

Università degli Studi di Padova

DIPARTIMENTO DI MATEMATICA “TULLIO LEVI-CIVITA”

CORSO DI LAUREA IN INFORMATICA



**Creazione di una Data Pipeline per il
trattamento dei dati con Apache Kafka e
Apache Druid**

Tesi di laurea

Relatore

Prof. Ombretta Gaggi

Laureando

Marco Brugin

Matricola: 2010012

ANNO ACCADEMICO 2022-2023

Leave a Little Sparkle wherever you go.

Dedicato a tutti coloro che ci sono stati vicini nei momenti più difficili.

Sommario

Il presente documento descrive il lavoro svolto durante il periodo di stage, della durata di circa trecento ore, dal laureando Marco Brugin presso l'azienda Sync Lab s.r.l.

Gli obiettivi da raggiungere sono stati: in primo luogo è stata richiesta la comprensione dei vantaggi e degli overhead portati da una architettura Event Driven, in secondo luogo è stata richiesta la comprensione e implementazione di una data pipeline con il trattamento dei dati tramite Apache Kafka e Apache Druid ed infine la comprensione e la gestione delle Time Series e dei Column-based Databases.

Il prototipo sviluppato presenterà una architettura distribuita, ad alta affidabilità, scalabile e resiliente, eseguibile tramite Docker Compose.

“Due cose sono infinite: l’universo e la stupidità umana, ma riguardo l’universo ho ancora dei dubbi.”

— Albert Einstein

Ringraziamenti

Innanzitutto, vorrei esprimere la mia gratitudine alla Prof. Ombretta Gaggi, relatrice della mia tesi, e al mio tutor aziendale Andrea per il sostegno e il supporto fornitomi durante lo svolgimento dello stage.

Desidero inoltre ringraziare con affetto i miei genitori e la mia famiglia per il sostegno e per essermi stati vicini in ogni momento durante questi lunghi anni segnati pure da una pandemia mondiale.

Infine desidero ringraziare tutti coloro che con la loro partecipazione hanno reso unico e inimitabile questo percorso.

Padova, 22 Settembre

Marco Brugin

Indice

1	Introduzione	1
1.1	Descrizione dell'azienda	1
1.2	Idea di fondo del progetto	2
1.2.1	Il ruolo dello stagista	2
1.3	Il progetto di stage	3
1.3.1	Descrizione del progetto	3
1.3.2	Obiettivi formativi	3
1.3.3	Risultati attesi e Obiettivi fissati	3
1.3.4	Analisi preventiva dei rischi	4
1.3.5	Obiettivi personali	5
2	Tecnologie e strumenti utilizzati	7
2.1	Linguaggi utilizzati	7
2.1.1	YAML	7
2.1.2	Python	8
2.2	Tecnologie utilizzate	8
2.2.1	Metodologia di sviluppo e strumenti di gestione di progetto . .	8
2.2.2	Ambiente di sviluppo	9
2.2.3	Versioning	10
2.2.4	Documentazione	12
2.2.5	Vincoli implementativi	12
3	Componenti di una Data Pipeline	14
3.1	Apache Kafka	14
3.1.1	Introduzione	14
3.1.2	Casi d'uso	15
3.1.3	Architettura e funzionamento	15
3.1.4	Garanzie di funzionamento	18
3.1.5	Il pattern Publisher-Subscriber	19
3.1.6	Alta affidabilità	20
3.1.7	Even Driven Architecture	20
3.2	Apache Druid	21
3.2.1	Casi d'uso	22
3.2.2	Architettura e funzionamento	22
3.2.3	Il modello dei dati	26
3.3	Streaming Data Pipelines	27
3.3.1	Introduzione	27
3.3.2	Approccio utilizzato	27

4	Il percorso di stage	29
4.1	Formazione	29
4.1.1	Daily stand-up meeting	31
4.2	Codifica	31
4.2.1	Configurazione di un cluster Kafka con Docker Compose	31
4.2.2	Configurazione del file di environment per il cluster di Apache Druid	32
4.2.3	Configurazione di un cluster di Apache Druid con Docker Compose	33
4.2.4	Creazione del produttore di eventi Kafka	34
4.3	Esecuzione e testing	35
4.3.1	Creazione di una Data Pipeline	35
4.3.2	Utilizzo delle tabelle di lookup in Apache Druid	41
5	Valutazioni e Conclusioni	43
5.1	Raggiungimento degli obiettivi	43
5.2	Attualizzazione dei rischi	43
5.3	Contenuti formativi acquisiti	43
5.4	Divario rispetto al percorso di studi	43
5.5	Valutazione personale	43
	Acronimi e abbreviazioni	45
	Glossario	46
	Bibliografia	50

Elenco delle figure

1.1	Logo dell'azienda Sync Lab s.r.l.	1
2.1	Logo di YAML	7
2.2	Logo di Python	8
2.3	Logo di ClickUp	9
2.4	Logo di Docker Compose	10
2.5	Logo di Git	10
2.6	Comandi di base di Git	11
2.7	Logo di GitHub	11
2.8	Logo di LaTeX	12
3.1	Logo di Apache Kafka	15
3.2	Architettura di Apache Kafka	16
3.3	Il pattern Publisher-Subscriber	19
3.4	Logo di Apache Druid	21
3.5	Architettura di Apache Druid	23
4.1	Board di ClickUp per il processo di formazione	29
4.2	Attività di hands-on per il processo di formazione	30
4.3	Board di ClickUp per il processo di coordinamento e documentazione	30
4.4	Injection del datasource all'interno di Apache Druid	36
4.5	Attivazione funzionalità di rollup all'interno di Apache Druid	38
4.6	Creazione della tabella di lookup all'interno di Apache Druid	41
4.7	Risultato della query eseguita con la funzionalità di lookup	42

Elenco delle tabelle

1.1	Tabella degli obiettivi	4
4.1	Risultati delle query eseguite su Apache Druid e PostgreSQL	37
4.2	Risultati delle query eseguite con, senza rollup e su PostgreSQL	40

Capitolo 1

Introduzione

1.1 Descrizione dell'azienda

Sync Lab s.r.l. è una azienda italiana attiva nell'ambito [Information and Communication Technology \(ICT\)](#), specializzata nello sviluppo e consulenza IT dal 2002 con sedi a Milano, Roma, Napoli, Verona, Como, con più di 300 dipendenti. È una azienda orientata verso la [Business Innovation](#), finalizzata alla creazione di soluzioni innovative che abbracciano i nuovi paradigmi della trasformazione digitale. Sync Lab possiede numerose certificazioni ISO LL-C per l'attestazione della qualità dei prodotti e servizi offerti. In particolare possiede le certificazioni ISO-9001 per la qualità dell'azienda, ISO-14001 per l'adozione quadro sistematico per l'integrazione delle pratiche a protezione dell'ambiente, ISO-27001 per la definizione di un [Sistema Gestione Sicurezza Informazioni \(SGSI\)](#), ISO-45001 per l'adozione di un [Occupational Health and Safety \(OH&S\)](#).

Attualmente Sync Lab lavora per più di 150 clienti diretti e finali, tra i più rilevanti ci sono nomi come: TIM, Trenitalia, PosteItaliane, UniCredit, ENI, ENEL, Vodafone, Fastweb.

Inoltre è un'azienda che si pone come obiettivo quello di essere un punto di riferimento per i propri clienti nella realizzazione di prodotti e soluzioni innovative per diversi settori di mercato, come: Sanità, Industria, Energia, Telco, Finanza e Trasporti & Logistica.

Lo spirito di Sync Lab è ampiamente rappresentato dal logo aziendale (Figura 1.1): un'onda che si propaga in modo circolare, che simboleggia la capacità di adattarsi e di evolversi in modo continuo.



Figura 1.1: Logo dell'azienda Sync Lab s.r.l.

1.2 Idea di fondo del progetto

Oggigiorno la gestione e l'analisi di grandi moli di dati in tempo reale sta diventando fondamentale per le aziende che vogliono rimanere competitive sul mercato.

Per questo motivo è necessario utilizzare tecnologie e software che permettano di analizzare e archiviare i dati in tempo reale in modo efficiente e veloce.

D'altra parte è necessario anche che tali tecnologie siano in grado di scalare in modo verticale e orizzontale in base al carico di lavoro da sostenere, mantenendo sempre alte prestazioni e resilienza in caso di guasti.

Per questo motivo Sync Lab ha deciso d'investire in un progetto di ricerca e sviluppo che ha come obiettivo quello di creare una [Data Pipeline](#) in grado di garantire le caratteristiche sopra descritte.

L'azienda ha già a disposizione un sistema di raccolta dati in real time, basato su Apache Kafka, che permette di ricevere dati da diversi sistemi e applicazioni per poi inviarli ad un sistema di archiviazione.

Il progetto prevede l'inserimento di un sistema di [Data Processing](#) basato su Apache Druid, che permetta di effettuare operazioni sui dati grezzi ricevuti da Apache Kafka, in modo da rendere più efficienti le successive operazioni di estrazione.

Particolarmente importante dovrà essere la fase di processing dei dati, in quanto dovrà permettere di eseguire operazioni di aggregazione e trasformazione dei dati in modo efficiente e veloce riducendo al minimo i tempi di latenza e mantenendo alte prestazioni.

1.2.1 Il ruolo dello stagista

Lo stagista ha un ruolo fondamentale in tale tipologia di progetto, infatti è colui che porta uno spirito d'innovazione e consolida il valore aggiunto aziendale.

Le attività che costituiscono il percorso che lo stagista ha intrapreso sono state elencate all'interno di un *Piano di lavoro*, concordato con il tutor aziendale, che ha lo scopo di guidare lo stagista durante il periodo di stage, permettere al tutor aziendale di monitorare l'andamento delle attività svolte e di valutare il raggiungimento degli obiettivi.

Inoltre al termine del periodo di stage, sotto la supervisione del tutor aziendale è stata svolta una presentazione rivolta a tutti *Stakeholder* aziendali, mirata a mostrare i risultati ottenuti con le tecnologie utilizzate e mettere in risalto le potenzialità del prototipo sviluppato.

Il progetto in sé fa parte di una rivoluzione tecnologica messa in atto da Sync Lab nel campo del [Data Processing](#) e [Data Analytics](#).

1.3 Il progetto di stage

1.3.1 Descrizione del progetto

Le attività descritte nel presente lavoro di stage illustrano la progettazione e lo sviluppo di un prototipo di una [Data Pipeline](#) eseguibile con Docker Compose, utilizzando Apache Druid come componente [Online Analytical Processing \(OLAP\)](#).

Il prototipo finale riceve i dati da un sistema di raccolta basato su Apache Kafka eseguendo il [Data Processing](#) con Apache Druid sui dati grezzi ricevuti.

L'obiettivo dello stage oltre a essere quello di sviluppare un prototipo funzionante, che soddisfi quanto richiesto, è anche quello di studiare e analizzare le funzionalità offerte dalle tecnologie utilizzate, in modo da poterne evidenziare i punti di forza e le differenze con le tecnologie tradizionali, utilizzate per il medesimo scopo.

1.3.2 Obiettivi formativi

In generale lo stage ha come obiettivo quello di far acquisire allo stagista concetti fondamentali riguardanti il contesto del prototipo sviluppato come:

- * Container technology;
- * Apache Kafka e le Event Driven Architecture, design pattern publisher/subscriber;
- * Column Based Database e la relazione/confronto con i classici Database relazionali;
- * [Middleware](#), [Data Pipeline](#), le architetture distribuite, scalabili e resilienti.

1.3.3 Risultati attesi e Obiettivi fissati

I risultati attesi e gli obiettivi fissati per lo stage sono riportati nella Tabella [1.1](#), con rispettivo identificativo, importanza e breve descrizione.

L'identificativo (riportato in breve con "ID") è la sigla che identifica ogni requisito e rispetta la seguente notazione **[Importanza][Identificativo]**.

L'importanza è indicata dalla sigla **O** oppure **F** ad indicare rispettivamente un obiettivo obbligatorio oppure facoltativo; mentre l'identificativo è un numero incrementale che segnala in modo univoco l'obiettivo o il risultato in esame.

Tabella 1.1: Tabella degli obiettivi

ID	Importanza	Descrizione
O1	Obbligatorio	comprensione e definizione di una piccola Data Pipeline che preveda il trattamento dei dati tramite Apache Kafka e Apache Druid
O2	Obbligatorio	comprensione dei vantaggi e degli overhead che le Event Driven Architecture portano con sé
O3	Obbligatorio	comprensione del pattern publisher/subscriber
O4	Obbligatorio	set-up di un cluster Apache Kafka in ambiente incapsulato in container
O5	Obbligatorio	gestione delle Time Series e dei Column-based Databases
O6	Obbligatorio	comprensione delle differenze tra i database relazionali classici e i Column-based Databases
O7	Obbligatorio	comprensione dell'impiego e utilità dei Middleware
F1	Facoltativo	produzione di documentazione e un pacchetto di configurazione dell'ambiente di sviluppo e esecuzione della Data Pipeline
F2	Facoltativo	produzione di documentazione che riporti le differenze di performance tra Apache Druid e altri database relazionali classici per alcune operazioni OLAP
F3	Facoltativo	realizzazione di una presentazione che illustri l'architettura sviluppata a personale di settore o Stakeholder

1.3.4 Analisi preventiva dei rischi

Durante la fase di analisi iniziale del progetto di stage, sono stati individuati i seguenti rischi, cui si è cercato di porre rimedio con le azioni di mitigazione indicate.

- 1. Inesperienza tecnologica:** il progetto prevede l'utilizzo di tecnologie con cui lo stagista non ha mai avuto a che fare.
Rischio: Medio.
Soluzione: Per mitigare tale rischio, è stato previsto un periodo di ambientamento e formazione sulle tecnologie coinvolte, in modo da poter affrontare il progetto con maggiore consapevolezza.
- 2. Scelte errate nella progettazione dell'architettura:** il progetto prevede la progettazione di un'architettura complessa, con molte componenti, di natura differente che interagiscono tra di loro.
Rischio: Alto.
Soluzione: Per mitigare tale rischio, è stato previsto un periodo di analisi e progettazione dell'architettura, con il supporto del tutor aziendale, in modo da poter ovviare tale rischio.
- 3. Prestazioni insufficienti delle macchine a disposizione:** il progetto prevede l'impiego di tecnologie che richiedono un elevato dispendio di risorse. Tale fattore se non tenuto in considerazione potrebbe portare a risultati penalizzanti.
Rischio: Alto.
Soluzione: Per mitigare tale rischio, è stato previsto una configurazione di tali macchine in modo da poter sfruttare al meglio le risorse a disposizione.

1.3.5 Obiettivi personali

Nonostante la realizzazione del progetto sia l'obiettivo principale, il percorso di stage offre anche la possibilità di raggiungere una serie di obiettivi personali come:

- * imparare a utilizzare nuove tecnologie e strumenti legati ad architetture distribuite;
- * comprendere i fattori da tenere in considerazione nella progettazione di un'architettura distribuita;
- * comprendere i vantaggi e come suddividere il lavoro tra i componenti, in modo da poter lavorare in parallelo;
- * imparare a lavorare in un team, condividendo le conoscenze e le esperienze;
- * confrontarsi con persone del settore, per capire come si lavora in un'azienda.

Capitolo 2

Tecnologie e strumenti utilizzati

Per il raggiungimento degli obiettivi del progetto di stage sono state utilizzate diverse tecnologie e strumenti. In questa sezione verranno riepilogate con una breve descrizione del loro utilizzo.

2.1 Linguaggi utilizzati

2.1.1 YAML

YAML, acronimo di **Y**AML **A**in't **M**arkup **L**anguage, è un linguaggio di Markup, noto per la sua leggibilità e la sua chiarezza espressiva.

La prima idea attorno al linguaggio YAML nasce attorno agli anni '90 quando Clark C. Evans, software developer, lo propone come alternativa a XML.

Nel 2001 Evans pubblica la prima specifica del linguaggio, che va a definire i principi fondamentali del linguaggio.

Negli anni YAML ha acquisito sempre più popolarità e interesse di utilizzo, in quanto ha offerto una configurazione semplice e leggibile per strumenti di DEVOPS, orchestrazione, automazione e molto altro (Figura 2.1).

La storia di YAML è strettamente legata alla esigenza di semplificare la rappresentazione di dati complessi, in un formato più comprensibile a un essere umano e a macchine.



Figura 2.1: Logo di YAML

2.1.2 Python

Python è un linguaggio di programmazione ad alto livello, orientato agli oggetti, che si distingue per la sua sintassi chiara e intuitiva (Figura 2.2).

Creato da Guido van Rossum e rilasciato per la prima volta nel 1991, è cresciuto fino a diventare uno dei linguaggi più utilizzati al mondo.

Data la sua semplicità e la sua versatilità, Python è utilizzato in diversi ambiti dallo sviluppo web, alla [Data Analytics](#), allo sviluppo di applicazione desktop e mobile, fino ad arrivare all'automazione e all'intelligenza artificiale.



Figura 2.2: Logo di Python

2.2 Tecnologie utilizzate

2.2.1 Metodologia di sviluppo e strumenti di gestione di progetto

Perseguendo la metodologia utilizzata da Sync Lab, il progetto di stage è stato sviluppato seguendo un approccio [agile](#), simil [Scrum](#) insieme a un [modello incrementale](#). Come risultato di tutto ciò, il carico di lavoro pianificato, suddiviso in task, è stato distribuito in più incrementi successivi, chiamati [sprint](#).

Come prima operazione sono state definite le attività da svolgere e inserite all'interno del [Product Backlog](#) e in seguito sono state pianificate all'interno di ogni [sprint](#).

L'adozione di tale metodologia di sviluppo, la si ritiene una scelta vincente, in quanto ha permesso di avere un'idea chiara delle attività da svolgere e ha reso possibile una stima accurata dei tempi di sviluppo. Inoltre ha permesso quanto prima di ottenere parti del prototipo funzionanti, che hanno consentito di avere un feedback immediato sul lavoro svolto da parte del tutor aziendale.

Per quanto riguarda il [modello incrementale](#), il maggiore vantaggio ottenuto è stato la metodologia di sviluppo: le componenti con maggiore priorità sono state sviluppate per prime, perchè hanno fornito la base su cui sviluppare le componenti successive. Ciò significa che le funzionalità essenziali del prototipo sono state disponibili sin da subito e sono state migliorate e ampliate con il progredire dello sviluppo del progetto.

ClickUp

ClickUp (Figura 2.3) è lo strumento di [project management](#) utilizzato per la gestione del progetto di stage.

È una piattaforma cloud che offre strumenti e funzionalità per la gestione di attività in modo efficiente.

Presenta una interfaccia intuitiva e semplice da utilizzare, che permette di gestire le attività in modo semplice e veloce.

Offre la possibilità di creare [board](#) personalizzate, in cui inserire le attività da svolgere, e di creare [task](#) personalizzati, permette di dare priorità alle attività, di assegnarle a un membro del team e d'impostare una data di scadenza.



Figura 2.3: Logo di ClickUp

2.2.2 Ambiente di sviluppo

Durante tutto lo sviluppo del progetto di stage ho fatto uso del sistema operativo **Ubuntu 22.04**. Tale scelta è stata dettata dal fatto che il progetto prevede la realizzazione di un ambiente incapsulato in [container](#), che verrà eseguito tramite [Docker](#) che sfrutta le funzionalità del kernel Linux.

L'utilizzo di un ambiente di questo tipologia rappresenta una svolta nell'approccio allo sviluppo e alla distribuzione del software, consentendo di risolvere sfide tradizionali legate alla compatibilità, alla portabilità e all'isolamento delle applicazioni.

Un [container](#) consente d'incapsulare un'applicazione, insieme a tutte le sue dipendenze e configurazioni, all'interno di un'unità standardizzata.

Tale approccio offre un ambiente isolato e autosufficiente in cui l'applicazione può essere eseguita in modo coerente, indipendentemente dall'ambiente in cui viene distribuita.

Inoltre un'applicazione contenuta in un [container](#) può essere eseguita su qualsiasi host o ambiente che supporti la tecnologia di containerizzazione, indipendentemente dal sistema operativo sottostante. Tutto ciò consente di eliminare il problema delle differenze tra ambienti di sviluppo, test e produzione, semplificando il processo di distribuzione.

Docker Compose

Docker Compose (Figura 2.4) è uno strumento che permette di definire e gestire applicazioni [Docker](#) multi-container.

Utilizza il linguaggio YAML per configurare i servizi dell'applicazione e fornisce un'interfaccia da riga di comando per la gestione dei [container](#).

Docker Compose permette di definire ed avviare più [container Docker](#) in modo coordinato, risolvendo la sfida dell'orchestrazione dei [container](#).

Mentre [Docker](#) permette di definire singoli [container](#), **Docker Compose** estende queste funzionalità permettendo agli sviluppatori di definire in modo dichiarativo, oltre ai servizi contenuti in ogni applicazione, anche le relazioni tra i [container](#) e le configurazioni di rete, volumi e variabili d'ambiente.



Figura 2.4: Logo di Docker Compose

2.2.3 Versioning

Git

Git è un sistema di controllo versione distribuito, utilizzato per il tracciamento delle modifiche ai file di un progetto.

Creato da Linus Torvalds nel 2005, GIT è stato pensato per la gestione del codice sorgente del *kernel* Linux, ma è stato adottato per progetti di ogni genere, di piccole e grandi dimensioni (Figura 2.5).



Figura 2.5: Logo di Git

È uno dei sistemi di controllo di versione più utilizzati al mondo, grazie alla sua velocità, alla sua efficienza e alla sua flessibilità.

Come tutti i sistema di controllo di versione si basa sul concetto di [repository](#), ovvero un archivio contenente i file e tutti i [metadati](#) relativi alle modifiche effettuate.

In **Git** un file può trovarsi in tre stati diversi: *committed* (versionati), *modified* (modificati) e *staged* (pronti per essere versionati).

Ogni nuovo modifica, se versionata all'interno del [repository](#) viene identificata da un *commit*, avente un identificativo univoco di 40 caratteri. *Modified* significa che il file è stato modificato ma non è ancora stato versionato, mentre *staged* significa che il file è stato modificato e preparato per essere inserito nel prossimo *commit*.

Quanto detto illustra le operazioni essenziali che possono essere effettuate con **Git** (Figura 2.6). Essenzialmente un workflow di base con **Git** prevede:

- **Clonare** un [repository](#), se già esistente;
- **Modificare** i file all'interno della [working directory](#);
- **Stage** dei file, ovvero prepararli per il prossimo *commit*, aggiungendoli alla *staging area* con il comando *git add*;
- **Commit** dei file, ovvero versionarli, con il comando *git commit*, i file così come son salvati nella *staging area* vengono versionati all'interno del [repository](#);
- **Push** delle modifiche sul [repository](#) remoto.



Figura 2.6: Comandi di base di Git

GitHub

Per quanto riguarda il servizio di hosting che ha ospita il [repository](#) remoto è stato utilizzato **GitHub**, andando a condividere i contenuti tra il mio account e quello del tutor aziendale (Figura [2.7](#)).



Figura 2.7: Logo di GitHub

GitHub è una piattaforma di hosting per progetti software, che utilizza **Git** come sistema di controllo di versione e contiene tutti i file e i [metadati](#) relativi alle modifiche validate lungo le fasi del progetto.

2.2.4 Documentazione

Per quanto riguarda la redazione della documentazione, Sync Lab non ha uno standard prefissato e mi ha permesso di scegliere quale software utilizzare per la produzione dei documenti. La scelta è ricaduta su **LaTeX**, un linguaggio di markup per la preparazione di testi.

LaTeX

LaTeX è un sistema di composizione tipografica ampiamente utilizzato per la creazione di documenti di alta qualità. A differenza dei tradizionali editor di testo, LaTeX si basa su comandi di formattazione e struttura, consentendo agli utenti di concentrarsi sul contenuto del documento anziché sul suo aspetto visivo.

È stato sviluppato da Leslie Lamport negli anni '80 come estensione di TeX, un linguaggio e motore di composizione sviluppati da Donald Knuth (Figura 2.8).

LaTeX semplifica notevolmente la creazione di documenti complessi, grazie alla sua capacità di gestire automaticamente numerazione delle sezioni, citazioni bibliografiche, tabelle dei contenuti e molte altre funzionalità tipografiche avanzate. L'ecosistema che **LaTeX** offre una vasta gamma di pacchetti e stili predefiniti che consentono di creare documenti sofisticati e professionali. Per quanto riguarda la scelta dell'editor da utilizzare l'azienda non ha dato vincoli rilevanti, quindi la scelta è ricaduta su **TexLive**, un distribuzione **LaTeX** per sistemi operativi Linux e su **Texworks** come editor di testo.



Figura 2.8: Logo di LaTeX

2.2.5 Vincoli implementativi

Per quanto riguarda l'implementazione del prototipo richiesto dall'azienda, non sono stati dati particolari vincoli implementativi, se non che il prodotto finale debba essere eseguibile con Docker Compose e che metta in evidenza l'architettura richiesta.

Capitolo 3

Componenti di una Data Pipeline

3.1 Apache Kafka

3.1.1 Introduzione

Apache Kafka è una piattaforma [open source](#), da Jay Kreps, Neha Narkhede e Jun Rao presso LinkedIn e successivamente donata alla [Apache Software Foundation](#) nel 2011. (Figura [3.1](#)).

Apache Kafka nasce con la necessità di LinkedIn di gestire grandi quantità di dati in tempo reale.

Già nel 2007 Jay Kreps e il suo team si resero conto che le soluzioni allora attuali, basate su database tradizionali, non erano in grado di gestire un carico di lavoro crescente e la complessità del formato dei dati generati da LinkedIn.

Dunque per affrontare tale sfida, nel 2010 LinkedIn iniziò a utilizzare **Apache Kafka** per gestire i dati di [log](#) generati dai vari servizi.

Tale adozione ha dimostrato nel tempo che **Apache Kafka** è in grado di gestire carichi di lavoro molto elevati, di scalare facilmente e di garantire un elevato livello di affidabilità nella consegna di messaggi.

Kafka è sviluppato in Java e Scala e rilasciata sotto licenza Apache 2.0. La versione attuale è la 3.5.1 rilasciata il 21 luglio 2023.

Kafka nasce originariamente come [message broker](#) e permette di gestire uno [streaming di eventi](#) in tempo reale.

In particolare fornisce funzionalità per:

- * pubblicare e sottoscrivere flussi di eventi, importandoli ed esportandoli da altri sistemi;
- * archiviare tali flussi in modo affidabile e duraturo;
- * elabora flussi di eventi in real time o in modo retrospettivo.



Figura 3.1: Logo di Apache Kafka

3.1.2 Casi d'uso

Apache Kafka viene ampiamente utilizzato in tutti quelli scenari in cui è richiesto la gestione affidabile di grandi quantità di dati in tempo reale.

I principali campi di utilizzo di **Apache Kafka** sono:

- * **messagistica:** **Apache Kafka** viene particolarmente utilizzato come [message broker](#), in applicazioni di messaggistica per disaccoppiare la produzione del messaggio dall'elaborazione dello stesso, **Kafka** rispetto ai tradizionali [message broker](#) offre velocità e [fault tolerance](#);
- * **elaborazione del flusso dati:** è possibile anche utilizzare **Kafka** come componente principale per creare [Data Pipeline](#) in cui i dati grezzi, provenienti da diverse sorgenti **Kafka** vengono aggregati, trasformati fino a ottenere un dato elaborato;
- * **monitoraggio e analisi:** **Kafka** può essere utilizzato per raccogliere dati di monitoraggio provenienti da applicazioni, sistemi di controllo o siti web;
- * **archiviazione dei dati:** **Apache Kafka** può essere utilizzato come sistema di archiviazione dei dati a lungo termine, permettendo così analisi storiche e ripristino di sistemi in caso di guasti;

3.1.3 Architettura e funzionamento

Apache Kafka nasce come sistema distribuito che opera su nodi, i quali comunicano tramite protocollo [Transmission Control Protocol \(TCP\)](#) ad alte prestazioni. Data la sua natura distribuita implementa funzionalità di [fault tolerance](#) con possibilità di rimpiazzo dei nodi che hanno avuto un malfunzionamento.

Kafka può essere distribuito e utilizzato in vari modi tra cui [virtual machine](#) e [container](#), [on-promise](#), o servizi cloud.

In generale **Apache Kafka** è costituito da due componenti essenziali: server e client.

Server

Kafka viene eseguito come un cluster di uno o più server, che rivestono diversi ruoli. Alcuni svolgono la funzione di **Kafka Broker**: ricevono i messaggi dai produttori, li archiviano e inviano i messaggi ai rispettivi consumatori, al momento della sottoscrizione.

Altri invece assolvono il compito di **Kafka Connect**: importano ed esportano i dati sotto forma di flussi dati, permettendo così d'interagire con altri sistemi esistenti.

Client

I **client** sono un insieme di librerie che consentono di scrivere applicazioni distribuite e microservizi che permettono d'interagire con il sistema di messaggistica di **Apache Kafka**, leggendo, scrivendo ed elaborando flussi di messaggi in parallelo, su larga scala e con **fault tolerance** anche in caso di problemi di rete o guasti della macchina.

In generale la scelta del client da utilizzare dipende dal linguaggio di programmazione che si vuole utilizzare per sviluppare l'applicazione.

I replicas

In **Apache Kafka** i **replicas** costituiscono una parte cruciale dell'architettura e permettono di disporre di più copie dei dati, distribuite su più **message broker**. Tale meccanismo permette di garantire la **fault tolerance** e la scalabilità del sistema.

Le repliche possono essere utilizzate a livello di partizione. **Kafka** ne designa una chiamata *Leader* mentre le altre sono partizioni *follower o in-sync*. Il numero totale di repliche incluso il leader costituisce **il fattore di replicazione**.

Il leader è responsabile della ricezione e dell'invio dei dati, per quella partizione, mentre i *follower* ricevono i dati dal leader e li replicano.

Produttori e consumatori

Il produttore **Kafka** invia i dati, con una richiesta di deposito, direttamente al **message broker**, *Leader* della partizione. Per velocizzare la ricerca del *Leader*, tutti i nodi possono rispondere alle richieste dei produttori fornendo **metadati** su quali nodi sono presenti i *Leader*, tale informazione consentirà al produttore d'indirizzare in modo appropriato i messaggi.

D'altra parte il consumatore emette una richiesta di recupero di messaggi al **message broker** che funge da *Leader* della partizione, specificando il suo offset nel registro dei messaggi.

Il **message broker** risponde con un messaggio contenente il suo offset nel registro con ogni richiesta e con una parte del registro a partire da quella posizione (Figura 3.2).

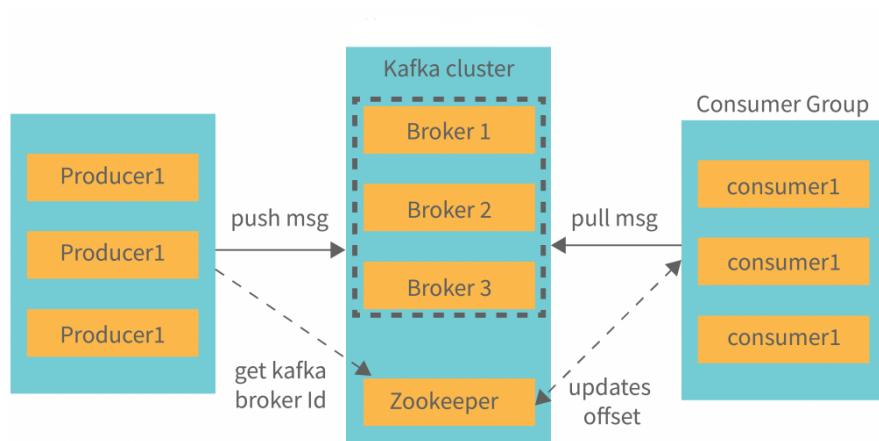


Figura 3.2: Architettura di Apache Kafka

La struttura dei messaggi

I messaggi inviati all'interno di **Apache Kafka** sono composti da una chiave, un valore e un timestamp.

Oltre a tali informazioni a ogni messaggio viene associato un **topic** o argomento che permette di eseguire operazioni di organizzazione e filtraggio dei messaggi.

Un messaggio, una volta inviato a un consumatore non viene eliminato dal **topic** ma viene mantenuto per un periodo di tempo configurabile attraverso un timeout.

I **topic** dei rispettivi messaggi vengono partizionati su più nodi per consentire a più consumatori di leggere gli stessi messaggi e permettere ai client di leggere e scrivere messaggi da/a molti **message broker**.

Quando un nuovo messaggio viene emesso, quest'ultimo si aggiunge alla rispettiva partizione relativa al **topic** e viene assegnato un numero di offset che identifica il messaggio all'interno della partizione.

Grazie a tale meccanismo **Apache Kafka** garantisce che i messaggi vengano letti nell'ordine in cui sono stati scritti.

Zookeeper

Zookeeper è un prodotto **open source** che si occupa della sincronizzazione tra i cluster distribuiti di sistemi come **Apache Kafka**.

È progettato per gestire operazioni di sincronizzazione, gestione dello stato e configurazioni nei sistemi distribuiti, garantendo coerenza e affidabilità.

Il suo funzionamento si basa sul protocollo Zookeeper Atomic Broadcast (ZAB), che è il cuore del servizio di coordinamento di **Zookeeper**.

Ogni nodo invia a intervalli regolari un messaggio *Keep-Alive* a **Zookeeper**, informandolo così che è vivo e funzionante. Se entro un tempo prestabilito il messaggio non viene ricevuto si presume che il nodo sia morto e se era un leader se ne elegge un altro. Inoltre **Zookeeper** permette di definire dei parametri che consentono di capire quando un nodo è guasto e quanto gli altri nodi, appartenenti a una partizione, sono in ritardo rispetto al *Leader* in termini di sincronizzazione sull'arrivo dei messaggi.

Il parametro *zookeeper.session.timeout.ms*, impostato a 6000 millisecondi per impostazione predefinita, indica che, se il leader non riceve l'evento *Keep-Alive* entro quel periodo temporale, determina che quel nodo sia guasto.

Il parametro *replica.lag.max.messages*, indica la massima differenza consentita tra **Replica's Offset** e **Leader's Offset**. Se tale differenza è maggiore di *replica.lag.max.messages-1*, il nodo viene considerato in ritardo e viene rimosso dall'elenco dei nodi in sincronizzazione dal leader.

Tutti i nodi che sono attivi e sincronizzati formano l' **In-Sync Replica Set (ISR)**.

Se tutti i nodi in sincronizzazione hanno memorizzato un messaggio nei rispettivi **log**, questo messaggio viene considerato confermato e quindi inviato ai consumatori.

In questo modo, un sistema come **Kafka** garantisce che un messaggio di cui è stato eseguito il *commit* non andrà perso, purché sia presente almeno una replica attiva e sincronizzata, in ogni momento.

Un nodo non sincronizzato può ricongiungersi all'**ISR** se può sincronizzarsi completamente di nuovo, anche se ha perso alcuni dati a causa del suo arresto anomalo.

Politiche di retention

Con **Retention** in **Kafka** si intende la possibilità di controllare la dimensione dei registri degli argomenti ed evitare di superare le dimensioni del disco esistente.

La conservazione può essere configurata o controllata in base alla dimensione dei [log](#) o in base alla durata configurata. Tale configurazione può essere impostata a grana fine o a grana grossa per ogni argomento o per tutti gli argomenti.

Retention basata sulle dimensione Una volta raggiunto il tempo di conservazione configurato per il segmento, quest'ultimo viene contrassegnato per l'eliminazione o la [compattazione](#) in base al criterio di pulizia configurato. Il periodo di conservazione predefinito per i segmenti è di 7 giorni. I parametri configurabili per la **Retention** basata sul tempo sono in ordine d'importanza e valutazione sono:

1. *log.retention.ms*;
2. *log.retention.minutes*;
3. *log.retention.hours*.

Nel momento in cui un parametro di livello di priorità superiore non è impostato si segue la politica di conservazione del parametro di livello di priorità inferiore.

Retention basata sulla dimensione In questo caso si configura la dimensione massima di una struttura di dati di registro per una partizione di argomento. Una volta che la dimensione del registro raggiunge questa dimensione, inizia a rimuovere i segmenti dalla sua fine.

I parametri configurabili per la **Retention** basata dimensione sono in ordine d'importanza e valutazione:

1. *log.segment.bytes*: la dimensione massima di un singolo file di registro;
2. *log.retention.check.interval.ms*: la frequenza in millisecondi con cui la pulizia del registro verifica se un registro è idoneo per l'eliminazione;
3. *log.segment.delete.delay.ms*: la quantità di tempo da attendere prima di eliminare un file dal file system.

3.1.4 Garanzie di funzionamento

In **Kafka** esistono produttori e consumatori che producono e sottoscrivono eventi. Gli uni, essendo in un ambiente distribuito, sono indipendenti l'uno dall'altro.

Apache Kafka in tale contesto può fornire una delle seguenti garanzie sulla consegna e ricezione dei messaggi:

- * **at most once**: i messaggi vengono consegnati al consumatore al più una volta. In questo caso, i messaggi possono essere persi, ma non duplicati;
- * **at least once**: i messaggi vengono consegnati al consumatore almeno una volta, i messaggi possono essere duplicati, ma non persi;
- * **exactly once**: i messaggi vengono consegnati al consumatore esattamente una volta. In questo caso, i messaggi non vengono né persi né duplicati, è la garanzia più costosa ma maggiormente richiesta.

3.1.5 Il pattern Publisher-Subscriber

Apache Kafka, nell'elaborazione dei messaggi, implementa il pattern **Publisher-Subscriber** (Figura 3.3) che permette di gestire flussi di dati in tempo reale.

Il pattern architetturale **Publisher-Subscriber** è un modello di progettazione software, utilizzato nei sistemi distribuiti, che impiegano una comunicazione asincrona tra i vari componenti.

Sebbene vada ad utilizzare tecniche già preesistenti come la sottoscrizione e l'accodamento di messaggi, la chiave di successo di tale pattern è il totale disaccoppiamento delle componenti: i componenti non sono a conoscenza dell'identità e della presenza degli altri.

Tale modello è nato dalla necessità del rendere i sistemi ridimensionabili in modo dinamico. Per raggiungere tale obiettivo, lo scambio di messaggi tra i due agenti in gioco viene gestito da un intermediario, chiamato **broker**, che si occupa di ricevere i messaggi dai **publisher** e di inoltrarli ai **subscriber**.

Vantaggi

- **Debole accoppiamento tra le componenti:** rende il sistema più flessibile e modificabile dinamicamente;
- **Elevata scalabilità:** non esiste limite al numero di **publisher** e **subscriber** che possano comunicare;
- **Utilizzo della comunicazione asincrona ad eventi:** non necessità della sincronizzazione degli attori coinvolti nella comunicazione;
- **Indipendente dal protocollo di comunicazione:** è integrabile con qualsiasi protocollo di comunicazione e stack tecnologico.



Figura 3.3: Il pattern Publisher-Subscriber

3.1.6 Alta affidabilità

Nel contesto si **Apache Kafka** con il termine di **alta affidabilità** ci si riferisce alla capacità di un sistema di gestire in modo robusto e coerente la ricezione, l'elaborazione e consegna dei messaggi anche in caso di guasti o malfunzionamenti.

L'alta affidabilità fornisce anche la garanzia che i flussi di messaggi non vengano persi o corrotti.

Numero dispari di nodi

Apache Kafka per garantire l'alta affidabilità utilizza le *repliche*: copie dei dati distribuite su più nodi.

Infatti nel caso in cui ci trovi nella condizione di prendere decisioni di consenso, eleggere *Leader* o eseguire altre operazioni di coordinamento, **Kafka** richiede il consenso da parte della maggioranza dei nodi. Pertanto per raggiungerla sempre è buona pratica utilizzare un numero di nodi dispari (3,5,7,9,...), per evitare situazioni di stallo.

3.1.7 Even Driven Architecture

L'**Even Driven Architecture** è un pattern architetturale basato su eventi: degli agenti, che sono in grado di ricevere tali eventi, agiscono solo nel momento in cui questi ultimi si verificheranno.

Una architettura basata su eventi fornisce una serie di altri vantaggi basati sul disaccoppiamento tra il produttore e il consumatore dell'evento, i quali, nel momento dell'emissione di quest'ultimo non è necessario siano sincroni ma possono andare a sfruttare una comunicazione di tipo asincrona.

Apache Kafka e l'architettura EDA

L'emissione di un evento indica che qualcosa è accaduto e può essere visto come un agglomerato di dati atomico in grado di soddisfare l'evento stesso.

Di solito **Apache Kafka** viene descritto anche come una piattaforma di [streaming di eventi](#) gestiti come flusso continuo di dati.

Inoltre **Kafka** memorizza tali dati in modo duraturo per il successivo recupero, analisi o elaborazione in tempo reale.

D'altra parte per utilizzare **Kafka** in un sistema [Event Driven Architecture \(EDA\)](#) la chiave è andare a sfruttare il disaccoppiamento: invece di effettuare un polling continuo di verifica della presenza di nuovi dati, basterà ascoltare il verificarsi di un evento per agire di conseguenza.

Inoltre grazie all'approccio sviluppato da **Kafka** un evento una volta soddisfatto, non viene eliminato, ma conservato per un periodo di tempo predeterminato, pertanto un evento potrà essere letto da più consumatori e potrà essere utilizzato per soddisfare una varietà di richieste, fino alla scadenza del suo periodo di conservazione.

3.2 Apache Druid

Apache Druid (Figura 3.4) è uno strumento [open source](#) di analisi [OLAP](#) progettato per analizzare e gestire grandi moli di dati in tempo reale, in modo scalabile.

Il progetto su cui si basa **Apache Druid** è stato sviluppato a partire dal 2011 da Metamarkets, una società di analisi dei dati in tempo reale per il settore pubblicitario, da parte di Gian Merlino, Eric Tschetter e Fangjin Yang.

L'obiettivo iniziale era quello di creare un sistema in grado di analizzare, in tempo reale, grandi quantità di dati per fornire ai clienti informazioni sulle campagne pubblicitarie. Purtroppo le tecnologie allora esistenti non erano in grado di soddisfare le esigenze di Metamarkets, pertanto il team di sviluppo decise di creare un nuovo sistema che potesse soddisfare le loro esigenze, che ha preso il nome di **Druid**.

Druid nasce per gestire dati di tipo evento, come strumento di [log](#), dati di transazione e molti altri.

Nel 2012, Metamarkets ha rilasciato **Druid** come progetto [open source](#) su GitHub, permettendo così alla comunità di contribuire e sviluppare ulteriormente la piattaforma. Nel 2015, **Druid** è diventato un progetto [open source](#) ufficiale sotto l'autorità della [Apache Software Foundation](#).

Apache Druid è comunemente utilizzato come back-end per le GUI di applicazioni analitiche o per [Application Programming Interface \(API\)](#) altamente simultanee che richiedono aggregazioni veloci.

Druid è sviluppato in Java e Scala, rilasciato sotto licenza Apache 2.0.

La versione attuale è la 27.0.0 rilasciata il 10 agosto 2023.



Figura 3.4: Logo di Apache Druid

3.2.1 Casi d'uso

Apache Druid viene utilizzato in tutti quegli scenari in cui è necessario analizzare grandi quantità di dati in tempo reale, in modo scalabile, con [fault tolerance](#) e con la possibilità di effettuare query complesse ad alta efficienza.

I principali campi di utilizzo di **Apache Druid** sono:

- * **analisi dati in real-time e applicazioni dati:** **Apache Druid** viene ampiamente impiegato in sistemi di acquisizione dati in tempo reale, query rapide e tempo di attività. **Druid** viene utilizzato per alimentare GUI di applicazioni analitiche o per [API](#) simultanee che necessitano di aggregazioni veloci, i migliori risultati vengono ottenuti nell'analisi di dati di tipo evento;
- * **elaborazione di metriche:** **Apache Druid** viene spesso utilizzato per effettuare misurazioni sul coinvolgimento degli utenti e il monitoraggio dei dati di test, calcolando metriche, conteggi finalizzati a elaborare tendenze su grandi moli di dati;
- * **operazioni OLAP:** **Apache Druid** viene utilizzato per accelerare l'esecuzione di query su grandi moli di dati e potenziare le applicazioni; è progettato per un'elevata concorrenza e query in meno di un secondo, alimentando l'esplorazione interattiva dei dati attraverso un'interfaccia utente.

3.2.2 Architettura e funzionamento

Apache Druid include molteplici configurazioni sia su singolo nodo, che distribuito su un [cluster](#).

Le distribuzioni su singolo nodo sono ormai poco utilizzate, ma esistono altre configurazioni pensate per macchine con bassa disponibilità di CPU e memoria, come i [container](#).

Distribuzione su cluster

Apache Druid nasce come sistema distribuito (Figura 3.5), scalabile, tollerante ai guasti, compatibile con il cloud.

Inoltre una architettura di questo tipo un malfunzionamento di una componente non influisce immediatamente sulle altre.

In generale un [cluster](#) di **Apache Druid** ospita i seguenti server:

- * **i server principali:** sono responsabili della gestione dei [metadati](#) e delle esigenze di coordinamento del [cluster](#), possono essere collocati insieme sullo stesso server;
- * **i server dati:** finalizzati a gestire i dati effettivi all'interno [cluster](#), traggono grandi vantaggi da CPU, RAM e SSD;
- * **i server d'interrogazione:** chiamati anche **Druid Broker**, accettano le richieste e le distribuiscono al resto del cluster.

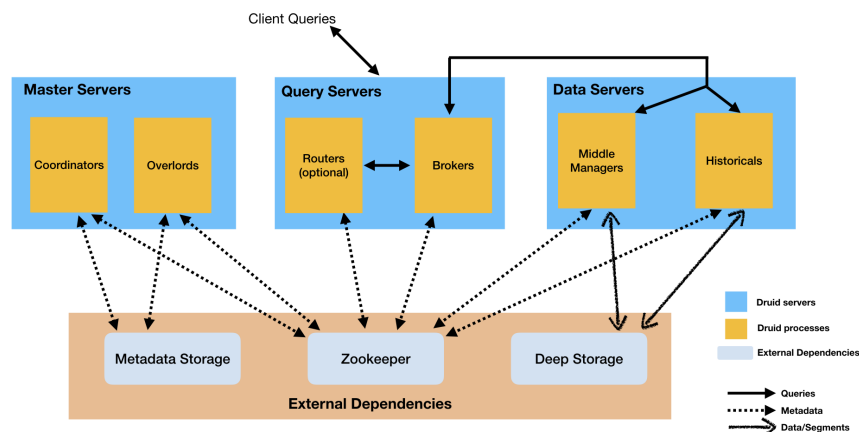


Figura 3.5: Architettura di Apache Druid

I **server principali** o **Master** gestiscono l'**injection** e la disponibilità dei dati: sono i responsabili dell'avvio di nuove operazioni di **injection** e del coordinamento della disponibilità dei dati sui **Data server**.

Ogni server mantiene una connessione con un **Zookeeper** per le informazioni sul **cluster** corrente. I server **Master** eseguono al loro interno i seguenti processi:

I **Coordinator** hanno il compito di gestire la disponibilità dei dati del **cluster**, comunicano agli **Historical** di caricare o rilasciare segmenti in base alle configurazioni, sono i responsabili del caricamento di nuovi segmenti, dell'eliminazione di segmenti obsoleti, della garanzia che i segmenti siano replicati (caricati su più nodi **Historical** diversi) un numero corretto di volte e sia presente un bilanciamento dei segmenti tra **Historical** per mantenere quest'ultimo caricato uniformemente. Il **Coordinator** ha anche una connessione a un database contenente l'elenco dei segmenti utilizzati.

Gli **Overlord** sono i responsabili dell'accettazione delle attività, del coordinamento della distribuzione delle attività, della restituzione degli stati ai chiamanti. Possono essere eseguiti sia in modalità locale che su server dedicato. Gli **Overlord** hanno anche il compito di creare i **Peon** per l'esecuzione delle attività.

I **Server dati** eseguono le attività d'acquisizione e archiviazione dei dati interrogabili. I server **dati** eseguono al loro interno i seguenti processi:

Gli **Historical** hanno il compito di copiare o estrarre i file di segmento dal **Deep Storage** al disco locale in un'area chiamata *segmentcache* e di rispondere alle query su tali segmenti. Il **Coordinator** controlla l'assegnazione dei segmenti agli **Historical** e il bilanciamento dei segmenti assegnati a questi ultimi.

Gli **Historical** non comunicano direttamente tra loro, ne comunicano direttamente con il **Coordinator**.

I MiddleManager hanno il compito di eseguire le attività di [injection](#) e di indicizzazione dei dati.

Sono i responsabili dell'inserimento di nuovi dati all'interno del [cluster](#), della lettura di dati da fonti esterne e della pubblicazione di nuovi segmenti.

I Peon sono i motori di esecuzione delle attività generate dai processi **MiddleManager**.

Ogni **Peon** viene eseguito su una JVM separata ed è responsabile dell'esecuzione di una singola attività. Tutti i Peon che sono stati generati da un processo **MiddleManager** vengono eseguiti sullo stesso nodo.

I Server d'interrogazione o Server query forniscono la funzionalità d'interazione con gli utenti, instradando le richieste ai **server dati** o ad altri **server query**, per la loro esecuzione.

I server query eseguono al loro interno i processi descritti di seguito.

I Broker sono i processi a cui instradare le query, analizzano i [metadati](#) forniti da **Zookeeper**, comprendono dove ricavare i dati necessari per elaborare i risultati delle query ed infine uniscono tutti i risultati elaborati per fornire un'unica risposta alla richiesta.

I Router sono i processi che hanno il compito di instradare le query a diversi processi **Broker**. Le query vengono instradate in base alle regole di caricamento dei segmenti che vengono applicate. Tale configurazione fornisce l'isolamento delle singole query, anche nel caso in cui si voglia applicare una priorità di esecuzione a determinate query.

Le dipendenze esterne Essendo **Apache Druid** un sistema distribuito e molto complesso, per il suo corretto funzionamento necessita di alcune dipendenze esterne.

Deep storage **Apache Druid** memorizza i suoi dati e indici in file di segmenti partizionati in base al tempo: crea un segmento per ogni intervallo di tempo che contiene dati.

Affinchè **Druid** funzioni bene con un carichi di query elevati, è importante che la dimensione del file del segmento rientri nell'intervallo consigliato di 300-700 MB. Se questo non accade è necessario valutare una modifica sulla configurazione di partizionamento dell'intervallo di tempo dei segmenti.

Il **deep storage** è il sistema nel quale vengono archiviati i segmenti.

Fino a che i processi di **Druid** possono vedere questa infrastruttura di archiviazione e accedere ai segmenti archiviati su di essa, non si perderanno i dati indipendentemente dal numero di nodi persi.

Ogni segmento viene creato da un **MiddleManager** come *mutable* e *uncommitted*.

I dati sono interrogabili non appena vengono aggiunti a un segmento senza *commit*. Periodicamente i segmenti vengono sottoposti a *commit* e pubblicati in **deep storage** diventano immutabili, passando dal **MiddleManager** agli **Historical**.

Inoltre nel **Metadata storage** viene scritta una voce al segmento che lo descrive. In generale si tratta di un sistema al di fuori di **Druid** può essere di due tipi:

- * **locale**: destinato a un solo server o a più server che hanno accesso ad un filesystem condiviso;
- * **su cloud**: è più conveniente, scalabile e robusto, **Druid** è compatibile con diversi sistemi cloud come: Azure, Amazon S3, Google.

Zookeeper **Apache Druid** utilizza **Zookeeper** per la gestione dello stato corrente del cluster.

In **Zookeeper** avvengono le seguenti attività:

- * elezione del servizio di coordinamento;
protocollo di pubblicazione del segmento degli storici;
- * protocollo di caricamento/rilascio del segmento tra **Coordinator** e **Historical**;
- * elezione del servizio di **Overlord**;
- * gestione delle attività di **Overlord** e **MiddleManager**;

Metadata storage è l'archivio dove vengono conservati tutti i **metadati** essenziali per il funzionamento di un **cluster** di **Apache Druid**, non viene utilizzato per archiviare i dati effettivi.

Nel caso di **Druid**, il **Metadata storage** viene spesso implementato utilizzando un database relazionale anche se non è l'unica soluzione possibile.

In generale contiene:

- * **record dei segmenti**: memorizza i **metadati** sui segmenti che dovrebbero essere disponibili nel sistema (chiamati anche segmenti usati);
- * **record delle regole**: memorizza le varie regole su dove allocare e deallocare i segmenti del cluster;
- * **record di configurazione**: memorizzare gli oggetti di configurazione di runtime, si tratta di un *feature* futura e avrà lo scopo di modificare alcuni parametri di configurazione nel **cluster** in fase di esecuzione.

In modalità predefinita **Apache Druid** utilizza **Derby**, ma database più adatti alla produzione sono **MySQL** e **PostgreSQL**.

3.2.3 Il modello dei dati

Apache Druid memorizza i dati sotto forma di [datasource](#) molto simili alle tabelle di un tradizionale database relazionale e alle serie temporali.

Il modello di dati di **Druid** presenta essenzialmente le seguenti componenti:

- **timestamp principale:** le [datasource](#) di **Apache Druid** devono sempre includere un *timestamp* primario, utilizzato per partizionare e ordinare i dati, per recuperare rapidamente i dati all'interno di un determinato intervallo di tempo;
- **dimensions:** sono colonne che **Druid** memorizza "così come sono"; possono utilizzare per qualsiasi scopo: filtrare, applicare aggregatori e così via;
- **metrics:** sono colonne che **Apache Druid** archivia in forma aggregata e diventano più utili se si utilizza la tecnica del rollup.

3.3 Streaming Data Pipelines

3.3.1 Introduzione

Con il termine di **Data Pipeline** si intende un software o un insieme di software che consentano il fluire automatico di dati da un punto ad un altro del sistema posto in essere. Di solito, nella definizione di un sistema di questo tipo, le origini sono molteplici e generano dati ad altissima velocità.

L'architettura che regge una **Data Pipeline** consente di consumare, elaborare, archiviare dati in tempo reale, man mano che vengono generati. Tale meccanismo consente di avere reazioni immediate ai dati processati.

Le tradizionali **Data Pipeline** sono state progettate per elaborare dati statici, infatti prima estraggono, trasformano e poi caricano i dati per poterli poi utilizzare.

In generale in sistemi come **Apache Kafka** le pipeline dati tradizionali non sono più utilizzabili, considerando l'enorme mole di dati che sono tenute a processare.

Per risolvere tale problema nelle Streaming **Data Pipeline** si è andati a creare due livelli per l'elaborazione dei dati:

- * **archiviazione**: livello che consente la memorizzazione dei dati, permettendo letture e scritture a basso costo in termini computazionali mantenendo l'ordine di arrivo dei dati;
- * **lavorazione**: livello che consente di elaborare e consumare i dati del livello di archiviazione andando a segnalare a quest'ultimo i dati non più necessari.

3.3.2 Approccio utilizzato

L'approccio comunemente utilizzato per la creazione di una Streaming **Data Pipeline** è quello di utilizzare un **message broker** come **Apache Kafka** e un sistema **OLAP** come **Apache Druid** per l'analisi dati, operando nel seguente modo:

- * **produzione dei dati**: all'interno dell'ambiente di sviluppo, i dati vengono prodotti da diverse fonti, sensori o applicazioni, e inviati al **message broker Apache Kafka** che ha il compito di archivarli e successivamente distribuirli ai consumatori;
- * **analisi dati**: i dati vengono letti dal **message broker Apache Kafka** e inviati al sistema **OLAP Apache Druid** che ha il compito di indicizzarli, elaborarli e renderli disponibili per eseguire query in tempo reale su di essi.

Capitolo 4

Il percorso di stage

4.1 Formazione

Il processo di formazione ha avuto un ruolo fondamentale nella buona riuscita del progetto di stage, con una durata complessiva di circa quattro settimane.

La causa del protrarsi del processo di formazione è stata provocata dalla mia inesperienza in merito a concetti legati all'architettura [EDA](#), **Apache Kafka**, **Apache Druid** e **Docker Compose**.

Tutto il processo di formazione è stato tracciato e monitorato da me medesimo e dal tutor aziendale attraverso le [board](#) offerte dal software di [project management](#) **ClickUp** (Figura 4.1).

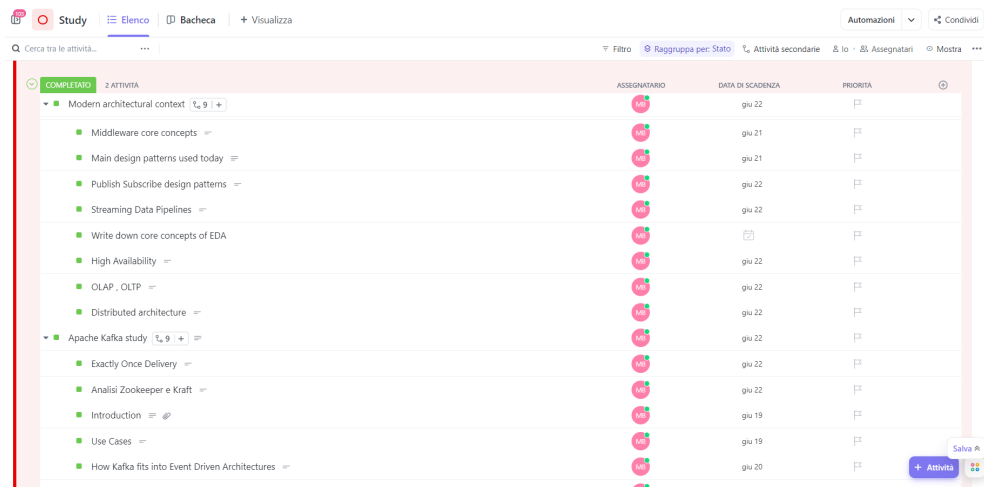
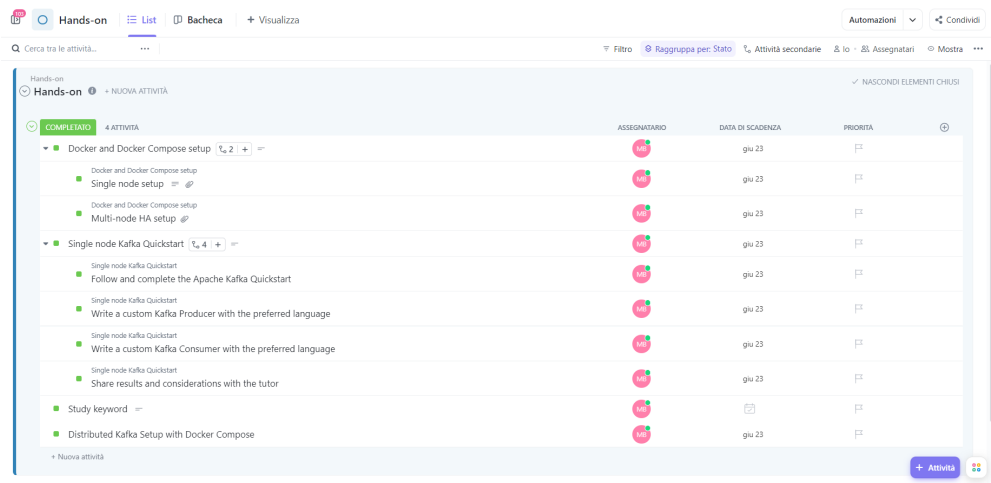


Figura 4.1: Board di ClickUp per il processo di formazione

Inoltre durante il processo di formazione, oltre a reperire informazioni da documentazione ufficiale, ho avuto anche modo di approfondire quanto appena appreso attraverso delle attività di [hands-on](#) che mi hanno permesso di mettere in pratica nell'immediato quanto appreso (Figura 4.2).

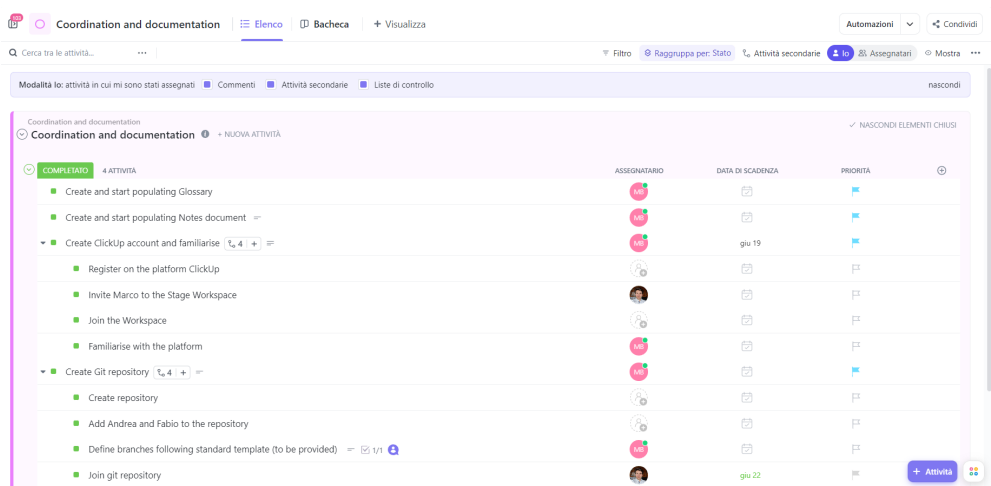


The screenshot shows a ClickUp board titled "Hands-on" with a sub-header "4 ATTIVITÀ". The board is organized into a list view. The first main activity is "Docker and Docker Compose setup" (2 items), followed by "Single node Kafka Quickstart" (4 items), and "Distributed Kafka Setup with Docker Compose" (1 item). Each item is a task with a green status icon, a due date of "giu 23", and a priority of "P3". The board includes a search bar, filters, and a "Raggruppa per: Stato" dropdown.

Attività	Assegnatario	Data di scadenza	Priorità
Docker and Docker Compose setup	[Avatar]	giu 23	P3
Docker and Docker Compose setup	[Avatar]	giu 23	P3
Single node Kafka Quickstart	[Avatar]	giu 23	P3
Single node Kafka Quickstart	[Avatar]	giu 23	P3
Single node Kafka Quickstart	[Avatar]	giu 23	P3
Single node Kafka Quickstart	[Avatar]	giu 23	P3
Distributed Kafka Setup with Docker Compose	[Avatar]	giu 23	P3

Figura 4.2: Attività di hands-on per il processo di formazione

Inoltre durante il processo di formazione, in collaborazione con il tutor aziendale, è stato definito anche un processo di coordinamento e produzione di documentazione che mi ha permesso, durante tutto lo svolgimento del percorso di stage, di avere un tracciamento dei concetti appresi, di avere un eventuale riferimento per un'eventuale risoluzione di problemi o dubbi sorti (Figura 4.3).



The screenshot shows a ClickUp board titled "Coordination and documentation" with a sub-header "4 ATTIVITÀ". The board is organized into a list view. The first main activity is "Create and start populating Glossary" (1 item), followed by "Create and start populating Notes document" (1 item), "Create ClickUp account and familiarise" (4 items), and "Create Git repository" (4 items). Each item is a task with a green status icon, a due date, and a priority. The board includes a search bar, filters, and a "Raggruppa per: Stato" dropdown.

Attività	Assegnatario	Data di scadenza	Priorità
Create and start populating Glossary	[Avatar]	[Icona]	P3
Create and start populating Notes document	[Avatar]	[Icona]	P3
Create ClickUp account and familiarise	[Avatar]	giu 19	P3
Create ClickUp account and familiarise	[Avatar]	[Icona]	P3
Create ClickUp account and familiarise	[Avatar]	[Icona]	P3
Create ClickUp account and familiarise	[Avatar]	[Icona]	P3
Create Git repository	[Avatar]	[Icona]	P3
Create Git repository	[Avatar]	[Icona]	P3
Create Git repository	[Avatar]	[Icona]	P3
Create Git repository	[Avatar]	[Icona]	P3

Figura 4.3: Board di ClickUp per il processo di coordinamento e documentazione

4.1.1 Daily stand-up meeting

In ausilio dei processi sopra descritti, è stato anche definito, in concomitanza con l'inizio del processo di formazione, un processo di **supporto** ispirato al metodo [Scrum](#), andando a programmare degli incontri giornalieri di circa 15 minuti finalizzati a:

- * **monitorare** lo stato di avanzamento delle attività svolte, da svolgere e in corso di svolgimento;
- * **risolvere** eventuali dubbi o problemi sorti durante lo svolgimento delle attività;
- * **definire** eventuali miglioramenti o cambiamenti da apportare alle da svolgere, o in corso di svolgimento, e al metodo di lavoro adottato.

4.2 Codifica

4.2.1 Configurazione di un cluster Kafka con Docker Compose

Dopo aver terminato le attività di formazione su **Apache Kafka** e **Docker Compose** ho iniziato la configurazione di un [cluster](#), che fa uso di [container](#), in grado di testare tali strumenti.

Seguendo la buona pratica dell'alta affidabilità, descritta del paragrafo [3.1.6](#), ho configurato un [cluster](#) con tre nodi **Kafka** e un nodo **Zookeeper**.

Innanzitutto per far sì che ogni [container](#) possa comunicare con gli altri è necessario creare una [Docker network](#), denominata **kafka-druid**, nel seguente modo.

Listing 4.1: docker network create

```
docker network create kafka-druid
```

Successivamente sono stati creati i nodi **Zookeeper** e **Kafka** con **Docker Compose** utilizzando il seguente file di configurazione (per motivi di spazio viene riportata la definizione di un solo nodo **Kafka**, la configurazione degli altri è del tutto analoga).

Listing 4.2: kafka-cluster-compose.yml

```
networks:
  kafka-druid:
    name: kafka-druid
    driver: bridge
    external: true
services:
  kafka:
    image: confluentinc/cp-kafka:7.4.0
    hostname: kafka
    container_name: kafka
    networks:
      - kafka-druid
    ports:
      - "29092:29092"
    environment:
      KAFKA_ADVERTISED_LISTENERS: INTERNAL://kafka:9092,EXTERNAL
        ://localhost:29092
      KAFKA_LISTENER_SECURITY_PROTOCOL_MAP: INTERNAL:PLAINTEXT,
        EXTERNAL:PLAINTEXT
```

```

KAFKA_INTER_BROKER_LISTENER_NAME: INTERNAL
KAFKA_ZOOKEEPER_CONNECT: "zookeeper:2181"
KAFKA_BROKER_ID: 1
KAFKA_LOG4J_LOGGERS: "kafka.controller=INFO,kafka.producer.async.DefaultEventHandler=INFO,state.change.logger=INFO"
KAFKA_OFFSETS_TOPIC_REPLICATION_FACTOR: 1
KAFKA_TRANSACTION_STATE_LOG_REPLICATION_FACTOR: 1
KAFKA_TRANSACTION_STATE_LOG_MIN_ISR: 1
KAFKA_AUTHORIZER_CLASS_NAME: kafka.security.authorizer.AclAuthorizer
KAFKA_ALLOW_EVERYONE_IF_NO_ACL_FOUND: "true"

```

È importante sottolineare che all'interno della configurazione, sopra descritta, vengono utilizzate le [immagini Docker](#)

confluentinc/cp-zookeeper:7.4.0 e **confluentinc/cp-kafka:7.4.0**.

Tale scelta è stata adottata, dal fatto che oramai **Confluent** è diventata una distribuzione molto alla avanguardia, in quanto fornisce soluzioni, utilizzate anche a livello aziendale, per la gestione di flussi di dati in tempo reale, come in Sync Lab.

Nonostante ciò si precisa che per i test che verranno elencati di seguito, sono utilizzabili anche le [immagini Docker](#) ufficiali di **Apache Kafka** e **Apache Zookeeper**, distribuite dalla [Apache Software Foundation](#).

4.2.2 Configurazione del file di environment per il cluster di Apache Druid

Dopo aver configurato il [cluster](#) di **Apache Kafka**, è stato necessario adattarne il file di [environment](#) nel seguente modo.

Listing 4.3: environment

```

DRUID_MAXDIRECTMEMORYSIZE=3072m
DRUID_SINGLE_NODE_CONF=nano-quickstart
druid_emitter_logging_logLevel=debug
druid_extensions_loadList=["druid-histogram", "druid-datasketches",
    "druid-lookups-cached-global", "postgresql-metadata-storage",
    "druid-multi-stage-query", "druid-kafka-indexing-service"]
druid_zk_service_host=zookeeper
druid_lookup_enableLookupSyncOnStartup=true
druid_lookup_lookupTierIsDatasource=false
druid_lookup_lookupTier=_default_tier
druid_broker_cache_useCache=true
druid_broker_cache_populateCache=true
druid_broker_cache_useResultLevelCache=true
druid_broker_cache_populateResultLevelCache=true
druid_cache_useCache=true
druid_cache_populateCache=true
druid_cache_useResultLevelCache=true
druid_cache_populateResultLevelCache=true
druid_metadata_storage_host=
druid_metadata_storage_type=postgresql
druid_metadata_storage_connector_connectURI=jdbc:postgresql://
    postgres:5432/druid
druid_metadata_storage_connector_user=druid
druid_metadata_storage_connector_password=FoolishPassword

```

```

druid_coordinator_balancer_strategy=cachingCost
druid_indexer_runner_javaOptsArray=["-server", "-Xmx1g", "-Xms1g",
    "-XX:MaxDirectMemorySize=3g", "-Duser.timezone=UTC", "-Dfile.encoding=UTF-8", "-Djava.util.logging.manager=org.apache.logging.log4j.jul.LogManager"]
druid_indexer_fork_property_druid_processing_buffer_sizeBytes=256
MiB
druid_storage_type=local
druid_storage_storageDirectory=/opt/shared/segments
druid_indexer_logs_type=file
druid_indexer_logs_directory=/opt/shared/indexing-logs
druid_processing_numThreads=1
druid_processing_numMergeBuffers=1
DRUID_LOG4J=<?xml version="1.0" encoding="UTF-8" ?><Configuration
    status="WARN"><Appenders><Console name="Console" target="SYSTEM_OUT"><PatternLayout pattern="%d{ISO8601} %p [%t] %c - %m%n"/></Console></Appenders><Loggers><Root level="info"><AppenderRef ref="Console"/></Root><Logger name="org.apache.druid.jetty.RequestLog" additivity="false" level="DEBUG"><AppenderRef ref="Console"/></Logger></Loggers></Configuration>

```

Tale azione è stata resa necessaria dal fatto che **Apache Druid** è uno strumento [OLAP](#), che necessita di grandi quantità di memoria RAM e spazio su disco per operare su moli considerevoli di dati.

Pertanto si sottolinea, che nella configurazione sopra citata, viene utilizzata una versione di **Druid**, denominata **nano-quickstart**, che permette di utilizzare tale strumento con un consumo di risorse ridotto.

4.2.3 Configurazione di un cluster di Apache Druid con Docker Compose

Per definizione del [cluster](#) di **Apache Druid**, al fine di ottimizzare il consumo di memoria RAM necessaria per la sua esecuzione, sono stati utilizzati un nodo per ogni processo di **Apache Druid**. Nel seguente file di configurazione vengono definiti: un nodo **Coordinator**, **Historical**, **Broker** e **MiddleManager**, implementato per mezzo di **PostgreSQL**.

Listing 4.4: druid-cluster-compose.yml

```
volumes:
  coordinator_var: {}
  druid_shared: {}
services:
  coordinator:
    image: apache/druid:26.0.0
    container_name: coordinator
    networks:
      - kafka-druid
    volumes:
      - druid_shared:/opt/shared
      - coordinator_var:/opt/druid/var
    depends_on:
      - zookeeper
      - postgres
    ports:
      - "8081:8081"
    command:
      - coordinator
    env_file:
      - environment
```

È importante sottolineare che all'interno del file di configurazione, sopra descritto, vengono definite anche i volumi necessari per l'archiviazione dei dati e dei log, essenziali al funzionamento del cluster.

4.2.4 Creazione del produttore di eventi Kafka

Al fine di poter testare i cluster appena creati, descritti nelle sezioni precedenti, ho creato un produttore di eventi **Kafka**, in grado d'instaurare una connessione con il cluster di **Apache Kafka**, di generare eventi in modo casuale, secondo un determinato schema predefinito e d'inviarli al cluster, sopra citato. Per la creazione del produttore di eventi, è stato utilizzato il linguaggio di programmazione **Python**, la libreria **Kafka-Python** e in particolare il modulo **KafkaProducer** nel seguente modo.

Listing 4.5: producer.py

```
producer = KafkaProducer(
    bootstrap_servers = ['localhost:29092'],
    value_serializer = lambda x: json.dumps(x).encode('utf-8')
)
```

È importante sottolineare che il produttore **Kafka** è stato creato in modo tale da inviare eventi, associati al relativo topic di appartenenza, in formato **JavaScript Object Notation (JSON)**, alla porta pubblica, unica porta che può essere raggiunta dall'esterno del message broker **Kafka**, precedentemente configurato.

4.3 Esecuzione e testing

Dopo aver configurato i [cluster](#) sopra descritti, e aver creato una [Data Pipeline](#) come illustrato nella sezione [3.3.2](#) ho creato dei test per mettere in evidenza le funzionalità di **Apache Druid** e per confrontarne le prestazioni con un classico **database relazionale**.

4.3.1 Creazione di una Data Pipeline

Confronto delle prestazioni di esecuzione tra Apache Druid e PostgreSQL

Al fine di mettere in risalto le prestazioni di **Apache Druid** è stato creato un [datasource](#) di cinque milioni di record, a partire dal seguente schema relazione.

```
CREATE TABLE accessi( nome text, cognome text, indirizzo text,
    citta text, stato text, cap int,
    email text, telefono text, eta int, altezza decimal(5,2), peso
    decimal(5,2), reddito decimal(6,2),
    datan date, professione text, istruzione text, hobby text,
    nfigli int, codice_cliente int, datareg timestamp, __time
    timestamp)
```

Si fa notare che all'interno del [datasource](#) è presente un **timestamp** primario, denominato **__time**, che viene utilizzato da **Apache Druid** per l'esecuzione efficiente delle query e il partizionamento dei dati all'interno dei segmenti di archiviazione.

Svolgimento

Per poter generare i dati necessari al [datasource](#) è stato utilizzato il produttore di eventi, sopra descritto e la libreria [Faker](#), utilizzando i seguenti metodi (per motivi di spazio viene riportata solo la generazione dei campi nome e cognome del [datasource](#)).

Listing 4.6: producer.py

```
fake = Faker()
max=random.randint(4,20)
nomi= [fake.first_name() for _ in range(max)]
max=random.randint(4,20)
cognomi= [fake.last_name() for _ in range(max)]
for n in range(500):
    for j in range(10000):
        nome= random.choice(nomi)
        cognome= random.choice(cognomi)
        my_data = {"nome": nome, "Cognome": cognome}
        producer.send("accessi", value = my_data)
```

Inoltre per far sì che il test eseguito sia il più veritiero possibile, tutti i dati generati sono stati salvati in un file **.csv** e successivamente importati all'interno di **PostgreSQL**. Dopo la creazione del [datasource](#) è stata eseguita l'[injection](#) all'interno di **Apache Druid** andando a utilizzare l'interfaccia web, reperibile all'indirizzo <http://localhost:8888/> (Figura [4.4](#)).

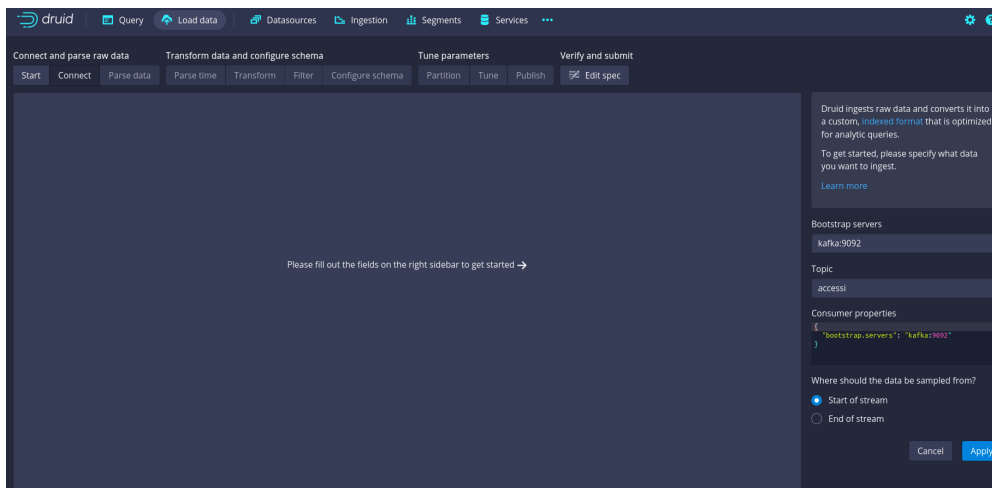


Figura 4.4: Injection del datasource all'interno di Apache Druid

In seguito sono state eseguite le seguenti query, sia su **Apache Druid** che su **PostgreSQL**.

Listing 4.7: query.sql

```
- query 1: SELECT DATE_TRUNC('day', __time), citta, COUNT(*)
           FROM accessi
           GROUP BY DATE_TRUNC('day', __time), citta

- query 2: SELECT stato, AVG(eta), AVG(reddito)
           FROM accessi
           GROUP BY stato

- query 3: SELECT DATE_TRUNC('year', __time), stato, professione
           , istruzione, nfigli, COUNT(*)
           FROM accessi
           WHERE nfigli > 0
           GROUP BY DATE_TRUNC('year', __time), stato,
                    professione, istruzione, nfigli
           ORDER BY 5 DESC

- query 4: SELECT DATE_TRUNC('hour', __time), citta, AVG(eta)
           FROM accessi
           GROUP BY DATE_TRUNC('hour', __time), citta

- query 5: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month',
           , __time), DATE_TRUNC('day', __time), stato, professione,
           istruzione, nfigli, COUNT(*)
           FROM accessi
           GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('month',
           , __time), DATE_TRUNC('day', __time), stato,
           professione, istruzione, nfigli
           ORDER BY 5 DESC
```

```
- query 6: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month',
    __time), DATE_TRUNC('day', __time),
    DATE_TRUNC('hour', __time), stato, professione, istruzione,
    nfigli, COUNT(*)
    FROM accessi
    GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('
        month', __time), DATE_TRUNC('day', __time),
        DATE_TRUNC('hour', __time), stato, professione,
        istruzione, nfigli
    ORDER BY 5 DESC
```

Risultati

I risultati ottenuti sono riportati nella tabella 4.1.

Tabella 4.1: Risultati delle query eseguite su Apache Druid e PostgreSQL

	Apache Druid [s]	PostgreSQL [s]
query1	0.6	1.2
query2	0.2	1.9
query3	1.2	2
query4	0.6	1.1
query5	1.05	2.8
query6	1.30	3.3

Considerazioni

Dai risultati ottenuti si può notare che **Apache Druid** ha prestazioni migliori rispetto a **PostgreSQL**.

Tale risultato è dovuto al fatto che **Druid** nella esecuzione delle query va a utilizzare solo le colonne richieste, grazie al fatto che il [datasource](#) viene archiviato per singole colonne.

Inoltre si può notare anche che il divario delle prestazioni tra i due strumenti aumenta nel momento in cui si eseguono query che coinvolgono operazioni che coinvolgono i timestamp primari; infatti in **Druid** i dati vengono archiviati secondo un partizionamento temporale definito al momento dell'[injection](#), che permette di eseguire le query in modo efficiente (vedi [3.2.2](#)).

Confronto delle prestazioni di esecuzione tra datasource con e senza rollup in Apache Druid

Con il termine di **Data rollup** si intende un'operazione di aggregazione eseguita sui dati, che permette di ridurre la dimensione di questi ultimi, andando a creare dei record di archiviazione più piccoli e migliorando così le prestazioni di esecuzione delle query. In **Apache Druid** l'operazione di **rollup**, viene eseguita attraverso la definizione di segmenti aggregati, creati in base a delle regole di aggregazione configurate al momento dell'**injection** del **datasource**, direttamente dall'interfaccia web di **Apache Druid** (Figura 4.5).

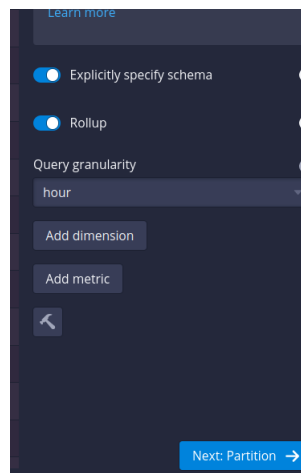


Figura 4.5: Attivazione funzionalità di rollup all'interno di Apache Druid

È importante sottolineare che l'operazione di **rollup** può essere configurata in base alla granularità temporale desiderata: si possono aggregare segmenti aventi la stessa ora, minuti o secondi e in seguito andare a considerare i valori degli altri campi. Per mettere in risalto la funzionalità appena descritta ho creato un **datasource** di cinque milioni di record, a partire dal seguente modello relazione.

```
CREATE TABLE accessi_rollup(__time timestamp, nome text,
                             cognome text, citta text, stato text, datan date, istruzione
                             text, hobby text)
```

In questo caso si tratta di un modello relazionale molto semplice e per riuscire ad apprezzare la funzionalità di **rollup** ho ridotto il numero di possibili valori, che ogni campo dati può assumere.

Pertanto il confronto è stato eseguito tra il **datasource** originale e quello a cui è stata applicata la funzionalità di **rollup**.

Inoltre per completezza tutti i dati generati sono stati salvati e importati anche all'interno di **PostgreSQL** per avere anche un confronto con un classico **database relazionale**.

Svolgimento

Per poter generare i dati necessari al [datasource](#) sopra descritto, è stato utilizzato il produttore di eventi e la libreria [Faker](#) nel seguente modo.

Listing 4.8: producer_rollup.py

```
fake = Faker()
locazione=[[fake.city(), fake.country()] for _ in range(200)]
utenti=list()
num_utenti=5000
for i in range(num_utenti):
    loc=locazione[random.randint(0,199)]
    citta=loc[0]
    stato=loc[1]
    utenti.append([fake.first_name(), fake.last_name(),fake.
        date_of_birth(minimum_age=18, maximum_age=89).strftime("
        %Y-%m-%d"), citta, stato, fake.random_element(elements=(
        "Scuola Secondaria", "Laurea triennale", "Laurea
        Magistrale", "Dottorato")), fake.random_element(elements
        =("Leggere","Viaggiare","Giocare a calcio","Giocare ai
        videogiochi","Fare sport")) ] )
for n in range(50):
    for j in range(100000):
        accesso=datetime.datetime.now().strftime("%Y-%m-%d %H:%M
        :%S")
        a=random.randint(0,4999)
        nome=utenti[a][0]
        cognome=utenti[a][1]
        datan=utenti[a][2]
        citta=utenti[a][3]
        stato=utenti[a][4]
        istruzione=utenti[a][5]
        hobby=utenti[a][6]
        my_data = {"__time": accesso, "nome": nome, "cognome":
            cognome, "datan": datan, "citta": citta, "stato":
            stato, "istruzione": istruzione,
            "hobby": hobby
        }
        producer.send("rollup", value = my_data)
```

In seguito sono state eseguite le seguenti query su i tre [datasource](#) creati.

Listing 4.9: query_rollup.sql

```
- query 1: SELECT DATE_TRUNC('day', __time), citta, COUNT(*)
           FROM accessi_rollup
           GROUP BY DATE_TRUNC('day', __time), citta

- query 2: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month
           ', __time), stato, COUNT(*)
           FROM accessi_rollup
           GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('
           month', __time), stato
           ORDER BY 4 DESC
```

```

- query 3: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month',
__, __time), DATE_TRUNC('day', __time), stato, citta, COUNT
(*)
        FROM accessi_rollup
        GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('
        month', __time), DATE_TRUNC('day', __time),
        stato, citta
        ORDER BY 5 DESC

- query 4: SELECT DATE_TRUNC('year', __time), DATE_TRUNC('month',
__, __time), DATE_TRUNC('day', __time), DATE_TRUNC('hour',
__, __time), stato, citta, COUNT(*)
        FROM accessi_rollup
        GROUP BY DATE_TRUNC('year', __time), DATE_TRUNC('
        month', __time), DATE_TRUNC('day', __time),
        DATE_TRUNC('hour', __time), stato, citta
        ORDER BY 5 DESC

```

Risultati

I risultati ottenuti sono riportati nella tabella 4.2.

Tabella 4.2: Risultati delle query eseguite con, senza rollup e su PostgreSQL

	Apache Druid con rollup [s]	Apache Druid senza rollup [s]	PostgreSQL [s]
query1	0.3	0.7	0.8
query2	0.4	0.75	1.1
query3	0.25	0.74	1.5
query4	0.16	0.8	1.6

Considerazioni

Dai risultati ottenuti si può notare che l'operazione di **rollup** permette di migliorare in modo considerevole le prestazioni di esecuzione delle query, all'interno di **Apache Druid**.

Infatti se si confronta le cardinalità dei due **datasource** con e senza **rollup** si nota che da cinque milioni di record iniziali si arriva a centocinquantamila record aggregati, che permettendo così una riduzione considerevole del volume di dati da analizzare.

Infine si sottolinea che la funzionalità di **rollup** non fa altro che aumentare il divario tra **Apache Druid** e un classico database relazione, come **PostgreSQL**.

4.3.2 Utilizzo delle tabelle di lookup in Apache Druid

Con il termine di tabelle di **lookup** si intende una funzionalità che consente sostituire i valori di un determinato attributo di un [datasource](#) con un altro valore, definito all'interno di una tabella di **lookup**.

Più in generale tale processo prende il nome di **Data enrichment**: processo di miglioramento, ampliamento o arricchimento dei dati esistenti con informazioni aggiuntive provenienti da diverse fonti.

In **Apache Druid** le tabelle di **lookup** sono costituite da un campo **chiave** a cui viene associato un campo **valore** che andrà a sostituire la chiave.

Inoltre è necessario sottolineare che le tabelle di **lookup** non hanno cronologia e lavorano indipendentemente dall'intervallo di tempo su cui si esegue la query: restituiscono sempre il dato corrente.

Per poter sperimentare tale funzionalità ho creato un [datasource](#) di cinquecento record a partire dal seguente modello relazione.

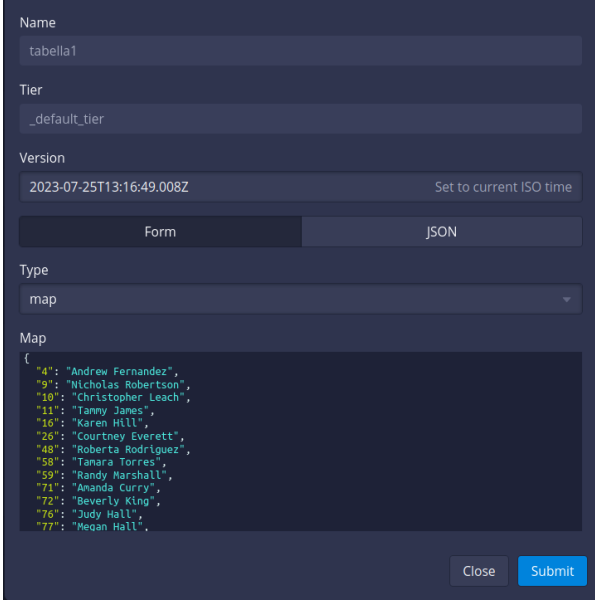
```
CREATE TABLE accessi_lookup(_time timestamp, citta text, stato text, codice_cliente text);
```

In questo caso la tabella di **lookup** è stata utilizzata per sostituire al *codice_cliente*, il *nome* e il *cognome* del cliente stesso.

Svolgimento

Per poter generare i dati necessari al [datasource](#) sopra descritto, è stato utilizzato un produttore di eventi analogo a quelli descritti nelle sezioni [4.3.1](#) e [4.3.1](#).

In questo caso è importante sottolineare che, dopo aver generato i dati necessari, tutti i record creati sono stati salvati per effettuare la creazione della tabella di **lookup** (Figura 4.6).



The screenshot shows the Apache Druid UI for creating a lookup table. The form includes the following fields and sections:

- Name:** tabella1
- Tier:** _default_tier
- Version:** 2023-07-25T13:16:49.008Z (with a "Set to current ISO time" button)
- Form/JSON:** Two tabs, with "Form" currently selected.
- Type:** map (selected from a dropdown menu)
- Map:** A configuration section showing a list of key-value pairs for client codes and names:

```
{
  "4": "Andrew Fernandez",
  "9": "Nicholas Robertson",
  "10": "Christopher Leach",
  "11": "Tanny Jones",
  "16": "Karen Hill",
  "26": "Courtney Everett",
  "48": "Roberta Rodriguez",
  "58": "Tamara Torres",
  "59": "Randy Marshall",
  "71": "Amanda Curry",
  "72": "Beverly King",
  "76": "Judy Hall",
  "77": "Megan Hall",
}
```
- Buttons:** "Close" and "Submit" buttons at the bottom right.

Figura 4.6: Creazione della tabella di lookup all'interno di Apache Druid

In seguito per poter testare tale funzionalità è stata eseguita la seguente query.

```
SELECT LOOKUP(codice_cliente,'tabella1'), datan, citta, stato
FROM accessi_lookup
```

Si sottolinea che, per poter utilizzare le tabelle di **lookup** in **Apache Druid**, è necessario inserire all'interno della query desiderata la funzione **LOOKUP**, riportando come primo parametro il campo chiave della tabella di **lookup** e come secondo parametro il nome della tabella stessa.

Risultati

Il risultato ottenuto è il seguente (Figura 4.7).

A EXP\$0 LOOKUP("codice_clien...	A datan	A citta	A stato	
Lisa Miles	1996-01-01	Rossmouth	Aruba	
Michelle Ferrell	1966-07-27	Jamieborough	Saudi Arabia	
Jessica Martinez	1953-02-02	East Michaeland	Iran	
Amanda Curry	1980-10-24	Gonzalezland	Cocos (Keeling) Islands	
Angela Douglas	1980-10-24	Lake Natashaview	North Macedonia	
Michelle Ferrell	1934-02-18	Lorimouth	Kiribati	
Jennifer Harrison	1972-06-28	Grahamton	Mexico	
Cassie Miller	1979-02-23	Port Andre	Hong Kong	
Christopher Leach	1987-07-02	Justinview	Turks and Caicos Island:	
Bryan Pena	1998-08-17	Justinview	Cameroon	

Figura 4.7: Risultato della query eseguita con la funzionalità di lookup

Considerazioni

Dai risultati ottenuti si sottolinea che per ottenere lo stesso risultato all'interno di un classico **database relazionale** è necessario ricorrere all'unione tra due tabelle, operazione che risulta essere molto onerosa.

Capitolo 5

Valutazioni e Conclusioni

- 5.1 Raggiungimento degli obiettivi
- 5.2 Attualizzazione dei rischi
- 5.3 Contenuti formativi acquisiti
- 5.4 Divario rispetto al percorso di studi
- 5.5 Valutazione personale

Acronimi e abbreviazioni

API [Application Programming Interface](#). 21, 22

EDA [Event Driven Architecture](#). 20, 29

ICT [Information and Communication Technology](#). 1, 47

JSON [JavaScript Object Notation](#). 34

OH&S [Occupational Health and Safety](#). 1, 48

OLAP [Online Analytical Processing](#). 3, 4, 21, 22, 27, 33, 48

SGSI [Sistema Gestione Sicurezza Informazioni](#). 1, 48

TCP [Transmission Control Protocol](#). 15, 49

Glossario

agile è un insieme di metodi di sviluppo software che si basano su un approccio iterativo e incrementale. L'obiettivo principale dei metodi agili è quello di fornire risultati di alta qualità in modo rapido ed efficiente, consentendo ai team di adattarsi ai cambiamenti delle specifiche o dei requisiti durante il processo di sviluppo.. [8](#)

Apache Software Foundation è un'organizzazione non profit che supporta lo sviluppo di progetti open source. [14](#), [21](#), [32](#)

API È un insieme di definizioni e protocolli che permettono a un software di comunicare con un altro. [45](#)

board è una bacheca virtuale che permette di visualizzare le attività da svolgere, quelle in corso e quelle completate. [9](#), [29](#)

Business Innovation è un processo che permette d' introdurre nuovi metodi, idee, prodotti e servizi per migliorare l'efficienza, la produttività e la competitività di un'organizzazione. [1](#)

cluster è un insieme di elaboratori connessi tra loro che lavorano in parallelo per eseguire un compito comune. [22–25](#), [31–35](#)

compattazione è una strategia che si concentra sulla rimozione efficiente dei duplicati nei log dei topic, mantenendo solo la versione più recente di ciascun messaggio con una determinata chiave. [18](#)

container è un'unità software standard che raggruppa il codice e tutte le sue dipendenze in modo da poter essere eseguito in modo affidabile e veloce in qualsiasi ambiente. [4](#), [9](#), [15](#), [22](#), [31](#)

Data Analytics è il processo che permette di esaminare i dati per trarne conclusioni sull'informazione che contengono. [2](#), [8](#)

Data Pipeline è un insieme di operazioni che permettono di trasformare e analizzare i dati in modo da renderli pronti per l'archiviazione. [2–4](#), [15](#), [27](#), [35](#)

Data Processing è un sistema di manipolazione, trasformazione e analisi di dati grezzi al fine di ottenere informazioni significative e approfondite. Comprende una serie di passaggi che permettono di convertire dati non strutturati in forme più utili e che facilitano la loro elaborazione. [2](#), [3](#)

datasource in Apache Druid è un'astrazione rappresenta un insieme di dati, organizzati in tabelle, che possono essere analizzati e interrogati. [26](#), [35](#), [37–41](#)

Docker è un progetto open-source che automatizza il deployment di applicazioni all'interno di container software. [9](#)

Docker network rappresenta un'interfaccia di rete virtuale che permette ai container di comunicare tra loro e con il mondo esterno. [31](#)

EDA è un pattern architetturale la cui interazione tra le componenti del sistema è guidata dalla generazione, trasmissione e ricezione di eventi. [45](#)

enviroment è un insieme di variabili d'ambiente che definiscono il comportamento di un processo, le risorse utilizzabili e le informazioni di configurazione. [32](#)

Faker è una libreria Python che permette di generare in modo casuale una varietà di dati, come nomi, indirizzi, numeri di telefono, indirizzi email e molti altro. [35](#), [39](#)

fault tolerance è la capacità di un sistema di continuare a operare anche in caso di malfunzionamenti. [15](#), [16](#), [22](#)

hands-on è un'attività pratica che permette di sperimentare quanto appena appreso. [30](#)

ICT è un termine generico che indica tutte le tecnologie che riguardano la trasmissione, la ricezione e l'elaborazione di informazioni sotto forma di segnali elettronici o elettromagnetici. [45](#)

immagini Docker è un pacchetto software leggero, autonomo ed eseguibile che include tutto il necessario per eseguire un'applicazione. [32](#)

injection è la tecnica attraverso cui si inseriscono dati da un sistema esterno ad un altro. [23](#), [24](#), [35](#), [37](#), [38](#)

JSON è un formato di testo per lo scambio di dati strutturati basato sul linguaggio JavaScript. È comunemente utilizzato per trasmettere dati su applicazioni web (ad esempio inviare dati da un server al client, in modo che possano essere visualizzati su una pagina web o viceversa). È un formato di dati molto più leggero rispetto all'XML perché è più conciso e può essere analizzato facilmente da un browser.. [45](#)

Kafka-Python è una libreria Python che consente agli sviluppatori di interagire con il sistema di messaggistica distribuita Apache Kafka utilizzando il linguaggio di programmazione Python; viene ampiamente utilizzato per la costruzione di sistemi di streaming o dove è necessario gestire grandi quantità di dati in modo scalabile e affidabile. [34](#)

log è un file che registra gli eventi che si verificano durante l'esecuzione di un sistema o di un'applicazione. [14](#), [17](#), [18](#), [21](#), [34](#)

message broker è il componente intermediario che permette d' inviare e ricevere messaggi da più sorgenti verso più destinazioni. Un message broker facilita lo scambio dei messaggi tra le componenti di un sistema distribuito, consentendo di comunicare in modo asincrono e disaccoppiato. [14–17](#), [27](#), [34](#)

metadati sono informazioni finalizzati a descrivere altre strutture dati, in modo da renderle più comprensibili e facili da individuare, analizzare e utilizzare. [10](#), [11](#), [16](#), [22](#), [24](#), [25](#)

Middleware è un software che si interpone tra un sistema operativo e le applicazioni che vengono eseguite al di sopra. Il suo scopo è quello di facilitare lo sviluppo di applicazioni e di nascondere la complessità del sistema operativo sottostante. [3](#), [4](#)

modello incrementale è un modello di sviluppo software che prevede la consegna di funzionalità in maniera incrementale, cioè il prodotto finale viene sviluppato attraverso una serie di rilasci parziali. [8](#)

OH&S è un sistema di gestione che permette di gestire in modo strutturato la salute e sicurezza dei lavoratori. [45](#)

OLAP è un insieme di metodi finalizzato a effettuare analisi rapide e approfondite su grandi volumi di dati, provenienti da uno o più sorgenti, per prendere decisioni a riguardo. [45](#)

on-promise è un modello di distribuzione software in cui l'applicazione viene ospitata sul server del cliente. [15](#)

open source è un modello di sviluppo del software basato sulla condivisione del codice sorgente, che permette a chiunque di leggere, studiare, modificare e distribuire il software stesso o parte di esso. [14](#), [17](#), [21](#)

Product Backlog è un elenco ordinato di requisiti che rappresentano le funzionalità del prodotto finale. [8](#)

project management è l'insieme di pratiche, metodi, processi e strumenti utilizzati per pianificare, organizzare, eseguire, monitorare e controllare le attività necessarie per raggiungere gli obiettivi di un progetto in modo efficace ed efficiente. [8](#), [29](#)

repository è un ambiente di archiviazione centralizzato in cui vengono conservati e gestiti i dati. [10](#), [11](#)

Scrum è un framework agile per la gestione del ciclo di sviluppo del software. [8](#), [31](#)

SGSI è un sistema di gestione che permette di gestire in modo strutturato la sicurezza delle informazioni aziendali. [45](#)

sprint è un periodo di tempo breve, della durata di una o due settimane, in cui viene sviluppata una funzionalità del prodotto finale. [8](#)

streaming di eventi è una pratica di acquisizione dei dati in tempo reale da fonti di eventi come database, flussi di eventi; memorizzando tutto ciò per un recupero futuro di tali informazioni, reagendo a flussi di eventi in tempo reale. [14](#), [20](#)

task è un compito che deve essere svolto per portare a termine un'attività più grande. [9](#)

TCP è uno dei principali protocolli di comunicazione della suite di protocolli Internet (TCP/IP). Si tratta di un protocollo di trasporto affidabile orientato alla connessione utilizzato per fornire comunicazioni dati affidabili e ordinate tra dispositivi in una rete, come ad esempio tra computer su Internet.. [45](#)

topic è un canale o area di comunicazione a cui vengono inviati e ricevuti messaggi categorizzati. [17](#), [34](#)

virtual machine è un ambiente computazionale autonomo e isolato che opera come una macchina fisica separata, ma è ospitato all'interno di un sistema operativo o di un altro ambiente hardware. [15](#)

volumi sono un'unità di archiviazione che possono essere montati e utilizzati da un container. [34](#)

working directory è la directory di lavoro corrente. [10](#)

Bibliografia

Siti web consultati

Cenni storici di LaTeX. URL: <https://en.wikipedia.org/wiki/LaTeX>.

Cenni storici di YAML. URL: <https://en.wikipedia.org/wiki/YAML>.

Comandi base sull'utilizzo di Git. URL: <https://git-scm.com/book/en/v2/Git-Basics-Getting-a-Git-Repository>.

Documentazione ufficiale del linguaggio di programmazione Python3. URL: <https://docs.python.org/3/>.

Documentazione ufficiale del linguaggio YAML. URL: <https://yaml.org>.

Documentazione ufficiale di Git. URL: <https://git-scm.com/docs>.

Documentazione ufficiale di GitHub. URL: <https://docs.github.com/en>.

Introduzione a ClickUp. URL: <https://clickup.com/on-demand-demo>.

Introduzione a Docker Compose. URL: <https://docs.docker.com/compose/>.

Introduzione al framework Scrum. URL: [https://it.wikipedia.org/wiki/Scrum_\(informatica\)](https://it.wikipedia.org/wiki/Scrum_(informatica)).

Introduzione al metodo incrementale. URL: https://en.wikipedia.org/wiki/Iterative_and_incremental_development.

Sito web ufficiale di SycLab. URL: <https://www.synclab.it/>.