up e uno per Scale down, per mantenere un equilibrio desiderato. Un allarme è una metrica e verifica se questa metrica supera un limite stabilito per un certo tempo.
Un'allarme è un'entità in grado di osservare con continuità una certa metrica e ci indica se tale metrica ha superato un certo valore prefissato per un certo tempo, per creare un allarme è necessario specificare:

- Metrica da osservare
- Il threshold della metrica
- Il numero di periodo di valutazione

Gli stati in cui l'allarme si può trovare sono:

- Ok, tutto bene! Metrica è sotto il threshold.
- Alarm: Metrica è sopra il threshold, è necessaria un azione.
- Dati Insufficiente, metrica non disponibile o non ci sono abbastanza dati.

Se l'allarme cambia lo stato e vi rimane per un determinato periodo di valutazione, un'azione viene invocata in base alla policy.
I scenari automatici dell'auto scaling sono:

- Fleet Management: assicura il performance ottimale della macchina, controllando la salute dell'istanza con Health Check: se un'istanza dovesse terminare questa viene individuata e si fa partire un'altra macchina.
- Scheduled scaling: Azioni vengono eseguite in maniera programmata, quindi è una sorta di evento temporale per eventi ricorsivi o scheduled.
- Scaling Dinamico: In base all'allarme su una metrica c'è una politica che risponde all'allarme. L'azione può dipendere step-

wise da gravità dell'allarme oppure facendo tracking di una certa metrica e aggiusto il sistema in modo da calmare l'allarme.

La terminazione della macchina può essere fatta anche per un ribilanciamento delle macchine nelle availability zone, la precedenza dello spegnimento viene data in modo da bilanciare tra le zone seguito dal fatto di voler preservare le macchine con il launch configuration più recente seguito dalla macchina prossima allo scatto dell'ora (per il prezzo che si paga ogni ora). Auto Scaling inizializza una nuova istanza prima di spegnerne una per non compromettere la disponibilità e le performance dell'applicazione. Dopo che è iniziata la fase di Auto Scaling esiste un periodo di cooldown, in questo periodo nessun altra attività di scaling può iniziare.

I canditati per autoscaling sono:

- Web Tier - sistemi che forniscono servizi web
- Application Tier
- Load Balancing Tier - sistemi che gestiscono il carico
- Stateless Tier - sistemi sono prive di stato es. funzioni pure, sono prive di memoria e quindi da un punto di visto funzionale non cambia se faccio autoscaling o no.

I candidati che non dovrebbero fare autoscaling:

- Database Relazionali
- Database non-relazionali
- Sistema di caching distribuito
- Elastic Search
- Sistemi con lo stato - es. Kafka, non avrebbe senso fare autoscaling automatico.

5 PaaS with Azure

Essenzialmente il PaaS mette a disposizione un'interfaccia di programmazione, un linguaggio di programmazione (runtime) e tutta una sorta di servizi necessari. Secondo l'approccio tradizionale, si sviluppa un'applicazione in locale e poi si deploya sul cloud, ma vi possono essere dei problemi come "it works on my machine": uno dovrebbe replicare esattamente le proprie condizioni di lavoro sul cloud. Mentre con il PaaS l'ambiente di sviluppo diventa l'ambiente di deployement e tale gestione dipende direttamente dal provider.

I 5 servizi Paas principali di Azure sono:

- Web App - Servizi PaaS per lo sviluppo di applicazioni Web;
- Mobile App - Servizi per lo sviluppo di applicazioni Mobile (gestione contenuti, utenti);
- Function App - Azioni microscopiche (piccole funzione) che possono essere deployate sul cloud;
- API App - Sviluppare API o usare API di terzi;
- Logic App - Serve per scrivere logica di integrazioni per applicazioni.

L'applicazione Web può avere 2 ruoli:

- Web Role: L'esecuzione di una web app, ha il compito della presentazione.
- Worker Role: Supporta l'esecuzione backend.

Osservazione: Un nuovo role non è una VM è simile ad una sandbox (l'applicazione vede se stessa come l'unica app in corso), quindi si possono avere più ruoli sulla stessa macchina, inoltre nel caso di una WebApp non sappiamo né la memoria né informazioni sul CPU, si pagano solo le richieste servite (o cicli del processore usati), inoltre viene garantito autoscaling e auto load balancing. La comunicazione tra Web Role e Worker Role avviene attraverso un sistema di code di messaggi, e un load balancer distribuisce

le richieste tra i vari Web Role.
Per la persistenza dei dati Azure mette a disposizione:

- **BLOB (Binary Large Obect) Storage**: è un object store molto simili al modello S3 di Amazon e può salvare qualunque tipo di file anche di grande dimensioni, è un filesystem distribuito ma la differenza rispetto all'S3 è che BLOB storage è a due strati, il primo livello prende il nome di Container e il secondo livello è BLOB. L'interfaccio con cui si interagisce con BLOBS è REST API, quindi PUT, GET, (UN)DELETE, COPY, SNAPSHOT, LEASE.
- **Azure Cloud SQL**: Server SQL senza overhead amministrativi, si può scegliere se si vuole questo DB su hardware condiviso o riservato, il modello che segue è Pay-as-you-grow. E' con alta disponibilità (99.99% mensili).
- **Tables**: Uno storage leggermente strutturato, è un database key-value quindi NoSQL, ciascuna entità può avere 255 proprietà ed essere grande al più 1 MB, come altri modello key-value abbiamo due chiavi: una chiave per partizione e una per la riga. La chiave identifica univocamente un'entità e usa Timestamp. Osservazione è fortemente **C**onsistente, ed è altamente **A**vailable, quindi secondo il teorema CAP non è particolarmente tollerante alle **P**artizioni.
- **Queues**: Un sistema di code di messaggi simile a Apache Kafka.
- **CosmoDB**: è un servizio equivalente a MongoDB offerto da Azure, è un poliglotta ed è un key-value, graph, column based e document type DB, supporta un indice automatico in base alle query molto richieste esattamente come Mongo. Inoltre garantisce una latenza molto bassa in tutto il mondo.

Azure offre un suo servizio di Auto Scaling, ma fra le varie possibilità offerte vi è la possibilità di creare auto scaling proprio basato sulle regole. Le regole possono essere:

- Constraint Rules: minimo e massimo numero di istanze in esecuzione per un certo role e lasciar decidere a Azure di decidere come e quando scalare;
- Reactive Rules: seguite in risposta a certi eventi, queste regole sono più complesse rispetto a quello che ci mette a disposizione Amazon, in particolare, possiamo aumentare il numero di Role in esecuzione ma Azure permette anche di cambiare il comportamento della nostra applicazione (Throttling) in maniera dinamica:
  - Rigetto delle richieste: Rigetta richieste di certi utenti in base a certe regole es. rigetta il 10% di richieste dalla Cina o da utenti che hanno già fatto n richieste in un determinato periodo per abbassare il carico;
  - Disabilitare o degradare certe funzionalità finché la situazione non si stabilizza;
  - Ritardare operazioni eseguite per applicazioni con bassa priorità

Come in Amazon si possono definire i Scaling Groups per raggruppare Role multipli e definire le Scaling Rules per tutti i Roles nel gruppo.

6 Containerization/Docker

è un progetto open source che ha l'obiettivo di automatizzare il deployement dell'applicazione all'interno di un software container, fornendo un livello di astrazione superiore a quello del sistema operativo. L'idea alla base è avere un sistema di packaging uniforme che possa essere eseguito ovunque (architetture x86(_64) e kernel linux). Il motivo

per cui è utile docker è che lavora sul kernel linux ed è così indipendente dalla particolare distro e posso installarci delle librerie che in generale sarebbero in contrasto con le librerie del sistema.
Docker aderisce alle policy delle OCI e non è l'unico sistema a farlo, tutti i sistema che vi si aderiscono in teoria dovrebbero permettere l'interoperabilità. Uno degli obiettivi principali è isolare le applicazioni, cioè voglio che non vedano i processi del sistema (per motivi di sicurezza). Inoltre sono lightweight poiché condividono il kernel di OS e non c'è uno strato di virtualizzazioni in mezzo.
L'idea è quella di eliminare hypervisor ed eliminare GuestOS, permettendo l'esecuzione delle applicazioni direttamente sul OS. La tecnologia container ha la dualità immagini - container, immagine è un file/template che contiene i meta data e configurazione e questo file viene usato per creare container, la differenza rispetto alle VM è che il container non è una copia dell'immagine (vedi AMI) bensì una sua esecuzione, un'altra differenza è il tempo di esecuzione rispetto alle VM che ci mettono tempi in ordini di minuti siccome docker è un processo parte in ordini di secondi.
Le immagini sono salvate di solito su Hub o registro locale. Il **filesystem** di docker è solo di lettura ed è a livelli (UnionFS), creare l'immagine a livello permette la massima riusabilità delle immagini, si può ripartire a creare un immagine a partire da una già esistente, ad esempio se ho un'immagine debian con emacs sopra posso partire da debian e mettere sopra nano e per fare ciò non devo riscaricare debian posso riusare già l'immagine di debian che possiedo. Riesce a creare questo modello a strati seguendo un approccio Copy-on-Write, cioè crea una copia del file modificato. Tutte le tecnologie tranne il filesystem a strati sono dovute usando il kernel linux, uno di queste tecnologie è il **namespaces**: permette l'isolamento dei processi, individuando le cose che può e non può vedere il processo. l'altra tecnologia principale che permette l'esecuzione dei docker è **cgroups**: che ha lo scopo di limitare e monitorare l'accesso alle risorse(CPU, Memoria, I/O, network).
Docker è un demone linux che mostra un'interfaccia REST che è utilizzabile tramite una CLI (command line interface) fornita da docker. Il ciclo di vita di un container è così

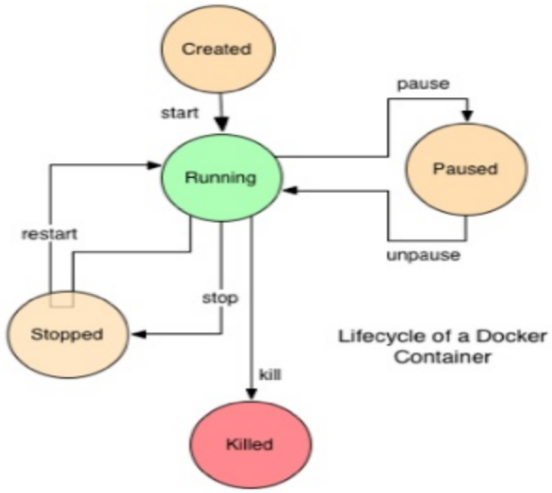

Figure 4: Ciclo di vita del Docker

come indicato in Figura 4 e i comandi principali sono:

- docker create: crea ma non esegue il docker
- docker run: crea ed esegue il docker
- docker stop
- docker start
- docker restart
- docker rm
- docker kill

Essendo un processo come gli altri docker di default può vedere tutta le risorse che il computer ha a disposizione, è possibile limitare con dei soft limits (limite inferiore delle risorse) e hard limits (limite superiore) l'accesso a queste risorse.
Quando si elimina l'immagine in esecuzione si perdono tutti i dati, quindi nasce il concetto di volume che sono cartelle condivise con il sistema (per non perdere i dati). I volumi sono di tre tipi:

- Bind Mount: è una cartella montata è in binding (collegata) a una cartella del sistema host;
- Volume Mount: sono anch'esse cartelle condivise ma gestiti dal docker, evitando problema di breech;
- tmpfs Mount: cartelle che non sono sul disco della macchina ma sulla memoria.

Tutto questo modello dei volumi quando ci troviamo nel cloud non funziona possono esservi diversi errori (fallisce una VM) e il volume è esplicitamente al Host, per rendere la persistenza del volume anche nel cloud docker è stato scritto in una maniera da poter usare dei plugin, il plugin per lavorare in locale (quello che abbiamo visto) si chiama local persist, esistono diversi plugin già creati dai vari provider cloud per funzionare sul loro sistema.
Il building process del docker è l'output di un dockerfile che contiene le istruzioni per far partire e il contesto che sono una serie di altre informazioni necessarie. Le istruzione sono sequenziali e vengono eseguite dal demone docker. Tra i comandi esistenti i più importanti sono:

- **ENTRYPOINT**: definisce un processo che deve essere eseguita al lancio del container, può avere due forme:

```
ENTRYPOINT[‘‘executable’’, ‘‘param1’’,\
          ‘‘param2’’]   (exec form)
```

```
ENTRYPOINT executable param1\
             param2      (shell form)
```

La differenza è che la shell form viene comandato dalla shell quindi ad es. si può terminare un processo con ctrl+c.

- **CMD** ha lo stesso significato della ENTRYPOINT, ma ha una differenza, se esiste un già ENTRYPOINT viene eseguito ENTRYPOINT e CMD gli passa i suoi parametri e se esistono molteplici CMD viene eseguito l'ultimo CMD. Es:

```
CMD[‘‘--port 27017’’]
ENTRYPOINT /usr/bin/mongod
# ENTRYPOINT riceve il parametro da CMD
```

La comunicazione tra i vari container si fa attraverso il docker networking, che è una tecnologia di virtualizzazione della rete, i docker connessi credono di avere a disposizione una propria connessione, questa connessione è un plugin del docker, questa connessione è inoltre distribuita e decentralizzata quindi ogni docker ha una parte d'informazione della rete. Questa idea viene formalizzata nel CNM (Container Network Model) che consiste di 4 elementi principali:

1. Sandbox: è una struttura che contiene le configurazioni della rete a livello di container, quindi gestisce le interfacce, la tabella di routing e una serie di informazioni del DNS. Una sandbox può avere più di un endpoint e collegata a più reti ma ogni container può essere collegato solo ad una Sandbox;
2. Endpoint: è l'implementazione dell'interfaccia di rete (es. veth - Virtual Ethernet) Un endpoint può essere associato ad una sola rete e una sola sandbox;
3. Network: è l'insieme di tanti endpoint che possono comunicare tra di loro;

4. Cluster: l'insieme di docker collegati ad una rete.

Quando creiamo una rete questa può essere di tipo:

- NULL: Il container non ha accesso alla rete;
- Host: Il container e l'host hanno la stessa interfaccia di rete;
- Bridge: E' un servizio a cui vari endpoint sono collegati;
- Overlay: Una rete che viene creata tra più host docker;
- Remote: E' un'interfaccia definita in modo tale da essere implementa da terze parti.

6.1 Docker-Compose

E' un linguaggio dichiarativo esprimibile attraverso file yaml, che descrive com'è costruita un'applicazione multicontainer come la rete e volume. In pratica usa i comandi docker dietro e semplifica la vita allo sviluppatore. Il file docker-compose.yml è composta da tre sezioni: Services, Networks e Volumes.

Un **Docker Swarm** è un cluster di macchine che usano docker e si uniscono logicamente in un sistema, ai fini di deployement è come se diventassero un'unica macchina. Il cluster viene gestito non più dalle singole istanze di docker sulle macchine ma da una istanza sola che prende il nome di swarm manager, le altre macchine vengono chiamate worker (o nodes). Il manager può usare diverse strategie per eseguire i container nei vari workers, un esempio è emptiest node dove i container vengono fatti partire nel nodo può vuoto. Solo il manager può decidere di eseguire i comandi o autorizzare l'aggiunga nello swarm di altri worker, i worker quindi mettono a disposizione solo le risorse computazionali e ricevono ordini dal manager.

Le funzionalità di docker swarm:

- Gestione del Cluster integrata con docker-machine, non necessita di software aggiuntivi esterni;
- Modello di servizio dichiarativo;
- Gestione della scalabilità automatica, creando anche repliche di container e costantemente continua a controllare lo stato di vita dei workers;
- Riconciliazione dello stato attuale con lo stato dichiarato dall'utente;
- Gestione della multi host networking con reti overlay;
- Implementazione del servizio discovery per l'individuazione dei nodi in maniera unica;
- Load Balancing, in caso di esistenza di varie istanze dello stesso servizio, queste riceveranno il carico distribuito;
- Sicuro di default poiché i vari nodi comunicano attraverso protocolli http criptati utilizzando tecnologie TLS.

Si possono avere più di un manager ma solo uno è leader gli altri sono detti Reachable e sono una sorta di leader di backup nel caso il leader fallisca ottenendo alta disponibilità, possono usare dei protocolli di consenso per prendere decisioni insieme.

7 Serverless

Esistono altri 2 modelli per il cloud oltre a IaaS, Paas e SaaS: **CaaS** (Container as a Service) e **FaaS** (Function as a Service).

Serverless può essere inteso come BaaS (Backend as a Service) oppure FaaS. Il BaaS è l'incorporazione di servizi autonomi in modalità PaaS di un'applicazione ad esempio il servizio di autenticazione. Nel FaaS lo sviluppatore non deve sviluppare applicazioni intere, ma solo scrivere delle funzioni che siano più semplici possi-

bili, queste funzioni devono avere un ruolo atomico e granulare. Sono dei container computazionali, che contengono una funzione effimera, che viene eseguita in caso di un evento esterno e queste funzioni sono gestite totalmente dal cloud provider.
In questo caso non vi è un gestione centralizzata, infatti preferisce la coreografia (coordinazione) delle funzione rispetto all'orchestrazione (non c'è un decisore centrale che richiama le funzioni), ottenendo così:

- Flessibilità e Estendibilità: si possono aggiungere altre funzioni senza alterare quelle pre-esistenti;
- Divisione delle preoccupazione;
- Costo ridotto: Non si paga se la funzione non è in esecuzione, quindi se una funzione è raramente usata non pago un server costantemente accesso.

Inoltre non sono più vincolato solo dai linguaggi offerti dal cloud provider, qualunque linguaggio che compila su un processo UNIX (quindi nel docker) va bene. Vi sono dei contraints, ad esempio in AWS una funzione può al massimo durare 5 minuti, se una funzione richiede più di 5 minuti forse conviene prendere un IaaS o PaaS, oppure si possono splittare le funzioni in più funzioni. Inoltre non vi è la garanzia del fatto che gli stati persistano su invocazioni multiple successive poiché le variabili sono salvati su memoria o un disco locale.
Le funzioni in FaaS sono tipicamente triggerate da eventi definiti dal cloud provider ad es. AWS include aggiornamenti sul S3, task schedulati, ricezioni di messaggi in un servizio di coda (come Kafka) oppure per le rispondere alle richieste HTTP che usano API Gateway.
Un altro svantaggio è il così detto cold start - crea una nuova istanza, inizia le funzioni host ecc. e il periodo di accensione dipende dal linguaggio di programmazione, le librerie da caricare e la configurazione della funzione e la quantità di codice, pur essendo in ordini di secondi è molto lento rispetto a warm start, che riusa una istanza di Lambda function esistente da una chiamata precedente.
I benefici sono molteplici:

- Costi Operazionali, di Sviluppo e Esecuzione Ridotti;
- Computazione più **Green** ed efficiente.

Vi sono anche dei svantaggi:

- Il controllo del Vendor: I downtime, i limiti, cambiamenti di costo, aggiornamenti di API;
- Problemi di mantenimento;
- La durata di esecuzione;
- Latenza di inizializzazione;
- Testing, Debugging e monitoraggio difficile e limitato.

Quindi il serverless è non ottimale per le funzioni che hanno bisogno di stati e sono più lunghe in generale ad es. Deep Learning Training, Streaming Pesante, Analitiche Hadoop, Gestione DB, streaming video ecc. Sono molto utili per eventi veloci, stateless e event-driven come microservizi, IoT, Inferenze ML, streaming leggero ecc.

8 Apache Spark

L'idea di fondo non è quella di spostare i dati nelle varie macchine ma i codici (data locality), quindi una volta ottenuto il parallelismo e data locality e in più implementiamo un sistema che sia in grado di gestire i fallimenti a vari livelli allora otteniamo il framework chiamato Hadoop. Spark piano piano sta sostituendo una parte dell'ecosistema di Hadoop.
Per introdurre il concetto degli operatori serve

capire cos'è il linguaggio funzionale: è un paradigma di programmazione, che vede la computazione come processo di valutazioni di funzioni matematiche, queste funzioni devono essere immutabili e prive di stato. Le proprietà della programmazione funzionale:

- Composizione di funzioni per ottenere funzione di ordine superiore
- Le funzioni sono senza stato (sono pure), prendono stato grazie ad un accumulatore che è una variabile passata in ingresso a una funzione
- Non possono modificare lo stato di altri funzioni (no side-effect)
- Possibilità di Ricorsione
- Lazy-Evaluation, esecuzione solo quando viene veramente richiesta
- Le funzioni non modificano le strutture dati, ma ne creano altre durante il processo

Grazie a queste proprietà si ottiene la parallelizzazione automatica, performance migliorata, nessun Bug e gestione della memoria.
I due operatori più interessanti per Spark sono:

- **Map**: un operatore di ordine superiore, prende in ingresso una funzione e una input list e applica questa funzione in ingresso a tutti gli elementi della input list restituendo un'altra lista;
- **Reduce**: un operatore di ordine superiore, prende in ingresso una funzione e una input list e applica questa funzione ricorsivamente a tutti gli elementi restituendo una combinazione della lista.

8.1 MapReduce

Una loro applicazione combinata è MapReduce, che serve a processare una grande quantità di dati con algoritmi parallelizzati e distribuiti su un cluster. Il sistema shuffla i dati nelle fasi in mezzo a Map e Reduce. I nodi del sistema condividono un filesystem distribuito e vengono sfruttati nelle due fasi:

- Fase **Map**: Il master node partiziona i files in M splits attraverso una chiave, e assegna ai slaves i maps da eseguire.
  Questi slaves dopo aver eseguito il task di Map, scrivono il loro output sul disco partizionato in R regioni attraverso una funzione hash;
- Ad altri slaves viene assegnato il task **Reduce**, ciascuna slave legge i dati dal disco del mapper corrispondente e li raggruppano in base alla chiave ed eseguono la funzione di Reduce.

Questo è l'architettura usata in Hadoop 1.x, uno dei suoi svantaggi sono il blocco che si crea ai vari slaves, poiché quando lavorano i Mapper i Reducer sono fermi e quando lavorano i Reducer i Mapper sono quelli fermi, quindi non si sfrutta la massima capacità del cluster. In Hadoop 2.x e Spark si è cercato di colmare questa problematica. Un altro svantaggio è che ad ogni iterazione di Map e Reduce il file viene scritto sul disco che rallenta tutto il processo. Inoltre non tutti lavori possono essere scritti in alberi di operazioni Map e Reduce, ad es. i cicli sui dati stessi.

8.2 Spark Ecosystem

Qua entra in gioco un'applicazione open source Spark: esso effettua i calcoli in memoria senza riscriverli sul disco se non necessario e permette di generalizzare i calcoli usando grafi aciclici ottenendo molta più flessibilità. Allarga il set delle operazioni da solo Map e Reduce in **trasformazioni** e **azioni**, dove trasformazioni e azioni sono un set di operazioni diverse che

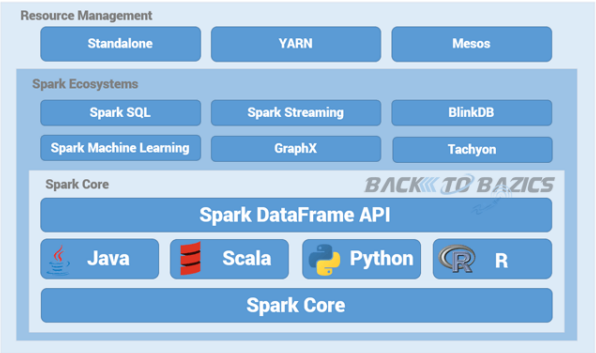

Figure 5: Spark Ecosystem

possono essere combinate in maniera arbitraria e Spark 2 ha implementato anche un ottimizzatore di query (Catalyst).
Apache Spark supporta:

- Manipolare i dati attraverso SQL, vedendo i file come delle tabelle SQL;
- Esiste una libreria MLlib per Data Science e Machine Learnig distribuito;
- Processamento di grafi di dimensioni enormi;
- Processamento di dati in tempo reale;
- File system Distribuito in-memory(cache);
- Query approssimate sui dati;
- Eredita da Hadoop il supporto dei file Avro, Parquet e le sorgenti di questi dati possono essere HDFS, Cassandra, Hive, Mongo ecc.;
- Sviluppo in linguaggi come Python, R, Scala, Java, .Net;

Al centro di tutto l'ecosistema c'è il Spark Core che si occupa di:

- Gestione del calcolo distribuito, quindi si occupa della distribuzione, coordinamento e scheduling dei task computazionali e inoltre si occupa di gestire i fallimenti e riduce al minimo la data shuffling tra i vari nodi;
- Fornire le API per l'astrazione delle **RDD**, che è un collezione di oggetti, immutabili e fault tollerant partizionati su un cluster, che possono essere manipolati in parallelo.

Spark può essere deployato in 3 modalità diverse:

1. Come una libreria che viene eseguita in un programma, fornendo le API delle RDD, ma si possono usare solo le risorse della macchina su cui gira il programma;
2. Può essere eseguito in maniera distribuita (su cluster), che può essere standalone (che non significa solo una macchina ma semplicemente il fatto che non ha bisogno di un scheduler esterno) oppure usando scheduler esterni tra cui YARN, Mesos e K8;
3. In modalità MR, in questo caso l'accesso ai sistemi di storage attraverso le API di Hadoop per usare HBase, S3, Cassandra ecc.

L'applicazioni Spark ha due elementi fondamentali più un terzo facoltativo:

1. **Spark Driver** che è la parte del codice che viene eseguita non in parallelo, questo driver crea uno SparkContext per gestire i jobs, lo SparkContext deve capire quali pezzi del nostro codice sono parallelizzabile e deve inviarli al cluster;
2. Nei vari cluster ci sono degli agenti **Executor** che eseguono gli ordini dal SparkContext e riescono a gestire una serie di task in parallelo, una volta concluso il task il risultato viene ri-inviato al driver;
3. Nel mezzo può esserci un **Cluster Manager** per permettere la comunicazione tra il driver e i vari nodi e di allocare le risorse.

Il context era un oggetto che creava delle connessioni ad es. per connettere SQL si creava SQLContext, per connettere Hive si creata HiveContext, ciò creava confusione e rendeva difficile una loro collaborazione. Questi diversi contesti, in Spark 2, vengono unificati in un unico oggetto

chiamato SparkSession, ottenendo un entrypoint unificato per gestire i dati in spark.

8.3 RDD

RDD (Resilient Distributed Dataset), essenzialmente una lista di oggetti distribuita e resiliente, distribuita nel senso che sono partizionati (in base a una chiave) in memoria di vari nodi ma viene vista come un'unica lista grazie alla chiave di partizionamento ciascuna partizione contiene un unico record che può essere operato indipendentemente (simile all'approccio Shared Nothing), ciò permettere la loro gestione in maniera parallela con le trasformazioni (Map, Filter) e in caso di fallimento vengono automaticamente ricostruiti.
Vengono di solito immagazzinati in memoria ma vi è la possibilità di scriverli sul disco, sono tipizzati (Integer, double, Objects), sono immutabili (quindi applicare una trasformazione significa creare un altro RDD) e sono lazy evaluated. Alcune possibili trasformazioni sono: map, filter, union, intersection, distinct e join (solo per RDD di tipo key-value).
Ciascun RDD tiene traccia delle trasformazioni usate per costruirlo, questa traccia è detta lineage, che può essere usato per ricostruire i dati persi. Le Action possono prendere un RDD e produrre un altro oggetto non RDD. Le trasformazioni sono lazy e non vengono eseguite finché non c'è un azione su RDD che li convoca.
Il Driver pianifica l'ordine di esecuzione convertendo le trasformazioni e azioni in un grafo aciclico diretto (DAG), queste DAG tracciano le dipendenze (Lineage) e i nodi del grafo sono RDD mentre gli archi rappresentano le trasformazioni. Le azioni sono il pezzo finale del grafo e triggerano l'esecuzione della DAG, un'azione per esempio può essere reduce, count, collect, save.
Attenzione se io devo effettuare due azioni sullo stesso DAG, le trasformazioni del DAG vengono eseguite 2 volte, per evitare tale problema si può inserire il comando cache prima dell'esecuzione delle azioni. Alcune trasformazioni, come sortByKey, join, groupByKey ecc., possono avere uno shuffle che può rallentare il sistema, le trasformazioni possono essere:

- Narrow dependency: Ciascuna RDD viene consumato da al più una partizione figlio ma un figlio può avere molteplici genitori, quindi non cambia il numero di partizioni tra una trasformazione e l'altra, e non avviene lo shuffle;
- Wide dependency: Ciascun RDD può essere usato da molteplici partizioni figli, quindi cambia il numero di partizioni

Spark fa questa distinzione perché eseguirà le operazione Narrow tutte insieme in parallelo, spark divide in lavoro in stage, in base alla presenza di operazioni shuffle, all'interno di ciascun stage le operazioni possono essere eseguite in parallelo, inoltre le trasformazioni nei vari stage vengono composte per velocizzare il tutto.
I vantaggi di RDD sono il controllo a basso livello, grazie alle API, su cosa fa spark e sono inoltre typesafe: spark ci permette di fare operazioni in base al tipo di RDD e infine ci insegnano in profondità come funzione spark.
I suoi svantaggi sono il fatto che sono difficile da gestire (per un data scientist che deve usare i DataFrames), inoltre cambia la velocità di esecuzione anche in base al linguaggio usato (usando Java/Scala si ottiene un boost di performance anche 2x rispetto a python) e infine l'ottimizzazione del codice non avviene e quindi programmi sono meno efficienti.

8.4 Spark SQL

E' una libreria che funziona su Spark Core e fornisce come astrazione del RDD il dataframe, con tutte le proprietà dei dataframe, sono delle tabelle con righe e colonne con i header e indici ecc. A questo punto si può usare la tecnologia molto sviluppata dei database SQL, Spark SQL ha implementato anche un ottimizzatore che viene usato da tanti motori SQL. Questo aumenta anche il supporto alle varie tipologie di file e sorgenti con cui spark può comunicare.
I Dataframe essendo un'astrazione delle RDD hanno le sue proprietà: sono immutabili, distribuiti, rappresentano una collezione, sono lazyily evaluated e le azioni sono eagerly evaluated.
Il risultato di usare i dataframe è non solo la stessa velocità indipendentemente dal linguaggio scelto ma addirittura un miglioramento rispetto a lavorare direttamente su RDD, questo anche perché i DataFrame sono compressi quindi usano meno memoria, inoltre viene tenuto in conto anche l'inefficienza dei programmatori mediamente parlando.

9 Stream Processing

La differenza tra il batch processing e stream processing è:

Batch Processing lavora su batch di dati limitati ottenuti da un data store e il risultato è un altro batch di dati. Batch ha accesso a tutti i dati, si può lavorare a qualcosa di grande e complesso, la latenza è misurata in minuti, di solito uno è interessato all'output (qualità) di dati che alla latenza.

Stream Processing lavora su un flusso continuo (infinito) di dati, il risultato è un nuovo stream di dati e per ciò è dovuto al fatto che il processo stream non finisce. La funzione in questo caso è applicata ad un datapoint o una finestra piccola di dati recenti, inoltre la computazione deve essere leggera poiché lo stream deve continuare ad andare e la latenza deve essere near real time o al più secondi.
Gli elementi principali di un'applicazione stream sono:

1. Sorgente: che produce i dati in input;
2. Trasformazioni: prende l'input del dataflow e producono alcuni output;
3. Sink: riceve/consuma i dati dell'output delle trasformazioni;
4. Data Pipeline: è una sequenza di trasformazioni, rappresentate di solito da un grafo aciclico diretto;
5. Event/Tuple/message: E' l'elemento atomico del data flow (come una riga di un dataframe).

Generalmente i sistemi di stream di dati sono generalmente distribuiti, per gestire più dati in parallelo e quindi con maggiore velocità, ma avvolte vi sono delle operazioni sequenziali e si fa fatica a renderli distribuiti. I dati in ingresso possono essere partizionati e ciascuna operatore può eseguirsi in una maniera concorrente. Ottiene le proprietà dei sistemi distribuiti tra quali: Fault-tollerance, Load Balancing.
Lo stato dello streaming può essere molto utile, ad esempio per riconoscere qualche pattern negli eventi o per aggregazioni di dati, e lo stato di fault tolerance viene gestito in utilizzando i due meccanismi di **Checkpoint and Replay**: salvataggio periodico dello stato dell'operatore su un storage sicuro (es. HDFS), in caso di fallimento tutti i dati dopo l'ultimo checkpoint dovranno essere rianalizzati (Replay).
Nel caso di dati "illimitati" in streaming di solito si preferisce usare il windowing per tagliare i dati

dando a loro il senso temporale. Queste finestre possono essere:

- Fixed (Tumbling) window es. calcolo di qualcosa ogni tot unità temporale
- Sliding window es. calcola di qualcosa in un unità temporale ogni tot unità temporale (possibilmente diversa da quella del calcolo)
- Sessione window, la sessione è definita come il periodo di attività terminato da un periodo di inattività più grande di una certa soglia, quindi sono finestre dinamiche e sono totalmente data driven, anzi sono molto compatibili con un sistema distribuito e ciascuna partizione avrà la sua finestra.

Esistono anche altri metodi di approssimazione, oltre a windowing, per gestire dati illimitati ad es. sampling con streaming K-means ecc., essendo delle approssimazioni vi sono degli errori. In generale sono algoritmi molto complessi poiché devono anche garantire un limitate superiore degli errori.

SQL on Streams consiste nel vedere uno stream di dati come una tabella relazionale di dimensione infinita, a questo punto si può usare SQL per generare un'analisi o trasformazione, il sistema ci mostra una tabella (dinamica) ma sotto ha un sistema di streaming distribuito che genera a partire da uno streaming in entrata uno streaming in uscita.

9.1 Apache Storm

STORM è disegnato per supportare lo stream di applicazioni e supporta fino ad 1 milione di messaggi al secondo, può scalare a miglia di nodi per cluster ed è fault tollerant e usando strumenti di terze parti, in particolare Trident è in grado di applicare anche la semantica temporale "exactly once".

Il modello concettuale è un modello molto semplice ed include:

- Tupla: è l'unità core (il messaggio) e può essere vista come una coppia chiave valore;
- Stream: è la sequenza infinità delle tuple;
- Spout: è la sorgente dello stream e il suo compito è quelli di emettere delle tuple;
- Bolt: riceve le tuple, fa una certa computazione (può anche leggere e scrivere dati da uno store) ed opzionalmente emette altre tuple.

La topologia è un DAG di spouts e bolts, nella quale ciascun spout/bolt esegue solo un task, quindi durante la definizione della topologia bisogna specificare come i spout e bolt sono tra loro collegati, il partizionamento dei dati dello stream fra i bolt è detto **stream grouping**, i vari tipi di stream grouping sono:

- Shuffle: il partizionamento è randomico, i spout mandano in maniera randomica i dati verso i bolts a cui sono collegati;
- Fields: il partizionamento è basato su un campo della tupla, può essere utile ad esempio nel conteggio di tuple con un certo valore in un campo;
- All: ciascun bolt riceve una istanza di una tupla;
- Custom;
- Direct: già la sorgente decide a quale bolt mandare la tupla;
- Global: Tutte le tuple generate da tutte le istanze di una sorgente vengono mandate a un unico target.

Il cluster totale è gestito da Nimbus che calcola gli assignment e li invia allo ZooKeeper e lo ZooKeeper li invia ai vari supervisori sui vari cluster che scaricano la topologia da Nimbus, questa topologia viene mandata in esecuzione attraverso un processo Java. Lo ZooKeeper si preoccupa anche di controllare la salute dei vari nodi con HeartBeat e se un Worker muore, il supervi-

sore lo riavvia e se continua a morire Nimbus assegna il worker (che può contenere più di un operatore) a un altro supervisor, mentre se muore tutto il nodo il lavoro viene assegnato ad un altro nodo. Se dovesse morire Nimbus, i supervisor continuano a lavorare ma il riassegnamento diventa impossibile, quindi conviene mettere Nimbus con alta disponibilità (magari tenerne uno di backup), mentre Zookeeper è fault tollerant per se.
I processi che garantiscono l'affidabilità (ad es. che una tupla venga per forza vista) sono:

- Ack and Fail: Ciascuna tupla generata ha un unico ID e ciascuna tupla processata deve essere ackata e fallita, in caso di processamento completato viene generato un ack e fail in caso di timeout, il messaggio viene tracciato da un task detto "acker" in caso di fallimento lo spout può far ripartire il messaggio se è programmato in tale modo;
- Anchoring: Un bolt può emettere una nuova tupla ancorata alla tupla input, quindi nel caso del fallimento lo spout tuple alla radice del DAG andrà rieseguito.

Esiste una libreria esterna Trident che permette di scrivere ad alto livello anche architetture molto complesse, mentre rimane stateful senza che sia l'utente a basso livello a definirla, permette di lavorare al livello micro batch, cioè non lavora ogni singolo messaggio che gli arriva ma accorpa un numero di messaggi, questo migliora il throughput ma peggiora la latenza. Riesce a passare da at least one di storm a exactly once.

9.2 Spark Streaming

Spark è fatto per funzionare con l'idea del batch, per implementare lo streaming sfrutta il così detto mini-batch, quindi si aumenta la latenza ma si alza anche il thoughput. Spark streaming quindi porta con sé tutte le proprietà già esistenti in spark tra cui: distributività, disponibilità e scalabilità. Il funzionamento inizia con i receiver che ricevono e emettono batch che in spark diventano RDD.
Grazie a spark streaming la creazione di un'architettura lambda avviene non solo nello stesso linguaggio ma addirittura nello stesso programma.
Alla base di spark ci sono gli RDD che possono essere distribuiti e replicati in memoria, quindi in caso di un problema di un nodo non perdo i dati, e grazie alla lineage posso anche capire da quale punto si dovrebbe far ripartire la computazione (a differenza di storm che deve far ripartire tutto). I minibatch discretizzati RDD si chiamano **Dstreams**, la loro misura è fatta in modo da avere una latenza di circa 1 secondo e hanno un potenziale di potersi combinare con i batch proccessing. Inoltre con la funzione union è stato possibile implementare il concetto di finestra.
Il master continua a salvare gli stati dei Dstream in un file checkpoint e in caso di fallimento del master questo può essere reinizializzato a partire da questo file checkpoint, permettendo un recupero rapido.
I Dstream richiedono una buona conoscenza di programmazione, quindi si è creata una sua astrazione sotto il nome di Spark Structured Streaming, che ha delle API a livello più alto rispetto a Dstreams. I Dstreams mostrano lo streaming come delle tabelle dinamiche, mentre le operazioni vengono eseguite incrementalmente, quindi il risultato è un altro dataframe con dati variabili nel tempo. L'analisi dei dati provenienti dal Output mode viene eseguita in base a un trigger: che può essere una funzione della sessione temporale.

9.3 Amazon Big Data

Amazon è il leader nel cloud computing, e offre un sacco di servizi direttamente lui pronti spesso anche per la scalabilità quindi non vi è neanche il bisogno di configurarli o mantenerli, e se un servizio non è direttamente offerto, è molto probabile che venga offerto nel marketplace.
La Big Data pipeline è costituita a un livello alto da:

- Data Ingestion/Data collection;
- Data Storage;
- Processare e analizzare i data;
- Consumare o visualizzare.

e il suo obiettivo è minimizzare la latenza e il costo e massimizzare il throughput, nel cloud cerco di minimizzare la latenza ad esempio raggruppando le macchine vicine e inoltre massimizzo il thorughput grazie alla scalabilità dello storage e tutte le macchine leggono alla stessa velocità quindi la velocità di lettura cresce linearmente col crescere dei nodi e per quanto riguarda i costi essi vengono minimizzati poiché pago solo ciò che uso. Il problema rimane il passaggio totale al cloud, ad esempio perché può essere costoso trasmettere tutti i dati, oppure potrebbero esserci problemi di privacy, esistono anche degli approcci ibridi in cui è possibile usare il proprio cloud per dati privati e Amazon per dati normali.
Lo stream storage di amazon è possibile con: Apache Kafka (esiste una cosa simile anche in amazon detta Amazon MQ), Amazon Kinesis e Amazon DynamoDB (simile a SQL streaming di spark). I file storage offerti possono essere HDFS, S3 e Amazon Glacier, sono tutte e tre supportati da Big Data Framework e sono altamente disponibili, la scelta avviene se i dati sono caldi (vengono acceduti molte volte) qua conviene usare HDFS o se sono freddi (pochi accessi)

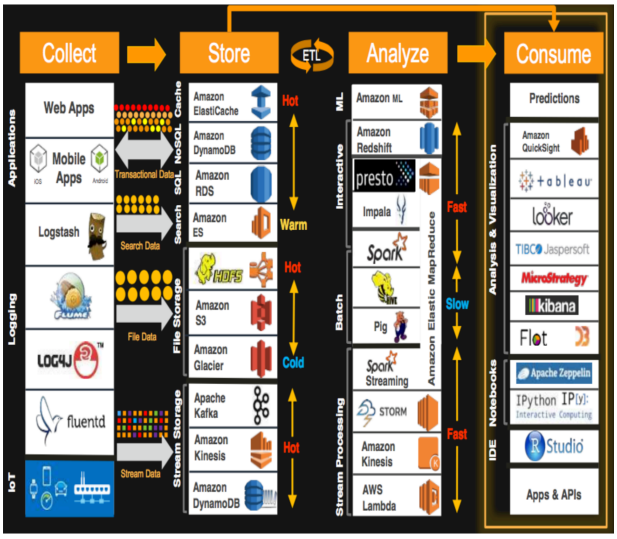

Figure 6: Insight of the Amazon Services

qua conviene Glacier, mentre S3 è una via di mezzo. Infine per i database e search engines abbiamo: Amazon ElastiCache, DynamoDB, RDS e ES.

Part II - Prof Melen