ALMA MATER STUDIORUM – UNIVERSITÀ DI BOLOGNA
CAMPUS DI CESENA

Scuola di Scienze
Corso di Laurea in Ingegneria e Scienze Informatiche

ANALISI COMPARATA DI TECNOLOGIE OPEN-SOURCE PER L'ELABORAZIONE DI FLUSSI DI DATI

Elaborato in
PROGRAMMAZIONE AD OGGETTI

Relatore
Prof. ANDREA OMICINI

Co-relatore
Dott. GIOVANNI CIATTO

Presentata da
MATTEO MINARDI

Seconda Sessione di Laurea
Anno Accademico 2018 – 2019

PAROLE CHIAVE

Big Data

Sistemi Distribuiti

Stream Processing

Lambda Architecture

Dedicato a chi non c'è più e a chi non ha mai smesso di esserci.

Indice

Elenco delle tabelle

Elenco delle figure

Introduzione

L'aumento della produzione di dati nel panorama informatico ha segnato da anni la nascita di vere e proprie realtà che basano le loro decisioni su informazioni estrapolate da elaborazioni intelligenti sui "dati". Le fonti che producono dati sono sempre di più, aumentano ogni giorno in maniera esponenziale. La quantità di dati prodotta supera di gran lunga le aspettative di qualche anno fa. L'obiettivo ora non è più quello di generare dati precisi e derivanti da dispositivi nuovi. Al giorno d'oggi qualsiasi smartphone possiede centinaia di sensori, ogni dispositivo ha accesso a piattaforme che producono flussi impensabili di dati connessi a informazioni preziose per realtà che hanno a che fare con il settore. Il problema odierno è come elaborare questi dati, come gestire questi flussi infiniti e imprevedibili di dati provenienti da fonti sconosciute e sparse. Una realtà informatica deve essere pronta a questo, una realtà informatica non può permettersi di lasciare indietro l'occasione di ricavare qualcosa da tutto questo.

Così come i dati anche questa tesi intraprende un percorso. Il percorso parte affrontando nel primo capitolo i concetti base del panorama informatico che affronta queste tematiche. Il primo capitolo infatti fornisce un background generale sul *dato* in sè e le sue caratteristiche; prosegue chiarificando *dove* i dati vengono salvati e in che modalità. Ad un livello più tecnico viene spiegato con quali architetture fisiche e con quali pattern progettuali vengono gestiti ed elaborati questi dati in maniera efficiente.

Dopo l'infarinatura generale vengono prese in esame una selezione di tecnologie moderne che manipolano flussi di dati di grandi dimensioni al fine di metterne in chiaro i vantaggi e le caratteristiche. Vengono presentati per ognuna di esse i concetti alla base, le motivazioni e gli obiettivi che rendono tutto ciò indispensabile per la creazione di un sistema complesso di elaborazione di dati.

Capitolo 1

Stato dell'arte

Questo capitolo introduce gli argomenti alla base dello sviluppo di sistemi che elaborano dati, rende chiari i concetti fondamentali per poter progettare e analizzare architetture scalabili, affidabili e distribuite su più macchine indipendenti tra di loro. Inizialmente viene preso in esame il mondo dei Big Data, vengono analizzate le caratteristiche del dato e il suo flusso di vita all'interno di un sistema. Successivamente viene presentata l'architettura più comune di un sistema Big Data con un background generale di quelli che sono i sistemi distribuiti e come vengono applicati in questo ambito.

1.1 Big Data

Il termine "Big Data" [1] non ha una definizione univoca: rappresenta concettualmente un trend tecnologico nato negli anni duemila come conseguenza dell'enorme espansione della quantità di dati disponibili grazie al progresso tecnologico e all'orientamento sempre più *data driven*, ossia guidato dall'informazione del dato. Questo processo coinvolge ambiti scientifici, industriali e di business operando quindi su un'enorme varietà di dati, introduce la complessità della raccolta ed elaborazione di alti volumi di dati e con l'aumentare delle entità in grado di produrli questo processo avviene ad elevate velocità, ossia un *dataset* può aumentare di dimensioni in un lasso di tempo molto breve visto che le sorgenti che producono dati sono tante e continuano a crescere esponenzialmente. Detto questo si esce dagli schemi classici di collezionamento, elaborazione e persistenza dei dati. Ora i sistemi sono messi a dura prova e richiedono garanzie di servizio molto elevate incrementando così il livello di complessità dello sviluppo.

1.1.1 Caratteristiche generali

I Big Data vengono descritti attraverso cinque proprietà base dette le 5V ossia cinque caratteristiche che li identificano: volume, varietà, velocità, valore e veracità.

- **Volume**: è la proprietà più ovvia ed è la prima che viene in mente quando ci si riferisce ai Big Data. I dati vengono generati ogni secondo da tantissimi servizi: social media, log di software, dati generati dagli utenti e hardware (IoT). Questo implica che questi dati devono essere conservati per poi essere successivamente utilizzati. I metodi tradizionali di persistenza dei dati non sono più una opzione percorribile, scalare verticalmente, quindi aumentare lo spazio di archiviazione o replicare i dati con metodologie RAID diventa costoso in termini di prestazioni e costoso in termini di infrastruttura fisica. Salvare i dati in un'unica copia è rischioso perché in caso di guasto tutti i dati andrebbero persi. Sempre più dati sono salvati in server remoti, architetture cloud o database. Secondo uno studio [2] è stato stimato che il volume totale dei contenuti nel web sarà di circa 40 *Zettabytes* (2^{21} byte);
- **Varietà**: i dati in ingresso provengono dalle fonti più disparate, sono generati in maniera differente, hanno contenuto diverso e molte volte non strutturato. I dati hanno formati diversi (XML, JSON, CSV ecc...), che cambiano nel tempo con l'evolversi della tecnologia. Non sempre è nota la destinazione dei dati quando vengono prodotti quindi la raccolta implicitamente necessita di dare una forma al dataset dandogli un significato;
- **Velocità**: indica il ritmo con cui i dati vengono processati all'interno del sistema. Solitamente formano un flusso continuo ed è molto importante mantenere alto quindi il *throughput*. Avere tanti dati disponibili e non essere in grado di elaborarli significa non sfruttare appieno le possibilità del sistema generando un collo di bottiglia che grava sulla qualità del servizio. All'aumentare della velocità aumenta anche la reattività del sistema rendendo così i dati subito disponibili aumentando il valore dell'applicativo. Un sistema reattivo è infatti in grado di fornire risposte, quindi informazioni in un lasso di tempo inferiore;
- **Valore**: avere un quantitativo enorme di dati grezzi, quindi senza significato, non porta vantaggi a chi li possiede. Dare un significato reale, creare informazione attraverso questi dati incrementa le possibilità di trarre reale valore e benefici lato business;
- **Veracità**: definisce l'accuratezza dei dati raccolti e dei dati prodotti. Avere dati sbagliati è peggio che non avere dati affatto. I dati devono raccontare il vero e chi li utilizza deve poter contare su di essi.

1.1.2 Valore e utilizzo

Lo scopo dello studio e dell'analisi dei Big Data non è soltanto quello di raccogliere informazioni in maniera organizzata ed efficiente ma è soprattutto quello di dare un valore [3] negli ambiti applicativi in cui vengono utilizzati. Questi dati, se utilizzati in maniera corretta, permettono un miglioramento della qualità delle previsioni, quindi forniscono strumenti efficaci per prendere decisioni più consapevoli e affidabili. Oltre all'utilizzo primario però, secondo Cukier e Mayer-Schönberger [4], il valore è dato anche dai possibili utilizzi secondari ricavabili se si ricombinano i dati. Il valore secondario deriva quindi dal possibile riutilizzo in futuro: la fusione e la combinazione di più dataset che permettono di ottenere informazioni combinate e arricchite. Tuttavia conservare questi dati per lunghi periodi può essere controproducente visto che hanno una rapida obsolescenza e a causa dei rapidi cambiamenti dei trend tecnologici perdono valore.
La sfida e le opportunità che i Big Data portano con sè possono essere visti da tre aspetti. Il primo è quello di *business*: i dati aprono la possibilità di nuovi modelli di business che consentono di ottenere vantaggi su quelli tradizionali. Un accesso facile e tempestivo ai Big Data rende disponibile una maggiore quantità di informazioni e facilita la condivisione dei dati tra le diverse unità organizzative di un'impresa. Per esempio, i dati delle unità di ricerca e sviluppo, produzione e ingegneria di un'azienda possono essere integrati al fine di favorire lo sviluppo concorrente di un prodotto, tagliando i tempi e migliorando la qualità. L'analisi della clientela acquisita o potenziale ha portato le aziende ad elaborare dati non strutturati. Un tempo il possesso di questi dati era sufficiente nei sistemi a profilare un cliente ma ora con l'unione tra questi e quelli non strutturati entrano in gioco analisi più avanzate [5] come:

- **Analisi di pattern comportamentali**: la disponibilità real time di come si comporta l'utente con il proprio device fornisce caratteristiche dettagliate sulle decisioni intraprese dal cliente;
- **Personalizzazione dell'esperienze**: dopo aver analizzato il comportamento e raccolto dati relativi alle preferenze è vantaggioso personalizzare le proposte e i suggerimenti in base ai parametri in possesso. L'utente finale così percepisce un'esperienza adatta a lui con soltanto i suoi interessi;
- **Aumento di performance di conversione**: la memorizzazione di dati relativi a indici di guadagno e stato dell'azienda consentono di capire e analizzare la variabilità delle prestazioni. Capire più a fondo queste dinamiche aziendali può portare vantaggi sulla gestione del business;
- **Migliorare previsioni**: l'analisi dello storico del passato porta sicuramente ad una previsione, almeno teorica, del futuro. Questa previsione

facilita un business a compiere decisioni ponderate su dati tangibili;
- **Supporto al decision making**: utilizzando strumenti adatti e dataset affidabili è possibile automatizzare processi decisionali, scoprire suggerimenti con lo scopo di migliorare modelli futuri per creare servizi innovativi.

Il secondo è quello *tecnologico*: la dimensione e la complessità delle strutture in ingresso rendono impossibile l'utilizzo di architetture e paradigmi classici utilizzati in passato. Sono necessarie figure professionali e tecnologie nuove che affrontino i problemi derivanti dai Big Data. Come vedremo nel Capitolo 2 possono essere utilizzati diversi strumenti individualmente o in combinazione con diversi scopi:

- *Data transfer*: questi strumenti sono utilizzati per introdurre i dati introdotti in tempo reale all'interno del sistema e smistarli in maniera organizzata trasferendoli da un punto all'altro. Comunemente vengono utilizzati dei bus o delle code di messaggi;
- *Data processing*: questi strumenti sono il cuore dell'elaborazione. Estraggono informazioni dai dati, effettuando delle mutazioni al dataset iniziale al fine di estrapolare valore che poi verrà fornito all'utilizzatore finale per migliorare l'esperienza;
- *Data storage*: questi strumenti sono usati come appoggio per i dati durante tutte le fasi dell'elaborazione. Questi solitamente sono file system distribuiti che hanno caratteristiche diverse da quelli comuni. Questi sistemi consentono aumentare le performance e la capienza semplicemente aggiungendo macchine e non incrementando la capienza dei dischi su una singola macchina.

Ultimo ma non meno importante è l'aspetto *finanziario*. Indubbiamente i Big Data rappresentano un'opportunità economica però occorre valutare a priori i costi per l'implementazione di tali soluzioni. Bisogna valutare la possibilità di utilizzare hardware a basso costo o per limitare l'investimento iniziale utilizzare tecnologie cloud a consumo che impattano meno l'economia di aziende con un traffico base contenuto.

1.1.3 Proprietà dei sistemi Big Data

Durante la realizzazione di un sistema Big Data la difficoltà principale non ricade sulla complessità algoritmica ma piuttosto sulla realizzazione di un'architettura scalabile. Ora vengono elencate le proprietà necessarie per realizzare un sistema in maniera efficiente.

Robustezza e tolleranza agli errori

Costruire sistemi che eseguano le operazioni giuste in maniera coerente in un gruppo di più macchine è una delle sfide più ardue durante lo sviluppo di sistemi distribuiti che operano su grandi quantitativi di dati. Con robustezza e tolleranza agli errori si intende che il sistema deve continuare a operare correttamente anche nel caso una macchina smetta di funzionare, ci siano inconsistenze nei dati, si operi in maniera concorrente sui dati o ci siano dati duplicati. Questo principio implica che il sistema sia tollerante anche all'errore umano ossia continui a funzionare anche se vengono commessi errori da parte di chi amministra il sistema o ne scrive il codice.

Bassa latenza

Quando viene richiesta una determinata informazione deve essere data una risposta entro un lasso di tempo molto breve. Il lasso di tempo si deve aggirare nell'ordine dei millisecondi altrimenti la richiesta effettuata perderebbe il suo valore informativo. L'incremento delle prestazioni però non deve impattare la robustezza del sistema. Spesso fornire garanzie costituisce un compromesso con la velocità delle operazioni però bisogna tenere conto anche del vantaggio di garantire consistenza e robustezza. L'overhead di queste operazioni garantisce continuità di servizio e il recupero dei dati anche in caso di guasti.

Scalabilità

É una delle proprietà più significative e indica come il sistema possa evolversi con facilita con il cambiamento del carico di dati e del carico computazionale. I sistemi Big Data generalmente scalano in maniera orizzontale [6] ossia significa che la potenza del sistema aumenta aggiungendo elementi nel gruppo di risorse invece che aggiungere potenza al singolo elemento. Un elemento isolato è incline a guasti o problemi di rete che possono provocare interruzioni o perdite di dati.

Estendibilità

Un sistema software Big Data deve essere sviluppato in modo che l'aggiunta di nuove funzionalità e l'ampliamento di quelle già presenti non comporti un costo di sviluppo alto. Il sistema, essendo sempre in evoluzione, risulta essere dinamico e incline al cambiamento vista la rapidità con cui aumentano i volumi dei dati.

Ottimizzazione

I dataset sono formati da tantissimi dati e poter ottenere quelli più utili in maniera efficiente è una proprietà importante quando si progetta un sistema di questo tipo. Poter interrogare i dati in più modi ottimizza il risultato dal punto di vista business e fornisce interfacce semplici per ottenere i dati calcolati.

Bassa manutenzione

La manutenzione ha un costo molto alto in termini di sviluppo ed è un'attività indispensabile per fare in modo che il sistema continui ad operare. Più il sistema è complesso più è alta la probabilità che sia richiesta analisi, debug e documentazione tecnica delle funzionalità. Bisogna anche tenere conto che i problemi sulle strutture interne dei sistemi distribuiti sono molto complesse e difficili da riprodurre per cui è fondamentale l'utilizzo di framework di test automatici, log di debug e benchmark preliminari.

Debug

Costruire un sistema che elabora dati e che ricava informazioni attraverso processi complessi deve fornire gli strumenti per effettuare debug in maniera rapida e chiara. La soluzione è quella di strutturare bene come vengono forniti log, come vengono gestiti gli errori imprevisti e come riprodurli in ambienti di test dedicati a questo scopo.

1.1.4 Criticità e problemi

I Big Data sono una realtà consolidata ormai e come tale questo trend sta cominciando ad avere impatto anche in ambiti delicati come la politica, la sanità e i servizi pubblici. Nel caso in cui vengano utilizzati male, i vantaggi potrebbero non essere sfruttati appieno o addirittura potrebbero trasformarsi in ulteriori problemi da gestire invece che semplificare i processi. Principalmente sorgono tre [7] criticità: i limiti fisici, la qualità del servizio e la privacy dei dati.

La prima criticità riguarda i limiti fisici che impone un'architettura che spazia esponenzialmente come questa. La progettazione di tali sistemi deve tenere in considerazione che le valutazioni e le stime fatte potrebbero essere ampiamente superate. L'andamento infatti, implicitamente fa capire che lo spazio per archiviare e la potenza computazionale necessaria per elaborare deve poter scalare. Anche se nel presente l'architettura rispetta le dimensioni probabilmente nel futuro non lo farà.

In secondo luogo, bisogna prestare attenzione e adottare gli accorgimenti necessari per evitare di utilizzare dati poco affidabili. La qualità è determinata da diverse caratteristiche:

- *Accuratezza*: quanto poco i dati si discostano da quelli reali quindi quanto sono corretti.
- *Consistenza*: quanto i dati sono coerenti e privi di contraddizioni interne.
- *Completezza*: la presenza di tutti i dati necessari per raggiungere il proprio obiettivo.
- *Univocità*: indica l'assenza di duplicazione dei dati. La duplicazione incide negativamente sulla qualità e necessita di manutenzione in modo da correggere l'errore.

Quando si affrontano tematiche legate ai Big Data non bisogna trascurare le implicazioni derivanti dal trattamento di certi tipi di dati e la loro analisi. Dal web è possibile profilare le persone, sono disponibili molti dati che sono accessibili da tutti ma questo non significa che sia lecito utilizzarli. Dai social network è possibile estrapolare informazioni sensibili che permettono di risalire a orientamento politico, religioso e sessuale sia in modo diretto (ossia da esplicita dichiarazione della persona) sia attraverso analisi dei contenuti. Oltretutto è molto difficile non lasciare tracce comportamentali online, basta pensare ai pagamenti elettronici, i dispositivi GPS integrati negli smartphone e tutti i sistemi informativi pubblici e sanitari a cui siamo registrati. L'errata gestione di questi dati è pericolosa e non è semplice da affrontare, per questo rappresenta uno degli aspetti critici più rilevanti.

1.2 Architetture Big Data

1.2.1 Lambda Architecture

La *Lambda architecture* [8] è un pattern architetturale consolidato nell'ambito Big Data descritto di Nathan Marz noto per essere uno dei principali contributor del progetto open source *Apache Storm*. Questo pattern implementa una metodologia generica in grado di fornire una organizzazione modulare, fault-tolerant e real-time del proprio sistema di elaborazione dei Big Data. Questa architettura fornisce un approccio generale per per applicare una funzione, quindi una elaborazione, su un dataset arbitrario ottenendo il risultato con una latenza bassa. Nessuno strumento singolarmente fornisce la possibilità di elaborare in maniera dinamica flussi di dati arbitrari in tempo reale. Per questo è necessario utilizzare un gruppo di tecnologie divise in moduli separati in cui ogni modulo si occupa di un ruolo ben preciso.

L'idea principale della Lambda architecture è quella di organizzare un sistema Big Data suddividendo il progetto in una serie di livelli che svolgono operazioni su un sottoinsieme di dati invece che sul dataset in ingresso.

```
query = function(all data)
```

Questa funzione applica un'elaborazione troppo costosa che, nell'ambito dei Big Data, non è possibile eseguire visti i volumi enormi su cui si va ad operare e anche se lo fosse, impiegherebbe troppo tempo per poter fornire un risultato utile in un lasso di tempo ragionevole.

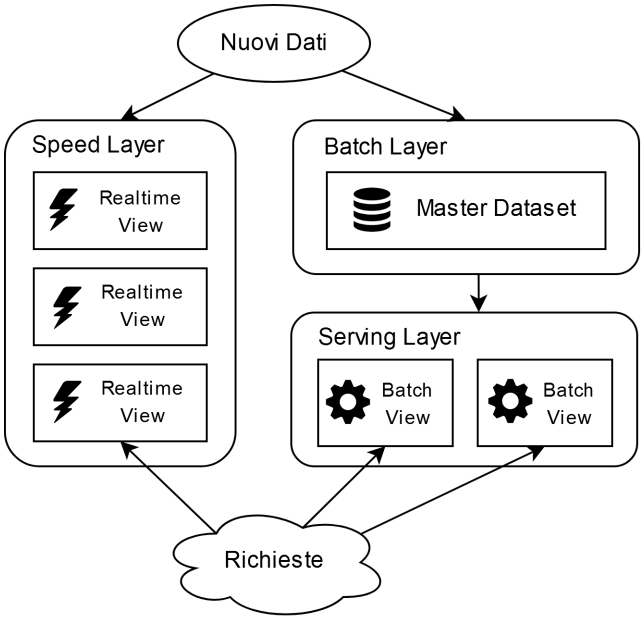


Figura 1.1: Livelli della architettura Lambda [8].

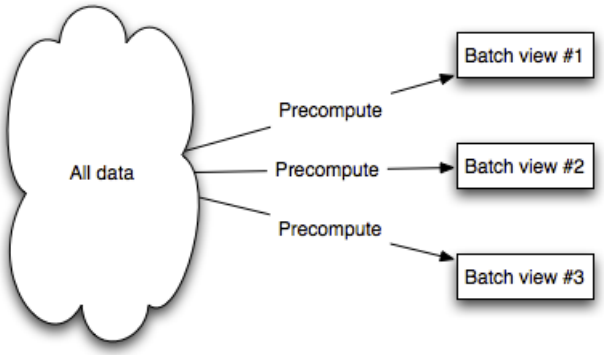
Per questo motivo ogni layer ha un suo scopo, una sua particolarità che lo rende indispensabile per garantire scalabilità al sistema e i risultati in tempo reale. I tre livelli, come visibile in figura 1.1 sono: Batch layer, Serving layer e Speed layer.

Batch Layer

Un'architettura orientata ai dati solitamente effettua operazioni su un dataset in modo da poter fornire risposte a interrogazioni. Idealmente le interrogazioni sono come funzioni che operano su tutti gli elementi e forniscono un risultato che è un sottoinsieme dei dati di partenza. Questo non può funzionare nell'ambito dei Big Data per due motivi: in primo luogo il numero di

elementi è molto grande e quindi l'operazione è molto pesante. Il secondo motivo è che anche se il risultato venisse prodotto impiegherebbe troppo tempo e quindi perderebbe di significato. Il livello *batch* quindi, per risolvere questo problema, svolge due funzioni:

- Salva il dataset principale in memoria.
- Suddivide il dataset iniziale in sottoinsiemi detti *viste*. Queste viste devono essere calcolate in modo che la funzione iniziale calcolata su di esse fornisca il risultato velocemente.



Con il crescere del dataset il livello batch si calcola nuovamente le viste da capo e applica l'elaborazione richiesta. Questo calcolo ripetuto permette di eliminare errori sulle viste esistenti correggendo i problemi sul dataset principale. Solitamente queste viste vengono salvate su un database di appoggio riscrivendo i risultati precedenti.

```
batch view = function(all data)
query = function(batch view)
```

L'elaborazione di queste viste, detta batch-processing, risulta essere semplice da implementare visto che opera come un processo indipendente su una vista singola. Questo la rende una tecnica robusta e implementa il parallelismo banalmente aggiungendo macchine che andranno ad effettuare l'elaborazione su altre viste.

Serving Layer

Questo livello è strettamente legato a quello Batch infatti come si vede in figura 1.1. Il livello *Serving*, dopo aver ricevuto in ingresso le viste calcolate precedentemente ha il compito di accedervi in maniera efficiente e a bassa latenza. Il livello batch con l'allargarsi del dataset fornisce viste aggiornate

e questo livello livello espone le interfacce piu appropriate per interrogare gli aggregati pre calcolati. Questi due livelli insieme soddisfano gran parte delle proprietà necessarie in un sistema Big Data, forniscono un sistema robusto con un'architettura distribuita della persistenza dei file, forniscono scalabilità e modularità data la presenza di viste separate che possono essere elaborate contemporaneamente e allo stesso tempo è possibile evolvere la funzione utilizzata per calcolare le viste cambiando così anche come vengono presentate con il livello serving.

Il livello serving quindi si specializza nella gestione di un database distribuito che prende le viste batch e fornisce la possibilità di effettuare letture performanti su di esso. Non necessariamente deve essere possibile fare scritture in maniera efficiente perché il ruolo di questo livello è principalmente concentrato sulle letture e sulla presentazione del dato potendo rinunciare così alle scritture. Le scritture in un database rappresentano il grado di complessità maggiore quindi rinunciandovi è possibile costruire database molto semplici che su cui è facile effettuare le operazioni. Le viste fornite sono sempre relative all'ultimo aggiornamento delle viste batch ed intrinsecamente non sono aggiornate in tempo reale. Questo non è considerato come uno svantaggio visto che i dati sono più accurati e che è i dati aggiornati in tempo reale sono disponibili sul livello *speed*.

Speed Layer

Con gli altri due livelli i dati vengono presentati soltanto quando il batch layer ha terminato l'elaborazione delle viste. Questo significa che per un lasso di tempo i dati non sono aggiornati. Per compensare la latenza degli altri due livelli l'ultimo, quello del livello *speed*, ha come obiettivo quello di raggiungere una maggiore reattività attraverso l'elaborazione dei dati in tempo reale. Per fare ciò i dati di questo livello non sono calcolati basandosi su tutto il dataset principale ma fanno riferimento ad una finestra temporale più ristretta, i dati infatti sono meno rispetto agli ordini di grandezza del livello batch e vengono gestiti in maniera più snella e modulare. Il livello speed è basato su un modello di computazione *incrementale* che, diversamente da quello batch, aggiorna dati già esistenti effettuando delle elaborazioni appunto incrementali. Il modello incrementale quindi non si impegna a salvare tutti i dati ma effettua tanti piccoli aggiornamenti su dati esistenti. Questo modello però aggiunge un grado di complessità elevato alla logica dell'applicazione quindi il calcolo e lo sviluppo di algoritmi incrementali risulta essere più difficile e incline ad errori.

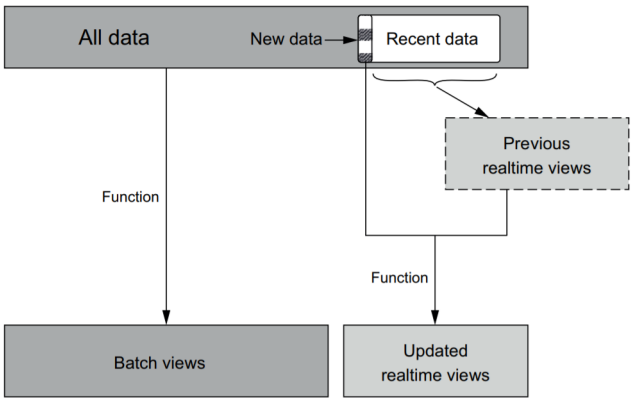


Figura 1.2: Strategia incrementale del livello speed [8].

In figura 1.2 è possibile notare come vengano selezionati diversamente i record coinvolti nell'elaborazione dei dati nei diversi livelli. L'idea è quella di ricreare viste real-time aggiornate ogni volta che vengono inseriti dei record nel dataset e quindi modificare le viste precedenti tenendo in considerazione soltanto i nuovi dati. Questo calcolo è molto più snello e si può riassumere con questa formula:

```
realtime view = function(prevous realtime view, new data)
```

Vantaggi e svantaggi

L'architettura Lambda per come è progettata fornisce l'immutabilità dei dati visto che mantiene su disco tutta la storia degli eventi in modo persistente. Struttura il suo modello in diverse fasi, ognuna delle quali modifica il dataset principale in modo modulare. Questo è un vantaggio sia perché separa concettualmente le operazioni sia perché rende più immediato dove intervenire in caso di problemi e debug.

Un altro vantaggio è che gestisce molto bene il ricalcolo delle viste in caso di errori. Il ricalcolo può avvenire con l'evoluzione dell'applicazione, con la scoperta di nuovi bug di cui è necessaria la correzione immediata. In questo caso è sufficiente aggiornare le viste fornite al livello serving. Se l'architettura non fosse in grado di ricalcolare le viste di output significherebbe che non potrebbe evolversi con l'avvenimento di nuove feature e renderebbe molto complicata la gestione di problemi legati a come viene generato l'output.

Questa architettura secondo l'articolo [9] di Nathan Marz ha come scopo quello di trovare una soluzione solida al popolare problema del Teorema CAP.

Teorema CAP Questo teorema [10] afferma che non esiste un modo in un sistema informatico distribuito di fornire allo stesso momento tutte e tre le seguenti garanzie:

- **Consistency – Coerenza**
  Ogni richiesta a qualsiasi nodo dello stesso dato deve fornire la stessa risposta oppure una più aggiornata rispetto alle altre;
- **Availability – Disponibilità**
  Ogni richiesta effettuata ad un nodo non in errore deve ritornare una risposta con successo. Una risposta corretta e veloce è preferibile ma per questa garanzia basta che venga ricevuta una risposta perché una risposta eventuale può creare più problemi di una risposta non consistente;
- **Partition Tolerance – Tolleranza di partizione**
  Visto che la comunicazione tra servizi separati intrinsecamente non è affidabile la perdita di messaggi può creare delle partizioni, ossia gruppi di nodi che non riescono a comunicare per periodi di tempo più o meno brevi. Nonostante queste arbitrarie perdite di comunicazione tra un nodo e l'altro il sistema deve continuare a funzionare.

Il problema non è risolvibile quindi è comunque necessario prendere una scelta su consistenza o disponibilità del sistema però semplifica la sua complessità.

Il problema della architettura Lambda è mantenere il codice che deve produrre lo stesso risultato sia nel livello batch sia nel livello speed. Entrambi i sistemi utilizzano tecnologie distribuite ed è molto complesso gestire due progetti di questo tipo in modo da ottenere lo stesso output finale. Inevitabilmente la tecnologia scelta per i vari livelli influisce sull'approccio, le scelte di sviluppo e di conseguenza sull'organizzazione dell'architettura. Un aspetto negativo comune e non risolvibile è la separazione del codice in due framework diversi che necessitano di manutenzione e debugging.

Per evitare questo problema è stata introdotta una semplificazione dell'architettura Lambda detta *Kappa architecture*.

1.2.2 Kappa Architecture

In un articolo Jay Kreps [11] principale sviluppatore di *Apache Kafka*, piattaforma distribuita per lo streaming e lo scambio di messaggi, ha portato in evidenza le criticità dell'architettura Lambda. Nell'articolo ha proposto una valida alternativa, chiamata in seguito architettura Kappa, in cui si pone come obiettivo quello di snellire la divisione in tre livelli rimuovendo il livello batch. La principale criticità del modello Lambda infatti è di dover duplicare la logica dell'applicazione sia sul livello *batch* sia sul livello *streaming*. Duplicare la logica implica anche riscrivere il codice in più livelli per ottenere un risultato

di elaborazione simile con diversi framework che spesso hanno architetture diverse difficili da mantenere.

La Kappa architecture quindi è un pattern architetturale in cui il flusso di elaborazione dati passa sotto forma di streaming attraverso un singolo percorso, mediante un sistema di elaborazione parallelo. Il ricalcolo delle viste viene effettuato quando avviene un evento, non periodicamente come il livello batch e le viste utilizzate dal livello serving ricevono degli aggiornamenti incrementali.

Innanzitutto, rimuovere il livello batch implica meno codice da mantenere e ci si può concentrare più sul modello logico target che, in questo caso, è univoco. In secondo luogo, al giorno d'oggi le tecnologie di streaming ed elaborazione in tempo reale forniscono tutti gli strumenti per ottenere un output consistente e tollerante ad errori.

In questa architettura, visto che non è presente un livello batch con tutte tutti i dati conservati all'interno, in caso di errore non è semplice recuperare i risultati di elaborazione passati. É comunque possibile gestire il ricalcolo incrementale dell'output attraverso i dati presenti. Un altro vantaggio per sistemi dinamici che spesso cambiano l'organizzazione è sulle migrazioni. Questa metodologia è più snella e non è necessario spostare tutto il dataset in un altro database ma è sufficiente ripopolare le viste ricalcolando l'output del livello speed.

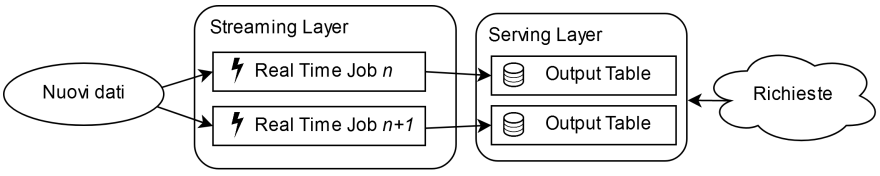


Figura 1.3: Divisione in livelli dell'architettura Kappa [8].

Come illustrato in figura 1.3 questa architettura sono presenti quindi soltanto due livelli:

- *Livello streaming*: questo livello riceve un flusso indefinito di dati e li elabora in modo incrementale gestendo parallelamente più unità di lavoro separate dette *job*. L'input di questo livello non deriva da un sistema di storage ma viene fornito da un sistema che scrive in modalità *append only* in un log di eventi immutabile.
- *Livello serving*: questo livello ha come obiettivo il fornire risposte rapide alle richieste dell'utente. Le richieste vengono dette *query* e rappresentano una funzione che interroga i dati in modo da ottenere una risposta efficace. In questo livello può essere utilizzato qualsiasi tipo di database, sia in memoria sia su disco. L'importante è tenere a mente che non

verrà usato come un database classico per la persistenza consistente dei dati ma ha lo scopo di fornire dati in maniera rapida e deve fornire gli strumenti per poterli cancellare completamente.

Conclusioni

La scelta dell'architettura è un passaggio importante che comporta un tradeoff. Se si sta sviluppando un modello affidabile che aggiorna continuamente le viste e calcola in modo robusto i risultati allora l'architettura Lambda resta la scelta migliore considerato il vantaggio fornito dal livello batch. Invece se si vuole puntare su un'architettura più diretta e che necessita di meno capacità hardware di storage ed elaborazione allora la scelta migliore è orientarsi su un modello Kappa per l'elaborazione in tempo reale.

1.3 Sistemi e applicativi Big Data

In questa sezione vengono elencate le fasi che percorre un dato quando, dopo essere stato prodotto, entra a far parte di un sistema Big Data. Durante il ciclo di vita i dati, rappresentati sotto forma grezza, subiscono diverse trasformazioni e vengono spostati in sistemi diversi in modo da acquisire un reale valore informativo. Questo richiede l'utilizzo di nuove e innovative tecnologie che garantiscano una serie di proprietà.

1.3.1 Proprietà delle applicazioni data-intensive

Le applicazioni che trattano grandi quantità dati si trovano tipicamente davanti a problemi di dimensione dei dati, complessità dei dati e la frequenza con cui essi variano. Per affrontare questi problemi vengono utilizzate tecnologie che nascondono la gestione di basso livello del problema fornendo un servizio in grado di risolverlo senza la necessità di creare un'applicazione da zero. Nonostante questo, quando si combinano diversi framework assieme, è l'applicazione che deve essere in grado di coordinare e gestire il lavoro.

Nelle applicazioni *data intensive* quindi bisogna preservare i dati in modo corretto e completo anche quando le cose vanno male. In ogni caso le performance non devono subire grosse penalizzazioni anche quando parte del sistema non è funzionante.

Ci sono molteplici fattori che influenzano la correttezza del design dell'architettura software ma i tre principali [12] sono: affidabilità, scalabilità e manutenibilità.

Affidabilità

É quanto un sistema software è in grado di operare correttamente anche a fronte di malfunzionamenti ed eventi inaspettati. Allo stesso tempo è definita anche da come l'utente finale percepisce il sistema. Solitamente un sistema affidabile viene percepito nei seguenti modi:

- Le operazioni che l'utente si aspetta vengono eseguite;
- L'utente può commettere errori senza compromettere lo stato del sistema;
- La qualità delle performance rimane ottimale a fronte di picchi di carico e volume.

Quando le cose vanno male, ci sono dei problemi all'interno del sistema o ci sono dei fallimenti, il sistema deve essere pronto e in grado di gestire la situazione. Questa caratteristica definisce un sistema *fault tolerant* ossia tollerante agli errori. Un sistema tollerante è un sistema robusto che continua a funzionare a fronte di *certi* tipi di errori. Generalmente non è possibile evitare totalmente che avvengano errori quindi, in un sistema software, si predilige la tolleranza per ottenere affidabilità.

I tipi principali di errori sono:

- *Hardware*: questi si presentano quando componenti delle macchine fisiche che fanno parte di sistemi distribuiti hanno problemi e smettono di funzionare. Questo avviene quando le macchine sono tante e sono in luoghi differenti. Nonostante tecniche di RAID, gruppi di continuità o CPU intercambiabili il problema non può essere completamente prevenuto.
- *Software*: questi sono problemi dovuti a bug software che causano dei crash del sistema, utilizzano in maniera errata le risorse o bloccano il servizio con una errata gestione di operazioni bloccanti. Questi tipi di errori sono difficili da prevedere ma possono essere mitigati da processi di testing, analisi di performance o metriche per la generazione di statistiche dettagliate sull'andamento nel tempo del software in modo da reagire ad anomalie.
- *Errori umani*: sia chi sviluppa software sia chi lo gestisce può incorrere in errori umani. Per definizione l'attività svolta da una persona non è affidabile e quindi può causare problemi.

Scalabilità

Se un sistema con le condizioni attuali funziona correttamente non significa che continuerà a farlo anche a fronte di cambiamenti futuri. La principale ragione del calo di performance è l'aumento del carico di lavoro. Con scalabilità

si intende la capacità di un sistema di gestire in maniera corretta l'aumento del carico di lavoro e nel caso dei sistemi Big Data, di trattare più dati.

Un corretto approccio per garantire questa proprietà è dividere il processo di progettazione dell'architettura in due step:

- *Analisi di carico*: è un processo che implica l'analisi del carico attuale del sistema. É necessario prendere in esame tutti i volumi e identificare i parametri che potranno subire un cambiamento nel corso del tempo. Nel caso di un database un parametro potrebbe essere il numero di scritture effettuate oppure nel caso di un server web il numero di chiamate HTTP ricevute;
- *Analisi di performance*: solo una volta delineati i parametri di carico è opportuno capire come gestire l'aumento dei valori. Per ognuno di essi infatti è buona pratica definire come agire se il carico aumenta ma le risorse rimangono invariate. Se questo impatta gravemente l'applicazione allora è bene migliorare l'architettura oppure garantire continuità anche in caso di degrado delle performance. Una seconda cosa da tenere in considerazione è il costo necessario per mantenere le prestazioni invariate a fronte dell'aumento di carico. Questo significa che una corretta progettazione è predisposta a scalare con pochi costi a livello hardware e in termini di sviluppo e refactor dell'applicazione.

Se si lavora con un servizio che cresce rapidamente bisogna pensare ad un modo per cambiare architettura in maniera semplice e immediata in modo da reagire in tempi brevi. Per scalare un'architettura solitamente esistono due modi [13]:

- *Scalabilità verticale*: questa modalità aumenta le capacità del sistema incrementando hardware e performance di una singola macchina. Questo metodo è semplice perché non va a modificare l'architettura e non introduce complessità all'applicazione;
- *Scalabilità orizzontale*: significa che il sistema scala aggiungendo macchine o introducendo un sistema distribuito che permette di aggiungere hardware di basso costo per aumentare l'elaborazione dei dati. Questo metodo aggiunge complessità di gestione del sistema. Per fare in modo che il carico venga distribuito correttamente è necessario un *load-balancer* che si occupa di decidere quale macchina dovrà occuparsi di una certa operazione. Allo stesso tempo è evidente che si introduce la complessità di coordinamento, gestione e manutenzione di un sistema distribuito.

Una buona architettura solitamente è elastica e implementa entrambe le modalità nel modo più opportuno. Avere un servizio distribuito *stateless* senza uno stato condiviso tra le macchine che effettuano operazioni è semplice, la

complessità però viene introdotta quando più macchine devono avere coerenza tra di loro. In questo caso è più opportuno scalare verticalmente e, ad esempio, mantenere il database su un singolo nodo più potente. Avere un sistema che si adatta automaticamente in base al carico e aumenta le risorse è più efficiente che apportare modifiche manualmente alle macchine ad ogni piccola variazione dei parametri di carico.

Manutenibilità

Il costo maggiore di un sistema software deriva nella maggior parte dei casi dalla manutenzione e non dallo sviluppo e la progettazione. Durante la maggior parte della vita di un software è necessario sistemare bug, mantenere il servizio attivo, analizzare fallimenti, renderlo più efficiente e modificarlo per nuove funzionalità. Rendere un sistema mantenibile significa minimizzare il costo della manutenzione futura. Gestire sistemi software sviluppati in un lungo periodo di tempo è difficile e analizzarlo richiede più tempo se non vengono seguite alcune linee guida che migliorano appunto la predisposizione di un sistema software ad essere mantenuto per lunghi periodi di tempo.

Le linee guida sono:

- Mantenere il progetto il più comprensibile e diretto possibile. Renderlo semplice non significa rimuovere funzionalità ma fare un buon utilizzo dei paradigmi di progettazione, mantenere una buona astrazione e aumentare la qualità del software rendendo più facile il lavoro all'interno di un gruppo di più persone.
- Fornire tutti gli strumenti necessari a chi dovrà gestire e fare analisi del sistema. Poter svolgere monitoraggio fornisce valore e soprattutto alleggerisce il costo di attività di manutenzione. I migliori accorgimenti da prendere sono: dare la possibilità di accedere allo stato interno del sistema, fornire una documentazione su come svolgere attività di base sul sistema e avere dei comportamenti di default corretti.
- Rendere i cambiamenti progettuali semplici da implementare. Frequentemente le specifiche di un progetto subiscono variazione e un'applicazione deve essere in grado di fornire l'implementazione in maniera rapida. Un sistema che non può cambiare è un sistema destinato a non evolversi o per cui i cambiamenti hanno un costo talmente elevato che non è possibile realizzarli.

1.3.2 Ciclo di vita del dato

Nel mondo dei Big Data il protagonista principale è il dato. Il dato è l'unità base dell'informazione, al suo stato grezzo è rappresentato da un flusso

di byte che viene prodotto attraverso un sistema software o hardware. Il dato rappresenta un grande valore per questo tipo di applicativi per questo motivo è molto importante che venga gestito al meglio e in modo affidabile. Il dato quindi intraprende un percorso attraverso una serie di livelli logici che uniti tra di loro permettono di utilizzarlo e conservarlo al meglio.

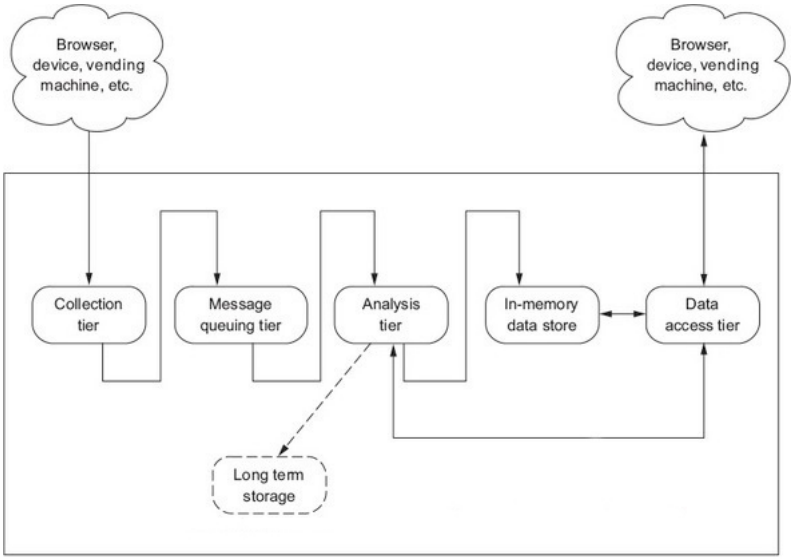


Figura 1.4: Schema concettuale del flusso dei dati in un sistema Big Data

Come rappresentato in figura 1.4, ci sono diversi livelli [14] che formano il percorso però bisogna specificare che nessuno vincola rigidamente l'architettura dell'applicazione. Ognuno di essi può essere visto come un mattone che suddivide il problema in step modulari. L'insieme di questi step rappresenta logicamente un problema da risolvere, soprattutto quando la mole di dati è molto elevata e non è facile gestire tanti dati tutti in una volta. Ognuno di essi quindi si pone l'obiettivo di garantire affidabilità e risolvere un problema ben preciso nella maniera più efficiente possibile. Ogni livello infatti in molti casi viene implementato con un framework distribuito in grado di scalare orizzontalmente a fronte di improvvisi aumenti di carico.

Collezionamento qui i dati vengono introdotti dentro il sistema e inizia il percorso all'interno l'architettura interna. Oggigiorno i dati vengono prodotti principalmente da tre fonti:

- *Social Media*: questi dati vengono generati dalle persone. Ogni persona infatti che ha un profilo social è indirettamente e direttamente un produttore di dati. Ogni azione e comportamento legato ad un attività

viene profilata è utilizzata per produrre valore e informazione. Ogni like, ogni commento e ogni post viene analizzato e trasformato in analisi di mercato utili per la crescita di business e aziende.
- *Internet of Things*: praticamente ogni dispositivo hardware è dotato di sensori in grado di produrre flussi enormi flussi di dati. Questi dati non sono strutturati e sono praticamente privi di significato se presi individualmente. Hanno bisogno di passaggi di elaborazione in cui viene rimosso il rumore delle misurazioni e i dati vengono aggregati in modo da alleggerire il carico di lavoro successivo.
- *Transactional data*: sono dati generati dalle aziende attraverso tutte le attività *data-driven*. Solitamente derivano da fonti conservate staticamente in strutture relazionali che rappresentano pagamenti, ordini, dati di produzione, d'inventario, vendite e dati finanziari.

Una volta capito come i dati vengono prodotti è necessario immagazzinarli. Il ruolo fondamentale del livello di collezionamento è interagire con i sistemi di produzione di dati in modo da riceverli, immetterli nel sistema senza eventuali perdite. Se questo livello fallisce il sistema non è più in grado di ricevere input dai sistemi esterni quindi è fondamentale preservare la tolleranza agli errori.

Code di messaggi Come visto per il livello di collezionamento i dati sono tanti e hanno bisogno di essere incanalati nella maniera più affidabile e corretta possibile. Il motivo per cui è utile implementare il livello delle code di messaggi deriva dal fatto che lavorare con sistemi distribuiti e micro-servizi che lavorano separati fa sorgere la necessità un bus di comunicazione. Questa comunicazione deve avvenire in modo efficace, rapida e affidabile. Il problema si presenta quando nasce la necessità di scambiare messaggi tra i vari livelli senza comunicare direttamente. Avere un legame forte tra questi servizi infatti limita la modularità e il disaccoppiamento dell'architettura.

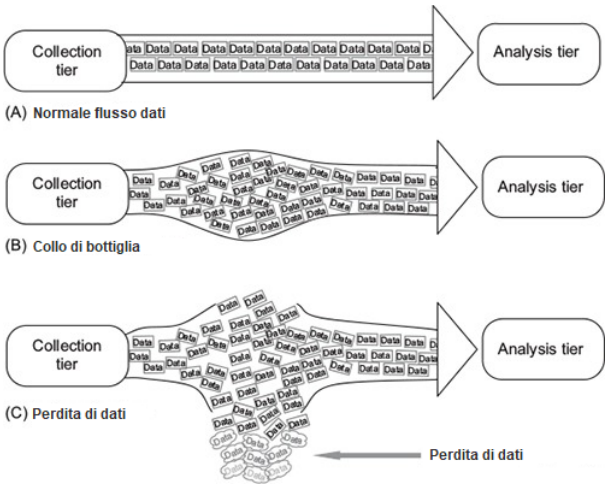


Figura 1.5: Superamento dei limiti di comunicazione [14].

In figura 1.5 è rappresentato come nel primo passaggio il sistema riesca a gestire il flusso dati verso il livello di analisi. Con una comunicazione diretta tra i livelli di collezionamento e analisi si avrebbe sicuramente il problema derivante dall'esponenziale aumento del flusso dati fino alla perdita dovuta ad errori e malfunzionamenti.
Per evitare ciò in un sistema Big Data è necessario fornire una garanzia di affidabilità [15] della spedizione del messaggio. Questa può essere di tre tipi:

- **At-most-once**: questo significa che il messaggio scambiato verrà ricevuto al più una volta e quindi potrebbe non arrivare mai.
- **At-least-once**: in questo caso abbiamo la garanzia che il messaggio verrà ricevuto ma potrebbe anche creare duplicati all'interno del sistema.
- **Exactly-once**: il messaggio deve essere ricevuto una volta sola e non deve essere perso.

Sicuramente chiunque sceglierebbe la semantica *exactly-once* però non è sempre possibile raggiungerla senza l'utilizzo di tecniche di tuning avanzate. Questa semantica inoltre introduce un livello di complessità che limita il throughput e la potenzialità di elaborazione di messaggi a fronte dell'affidabilità.

Analisi ed elaborazione Per filtrare, eseguire trasformazioni e standardizzare i dati provenienti da diverse fonti si utilizza un livello intermedio che esegue operazioni sui dati. Questo livello consente di estrarre informazione dal dato per trarne l'effettivo valore, implementando operazioni che modificano il dataset di partenza. Un serie di modifiche, calcoli e aggregazioni vengono effettuate con lo scopo di produrre un'analisi più dettagliata del dominio su cui il sistema Big Data opera.

L'elaborazione dati prende in ingresso il flusso dati sia dalla memoria a lungo termine dove sono salvati tutti i dati sia dal continuo flusso di dati organizzato dalle code di messaggi. Il calcolo del risultato finale viene prodotto in due modalità che hanno prestazioni e limitazioni diverse:

- *Approccio generale*: la tecnica generale di elaborazione di un dataset implica il calcolo completo su tutti i dati oppure in una finestra temporale limitata. Un tipo di approccio del genere fornisce un risultato preciso e deterministico. Tutti i dati vengono presi in esame e in caso di errori, bug o problemi è sufficiente rielaborare i dati disponibili in memoria ottenendo un nuovo risultato corretto. Lo svantaggio però è che il costo computazionale di calcolare il risultato basandosi su tanti dati è molto alto. Non è quindi possibile ottenere l'output rapidamente in tempo reale. Spesso nei sistemi Big Data è un requisito fondamentale poter reagire ad eventi o a situazioni particolari in base all'andamento in tempo reale.
- *Computazione incrementale*: questa tecnica di elaborazione è significativamente più performante rispetto agli algoritmi tradizionali perché non effettua calcoli su tutto il dataset. Nella computazione incrementale viene ottimizzato il calcolo ricalcolando soltanto l'output che dipende dai dati che sono effettivamente cambiati. Mentre il flusso dati arriva anche l'output viene aggiornato di conseguenza. Questi algoritmi però sono molto complicati da implementare e da sviluppare in modo che il calcolo funzioni anche distribuito su più macchine. La difficoltà aumenta anche per garantire che in caso di perdita dei dati sia possibile recuperare l'output. L'output fornito quindi è meno affidabile dell'approccio generale ma consente di fornire l'output immediatamente mentre arrivano i dati.

Persistenza dei dati Questo livello prende in carico i dati dopo che sono stati elaborati. L'obiettivo finale è immagazzinare i dati in diverse forme e con diverse tecnologie per poterli utilizzare in un secondo momento. Se ad esempio in un sistema non salvassimo i dati in nessuna maniera probabilmente sarebbe comunque possibile fornire un risultato finale dopo l'analisi. In questo modo però non esisterebbe una copia fisica quindi i dati non possono essere verificati e in nessuna maniera recuperati in caso di errori. Nei sistemi Big Data è necessario conservare i dati sia per poterli riutilizzare in futuro sia per fornire un accesso veloce all'esterno. Per questo motivo la persistenza dei dati viene realizzata in due modi:

- *Memoria a lungo termine*: questo tipo di memoria è utilizzato per scriverci un quantitativo molto alto di dati su cui non vengono effettuati accessi frequenti. I dati sono presenti in memoria per utilizzi futuri e le

elaborazioni che vengono effettuate sono molto lunghe e costose. I risultati prodotti dall'analisi di storage di questo tipo sono molto accurati perché operano su tutto il dataset o comunque su una finestra di tempo molto ampia. Questo tipo di memorizzazione viene gestito tramite un file system distribuito che effettua operazioni di calcolo molto potenti.
- *In memoria*: per un sistema che opera con un flusso di dati fornirli in tempo reale e con prestazioni alte è un requisito importante. Avere un supporto che sia in grado di rispondere in maniera tempestiva permette di fornire accesso immediato alle risorse elaborate. Il supporto in memoria permette di evitare l'intervento di chiamate IO al sistema operativo e migliora nettamente le performance. Il vincolo di questo tipo di memorizzazione è che limita la capacità di immagazzinamento ed è volatile. Lo spazio in memoria è limitato ma molto performante quindi tipicamente si utilizza questa metodologia come interfaccia di appoggio per l'accesso ai dati su cui bisogna effettuare tante letture.

Accesso ai dati Questo livello ha lo scopo di fornire all'utente finale la possibilità di accedere al flusso di dati elaborato e salvato in memoria. I dati del livello di analisi sono dati sempre aggiornati in finestre di tempo brevi mentre in memoria sono presenti tutti i dati. É utile mantenere il dato in entrambi i modi per avere sia analisi più accurate sia effettuate in periodi di tempo più brevi. Oltre al flusso di dati però deve essere disponibile anche una rappresentazione visiva del dato finale. Questa rappresentazione fornisce all'utente la percezione e la possibilità di vedere il dato in una forma comprensibile e significativa. In sostanza questo livello deve fornire l'accesso al sistema Big Data in tempo reale a quello che è il risultato dei livelli precedenti.

In breve, le principali modalità di accesso ai dati [14] per un client attraverso tecniche basate su protocolli web sono:

- *Webhooks*: questa modalità permette al client di registrare una o più *callback* alla API di streaming. Dopo aver ricevuto da parte del client la funzione da chiamare la API di streaming attende cambiamenti dal livello di archiviazione dati. Quando avviene una modifica nei dati allora vengono invocate la callback registrate inizialmente.
- *HTTP Long Polling*: in questo tipo di comunicazione viene instaurata una connessione HTTP che viene mantenuta aperta nei confronti della API di streaming. Mantenere una connessione aperta per ogni client è costoso però semplifica molto l'invio dei dati quando i dati vengono aggiornati.
- *Server-sent events*: questo è uno standard che consente al server di inviare aggiornamenti al client in modo unidirezionale. La comunicazione

avviene solo dal server verso il client e permette di mandare eventi quando i dati cambiano. Una volta che il client si è connesso è il server che si occupa di inviare eventi a cui il client decide in che modalità reagire.
- *WebSockets*: è un tipo di comunicazione TCP *full-duplex* cioé che consente la comunicazione bidirezionale tra server e client che deve accedere ai dati. La differenza tra questa modalità e quella precedente è che è possibile inviare messaggi dal client al server in modo da modificare la richiesta dei dati. Inizialmente si instaura una connessione, il server comincia a inviare messaggi al cliente e il client in modalità arbitraria può mandare messaggi al server per modificare la richiesta, cambiare la frequenza di invio di messaggi o metterla in pausa.

L'ultima parte del ciclo di vita di un sistema Big Data che implica l'accesso ai dati è la visualizzazione dei dati prodotti. La visualizzazione [16] è la presentazione in formato grafico delle informazioni prodotte dall'elaborazione dei dati. L'impatto visivo semplifica quello che si vuole comunicare e lo rende accessibile a tutti. Rappresentare graficamente significa fare un riassunto attraverso torte, istogrammi o grafici di quello che è l'andamento del sistema in modo da comunicare qualcosa all'utilizzatore in pochi secondi. Le utilità principali della visualizzazione sono:

- *Individuare trend*: con la creazione e l'analisi di mappe, è possibile fare predizioni e capire l'andamento di un flusso informativo.
- *Individuare correlazioni*: situazioni anomale o situazioni ricorrenti possono essere individuate più semplicemente mettendo a fattor comune i dati in diverse rappresentazioni grafiche.
- *Visualizzare tanti dati*: visto che una delle caratteristiche principali dei Big Data è il volume per poter analizzare correttamente le informazioni fornite bisogna poter inquadrare in maniera generale la situazione.
- *Presentare i dati*: la comunicazione della situazione del sistema è più semplice e diretta con dati elaborati anche visivamente.

Grandissime quantità di dati possono essere interpretate grazie ad una corretta rappresentazione grafica. Non è però semplice costruire un livello di questo tipo. Per ottenere il massimo dei risultati è necessario avere un'alta qualità dei dati, avere un'analisi accurata da parte di figure professionali qualificate in grado di ottenere il meglio dai dati e infine è richiesto un'alta potenza di calcolo in grado di rappresentare questi modelli dati molto complessi.

1.3.3 Apache Zookeeper

Le applicazioni Big Data utilizzano spesso framework che operano ad alte performance e in maniera distribuita. L'utilizzo dei sistemi distribuiti non è

soltanto un vantaggio visto che introducono un livello di complessità notevole di comunicazione. La scambio di messaggi attraverso la rete su macchine separate è intrinsecamente poco affidabile e spesso è necessario implementare strategie per evitare *race conditions* e *deadlock*.

Per fare in modo di evitare all'applicazione di dover implementare da zero un servizio di coordinazione in Yahoo! nasce *Zookeeper*. Attualmente è un progetto open-source che fa parte della *Apache Software Foundation* [17] che permette ai processi distribuiti di coordinarsi e comunicare fornendo un insieme di funzionalità su cui costruire servizi di sincronizzazione distribuita, naming univoco e la condivisione di risorse come configurazione e lock distribuiti.

Le caratteristiche principali di Zookeeper sono:

- *Affidabilità*: anche se un nodo non è più disponibile il servizio continua a funzionare;
- *Semplicità*: l'architettura e le primitive di Zookeeper sono facili da capire e implementare visto che ricordano un file system;
- *Performance*: le letture sono molto veloci nonostante garantiscano consistenza;
- *Scalabilità*: l'aggiunta di macchine all'applicazione aumenta le prestazioni in maniera lineare.

Può essere visto come un sistema di *atomic broadcast* dove tutti gli aggiornamenti del modello dati sono totalmente ordinati grazie al suo protocollo proprietario chiamato *Zookeeper Atomic Broadcast* (ZAB).

Zookeeper è logicamente organizzato come un registro chiave valore distribuito simile ad un file system. A differenza di un file system però la struttura dati viene mantenuta in memoria per garantire alte performance e basse latenze sulle risposte.

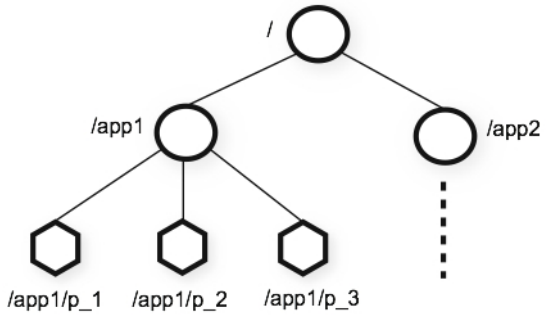


Figura 1.6: Modello dati di Zookeeper [17].

Come si vede in figura 1.6 il registro è rappresentato da una struttura ad albero dove ogni nodo, detto *ZNode* può avere un valore assegnato e dei nodi

figli. Ogni nodo è identificato da un nome che corrisponde ad una sequenza di altri elementi separati da *slash* (/).

Questo framework viene spesso utilizzato all'interno di tecnologie open-source per operazioni complesse come:

- *Leader election*: nelle applicazioni *master/slave* è spesso necessario una strategia di elezione del nodo master sia all'avvio sia quando non è più disponibile;
- *Gestione di configurazione*: viene mantenuta una configurazione univoca e consistente su tutte le macchine dell'applicazione;
- *Sistemi di notifica*: grazie alle sue primitive che consentono di registrarsi ad eventi dei nodi dove è possibile reagire a eventi di aggiornamento, creazione ed eliminazione.

Capitolo 2

Contributo

In questo capitolo vengono analizzati più nel dettaglio tutti i concetti che sono alla base del percorso di un dato. Per ognuno di essi viene messa in evidenza la motivazione che spinge chi progetta il sistema alla scelta di una determinata tecnologia, viene analizzato un progetto che risolve quella esigenza e infine vengono evidenziati gli aspetti critici. Gli aspetti critici vengono supportati da eventuali soluzioni oppure viene presentata un'alternativa che affronta il problema diversamente.

2.1 Message Oriented Middleware

In questa sezione viene analizzato il problema del collezionamento e dello scambio di messaggi tra sistemi software in un'architettura distribuita. Viene spiegato il concetto di Message Oriented Middleware (2.1.2) inizialmente introducendo un background teorico che illustra i concetti principali della messaggistica poi in seguito viene presa in analisi una tecnologia open source che risolve il problema: *Apache Kafka* (2.1.4). Ne vengono analizzati i concetti principali, i vantaggi che portano alla sua scelta e infine ne vengono messe in evidenza alcune criticità principali.

2.1.1 Middleware

Un sistema software composto da moduli diversi eventualmente distribuito su più processi o più macchine ha la necessità di comunicare, scambiare messaggi e coordinarsi. Lo scambio di messaggi tra sistemi che non condividono le stesse aree di memoria, la stessa rete o addirittura sono geograficamente distanti è uno dei problemi principali affrontati nelle architetture distribuite a servizi. Ogni servizio ha vita propria, ha la necessità di comunicare il suo stato, le operazioni che sta svolgendo ed eventualmente è in ascolto in attesa di input esterni.
Da qui nasce l'esigenza di ricevere messaggi, prendere in carico grandi flussi di dati e garantire che arrivino a destinazione in maniera organizzata ed efficiente. Questi quantitativi di dati possono mettere a dura prova un sistema non predisposto e possono rallentare notevolmente il sistema causando un eccessivo overhead che porta a situazioni critiche in cui possono esserci disservizi e perdite di dati. I dati sono molto importanti e ogni byte di informazione può essere rilevante in sistemi bancari e in settori riguardanti la sicurezza informatica. Per questo motivo è molto importante fare in modo che il collezionamento e lo spostamento di dati siano il più veloce possibili in modo che l'architettura sia reattiva e modulare. Questo aspetto della progettazione consente di concentrarsi meno sulla coordinazione dei vari moduli, sui collegamenti statici tra servizi e più sul ruolo principale che deve svolgere il sistema.

Analizzando un esempio, supponiamo che un'applicazione molto semplice ad un certo punto per mandare informazioni di monitoraggio ad una dashboard introduca una connessione con un servizio esterno che visualizzerà le metriche in un'applicazione web. Per un'esigenza così semplice introduce un interfacciamento diretto con i vari server frontend.

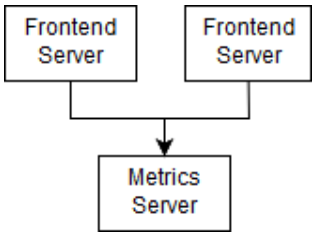


Figura 2.1: Una connessione diretta ad un servizio di metriche [18].

Al crescere dell'applicazione però, cresce anche la necessità di analizzare le metriche, salvarle su database e monitorarle. Qui nasce uno dei problemi organizzativi centrali: le connessioni punto a punto sono fragili, poco reattive a cambiamenti e difficili da mantenere nel tempo. All'aumentare delle dimensioni di un progetto aumenta anche il numero messaggi da scambiare e fornire in tempo reale. Quando viene introdotto un nuovo servizio infatti, se i servizi sono interconnessi in modalità punto a punto si rende necessario ridefinire tutte le modalità di comunicazione.

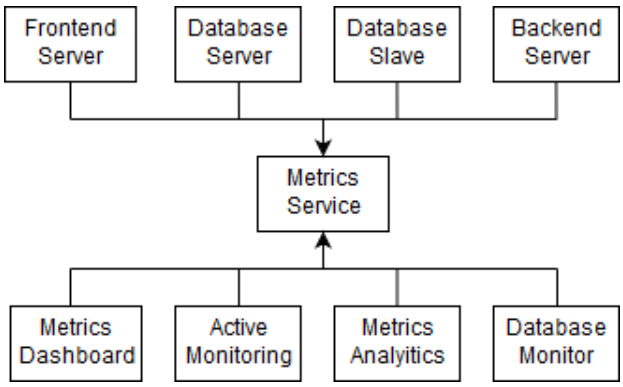


Figura 2.2: Un servizio di metriche che opera come Middleware [18].

Una soluzione è quella di implementare un'applicazione che resta in ascolto e quindi riceve le metriche da tutti i nuovi servizi ma allo stesso tempo fornisce, quando necessario, la possibilità di richiedere metriche.

Questo esempio introduttivo evidenzia alcune motivazioni che spingono a introdurre un livello intermedio che si occupi di gestire in maniera efficiente lo smistamento delle comunicazioni all'interno di un'architettura software. I sistemi che non prevedono un aumento delle dimensioni del progetto e delle moli di dati da supportare sono destinati ad incontrare difficoltà organizzative e implementative nel progetto.

Definizione di Middleware

Quando si parla di middleware le definizioni sono tra le più svariate visto che è un concetto molto ampio e complesso.
Un middleware [19] può essere pensato come un software che fornisce un insieme di servizi che rendono semplice e trasparente la comunicazione tra processi su una o più macchine attraverso la rete. I middleware inoltre forniscono un ulteriore strato di astrazione logica che rende semplice le integrazioni tra software e talvolta hardware eterogenei.

Come emerso dalla precedente definizione possiamo affermare che i casi di utilizzo di un middleware è estremamente ampio. Lo scenario tipico è quello di un sistema di mediazione e con questo ci riferiamo a sistemi con responsabilità come il monitoraggio, logging, esecuzione di funzioni di calcolo distribuite ed esecuzioni di operazioni attraverso la rete su diversi dispositivi. La potenzialità principale dei middleware in questo caso è quella di mantenere astratta la connessione tra i dispositivi e fornire semplicemente un canale logico su cui scambiare i messaggi.

Quindi considerando la definizione e il caso d'uso principale possiamo dire che tutti i middleware hanno quattro caratteristiche principali:

- **Nascondere la distribuzione**: ossia astrarre che il sistema è distribuito su più dispositivi hardware e interconnessi tra di loro;
- **Nascondere l'eterogeneità**: questo permette a dispositivi con hardware, sistema operativo e protocolli differenti di interfacciarsi tra di loro;
- **Fornire un'interfaccia di alto livello**: facilita a chi sviluppa software di integrare questo tipo di comunicazione;
- **Fornire servizi comuni**: fornire un insieme di funzioni generiche per per evitare duplicazione e facilitare la collaborazione tra le applicazioni.

2.1.2 Classificazione di un middleware

In questa sottosezione vengono classificate brevemente le tre categorie di middleware in modo da metterne in risalto le particolarità e gli svantaggi.

- **Middleware procedurale**: tipicamente implementa le remote procedure call dove un client chiama delle funzioni su un server e si aspetta un valore di ritorno (Sezione 2.1.2);
- **Object-oriented middleware**: permette di usufruire dei concetti dell'object-oriented programming in maniera sincrona offrendo accesso ad oggetti in memoria attraverso la rete (Sezione 2.1.2);
- **Message oriented middleware**: si concentra sullo scambio di messaggi asincrono tra un applicazione che produce e una o più applicazioni che

consumano questi messaggi. Tipicamente c'è la necessità di un broker centralizzato che si occupa dello smistamento (Sezione 2.1.2).

Remote Procedure Calls

Le Remote Procedure Calls (RPC) [20] sono uno dei primi protocolli di comunicazione a messaggi nati negli anni '70 quando i computer hanno cominciato ad essere connessi alla rete e ad avere la possibilità di comunicare tra di loro. Lo scopo principale di questo metodo di interconnessione tra processi è quello di avere la possibilità di effettuare una chiamata a funzione o metodo di uno specifico linguaggio di programmazione su una macchina remota come se la si stesse effettuando in memoria. Il vantaggio di questa operazione è la possibilità di richiamare codice non presente sulla memoria della macchina chiamante astraendo quindi la complessità di comunicazione attraverso la rete.

Nella figura 2.3 è presentato un esempio di comunicazione RPC: l'applicazione chiamante dopo aver lanciato la richiesta si mette in attesa di una risposta in maniera sincrona. La richiesta è trasparente al programmatore mentre internamente viene fatta una serializzazione della richiesta, e viene spedita attraverso la rete in modalità client-server. Questo significa che, una volta instaurata una sessione tra due macchine, la macchina che riceve la chiamata fa da "server" alla macchina chiamante che esegue il ruolo di "client" della comunicazione.

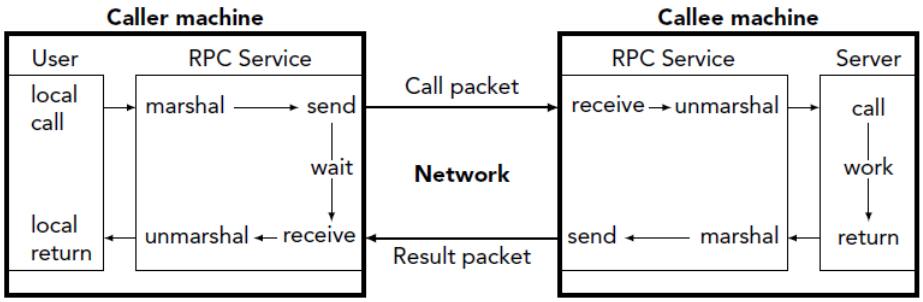


Figura 2.3: Un esempio di comunicazione RPC [20].

Nelle RPC possono essere evidenziati alcuni vantaggi che hanno portato successivamente allo sviluppo di altri protocolli di comunicazione:

- In un sistema che implementa RPC ogni applicazione deve fornire delle interfacce ben definite e note a tutte le implementazioni. Questo è uno dei maggiori limiti di scalabilità infatti al crescere del numero di applicazioni cresce anche il numero di interfacce esposte alle chiamate remote;

- La sincronia della comunicazione subisce le conseguenze della latenza delle operazioni di rete. Questo significa che eventi aleatori possono causare attese anche molto lunghe provocando così colli di bottiglia in punti di codice che subiscono un alto carico di operazioni.
- Una chiamata a funzione locale ha un esito predicibile e il ritorno dipende solo dai parametri di ingressi. Una richiesta attraverso la rete è imprevedibile e fa insorgere problemi che vanno gestiti con ulteriori tentativi. I problemi possono essere dovuti alla rete oppure possono essere dovuti ai rallentamenti sulla macchina chiamata;
- Una funzione locale che restituisce un valore può sollevare eccezioni oppure causare loop infiniti e crash. Una chiamata remota può non restituire mai una risposta rendendo impossibile sapere se è stata effettivamente eseguita oppure no. Addirittura, i tentativi successivi, possono causare la multipla esecuzione del metodo.

Nonostante i problemi evidenziati le RPC sono ampiamente utilizzate e stanno procedendo in una direzione che punta a marcare la differenza tra chiamata locale con una chiamata remota, fornisce supporto a nuovi tipi di serializzazione come Avro e JSON.

Object-Oriented Middleware

Mentre le RPC forniscono la possibilità di accedere a funzionalità e operazioni di un'altra applicazione gli object-oriented middleware consentono di astrarre la comunicazione al modello dati ad oggetti dell'object-oriented programming (OOP) [20]. In questa maniera gli oggetti locali o remoti dell'applicazione vengono visti alla stessa maniera in modo del tutto trasparente. Gli oggetti remoti sono accessibili grazie ad un proxy che fa riferimento ad un broker che si occupa dell'associazione tra riferimento e oggetto in memoria in un'altra macchina. Il broker oltre ad occuparsi della comunicazione si occupa anche della trasformazione dei dati in byte e quindi della serializzazione degli oggetti. Tali sequenze sono poi trasmesse e deserializzate dal broker che riceve i dati. I concetti principali sono mostrati in figura 2.4.

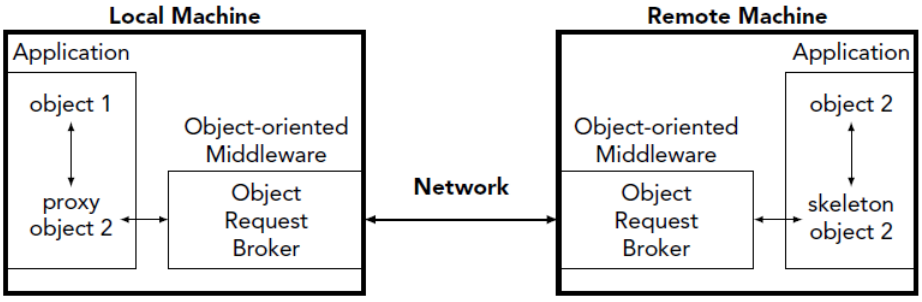


Figura 2.4: Architettura generica di un object-oriented middleware [20].

Nonostante la comodità di far riferimento ad oggetti remoti anche questo modello di messaggistica soffre di problemi simili alle RPC. La comunicazione nella maggior parte dei casi rimane sincrona e lascia spazio ai problemi delle RPC. Oltre questo si aggiunge il problema del garbage collector distribuito su più macchine che pone una sfida ulteriore alle applicazioni che utilizzano tanta memoria e non hanno risorse limitate. Inoltre, le applicazioni che utilizzano in questa maniera lo stesso modello logico e legano il modello logico con quello di comunicazione limitando la scalabilità del sistema.
La Common Object Request Broker Architecture (CORBA) è uno standard dell'Object Management Group (OMG) principale che fornisce l'object-oriented middleware tra più nodi distribuiti indipendentemente dal linguaggio di programmazione con cui sono sviluppati.

Message Oriented Middleware

Dato che la rete è intrinsecamente per definizione poco affidabile, lo scambio di messaggi non potrebbe essere sufficiente senza un sistema che svolga la funzione di intermediario e riesca a garantire affidabilità e solidità alla comunicazione. I sistemi Message Oriented Middleware (MOM)[21] forniscono uno scambio distribuito di messaggi sulla base di un modello asincrono fornendo la possibilità di gestire la latenza evitando il comportamento bloccante all'interno dell'applicazione. Questo risolve la maggior parte dei problemi evidenziati nelle RPC che richiedono di bloccare l'esecuzione del programma e attendere una risposta dal chiamato.

L'obiettivo principale di un MOM quindi è quello di fornire un canale virtuale, ossia un canale di comunicazione attraverso la rete che consenta lo scambio di informazioni incapsulate in un messaggio fisico.

Il messaggio è composto da: header, properties e body. L'header contiene informazioni riguardo la destinazione della risposta, il tipo di messaggio e la data di scadenza. L'header è solitamente necessario al broker ma è utile anche

a scopo di debug. Le properties sono informazioni speciali che servono ai consumer per filtrare facilmente i messaggi. Il body è il contenuto del messaggio che può essere un semplice messaggio di testo o byte di dati di un oggetto serializzato. La serializzazione svolge un ruolo chiave nella messaggistica visto che il modello dati è spesso legato all'implementazione.

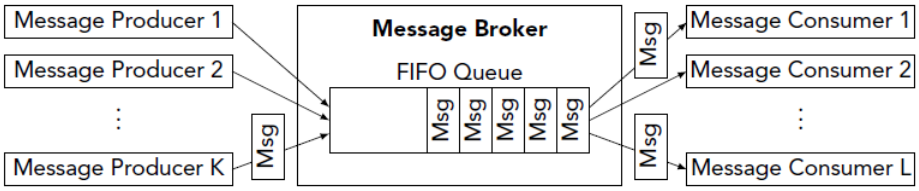


Figura 2.5: Architettura di un message oriented middleware [21].

Nella figura 2.5 vengono evidenziati i principali concetti generali dei message oriented middleware. Questi sono: message producer, message consumer e message broker. Il broker dei messaggi a volte è anche definito come agente che trasferisce i messaggi e svolge il ruolo di bus di comunicazione. Il producer dei messaggi è responsabile della generazione e dell'invio di messaggi al message broker il quale ad alto livello si occupa di gestire lo smistamento e la conservazione dei messaggi per un determinato periodo di tempo. Questo periodo detto di retention dei messaggi, è una scelta di progettazione che permette di garantire la disponibilità del messaggio per più tempo ma d'altra parte appesantisce il sistema fornendogli la responsabilità di conservare i dati per più tempo. Di conseguenza, i message producer e il message consumer sono logicamente disaccoppiati e non dipendono l'uno dall'altro visto che per fare in modo che il flusso dei messaggi vada a buon fine non è necessario che siano entrambi disponibili. Il disaccoppiamento inoltre permette risposte più veloci visto che non si rimane in attesa dell'elaborazione e si isolano le implementazioni delle due parti. L'isolamento permette al sistema di scalare e non influenzare altre parti del sistema distribuito. Se tutte le parti sono isolate il sistema scala anche in caso di malfunzionamenti e problemi di rete.

Il concetto alla base dei message oriented middleware sono le code di messaggi. Il broker fornisce almeno una coda e permette ai client di crearne altre. Queste code rappresentano un'interfaccia di comunicazione su più canali logicamente distinti nonostante un unico collegamento fisico. Alla base una coda di un message oriented middleware utilizza un ordinamento FIFO (First In First Out) che in base al timestamp di arrivo dei messaggi li fornisce al consumer in ordine crescente. I messaggi ricevuti vengono temporaneamente immagazzinati in dei buffer ed elaborati in background; in questo modo quando un consumer

richiede i messaggi, anche se non era attivo nel momento esatto del loro arrivo, può ottenerli semplicemente riprendendo dall'ultimo messaggio ricevuto.

In conclusione, riassumendo l'analisi dell'architettura di comunicazione fornita tra le particolarità che fornisce un message oriented middleware sono:

- **Disaccoppiamento**: i message broker forniscono un interfaccia che astrae la comunicazione e permette l'isolamento delle applicazioni che non devono più essere strettamente legate le une con le altre;
- **Affidabilità**: con l'idea di conservare il messaggio e poi inviarlo viene impedita la perdita di messaggi quando l'applicazione chiamata non è disponibile;
- **Scalabilità**: oltre a disaccoppiare il legame tra le applicazione la suddivisione del lavoro permette anche di avere performance migliorate rispetto ad un collegamento diretto tra macchine;
- **Disponibilità**: un sistema con questa architettura fa riferimento ad un broker che si interfaccia a chi richiede il messaggio ed è garante del servizio e della sua corretta esecuzione. Oltre a questo quando un consumatore non è disponibile non si interrompe tutto il processo ma può intervenire un processo simile che svolge il medesimo ruolo.

Confronto tra MOM e RPC

In base allo scenario in cui ci si trova, MOM e RPC hanno i loro rispettivi vantaggi e svantaggi. RPC fornisce un approccio di sviluppo semplice e diretto che fornisce un modello sincrono. Ma nonostante questo è un modello poco flessibile, penalizzante in caso di crescita delle interfacce funzionali e poco incline a cambiamenti di sotto parti del sistema. RPC assume che un sistema cambi poco spesso nel tempo e sia sempre completamente disponibile: se il sistema ha dei problemi di connessione, vengono effettuati dei riavvii o una macchina smette di funzionare, il risultato potrebbe essere in uno stallo del programma in attesa della risposta sincrona. Una chiamata RPC costa molto più overhead di una interazione MOM e necessita di più banda di trasferimento e la banda è un fattore incisivo in alcuni scenari. Un vantaggio di RPC rispetto a MOM è la garanzia dell'elaborazione sequenziale delle informazioni infatti con il suo modello dati sincrono è più semplice controllare il flusso delle operazioni. Con il modello asincrono mancano queste garanzie infatti le code di messaggi non garantiscono che il messaggio venga consumato in un lasso di tempo breve e quindi lo stato attuale non è del tutto consistente.

In conclusione RPC risulta essere lento, poco flessibile ma diretto e consistente mentre MOM semplifica il processo di costruzione di sistemi dinamici, flessibili ed è ideale nel caso in cui si è sicuri che il sistema debba scalare nel tempo come dimensioni, implementazioni e architettura.

2.1.3 Modelli di messaggi

In questa sezione vengono presentati i due principali modelli di messaggistica che consentono di organizzare in maniera efficace le due parti: chi produce il messaggio e chi lo riceve.

Point-to-point

Nel modello chiamato point-to-point diversi *producer* e *consumer* si connettono a una o più *queue* per potersi scambiare direttamente messaggi asincronamente. Questa è una coda FIFO in cui i messaggi sono ordinati rispetto a quando vengono ricevuti e c'è solitamente un solo consumer che riceve un messaggio quindi il messaggio viene prodotto da un singolo producer e ricevuto da un singolo consumer rendendo la comunicazione diretta da un punto all'altro.

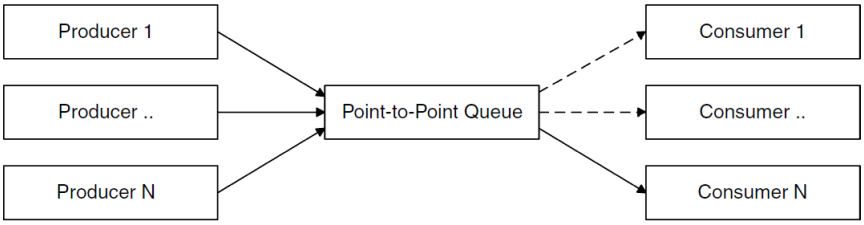


Figura 2.6: Modello point-to-point. Solo un consumer riceve il messaggio [18].

Questa tecnica può essere usata per casi in cui è necessario il bilanciamento di carico di lavoro in un sistema che ha tanto carico e più più macchine disponibili per svolgerlo.

Publish-Subscribe

Il Publish/Subscribe è un pattern di messaggistica che offre una comunicazione uno a molti e molti a molti. Solitamente è presente una entità che si occupa della produzione dei messaggi ed è chiamato appunto *publisher*. Il publisher non ha una intenzione diretta di inviare informazioni ad uno specifico destinatario ma piuttosto fornisce la possibilità di consumare il messaggio da lui fornito da un'entità ricevente chiamata *subscriber*. Il subscriber può iscriversi ad uno o più topic a cui è interessato che rappresenta un canale su cui fluiscono messaggi logicamente organizzati. La comunicazione di questo pattern ha una semantica broadcast ossia concede la possibilità a chiunque di iscriversi ad un topic e quindi di ricevere i messaggi relativi.

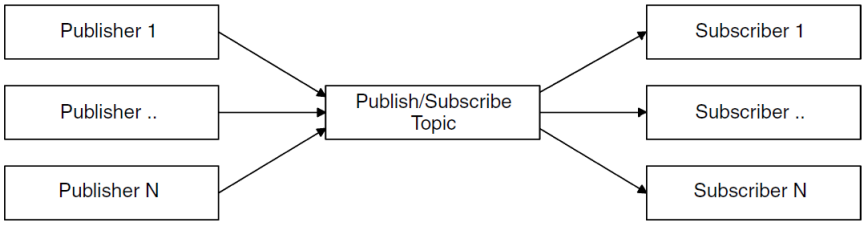


Figura 2.7: Modello publish/subscribe [18].

2.1.4 Apache Kafka

Nelle sezioni precedenti, dopo aver parlato di messaggistica e della sua importanza, abbiamo affrontato le varie strategie implementate per affrontare al meglio questa tematica. In questa sezione viene analizzato uno tra i principali framework open-source per la gestione di streaming di dati distribuiti che permettono l'interscambio di messaggi tra sistemi logicamente e fisicamente distribuiti. La tecnologia presa in esame è **Apache Kafka**.
Kafka è leader del mercato da parecchi anni, vanta di migliaia di contributors su Github anche se ormai e ora si trova a dover competere con tecnologie emergenti che sfruttano nuove implementazioni innovative che sono prese in esame alla fine della sezione.

Storia

Kafka [18] è stato sviluppato da LinkedIn per risolvere il problema dell'altissima mole di dati prodotta delle varie fonti di informazioni all'interno del social network che in quel periodo era in esponenziale espansione. Questo flusso informativo incontrollabile era formato da log di messaggi e metriche di richieste HTTP. L'obiettivo era quello di sviluppare un software che gestisse tanti messaggi e ne consentisse l'accesso ripetuto nel tempo e in maniera fault-tolerant. I dati provenienti dall'interno erano frutto dell'interazione che gli utenti avevano con i sistemi in produzione e quindi fornivano informazioni preziose agli ingegneri. In seguito, il framework ha guadagnato l'attenzione della community e nel 2011 l'Apache Software Foundation ha accettato la proposta di farlo diventare un progetto open-source tutt'ora disponibile su Github.
Riassumendo, gli obiettivi che hanno dato vita e tutt'ora alimentano il progetto sono:

- Creare un modello push-pull che disaccoppia l'utente dai sistemi in produzione. Viene definito push-pull un sistema che implementa due primitive: una che si occupa di fornire messaggi detta push e una che si occupa di richiederli in un secondo momento datta pull;
- Scrivere questi messaggi in un posto fisicamente sicuro che consenta a più sistemi di ottenere i dati,
- Ottimizzare il numero di messaggi gestiti in frangenti di tempo molto ristretti;
- Tollerare le variazioni improvvise di volume del flusso di dati in ingresso.

Definizione e casi d'uso

Kafka [22] è una piattaforma di streaming di messaggi distribuita che segue il modello publish-subscribe. É un sistema che riceve dati da multiple sorgenti e

rende disponibili i messaggi in real-time in maniera fault-tolerant ed efficiente. Kafka opera in real-time ossia consente di gestire i messaggi come un flusso di dati disponibili subito dopo il loro arrivo. Allo stesso tempo con fault-tolerant si intende che il design architetturale distribuito su più servizi ed eventualmente su più macchine garantisce che il sistema continui ad operare anche in caso di malfunzionamenti di un sottoinsieme dei suoi componenti. Kafka, come gli altri message broker, rende possibile lo scambio asincrono di messaggi ma allo stesso tempo conserva i messaggi su più processi detti broker che si occupano di mantenere il servizio attivo su più macchine. Le macchine condividono il loro stato attraverso Apache Zookeeper dove quindi si coordinano e scambiano informazioni tra di loro.

I casi d'uso [23] sono:

- **Tracking Attività**: questa è l'idea iniziale per cui era stato pensato Kafka in LinkedIn e consiste nel fornire i dati comportamentali di un utente finale sul Frontend dell'applicazione. Con attività si intende sia quella "attiva" quindi i click, le decisioni e la navigazione sia quella"passiva" cioè quella che indirettamente l'utente compie muovendo il mouse, restando su una certa pagina per tanto tempo o generando errori non visibili;
- **Metriche e Logging**: questo permette ad un sistema con diverse applicazioni di far riferimento a Kafka per la produzione di informazioni e metriche relative all'andamento delle attività in produzione. Chi consuma questi dati li può utilizzare per degli alert in caso di errore o monitoraggio di situazioni anomale in cui il sistema presenta rallentamenti o malfunzionamenti. Possono anche essere utilizzate per analisi a lungo termine per consentire di individuare pattern ricorrenti e prevedere incidenti e situazioni di stallo;
- **Commit Log**: vista la velocità di produzione di record e che Kafka viene spesso identificato come un sistema di "commit log" in cui i record vengono scritti in maniera *append only*, gli si possono scrivere ad esempio tutti i cambiamenti di un database e i consumatori possono registrarsi a questo flusso in modo da reagire immediatamente mentre le cose succedono. Questi cambiamenti possono essere utilizzati anche per replicare una situazione in un'altra installazione. La retention dei dati aumenta quindi aumenta il buffer di modifiche salvate;
- **Stream Processing**: come vedremo anche successivamente, Kafka può operare anche come strumento che fornisce i dati in real-time appena vengono prodotti e quindi al contrario dell'idea del Map Reduce che agisce su un insieme di dati molto grande e distribuito in un lasso temporale ampio questa implementazione fornisce i dati a mano a mano che sono disponibili e consente di reagire immediatamente.

Architettura e Funzionalità Interne

In questa sezione vengono analizzate le funzionalità base del framework e i componenti che permettono al framework di funzionare in maniera organizzata e affidabile attraverso tecniche di separazione dei concetti logici e suddivisioni replicate fisiche.

Message

L'unità base e fondamentale di Kafka è il messaggio (o record) che svolge il ruolo centrale in tutto il percorso che fa il dato nel framework. Un messaggio può essere visto come un record o una riga su database. Quest'ultimo è essenzialmente gestito come un array di byte che per Kafka non ha alcun significato logico se non essere il protagonista di tutto ciò che riguarda la sua trasmissione, gestione ed eventualmente la sua persistenza su disco.

Una parte molto importante è la chiave del messaggio, la quale è usata per lo smistamento del messaggio su una specifica partizione. Alla chiave infatti viene applicato un algoritmo di hashing che consente di selezionare una e una sola partizione di destinazione riguardante quel topic, questa operazione è detta partizionamento.

Per motivi di efficienza, i messaggi non vengono gestiti individualmente ma vengono raggruppati in batch ossia gruppi di messaggi che vengono mantenuti in buffer di memoria veloci. Aumentare la dimensione di queste batch è possibile, migliora le performance di processing di messaggi ma richiede molte più risorse computazionali e comporta un tradeoff tra velocità di elaborazione e latenza di propagazione, aspettare che si accumulino diversi messaggi prima che vengano effettivamente trasmessi può dare vita a situazioni di starvation e inefficienze.

Topic e Partition

I messaggi sono categorizzati in topic. Un topic può essere visto come una separazione logica tra tipi di messaggi ma allo stesso tempo, come un canale fisico su cui far fluire i messaggi di una certa tipologia. Questo vincolo è puramente organizzativo visto che, come detto prima, Kafka tratta i messaggi come semplici array di byte e non conosce il modello organizzativo sottostante. Un topic viene utilizzato per la pubblicazione di messaggi, e permette di usufruire di un'architettura di tipo publish-subscribe verso tutti i possibili consumatori che decidono di iscriversi al topic in questione.

Kafka mantiene una lista di messaggi partizionata, dove ogni partizione è rappresentabile come una sequenza ordinata immutabile di messaggi che vengono continuamente aggiunti in coda formando un "commit log" dove i primi

ad essere aggiunti sono i primi ad essere emessi quando richiesti. Se un topic viene suddiviso in più partizioni non c'è garanzia da parte del sistema che i messaggi siano ordinati temporalmente all'interno dell'intero topic ma solamente all'interno della partizione. Peculiarità del sistema è l'assegnazione automatica di un identificativo univoco ai messaggi in ogni partizione che permetta l'identificazione di ognuno di essi all'interno della partizione, i cosiddetti "offset". Questo è un concetto molto importante in quanto l'utilizzo degli offset permette di mantenere traccia del punto di lettura da parte dei consumatori, con la possibilità inoltre di ripartire in maniera arbitraria quando vengono richiesti i messaggi.

Altro aspetto interessante di questo approccio al consumo dei messaggi è la gestione dei dati pubblicati sui topic: tutti i record vengono mantenuti mediante persistenza su disco per un tempo predefinito configurabile; siano essi stati consumati o no, scaduto il tempo in questione verranno cancellati per liberare spazio. Ogni consumatore in Kafka è responsabile del controllo del proprio offset, questo permette di mantenere sempre un basso consumo di risorse da parte del cluster Kafka, in quanto l'unico dato che deve effettivamente mantenere in memoria per ogni consumatore è di fatto la posizione del consumatore stesso nel log. In questo modo anche con una retention policy dei dati molto ampia e con un grande accumulo di messaggi nel cluster, le performance rimangono invariate.

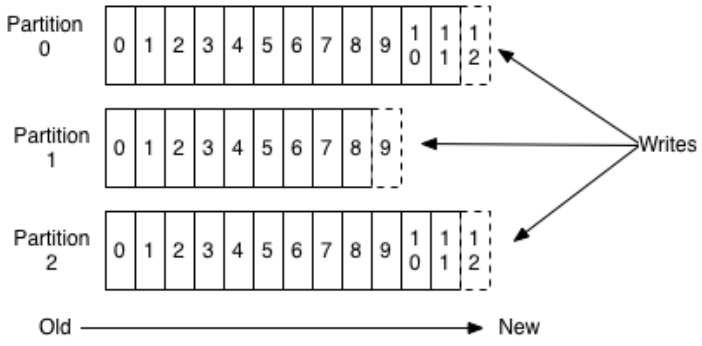


Figura 2.8: Andamento delle scritture su un topic [18].

Broker

Un server che esegue il servizio di Kafka è chiamato broker. Un broker riceve i messaggi, gli assegna un offset e quando necessario effettua le operazioni di input e output che scrivono i messaggi su disco. Il broker risponde anche alle

richieste di fetch di messaggi delle singole partizioni restituendo i messaggi che sono stati scritti su disco.

Un broker è pensato come parte di un cluster di più membri che collaborano al fine di gestire al meglio il carico, di distribuire dei messaggi in maniera replicata e di mantenere vivo il servizio anche in caso di malfunzionamenti. Tra i broker che fanno parte del cluster uno viene detto controller e svolge operazioni amministrative come: assegnare le nuove partizioni tra i vari broker, monitorare lo stato di tutti i broker e la loro connettività. Una partizione appartiene soltanto ad un broker e questo broker è chiamato *leader* di quella partizione. Successivamente se specificato la partizione viene assegnata anche ad altri broker in modo che sia replicata in più punti del cluster (possibilmente anche più macchine) in modo da offrire garanzie di ridondanza dei dati. Tutto questo avviene in modo trasparente al consumatore che deve comunque connettersi al leader della partizione.

Un'altra opzione chiave gestita dal broker è la retention dei messaggi. La retention indica il periodo di tempo per il quale i messaggi rimangono salvati all'interno delle partizioni. I messaggi possono essere conservati in base al tempo, ad esempio per 10 giorni, oppure in base alla dimensione ad esempio fino ad un massimo di 10Gb. Quando i messaggi scadono, il broker si occupa della loro eliminazione e i consumatori che si connettono al servizio e non hanno consumato quel messaggio in precedenza non lo riceveranno.

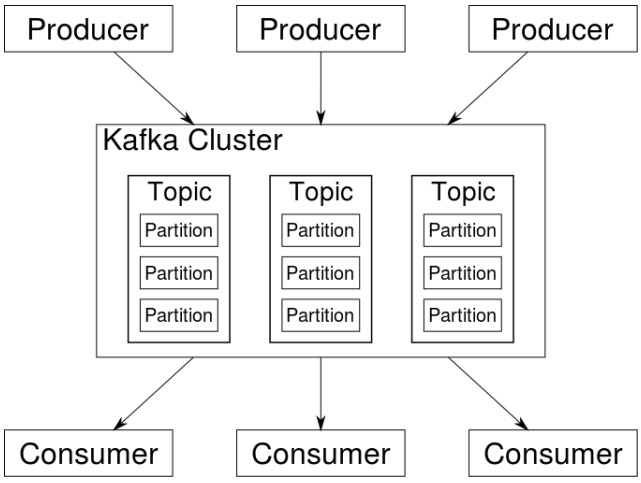


Figura 2.9: Architettura organizzativa di un cluster a più borker.

Per l'implementazione del cluster distribuito di broker Kafka utilizza **Apache Zookeeper**. Per questo motivo nella configurazione del broker ci deve essere il parametro `zookeeper.connect` per specificare dove effettuare la connessione e dove saranno registrati tutti gli altri servizi del cluster. Ogni broker

ha un identificativo univoco che viene impostato nella configurazione del broker attraverso `broker.id`. Ogni volta che un processo parte notifica a Zookeeper la sua presenza nel cluster. Gli altri servizi presenti nel cluster fanno riferimento a Zookeeper per essere a conoscenza dello stato distribuito e allo stesso tempo mantengono attiva la loro connessione. Uno soltanto di questi servizi svolge il ruolo di leader e solitamente è il primo nodo che si collega al server Zookeeper ed è il primo ad acquisire un lock distribuito che mantiene finché è attivo nel cluster. Questo sistema consente di coordinare i processi, gestire la replicazione intelligente dei dati su più macchine e capire in caso di guasti o malfunzionamenti come intervenire e quale servizio non è più disponibile.

Producer

Quando si utilizza un framework di messaggistica come Kafka allora di conseguenza ci saranno entità da costruire che si occuperanno di generare e inviare messaggi. Queste entità vengono definite come producer e permettono di spedire dati al cluster di broker in ascolto. Nell'architettura publish-subscribe i producer svolgono il ruolo di publisher e mandano in modo broadcast i flussi di byte prodotti in un topic senza poi però preoccuparsi in quale partizione fisica saranno assegnati. Diversi producer possono contribuire alla scrittura di messaggi su un singolo topic e il sistema ha l'incarico di smistarli in maniera bilanciata tra le partizioni.

Quando viene configurato un producer Kafka devono essere presi in analisi diversi fattori che incidono sulle performance e sul comportamento del framework nei confronti del messaggio. La questione principale è l'importanza della consegna del messaggio, ossia bisogna chiarire con che garanzie deve essere ricevuto dal broker di destinazione. I dati prodotti da un sistema embedded riguardante le misurazioni di uno strumento, ad esempio, non hanno la stessa importanza di un evento riguardante una transazione bancaria. Allo stesso tempo una maggior garanzia di consegna può costare caro in termini di latenza temporale. La gestione degli ack di consegna influisce molto sulla reattività del sistema, una maggiore garanzia comporta un costo computazionale maggiore. Quindi bisogna prendere in esame l'importanza del messaggio e le performance di consegna.
Nei seguenti paragrafi vengono elencati i temi principali da analizzare quando si costruisce un producer [24] e gli argomenti su cui ragionare in base alle proprie esigenze.

Configurazione base Per costruire un producer bisogna personalizzare alcune configurazioni base che determinano il suo comportamento all'interno del sistema. Tra i parametri richiesti e necessari per l'avvio di un producer c'è

`bootstrap.servers` ossia l'elenco di hostname e porta dei broker del cluster. Questo serve all'avvio per indicare dove effettuare la connessione. Anche se non è obbligatorio è utile specificare il `client.id` per identificare univocamente il producer all'interno del cluster Kafka.

Durabilità dei messaggi Si può controllare la durabilità della scrittura di un messaggio su Kafka attraverso l'impostazione della proprietà `acks` che di default è impostata a 1. Questa impostazione consente di configurare quante repliche della partizione devono ricevere il messaggio prima di considerare positivo l'esito della scrittura. Il flag può avere tre stati:

- `acks=0`: è il livello minimo di garanzia che consente di ottenere il massimo throughput dal sistema, aumenta vertiginosamente il numero di messaggi che possono essere inviati al secondo. Questo livello significa che non si attende nessuna conferma dal broker della ricezione del messaggio ed è sufficiente che sia stato inviato con successo;
- `acks=1`: in questo modo il producer attende che la partizione leader riceva il messaggio e se non è disponibile ritorna un errore che consente di riprovare l'invio per evitare la perdita di dati. In questo caso la latenza della comunicazione incide sul throughput ed è consigliato l'invio del messaggio in maniera non bloccante con la API asincrona;
- `acks=all`: questo livello è il più costoso e implica che il messaggio sia ricevuto dal leader e passato a tutte le repliche sparse nel cluster per considerare l'operazione terminata correttamente.

Ordinamento dei messaggi Generalmente viene mantenuto un ordine interno alle partizioni riguardo l'arrivo dei messaggi nel broker. Quando però ci sono dei problemi che causano il fallimento di alcuni invii si può indicare il numero di tentativi di invio del messaggio tramite la proprietà `retries`. Combinando questa opzione con `max.in.flight.requests.per.connection` che permette di limitare il numero di invii effettuati senza ricevere risposta si può preservare l'ordine dei messaggi senza ottenere inconsistenze temporali causati da tentativi multipli. Limitando quel parametro a 1 se l'invio di un messaggio fallisce viene riprovato il suo invio evitando di proseguire con i messaggi successivi.

Batching e compressione Per aumentare il throughput Kafka tende a raggruppare i messaggi in batch prima di inviarli visto che se spediamo più messaggi in una volta non c'è l'overhead di invio per ogni messaggio e il buffer dei messaggi può riempirsi anche se il sistema si sta occupando di altre operazioni

sul thread di invio. Per gestire la dimensione massima dopo la quale i messaggi vengono inviati è sufficiente impostare `batch.size` espresso in bytes. Infine può essere abilitata la compressione dei messaggi che riguarda tutta la batch attraverso `compression.type` indicando come opzione una tipologia tra: gzip, snappy, lz4 e zstd.

Implementazione di un producer

Quando si implementa un producer Kafka attraverso le API Java fornite è importante prestare attenzione a diversi punti durante la progettazione del codice. Il ruolo del producer è sostanzialmente semplice: creare un messaggio e inviarlo ad un servizio Kafka di riferimento. Non è però semplice ottenere un risultato efficiente che considera la gestione degli errori e gestisce in maniera efficiente la comunicazione con i broker di riferimento a cui si vogliono spedire i messaggi.
Ora con alcuni brevi esempi di codice vengono messi in evidenza alcuni concetti interessanti da tenere in considerazione durante la progettazione di un producer Kafka.

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "broker1:9092,broker2:9092,broker3:9092");
props.setProperty("key. serializer ",
    "org.apache.kafka.common.serialization. StringSerializer ");
props.setProperty("value. serializer ",
    "org.apache.kafka.common.serialization. StringSerializer ");
KafkaProducer<String, String> producer = new KafkaProducer<>(props);
```

Con queste quattro linee di codice è stato creato un oggetto che al suo interno gestisce la comunicazione con il servizio di Kafka ed è in grado di spedire messaggi con chiave di tipo `String` e messaggio `String` serializzati attraverso implementazioni fornite da Kafka. Successivamente è necessario costruire un messaggio attraverso un `ProducerRecord` specificando il topic al quale è destinato e il contenuto effettivo del record che, nel nostro caso specifico, sono due stringhe.

```
String recordKey = "test-key";
String recordMessage = "Example message";
ProducerRecord<String, String> record = new ProducerRecord<>(
    "test-topic", recordKey, recordMessage
);

/* 1. Fire and Forget */
try {
    producer.send(record);
} catch (Exception e) {
    e.printStackTrace();
}
/* 2. Synchronus send */
try {
    RecordMetadata metadata = producer.send(record).get();
} catch (Exception e) {
    e.printStackTrace();
}
/* 3. Asynchronus send */
private class DebugCallback implements Callback {
    @Override
    public void onCompletion(RecordMetadata metadata, Exception e) {
        System.out.printf("sent record(key=%s value=%s) " +
                          "meta(partition=%d, offset=%d)\n",
                record.key(), record.value(),
                metadata.partition(), metadata.offset()
        );
    }
}
producer.send(record, new DebugCallback());
```

In questo esempio lo stesso record viene spedito con i tre principali metodi di invio di un record attraverso il metodo `send` della Producer API:

1. **Fire and forget**: questo è il metodo più semplice e allo stesso tempo più efficiente in termini di performance. Inviamo il messaggio al server e non è importante se viene ricevuto con successo. In ogni caso viene rispettata la politica di re-invio dei messaggi configurata ma con questa implementazione è possibile che vengano persi dei messaggi;
2. **Invio sincrono**: con questo metodo, dopo aver inviato il messaggio, si rimane in attesa della risposta. Il metodo `get` infatti è bloccante e ottiene una risposta positiva in caso di successo altrimenti viene lanciata un'eccezione;
3. **Invio asincrono**: questo metodo sfrutta appieno le capacità di Kafka di ottimizzare l'invio e la gestione dei messaggi in uscita. Il secondo parametro della `send` in questo caso è una `Callback` che viene chiamata sia in

caso di successo sia in caso di errore. Con questa implementazione è possibile gestire l'invio del messaggio in modo che in caso di fallimento venga salvato temporaneamente ed eventualmente re-inviato in un secondo momento. Questo stratagemma è utile perché non blocca il programma in attesa di una risposta, consente di reagire ad un fallimento e anche in caso di successo fornisce informazioni riguardanti la partizione e l'offset di scrittura sul broker.

Per quanto riguarda invece la gestione degli errori ci sono due tipi di errori: quelli che il producer gestisce in automatico e quelli che come programmatore vanno gestiti in base all'esigenza.
Il producer può gestire gli errori che vengono mandati dal broker perché c'è stato un problema attraverso il codice di risposta all'interno dell'eccezione. Questi errori possono essere di due categorie:

- Gli errori che consentono di eseguire nuovamente un tentativo come ad esempio un errore temporaneo di connettività o un down dovuto al leader non disponibile (`LEADER_NOT_AVAILABLE`);
- Quelli che non possono essere risolti con altri tentativi come ad esempio un errore di configurazione configurazione (`INVALID_CONFIG`) o di serializzazione del messaggio.

In generale, l'obiettivo è quello di non perdere messaggi quindi gestire gli errori consente di minimizzare questo rischio. Gli errori dovuti a leader election e problemi di connettività durano spesso pochi secondi quindi utilizzare una implementazione che riprova a inviare i messaggi che hanno dato errore migliora l'affidabilità generale del sistema.

```
private class ErrorHandler implements Callback {
    private Map<String, String> undeliverdMessages = new HashMap<>();

    public ErrorHandler(Map<String, String> undeliverdMessages) {
        this.undeliverdMessages = undeliverdMessages;
    }

    @Override
    public void onCompletion(RecordMetadata metadata, Exception e) {
        if (e == null) {
            return;
        }
        undeliverdMessages.put(metadata.key(), metadata.value());
    }

    public Map<String, String> getUndeliveredMessages() {
        return this.undeliverdMessages;
```

```
    }
}

Callback sendCallback = new ErrorHandler(undeliverdMessages);
for (int i = 0; i < 10; i++) {
    producer.send(record, sendCallback);
}
int count = sendCallback.getUndeliveredMessages().size();
System.out.println("There are " + count + " undelivered messages.");
```

Questo è un esempio di una implementazione in cui, in caso di fallimento, il messaggio viene aggiunto in una struttura temporanea in memoria che conserva il messaggio prodotto. Questo può essere utile per poterlo inviare in un secondo momento.

Partizionamento

I messaggi prodotti da Kafka hanno solitamente una chiave e un valore. É possibile effettuare l'invio di un messaggio anche solamente impostando un topic e lasciando la chiave a null. La chiave viene utilizzata per tre scopi:

- Identificare il record in modo informazione al contenuto. La chiave viene salvata su disco ed è quindi parte integrante del messaggio;
- Decidere a quale partizione del topic far appartenere un messaggio;
- Implementare politiche di compattamento dei messaggi in modo da risparmiare spazio se vengono emessi più record con la stessa chiave. Questa procedura viene chiamata "compattamento dei log".

Qui nasce l'esigenza di bilanciare nel modo più opportuno il carico di record sulle partizioni in base alla chiave del messaggio. Tutti i messaggi con la stessa chiave vanno nella stessa partizione: questo significa che se un consumatore legge da una sola partizione, allora tutti i record con quella chiave vengono letti dalla stessa entità.

Il comportamento di default di Kafka è quello di utilizzare una tecnica di partizionamento round-robin che ri-bilancia equamente i messaggi sulle partizioni e ciò è reso possibile da una funzionalità che associa ogni chiave ad una partizione basandosi sul suo hash.

Per migliorare la strategia di bilanciamento e per esempio spostare gran parte del carico in una partizione rispetto ad un'altra è necessario implementare un `Partitioner` che consente di gestire tutto il flusso del messaggio.

```
public class ExamplePartitioner implements Partitioner {
    public void configure(Map<String, ?> configs) {}

    public int partition(String topic,
```

```
                        Object key, byte[] keyBytes,
                        Object value, byte[] valueBytes, Cluster cluster) {
        List<PartitionInfo> partitions = cluster.partitionsForTopic(topic);
        int numPartitions = partitions.size();
        String skey = (String) key;
        if (skey.contains("test")) {
            return numPartitions;
        }
        return (Math.abs(Utils.murmur2(keyBytes)) % (numPartitions - 1));
    }

    public void close() {}
}
```

In questo esempio `ExamplePartitioner` ridireziona tutti i messaggi che contengono *test* nella chiave sull'ultima partizione del topic. Agli altri record viene applicato l'algoritmo di hashing *murmur2*.

Consumer

Una volta scritti i messaggi su Kafka è possibile ottenerli attraverso dei servizi che si occupano di ottenere in maniera efficiente il dato. L'entità che si occupa di questo è il *consumer* che quindi è un processo che si occupa di ricevere e leggere da Kafka i dati inviati su un certo topic. I consumer non sono un'entità singola e indipendente come i producer ma introducono un concetto di *consumer group*. Un consumer group può essere visto come un cluster di consumatori, un insieme logico che può essere sparso su più macchine ed è composto di uno o più consumer legati allo stesso topic. Ogni consumer nel gruppo riceve i messaggi da un sottoinsieme di partizioni del topic, per questo motivo è importante suddividere un topic in più partizioni replicate. I vantaggi di costruire gruppi di consumer che leggono da diverse partizioni sono:

- **Aumento di performance**: un gruppo di elementi che richiede dati parallelamente è uno dei fattori più importanti per l'aumento di prestazioni in lettura da Kafka;
- **Distribuzione del carico**: con l'aumentare dei messaggi in un topic aumenta che il numero di record da elaborare nei consumer, per questo motivo per un consumer avere soltanto un sottoinsieme di partizioni da gestire alleggerisce il lavoro sul singolo consumer;
- **Tolleranza ai guasti**: avere diversi consumer aumenta le prestazioni però allo stesso tempo consente una continuità di servizio nel caso in cui un consumer smetta di funzionare. Il sistema capisce che il consumer non è più attivo e si occupa di ri-assegnare le partizioni da cui stava leggendo

tra gli altri consumer. Questo non può avvenire se si consuma soltanto con un elemento tutti i record da una partizione.

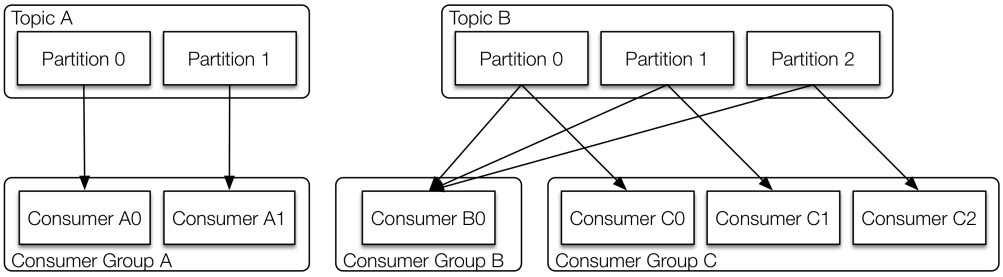


Figura 2.10: Esempio di consumer e consumer group [25].

Nella figura 2.10 si può notare come più consumer group possono essere legati allo stesso topic e come un consumer group può essere formato da un solo elemento che legge da tutte le partizioni i record. L'ordine cronologico dei messaggi è preservato solo all'interno di una singola partizione quindi durante il consumo dei messaggi bisogna tenere in considerazione che non è garantito l'ordine all'interno dell'intero topic se questo è formato da più partizioni.

Implementazione di un consumer

Quando si devono leggere dei messaggi è necessario restare in ascolto per l'arrivo di nuovi record. Dopo la configurazione delle proprietà del consumer è necessario creare un `KafkaConsumer` in modo da potersi collegare ad un topic.

```
Properties props = new Properties();
props.setProperty("bootstrap.servers", "broker1:9092,broker2:9092,broker3:9092");
props.setProperty("group.id", "test-group");
props.setProperty("key. serializer ",
    "org.apache.kafka.common.serialization. StringSerializer ");
props.setProperty("value. serializer ",
    "org.apache.kafka.common.serialization. StringSerializer ");
KafkaConsumer<String, String> consumer = new KafkaConsumer<>(props);
```

Dopo aver creato un consumer bisogna prestare attenzione a come leggere i dati. Solitamente si crea un ciclo detto *Poll loop* che rimane in attesa di nuovi messaggi cercando di capire se ne sono arrivati nuovi rispetto all'offset salvato per quello specifico consumer.

```
consumer.subscribe("test-topic");
try {
    while (true) {
        ConsumerRecords<String, String> records = consumer.poll(100);
        for (ConsumerRecord<String, String> record : records) {
            System.out.printf("recived record(key=%s value=%s) " +
                              "meta(partition=%d, offset=%d)\n",
                    record.key(), record.value(),
                    metadata.partition(), metadata.offset()
        )   ;
    }
} finally {
    consumer.close();
}
```

Nella prima viene fatto il `subscribe` al topic e si entra in un loop infinito che chiama il metodo `poll()`. Questo metodo, la prima volta che viene chiamato, permette al consumer di registrarsi al consumer group specificato e si mette in attesa di record da ottenere dal broker Kafka. Il primo parametro del metodo poll è il tempo per cui il consumer deve rimanere bloccato in attesa di nuovi record. Questo implica che minore è il tempo in attesa maggiore è la reattività del sistema ad ottenere nuovi record. Il costo di elaborazione è maggiore ma a volte è necessario che le attese siano ridotte al minimo. Attraverso il metodo inoltre il consumer fa capire al consumer group e inviando un messaggio a Kafka indicandogli che è attivo in attesa di record e non è bloccato o malfunzionante.

Infine, un fattore molto importante durante il consumo di record è il *commit* dei messaggi ricevuti dal consumer. In Kafka è possibile far sapere al cluster che un messaggio è stato letto da un certo consumer in maniera automatica oppure gestire manualmente come l'offset a cui si è arrivati a leggere viene inviato. L'offset in Kafka viene salvato in un apposito topic chiamato `__consumer_offsets` dove per ogni consumer è indicato a quale messaggio della partition è arrivato a leggere in modo da poter riprendere dallo stesso punto in caso di interruzione del servizio. Nella modalità automatica dopo un certo periodo di tempo verrà effettuato commit mentre per attivare il commit manuale bisogna impostare la proprietà `auto.commit.offset=false`. In questo modo si sta comunicando all'applicazione che il consumer deve occuparsi di questa operazione manualmente attraverso una chiamata all'API.

Serializzazione e Deserializzazione

Sia nei producer che nei consumer è fondamentale la tecnica di serializzazione dei messaggi [12] prima di essere inviati e la deserializzazione prima di essere passati al client che riceve i messaggi. Questo processo consiste nella

conversione di un oggetto in uno stream di byte. Questa operazione è necessaria perché Kafka, quando trasferisce dati a basso livello, tratta soltanto stream di byte.

In Kafka è spesso utile specificare nei producer e consumer come deve avvenire questo processo per la chiave e il valore del messaggio attraverso le proprietà di configurazione `key.serializer` e `key.deserializer`. Spesso la serializzazione necessita di implementazioni custom che implicano lo sviluppo di algoritmi che a partire da un oggetto in memoria producono un array di byte. Kafka fornisce un insieme molto vasto di queste implementazioni per i tipi di dato base come ad esempio `StringSerializer` che consente di serializzare i valori di tipo stringa. Per implementazioni più complesse su modelli di dato avanzati è consigliato utilizzare la libreria open-source per la serializzazione indipendente da ogni linguaggio: **Apache Avro**. Avro ha una specifica implementazione che abilita questa serializzazione custom ed è `KafkaAvroSerializer`.

Apache Avro è un formato di serializzazione binario che viene specificato attraverso il linguaggio JSON e definisce quali campi e di che tipo sono presenti nell'oggetto da serializzare. I motivi per cui viene spesso utilizzato in Kafka sono due:

- **Evoluzione dello schema**: il modello dati di un messaggio varia nel tempo e quindi non è sempre compatibile con il passato. Avro permette di avere una gestione versionata dei dati e quindi il messaggio inviato e il messaggio letto vengono interpretati con lo stesso schema di serializzazione;

- **Utilizzo di un registro remoto**: invece che mantenere una copia locale e statica del modello dati è possibile sviluppare uno schema remoto che viene fornito a tutto l'applicativo in maniera centralizzata. Questo approccio è rappresentato in figura 2.11.

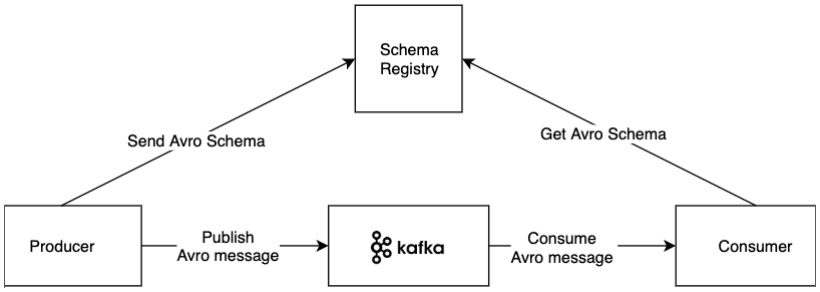


Figura 2.11: Gestione della serializzazione in un registro esterno.

Limiti di Kafka

Kafka è una tecnologia presente sul mercato da quasi un decennio ormai. Come ogni tecnologia che si sviluppa nel tempo soffre del peso delle decisioni e scelte prese nel passato. Anche se Kafka, nato nel 2011, ha una logica di funzionamento tutt'ora attuale al pari delle più moderne piattaforme di streaming di dati ma nonostante questo soffre se confrontata a tecnologie emergenti. In questa sezione viene fatta una overview di quelli che sono i suoi punti deboli e viene illustrata brevemente l'architettura di Apache Pulsar, un nuovo framework di message passing, e perché si prospetta essere la soluzione alle limitazioni di Kafka.

Kafka nasce come piattaforma di streaming con obiettivo principale quello di fornire un altissimo throughput di messaggi. La sua specialità è ottenere performance di invio di messaggi altissime e in brevi frangenti di tempo. Kafka però soffre di alcune limitazioni [26] progettuali e concettuali:

- **Mancanza di tool di monitoraggio**: questo significa che non dispone di nessun sistema che gestisca le performance e l'utilizzo delle operazioni. Solitamente in un sistema ad alte prestazioni è necessario per capire come evolvere la configurazione nel tempo facendo tuning dei parametri relativi ai tradeoff citati precedentemente;
- **Riduzione di performance**: il sistema spesso e volentieri svolge attività di routine all'interno dei broker come ad esempio l'eliminazione dei messaggi che non devono più essere mantenuti, il ribilanciamento dei messaggi all'interno dei topic e la compattazione dei record con chiave duplicata. Queste funzionalità sono utili per risparmiare memoria all'interno del broker ma allo stesso tempo fanno soffrire il sistema visto il grande numero di accessi su disco da effettuare. Kafka si comporta in maniera strana se deve effettuare queste operazioni, può subire rallentamenti e perdere messaggi visto che essenzialmente mantiene tutti i record ricevuti e replicati su disco;
- **Rallentamenti su grandi numeri di topic**: qui entrano in gioco le limitazioni del registro distribuito Zookeeper. Un grande numero di topic mette a dura prova la gestione delle notifiche e delle *leader election* che vengono eseguite su Zookeeper;
- **Difficoltà nella migrazione dei dati**: gli strumenti admin di migrazione sono poco efficienti, lo spostamento di topic e il cambiamento nel tempo del numero di partizioni è molto costoso e rischioso se si tratta con dati che non devono essere persi. Inoltre, l'aggiunta di nuovi broker comporta il ribilanciamento di tutto il cluster e la creazione di nuove partizioni che andranno riempite diminuendo così le risorse da dedicare alla computazione dei messaggi;

- **Non ha retention sui messaggi letti**: una pecca sulla garanzia di messaggi è che non c'è nessun modo di assicurarsi che un messaggio venga letto da tutti i consumer (anche eventualmente da quelli nuovi) a prescindere dal periodo di retention. Un sistema dinamico in cui spesso aumentano i consumer si trova senza la possibilità di avere un messaggio per un periodo di tempo che varia in base a quando non c'è più richiesta per quel topic. Nonostante però i punti emersi in questi punti Kafka rimane uno dei migliori framework di streaming di messaggi publish-subscribe. Rimane flessibile e con una vastissima community che contribuisce al progetto, fornisce assistenza e all'ordine del giorno mantiene in produzione sistemi che gestiscono flussi di miliardi di messaggi.

2.1.5 Apache Pulsar

Apache Pulsar [27] è un un sistema ad elevate performance distribuito che gestisce le comunicazioni server-to-server in maniera replicata geograficamente. É un sistema che fornisce un ulteriore livello di astrazione della persistenza dei dati disaccoppiando il ruolo della persistenza in memoria dei messaggi dai broker in modo da fornire maggiore affidabilità dei dati e molta flessibilità nella scalabilità del sistema.

Architettura Una istanza Pulsar è composta da uno o più cluster. I cluster tra di loro all'interno di una istanza possono replicare tra di loro le informazioni.

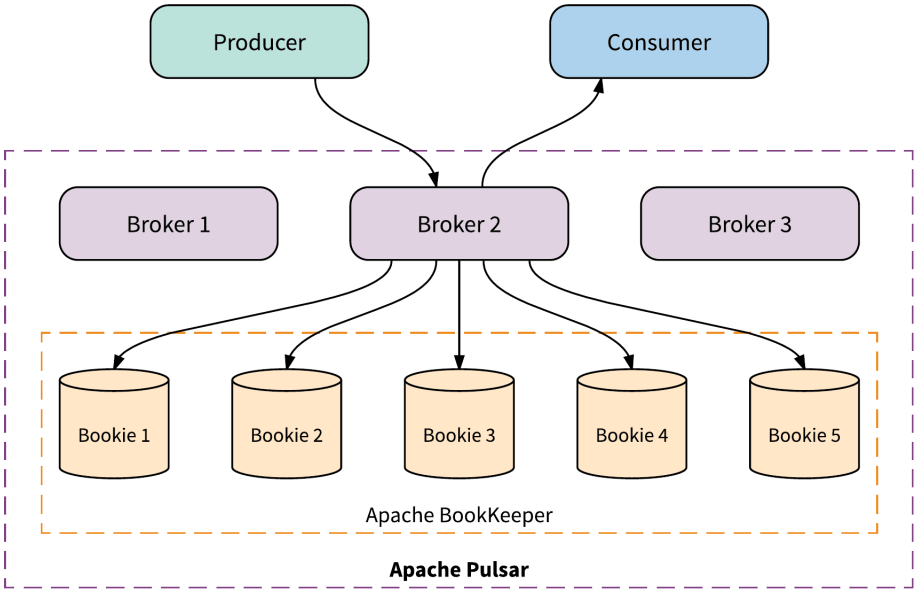


Figura 2.12: Architettura logica di Apache Pulsar [28].

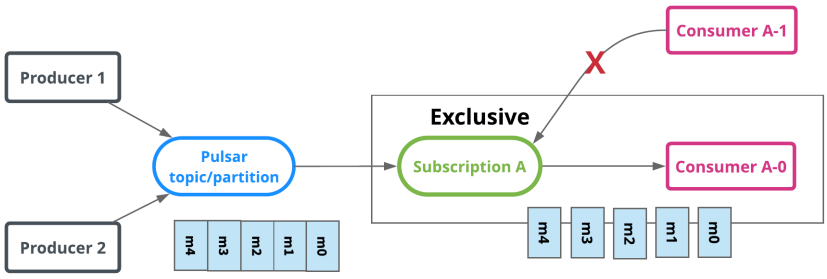
Come illustrato in figura un cluster è composto da:

- **Broker**: uno o più servizi che agiscono da broker. Un broker Pulsar è simile ad un broker Kafka per quanto riguarda il ruolo ma agisce in modalità stateless delegando un servizio esterno alla persistenza dei dati;
- **Zookeeper**: un *ensemble* (gruppo di nodi separati) che gestisce la persistenza della configurazione e gestisce il coordinamento delle entità distribuite;

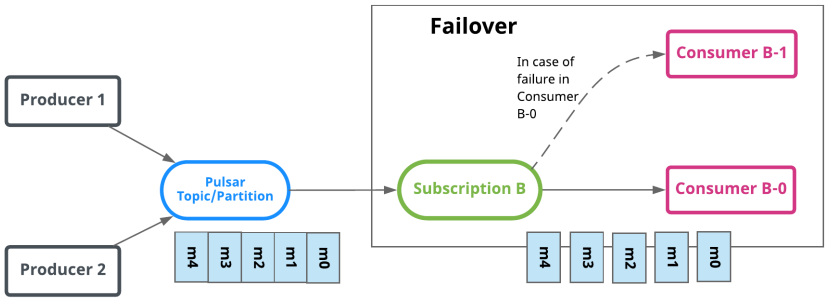
- **Bookkeeper**: un sistema di storage distribuito per la persistenza dei messaggi in maniera asincrona e coordinata. Questo sistema permette di aggiungere affidabilità ai messaggi ricevuti che vengono salvati in contenitori detti *Ledgers* e vengono replicati in maniera consistente nel cluster. Questa particolarità di Pulsar astrae ancora di più la gestione fisica del dato sui broker delegando questa responsabilità a Bookkeeper.

Modello Logico Pulsar implementa sia una gestione in streaming dei dati sia una gestione a coda di messaggi. Il suo modello logico [28] è composto da: producer, topic, subscription e consumer. Il flusso di dati, come in Kafka, parte da un producer che utilizza un topic come canale per inviare i messaggi. Ogni topic è diviso a sua volta in partition e viene fatto persistere in memoria su diversi *Bookie* (Server Bookkeeper) e rimangono a disposizione dei consumer per essere letti. Una delle più grandi particolarità di Pulsar è il concetto di *subscription* che consiste in un gruppo di consumer che instaura una iscrizione verso un topic in modo da consumarne i dati con una strategia.
Questa strategia può essere di tre tipi:

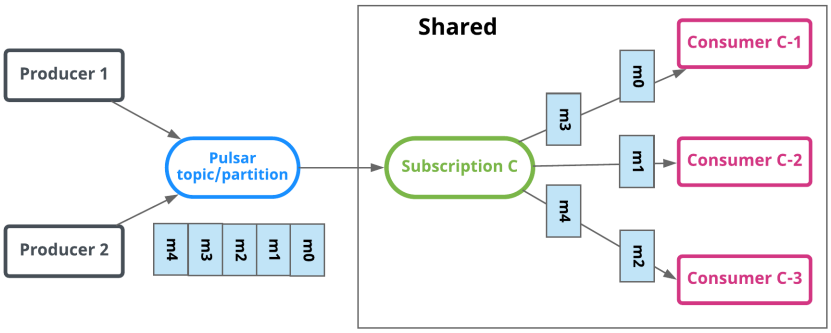
- **Subscription esclusiva**: questa modalità è in streaming e consente ad un solo consumer di registrarsi ad un topic;



- **Subscription a fallimento**: questa modalità è in streaming e consente a più consumer di registrarsi ma uno soltanto prende i record in ordine crescente. La differenza da quello esclusivo è che in caso di disconnessione, fallimento o problemi nel consumer attivo, detto *master* intervengono gli altri garantendo una continuità del servizio di elaborazione;



- **Subscription condivisa**: questa modalità è considerata come una coda di messaggi. Più consumer si possono registrare e le informazioni vengono distribuite attraverso un algoritmo round-robin tra i consumer e questa modalità permette di scalare l'elaborazione senza aumentare il numero di partizioni del topic. Questa differenza è molto importante visto che non pone limiti sulla scalabilità del sistema. In Kafka sarebbe stato necessario aumentare il numero di partizioni e quindi cambiare l'organizzazione dei dati sui broker.

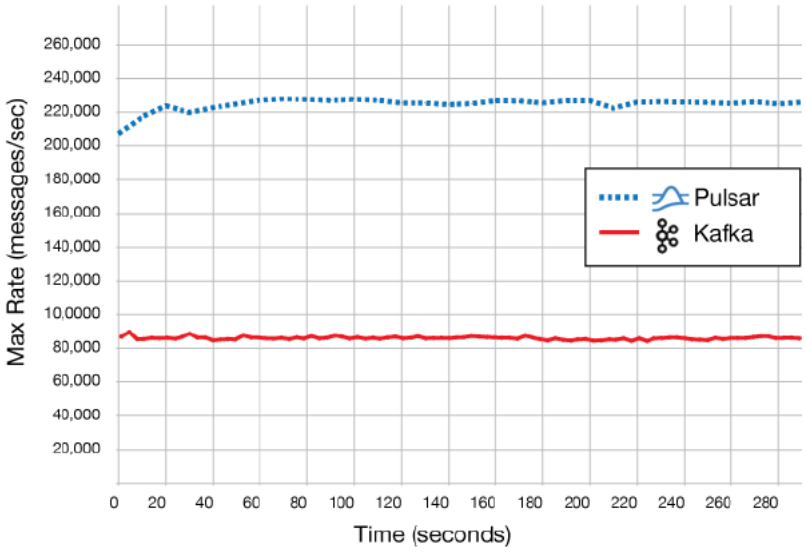


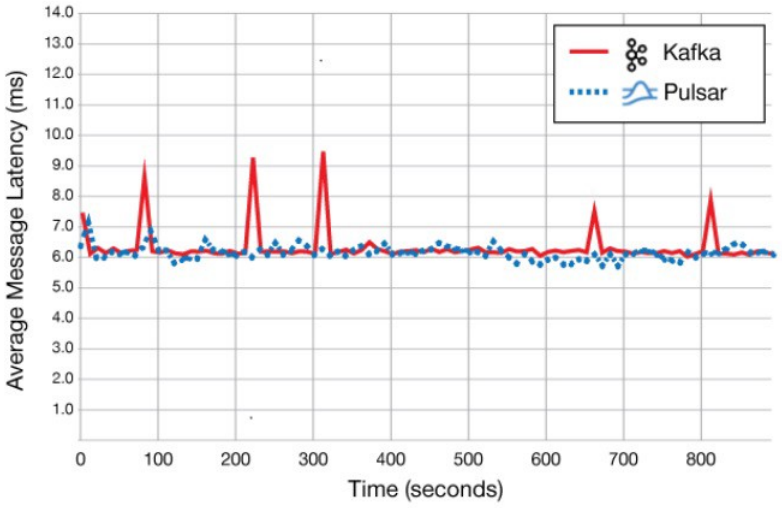
L'utilizzo delle subscription consente di consumare i dati in base alle esigenze implementative. Una subscription essenzialmente corrisponde ad un consumer group di Kafka, se si ha bisogno di preservare l'ordinamento dei messaggi si utilizza una subscription esclusiva o a fallimento altrimenti se si vogliono aumentare le prestazioni si può utilizzare quella condivisa. Quella condivisa non preserva l'ordine di arrivo ma scala nonostante il numero di partizioni nel topic.

Ack e retention dei messaggi Mentre si consumano i messaggi possono avvenire fallimenti sia da parte dei broker sia da parte dei consumer. In Pulsar, come in Kafka, vengono gestiti dei cursori all'interno delle subscription che indicano l'avanzamento di lettura dei messaggi da parte dei vari consumer.

In Kafka il punto in cui un consumer è arrivato a leggere è detto *offset* mentre in Pulsar ogni subscription ha un cursore che viene aggiornato ogni volta che un messaggio è stato ricevuto correttamente. A differenza di Kafka, Pulsar dà la possibilità di confermare la lettura in due modi: acknowledgment singolo o acknowledgment multiplo. La modalità multipla è la stessa di Kafka infatti indicando qual è stato l'ultimo messaggio letto tutti i precedenti vengono considerati ricevuti e non verranno più richiesti nuovamente. Con l'acknowledgment singolo si ha maggior controllo e, con una gestione più avanzata della ricezione dei messaggi, si ottengono risultati che reggiungono una maggiore affidabilità del servizio.
Una volta confermato l'arrivo dei messaggi da un consumer questi potrebbero venir richiesti da altri arrivati successivamente. Pulsar non è un sistema di storage quindi il suo scopo non è persistere i dati dei messaggi su disco a tempo indefinito anche se di default mantiene i dati non mai letti fino a quando almeno un consumer lo ricevuto. Mentre Kafka persiste in memoria i messaggi ricevuti Pulsar può distribuirli su più macchine senza avere un impatto diretto su scritture e stato del broker. Kafka se ha un elevato flusso di messaggi dovrà cancellare i dati molto in fretta per evitare di superare i limiti di numero e dimensione dei messaggi impostati nelle politiche di retention. Oltre la retention dei messaggi però Pulsar implementa anche un parametro di TTL (Time To Live). Di base se un messaggio non viene mai letto rimane salvato per sempre ma questo è molto costoso in termini di spazio disco occupato. Questo TTL quindi indica che, se non viene ricevuto nessun acknowledgment di un messaggio per un lasso di tempo allora dal sistema viene considerato come se fosse stato letto.

Conclusione e performance Pulsar ha una diversa gestione degli offset, ha una diversa modalità di persistenza dei dati che consente di avere dei broker senza stato e ha funzionalità avanzate di geo distribuzione dei dati. Il vero confronto da prendere in considerazione però è quello tra le performance di scritture tra i due sistemi. Secondo un confronto di GigaOm [29] Pulsar è 2.5 volte più veloce e risulta avere una latenza più bassa del 40%.


Max Rate (messages/sec)
260,000
240,000
220,000
200,000
180,000
160,000
140,000
120,000
10,0000
80,000
60,000
40,000
20,000
0 20 40 60 80 100 120 140 160 180 200 220 240 260 280
Time (seconds)
Pulsar
Kafka


Average Message Latency (ms)
14.0
13.0
12.0
11.0
10.0
9.0
8.0
7.0
6.0
5.0
4.0
3.0
2.0
1.0
0 100 200 300 400 500 600 700 800
Time (seconds)
Kafka
Pulsar

2.2 Stream Processing

In questa sezione vengono analizzate le metodologie che operano sui dati in maniera distribuita su più nodi di elaborazione. Come prima cosa vengono classificate le tipologie di sistemi *real-time* con particolare attenzione sullo stream processing e le sue caratteristiche. Dopodiché viene descritto un framework distribuito e open-source che si occupa della gestione dei flussi dati: *Apache Storm*.

2.2.1 Definizione di sistema real-time

Al giorno d'oggi i dati vengono prodotti da telefoni, sensori, strumenti hardware, mezzi di trasporto, social media e anche se il concetto di Big Data esiste da tempo ora esistono anche le tecnologie in grado di gestire e analizzare questi flussi continui. Visto che viviamo in un periodo in cui è fondamentale fornire risultati efficaci immediatamente è quindi importante riuscire a costruire un sistema robusto in grado di soddisfare questa esigenza. Per questo motivo nascono i sistemi software *real-time* [14] e l'elaborazione dei dati in tempo reale. I sistemi real-time nella pratica ricevono dati in ingresso, effettuano operazioni su di essi e il risultato viene servito in tempistiche utili per avere un effetto sull'attuale contesto. Questi sistemi vengono identificati in tre modi: *hard*, *soft* e *near*.

Tipo	**Esempio**	**Latenza**	**Tolleranza**
Hard	Sistemi medici	Microsecondi, millisecondi	Nessuna — fallimento del sistema
Soft	Prenotazioni online, chiamate VoIP	Millisecondi, secondi	Bassa—nessun fallimento del sistema
Near	Automazione, domotica, Skype video	Secondi, minuti	Alta—nessun fallimento del sistema

Tabella 2.1: Differenze tra sistemi real-time

Questa classificazione è legata alla rigidità del sistema e alla sua capacità di rispettare le tempistiche definite. Un sistema *hard* infatti, è facilmente riconoscibile in tutti gli ambiti in cui è necessario operare all'interno di *deadline* temporali che non possono essere superate. Il superamento causa il totale fallimento del sistema e può causare gravi conseguenze.
La differenza tra un sistema *soft* e *near* è più difficile da distinguere visto che entrambe non causano il fallimento del sistema e l'ordine di grandezza della reattività in termini di tempo è simile. La differenza sostanziale è che in un sistema *near* se la risposta del sistema arriva dopo la deadline non ha più un valore e causa un crescente degrado del sistema in termini di qualità. L'esempio più evidente è una chiamata video Skype: quando la latenza aumenta il sistema degrada e i frammenti video che vengono ricevuti in ritardo non possono essere

utilizzati nel momento in cui arrivano. In un sistema *soft* un risultato che arriva in ritardo viene comunque accettato senza essere poi filtrato o scartato.

Se si separa la parte elaborazione da quella di consumo del risultato finale si può arrivare alla definizione di un particolare tipo di sistema real-time detto *Streaming Data System*. Un sistema che si occupa di streaming di dati è un sistema real-time non di tipo hard che rende disponibili i dati al client ogni volta che gli vengono richiesti. Questo tipo di sistema piuttosto che concentrarsi sulle tempistiche che caratterizzano i sistemi real-time *soft* e *near* consente di progettare un sistema che, in maniera affidabile, risponde alle richieste del consumatore finale. Il sistema ha a che fare con "data stream" cioè flussi dati illimitati che sono in costante crescita. Con il passare del tempo infatti il sistema riceve continuamente degli *eventi* dall'esterno che rappresentano un'occorrenza di qualcosa che è avvenuto, o che si prevede che possa avvenire. Questi eventi al loro interno hanno un contenuto detto *payload* che contiene il dato che subirà una eventuale trasformazione o che comunque entrerà a far parte del sistema.

Gli eventi in ingresso per definizione sono:

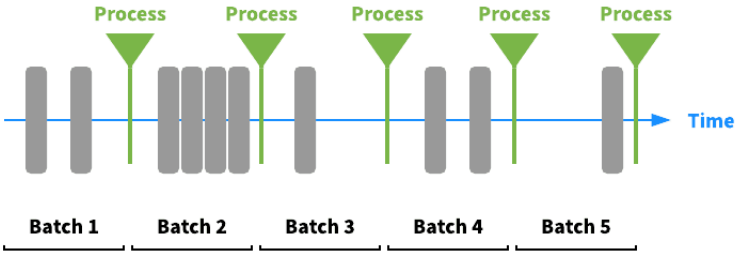
- *Ordinati*: un evento è legato ad un istante di tempo che lo colloca concettualmente prima e dopo un altro evento;
- *Immutabili*: una volta avvenuto un evento non può essere modificato;
- *Recuperabili*: anche eventi avvenuti tanto tempo prima devono essere disponibili per eseguire nuovamente operazioni di calcolo.

2.2.2 Tipologie di elaborazione

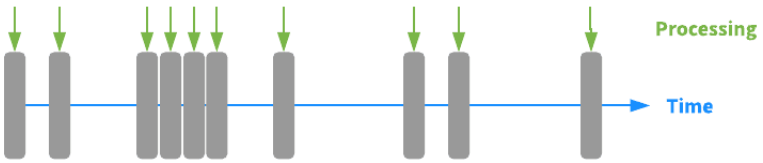
Esistono diversi paradigmi di progettazione per fare in modo di elaborare i dati nella maniera più efficace in termini di prestazioni e latenza del risultato. Spesso quando si va ad analizzare un sistema si presenta il tredoff tra *latenza* e *throughput*: il sistema deve essere in grado di produrre il risultato più in fretta possibile però allo stesso tempo deve riuscire a gestire un altro quantitativo di dati. Con latenza si intende il tempo necessario affinché un evento venga elaborato, più nello specifico la differenza tra l'arrivo dell'evento e il momento in cui è disponibile l'output finale. La latenza bassa è un requisito fondamentale dei sistemi di *stream processing*. Il throughput invece è la capacita di elaborazione di eventi, definisce un rapporto tra il numero di eventi elaborati all'interno dell'unità di tempo. Più eventi riesce a gestire più sono alte le performance di calcolo. Questo valore è tipicamente alto in sistemi di *batch processing*.

Esistono tre paradigmi di programmazione di un sistema real-time utilizzati anche nella Lambda Architecture: batch processing, stream processing e micro-batch processing.

Batch processing Questa modalità [30] fornisce un elevato numero di record elaborati, però impiega lunghi periodi di tempo per completare la computazione di tutti i record. L'elaborazione avviene ciclicamente in base alla frequenza di esecuzione: viene letto tutto l'input che è formato da una parte o da tutto il dataset, effettua le operazioni necessarie per generare l'output e aspetta fino all'esecuzione successiva. Questo paradigma è predisposto per scalare orizzontalmente dividendo il *workload* su più nodi però persiste il problema della latenza visto che il risultato finale non è disponibile fino al termine dell'elaborazione di tutto il *batch* di record che può impiegare molto tempo ad essere ottenuto.



Stream processing L'elaborazione in streaming [31] è un'operazione continua e non bloccante che consente di mantenere bassa la latenza fornendo una risposta immediata all'elaborazione di flussi dati illimitati. Questo modello agisce in tempo reale sul flusso in ingresso e permette di effettuare interrogazioni con tempi di risposta molto bassi che si aggirano nell'ordine dei millisecondi. Nei modelli business sono presenti innumerevoli fonti di dati che hanno la necessità di essere analizzati al volo poco tempo dopo essere stati prodotti e non possono essere completamente salvate su disco per limitazioni fisiche. In questo caso si utilizzano tecnologie che suddividono il lavoro su più nodi di elaborazione. L'aggiornamento delle viste real-time è continuo e quindi, al contrario di quelle batch, non vengono aggiornare soltanto dopo lunghi intervalli di tempo.



Micro-batch processing É la pratica [32] di collezionare i dati in piccoli gruppi detti anche questi *batch*. Il gruppo di record perso in considerazione per l'elaborazione è molto più piccolo rispetto a quello del *batch processing* però l'elaborazione avviene più frequentemente in modo da diminuire i tempi di aggiornamento dei risultati. Rispetto allo *streaming processing* è più semplice la gestione delle operazioni su un insieme di record però d'altra parte non è in grado di fornire la stessa reattività di un sistema in *stream*.

2.2.3 Concetti

L'elaborazione in tempo reale di flussi di dati è simile a qualsiasi altro tipo di elaborazione. Ci sono però alcuni concetti chiave specifici per questo dominio utili quando si progetta e analizza uno di questi sistemi. In questa sottosezione vengono spiegati i tre principali [33]: tempo, stato, finestra di tempo.

Tempo

É il parametro più importante all'interno di un sistema di elaborazione di dati. A mano a mano che il tempo scorre il sistema riceve eventi, ogni evento che viene prodotto, immagazzinato ed elaborato ha un riferimento temporale. Questo riferimento ha un valore che determina come e quando verrà utilizzato, determina la sua eventuale correttezza e in che modo verrà presentato all'interno dell'output finale. Nell'elaborazione dati distribuita il concetto di tempo acquista complessità e necessita di una gestione organizzata perché la maggior parte delle operazioni agisce su finestre temporali.
Tipicamente i sistemi di elaborazione dati si riferiscono al tempo in tre modi:

- *Event time*: è il momento in cui l'evento è effettivamente accaduto. Questo momento è espresso come un *timestamp* legato all'evento del flusso di elaborazione;
- *Ingestion time*: è l'istante di tempo in cui entra nel sistema di destinazione ed è in attesa di essere processato;
- *Processing time*: è il momento nel tempo dell'orologio locale della macchina in cui si iniziano le operazioni per elaborarlo.

Gli eventi però possono arrivare fuori sequenza e quindi compromettere l'ordine temporale. Un'applicazione che elabora dati o, più in generale, un sistema real-time, deve essere in grado di gestire questa situazione perché è molto frequente che ci siano problemi di connettività e i dati arrivino con un ritardo e quindi fuori sequenza.

Per fare in modo di minimizzare il numero di eventi scartati perché arrivati fuori sequenza e massimizzare l'accuratezza dell'elaborazione esiste il concetto

di *watermark*. É un metodo euristico che aiuta i sistemi di stream processing a gestire il ritardo dell'arrivo dei messaggi. In sistemi di questo tipo si utilizza intensamente la rete che per definizione non è affidabile quindi è necessario stabilire un limite temporale che indica quanto si aspetterà l'arrivo di eventi in ritardo. Il watermark è un timestamp che rappresenta la progressione dell'*event time* nel flusso dati. Se un evento in ritardo arriva prima del watermark allora viene accettato per l'elaborazione mentre se arriva dopo viene definitivamente scartato senza essere preso in considerazione.

Stato

Fintanto che il sistema deve elaborare un record alla volta è semplice mantenere l'elaborazione *stateless*, cioè senza uno stato interno da preservare, aggiornare ed eventualmente recuperare in caso di errori. Questo è il caso di semplici applicazioni che filtrano, modificano al volo i dati per correzioni e decodifica però appena la complessità aumenta è necessario gestire molteplici eventi simultaneamente. L'informazione derivante dall'elaborazione di più elementi viene chiamata *stato*.

Un tipico utilizzo dello stato è quello di mantenere in memoria un dato semplificato derivante dal gruppo di eventi presi in considerazione e aggiornato incrementalmente. Questa informazione riassume in una forma più compatta il flusso potenzialmente illimitato di dati. Se ad esempio abbiamo degli eventi che rappresentano la temperatura di un sensore, convertire il valore da gradi centigradi a gradi fahrenheit è un'operazione stateless mentre ottenere la temperatura minima e massima giornaliera richiede uno stato che conservi i valori risultanti.

Però non è triviale conservare lo stato dell'applicazione visto che la memoria, una volta fermato il processo o riavviata la macchina viene persa. Ci sono due modi per affrontare il problema:

- *Stato locale*: è gestito con un database in memoria nello stesso processo dell'applicazione, ha memoria limitata e quindi va gestita in modo da non sovraccaricarla di informazioni;
- *Stato esterno*: viene salvato in un database solitamente NoSQL capace di svolgere operazioni in maniera rapida mantenendo una capacità di memorizzazione virtualmente illimitata.

Queste due metodologie possono anche essere combinate in modo da creare una strategia di *caching* con lo stato interno e far persistere più a lungo e in maniera affidabile lo stato su database.

Finestra di tempo

La maggior parte delle operazioni in un sistema che elabora flussi di dati avviene in una finestra limitata di tempo. Alcune operazioni infatti per poter elaborare il risultato devono prima collezionare i record in un buffer creato all'interno di un frangente di tempo ben preciso. Lavorare su un gruppo di dati ristretto, oltre al vantaggio pratico sulle performance, permette di creare query semanticamente interessanti sui flussi dati. Non sempre interessa un frangente di tempo alto, ad esempio in un sistema che calcola lo stato del traffico soltanto l'andamento di pochi minuti interessa realmente a chi lo utilizza.

I parametri principali [18] quando si sceglie una finestra di tempo sono:

- *Dimensione della finestra*: avere una finestra di tempo più ampia causa meno overhead sulle operazioni però ha una latenza più elevata per quanto riguarda la disponibilità del risultato calcolato sulla finestra;
- *Frequenza di avanzamento*: indica secondo quale criterio viene creata una nuova finestra di tempo. Se ad esempio abbiamo una dimensione di cinque minuti, e la frequenza di avanzamento è di un minuto alcuni elementi faranno parte di più finestre e varranno utilizzati più volte per l'elaborazione;
- *Periodo di validità*: indica quanto tempo dopo la fine della finestra di tempo gli eventi in arrivo verranno considerati per l'elaborazione. Se ad esempio un evento arriva qualche minuto dopo possiamo decidere di mantenerlo e aggiornare il risultato della finestra mentre se arriva un giorno dopo è più corretto scartarlo.

Considerati questi fattori le modalità più comuni delle finestre di tempo sono:

- *Finestra fissa*: questa modalità detta *tumbling* pone una dimensione fissa e gestisce gli eventi in modo che vengano assegna ad una sola finestra senza sovrapposizione;

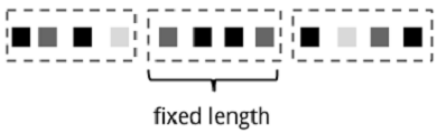


Figura 2.13: Esempio di finestra di tempo fissa [33].

- *Finestra scorrevole*: è una modalità simile a quella fissa, ha una dimensione fissa però la differenza sostanziale è che un evento può appartenere a due finestre distinte perché la finestra successiva non viene calcolata

quando quella precedente termina ma parte dopo un fattore diverso detto *sliding offset* che può essere definito in tempo o numero di eventi;

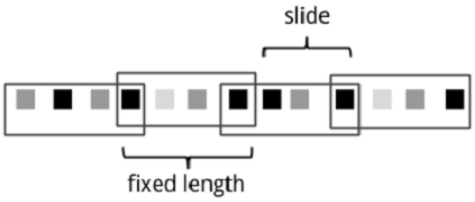


Figura 2.14: Esempio di finestra di tempo scorrevole [33].

- *Finestra a sessione*: questa modalità non è basata su un fattore fisso ma tiene in considerazione il comportamento dell'arrivo degli eventi da elaborare. Viene definita sessione un insieme di eventi *adiacenti* tra di loro. Questa modalità è utile quando la sporadicità è elevata e ci sono dei *gap* di tempo in cui non è necessaria elaborazione.

Figura 2.15: Esempio di finestra di tempo a sessione [33].

2.2.4 Pattern

Ogni sistema che elabora i dati è differente, ha un suo dominio applicativo e ha requisiti di performance ed elaborazione dati diverso. Esistono diversi principi di design base che forniscono soluzioni generali ai requisiti di sistemi di questo tipo. In questa sottosezione vengono presentati tre pattern ben noti e come vengono usati.

Single Event Processing É il pattern più semplice in assoluto visto che tratta singolarmente ogni evento in maniera isolata. Con questa tecnica viene preso in esame un evento, viene elaborato con una funzione base e viene emesso in un altro stream. Le funzioni utilizzate in questo pattern sono soltanto di trasformazione o che filtrano l'insieme di eventi.

Multi Event Processing Le applicazioni di stream processing si occupano principalmente di aggregazione di informazione, specialmente aggregazione in brevi finestre di tempo. Questo calcolo richiede di mantenere uno stato

interno che deve rispettare alcune limitazioni di memoria. Deve essere performante, non deve superare i limiti della macchina e deve persistere quando viene spenta. Questo pattern solitamente comprende anche la divisione in più fasi dell'elaborazione e la distribuzione su più macchine che eseguono una parte del lavoro.

Stream-Table Join É il pattern più comune quando il sistema ha la necessità di integrarsi con un sistema esterno. Il pattern comune è quello di associare il dato di un evento ad un elemento presente in un database. Questa associazione arricchisce il dato all'interno dell'evento trasformandolo in informazione. Questo *join* introduce una significativa latenza che viene mitigata da sistemi di *cache*.

Streaming Join A volte è più utile unire un flusso dati piuttosto che una tabella. In questo caso è più complesso perché va mantenuta tutta la storia degli eventi e tentare di trovare una corrispondenza soltanto su eventi avvenuti all'interno della stessa finestra di tempo. Questo tipo di *join* è anche detto *windowed-join* ossia una unione all'interno di una finestra di tempo.

2.2.5 Apache Storm

Nella sezione introduttiva dello *stream processing* sono emersi i concetti che definiscono un sistema real-time e le valutazioni da fare quando si sviluppa un sistema di quel tipo. In questa sezione viene presentato uno tra i principali framework di elaborazione distribuita open-source: **Apache Storm**. É uno dei framework che opera più a basso livello con concetti riconducibili alle basi teoriche viste in precedenza. Infine, dopo l'analisi vengono proposte due valide alternative attualmente utilizzate per la computazione ed elaborazione di dati in maniera distribuita.

Definizione

Inizialmente pensato e sviluppato da Nathan Marz [34], è entrato a far parte dell'ecosistema Apache dal Dicembre 2013. Successivamente è stato inserito all'interno del progetto Apache Incubator fino a diventare grazie una community molto vasta uno tra i principali progetti della *Apache Software Foundation* nel Novembre 2015. Ancora oggi è utilizzato all'interno di aziende come Yahoo, Spotify e Twitter come principale tool di analisi dati in tempo reale ed elaborazione di log di sistema.

Apache Storm [35] è un framework computazionale distribuito che elabora flussi di dati in tempo reale in maniera fault-tolerant. É un servizio che agisce su singola macchina oppure su più macchine separate rimanendo in attesa di nuovi dati. Una volta arrivati, i dati subiscono elaborazioni e trasformazioni attraverso i diversi moduli e vengono successivamente emessi in output sotto forma stream continuo.
I principali motivi per cui viene utilizzato Apache Storm in produzione sono:

- Fornisce modularità e può essere utilizzato in un'ampia varietà di ambiti;
- Si integra facilmente con molti framework open-source Big Data come, ad esempio, Apache Kafka;
- É scalabile e introduce concetti che permettono di dividere il carico;
- Fornisce una API semplice e il servizio è in grado di essere eseguito su ogni sistema che supporta la *JVM* (Java Virtual Machine);
- É un sistema robusto e tollerante agli errori che garantisce che i messaggi arrivino a destinazione almeno una volta (*at-least-once*).

Prima di Storm, per fare elaborazione veniva utilizzato soltanto *Apache Hadoop* [36]. Hadoop è un framework che consente l'elaborazione distribuita di grandi set di dati su un cluster di computer utilizzando semplici modelli di programmazione. Inizialmente era soltanto un framework di elaborazione *batch* mentre con l'uscita della versione 2 si è evoluto in un vero e proprio ecosistema composto di diversi moduli [37]:

- *Hadoop Common*: utilità di supporto agli altri moduli;
- *YARN*: un framework per la pianificazione del lavoro e la gestione delle risorse del cluster;
- *Map Reduce*: un modulo per l'implementazione del paradigma di elaborazione di grandi quantitativi di dati;
- *HDFS*: un file system distribuito e scalabile che fornisce accesso consistente ai dati che sono salvati e replicati su più macchine.

Storm introduce una progettazione dell'architettura completamente rivoluzionaria basata su concetti che saranno analizzati nella sezione successiva.

Architettura logica e componenti

Come si può vedere in Figura 2.17 l'architettura logica di un'applicazione Storm è formata da pochi componenti collegati tra di loro sotto forma di grafo orientato. In questa sottosezione sono elencati i concetti principali che consentono di progettare un sistema di elaborazione real-time con solide garanzie di servizio.

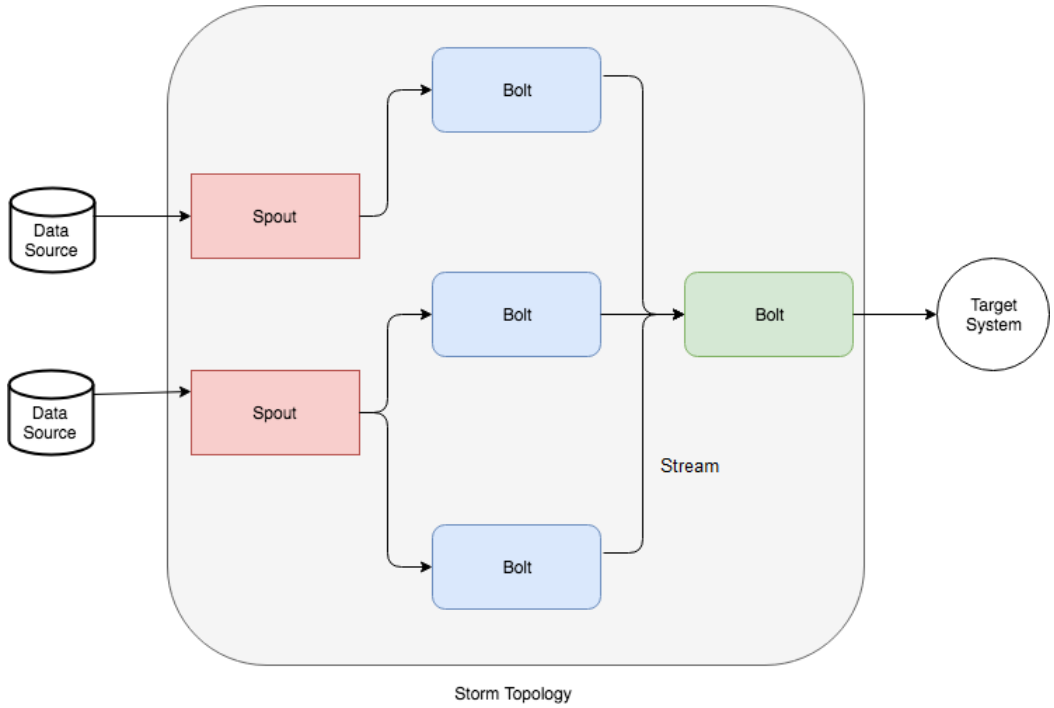


Figura 2.16: Architettura logica di un'applicazione Storm.

Topologia Una topologia è un grafo orientato di elaborazione dove i nodi sono gli elementi in grado di eseguire codice e computazione mentre i vertici

sono le connessioni che permettono ai dati di transitare all'interno della topologia. I dati vengono rappresentati sotto forma di *Tuple* mentre i flussi che collegano i vari nodi vengono chiamati *stream*.

Tupla I nodi della topologia si scambiano messaggi e informazioni sotto forma di strutture dati dette Tuple. Queste sono la principale astrazione logica di Storm e sono considerate come una lista di valori dinamicamente tipizzati. I valori formano una lista ordinata in cui ad ogni elemento viene assegnato un nome con il quale accedere al valore interno della lista. Ogni nodo è in grado di creare ed eventualmente *emettere* tuple che, dopo essere state serializzate, vengono inviate all'esterno come output sotto forma di stream.

Stream É una sequenza illimitata di tuple tra due nodi all'interno della topologia. Oltre il nodo principale che si occupa dell'introduzione dei all'interno del sistema gli altri possono ricevere uno o più stream come input.

Spout Si occupa della ingestione dei dati. É la sorgente principale dello stream di dati in ingresso. Questo particolare tipo di nodo si connette ad un servizio e resta in attesa di eventi real-time. Lo spout può ricevere messaggi in arrivo da code di messaggi, record aggiornati ed inseriti in un database, oppure feed di eventi di API. Una particolarità rilevante di questo tipo di nodo è che non possono effettuare elaborazione ma si occupano soltanto di emetterli nella topologia.
Le primitive [38] di questo nodo sono:

- `nextTuple()`: metodo chiamato ogni volta che arriva un record da elaborare;
- `ack(msgId)`: meccanismo che notifica quando una tupla è stata completamente elaborata dalla topologia;
- `fail(msgId)`: metodo chiamato ogni volta che una tupla non ha terminato l'elaborazione o è andata in timeout.

Bolt Tutta l'elaborazione del sistema viene fatta nei nodi definiti nel grafo della topologia detti bolt. In ingresso ricevono uno stream di tuple dallo spout oppure da un altro bolt. Subito dopo aver ricevuto una tupla si occupano di svolgere trasformazioni, filtrare oppure unire il dato ad altri flussi avanzati. Una volta elaborato infine i bolt, attraverso le funzionalità fornite, eventualmente possono emettere una tupla nel loro stream di output a cui faranno riferimento i bolt connessi.
Le primitive [38] di questi nodi sono:

- `prepare(Map conf, TopologyContext ctx, OutputCollector oc)` : questo metodo viene invocato quando il bolt è stato creato e può essere eseguito lato server;
- `declareOutputFields(OutputFieldsDeclarer declarer)`: in questo metodo si dichiara quali saranno i campi eventualmente emessi durante l'elaborazione;
- `execute(Tuple input)`: ogni tupla ricevuta chiama questo metodo che può eseguire qualsiasi tipo di elaborazione, inviare l'output oppure salvare su database il dato.

Entrambi i metodi vengono eseguiti da Storm nel momento più opportuno e non hanno un valore di ritorno. Dentro `prepare` viene inizializzato il contesto interno al Bolt mentre in `execute` viene utilizzato `OutputCollector` per emettere all'interno dello stream un valore.

```
public class MathBolt extends BaseRichBolt {
    private OutputCollector outputCollector;
    @Override
    public void prepare(Map conf, TopologyContext context, OutputCollector collector) {
        this.outputCollector = collector;
    }
    @Override
    public void declareOutputFields(OutputFieldsDeclarer declarer) {
        // Dichiaro i campi che verranno emessi
        declarer.declare(new Fields("double", "triple"));
    }
    @Override
    public void execute(Tuple tuple) {
        int value = tuple.getInteger(0);
        // Nuovo valore emesso
        this.outputCollector.emit(tuple, new Values(value*2, value*3));
    }
}
```

Questa semplice implementazione mostra che per costruire un Bolt è sufficiente estendere la classe fornita `BaseRichBolt` e implementare i metodi elencati sopra. Nell'esempio viene ricevuto un numero intero e ne viene emesso il suo doppio e il suo triplo. Inizialmente viene mantenuto il riferimento ad `OutputCollector` e vengono dichiarati i `Fields` che verranno emessi ossia *double* e *triple*. Quando viene effettivamente chiamato il metodo `execute` viene calcolato il risultato a partire dal valore intero che si suppone essere all'interno della tupla ricevuta.

Tipi di grouping Visto che uno stream è un flusso illimitato di tuple che transita tra spout e bolt è necessario definire il modo in cui vengono inviati i

dati per massimizzare le prestazioni ed aumentare il bilanciamento del carico. La tecnica di grouping [39] quindi stabilisce come un bolt deve inviare le tuple ai nodi collegati.

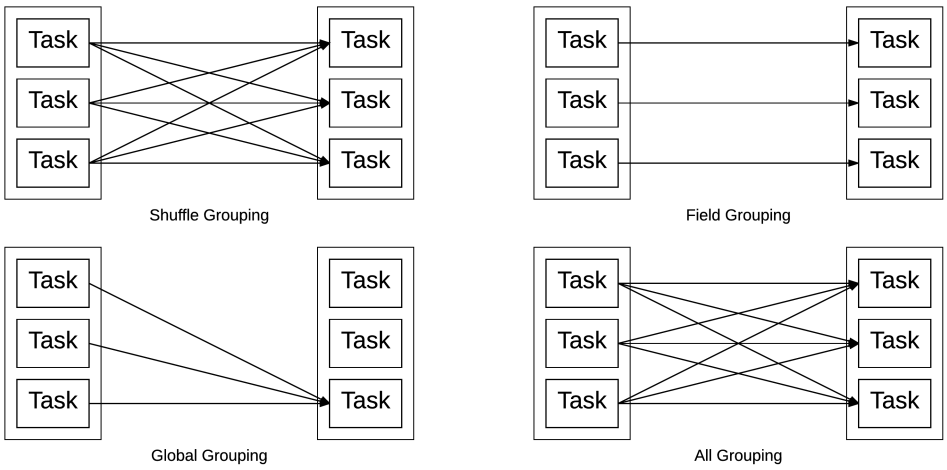


Figura 2.17: Tipologie di Grouping tra diversi Bolt [39].

Com'è visibile in figura 2.17 è possibile categorizzare gli *stream grouping* in quattro tipologie:

- *Shuffle grouping:* le tuple emesse vengono distribuite in maniera casuale tra i bolt. Questo permette di bilanciare equamente il carico garantendo che arrivino circa lo stesso numero ad ogni destinatario;
- *Field grouping:* assicura che le tuple con lo stesso valore in un particolare campo vengano sempre emesse allo stesso destinatario. Il vantaggio è che all'interno dei nodi è possibile assumere una coerenza logica tra i record ossia tutte le tuple con una certa caratteristica saranno processate dallo stesso bolt;
- *Global grouping:* questo tipo invece permette di far confluire tutti gli stream in un singolo nodo (quello con identificativo minore) in modo da aggregare i risultati delle elaborazioni precedenti;
- *All grouping:* invia una copia della tupla ad ogni bolt collegato. Questa operazione *boradcast* permette di mandare direttive a tutti i nodi connessi.

Per casi d'uso più complessi è anche possibile implementare un algoritmo personalizzato di smistamento estendendo la classe `CustomStreamGrouping` fornita dalle API di Storm.

Struttura fisica

Storm è sviluppato per massimizzare il throughput e minimizzare la latenza di risposta continuando a garantire tolleranza agli errori. Questo è possibile grazie ai vantaggi forniti dall'elaborazione distribuita su più macchine. La struttura fisica (figura 2.18) è organizzata secondo l'architettura standard *master/slave* dei sistemi distribuiti. Questa architettura è composta da due tipi di elementi: il master detto *Nimbus* gli slave detti *Supervisor*.

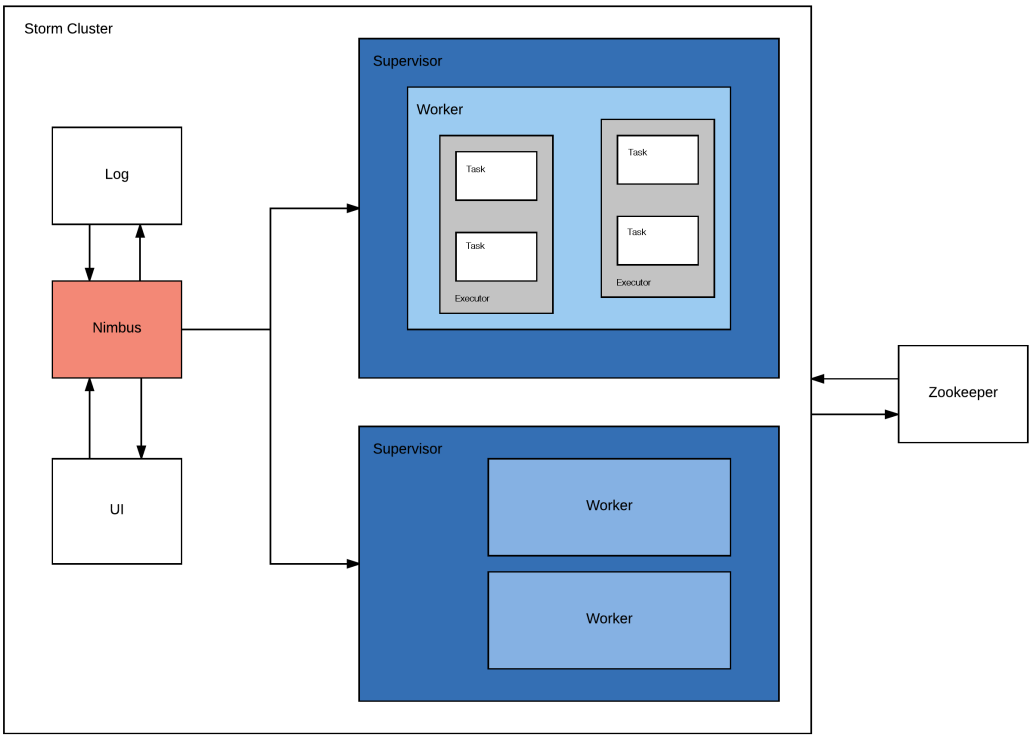


Figura 2.18: Architettura fisica di un cluster Storm.

Il Nimbus è il principale responsabile della gestione del codice da eseguire sul cluster e si occupa di dove vengono caricate fisicamente le topologie prima dell'elaborazione. É una entità unica pensata come il centro amministrativo che però non è possibile replicare, infatti non è possibile avere più di un servizio Nimbus attivo nello stesso cluster. Nonostante ciò Storm riesce a gestire il fallimento in tre modi:

- Il processo in esecuzione è *stateless* ossia non ha nulla riguardante lo stato interno del cluster salvato in memoria;

- Il sistema è *fail-fast* ossia ogni volta che c'è un'inconsistenza il Nimbus si autodistrugge in modo da ricreare uno stato consistente leggendolo da disco o dal registro distribuito di *Zookeeper*;
- Il fallimento del nodo Nimbus non blocca i task in esecuzione sui superivsors.

Oltre a svolgere funzioni amministrative il master si occupa di verificare l'integrità degli altri nodi verificandone il carico di lavoro e il loro stato.

I nodi slave a loro volta hanno un servizio *daemon* detto Supervisor che si occupa di di mantenere i processi di calcolo in uno stato di esecuzione. Normalmente i Supervisor si trovano su macchine diverse che possono essere aggiunte o rimosse modularmente, scalando così linearmente fino a formare cluster con centinaia o migliaia servizi slave in esecuzione. I processi di calcolo, detti *worker process*, eseguono un eventuale sottoinsieme della topologia caricata dal Nimbus.

Un'applicazione Storm, insomma, si identifica come nient'altro che un insieme di molti *worker process* che vengono eseguiti su più macchine all'interno del cluster. Ogni *worker process*, esegue una JVM che esegue una porzione di codice dell'intera applicazione. Analogamente, ogni worker esegue al suo interno più executor (*thread*) che, a loro volta, comprendono uno o più task paralleli in esecuzione. Il numero di task determina il grado di parallelismo dell'applicazione: un maggior numero di task modulari implica anche una maggior distribuzione all'interno dei nodi di elaborazione ma allo stesso tempo anche un overhead causato dalla gestione e la comunicazione di questi servizi.

Storm utilizza Apache Zookeeper per la comunicazione tra le macchine e permette al nimbus di rimanere stateless. Nel registro distribuito condivide la configurazione con tutto il cluster in modo che in ogni momento ogni nodo possa accedervi e, in caso di errore, possa recuperarla quando il servizio riprende a funzionare.

2.2.6 Alternative a Storm

Nella seguente sezione vengono presentate le due principali alternative ad Apache Storm per l'elaborazione di flussi dati. I framework per fare ciò sono due progetti Apache che implementano in maniera differente il modello dati e che hanno un approccio diverso al problema. Il primo, *Apache Spark* è più orientato verso un pattern *micro-batch* mentre il secondo *Apache Flink* fornisce interfacce ad alto livello per gestire in modo semplice il flusso dati.

Apache Spark

Spark [40] è stato creato nel 2009 come progetto all'interno dell'AMPLab all' università della California, Berkeley. É un framework open-source per l'analisi di grandi quantità di dati su cluster, nato per essere veloce e flessibile. É sviluppato in *Scala*, un linguaggio di programmazione funzionale che opera sulla JVM, su cui è possibile lavorarci tramite l'apposita shell oppure sviluppando vere e proprie applicazioni.

Spark implementa un modulo di computazione ottimizzato che viene sfruttato dai suoi componenti. Il core di Spark si occupa delle funzionalità base, come lo scheduling, la distribuzione e il controllo dell'esecuzione dell'applicazione utente.

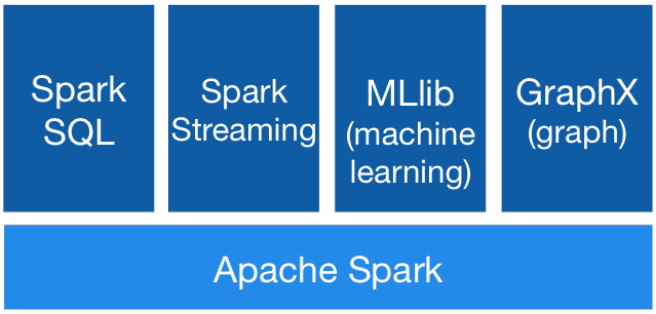


Figura 2.19: Stack dei moduli Spark [40].

Come si vede in figura 2.19 ad utilizzare il core di Spark ci sono vari moduli [41] che operano in contesti completamente diversi tra di loro:

- *Spark SQL*: permette di fare interrogazioni su dati strutturati e semi strutturati usando una variante del linguaggio SQL. Questo modulo rende semplice la visita di file di testo, file JSON o file serializzati attraverso query simili a quelle dei database relazionali;
- *Spark Streaming*: questo modulo permette di analizzare flussi di dati in tempo reale operando sulle astrazioni logiche in memoria del modulo core;
- *MLib*: è una libreria che implementa algoritmi di Machine Learning ottimizzati per essere eseguiti sullo Spark core in memoria. Il pieno potenziale di questa libreria è sfruttato su algoritmi parallelizzabili e quindi distribuibili su più istanze Spark;
- *Graph X*: è una libreria per la visita e l'analisi di grafi molto grandi che non possono essere analizzati su una singola macchina.

La principale astrazione di Spark nel modello dati sono gli **RDD** (Resilient Distributed Dataset). Un RDD quindi è una collezione fault-tolerant

e non modificabile di elementi sulla quale si può operare in parallelo. Essa è memorizzata in maniera distribuita, spezzata in parti chiamate partizioni, ognuna delle quali sta in un esecutore di Spark. Ogni applicativo Spark si occupa solitamente di caricare dati su un RDD, trasformare i dati ed infine estrarli grazie alle semplici API fornite in diversi linguaggi come Java, Scala e Python.

Un componente importante è lo **SparkContext**, cioè cuore di un'applicazione Spark. La sua funzione è creare una connessione con l'ambiente di esecuzione Spark, viene utilizzato per creare gli RDD, accumulatori e variabili broadcast, accedere ai servizi Spark ed eseguire le unità di lavoro rimanenti. Lo SparkContext è un client dell'ambiente di esecuzione Spark e funge da master dell'applicazione. Alcune principali mansioni dello Spark Context sono il controllo dello stato attuale dell'applicazione, l'annullamento di un lavoro o di uno stage, l'esecuzione del lavoro in modo sincrono o asincrono, l'accesso ad RDD persistenti o non e l'allocazione dinamica programmabile

La principale differenza tra Spark e Storm [42] è che il primo agisce su piccole *batch* di dati mentre il secondo elabora ogni elemento in input singolarmente. Spark garantisce una elaborazione *stateful*, ossia che mantiene uno stato interno in memoria. Anche se Spark implementa un pattern *micro-batch* la latenza è molto maggiore perché l'elaborazione viene fatta su più dati e nonostante queste batch siano di piccole dimensioni. Il vantaggio di Spark rispetto a Storm invece è che è compatibile con l'ecosistema *Hadoop* e quindi sfrutta l'efficienza del modulo YARN. L'integrazione nativa con questo modulo offre infatti la funzione di gestione delle risorse e scheduling delle operazioni in maniera separata all'applicazione mantenendo quindi la modularità e l'efficienza.

Apache Flink

Apache Flink [43, 33] è un framework di elaborazione distribuita di flussi di dati illimitati e anche limitati. É progettato per essere eseguito in tutti gli ambienti cluster con prestazioni riconducibili all'elaborazione in memoria. Come già detto i flussi possono essere classificati in due tipi [43]:

- *Flussi limitati*: hanno un inizio ma nessuna fine definita. Non terminano e forniscono dati così come sono generati. Devono essere elaborati continuamente, vale a dire che gli eventi devono essere prontamente gestiti dopo essere stati immagazzinati. Non è possibile attendere l'arrivo di tutti i dati di input poiché l'input non sarà mai completo;
- *Flussi illimitati*: hanno un inizio e una fine definiti. Possono essere elaborati immagazzinando tutti i dati prima di eseguire qualsiasi calcolo.

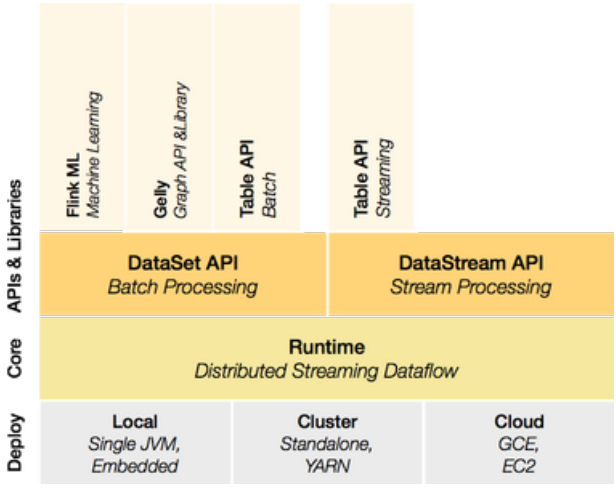


Figura 2.20: Architettura di Apache Flink [44].

Come illustrato in figura 2.20 Apache Flink include due API di base: una è la *DataStream* per flussi di dati limitati o illimitati e l'altra *DataSet* per set di dati limitati. Flink offre anche la Table API, un linguaggio simile a SQL per il flusso relazionale e l'elaborazione batch che può essere facilmente incorporato nella DataStream API e DataSet API [45].

La DataStream API Consente trasformazioni, come ad esempio filtri, aggregazioni e window functions, su flussi di dati limitati o illimitati. Questa API include più di 20 diversi tipi di trasformazioni ed è disponibile sia in Java che Scala. La DataSet API consente trasformazioni, come filtri, mappature, unioni e raggruppamenti su set di dati limitati.

Al momento dell'esecuzione, i programmi Flink sono mappati in *stream* dei flussi di dati. Ogni flusso di dati inizia con una o più fonti come input, ad esempio una coda di messaggi o un file system, e termina con uno o più sink come output, ad esempio una coda di messaggi, un file system o un database. Può essere eseguito sullo stream un numero arbitrario di trasformazioni. I componenti di base con cui Flink lavora sono:

- *Streams*: set di dati immutabili e illimitati che attraversano il sistema;
- *Operators*: funzioni che operano su flussi di dati per produrre altri flussi;
- *Sources*: il punto di ingresso per gli stream che entrano nel sistema;
- *Sinks*: il luogo in cui i flussi escono dal sistema Flink. Potrebbero rappresentare un database o un connettore a un altro sistema.

Il principale vantaggio di Flink è che permette con un framework nativamente sia l'elaborazione batch sia lo stream processing. Mentre Storm si occupa esclusivamente dello streaming, Flink è possibile utilizzarlo sia nel livello *batch* che nel livello *speed* dell'architettura Lambda. Questo facilita lo

sviluppo riducendo il numero di framework con architetture diverse e applicativi da mantenere attivi allo stesso momento.
Confrontato a Storm però Flink offre una API più ad alto livello più semplice da implementare. Invece che implementare i bolt a basso livello, la Datastream API fornisce funzioni come `map`, `flatMap`, `groupBy`, gestione delle finestre di tempo (*Window*) e *join* degli stream.
L'ultima differenza è quella della semantica di elaborazione. Storm offre una garanzia di *at-least-once* mentre Flink garantisce una *exactly-once*. Flink include un meccanismo di tolleranza agli errori basato su checkpoint distribuiti. Un checkpoint è uno snapshot asincrono e automatico dello stato di un'applicazione e la sua posizione all'interno di un flusso di dati. In caso di guasto, con questo sistema di ripristino, l'applicazione riprende l'elaborazione dall'ultimo checkpoint completato, assicurando che il framework analizzi esattamente una volta ogni elemento all'interno di un'applicazione. Storm offre una semantica *exactly-once* con la sua implementazione ad alto livello chiama Trident che però è basata su micro-batch quindi è riconducibile più a Spark.

2.3 Database NoSQL

In questa sezione vengono analizzate le proprietà e le particolarità del trend NoSQL. Viene poi fatto un confronto tra i database tradizionali, che utilizzano il modello relazionale, e i database non relazionali evidenziandone i vantaggi e gli svantaggi. Infine, viene presentato uno tra i principali database non relazionali: **Apache HBase**.

2.3.1 Introduzione

I dati rappresentano al giorno d'oggi una quantità enorme di informazioni raccolte continuamente e con volumi senza precedenti. Negli anni Settanta Edgar F. Codd dava vita al modello logico più utilizzato per la rappresentazione e gestione dei dati. Il concetto degli RDBMS (*Relational Database Management System*) rappresenta ancora uno tra i più importanti ambiti dei sistemi informativi, basti pensare alla vasta quantità di DBMS (*Database Management System*) che vengono utilizzati come principale supporto per la memorizzazione dei dati strutturati come Oracle, MySQL e Microsoft SQL Server, tutti quanti DBMS relazionali ancora utilizzati in larga scala.

Database relazionali Dopo la pubblicazione di Edgar Codd dell'articolo "*A Relational Model of Data for Large Shared Data Banks*" quello relazionale è divenuto il modello logico più utilizzato nel panorama della rappresentazione e gestione dei dati: la teoria della basi di dati relazionali. La *tabella* è l'elemento base dei database relazionali, ne esiste una per ogni elemento del dominio applicativo da trattare, ognuna costituita da *attributi*, uno per ogni aspetto o caratteristica dei dati. Ogni tabella possiede uno o più attributi che svolgono il ruolo di *chiave primaria* che identifica univocamente una certa tupla appartenente alla tabella. Tra le tabelle di un database relazionale possono esistere appunto delle *relazioni* ossia delle connessioni tra tabelle che indicano un legame logico tra le entità rappresentate. In questo tipo di database viene applicato un livello di astrazione tra il modello implementativo e la gestione fisica dei dati. Questo per la prima volta nel panorama dei dati fornisce la possibilità di non doversi curare dell'effettiva strutturazione fisica dei dati né delle sue possibili variazioni, così come di eventuali modifiche apportate.

I database relazionali hanno dominato in modo indiscusso il settore delle basi di dati per gli ultimi cinquant'anni. Nonostante ciò nell'ultimo decennio sono avvenuti cambiamenti che hanno portato in evidenza limitazioni notevoli. I dati non sono più organizzati rigidamente come in passato: le informazioni derivano da fonti disparate e non sempre affidabili. Il modello dati varia rapidamente nel tempo e il database che si occupa dei dati deve poter variare

allo stesso modo in maniera dinamica. Anche se esistono tecniche di distribuzione del lavoro sui database relazionali come ad esempio lo *sharding* non possiamo affermare che renda stabile e robusto il sistema. Aggiungere macchine ad un database che effettua sharding migliora soltanto le scritture lasciando però aperti i problemi di tolleranza agli errori e scalabilità orizzontale. Tutto ciò porta, con il crescere dei dati, alla crescita della domanda di prestazioni e performance.

Database NoSQL Tutte le tecnologie che si distaccano dai tradizionali RDBMS vengono definite come NoSQL che è l'acronimo di "*Not only SQL*" [46]. Questo movimento nasce con l'obiettivo di supportare e colmare le limitazioni dei database relazionali per affrontare in modo efficiente le sfide poste dal mondo Big Data.

Il modello relazionale possiede una rigidità intrinseca che permette di dare uno schema logico ben strutturato ad un dominio applicativo. Questa rigidità viene persa nei sistemi NoSQL visto che si identificano come *schemaless*, ossia senza schema, consentono quindi alla struttura logica di variare nel tempo senza incontrare problemi o danni sulle performance.

Un altro aspetto fondamentale riguarda la possibilità di scalare orizzontalmente con semplicità. La maggior parte dei DBMS NoSQL *by-design* viene progettato con lo scopo di operare su un cluster distribuito su più macchine fisiche diverse. Questa esigenza nasce per quattro motivi:

- Riuscire a gestire volumi che non è possibile archiviare su singola macchina;
- Migliorare le prestazioni di risposta e disponibilità;
- Distribuire letture e scritture sui vari nodi del cluster per aumentare lo throughput;
- Non si ha necessità di acquistare hardware particolarmente performante e costoso per poter mantenere prestazioni elevate.

I sistemi NoSQL comunemente mantengono più copie dello stesso dato sparse in maniera intelligente sulle varie macchine per motivi di sicurezza, per favorire la disponibilità dei dati e per poter distribuire fra i server le operazioni di lettura. Tali sistemi, infatti, spesso mettono a disposizione semplici ed efficaci politiche non solo di scaling orizzontale ma anche di *replication*. Con replicazione [12] si intende quel meccanismo che mantiene la stessa copia del dato su più macchine che sono interconnesse tra di loro. Questo introduce il vantaggio di non perdere il dato in caso di fallimento di una macchina ma introduce la complessità di dover mantenere aggiornate le copie a fronte del cambiamento dei dati.

2.3.2 RDBMS vs NoSQL

La caratteristica principale dei database relazionali è quella di garantire la consistenza dei dati mentre, al contrario, i database NoSQL si pongono un altro obiettivo: garantire una grande disponibilità a scapito però della consistenza. Le principali differenze sono:

- *Struttura dei dati*: i database relazionali che utilizzano SQL per memorizzare i dati necessitano di una struttura con degli attributi definiti, a differenza dei database NoSQL che possiedono una struttura più libera;
- *Query*: tutti i database relazionali SQL implementano un certo linguaggio standard per eseguire le query sui dati. I database NoSQL invece definiscono un modo proprietario di accedere ai dati;
- *Scalabilità*: i database NoSQL offrono solitamente un modo semplice di scalare orizzontalmente;
- *Complessità dei dati*: i database relazionali si basano su dati con relazioni e strutture più complesse mentre i database NoSQL trascurano intenzionalmente vincoli di integrità e forme normali dei database tradizionali.

I database relazionali si basano sulle proprietà ACID mentre i database NoSQL si basano sulle proprietà BASE. In questa sezione vengono spiegati entrambi gli approcci.

Proprietà ACID

Le proprietà ACID [47] descrivono le caratteristiche che devono avere le transazioni. Una transazione consiste in un insieme di operazioni effettuate su un database che, partendo da una situazione consistente, una volta terminate lo portano ad una situazione finale consistente. Una transazione ti dà la garanzia che, nel caso non andasse a buon fine, verrebbe avviata una operazione di ripristino, detta *rollback*, che lo riporta ad una situazione consistente.

L'acronimo ACID si riferisce dunque alle quattro proprietà:

- **Atomicity – Atomicità**
  L'esecuzione di una transazione deve essere completa, ossia tutte le operazioni devono andare a buon fine oppure nulla cioè nessuna deve avere effetto;
- **Consistency – Coerenza**
  Si richiede che la transazione non violi i vincoli di integrità. Questo significa che il database deve essere in uno stato consistente sia prima che dopo l'esecuzione;

- **Isolation – Isolamento**
  Le transazioni non devono andare in conflitto e devono rimanere indipendenti anche in caso di fallimento;
- **Durability – Persistenza**
  Una volta terminata la transazione bisogna garantire la permanenza dei dati.

Proprietà BASE

L'approccio ACID è caratteristico dei sistemi relazionali poiché pone al centro dell'attenzione la correttezza dei dati e la consistenza (*consistency*) delle operazioni. Questo tipo di garanzie sono difficili da ottenere e mantenere con una mole elevata di operazioni. Questa sfida mette a dura prova i tradizionali RDBMS. Nel caso dei sistemi NoSQL invece, ruolo centrale viene spesso riconosciuto alla disponibilità dei dati (*availability*), frequentemente si accetta infatti di sacrificare parzialmente la consistenza per poter garantire disponibilità e tolleranza al partizionamento (*partition tolerance*).

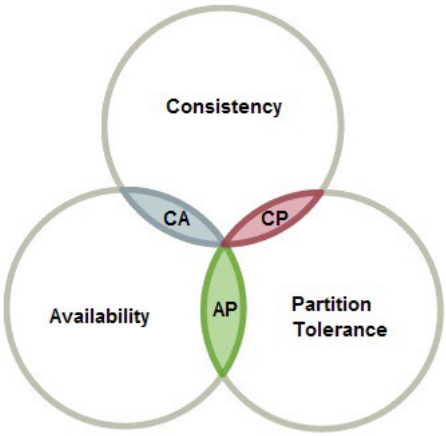


Figura 2.21: Rappresentazione del teorema CAP e i possibili casi [10].

Come possiamo notare dall'intersezione delle tre proprietà del teorema CAP [48] in figura 2.21 ci sono tre coppie di garanzie possibili. La prima cioé *Consistency e Availability* (CA) si può individuare nei database relazionali e non è realizzabile nel caso di sistemi distribuiti dove la tolleranza ai partizionamenti costituisce una proprietà che difficilmente potrà essere sacrificata a favore di altre garanzie che potrebbero, in un contesto altamente decentralizzato, non essere strettamente necessarie. Quindi i casi più comuni nelle soluzioni NoSQL sono:

- *Consistency e Partition Tolerance* (CP): questa scelta permette di mantenere i dati presenti consistenti su tutti i nodi che fanno parte del sistema. Se un nodo però fallisce o non è momentaneamente disponibile non c'è garanzia che il sistema risponda correttamente. Questa scelta è opportuna quando è necessario che tutti i nodi ricevano una vista coerente tra tutti i nodi. Per poter procedere con letture e scritture quindi è necessario che abbiano ricevuto il dato e siano in accordo sulla consistenza dello *snapshot* del sistema;
- *Availability e Partition Tolerance* (AP): garantire disponibilità del sistema implica la rinuncia alla consistenza dei dati nei vari nodi. Spesso è necessario che il sistema risponda in ogni caso alle richieste anche se ci sono dei malfunzionamenti parziali. Una volta risolti però possono presentarsi situazioni di conflitto tra dati non sincronizzati correttamente. Questo compromesso diminuisce la latenza di risposta, garantisce la ricezione di una risposta e soprattutto fornisce un incremento di performance lineare con l'aggiunta di nuove macchine scalando così orizzontalmente.

Questa scelta nei principali framework NoSQL moderni non è sempre totalmente binaria. É possibile fare riferimento ad un modello più flessibile che viene identificato con l'acronimo BASE. In questo modello introdotto da *Eric Browers*, autore dello stesso teorema CAP, si ha un sistema pressoché sempre disponibile (*Basically Availble*) dove si hanno fasi transitorie di inconsistenza (*Soft state*) che comunque a tendere nel tempo portano ad una situazione finale consistente (*Eventual consistency*).

2.3.3 Classificazione dei sistemi NoSQL

Il movimento NoSQL offre una ricca gamma di soluzioni fra le quali è possibile scegliere sulla base della specifica classe di problemi che si ha necessità di affrontare. I database NoSQL sono classificati in base al tipo di modello [49] che utilizzano per la memorizzazione dei dati e per come vi accedono in lettura e scrittura. In particolare, possono essere individuate quattro grandi famiglie:

1. Key-Values stores;
2. Column-oriented database;
3. Document database;
4. Graph database.

Nelle seguenti sottosezioni, per ogni tipologia, viene analizzata la struttura logica, all'organizzazione dei dati e i contesti in cui è opportuno utilizzarla.

Key-Value stores

I key-value stores [50, 48] sono il modello più semplice ed intuitivo tra i quattro proposti. In questa tipologia vengono memorizzate coppie chiave-valore dove ciascuna chiave identifica univocamente il valore associato. Solitamente forniscono accesso ai dati soltanto tramite chiave primaria e non implementano indici secondari sul valore di riferimento. Come si più vedere dalla figura 2.22 ogni tabella può avere una implementazione di una *key space* che è un insieme logico di chiavi.

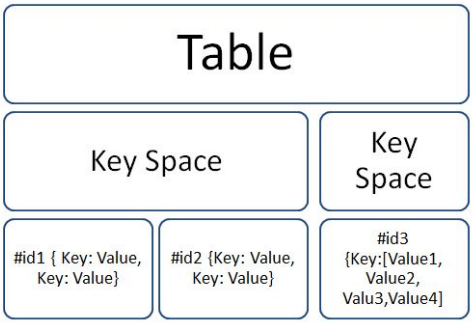


Figura 2.22: Struttura di un database key-value [49].

Questa soluzione fornisce altissime performance (tipicamente in lettura), facilitando, al tempo stesso, il lavoro di distribuzione del carico su più macchine (partizionamento o *sharding*) rendendo la scalabilità del database quasi lineare. Il vantaggio di un sistema di questo tipo è l'altissima concorrenza e quindi, la disponibilità delle informazioni, a discapito della consistenza dei dati. É molto utile utilizzarlo nei casi in cui non è possibile definire uno schema sui dati e quando c'è la necessità di accedere velocemente alle informazioni come ad esempio un sistema di *cache*.

Column-oriented database

I column-oriented database [50, 51] riprendono lo schema dei RDBMS tradizionali, infatti salvano i dati in strutture tabellari composte da righe e colonne. La differenza sostanziale è che le informazioni non sono memorizzate per riga ma per colonna. Questo data model può essere considerato come un'evoluzione del "key-value store", in quanto, i dati hanno comunque la rappresentazione tipica delle *hash table* ma in questo caso possono arrivare a due o più livelli di indicizzazione.

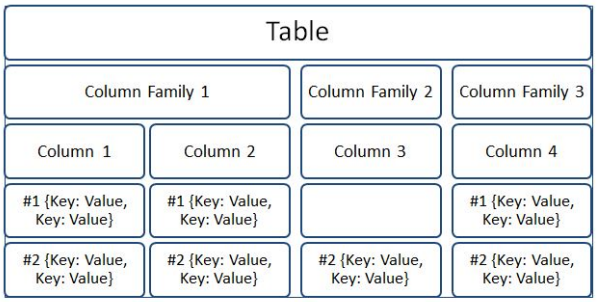


Figura 2.23: Struttura di un database column-oriented [49].

Le colonne vengono invece organizzate in gruppi detti *column-family*. Ognuno di essi oltre a raggruppare un insieme di dati che fanno parte dello stesso tipo di informazione consentono di ottimizzare l'organizzazione fisica dei dati. Memorizzare consecutivamente i valori contenuti in ogni colonna consente, per query che coinvolgono pochi attributi selezionati su un gran numero di record, di ottenere tempi di risposta migliori. É inoltre possibile applicare efficienti tecniche di compressione dei dati, in quanto ogni colonna è costituita da un unico tipo di dati riducendo lo spazio occupato.

Questo modello di dati quindi è molto utile nei casi in cui abbiamo a che fare con grandi quantità di scritture, un'elevata dinamicità dello schema e tabelle sparse che hanno valori nulli in alcune colonne.

Document database

I document database [48] hanno come unità primaria di memorizzazione documenti rappresentati tramite formati come JSON e XML. Il vantaggio è che questi documenti sono organizzati in maniera simile ad un oggetto dei più comuni linguaggi di programmazione cioè hanno dei campi con valori tipizzati.

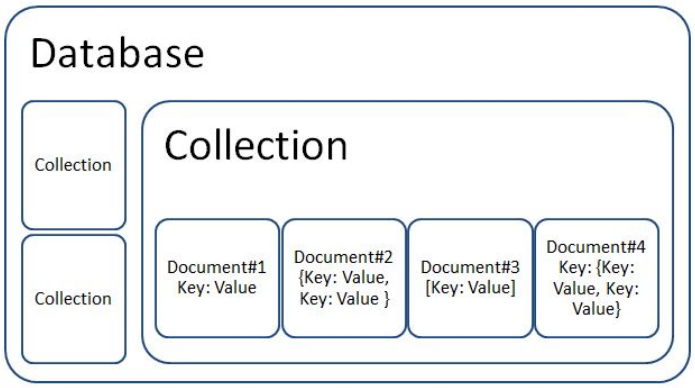


Figura 2.24: Struttura di un document database [49].

Una delle caratteristiche principali di questo tipo di data model è che i documenti possono essere indicizzati sia per singolo attributo che grazie alla creazione di indici composti permettendo così di fare ricerche relative ai valori dei documenti memorizzati.

Questi tipi di DBMS sono molto apprezzati per la grandissima flessibilità, che permette di ottenere modelli dei dati complessi senza penalizzare le performance. Per questo motivo sono molto utilizzati per la realizzazione di siti web, per l'e-commerce e appunto la gestione di documenti.

Graph database

I graph oriented database [52] utilizzano i concetti base delle strutture dati a grafo. Essenzialmente sono formati da nodi e archi per rappresentare e archiviare le informazioni.

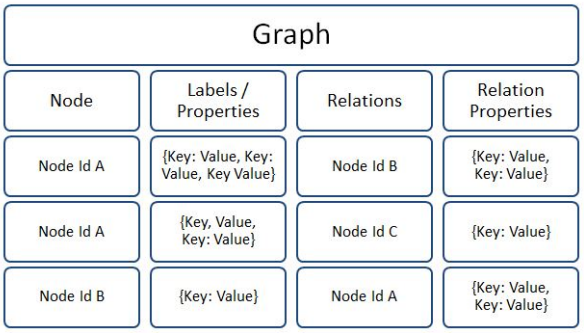


Figura 2.25: Struttura di un graph database [49].

Il caso d'uso più comune è quando esiste un legame stretto tra le entità che fanno parte delle tabelle. La caratteristica più importante di questi data-

base è che permettono di effettuare interrogazioni tramite cammini tra i nodi del grafo. Grazie ad algoritmi efficienti per la visita dei grafi ciclici anche strutture grandi con molti nodi e molti archi mantengono latenze basse sulle interrogazioni.

2.3.4 Apache HBase

Dopo la panoramica sul mondo NoSQL e sulle motivazioni che spingono la scelta del modello non relazionale rispetto ad uno relazionale viene descritto **Apache HBase**, uno tra i più diffusi DBMS NoSQL sul mercato open-source. Di questa tecnologia ne viene analizzata l'architettura, le primitive di accesso ai dati che si utilizzano per operare fisicamente sui dati ed alcune tecniche avanzate per la progettazione logica dello schema.

Storia

Negli anni Novanta Google ha cominciato a indicizzare il Web attraverso il suo motore di ricerca. Con il passare del tempo, vista la crescita esponenziale dell'utilizzo di questa tecnologia, si è scontrata con grandi difficoltà di gestione, conservazione e accesso ai dati. Questo ha portato alla creazione del *Google File System* (GFS), un semplice file system distribuito che si appoggia ad hardware di basso costo e apre la possibilità di scalare orizzontalmente. Nel 2006 è stato pubblicato "*Bigtable: A Distributed Storage System for Structured Data*" [53] dove viene spiegato come è possibile salvare i dati con alte performance in un sistema in grado di scalare e gestire grandi quantitativi di dati.

La community open-source Apache ha visto una grande opportunità in questo ambito applicativo e nel 2008 sono iniziati gli sviluppi di *Apache HBase*. É nato come l'implementazione di BigTable ed inizialmente faceva parte del progetto Hadoop.

Definizione

HBase [54] è un database NoSQL scritto in Java, fortemente distribuito e costruito sopra un file system che attiva la replicazione dei dati su più nodi. Il file system su cui si appoggia è HDFS, uno dei principali componenti di Hadoop che si occupa di far persistere i dati in maniera consistente su più macchine che sono in grado di comunicare attraverso la rete. HBase è progettato per essere tollerante agli errori ed è in grado di fornire accesso ai dati con latenze molto basse. É in grado di operare in maniera efficiente anche su tabelle con milioni/miliardi di record sparsi, fornendo tutti gli strumenti necessari per interrogarle in maniera pressoché real-time.

É un database della famiglia *column-oriented*. In questo tipo di database si avvantaggiano le letture e si limita la dimensione dei dati salvati per le tabelle sparse visto che alcune celle non sono valorizzate. Questo tipo di tabelle non hanno gli stessi dati per tutte le righe, i dati sono spesso disomogenei e la struttura logica varia nel tempo. Questo scenario, tipico dei database NoSQL, mette in difficoltà il modello relazionale.

HBase fornisce molteplici funzionalità come:

- Replicazione;
- Java e REST API;
- MapReduce sui dati del framework;
- Sharding automatico delle tabelle;
- Bilanciamento del carico;
- Compressione;
- Elaborazione server-side.

HBase è utilizzato [55] in produzione in larga scala con installazioni multi macchina in migliaia di aziende. Tra i principali esempi ci sono:

- *Facebook*: ha utilizzato HBase come infrastruttura per il suo sistema di messaggistica con oltre 350 milioni di utenti che si scambiavano 15 miliardi di messaggi al mese. Facebook lo utilizza anche attualmente per il conteggio dei like e i calcoli effettuati su quelle statistiche;
- *Twitter*: utilizza HBase come backup distribuito di tutte le tabelle relazionali nel suo backend. Dopo aver immagazzinato i dati vengono eseguite operazioni di MapReduce;
- *Yahoo!*: in questo caso viene utilizzato per analisi di documenti e ricerche con accessi casuali per avere feedback real-time di duplicazioni di file.

Modello dati

La struttura *column-oriented* può sembrare simile a quella di un database relazionale visto che è formata da righe e colonne però è molto differente come vengono organizzati e gestiti i dati. I database relazionali sono orientati alle righe invece in HBase l'unità base sono le colonne.

HBase è formato da tabelle e ognuna di esse può essere pensata come una grande mappa multidimensionale facile da partizionare e memorizzare distribuita su più macchine. Le colonne sono entità modulari dove non tutte le righe hanno un valore assegnato. Il modello rigido di un database tradizionale impone che per ogni colonna sia presente un valore tipizzato che, nel caso di assenza di informazione, corrisponde a null. Nel modello dati in questione si ha molta libertà sulle colonne: ogni valore inserito e passato al DBMS è un semplice array di byte. Il valore all'interno è il risultato della serializzazione del dato che si vuole inserire e, nel caso non si voglia passare informazione, è sufficiente omettere la colonna risparmiando così spazio di archiviazione alleggerendo le interrogazioni relative a quella colonna. I dati all'interno ad una colonna non hanno un vincolo sul tipo anche se è una buona pratica non valorizzare con tipi diversi la stessa colonna per evitare inconsistenze in fase di lettura. Questo

fa capire che l'unità base, ossia la colonna, è uno strumento molto libero che consente modellare logicamente e fisicamente ogni riga della tabella.

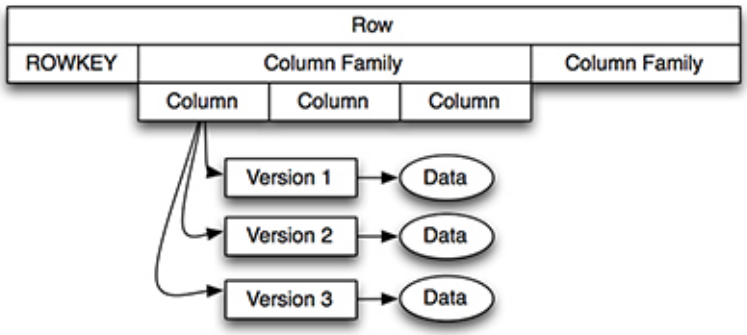


Figura 2.26: Modello dati HBase [56].

Come illustrato in figura 2.26 più colonne formano una *column family* che è un insieme di colonne. Questo raggruppamento logico oltre a separare i concetti garantisce che le colonne vengano salvate assieme anche su disco nell'unità base di memorizzazione chiamata *HFile*. Per migliorare le performance è quindi opportuno progettare le column family in modo da raggruppare i dati in base al metodo di accesso.

Durante le operazioni di lettura è possibile richiedere anche soltanto un sottoinsieme limitato di colonne. Se si accede spesso ad un sottoinsieme di colonne il modo più efficiente per farlo è creare una column family che le contenga e interrogare HBase richiedendo soltanto quella specifica column family.

É possibile accedere al valore di ogni colonna, detto cella, attraverso il suo nome formattato con *family:qualifier* dove il qualificatore indica un il nome che rappresenta la colonna all'interno della column family. Il sistema aggiunge alla cella un *timestamp* interno che indica quando è stato inserito quel valore. Una cella mantiene uno storico di tutti i valori che ha assunto con il rispettivo valore temporale. Questa strategia di versionamento consente di accedere anche ai precedenti valori senza dover inserire più righe permettendo comunque di accedere all'ultima versione del dato.

Una riga è formata da almeno una column familiy ed univocamente identificata dalla chiave di riga. É salvata come un array di byte ed è possibile accedervi velocemente perché sono ordinate in maniera lessicografica. Avere un buon design della chiave è fondamentale per mantenere elevate le prestazioni e mantenere contigue le righe relazionate tra di loro.

Denormalizzazione Uno dei concetti chiave durante la progettazione delle tabelle su HBase è la denormalizzazione [46, 57]. Questo è il processo inverso

della *normalizzazione* dei database relazionali. Il principio fondamentale della normalizzazione è la divisione dei concetti in tabelle separate. Ogni tabella è strutturata in modo da non ripetere i dati logicamente e quindi anche fisicamente. Il vantaggio è quello di evitare di aggiornare tutte le copie del dato ogni volta che deve essere modificato. L'assenza di duplicazione inoltre è utile per limitare lo spazio occupato dalla stessa unità informativa risparmiando memoria fisica.

La denormalizzazione segue il processo opposto ossia incentiva la duplicazione dei dati su varie tabelle per offrire migliori performance durante le letture. Il fattore dello spazio non è rilevante visto che grazie alle tecniche di distribuzione dei dati è possibile conservarli su più macchine. Il vero impatto sulle prestazioni è dovuto alla limitazione delle aggregazioni sui dati. Avere più dati sulla stessa tabella limita le letture ed evita di fare JOIN su altre tabelle per ottenerle.

Row-key design Durante la progettazione di una tabella il design della chiave [58] è forse il più importante fattore da prendere in considerazione. La chiave determina le performance di accesso ai dati visto che è l'unico indice nativo della tabella. Questo indice consente di effettuare *scan* su milioni di record in pochi millisecondi. L'efficienza nasce dal fatto che la chiave è un array di byte che può essere formato anche da più dati composti. Questo permette di avere un indice più ricco di informazioni che è in grado di caratterizzare una relazione all'interno della riga.

Un cattivo design della chiave può dare vita ad uno sbilanciamento del carico di lavoro del cluster. All'aumentare della dimensione le tabelle vengono partizionate e distribuite su più macchine in base al valore nella chiave. Avere una chiave che non può essere equamente distribuita aumenta il carico su un numero ristretto di nodi portando al rallentamento delle letture. Questo fenomeno viene chiamato *hotspotting* ed è frequente quando si ha una chiave che concentra i valori solo su una partizione.

I metodi più utilizzati per una corretta progettazione delle chiavi sono:

- *Salting*: consiste nell'aggiungere un "sale" cioè un prefisso arbitrario alla chiave per poter equamente distribuire i valori;
- *Hashing*: applicando un algoritmo di hashing si è in grado di ottenere il risultato bilanciato;
- *Timestamp invertito*: alla fine della chiave viene aggiunto un valore derivante dal risultato della sottrazione tra un numero arbitrario e il timestamp corrente.

Processo di denormalizzazione Prendiamo un esempio in cui è necessario trasformare uno schema relazionale in forma normale in uno schema deormalizzato e facilmente interrogabile da HBase.

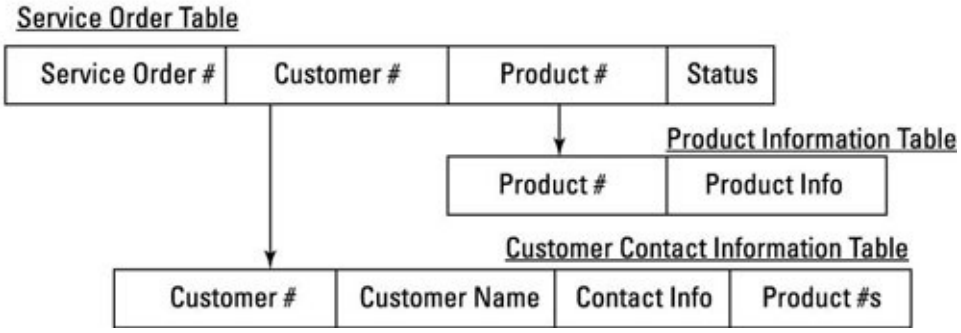


Figura 2.27: Schema normalizzato di un ordine effettuato da un cliente [57].

Nel caso in questione (figura 2.27) abbiamo una tabella di associazione che mette in relazione un cliente con il prodotto che ha acquistato. Questa tabella è identificata dal numero dell'ordine e ha due colonne che si riferiscono al cliente ed al prodotto. Con questa organizzazione non ci sono dati duplicati e sono sufficienti alcuni JOIN per ottenere risultati complessi sulle tabelle.

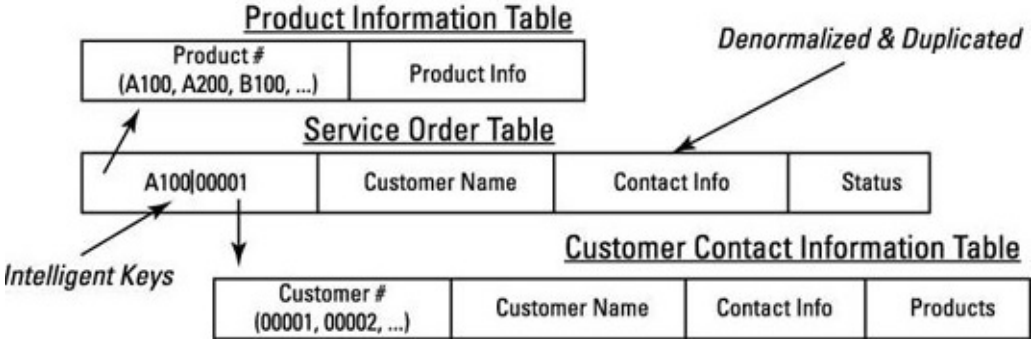


Figura 2.28: Schema denormalizzato [57].

Per prima cosa è necessario specificare la modalità con cui si accede alle informazioni. Se per esempio volessimo evitare di interrogare la tabella dei clienti continuamente con ripetuti JOIN la soluzione è unificare la tabella formando la chiave con la concatenazione tra quella del prodotto e quella del cliente. Una volta fatto ciò bisogna duplicare il nome e le informazioni del cliente in modo da non interrogare la relativa tabella ogni volta. In figura 2.28 vengono mostrati queste due operazioni molto semplici di denormalizzazione

che mostrano come il passaggio ad uno schema NoSQL richieda comunque una fase di analisi del piano di accesso e una progettazione dello schema.

Architettura fisica

HBase ha una architettura *master/slave* ed è composto fisicamente [59] da tre tipi di server differenti: HMaster Server, Region Server e Zookeeper. La figura sottostante spiega la gerarchia dei componenti mostrando come i region server siano la componente *slave* e l'HMaster rappresenti appunto il *master* dell'architettura.

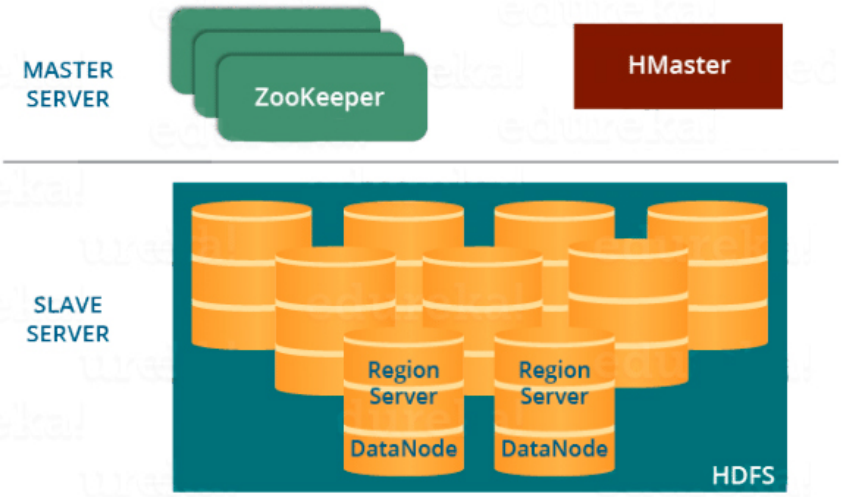


Figura 2.29: HMaser e i RegionServer [60].

In questa sottosezione verranno quindi spiegati i ruoli che svolgono questi componenti, le funzionalità offerte e come viene strutturato un cluster distribuito HBase.

Region Sono l'unità base della scalabilità e del bilanciamento del carico. Le region essenzialmente sono un range di righe contigue e ordinate in base alla chiave salvate fisicamente sulla stessa macchina. Le tabelle vengono divise orizzontalmente in base alla chiave quindi sono delimitate da una chiave di inizio e una chiave di fine e vengono gestite dinamicamente dal sistema. Inizialmente una tabella è composta da una sola region, quando con l'aumentare dei dati scritti supera il limite di spazio viene divisa in più region ed eventualmente ridistribuita nei vari region server disponibili. Quando invece è necessario ridurre il numero di file salvati tra due o più region viene effettuato il *merge* delle chiavi che quindi vengono unite fino a formare un'unica region. Questo

procedimento è dinamico ed avviene a seconda dei parametri di configurazione passati.

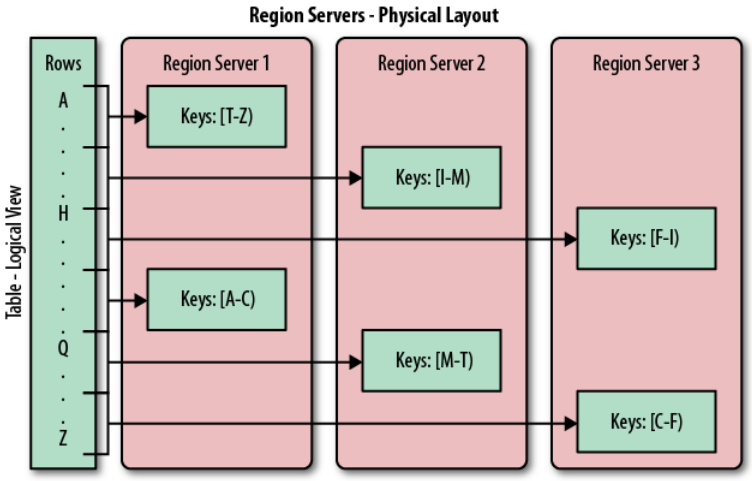


Figura 2.30: Righe raggruppate in diverse region e salvate su diversi region server [46].

Come si può vedere in figura 2.31 la tabella presa in considerazione viene divisa in sei region diverse ognuna delle quali viene salvata in uno dei tre region server presenti.

Region Server Sono macchine che contengono fisicamente le region e si appoggiano ad HDFS per la persistenza distribuita dei file su disco. Si occupano di accedere ai dati durante operazioni di lettura e scrittura. Il client quando si connette ad HBase per interrogare il database invece che connettersi al master per sapere quale region server contiene i dati comunica direttamente con il region server. Questa strategia di accesso evita che il master rappresenti un *single point of failure* del sistema e gli alleggerisce il carico di lavoro.

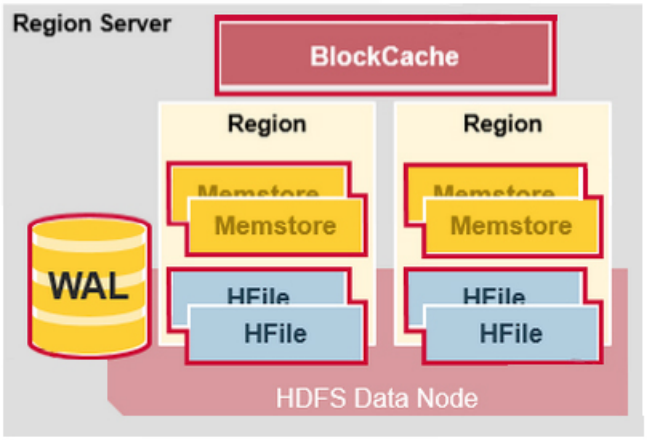


Figura 2.31: Componenti di un region server [59].

Un region server solitamente opera su dati salvati su un DataNode HDFS e come illustrato in figura è formato dai seguenti componenti [59]:

- *WAL* (Write Ahead Log): è un file presente in ogni region server dove vengono salvati i dati prima che vengano scritti permanentemente su disco o prima che venga effettuato un *commit*. Questa strategia è utilizzata per il *recovery* quindi il recupero di dati in caso di fallimento di un nodo;
- *BlockCache*: è una cache di lettura che risponde rapidamente alle richieste di lettura effettuate frequentemente. Agisce con strategia *LRU* (Last Recently Used) ossia una volta piena la cache al suo interno rimangono validi i record che sono stati utilizzati più recentemente;
- *Memstore*: è la cache di scrittura che salva i dati che non sono ancora stati scritti su disco. Esiste un memstore per ogni column family della region;
- *HFile*: questo è un file indicizzato ad albero salvato su HDFS. Quando il memstore non è più in grado di contenere altri record avvia una operazione di flush su disco che scrive tutti gli elementi su un nuovo file.

HMaster Questo servizio si occupa della gestione del cluster e delle operazioni amministrative. Da questo componente vengono eseguite operazioni DDL come creazione ed eliminazione di tabelle e gestisce l'assegnazione delle region ai region server bilanciando quindi il carico in modo da non avere macchine in sovraccarico di operazioni.

Più servizi HMaster possono essere avviati contemporaneamente anche se soltanto uno può essere operativo a tutti gli effetti. Gli altri HMaster vengono

definiti inattivi visto che non svolgono nessuna attività ma nel contempo si mettono in attesa che l'attuale master fallisca. Nel caso succeda il loro obiettivo è quello di acquisire il controllo il più in fretta possibile per evitare lunghi periodi in cui non è disponibile il master.

Zookeeper Ancora una volta è un componente fondamentale all'interno dell'architettura distribuita del framework visto che coordina tutto l'ambiente distribuito attraverso sessioni di comunicazione tra tutti i servizi. Ogni region server continuamente manda dei messaggi detti *heartbeat*. Questi segnali periodici sono utili per notificare la propria presenza nel cluster. Solitamente la ricezione dell'heartbeat indica che il server è in un buono stato di salute, è attivo ed è funzionante mentre nel caso per un certo periodo di tempo non arrivassero questi segnali il master verrebbe immediatamente notificato che uno dei region server non è più disponibile avviando così tutto il procedimento di recupero dei dati.

Un'altra funzionalità fondamentale di Zookeeper è quella di mantenere una vista aggiornata dell'organizzazione fisica del cluster. Nelle versioni precedenti (v0.96), HBase manteneva una tabella di sistema detta *Meta Table* (fisicamente chiamata .META) [61]. Attualmente all'interno del registro distribuito viene salvato un nodo `hbase:meta` con all'interno la lista di tutte le region e le loro informazioni. Ogni elemento della lista è identificato da: nome della tabella, region e id della chiave primaria da cui inizia la region. Quando un client si connette per la prima volta è in grado di individuare a partire dal dato che deve accedere in che region server si trova.

Accesso ai dati

In questa sottosezione viene spiegato come un client, grazie alle interfacce fornite dal framework, accede fisicamente ai dati. Viene descritto il flusso di operazioni che fanno in modo che il sistema sia tollerante agli errori, affidabile ed efficiente allo stesso tempo. Dopodiché vengono forniti alcuni semplici esempi che permettono tramite API di connettersi ed utilizzare i dati.

Se è la prima volta che si vuole leggere [59] o scrivere il client si connette a Zookeeper e ottiene il valore del nodo `hbase:meta` per sapere dove si trovano le region all'interno dei region server. Questi valori vengono salvati in una cache interna per evitare di interrogare nuovamente Zookeeper. Per le operazioni successive data la chiave che si vuole inserire o che si vuole leggere ci si connette direttamente con il region server di riferimento. Nel caso non sia più presente la relativa region viene invalidata la cache e viene richiesto nuovamente il nodo delle region a Zookeeper.

Nel caso di scrittura, per garantire consistenza e performance, vengono effettuati alcuni passaggi intermedi prima dell'effettiva scrittura su disco:

1. Viene effettuata la scrittura sul WAL che è salvato su HDFS. Questo file è utilizzato per recuperare i dati non ancora salvati su disco quando una macchina fallisce. Il file prodotto è un log in cui si scrive in maniera *append only*;
2. Una volta effettuata la scrittura il dato viene aggiunto al memstore. Fatto questo si può garantire al client che verrà scritto su disco inviandogli un ack che conferma che l'operazione è andata a buon fine;
3. Se il memstore ha accumulato abbastanza dati tutto il gruppo viene scritto su un nuovo HFile.

Nei seguenti paragrafi vengono brevemente spiegate le primitive di HBase per accedere ai dati tramite le API fornite. Tutti i qualificatori e i valori che si utilizzano devono essere passati come array di byte quindi viene spesso utilizzata la classe di utilità `Bytes` che rende semplici le conversioni.

Create Table e Delete Table Queste sono operazioni amministrative cioè consentono di gestire le tabelle all'interno del database. Per queste operazioni è necessaria la connessione con l'HMaster che si occupa di comunicare a Zookeeper e a tutti i region server la creazione o l'eliminazione di una tabella.

```
final byte[] TABLE_NAME = Bytes.toBytes("table-test");
final byte[] COLUMN_FAMILY_NAME = Bytes.toBytes("col-family");
final byte[] COLUMN_NAME = Bytes.toBytes("col-qualifier");

Configuration conf = HBaseConfiguration.create();
Connection conn = ConnectionFactory.createConnection(conf);
Admin admin = conn.getAdmin();

// Creazione di una tabella
HTableDescriptor descriptor = new HTableDescriptor(
    TableName.valueOf(TABLE_NAME)
);
descriptor.addFamily(new HColumnDescriptor(COLUMN_FAMILY_NAME));
admin.createTable(descriptor);

// Eliminazione di una tabella
admin.disableTable(TableName.valueOf(TABLE_NAME));
admin.deleteTable(TableName.valueOf(TABLE_NAME));
```

Dopo aver creato opportunamente un oggetto di configurazione viene richiesta l'interfaccia `Admin`. Con questa è possibile creare la tabella in questione che avrà almeno una column family e una colonna configurate. Dopo aver

creato la tabella sarà abilitata ed utilizzabile da parte degli altri client connessi. L'eliminazione invece avviene in due passaggi: prima viene disabilitata e successivamente eliminata a tutti gli effetti. Quando una tabella è disabilitata non è più possibile effettuarvi operazioni però sarà comunque visibile dagli altri client.

Put Questo tipo di operazione rappresenta una scrittura e può essere divisa in due modalità: inserimento di un record singolo e inserimento di un insieme di record.

```
Configuration conf = HBaseConfiguration.create();
Connection conn = ConnectionFactory.createConnection(conf);
Table table = conn.getTable(TableName.valueOf(TABLE_NAME));

// Creazione di un Put singolo
Put put = new Put(Bytes.toBytes("row-1"));
put.add(
    COLUMN_FAMILY_NAME,   // Nome della column family
    COLUMN_NAME,          // Nome della colonna
    Bytes.toBytes("value")  // Valore da inserire
);
table.put(put);

// Creazione di un batch Put
List<Put> puts = new ArrayList<>();
for (int i = 2; i < 10; i++) {
    Put currentPut = new Put(Bytes.toBytes("row-" + i));
    currentPut.add(
        COLUMN_FAMILY_NAME,
        COLUMN_NAME,
        Bytes.toBytes("value-" + i)
    );
    puts.add(currentPut);
}
table.put(puts)
```

In entrambi i casi viene creato un oggetto `Put` per ogni record da inserire e viene inizializzato il nome della column family e della colonna di riferimento. Internamente entrambi i metodi funzionano allo stesso modo e non c'è un effettivo vantaggio pratico nell'utilizzo della lista di put. Il vero vantaggio è dato dal parametro di configurazione `hbase.client.write.buffer` che rappresenta la dimensione del buffer mantenuto dal client prima dell'effettivo *flush* delle scritture al server.

Get Questo metodo consente di accedere ai dati in lettura ottenendo come risultato un solo elemento. Visto che la chiave è univoca è sufficiente

passarla come un array di byte per ottenere la riga cercata. Considerando che è indicizzata e lessicograficamente ordinata l'accesso alla tabella è molto efficiente.

```
Configuration conf = HBaseConfiguration.create();
Connection conn = ConnectionFactory.createConnection(conf);
Table table = conn.getTable(TableName.valueOf(TABLE_NAME));

// Creazione di un oggetto Get
Get get = new Get(Bytes.toBytes("row-1"));
get.addFamily(COLUMN_FAMILY_NAME);

// Utilizzo dell'oggetto Result
Result result = table.get(get);
String exampleResult = Bytes.toString(
    result.getValue(
        COLUMN_FAMILY_NAME,
        COLUMN_NAME
    )
);
System.out.println("Result: " + exampleResult);
```

Nell'oggetto `Get` è possibile limitare le colonne risultanti dall'interrogazione. In maniera semplice è possibile specificare quali column family e colonne sono necessarie nel `Result`. Le colonne che non vengono inserite non vengono proprio passate attraverso la rete dal server al client alleggerendo così la comunicazione.

Scan É un metodo per ottenere i dati simile a `Get` con la differenza che è possibile ottenere un risultato composto da diverse righe. Il risultato è un `ResultScanner` il cui corrispettivo nei database relazionali è un cursore che punta alla prima riga del risultato. I due principali modi per specificare la scan sono:

- *Range di chiavi*: è possibile indicare la chiave di inizio e la chiave di fine su cui eseguire lo scan sequenziale;
- *Filtro*: è possibile utilizzare un oggetto `Filter` che indica ad HBase come cercare i record da inserire nel risultato.

Se non viene specificato nessuno di questi due passando l'oggetto `Scan` vuoto verrà eseguito un FTS (*Full Table Scan*).

```
Configuration conf = HBaseConfiguration.create();
Connection conn = ConnectionFactory.createConnection(conf);
Table table = conn.getTable(TableName.valueOf(TABLE_NAME));

// Creazione di un oggetto Scan
Scan scan = new Scan();
```

```
scan.addColumn(
    COLUMN_FAMILY_NAME,
    COLUMN_NAME
);
// Creazione di un filtro Regex
Filter rowFilter = new RowFilter(
    CompareFilter.CompareOp.EQUAL,
    new RegexStringComparator("row-*")
);
scan.addFilter(rowFilter);

ResultScanner scanner = table.getScanner(scan);
for (Result result : scanner) {
    System.out.println(result);
}
```

In questo esempio viene creato un oggetto `Scan` che accede soltanto alla colonna interessata. Viene poi creata e passata alla scan un'istanza di un filtro che effettua pattern matching verso la chiave primaria indicando l'espressione regolare.

Esistono numerosi filtri nativi ed efficienti forniti da HBase però è possibile estendere la classe `FilterBase` per poter implementare strategie personalizzate per effettuare filtri più complessi.

Limiti di HBase

HBase non sostituisce completamente tradizionali database relazionali [62, 63]. Utilizzare questo database in certi casi può portare a performance peggiori e inconsistenze causando il degrado del sistema in cui viene utilizzato.

Bisogna prestare attenzione in fase di progettazione e, quando si decide di passare ad HBase, bisogna tenere in considerazione alcune limitazioni del framework dovute all'architettura. I principali punti critici analizzati sono:

- **Single point of failure**: il modello master/slave presenta sempre la fragilità del nodo master. In HBase visto che soltanto un HMaster può essere attivo rappresenta un collo di bottiglia sulle performance. In un cluster formato migliaia di region server c'è un carico eccessivo di operazioni che mettono in difficoltà il servizio;
- **Assenza di transazioni**: le operazioni non sono atomiche che potrebbero portare al fenomeno di *Dirty Read*;
- **Assenza di indici secondari**: nei database *column-oriented*, quindi anche in HBase, soltanto la chiave della riga viene indicizzata quindi non esistono chiavi secondarie;
- **Mancanza di un query language**: nativamente non è implementato nessun tipo di linguaggio per interrogare semplicemente i dati. Nei da-

tabase relazionali viene utilizzato l'SQL ed è possibile fare JOIN su più tabelle mentre su HBase i JOIN vengono fatti attraverso la funzionalità MapReduce di Hadoop.

Conclusioni

In questo capitolo si va a chiudere la tesi discutendo brevemente quello che è il risultato dell'analisi e le considerazioni sulle tecnologie affrontate.
L'elaborato si è concentrato esclusivamente sulle fasi dell'elaborazione dei dati in contesto distribuito. Con l'ampia crescita dei dati c'è stata un radicale cambiamento dell'approccio con cui vengono affrontati i limiti tecnici. Le richieste da parte dei servizi sono sempre più esigenti e necessitano alte prestazioni e deve essere garantita la continuità di servizio. Per anni le informazioni accumulate non acquistavano importanza per le aziende mentre ora, con il trend Big Data, diventano protagoniste in ambiti di Business Intelligence, Data Mining e Machine Learning. Come conseguenza della domanda di nuovi strumenti per la gestione dei dati sono nati numerosi framework open-source. Queste nuove tecnologie Big Data non sono state sviluppare per sostituire le tecniche fino ad oggi utilizzate, ma per affiancare gli strumenti già in produzione al fine di estrapolare valore da questa nuova tipologia di dati. Molte tecnologie fanno parte dell'Apache Software Foundation. Questa fondazione ha come principio alla base la collaborazione e contribuzione spontanea a progetti utilizzati in produzione in larga scala.

Nella tesi sono state prese in considerazioni tutte le fasi che attraversa il dato da quando viene generato a quando arriva all'utilizzatore finale sotto forma di informazione. Per ognuno di questi step intermedi è stata analizzata la motivazione che spinge ad affrontare il problema, sono stati illustrati i fattori che sono da tenere in considerazione e quali sono le tecnologie che risolvono questo problema. Nel panorama Big Data sono presenti centinaia di framework diversi che affrontano ognuno di questi problemi; è stato scelto quello che, secondo la mia valutazione, poteva essere un buon candidato in termini di performance, tolleranza agli errori e semplicità di architettura. I concetti alla base sono simili tra tutti i sistemi concorrenti però ognuno ha i suoi vantaggi e le sue limitazioni. Sono state analizzate infine, di ogni ambito, quali sono le limitazioni o è stata argomentata una valida alternativa.

Ognuno degli ambiti affrontati (Message Oriented Middleware, Stream Processing e Database NoSQL) rappresenta un ambito molto vasto. La letteratura a riguardo è molto ricca ma allo stesso tempo dispersiva. In questa tesi si è

voluto racchiudere in un'analisi comparativa tutto ciò che serve per riuscire a dare valore reale dal dato senza imbattersi in casi in cui non si è in grado di gestire così tanti dati. Operare con quantitativi elevati di carico su un sistema software mette in difficoltà i metodi tradizionali consolidate e incentiva l'utilizzo di nuove architetture, nuovi modelli e nuovi framework. Bisogna sempre tenere in considerazione che le stime attuali sulle performance e sui quantitativi di dati è destinato a variare nel tempo e con l'attuale crescita esponenziale, è necessario mettere in contro che bisogna essere pronti.

Sviluppi futuri

Nonostante sia stata svolta un'ampia analisi sulle tecnologie, la sperimentazione pratica è stata limitata ad installazioni e prove a singola macchina. Per capire come effettivamente operano i sistemi e i reali vantaggi che si ottengono dall'elaborazione distribuita sono necessarie installazioni e configurazioni molto più avanzate. Un aspetto non trattato in questa tesi è il *deploy* dei vari framework su server remoti quindi sarebbe interessante capire com'è il flusso di aggiornamento, manutenzione e orchestrazione dei servizi in maniera automatizzata.

Uno dei vantaggi di Kafka, Storm e HBase è la loro scalabilità, quindi sarebbe interessante effettuare test e benchmark di performance su installazioni multi macchina. Servizi come *Amazon AWS* e *Microsoft Azure* forniscono la possibilità avviare in modo semplice più macchine in grado di comunicare e avviare servizi distribuiti. Una sfida interessante sarebbe costruire una pipeline basata su *Lambda Architecture* costruita su un servizio cloud con installati strumenti di monitoraggio Web in grado di rappresentare lo stato del sistema in maniera visuale. Un ambito interessante che sta prendendo piede in ambito Big Data è il tracciamento comportamentale web, detto Web Tracking. Questa operazione richiede collezionamento, analisi ed elaborazione di dati derivanti dal comportamento di un individuo all'interno di un sito web, un e-commerce o una piattaforma. Questo ambito è molto stimolante e richiede competenze relative al mondo Big Data per poter gestire moli di dati così vaste. L'analisi di queste tecnologie e che parametri di confronto utilizzare mi ha aiutato a prendere coscienza in questo ambito informatico e mi ha fatto anche capire il reale impatto che ha anche sulle aziende che ne traggono benefici.

Ringraziamenti

Dopo anni di studio e impegno è arrivato finalmente il momento in cui si chiude il sipario finale. È stato un periodo di profondo apprendimento, non solo a livello scientifico, ma anche personale. Scrivere questa tesi ha avuto un forte impatto sul mio percorso di vita. A questo punto mi sembra il minimo spendere due parole di ringraziamento alle persone che mi hanno accompagnato in questo viaggio e mi hanno sostenuto nei momenti più complicati del percorso.
Ringrazio la mia famiglia, che ha creduto in me, che mi ha spinto a non lasciare mai la presa e che mi ha incentivato a intraprendere la carriera universitaria, il loro ruolo guida mi ha dato un supporto costante in quelli che sono stati i momenti felici e, allo stesso tempo, quelli che sono stati i momenti più duri.
Un ringraziamento va al mio relatore e i suoi preziosi consigli che mi hanno dato modo di portare a compimento la mia tesi in tutto il suo contenuto e la sua struttura.
Ringrazio chi tutti i giorni mi ha spinto a non mollare e in silenzio mi ha teso la mano. Infine un grazie speciale va a chi mi ha sostenuto in ogni occasione sin dal principio, chi mi ha dato forza e conforto nella buona e nella cattiva sorte.

Un sentito grazie a tutti,
Matteo.

Bibliografia

[1] Nathan Marz e James Warren. *Big data: principles and best practices of scalable real-time data systems*. Manning, 2015.

[2] Inc. Cyclone Interactive Multimedia Group e Inc Cyclone Interactive Multimedia Group. *The Digital Universe of Opportunities: Rich Data and the Increasing Value of the Internet of Things*. URL: https://www.emc.com/leadership/digital-universe/2014iview/executive-summary.htm.

[3] Alessandro Rezzani. *Big data: architettura, tecnologie e metodi per l'utilizzo di grandi basi di dati*. Apogeo, 2015.

[4] Mayer-Schonberger Viktor e Kenneth Cukier. *Big data: a revolution that will transform how we live, work and think*. John Murray, 2017.

[5] Paola Monti. *Analisi comparativa degli approcci Analytics nelle funzioni Aziendali*. 2016.

[6] Abhinav Korpal. *Scaling Horizontally and Vertically for Databases*. 2019. URL: https://medium.com/@abhinavkorpal/scaling-horizontally-and-vertically-for-databases-a2aef778610c.

[7] Sara Dotti. *Big Data: Nuove potenzialità per le aziende*. 2013.

[8] Michael Hausenblas e Nathan Bijnens. *Lambda Architecture*. URL: http://lambda-architecture.net/.

[9] *thoughts from the red planet*. URL: http://nathanmarz.com/blog/how-to-beat-the-cap-theorem.html.

[10] *CAP Theorem*. 2019. URL: http://www.gianlucatramontana.it/blog/2018/11/cap-theorem/.

[11] Jay Kreps. *Questioning the Lambda Architecture*. 2014. URL: https://www.oreilly.com/radar/questioning-the-lambda-architecture/.

[12] Martin Kleppmann. *Designing data-intensive applications: the big ideas behind reliable, scalable, and maintainable systems*. OReilly Media, 2017.

[13] Vaquarkhan. *Difference between scaling horizontally and vertically*. URL: https://github.com/vaquarkhan/vaquarkhan/wiki/Difference-between-scaling-horizontally-and-vertically.

[14] Andrew G. Psaltis. *Streaming data: understanding the real-time pipeline*. Manning, 2017.

[15] Sylvester Daniel. *Delivery Semantics - DZone Big Data*. 2019. URL: https://dzone.com/articles/kafka-producer-delivery-semantics.

[16] *Big Data Visualization*. URL: https://www.datamation.com/big-data/big-data-visualization.html.

[17] *Apache ZooKeeper*. URL: https://zookeeper.apache.org/.

[18] Neha Narkhede, Gwen Shapira e Todd Palino. *Kafka: the definitive guide: real-time data and stream processing at scale*. OReilly, 2017.

[19] *Fusion Middleware Concepts Guide*. 2010. URL: https://docs.oracle.com/cd/E21764_01/core.1111/e10103/intro.htm#ASCON113.

[20] Edward Curry. *Message-Oriented Middleware*.

[21] Raphael Thomas Seebacher. *Messaging challenges in a Global Distributed Network*. 2013.

[22] *What is Apache Kafka? - Definition from WhatIs.com*. URL: https://whatis.techtarget.com/definition/Apache-Kafka.

[23] *Apache Kafka*. URL: https://kafka.apache.org/.

[24] *Kafka Java Producer*. URL: https://docs.confluent.io/current/clients/producer.html.

[25] *Scaling out with Kafka Consumer Groups*. 2017. URL: https://simplydistributed.wordpress.com/2016/11/29/scaling-out-with-kafka-consumer-groups/.

[26] DataFlair Team. *Advantages and Disadvantages of Kafka*. 2019. URL: https://data-flair.training/blogs/advantages-and-disadvantages-of-kafka/.

[27] *Apache Pulsar*. URL: https://pulsar.apache.org/.

[28] *Comparing Pulsar and Kafka: unified queuing and streaming*. URL: https://streaml.io/blog/pulsar-streaming-queuing.

[29] William McKnight. *Benchmarking EnterpriseStreaming Data and MessageQueuing Platforms*. http://aiweb.techfak.uni-bielefeld.de/content/bworld-robot-control-software/.

[30] *Understanding Batch, Microbatch and Streaming*. URL: https://streaml.io/resources/tutorials/concepts/understanding-batch-microbatch-streaming.

[31] Srinath Perera. *A Gentle Introduction to Stream Processing*. 2019. URL: https://medium.com/stream-processing/what-is-stream-processing-1eadfca11b97.

[32] *Micro-Batch Processing vs Stream Processing*. URL: https://hazelcast.com/glossary/micro-batch-processing/.

[33] Fabian Hueske e Vasiliki Kalavri. *Stream processing with Apache Flink: fundamentals, implementation, and operation of streaming applications*. OReilly Media, 2019.

[34] Alessio Addimando. *Progettazione e prototipazione di un sistema di Data Stream Processing basato su Apache Storm*. 2016.

[35] Sean T. Allen, Matthew Jankowski e Peter Pathirana. *Storm applied: strategies for real-time event processing*. Manning, 2015.

[36] *Apache Hadoop*. URL: https://hadoop.apache.org/.

[37] *Hadoop Ecosystem Components and Its Architecture*. URL: https://www.dezyre.com/article/hadoop-ecosystem-components-and-its-architecture/114.

[38] Vivek HJ. *Apache Storm Topology: CoreJavaGuru*. URL: http://www.corejavaguru.com/bigdata/storm/storm-topology.

[39] Vivek HJ. *Apache Storm Stream Groupings*. URL: http://www.corejavaguru.com/bigdata/storm/stream-groupings.

[40] *Apache Spark™ - Unified Analytics Engine for Big Data*. URL: http://spark.apache.org/.

[41] Lorenzo Gatto. *Analisi e valutazione della piattaforma Spark*. 2014.

[42] Paolucci Christian. *Prototyping a scalable Aggregate Computing cluster with open-source solutions*. 2016.

[43] *Apache Flink: Stateful Computations over Data Streams*. URL: https://flink.apache.org/.

[44] *Getting started with Python and Apache Flink*. URL: https://www.kdnuggets.com/2015/11/getting-started-python-apache-flink.html.

[45] Matteo Berti. *Modelli di programmazione scalabile per Big Data: analisi comparativa e sperimentale*. 2017.

[46] Lars George. *HBase: the Definitive Guide*. OReilly, 2011.

[47] *ACID properties of transactions*. URL: https://www.ibm.com/support/knowledgecenter/en/SSGMCP_5.4.0/product-overview/acid.html.

[48] Francesco Romeo. *Analisi del fenomeno dei Big Data e comparazione di DBMS NoSQL a loro supporto*. 2013.

[49] Saravanan Subramanian. *A Primer on Open-Source NoSQL Databases*. 2018. URL: https://dzone.com/articles/a-primer-on-open-source-nosql-databases.

[50] Alice Gambella. *Analisi e sperimentazione del DBMS NoSQL MongoDB: il caso di studio della Social Business Intelligence*. 2013.

[51] Luca Merelli. *Analisi delle performance dei database non relazionali. Il caso di studio di MongoDB*. 2012.

[52] Ian Robinson. Jim Webber. Emil Eifrem. *Graph Databases, 2nd Edition*. 2015.

[53] *Bigtable:A Distributed Storage System for Structured Data*. 2006. URL: http://static.googleusercontent.com/media/research.google.com/en/us/archive/bigtable-osdi06.pdf.

[54] Jean-Marc Spaggiari e Kevin ODell. *Architecting HBase applications: a guidebook for successful development and design*. OReilly, 2016.

[55] Nishant Garg. *HBase Essentials*. 2014.

[56] *Introduction to HBase, the NoSQL Database for Hadoop*. URL: http://www.informit.com/articles/article.aspx?p=2253412.

[57] Dirk deRoos. *Transitioning from an RDBMS model to HBase*. URL: https://www.dummies.com/programming/big-data/hadoop/transitioning-from-an-rdbms-model-to-hbase/.

[58] V. Naresh Kumar e Prashant Shindgikar. *Modern big data processing with Hadoop: expert techniques for architecting end-to-end big data solutions to get valuable insights*. Packt Publishing, 2018.

[59] *An In-Depth Look at the HBase Architecture*. URL: https://mapr.com/blog/in-depth-look-hbase-architecture/.

[60] *HBase Architecture: HBase Data Model: HBase Read/Write*. 2019. URL: https://www.edureka.co/blog/hbase-architecture/.

[61] Apache HBase Team. URL: https://hbase.apache.org/book.html#arch.catalog.meta.

[62] *Apache HBase Do's and Don'ts*. 2019. URL: https://blog.cloudera.com/apache-hbase-dos-and-donts/.

[63] DataFlair Team. *HBase Pros and Cons: Problems with HBase*. 2018. URL: https://data-flair.training/blogs/hbase-pros-and-cons/.

[64] *Kafka Java Consumer*. URL: https://docs.confluent.io/current/clients/consumer.html.

[65] Hui Jiang et al. «Energy Big Data: A Survey». In: *IEEE Access* 4 (gen. 2016), pp. 3844–3861. DOI: 10.1109/ACCESS.2016.2580581.

[66] *Windowing data in Big Data Streams - Spark, Flink, Kafka, Akka*. URL: https://softwaremill.com/windowing-in-big-data-streams-spark-flink-kafka-akka/.

[67] Shilpi Saxena e Saurabh K. Gupta. *Practical real-time data processing and analytics: distributed computing and event processing using Apache Spark, Flink, Storm, and Kafka*. Packt Publishing, 2017.

[68] Chandan Prakash. *Spark Streaming vs Flink vs Storm vs Kafka Streams vs Samza*. 2018. URL: https://medium.com/@chandanbaranwal/spark-streaming-vs-flink-vs-storm-vs-kafka-streams-vs-samza-choose-your-stream-processing-91ea3f04675b.

[69] Marina Pernini. *NoSql Databases: Analisi prestazionale e confronto col modello relazionale. Un'applicazione: MongoDB*. 2014.

[70] Ruchir Choudhry. *HBase High Performance Cookbook*. Packt Publishing, 2017.