Kafka: streaming data

Indice

1. Motivazioni

Negli ultimi anni l'avvento delle architetture a microservizi ha portato la necessità di studiare nuove soluzioni al problema della gestione di molteplici fonti di dati.

In sistemi complessi formati da più microservizi tanti componenti interdipendenti comunicano tra loro scambiandosi dati e attingendo da numerose fonti di dati comuni come database, data warehouses oppure servizi esterni.

La necessità di filtrare, standardizzare e gestire molte fonti di dati aveva portato alla nascita del processo di **Extract, Transform, Load** (ETL) per l'estrazione, trasformazione e caricamento di dati in sistemi di sintesi come data warehouse o data mart, questo processo si sta però rivelando complicato ed impegnativo in un mondo dove la mole di dati prodotta dal logging di eventi critici ad un qualsiasi business è in continua crescita: semplici esempi sono la gestione degli eventi in un sistema **Internet of things** (IoT) oppure lo studio delle abitudini dei propri clienti per un servizio di e-commerce.

Lo stream processing tra microservizi propone un nuovo approccio per la gestione di questi problemi, fornendo una soluzione adatta alla gestione di dati in real-time altamente scalabile e ad alto throughput.

Apache Kafka è una piattaforma di streaming dati nata in un contesto aziendale importante che mira a rivoluzionare il modo con cui i microservizi di un business comunicano tra loro, favorendo un approccio improntato sulla gestione di eventi legati al comportamento dei dati, più che i dati in se.
Kafka nasce per sfruttare a pieno lo stream processing e favorire una gestione intelligente di grosse moli di dati, abbandonando il classico processo "batch" ETL per una soluzione, appunto, basata sullo streaming dei dati tra microservizi.

2. Introduzione

Prima di poter discutere della soluzione architetturale fornita da Apache Kafka e quali vantaggi propone rispetto alle soluzioni di batch ETL è necessario approfondire alcuni temi, tra cui il più importante è sicurante event sourcing (ES).

Il primo capitolo presenta una veloce descrizione di un processo ETL in modo da dare visione del oggetto di cui si vuole dibattere l'effetiva praticità.

Il secondo capitolo è utilizzato per illustrare uno dei concetti chiave della tesi: l'importanza di gestire e vedere le basi di dati come eventi, perchè utilizzare il concetto di eventi per modelizzare i dati ed infine, quale strumento è possibile utilizzarlo per farlo.

Il terzo capitolo è dedicato ad esaminare la piattaforma Apache Kafka, sia da un punto di vista tecnico-architetturale, esaminando le singole parti che compongono la piattaforma, sia l'ecosistema che si è venuto a creare intorno alla piattaforma, principalmente l'utilizzio delle librerie Kafka Connect, Kafka Streams e il recente sviluppo di KSQL, un linguaggio SQL-like per ricerche su stream di dati in tempo reale; Viene inoltre presentato come event sourcing si collega perfettamente a Kafka.

Infine vengono presentati degli esempi di utilizzo di Kafka e, nelle conclusioni, vengono messi a confronto i processi ETL e soluzioni di streaming come Kafka nel contesto di gestione di grosse moli di dati tra microservizi.

3. ETL

Un processo di Extract, Transform, Load (ETL) è un processo mirato alla trasformazione di dati contenuti su più database per ottenere un nuovo insieme di dati, filtrato e traformato secondo una particolare logica, destinato ad essere salvato in una data warehouse.

Verso la fine degli anni '70 molte aziende iniziarono ad utilizzare molteplici database per salvare e gestire informazioni, è proprio in questo contesto che nascono i processi di ETL: con l'avanzare del tempo è stato necessario studiare un metodo per l'aggregazione e gestione delle varie fonti di dati.

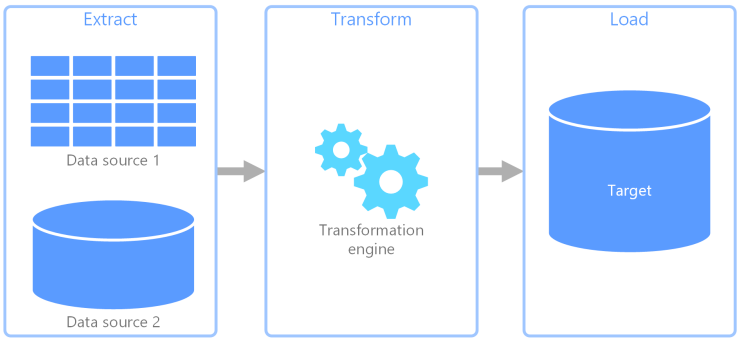

Figure 1: etl_process

Un processo di ETL si compone di tre parti:

La prima parte del processo di ETL è la fase di **Extract** ed involve l'estrazione dei dati da più data sources come database relazionali o non-relazionali, file JSON od XML ma anche risorse "adhoc" come, ad esempio, dei dati generati da programmi di web analytics.

L'obiettivo di questa fase è estrarre tutti i dati necessari dalle possibili sorgenti e prepararli alla fase di Transform.

Un importante problema legato a questa fase è il processo di **validazione delle sorgenti dati**: con più sorgenti dati spesso ci si ritrova a dover gestire più *formati* non necessariamente compatibili tra loro.
Per poter garantire alla fase di Transform dei dati comprensibili, durante la fase di Extract vengono definite delle *regole di validazione* per filtrare i dati provenienti dalle varie sorgenti, un esempio di regola di validazione è il controllo dei tipi di dati presenti nella fonte.

Nella fase di **Transform** una serie di regole e funzioni vengono applicate ai dati generati dalla fase di Extract per prepararli alla fase di Load nella data warehouse.

Il primo compito della fase di Transform è la **pulizia dei dati**: spesso le varie fonti di dati, nonostante siano state validate, possono presentare incongruenze tra loro come caratteri speciali legati all'encoding della propria sorgente oppure formati dei dati diversi ma compatibili (un esempio può essere la differneza di formattazione tra date americane ed europee).
Per garantire un corretto funzionamento delle operazioni di trasformazione è quindi necessario pulire i dati ed adattarli ad un formato comune.

Il secondo compito della fase di Transform è la **trasformazione dei dati** in nuovi dati richiesti dal business, esempi di trasformazioni sono:

- Joining di tabelle da più sorgenti
- Mapping e trasformazione di dati (esempio: "Maschio" in "M")
- Aggregazione di dati
- Generazione/calcolo di nuovi dati
- Selezione di insiemi di dati
- Validazione del nuovo formato di dati prodotto

Nella fase di **Load** l'insieme di dati generati dalla fase di Transform vengono inseriti in un target, il quale potrebbe essere una data warehouse ma anche più semplicemente un file in un formato utile.
Business diversi hanno necessità diverse, per questo l'implementazione della fase di load può avere più modalità implementative, il punto focale di questa fase è proprio stabilire la frequenza e le modalità di aggiornamento dei dati presenti nel target.
Decidere la frequenza (giornaliera, mensile, ecc.) e le modalità (sovrascrizione dei vecchi dati o meno) del target possono portare ad un processo di ETL più o meno utile ad una azienda.

Per generare un buon target è buona norma definire uno schema *preciso e chiaro* della tipologia di dati a cui il target deve aderire.
Come detto in precedenza un processo di ETL è utilizzato per aggregare più fonti di informazioni comuni ad un processo aziendale, questo suppone che le informazioni presenti nella data warehouse potrebbero venire usate da più parti di una azienda, le quali potrebbero essere abituate a particolari formati dei dati. Senza definire uno schema dei dati chiaro e preciso, si correrebbe il rischio di generare un insieme di dati inutilizzabile da determinati reparti in quanto non conforme al formato di dati da loro conosciuto.

4. Event sourcing

4.1. L'importanza dei dati e degli eventi

Lo status quo delle moderne applicazioni web è basato sul utilizzo di database per rappresentare le specifiche di dominio, spesso espresse da un cliente e/o da un esperto del dominio esterno all'ambiente di sviluppo.

Durante la fase di analisi dei requisiti (supponendo un modello di sviluppo del software agile) cliente e team di sviluppo si confrontano, cercando di trovare un linguaggio comune per definire la logica e l'utilizzo del software richiesto; Una volta stabiliti i requisti, il team di sviluppo generalmente inizia uno studio interno atto a produrre un **modello dei dati** che verrà usato come base per definire lo schema dei database utilizzati dal sistema.
Un cliente comune molto spesso non ha padronanza del concetto di 'stato di una applicazione', ma piuttosto si limita ad esporre i propri requisiti descrivendo i possibili **eventi** che, traslati sul modello di sviluppo incentrato su i database, portano il team di sviluppo a ragionare sui possibili stati di un database in risposta a questi eventi.

Lo stato di un database di una applicazione è strettamente legato all'insieme degli eventi del dominio applicativo; L'unico modo per modificare o interagire con questo database è tramite i comandi di inserimento, cancellazione o lettura, tutti comandi che vengono eseguiti solamente all'avvenire di un particolare evento.

Un database mantiene solo lo stato corrente di una applicazione; Non esiste il concetto di cronologia del database a meno di utilizzare soluzioni basate su **Change Data Capture** (CDC), generalmente utilizzate per generare un transactional log contenente tutte le operazioni eseguite sul suddetto database. In questo modello database-driven, un evento genera un cambiamento su una base di dati; Gli eventi e lo stato di un database sono però concetti diversi e slegati tra loro, l'esecuzione di un evento a volte può portare ad una asincronia tra l'esecuzione di un evento e lo stato di un database, tanto più se questo database è utilizzato da tutti i microservizi di una applicazione.

Una soluzione al problema di più microservizi che utilizzano lo stesso database è di utilizzare delle views del database locali ad ogni microservizio: ogni servizio lavorerà su una copia locale del database ed un job esterno si occuperà di compattare le views e mantenere il database aggiornato rispetto a tutti i cambiamenti.
Questa soluzione ha un enorme problema: supponiamo di notare un errore sul database e di doverlo correggere, come possiamo decidere quale delle views è "più corretta" delle altre? Per aiutarci nella ricerca dell'errore potremmo utilizzare il transactional log di ogni views, ma su database di grandezze importanti esaminare il log di ogni views che lo compone potrebbe essere un problema complesso e dispendioso in termini di tempo.

Event sourcing propone di risolvere questo genere di problemi allontanandosi da

una progettazione state-driven elevando gli eventi a elementi chiavi del modello dei dati di una applicazione.

4.2. Descrizione

Event sourcing (ES) è un design pattern che si contrappone ad una visione del mondo basata sullo stato di una applicazione fornendo come alternativa l'uso degli eventi, ovvero delle azioni o accadimenti che l'applicazione è in grado di riconoscere e gestire.

Durante l'analisi dei requisiti di una applicazione, spesso ci si trova a confronto con esperti di un dominio applicativo che non hanno particolare conoscenza delle tecnologie necessarie per implementare le loro richieste, è compito del programmatore (o del team di programmatore) analizzare le sue richieste e trasformarle in idee gestibili.
In genere questi esperti spiegheranno al programmatore le loro necessità illustrando il funzionamento del dominio utilizzando concetti molto più vicini a degli *eventi* piuttosto che *sequenze di richieste/risposte a/da un database*; Supponendo di dover sviluppare una piattaforma di e-commerce, è molto più probabile che l'esperto di dominio richieda di gestire eventi come "aggiungere un oggetto al carrello" oppure "comprare un oggetto" piuttosto che "creare dei database per gestire carrello, stock oggetti rimamenti, oggetti comprati".

La struttura dati fondamentale alla base di ES è l'**event store**, una tipologia di database ottimizzata per la gestione di eventi.
In un event store, gli eventi vengono inseriti in fondo alla struttura in ordine di avvenimento e non possono essere modificati o cancellati; Nel caso venga pubblicato per errore un evento sbagliato o inesatto per annularlo basterà pubblicare un evento contrario. Questo meccanismo garantisce che la ripetizione della storia degli eventi *porterà sempre allo stesso stato, errore compreso.*

Un event store è comunemente implementato utilizzando un **log**, una sequenza di record append-only e totalmente ordinata in base al tempo di scrittura del record.

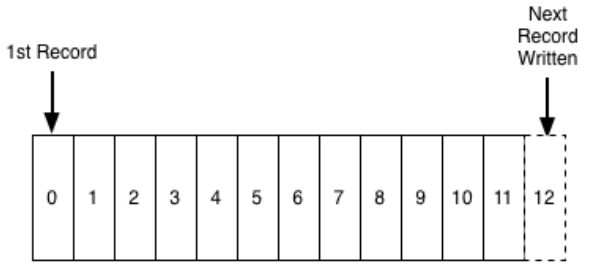

Figure 2: log_data_structure

I record sono inseriti in fondo al log e il processo di lettura è eseguito partendo dall'inizio del log.

Generalmente in un processo di sviluppo basato su ES, si tende a nominare gli eventi con il tempo passato per esplicitare il concetto che un evento è un avvenimento passato, un esempio nome per un evento potrebbe essere `item_created` oppure `item_bought`.

L'ordine di pubblicazione degli eventi è di estrema importanza in quanto è ciò che permette al pattern di rappresentare correttamente lo stato di una applicazione.

E' possibile vedere lo **stato corrente di una applicazione** come una **sequenza di operazioni di modifica dello stato eseguite partendo da uno stato iniziale**, questo implica che è possibile vedere un **evento** come il **delta tra lo stato iniziale di una applicazione e lo stato corrente dell'applicazione dopo l'esecuzione dell'evento**.
La possibilità di trasformare lo stato corrente di una applicazione in una funzione dello stato iniziale dell'applicazione e una sequenza di eventi è il meccanismo che permette ad event sourcing di avere una validità tecnica per la gestione dei dati di una applicazione.

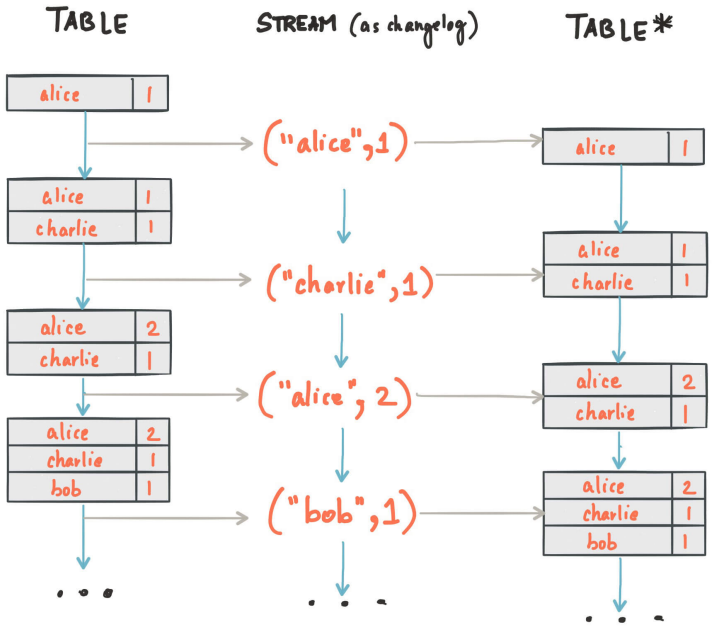

Figure 3: event_log_database_duality

4.3. Vantaggi

Event sourcing è un pattern estremamente utile per tutti quegli use-case dove è assolutamente necessario mantenere una storia dello sviluppo dello stato dell'applicazione del tempo. Tipici esempi sono gli strumenti di versioning del codice oppure la gestione di un storico bancario.

La capacità dell'event store di essere sia una struttura dati performante per la scrittura dei dati (è una semplice operazione di scrittura in fondo ad una sequenza di complessità O(1)) che una cronologia di tutti gli avvenimenti del sistema, permette una gestione degli errori estremamente semplice.

In un qualsiasi database relazionale se durante il normale utilizzo dell'applicazione avviene un errore logico che porta il database ad uno stato non corretto, è sempre necessario un rollback dell'intero database ad un data antecedente l'errore per poter sperare di correggiare l'errore.
Tale processo è dispendioso in termini di tempo e non è di facile esecuzione in quanto spesso il processo di backup di un database non viene eseguito dopo ogni inserimento o update di un record; Per poter correggere l'errore sarà quindi necessario calcolare partire dal backup più recente e applicare nuovamente tutte le transformazioni del database, meno l'errore.

L'analisi del motivo dell'errore può inoltre non essere di facile realizzazione con un database relazionale a meno che non siano in uso i meccanismi di CDC: senza una cronologia delle transazioni può essere molto difficile risalire al motivo dell'errore.

Diversamente nel caso dell'utilizzo di Event Sourcing, la gestione e l'analisi di un errore è estremamente semplice. Nel caso in cui l'errore, che sarà sempre un evento, non ha generato un effetto "domino" sul sistema (ovvero l'errore non ha portato all'esecuzione di uan catena di errori), una volta individuato è possibile pubblicare un evento "contrario" alla causa dell'errore in modo tale da cancellare l'apporto dell'errore sul sistema.
Nel caso contrario, ovvero il caso in cui l'errore ha generato una sequenza di errori, per ottenere lo stato corretto del sistema basterà ripetere l'esecuzione di tutti gli eventi del sistema escludendo quello che ha generato l'errore sulla base di dati.

E' bene notare che con ES la gestione degli errori nello stato del sistema è strettamente legata all'atomicità e definizione degli eventi del sistema: una corretta (semplice) definizione degli eventi del sistema porterà ad una cronologia del sistema più chiara e comprensibile.

4.4. Svantaggi

Event sourcing potrebbe non essere utile per una applicazione che richiede frequenti e continue query di richiesta sullo stato del sistema.
Come descritto in precedenza, per ottenere lo stato corrente del sistema è necessario eseguire tutti gli eventi pubblicati sull'event store partendo da uno stato iniziale; Se la nostra applicazione richiede di eseguire molte query di ricerca sullo stato corrente del database sarà quindi necessario calcolare lo stato del sistema *ogni volta che viene eseguita una nuova richiesta* (un esempio di richiesta sullo stato è la ricerca di tutti i record che presentano una particolare caratteristica).

Le modalità per risolvere questo problema sono determinate dal dominio e uso dell'applicazione che utlizza ES, ma generalmente per ovviare a questa debolezza vengono realizzati degli snapshot dello stato dell'applicazione da utilizzare per l'esecuzione delle query di ricerca. La frequenza di generazione ed aggiornamento di questi snapshot è strettamente legata al dominio applicativo dell'applicazione.

5. Apache Kafka e l'ecosistema

Publish/Subscribe è un pattern architetturale utilizzato per la comunicazione asincrona tra diversi processi od oggetti.

In questo schema mittenti e destinatari dialogano tra loro per mezzo di un *broker*, un processo incaricato, da una parte, di ricevere messaggi da dei mittenti e dall'altra di consegnare gli stessi messaggi a dei destinatari.

I destinatari non conoscono i mittenti, ed i mittenti non si interessano di chi sono i destinatari: l'unico compito del mittente è quello di pubblicare dei messaggi sul broker, starà poi al destinario il compito di abbonarsi (dall'inglese *subscribe*) al broker in modo da ricevere tutti i nuovi messaggi.

Questo pattern viene spesso utilizzato quando ci si trova ad avere più processi o servizi che generano delle metriche o dei dati, i quali sono di vitale importanza per altrettanti servizi; Una soluzione alternativa sarebbe creare dei canali dedicati tra produttori e consumatori ma questo non permetterebbe alla struttura di supportare un numero sempre più elevato di servizi od oggetti, ed in un mondo dove è sempre più frequente l'utilizzo di microservizi e il logging di eventi e dati (Big Data) porterebbe ad un debito tecnologico elevato e difficile da correggere.

E' in questo contesto che nasce Apache Kafka, una *streaming platform* basata su un append-only log utilizzato da dei *producer* per pubblicare dei messaggi utilizzati da dei *consumer*.
I messaggi pubblicati vengono persistiti nel tempo, sono leggibili deterministicamente da qualsiasi consumer ed distributi all'interno del sistema secondo particolari logiche in modo da garantire protezione da crash e scalabità del sistema.

Messaggi

L'unità dati fondamentali in Kafka è chiamata *messaggio* o *record*.

Ogni messaggio è suddiviso in *key* (opzionale) e *value* e possono essere di qualsiasi formato; Kafka non impone particolari standard riguardo i formati dei dati utilizzabili all'interno del sistema ma con lo scorrere del tempo *Avro* è diventato lo standard de facto.
Il campo `key`, quando definito, è un byte array utilizzato come metadata per garantire un particolare ordinamento all'interno di un *topic*, un altro elemento fondamentale dell'architettura.

Nonostante Kafka sia una streaming platform, la scrittura e propagazione dei messaggi all'interno della rete non avviene necessariamente per messaggio, invece, piccoli gruppi di messaggi diretti verso la stesso topic vengono raggruppati in *batches*.
La gestione dei messaggi in batch nasce per motivi di efficienza per bilanciare throughput e latenza: a fronte di una latenza più alta per la consegna di un batch, vengono sprecate meno le risorse del sistema che altrimenti si ritroverebbe costretto a gestire l'overhead di conoscegna di un batch per ogni singolo messaggio.

Topic e partizioni

Un *topic* è un elemento utilizzato in Kafka per categorizzare una collezione di messaggi, e consiste in un unico stream di dati.

Un topic è suddiviso in *partizioni*, append-only logs sui quali vengono persisti i messaggi generati dai *producers*.

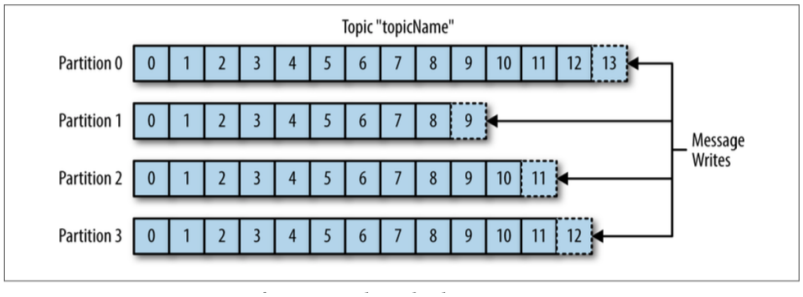

Figure 4: Un topic suddiviso in più partizioni

I messaggi sono inseriti in una partizione da un producer nell'ordine in cui sono stati inviati posizionandoli in fondo al log, non sono modificabili o cancellabili e sono contradistinti da un *offset*, un indice numerico che funziona da timestamp del messaggio. Un consumer legge i messaggi di un topic partendo dalla testa (o da uno specifico offset) del log proseguendo fino alla coda.

L'ordine di scrittura è garantito solo per ogni singola partizione: non è detto che messaggi appartenenti al medesimo topic siano in ordine cronologico se inseriti su partizioni diverse.

Per dare un esempio pratico di topic, supponiamo di utilizzare Kafka per creare uno storage di eventi ricevuti dal front-end di una applicazione: tipici eventi che vengono spesso loggati da un front-end possono essere *i link cliccati in una pagina*, *quali pagine sono state visualizzate in una sessione* oppure *se è stato visualizzato un particolare video embdeed*.
Per ogniuno di questi eventi verrà creato un singolo `topic` per raggruppare tutte le notifiche e dati generati da uno di quei particolari eventi a front-end: ad esempio avremo il topic `views-video-embdeed` sul quale verranno registrati dei semplici `yes` o `no` con magari l'aggiunta di un `timestamp` (l'ora di visualizzazione), in questo modo il topic ci permetterà di calcolare la frequenza di visualizzazione del video.

Producers e consumers

L'architettura offerta da Kafka è utilizzata da due genere di client: *producers* oppure *consumers*.

I *producers* hanno il compito di creare messaggi indirizzati a specifici topic indipendentemente dal numero di partizioni che formano il topic.
Come illustrato in precedenza, un topic è formato da un numero variabile di partizioni utilizzate come meccanismo di replica e gestione dei messaggi; Alla creazione di un messaggio è possibile indicare al producer su quale partizione andare a scrivere il record specificando l'identificativo di una partizione specifica.

Nella maggior parte dei casi d'uso di Kafka, il producer non si pone mai il problema di decidere su quale partizione andare a scrivere un particolare messaggio ma piuttosto vengono utilizzati dei meccanismi di load-balancing per spartire correttamente i messaggi su tutte le partizioni disponibili presenti nel topic.
Tipici esempi di algoritmi di load-balancing sono il calcolo della partizione in base ad una hash key derivata dall'offset del messaggio oppure utilizzando un algoritmo round robin, se necessario è presente la possibilità di specificare un *partioneer* creato su misura al caso d'uso.

I *consumers* leggono i messaggi pubblicati sui topic ai quali si sono iscritti.
I messaggi possono essere letti partendo dalla testa (o inizio) del topic oppure uno specifico *offset* fino ad arrivare alla coda (o fine).
Un *offset* è un identicativo numerico corrispondente ad una chiave per uno specifico messaggio del topic ed è compito del consumer di tener traccia degli offset di tutti i messaggi che lui stesso ha già letto.
L'offset di un messaggio è *specifico ad una specifica* ***partizione***.
La capacità di mantenere in memoria gli offset dei messaggi già letti garantisce al consumer la capacità di fermare, ed in un secondo momento reiniziare, il processo di lettura di un topic.

I consumers lavorano in *gruppi di consumers*: uno o più consumer lavorano per leggere un intero topic, con la proprietà che *consumers diversi non possono leggere dalla stessa partizione*.

Questa struttura porta ad un alto throughput in lettura di un topic permettendo uno sviluppo orizzontale del numero di consumers necessari per leggere un numero elevato di messaggi per partizione. Nel caso di un crash di uno dei consumer un consumer group è dotato di un meccanismo di load balancing che permetterà ad un altro consumer del gruppo di continuare a leggere i messaggi della partizione che stava venendo consumata.

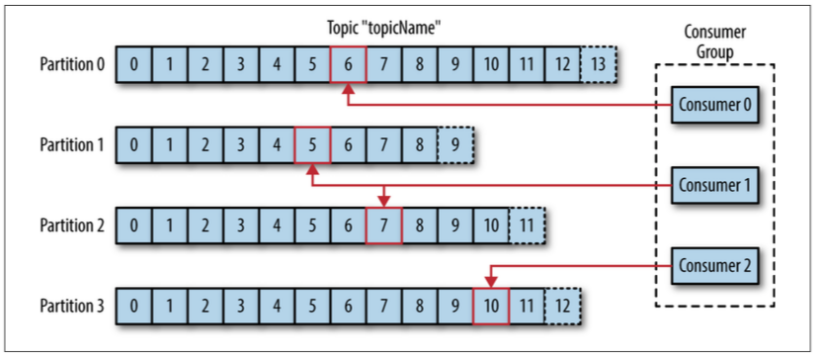

Figure 5: Esempio di un topic letto da un gruppo di consumers

3.1 Descrizione generica Kafka

Struttura di base: log => log compaction

Struttura architetturale: * brokers * clusters * topic

Come collegare event sourcing e kafka => perchè kafka è una buona piattaforma per event sourcing

3.2 Kafka Connect

Schema

Source connectors

Sink connectors

Community involment

3.3 Kafka Streams

cos'è streams

KSQL/LSQL

8. Esempi di utilizzo di Kafka

9. Bibliografia

https://engineering.linkedin.com/distributed-systems/log-what-every-software-engineer-should-know-about-real-time-datas-unifying https://www.confluent.io/blog/stream-data-platform-1/ https://martinfowler.com/eaaDev/EventSourcing.html
https://en.wikipedia.org/wiki/Event_store
https://www.confluent.io/blog/data-dichotomy-rethinking-the-way-we-treat-data-and-services/
https://www.confluent.io/blog/build-services-backbone-events/
https://www.confluent.io/blog/apache-kafka-for-service-architectures/
https://www.confluent.io/blog/messaging-single-source-truth/
https://www.confluent.io/blog/building-a-microservices-ecosystem-with-kafka-streams-and-ksql/
https://content.pivotal.io/blog/understanding-when-to-use-rabbitmq-or-apache-kafka
https://qconsf.com/sf2016/system/files/keynotes-slides/etl_is_dead_long-live_streams.pdf <= https://www.youtube.com/watch?v=I32hmY4diFY