ByteOps.swe@gmail.com

Specifica tecnica

Informazioni documento

Redattori	A. Barutta R. Smanio L. Skenderi F. Pozza D. Diotto N. Preto
Verificatori	E. Hysa A. Barutta D. Diotto L. Skenderi R. Smanio F. Pozza
Destinatari	ByteOps T. Vardanega R. Cardin

Versione	Data	Autore	Verificatore	Dettaglio
0.0.1	01/02/2024	D. Diotto	L.Skenderi	Impostazione sezioni

Indice

ByteOps

Contents

List of Figures

1 Introduzione

1.1 Scopo del documento

Il presente documento si propone come una risorsa esaustiva per la comprensione degli aspetti tecnici chiave del progetto "InnovaCity". La sua finalità principale è fornire una descrizione dettagliata e approfondita di due aspetti centrali: l'architettura implementativa e l'architettura di deployment.
Nel contesto dell'architettura implementativa, è prevista un'analisi approfondita che si estenda anche al livello di design più dettagliato. Ciò include la definizione e la spiegazione dettagliata dei design pattern e degli idiomi utilizzati nel contesto del progetto.
Gli obbiettivi del presente documento sono: motivare le scelte di sviluppo adottate, fungere da guida fondamentale per l'attività di codifica ed infine garantire una completa copertura dei requisiti identificati nel documento *Analisi dei Requisiti v2.0.0*.

1.2 Scopo del progetto

Sviluppare una piattaforma di monitoraggio di una "Smart City" che consenta di avere sotto controllo lo stato di salute della città in modo tale da prendere decisioni veloci, efficaci ed analizzare poi gli effetti conseguenti. A tale scopo il proponente richiede di simulare dei sensori posti in diverse aree per reperire informazioni relative alle condizioni della città come, ad esempio, temperatura, umidità, quantità di polveri sottili nell'aria, traffico, livelli di acqua, stato di riempimento delle isole ecologiche, guasti elettrici e molto altro. I dati trasmessi in tempo reale dai sensori devono poter essere memorizzati in un database in modo tale da renderli disponibili per la visualizzazione tramite una dashboard, composta da widget e grafici, per una visione d'insieme delle condizioni della città in tempo reale. L'applicativo potrà consentire alle autorità locali di prendere decisioni informate e tempestive sulla gestione delle risorse e sull'implementazione di servizi e, inoltre, si potrebbe rivelare uno strumento essenziale per coinvolgere i cittadini nella gestione e nel miglioramento della città.
L'implementazione di una città monitorata da sensori rappresenta un approccio promettente nell'ottica di ottimizzare l'efficienza e la qualità della vita urbana. Tale sistema consente una raccolta continua di dati e informazioni cruciali, fornendo una base solida per l'ottimizzazione dei servizi pubblici, la gestione del traffico, la sicurezza e la sostenibilità ambientale.

1.3 Glossario

Per evitare possibili ambiguità che potrebbero sorgere durante la lettura dei documenti, alcuni termini utilizzati sono stati inseriti nel documento *Glossario v 2.0.0* . Sarà possibile individuare il riferimento al Glossario per mezzo di una G a pedice del termine.

1.3.1 Riferimenti normativi

- *Norme di progetto v2.0.0*
- Capitolato d'appalto C6 - InnovaCity: url

1.3.2 Riferimenti informativi

- *Analisi dei Requisiti v2.0.0*

1.4 Riferimenti

2 Tecnologie

In questa sezione vengono definiti gli strumenti e le tecnologie impiegati per lo sviluppo e l'implementazione del software relativo al progetto InnovaCity. Si procederà quindi con la descrizione delle tecnologie e dei linguaggi di programmazione utilizzati, delle librerie e dei framework necessari, nonché delle infrastrutture richieste. L'obiettivo principale è garantire che il software sia sviluppato utilizzando le tecnologie più appropriate e selezionando le opzioni ottimali in termini di efficienza, sicurezza e affidabilità.

Ambiente di sviluppo

Per lo sviluppo del progetto sono stati utilizzati container Docker per garantire un ambiente di sviluppo consistente e riproducibile. Di seguito sono elencate le immagini Docker utilizzate:

- **Python:** Python:3.9
- **Apache Kafka:** bitnami/kafka:latest
- **ClickHouse:**
- **Grafana:**

2.1 Linguaggi e formato dati

2.1.1 Python

Linguaggio di programmazione ad alto livello, interpretato e multi-paradigma.

Versione

La versione utilizzata è: 3.9

Librerie o framework

- **Confluent Kafka**
  - **Documentazione:** https://developer.confluent.io/get-started/python/
  - **Versione:** 2.3.0
  - Libreria Python che fornisce un insieme completo di strumenti per agevolare la produzione e il consumo di messaggi da Apache Kafka.
- **Faust**
  - **Documentazione:** https://faust.readthedocs.io/en/latest/
  - **Versione:** 1.10.4
  - Framework Python per la creazione di applicazioni di streaming in tempo reale, con un'enfasi particolare sull'elaborazione di eventi e dati in tempo reale. Fornisce un'API dichiarativa e funzionale per definire i flussi di dati e le trasformazioni, consentendo agli sviluppatori di scrivere facilmente applicazioni scalabili e affidabili per il trattamento di grandi volumi di dati in tempo reale. Faust si integra nativamente con Apache Kafka e offre funzionalità avanzate come il bilanciamento del carico, la gestione dello stato, la gestione delle query, e la tolleranza ai guasti, rendendolo una scelta potente per lo sviluppo di sistemi di streaming complessi e robusti.
- **Pytest**
  - **Documentazione:** https://docs.pytest.org/en/7.1.x/contents.html
  - **Versione:** 8.0.2
  - Framework di testing per Python, noto per la sua semplicità e potenza. Consente agli sviluppatori di scrivere test chiari e concisi utilizzando una sintassi intuitiva e flessibile. Pytest supporta una vasta gamma di funzionalità, tra cui test di unità, integrazione e accettazione, parametrizzazione dei test e gestione delle fixture. Da citare anche l'utilizzo di *Pytest-asyncio* per testare codice asincrono e *Pytest-cov* per la copertura del codice.
- **Pylint**
  - **Documentazione:** https://pylint.readthedocs.io/en/stable/
  - **Versione:** 3.1.0
  - Strumento di analisi statica per il linguaggio di programmazione Python. Esamina il codice sorgente per individuare potenziali errori, conformità alle linee guida di stile

e altre possibili problematiche. Pylint fornisce un punteggio di qualità del codice e suggerimenti per migliorare la leggibilità, la manutenibilità e la correttezza del codice Python.

- **Clickhouse-connect**
  - **Documentazione:** https://clickhouse.com/docs/en/integrations/python
  - **Versione:** 1.10.4
  - ClickHouse Connect è un modulo Python che viene utilizzato nei test. Fornisce un'API semplice e intuitiva per eseguire query su ClickHouse direttamente da codice Python

Utilizzo nel progetto

- Creazione di simulazioni dei sensori e dei microcontrollori, incluse le logiche di scrittura e invio dei dati registrati.
- Modello per il calcolo del punteggio di salute della città;
- Testing.

2.1.2 SQL (Structured Query Language)

Linguaggio standard per la gestione e la manipolazione dei database che lo supportano

Utilizzo nel progetto

Gestione e interrogazione database Clickhouse.

2.1.3 JSON (JavaScript Object Notation)

JSON è un formato di scrittura leggibile dalle persone e facilmente interpretabile dai computer. È utilizzato principalmente per lo scambio di dati strutturati attraverso le reti, come Internet.
Il formato JSON si basa su due strutture di dati principali:

- **Oggetti**: Rappresentati da coppie chiave-valore racchiuse tra parentesi graffe { }, dove la chiave è una stringa e il valore può essere un altro oggetto, un array, una stringa, un numero, un booleano o `null`.
- **Array**: Una raccolta ordinata di valori, racchiusi tra parentesi quadre [ ], in cui ogni elemento può essere un oggetto, un array, una stringa, un numero, un booleano o `null`.

JSON offre una sintassi semplice e chiara per la rappresentazione dei dati, che lo rende ampiamente utilizzato in molti contesti, inclusi lo sviluppo web, le API di servizi web e lo scambio di dati tra applicazioni. La sua leggibilità e la sua natura basata su testo lo rendono particolarmente adatto per l'interazione tra sistemi eterogenei.

Utilizzo nel progetto

- Formato dei messaggi spediti dai simulatori dei sensori al broker Kafka;
- Impostazione dashboard grafana.

2.1.4 YAML (YAML Ain't Markup Language)

Formato di serializzazione leggibile dall'uomo utilizzato per rappresentare dati strutturati in modo chiaro e semplice.

Utilizzo nel progetto

- Configurazione docker compose;
- Configurazione pipeline Git-Hub workflow per Countinuous Integration;
- Configurazione provisioning Grafana e politiche di notifica allerte.

2.2 Database e servizi

2.2.1 Apache Kafka

Apache Kafka è una piattaforma open-source di streaming distribuito sviluppata da Apache Software Foundation. È progettata per la gestione di flussi di dati in tempo reale in modo scalabile, affidabile e efficiente. Kafka è utilizzato ampiamente nell'ambito del data streaming e del data integration in molte applicazioni moderne.

Versione

La versione utilizzata è: 3.7.0

2.2.2 Link download

https://kafka.apache.org/downloads

Funzionalità e Vantaggi di Apache Kafka

Le principali funzionalità e vantaggi di Apache Kafka includono:

- **Pub-Sub Messaging:** kafka utilizza un modello di messaggistica publish-subscribe, dove i produttori di dati inviano messaggi a un topic e i consumatori possono sottoscriversi a tali topic per ricevere i messaggi;
- **Disaccoppiamento Produttore - Consumatore:**il disaccoppiamento avviene perché Produttori e Consumatori non devono essere a conoscenza l'uno dell'altro o interagire direttamente. Invece, interagiscono attraverso il broker Kafka, che funge da intermediario per la comunicazione;
- **Architettura Distribuita:** kafka è progettato per essere distribuito su un cluster di nodi, consentendo una scalabilità orizzontale per gestire grandi volumi di dati e carichi di lavoro;
- **Persistenza e Affidabilità:** Kafka conserva i dati in modo persistente su disco, garantendo la durabilità dei messaggi anche in caso di guasti hardware o arresti anomali. Questo assicura anche un alto livello di affidabilità;
- **Alta Disponibilità:** grazie alla sua architettura distribuita, Kafka offre alta disponibilità e tolleranza ai guasti, consentendo ai cluster di continuare a funzionare anche in presenza di nodi o componenti falliti;
- **Elaborazione degli Stream:** kafka supporta anche l'elaborazione degli stream di dati in tempo reale tramite API come Kafka Streams e Kafka Connect, consentendo agli sviluppatori di scrivere applicazioni per l'analisi e l'elaborazione dei dati in tempo reale.

Casi d'uso di Apache Kafka

Apache Kafka è utilizzato in una vasta gamma di casi d'uso, tra cui:

- **Data Integration:** Kafka viene utilizzato per integrare dati provenienti da diverse fonti e sistemi, consentendo lo scambio di dati in tempo reale tra applicazioni e sistemi eterogenei.
- **Streaming di Eventi:** Molte applicazioni moderne, come le applicazioni IoT (Internet of Things) e le applicazioni di monitoraggio in tempo reale, utilizzano Kafka per il streaming di eventi in tempo reale e l'analisi dei dati.
- **Analisi dei Log:** Kafka è spesso utilizzato per l'analisi dei log di sistema e applicativi in tempo reale, consentendo il monitoraggio delle prestazioni, la rilevazione degli errori e l'analisi dei pattern di utilizzo.

- **Elaborazione di Big Data:** Kafka è integrato con tecnologie di big data come Apache Hadoop e Apache Spark, consentendo l'elaborazione di grandi volumi di dati in tempo reale.
- **Messaggistica Real-time:** Kafka è ampiamente utilizzato per la messaggistica real-time in applicazioni di social media, e-commerce e finanziarie, dove la velocità e l'affidabilità della messaggistica sono cruciali.

Utilizzo nel progetto

Kafka funge da intermediario dei messaggi che riceve i dati dai produttori di dati e li rende disponibili ai consumatori. Nel contesto di questo progetto, i dati provenienti dalle simulazioni di sensori vengono inviati a *Kafka* come messaggi in formato *JSON*.

Consumatori di dati:

- ***ClickHouse:*** *Kafka* invia i dati ai consumatori, inclusi i database come *ClickHouse*, dove i dati vengono salvati per l'analisi e l'archiviazione a lungo termine.
- ***Faust:*** Per soddisfare il requisito opzionale del calcolo del punteggio di salute, *kafka* rende disponibili i dati in tempo reale ad una app di Faust per il processing il quale calcoli il punteggio attraverso una funzione di aggregazione complessa e renda disponbile il risultato in una coda dedicata *kafka* ai servizi interessati.

In breve, *Kafka* funge da ponte tra i produttori di dati (simulazioni di sensori) e i consumatori di dati (*ClickHouse* o altri servizi futuri). Gestisce il flusso dei dati in tempo reale e garantisce che i dati siano disponibili per l'elaborazione e la visualizzazione in modo efficiente e scalabile.

2.2.3 Clickhouse

Clickhouse è un sistema di gestione di database (DBMS) di tipo column-oriented, progettato principalmente per l'analisi di grandi volumi di dati in tempo reale. È un progetto open-source sviluppato da Yandex, un motore di ricerca russo, ed è stato creato per rispondere alle esigenze di elaborazione analitica ad alte prestazioni.

Versione

La versione utilizzata è: 24.1.6.52

Link download

https://clickhouse.com/

Funzionalità e Vantaggi di Clickhouse

- **Modello di dati column-oriented:** a differenza dei tradizionali DBMS che memorizzano i dati in modo row-oriented, dove le righe complete sono memorizzate in sequenza, Clickhouse memorizza i dati in modo column-oriented. Questo significa che i dati di ogni colonna sono memorizzati insieme, permettendo una maggiore compressione e velocità di query per le analisi che coinvolgono molte colonne;
- **Architettura Distribuita e scalabilità:** Clickhouse è progettato per funzionare in un ambiente distribuito, consentendo la scalabilità orizzontale per gestire grandi carichi di lavoro;
- **Compressione dei Dati:** utilizza algoritmi efficienti per ridurre lo spazio di archiviazione richiesto per i dati, riducendo i costi di archiviazione;
- **Alte Prestazioni:** ottimizzato per eseguire query analitiche su grandi volumi di dati in tempo reale, garantendo tempi di risposta bassi anche con carichi di lavoro elevati.
- **Supporto per SQL:** supporta un sottoinsieme del linguaggio SQL, consentendo agli sviluppatori di scrivere query complesse per l'analisi dei dati;
- **Integrazione con Strumenti di Business Intelligence (BI):** può essere integrato con strumenti di BI popolari come Tableau, Power BI, Qlik, Grafana per la visualizzazione e l'analisi dei dati.

Casi d'Uso di Clickhouse

Clickhouse è adatto per una vasta gamma di casi d'uso, tra cui:

- **Analisi dei Log:** clickhouse può essere utilizzato per analizzare i log di grandi dimensioni generati da server, applicazioni web e dispositivi IoT;
- **Analisi dei Dati in Tempo Reale:** Clickhouse è ideale per l'analisi dei dati in tempo reale, consentendo agli utenti di eseguire query complesse su flussi di dati in continua evoluzione;
- **Reporting e Dashboard:** Clickhouse può essere utilizzato per generare report e dashboard interattivi per monitorare le prestazioni del business e identificare tendenze.

Utilizzo nel progetto

Nel contesto del progetto, **Clickhouse** svolge una serie di ruoli cruciali per garantire l'efficacia e l'efficienza dell'analisi dei dati provenienti dai sensori IoT:

- **Integrazione con Kafka**: **Clickhouse** viene utilizzato per recuperare in tempo reale i dati dal server Kafka, consentendo una continua acquisizione dei dati dai sensori IoT. Questa integrazione permette di assicurare che le informazioni più recenti siano immediatamente disponibili per l'analisi.
- **Organizzazione efficiente dei dati**: Grazie alla sua architettura columnar, **Clickhouse** è in grado di organizzare i dati in modo ottimale per l'analisi di grandi volumi di dati. La struttura columnar consente una compressione dei dati efficace e un accesso rapido alle informazioni, migliorando le prestazioni complessive del sistema.
- **Aggregazione rapida dei dati**: **Clickhouse** offre potenti funzionalità per eseguire operazioni di aggregazione sui dati in modo rapido e incrementale. Ciò significa che è possibile ottenere risposte rapide alle query di aggregazione anche su enormi quantità di dati, consentendo analisi in tempo quasi reale delle misurazioni dei sensori IoT.
- **Integrazione con Grafana**: I dati elaborati e aggregati da **Clickhouse** sono resi disponibili per il reperimento tramite Grafana. Grafana consente di creare dashboard interattive e report visivi basati sui dati dei sensori IoT, offrendo agli utenti un'interfaccia intuitiva per l'analisi e la visualizzazione dei dati.

2.2.4 Grafana

Grafana è una piattaforma open-source per la visualizzazione e l'analisi dei dati, utilizzata per creare dashboard interattive e grafici da fonti di dati eterogenee.

Versione

La versione utilizzata è: x.x.x

Link download

https://clickhouse.com/

Funzionalità e Vantaggi di Grafana

- **Dashboard interattive**: Creazione di dashboard personalizzate e interattive per visualizzare dati provenienti da diverse fonti in un'unica interfaccia.
- **Connessione a sorgenti di dati eterogenee**: Supporto per una vasta gamma di sorgenti di dati, inclusi database, servizi cloud, sistemi di monitoraggio, API e altro ancora.

- **Ampia varietà di visualizzazioni**: Selezione di pannelli e visualizzazioni, tra cui grafici a linea, a barre, a torta, termometri, mappe geografiche e altro ancora, per adattarsi alle esigenze specifiche di visualizzazione dei dati.
- **Query e aggregazioni flessibili**: Esecuzione di query flessibili e aggregazione dei dati in modi personalizzati per ottenere insight approfonditi dai dati.
- **Notifiche e allarmi**: Impostazione di avvisi in base a criteri predefiniti, come soglie di performance, e ricezione di notifiche tramite diversi canali, tra cui email, Slack e molti altri.
- **Gestione degli accessi e dei permessi**: Controllo degli accessi e dei permessi degli utenti in modo granulare, gestendo chi può visualizzare, modificare o creare dashboard e pannelli.
- **Integrazione con altre applicazioni e strumenti**: Integrazione con una vasta gamma di applicazioni e strumenti, tra cui sistemi di log management, strumenti di monitoraggio delle prestazioni, sistemi di allerta e altro ancora.

Casi d'Uso di Grafana

- **Monitoraggio delle prestazioni**: Monitoraggio in tempo reale delle metriche di sistema come CPU, memoria e rete per identificare e risolvere rapidamente problemi di prestazioni.
- **Analisi dei log**: Analisi e visualizzazione dei log delle applicazioni e dell'infrastruttura per individuare pattern e risolvere problemi operativi.
- **Monitoraggio dell'infrastruttura**: Monitoraggio dello stato e delle prestazioni di server, servizi cloud, database e altri componenti IT per garantire un funzionamento ottimale dell'infrastruttura.
- **DevOps e CI/CD**: Monitoraggio dei processi di sviluppo, test e distribuzione del software per migliorare la collaborazione e l'efficienza del team.
- **Monitoraggio di dispositivi IoT**: Monitoraggio dei dispositivi IoT per raccogliere e visualizzare dati di sensori e dispositivi connessi, consentendo una gestione efficiente degli ambienti IoT.

Utilizzo nel progetto

Nel contesto di un progetto che coinvolge la visualizzazione e l'analisi di miliardi di misurazioni di sensori IoT, Grafana viene utilizzato principalmente per:

- **Visualizzazione dei dati**: Grafana consente agli utenti di creare dashboard personalizzate e grafici interattivi che mostrano i dati provenienti dai sensori IoT in modo chiaro e comprensibile. Questi grafici possono essere configurati per visualizzare metriche specifiche nel formato desiderato, consentendo agli utenti di monitorare facilmente le prestazioni dei sensori e rilevare eventuali pattern o anomalie nei dati.
- **Analisi dei dati**: Grafana offre una vasta gamma di opzioni per analizzare i dati, inclusi filtri, aggregazioni, calcoli e altro ancora. Gli utenti possono eseguire query sui dati direttamente da Grafana e visualizzare i risultati in grafici, permettendo loro di ottenere una comprensione più approfondita delle tendenze e dei modelli presenti nei dati dei sensori IoT.
- **Monitoraggio in tempo reale**: Grafana supporta il monitoraggio in tempo reale dei dati, consentendo agli utenti di visualizzare aggiornamenti istantanei sui valori dei sensori e le metriche correlate. Ciò è particolarmente utile per l'analisi delle prestazioni in tempo reale e per la rilevazione immediata di problemi o anomalie nei dati dei sensori.
- **Allerta e notifica**: Grafana offre funzionalità avanzate di allerta e notifica che consentono agli utenti di impostare avvisi basati su condizioni specifiche dei dati. Ad esempio, è possibile configurare Grafana per inviare notifiche via email o tramite servizi di messaggistica istantanea quando un determinato sensore supera una soglia prestabilita o quando si verifica un'anomalia nei dati.

3 Architettura di sistema

3.1 Architettura di implementazione

Il sistema richiede la capacità di elaborare dati provenienti da diverse fonti in tempo reale e di fornire una visualizzazione immediata e continua di tali dati, permettendo di monitorarne gli andamenti e di rilevare eventuali anomalie. Per tale scopo, l'architettura di sistema adottata è la κ-*architecture*.

3.1.1 κ-architecture

L'architettura Kappa è un modello di elaborazione dati in streaming che offre un'alternativa all'architettura Lambda. Il suo obiettivo principale è unificare l'elaborazione in tempo reale e batch (per i dati storici) all'interno di un unico stack tecnologico.

Vantaggi

- Semplice da implementare e gestire, costi di manutenzione ridotti;

- Assicura coerenza tra l'analisi in tempo reale e batch.

Svantaggi

- Potenziale rallentamento dell'analisi in tempo reale, meno flessibile rispetto a Lambda.

3.1.2 Componenti di sistema

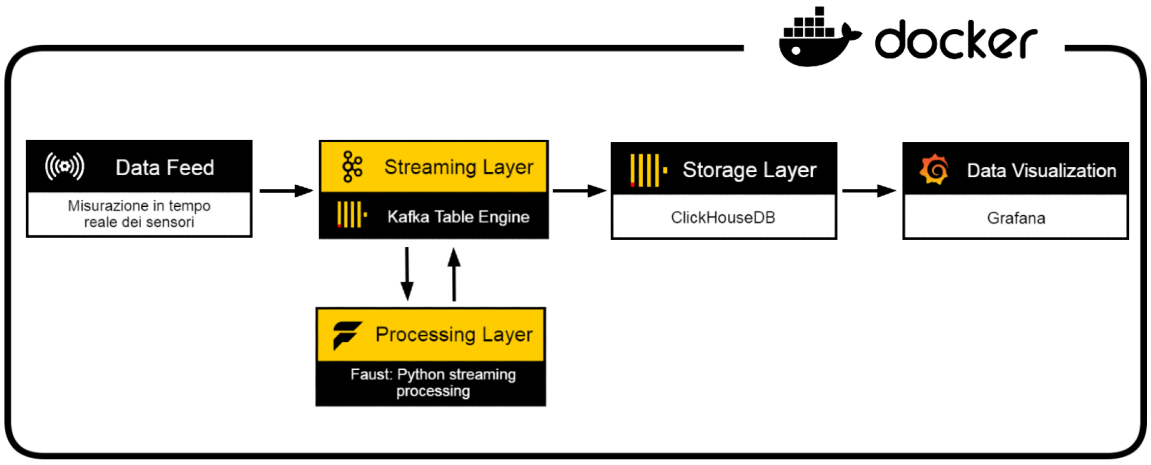

Figure 1: Componenti dell'architettura - Innovacity

- **Data feed**:Le sorgenti dati sono costituite da sensori IoT dislocati sul territorio cittadino. Questi sensori sono in grado di inviare, ad intervalli regolari, messaggi contenenti misurazioni allo streaming layer;
- **Streaming layer**:Lo streaming layer gestisce i dati in arrivo in tempo reale, per poi archiviarli sistematicamente nello storage layer. Lo streaming Layer è composto da:
  - **Apache Kafka**: Kafka è un sistema di messaggistica distribuito che consente di pubblicare, sottoscrivere e archiviare messaggi in tempo reale. Kafka è utilizzato per ricevere i dati dai sensori IoT e renderli disponibili per l'elaborazione in tempo reale e batch.

  - **Clickhouse Kafka table engine**:consumatore che legge i dati dal server Kafka per persisterli nello storage layer.

- **Processing Layer:** Il processing Layer è costituito da Faust che consuma i dati dallo streaming layer e li processa in tempo reale. Faust è un framework Python che consente di scrivere applicazioni di streaming in tempo reale. Faust è utilizzato per elaborare i dati in arrivo tramite un modello per il calcolo del punteggio di salute che poi viene reso nuovamente disponibili allo streaming layer.
- **Storage layer**:Lo storage layer è costituito da un database column-oriented, ClickHouse, che archivia i dati in arrivo dallo streaming layer. Questi dati sono disponibili per l'analisi e la visualizzazione in tempo reale e batch.
- **Data Visualization Layer**: composto da Grafana, si occupa della visualizzazione dei dati elaborati ottenuti dallo storage layer e della gestione delle notifiche in caso di anomalie rilevate.

3.2 Architettura dei simulatori

Nonostante i simulatori non siano ufficialmente considerati parte integrante del prodotto dalla proponente, il nostro team ha scelto di dedicare alcune risorse alla progettazione di questa componente nell'ambito del progetto didattico. Inoltre, abbiamo deciso di implementare e tenere conto delle possibili logiche dei microcontrollori associati ai sensori IoT, che possono effettuare operazioni per rendere più efficiente l'intero sistema.
Nei paragrafi successivi, verrà presentata l'architettura individuata mediante l'utilizzo di diagrammi delle classi e relative descrizioni rapide. Inoltre, saranno motivate le scelte dei design pattern individuati e le decisioni progettuali rilevanti. Successivamente, per ogni classe, saranno illustrati metodi e attributi.

3.2.1 Modulo simulatori sensori

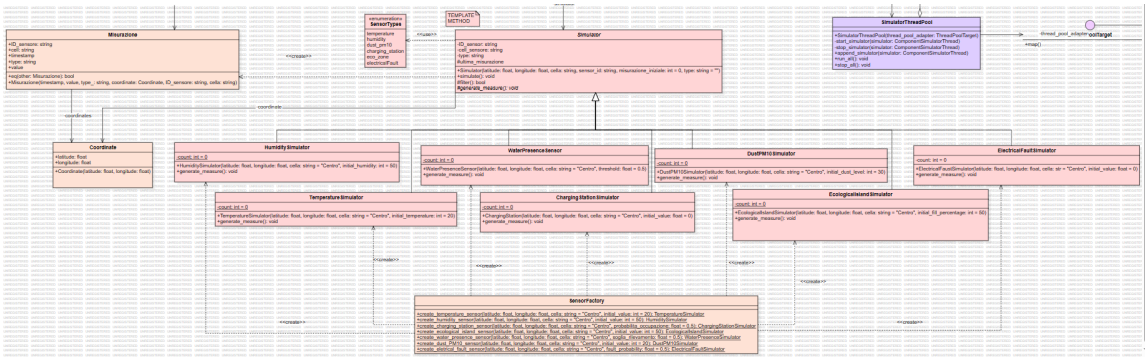

Figure 2: Modulo simulatori sensori - Innovacity

Questo modulo si occupa della generazione di dati di misurazione per diverse tipologie di sensori. In particolare sono stati implementati simulatori per i seguenti tipi di sensori:

- Sensori di temperatura;
- Sensori di umidità;
- Sensori di polveri sottili PM10;
- Sensori di stato occupazione colonnine di ricarica;
- Sensori di stato riempimento isole ecologica;
- Sensori di presenza d'acqua;
- Sensori di guasto elettrico.

Design pattern Template Method:

La classe astratta *Simulator* implementa il design pattern *Template Method*. Il metodo *simulate()* fornisce lo scheletro dell'algoritmo per la generazione e la gestione delle misurazioni. Le classi concrete che estendono *Simulator* implementano:

- **Metodo *generate_measure()***: per la generazione semi randomica della misurazione associata al tipo di sensore;
- **Metodo *filter()***: per la logica di filtrazione di misurazioni errate o non attendibili (ad esempio, negative, fuori range o consecutive troppo distanti). Il metodo *filter()* offre un'implementazione di default che lascia passare ogni misurazione senza modifiche.

Il design pattern *Template Method* è stato scelto per:

- Permettere una facile estensione del sistema con nuovi tipi di sensori che dovranno unicamente implementare la loro logica di generazione delle misurazioni e di filtering se necessario;
- Standardizzare i passi per la generazione delle misurazioni, garantendo coerenza e manutenibilità del codice;
- Ridurre la duplicazione del codice.

Una volta ottenuto lo stato del sensore, esso viene inserito in un oggetto di tipo *Misurazione*. Questo oggetto contiene informazioni di contesto come:

- Identificativo del sensore;
- Cella della città in cui è presente;
- Timestamp della misurazione;
- Valore della misurazione;
- Coordinate;
- Tipologia di misurazione.

L'oggetto *Misurazione* viene poi ritornato al chiamante che si occuperà di inviarlo al server *Kafka*. Un oggetto di tipo *Simulator* verrà assegnato ad ogni *SimulatorThread* che chiamerà ad intervalli regolari il metodo *simulate()* ottenendo appunto la misurazione che invierà al server *Kafka* tramite un modulo apposito e indipendente.

Design pattern Factory:

SensorFactory implementa il design pattern *Factory* per la creazione di simulatori dei sensori. Il pattern FACTORY è un pattern di tipo "Creazionale" secondo la classificazione della GoF. I pattern di tipo creazionali si occupano della costruzione delle simulazioni dei sensori e delle problematiche che si possono originare, astraggono il processo di creazione degli oggetti, nascondono i dettagli della creazione e rendono i sistemi indipendenti da come gli oggetti sono creati e composti. Il pattern Factory incapsula la creazione concreta dei sensori, consentendo al client (l'utilizzatore) di non conoscere i dettagli.

Classi: metodi e attributi

- **Classe astratta: *Simulator***
  - **Attributi**:
    - **ID_sensor:str [private]** - Identificatore univoco del sensore.
    - **cella_sensore:str [private]** - Identificatore della cella del sensore.
    - **coordinate:coordinate [private]** - Coordinate geografiche del sensore.
    - **misurazione: T [protected]** - Misurazione corrente del sensore.
    - **type:str [private]** - Tipo di sensore.
  - **Metodi**:
    - **simulate():Misurazione [public]** - Metodo principale per simulare la generazione di una misurazione. Si basa sul design pattern Template Method:
      1. Chiama generate_measure() per generare un valore di misurazione.
      2. Verifica con filter() se la misurazione è valida (ripete la generazione finché non lo è).
      3. Restituisce un oggetto Misurazione con data e ora corrente, valore misurato, tipo di sensore, coordinate e identificativo del sensore.
    - **generate_measure():None [protected]** - Metodo astratto da implementare nelle classi concrete per generare un valore di misurazione semi-casuale coerente con la tipolgia di sensore da salvare nell'attributo *misurazione*.
    - **filter():bool [protected]** - Metodo di filtro per la validazione della misurazione (implementazione di default che accetta sempre la misurazione). Può essere ridefinito nelle classi concrete per implementare la logica di filtraggio.
  - **Note**:
    - La classe Simulator è astratta e definisce il comportamento generale della simulazione della misurazione.
    - Le classi concrete che ereditano da Simulator devono implementare il metodo astratto generate_measure().
    - Il metodo filter() può essere ridefinito nelle classi concrete per implementare la logica di validazione specifica del sensore.
- **Enumerazione: *SensorTypes***
  - **Costanti**:
    - **TEMPERATURE:str [public]** - Rappresenta la nomenclatura dei sensore di temperatura.
    - **HUMIDITY:str [public]** - Rappresenta la nomenclatura dei sensore di umidità.

        * **DUST_PM10:str [public]** - Rappresenta la nomenclatura dei sensore di "polvere PM10".
        * **CHARGING_STATION:str [public]** - Rappresenta la nomenclatura dei sensore di stato delle colonnine di ricarica.
        * **ECOLOGICAL_ISLAND:str [public]** - Rappresenta la nomenclatura dei sensore di stato riempimento isole ecologica.
        * **WATER_PRESENCE:str [public]** - Rappresenta la nomenclatura dei sensore di presenza d'acqua.
        * **ELECTRICAL_FAULT:str [public]** - Rappresenta la nomenclatura dei sensore di guasti elettrici.
    - **Note**:
        * L'enumerazione viene utilizzata per centralizzare la gestione della nomenclatura dei tipi di sensori che verrà salvata nelle misurazioni.

- **Classe: *TemperatureSimulator***
    - **Attributi:**
        * **count:int [private, static]** - Contatore statico per generare un ID univoco per ogni istanza.
    - **Metodi**:
        * **generate_measure():None [protected]** - Genera una misurazione di temperatura semi-casuale e aggiorna la misurazione corrente.
    - **Note**:
        * La classe TemperatureSimulator è una classe concreta che eredita dalla classe astratta Simulator.
        * Il costruttore genera automaticamente un ID sensore univoco per ogni istanza.

- **Classe: *HumiditySimulator***
    - **Attributi:**
        * **count:int [private, static]** - Contatore statico per generare un ID univoco per ogni istanza.
    - **Metodi**:
        * **generate_measure():None [protected]** - Genera una misurazione di umidità semi-casuale e aggiorna la misurazione corrente.
    - **Note**:

        - La classe HumiditySimulator è una classe concreta che eredita dalla classe astratta Simulator.
        - Il costruttore genera automaticamente un ID sensore univoco per ogni istanza.

- **Classe: *ChargingStationSimulator***
    - **Attributi**:
        - **count:int [private, static]** - Contatore statico per generare un ID univoco per ogni istanza.
    - **Metodi**:
        - **generate_measure():None [protected]** - Genera lo stato della colonnina di ricarica (Occupato: True, Libero: False) basata su una probabilità di transizione.
    - **Note**:
        - La classe ChargingStationSimulator è una classe concreta che eredita dalla classe astratta Simulator.
        - Implementa il metodo astratto generate_measure() per generare una misurazione basata sulla probabilità di transizione.
        - Il costruttore genera automaticamente un ID sensore univoco per ogni istanza.

- **Classe: *DustPM10Simulator***
    - **Attributi**:
        - **count:int [private, static]** - Contatore statico per generare un ID univoco per ogni istanza.
    - **Metodi**:
        - **generate_measure():None [protected]** - Genera una variazione di polvere PM10 semi-casuale e aggiorna la misurazione corrente.
    - **Note**:
        - La classe DustPM10Simulator è una classe concreta che eredita dalla classe astratta Simulator.
        - Il costruttore genera automaticamente un ID sensore univoco per ogni istanza.

- **Classe: *ElectricalFaultSimulator***
    - **Attributi**:
        - **count:int [private, static]** - Contatore statico per generare un ID univoco per ogni istanza.
    - **Metodi**:

        - **generate_measure():None [protected]** - Genera lo stato di una centralina elettrica (Guasto verificato: True, Operativa: False) basata sulla probabilità di guasto.
    - **Note**:
        - La classe ElectricalFaultSimulator è una classe concreta che eredita dalla classe astratta Simulator.
        - Il costruttore genera automaticamente un ID sensore univoco per ogni istanza.

- **Classe: *EcologicalIslandSimulator***
    - **Attributi**:
        - **count:int [private, static]** - Contatore statico per generare un ID univoco per ogni istanza.
    - **Metodi**:
        - **generate_measure():None [protected]** - Genera una misurazione della percentuale di riempimento di un isola ecologica.
    - **Note**:
        - La classe EcologicalIslandSimulator è una classe concreta che eredita dalla classe astratta Simulator.
        - Il costruttore genera automaticamente un ID sensore univoco per ogni istanza.

- **Classe: *WaterPresenceSensor***
    - **Attributi**:
        - **count:int [private, static]** - Contatore statico per generare un ID univoco per ogni istanza.
    - **Metodi**:
        - **generate_measure():None [protected]** - Genera una misurazione basata sulla soglia di presenza dell'acqua (Acqua rilevata: True, Acqua non rilevata:False).
    - **Note**:
        - La classe EcologicalIslandSimulator è una classe concreta che eredita dalla classe astratta Simulator.
        - Il costruttore genera automaticamente un ID sensore univoco per ogni istanza.

- **Classe: *Misurazione***
    - **Attributi**:
        - **timestamp: datetime [private]** - Timestamp della misurazione.

    - **value: T [private]** - Valore della misurazione.
    - **type: str [private]** - Tipo della misurazione.
    - **coordinates: coordinate [private]** - Coordinate della misurazione.
    - **ID_sensore: str [private]** - ID del sensore che ha effettuato la misurazione.
    - **cella: str [private]** - Cella in cui è stata effettuata la misurazione.
  - **Metodi**:
    - **__eq__(other:Misurazione):bool [public]** - Ridefinizione dell'operatore di uguaglianza per confrontare due oggetti Misurazione.

- **Classe: *coordinate***
  - **Attributi**:
    - **latitude:float [private]** - Latitudine della coordinata.
    - **longitude:float [private]** - Longitudine della coordinata.
  - **Metodi**:
    - **__eq__(other:coordinate):bool [public]** - Ridefinizione dell'operatore di uguaglianza per confrontare due oggetti Coordinate.

- **Classe: *SensorFactory***
  - **Metodi**:
    - **create_temperature_sensor(latitude: float, longitude: float, cella: str, initial_value:float):TemperatureSimulator [public, static]** - Crea un simulatore di temperatura.
    - **create_humidity_sensor(latitude: float, longitude: float, cella: str, initial_value:float):HumiditySimulator [public, static]** - Crea un simulatore di umidità.
    - **create_charging_station_sensor(latitude: float, longitude: float, cella: str, probabilita_occupazione:float):ChargingStationSimulator [public, static]** - Crea un simulatore di stazione di ricarica.
    - **create_ecological_island_sensor(latitude: float, longitude: float, cella: str, initial_value:float):EcologicalIslandSimulator [public, static]** - Crea un simulatore di isola ecologica.
    - **create_water_presence_sensor(latitude: float, longitude: float, cella: str, soglia_rilevamento:float):WaterPresenceSensor [public, static]** - Crea un sensore di presenza d'acqua.
    - **create_dust_PM10_sensor(latitude: float, longitude: float, cella: str, initial_value:float):DustPM10Simulator [public, static]** - Crea un simulatore di polvere PM10.

* **create_eletrical_fault_sensor(latitude: float, longitude: float, cella: str, fault_probability:float):ElectricalFaultSimulator [public, static]** - Crea un simulatore di guasto elettrico.

Note:

* Implementazione del Pattern Factory;
* Fornisce metodi per la creazione di simulatori di sensori;
* Astrae il processo di creazione dei sensori, nascondendo i dettagli della creazione.

3.2.2 Modulo Writers

??

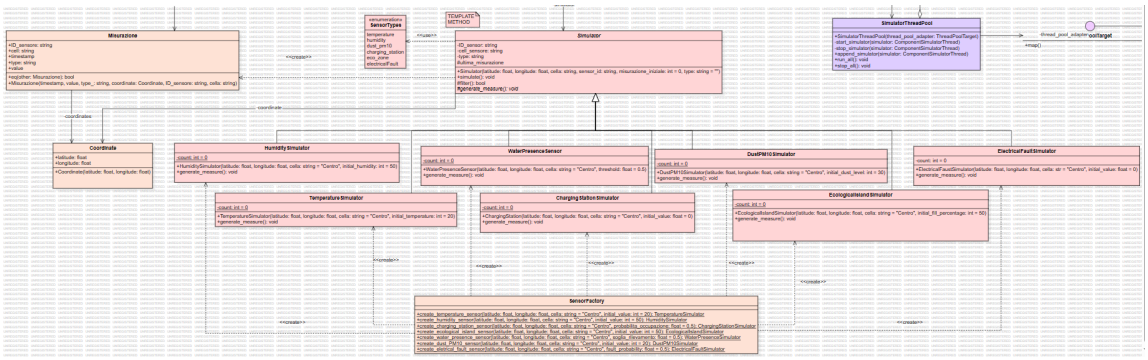

Figure 3: Modulo writers - Innovacity

Questo modulo si occupa della scrittura e/o invio di informazioni a diverse tipologie di servizi e vuole essere completamentemente indipendendente e non influenzato dal modulo della simulazione dei sensori cosi da poter consentire un suo riutilizzo.

Design pattern Strategy + Composite:

Il modulo presenta un interfaccia *Writer* che offre il metodo di scrittura *write()* di oggetti di tipo *Writable*. Questo metodo è implementato da diverse classi concrete che rappresentano i vari servizi a cui è possibile inviare le informazioni. Questo approccio implementa il design pattern *Strategy* per la scrittura dei dati su diverse piattaforme/servizi e il design pattern *Composite* per la gestione di più servizi a cui scrivere contemporaneamente in modo completamentemente indifferenziato dalla scrittura ad un singolo servizio. Nello specifico sono state implentate tre strategie di scrittura:la prima, (*KafkaWriter*), atta a permettere al simulatore di inviare messaggi a Kafka, la seconda (*StdOutWriter*) atta a permettere di

stampare i *Writable*su terminale e la terza (*ListWriter*) per il salvataggio su una lista degli oggetti *Writable*. L'utilizzo del design pattern Composite e Strategy in questo caso ha diverse motivazioni:

- **Gestione uniforme dei servizi**: Il pattern Strategy consente di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. In questo caso, i servizi di scrittura sono trattati come algoritmi intercambiabili, consentendo di scrivere informazioni su diversi servizi senza dover conoscere i dettagli di implementazione di ciascuno.
- **Gestione gerarchica dei servizi**: Il pattern Composite consente di trattare gli oggetti singoli e le loro composizioni (gruppi di oggetti) allo stesso modo. Nel contesto del modulo, potrebbe esserci la necessità di gestire non solo singoli servizi, ma anche gruppi di servizi. Ad esempio, potrebbe essere utile inviare informazioni contemporaneamente a diversi servizi, come un database, un file di log e un servizio di notifica. Il Composite consente di comporre questi servizi in modo gerarchico e trattarli uniformemente.

Design pattern Object Adapter:

Nello specifico, la classe *KafkaWriter* realizza la sua funzionalità attraverso l'utilizzo del design pattern *Adapter*, nella sua variante *Object Adapter*. Tale scelta è stata motivata dall'impiego della classe *Producer* della libreria *confluent_kafka*, la quale potrebbe subire variazioni non controllabili da noi. Per garantire la capacità di rispondere prontamente a tali cambiamenti senza dover modificare la classe *KafkaWriter* o altri parti di sistema, si è optato per l'utilizzo di questo pattern, trasferendo così la complessità derivante da tali modifiche proprio nell'adapter. Inoltre grazie all'interfaccia *KafkaTarget*, si è garantita la possibilità di estendere il sistema con nuovi metodi di scrittura su Kafka o l'utilizzo di nuove librerie senza dover modificare la classe *KafkaWriter* ma solamente aggiungendo una nuova classe adapter che implementi *KafkaTarget*.

Classi: metodi e attributi

- **Interfaccia: *Writable***
  - **Metodi**:
    - **to_json(): [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per convertire l'oggetto in una stringa JSON.
  - **Note**:
    - L'interfaccia *Writable* definisce un insieme di metodi che una classe deve implementare perchè possa essere utilizzata dalle strategie di scrittura.

- **Interfaccia: *Writer***
  - **Metodi:**
    - **write(to_write: Writable): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per scrivere un oggetto Writable.
  - **Note**:
    - L'interfaccia *Writer* definisce un insieme di metodi che una classe deve implementare perchè possa essere utilizzata come strategia di scrittura;
    - Rappresenta l'interfaccia "Component" del pattern *Composite* che descrive le operazioni comuni sia agli elementi semplici che a quelli complessi dell'albero.
- **Classe: *StdoutWriter***
  - **Attributi**:
    - **lock:threading.Lock [private]** - Lock per garantire l'accesso esclusivo alla stampa ed un esecuzione Thread safe.
  - **Metodi:**
    - **write(to_write: Writable): None [public]** - Stampa l'oggetto Writable come stringa JSON nella console;
  - **Note**:
    - La classe è una strategia di scrittura del pattern *Strategy* ma anche la componente "Leaf" del pattern *Composite*, ovvero l'elemento base che non ha sottoelementi.
- **Classe: *ListWriter***
  - **Attributi**:
    - **data_list:list [private]** - Lista per memorizzare gli oggetti Writable.
    - **lock:threading.Lock [private]** - Lock per garantire l'accesso esclusivo alla lista ed un esecuzione Thread safe.
  - **Metodi:**
    - **write(to_write: Writable): None [public]** - Aggiunge l'oggetto Writable alla lista.
    - **get_data_list(): list [public]** - Restituisce la lista di oggetti Writable.
  - **Note**:
    - La classe è una strategia di scrittura del pattern *Strategy* ma anche la componente "Leaf" del pattern *Composite*, ovvero l'elemento base che non ha sottoelementi.

- **Classe: *KafkaWriter***
  - **Attributi**:
    - **lock:threading.Lock [private]** - Lock per garantire l'accesso esclusivo alla scrittura su Kafka ed un esecuzione Thread safe.
    - **kafka_target:KafkaTarget [private]** - Riferimento ad un implementazione di KafkaTarget per effettuare l'effettiva scrittura in Kafka tramite librerie.
  - **Metodi:**
    - **write(to_write: Writable): None [public]** - Scrive l'oggetto Writable come stringa JSON su Kafka.
  - **Note**:
    - La classe è una strategia di scrittura del pattern *Strategy* ma anche la componente "Leaf" del pattern *Composite*, ovvero l'elemento base che non ha sottoelementi.
    - La costruzione dell'oggetto KafkaWriter richiede un riferimento ad un oggetto che implementi l'interfaccia KafkaTarget.
- **Classe: *CompositeWriter***
  - **Attributi**:
    - **writers:Writer* [protected]** - Lista di oggetti Writer.
  - **Metodi:**
    - **add_writer(writer: Writer): CompositeWriter [public]** - Aggiunge un oggetto Writer alla lista di writers.
    - **add_kafkaConfluent_writer(topic: str, host: str, port: int): CompositeWriter [public]** - Crea un KafkaWriter con un KafkaConfluentAdapter e lo aggiunge alla lista di writers.
    - **add_stdOut_writer(): CompositeWriter [public]** - Crea un StdoutWriter e lo aggiunge alla lista di writers.
    - **add_list_writer(writer_list: ListWriter): CompositeWriter [public]** - Aggiunge un ListWriter alla lista di writers.
    - **remove_writer(writer: Writer): None [public]** - Rimuove un Writer dalla lista di writers.
    - **write(to_write: Writable): None [public]** - Chiama il metodo write su ogni Writer nella lista di writers passando come attributo il *Writable* ricevuto.
  - **Note**:

        * La classe è la componente "Composite" del pattern *Composite*, ovvero l'elemento che può avere sottoelementi;
        * Dopo aver ricevuto una richiesta, il contenitore (detto compisite) delega il lavoro ai suoi sottoelementi:foglie o altri contenitori.

- **Interfaccia: *KafkaTarget***
    - **Metodi:**
        * **write_to_kafka(data: str): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per scrivere dati su Kafka.
    - **Note**:
        * La classe è una interfaccia che fornisce un contratto per le operazioni di scrittura e invio a Kafka.
        * Rappresenta il componente Target del pattern *Object Adapter*.

- **Classe: *KafkaConfluentAdapter***
    - **Attributi**:
        * **topic:str [private]** - Il topic su cui scrivere in Kafka.
        * **producer:Producer [private]** - Il producer Kafka per inviare messaggi.
    - **Metodi:**
        * **write_to_kafka(data: str): None [public]** - Scrive i dati su Kafka.
        * **Note**:
            - La classe è un'implementazione concreta dell'interfaccia KafkaTarget, utilizzando la libreria confluent-kafka per interagire con Kafka;
            - Rappresenta il componente "Adapter" del pattern *Object Adapter*.
            - Il Producer kafka rappresenta la componente "service" del pattern *Object Adapter*.

3.2.3 Modulo Threading/Scheduling

Questo modulo si propone di gestire la logica di pianificazione per il recupero dei dati dai simulatori dei sensori e di inviare/scrivere tali dati utilizzando gli "Writers". Funge da orchestratore per i due moduli appena descritti, offrendo la possibilità di configurare la frequenza di campionamento o il numero di misurazioni da eseguire. Inoltre, incorpora una logica di ottimizzazione, simile a quella impiegata dai microcontrollori dei sensori nella realtà, al fine di evitare la trasmissione di dati ridondanti, inviando solo i cambiamenti di stato dei sensori.

Dependecy Inversion principle

Il modulo è stato progettato per rispettare il principio di inversione delle dipendenze. Infatti, la classe *SimulatorThreadPool* è stata progettata per essere indipendente dalla libreria di gestione dei thread utilizzata, consentendo di sostituire la libreria di gestione dei thread senza dover modificare il codice di *SimulatorThreadPool*. Inoltre, i componenti del modulo sono progettati per essere indipendenti dai dettagli di implementazione dei simulatori e dei writers, consentendo di sostituire i simulatori e i writers senza dover modificare il codice.

Design pattern Composite:

Come per il modulo di scrittura anche questo è sviluppato secondo il pattern Composite che permette di gestire un singolo Thread di esecuzione o un gruppo di Thread in modo uniforme.

Design pattern Object Adapter:

Inoltre, considerando l'impiego di più thread per un esecuzione parallela, per delegare l'orchestrazione delle operazioni, si è fatto ricorso alle ThreadPool. Al fine di evitare modifiche dirette al codice di *SimulatorThreadPool*, è stato adottato il pattern *Object Adapter* per adattare la ThreadPool di Python a un'interfaccia comune con cui *SimulatorThreadPool* possa interagire. Questo approccio consente di modificare la logica o la libreria utilizzata per la gestione dei thread senza richiedere modifiche al codice di *SimulatorThreadPool*, ma semplicemente aggiungendo una nuova classe adapter che implementi *ThreadPoolTarget*.

Un'altra implementazione del pattern *Object Adapter* viene impiegata per adattare gli oggetti *Misurazione* del modulo *Simulatori* agli oggetti *Writable* del modulo *Writers*. La classe *AdapterMisurazione*, implementando l'interfaccia *Writable*, fornisce un'implementazione del metodo *to_json()* che consente di convertire un oggetto *Misurazione* nel formato JSON, compatibile e riconosciuto da Kafka. 3.3.2

Classi: metodi e attributi

- **Interfaccia: *ComponentSimulatorThread***
  - **Metodi:**
    - **run(): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per definire il comportamento del thread quando viene avviato.

        - **task(): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per definire il compito specifico che il thread deve eseguire.
        - **stop(): None [public, abstract]** - Metodo astratto che deve essere implementato nelle sottoclassi per definire come fermare il thread.
    - **Note**:
        - Eredita le proprietà e i metodi della classe Thread della *Standard Library*;
        - *ComponentSimulatorThread* è un interfaccia di threading per la simulazione sensori, fornendo un contratto per le operazioni di avvio, esecuzione del compito e arresto;
        - Rappresenta il componente "Component" del pattern *Composite*,descrive le operazioni comuni sia ai singoli Thread sia alle composizioni;
        - L'utilizzatore dei simulatori può lavorare allo stesso modo con elementi semplici (Singoli Thread) o complessi dell'albero (Insiemi di Thread in forma di albero).
- **Classe: *SimulatorThread***
    - **Attributi**:
        - **simulator:Simulator [private]** - Il simulatore da utilizzare per generare i dati.
        - **frequency:float [private]** - La frequenza con cui generare i dati.
        - **is_running:bool [private]** - Flag per controllare se il thread è in esecuzione.
        - **data_to_generate:int [private]** - Il numero di dati da generare.
        - **writers:Writer [private]** - L'oggetto Writer per scrivere i dati generati. (Singolo o albero)
    - **Metodi:**
        - **run(): None [public]** - Avvia il thread del simulatore.
        - **task(): None [public]** - Definisce il compito specifico che il thread deve eseguire, contiene la logica per generare il numero di misurazioni richieste con l'intervallo specificato alla costruzione. Inoltre evita l'invio di misurazioni consecutive uguali cosi da evitare ulteriore sovraccarico di dati ridondanti e deducibili inviando agli Writers solo i cambi di stato del sensore da cui acquisisce la misurazione. All'interno del metodo la misurazione restituita dal simulatore viene adattata ad un oggetto *Writable* tramite *AdapterMisurazione* ed inviata agli *Writers*.
        - **stop(): None [public]** - Ferma il thread del simulatore.
    - **Note**:

        - La classe è un'implementazione concreta dell'interfaccia ComponentSimulatorThread, utilizzando un oggetto Simulator per generare dati a una certa frequenza e un oggetto Writer per scrivere i dati generati.
        - Rappresenta il componente Leaf del pattern *Composite*.
        - Se data_to_generate < 0 => Genera misurazioni finchè il thread non viene interroto dall'esterno.
        - Sebbene i simulatori non siano considerati dalla proponente parte del prodotto, la logica di ottimizzazione per inviare solo i cambi di stato dei sensori viene implementata nella realtà IoT, quindi si è deciso di replicarla. Di conseguenza, è stata presa la decisione di replicarla. È importante notare che questa logica non è incorporata nel Simulatore del sensore, il quale ha unicamente il compito semantico di generare dati come un vero sensore. Invece, essa è implementata nel SimulatorThread, il quale agisce in modo simile a un microcontrollore, responsabile sia della gestione dell'intervallo di campionamento che della logica per l'invio delle misurazioni.
        - Nel corso dello sviluppo futuro, potrebbe risultare vantaggioso considerare l'implementazione di un pattern *Strategy* per gestire la strategia/criterio di invio dei dati, che possa distinguere tra un invio continuo e la trasmissione solo in caso di cambiamenti di stato. Tuttavia, al momento della decisione, si è optato per non includerlo al fine di evitare un'eccessiva complessità nell'architettura, nota come sovraingegnerizzazione. Tale scelta è stata dettata dalla volontà di mantenere un equilibrio tra la completezza del sistema e la sua semplicità, favorendo un'implementazione più diretta e immediata delle funzionalità richieste.

- **Classe: *AdapterMisurazione***
    * **Attributi**:
        - **misurazione:Misurazione [private]** - L'oggetto Misurazione da adattare.
    * **Metodi:**
        - **to_json(): dict [public]** - Converte l'oggetto Misurazione in un dizionario JSON conforme a quanto definito in 3.3.2.
        - **from_json(json_data: dict): Misurazione [staticmethod, public]** - Crea un oggetto Misurazione da un dizionario JSON.
    * **Note**:
        - La classe è un'implementazione concreta dell'interfaccia Writable. Fornisce metodi per convertire un oggetto Misurazione in un formato JSON e viceversa.
        - Rappresenta la come componente "Adapter" del pattern *Object Adapter*.

- **Classe: *SimulatorExecutorFactory***
  - **Attributi**:
    - **simulator_executor:SimulatorThreadPool [private]** - L'executor del simulatore per gestire l'esecuzione dei Thread dei simulatori.
  - **Metodi:**
    - **add_simulator(simulator: Simulator, writers: Writer, frequency: float, data_to_generate: int): SimulatorExecutorFactory [public]** - Aggiunge un simulatore all'executor.
    - **add_simulator_thread(thread_simulator: ComponentSimulatorThread): SimulatorExecutorFactory [public]** - Aggiunge un thread di simulatore all'executor.
    - **run(): None [public]** - Avvia tutti i simulatori nell'executor.
    - **stop(): None [public]** - Ferma tutti i simulatori nell'executor.
    - **task(): None [public]** - Avvia tutti i simulatori nell'executor.
  - **Note**:
    - La classe è un'implementazione concreta dell'interfaccia ComponentSimulatorThread, utilizzando un oggetto SimulatorThreadPool per gestire l'esecuzione di vari simulatori.
    -

- **Classe: *SimulatorThreadPool***
  - **Attributi**:
    - **simulators:List[ComponentSimulatorThread] [private]** - La lista dei *ComponentSimulatorThread* da eseguire. (Singoli Thread o alberi di Thread)
    - **thread_pool:ThreadPoolTarget [private]** - Thread pool per gestire l'esecuzione parallela dei simulatori.
  - **Metodi:**
    - **run_all(): None [public]** - Avvia tutti i simulatori nel thread pool, utilizzando l'interfaccia fornita da *ThreadPoolTarget* per l'esecuzione controllata di attività in parallelo.
    - **stop_all(): None [public]** - Ferma tutti i simulatori nel thread pool, utilizzando l'interfaccia fornita da *ThreadPoolTarget* per l'esecuzione controllata di attività in parallelo..
    - **append_simulator(simulator: ComponentSimulatorThread): None [public]** - Aggiunge un *ComponentSimulatorThread* al thread pool.
    - **start_simulator(simulator: ComponentSimulatorThread): None [private,static]** - Avvia un *ComponentSimulatorThread*.

        - **stop_simulator(simulator: ComponentSimulatorThread): None [private,static]** - Ferma un *ComponentSimulatorThread*.
    - **Note**:
        - La classe gestisce un pool di thread per l'esecuzione di vari simulatori, utilizzando un oggetto *ThreadPoolTarget* per gestire l'esecuzione dei simulatori;
        - I metodi *run_all()* e *stop_all()* utilizzano l'interfaccia fornita da *ThreadPoolTarget* per mappare rispettivamente la funzione statica *start_simulator()* e *stop_simulator()* per ogni *ComponentSimulatorThread* in *simulators*.
        - Grazie all'utilizzo di *ThreadPoolTarget* è possibile estendere il sistema con nuovi metodi di esecuzione controllata di attività in parallelo o l'utilizzo di nuove librerie senza dover modificare la classe *SimulatorThreadPool* ma solamente aggiungendo una nuova classe adapter che implementi *ThreadPoolTarget*.
- **Classe: *ThreadPoolTarget***
    - **Metodi:**
        - **map(func, iterable): [abstractmethod]** - Un metodo astratto che deve essere implementato nelle sottoclassi. Questo metodo applica la funzione 'func' a ogni elemento nell''iterable'.
    - **Note**:
        - L'interfaccia rappresenta la componente "Target" del pattern *Object Adapter* fornendo un contratto per le operazioni di esecuzione controllata di attività in parallelo.

- **Classe: *ThreadPoolExecutorAdapter***
    - **Attributi**:
        - **executor:concurrent.futures.ThreadPoolExecutor [private]** - L'executor della thread pool per gestire l'esecuzione dei thread.
    - **Metodi:**
        - **map(func, iterable): [public]** - Applica la funzione 'func' a ogni elemento nell' 'iterable' utilizzando l'executor del thread pool.
    - **Note**:
        - La classe è un'implementazione concreta dell'interfaccia ThreadPoolTarget, utilizzando un oggetto concurrent.futures.ThreadPoolExecutor per gestire l'esecuzione dei thread.

* Rappresenta il componente "Adapter" del pattern *Object Adapter*.
* Adatta l'oggetto ThreadPoolExecutor dalla libreria *concurrent.futures*
* Al momento della costruzione deve essere fornito il parametro intero "workers" ovvero Il numero massimo di thread che è possibile utilizzare per eseguire le task indicate.

3.3 Kafka

3.3.1 Kafka topic

I topic in Kafka possono essere considerati come le tabelle di un database, utili per separare logicamente diversi tipi di messaggi o eventi che vengono inseriti nel sistema. Noi li utilizziamo per separare le diverse misurazioni dei sensori, quindi per ogni tipo di sensore è presente un topic dedicato. Ciò ci consente di creare all'interno di ClickHouse delle "tabelle consumatrici" che acquisiscono automaticamente i dati. Questo è possibile grazie alla separazione logica dei topic, che garantisce che tutti i messaggi all'interno di ciascun topic abbiano lo stesso formato.

3.3.2 Formato messaggi

La struttura di un messaggio contenente le informazioni della misurazione è la seguente in formato Json:

```
{
  "timestamp": "AAAA-MM-DD HH:MM:SS.sss",
  "value": "Valore della misurazione",
  "type": "Tipologia Simulatore",
  "latitude": "Latitudine",
  "longitude": "Longitudine",
  "ID_sensore": "ID sensore",
  "cella": "Partizione della citt\‘{a} dove \‘{e} presente il sensore"
 }
```

Mentre la struttura di un messaggio contenente le informazioni di una misurazione del punteggio di salute è la seguente in formato Json:

```
{
  "timestamp": "AAAA-MM-DD HH:MM:SS.sss",
  "value": "Valore della misurazione",
  "type": "Tipologia Simulatore",
  "cella": "Cella relativa al punteggio di salute"
```

```
6   }
```

Sebbene le misurazioni vengano divise in topic diversi a seconda della tipoligia di sensore che ha effettuato la misurazione si è comunque deciso di inviare e salvare il campo della tipoligia di misurazione per i seguenti motivi:

- **Backup e ripristino dei dati:** Se per qualche motivo si dovesse perdere la struttura dei topic o occorre ripristinare i dati in un altro sistema, il campo type può aiutare a identificare il tipo di sensore che ha effettuato la misurazione, anche se i dati sono stati conservati insieme in un unico topic.
- **Flessibilità futura:**
  - Potrebbero sorgere esigenze future che richiedono l'analisi dei dati provenienti da diversi tipi di sensori all'interno dello stesso topic. In questo caso, il campo type sarebbe utile per distinguere le misurazioni provenienti da sensori diversi;
  - includere il campo type potrebbe essere particolarmente utile se si prevede di supportare diverse unità di misura per una stessa tipologia di sensore in futuro. Ad esempio, potrebbe essere necessario gestire misurazioni di temperatura in gradi Celsius, Fahrenheit o Kelvin. In tal caso, includendo il campo type, si può associare ad ogni misurazione l'unità di misura corretta.

3.3.3 Kafka patterns

Pattern di Pub/Sub

- **Descrizione:** Il pattern Pub/Sub (Publish/Subscribe) permette ai producer di inviare messaggi a topic e ai consumer di ricevere messaggi da tali topic.
- **Funzione in Kafka:** Decoupling tra producer e consumer, favorendo la scalabilità e l'asincronia.
- **Esempio:** Un sensore invia dati a Kafka come producer. I dati vengono pubblicati su un topic specifico, e più consumer, come un'applicazione di analisi in tempo reale e un sistema di archiviazione, si iscrivono al topic.

Partizionamento

- **Descrizione:** Distribuisce i messaggi su più partizioni all'interno di un topic per migliorare la scalabilità e le prestazioni.
- **Funzione in Kafka:** Permette di distribuire il carico di lavoro su più broker e di aumentare la resilienza ai guasti.

- **Esempio:** I dati di un sensore possono essere partizionati in base al tipo di sensore o alla posizione geografica.

Replicazione

- **Descrizione:** Duplica i dati su più broker per garantire la disponibilità e la tolleranza ai guasti.
- **Funzione in Kafka:** I messaggi vengono replicati su un numero configurabile di broker per massimizzare la ridondanza.
- **Esempio:** Se un broker fallisce, i dati sono ancora disponibili su altri broker.

Leader Election

- **Descrizione:** Algoritmo per eleggere un leader per ogni partizione, responsabile dell'ordinamento e della replica dei messaggi.
- **Funzione in Kafka:** Garantisce la coerenza dei dati e la gestione efficiente delle partizioni.
- **Esempio:** Un leader viene eletto per ogni partizione del topic, garantendo che solo un broker riceva e replichi i messaggi per quella partizione.

Log Compaction

- **Descrizione:** Rimuove i messaggi obsoleti da un topic per ottimizzare l'utilizzo dello storage.
- **Funzione in Kafka:** Le vecchie versioni dei messaggi vengono eliminate dopo un periodo di tempo configurabile.
- **Esempio:** I messaggi di sensore con valori vecchi possono essere compattati per risparmiare spazio di archiviazione.

Altri Pattern

Oltre a quelli sopra elencati, Kafka implementa altri pattern come:

- **Consumer Group**: Raggruppamento di consumer che collaborano per ricevere messaggi da un topic.
- **Coordinated Commit**: Meccanismo per garantire che tutti i consumer in un gruppo ricevano correttamente tutti i messaggi di una partizione.

- **Rate Limiting**: Controllo del numero di messaggi che possono essere inviati o ricevuti da un topic in un determinato intervallo di tempo.
- **Dead Letter Queue (DLQ)**: Coda speciale dove vengono inviati i messaggi che non possono essere elaborati correttamente.
- **Monitoring & Metrics**: Fornisce un'ampia gamma di metriche per monitorare le prestazioni e l'utilizzo del sistema.

Conclusione

L'utilizzo di questi design pattern rende Kafka una piattaforma di messaggistica robusta, scalabile e affidabile per una varietà di casi d'uso. L'implementazione di questi pattern permette di ottenere un'architettura efficiente e performante per l'elaborazione dati in streaming.

3.4 Faust - Processing Layer

3.4.1 Introduzione

Premessa

Per soddisfare il requisito opzionale del calcolo del punteggio di salute, si è scelto di utilizzare Faust, una libreria *Python*$_G$ ispirata al modello di *Kafka*$_G$ Streams. Faust facilita l'elaborazione di flussi di dati distribuiti in tempo reale, rendendola ideale per questo caso d'uso. Offre un'interfaccia di alto livello che astrae le complessità di *Kafka*$_G$, rendendo la raccolta dati semplice e intuitiva. Inolltre Faust è progettato per essere scalabile e può essere utilizzato per gestire grandi volumi di dati.

Calcolo del Punteggio

Il punteggio di salute rappresenta un indicatore sintetico del benessere generale di una città, misurandolo in base a diversi aspetti chiave. In questo caso, le tre tipologie di misurazioni considerate sono:

- Temperatura;
- Umidità;
- Livello di polveri sottili (PM10).

Il calcolo del punteggio avviene in due fasi:

1. **Incrementi**:

- A intervalli regolari, si calcolano incrementi al punteggio di salute basandosi sulle misurazioni acquisite nell'intervallo precedente.
- Ciascuna tipologia di misurazione ha un suo algoritmo di calcolo dell'incremento, basato su soglie predefinite di benessere.

2. **Punteggio Finale**:
   - Il punteggio di salute finale si ottiene sommando gli incrementi calcolati per le tre tipologie di misurazioni.
   - Punteggi più alti indicano un minore stato di benessere, con la necessità di interventi per migliorare la qualità della vita.

3.4.2 Componenti Faust & Processing Layer

- **Applicazione Faust:**

```
1       faust.App(<nome_app>, \textit{broker}\textsubscript{\textit{G}}=<
            broker_kafka>)
```

  - Un'applicazione Faust è un *programma*$_G$ *Python*$_G$ che elabora flussi di dati in tempo reale da *Kafka*$_G$.
  - **nome_app**: Identifica l'applicazione.
  - **broker_kafka**: Indirizzo del *broker*$_G$ *Kafka*$_G$ (hostname:porta).

- **Topic:**

```
1       app.topic(<nome_topic>, value_type=<tipo_dato>)
```

  - **nome_topic**: Nome del topic *Kafka*$_G$.
  - **tipo_dato:** Classe che rappresenta il tipo di dato del topic (es. FaustMeasurement).
  - Nel caso si voglia aggiungere altri topic da cui consumare dati basterà aggiungerne prima del parametro value_type.

- **Tipo di dato atteso:**

```
1       class FaustMeasurement(faust.Record)
```

  - È una classe che eredita da **faust.Record**.
  - **faust.Record** è una classe fornita dalla libreria Faust che semplifica la definizione di record per la rappresentazione dei dati in streaming.

  - Rappresenta una singola misurazione proveniente da un *sensore*$_G$. Viene usata nella applicazione Faust per definire il tipo di dati atteso nei topic *Kafka*$_G$.

- **Modello per il calcolo del punteggio di salute:**:
  - **Processore di misurazioni**: Tramite il *pattern*$_G$ *Object Adapter* e l'interfaccia *Processor* l'app faust invia le misurazioni ottenute dai topic al modello per il calcolo del punteggio di salute che verrà adattato come *Processor.*

- **Agente di elaborazione**:

```
@app.agent(<topic>).
```

  - Funzioni asincrone che elaborano i dati dai topic;
  - Ricevono un iteratore di oggetti del tipo specificato per il topic;
  - Eseguono l'elaborazione desiderata su ogni misurazione.

- **Interfaccia Processor**:
  - Per l'incapsulamento di logiche di elaborazione.
  - Utilizzata dagli agenti di elaborazione per inviare le misurazioni al modello per il calcolo del punteggio di salute.

- **Task aggiuntivo** (opzionale):

```
@app.task()
```

  - Definisce una funzione eseguita una sola volta all'avvio dell'applicazione.

3.4.3 Modello per il calcolo del punteggio di salute

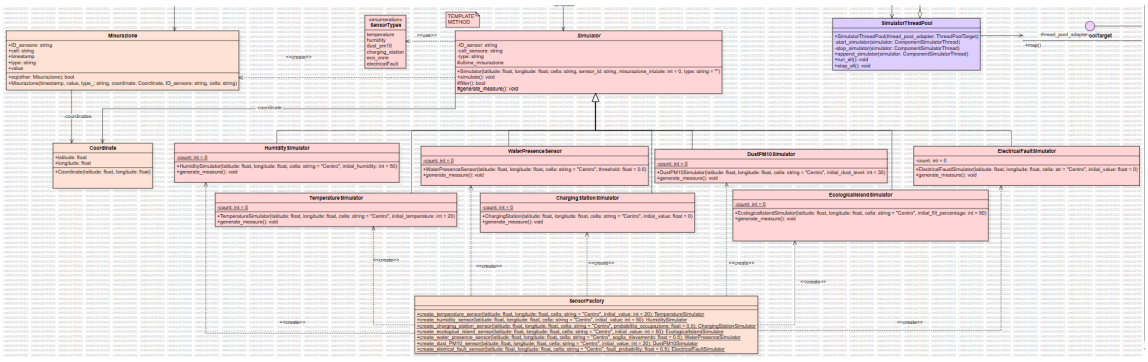

Figure 4: Modulo simulatori sensori - Innovacity

Il processo per il calcolo del punteggio di salute riceve le letture dei sensori attraverso gli agenti di elaborazione dell'applicazione Faust, i quali sono in ascolto sui topic *Kafka*$_G$ relativi alle misurazioni di temperatura, umidità e polveri sottili PM10. Ad intervalli regolari, il *sistema*$_G$ calcola il punteggio di salute della città basandosi su tali misurazioni. Una volta effettuato il calcolo, il risultato è reso disponibile in un topic *Kafka*$_G$ dedicato. Il modello che racchiude la logica per il calcolo, richiamato a intervalli regolari, è quello attualmente preso in esame.
In sintesi, il modello:

- Riceve le misurazioni di temperatura, umidità e polveri sottili dall'agente dell'app Faust.
- Riceve da un thread la richiesta ad intervalli regolari di calcolare i punteggi di salute per le celle della città con le misurazioni ottenute in tempo reale.

In accordo con l'*architettura*$_G$ esagonale, la logica del modello è completamente disaccoppiata dai suoi utilizzatori, i quali interagiscono con il modello tramite specifiche classi adapter. Questo approccio promuove la separazione delle preoccupazioni e favorisce la modularità del *sistema*$_G$. Gli adapter fungono da ponte tra il modello e gli utilizzatori, consentendo una comunicazione fluida e senza dipendenze dirette. Così, eventuali cambiamenti nella logica del modello possono essere implementati senza influenzare gli utilizzatori, garantendo una maggiore flessibilità e manutenibilità del *sistema*$_G$ nel suo complesso.
Il presente modulo è concepito per fornire la logica relativa al puro calcolo del punteggio di salute della città. Tale calcolo si basa su un modello che tiene conto delle misurazioni di temperatura, umidità e polveri sottili PM10. Il modello è stato progettato al fine di determinare un punteggio di salute per ciascuna cella della città in cui sono presenti misurazioni delle suddette tipologie.

Design Pattern Strategy

Il modello per il calcolo del punteggio di salute è stato ideato mediante l'utilizzo del design *pattern*$_G$ Strategy. Tale *pattern*$_G$ consente di definire una famiglia di algoritmi, di incapsularli e renderli intercambiabili. Ciò permette di variare l'algoritmo impiegato per il calcolo del punteggio di salute senza incidere sui *processi*$_G$ di elaborazione dell'applicazione Faust o sugli altri componenti del *sistema*$_G$. In particolare, l'interfaccia *HealthAlgorithm* stabilisce il contratto che deve essere rispettato da tutti gli algoritmi per il calcolo del punteggio di salute. Inoltre, un'implementazione del *pattern*$_G$ *Strategy* è presente anche negli "Incrementatori". Questi, a partire dalle misurazioni fornite, restituiscono un incremento al punteggio di salute della città. Tale incremento è determinato in base a delle soglie predefinite di temperatura, umidità e polveri sottili PM10, le quali sono definite di default in *health_constants* ma possono essere impostate al momento della costruzione. In particolare, l'interfaccia *Incrementer* specifica il contratto che deve essere rispettato da tutti gli incrementatori. Vengono

implementati tre incrementatori, uno per il calcolo dell'incremento di temperatura, uno per l'umidità e uno per le polveri sottili PM10, come strategie del *pattern*$_G$.

Classi: metodi e attributi

- **Interfaccia: *HealthAlgorithm***
  - **Metodi:**
    - **generate_new_health_score(): List[MisurazioneSalute] [abstractmethod]** - Un metodo astratto che deve essere implementato nelle sottoclassi. Questo metodo dovrebbe generare un nuovo punteggio di salute.
  - **Note**:
    - L'interfaccia definisce il contratto per un algoritmo di salute. Le sottoclassi devono implementare il metodo *generate_new_health_score*;
    - Rappresenta la componente "Strategy" del *pattern*$_G$ omonimo.
    - Per rispettare il Single responsibility principle, noto anche come principio di coesione, è stata divisa dalla logica di buffering delle misurazioni presente nella classe astratta *HealthProcessorBuffer* poiché l'utilizzatore *HealthCalculatorThread* non utilizza i metodi per il buffering.
- **Classe astratta: *HealthProcessorBuffer***
  - **Attributi**:
    - **lista_misurazioni:lista_misurazioni [private]** - Una lista di oggetti Misurazione.
    - **lock:threading.Lock [private]** - Un oggetto lock per gestire l'accesso concorrente alla lista di misurazioni.
  - **Metodi:**
    - **add_misurazione(timestamp, value, type_, latitude, longitude, ID_sensore, cella): None [public]** - Aggiunge una nuova misurazione alla lista di misurazioni.
    - **clear_list(): None [public]** - Svuota la lista di misurazioni.
  - **Note**:
    - La classe astratta definisce un buffer di misurazioni per effettuare il processing su un set di misurazioni. Tale buffer contiene una lista di misurazioni e fornisce metodi per aggiungere misurazioni, ottenere la lista di misurazioni e svuotare la lista.
    - La logica di buffering e quella dell'algoritmo per il calcolo del punteggio di salute vengono separate in due astrazioni per rispettare il principio di Single Responsibility. Gli utilizzatori di questa classe, i *Processor*, sono interessati esclusivamente al metodo per l'invio del dato al buffer.

        - La classe astratta definisce un'interfaccia per la comunicazione con gli utilizzatori esterni al modello.

- **Classe: *HealthCalculator***
  - **Attributi**:
    - **tmpInc:TemperatureIncrementer [private]** - Utilizzato per il calcolo dell'incremento di temperatura;
    - **umdInc:HumidityIncrementer [private]**; - Utilizzato per il calcolo dell'incremento di umidità;
    - **dstPm10Inc:DustPM10Incrementer [private]** - Utilizzato per il calcolo dell'incremento di PM10;
    - **temperature_measure_type_naming:string [private]** - Nomenclatura dei tipi di misurazione di temperatura.
    - **humidity_measure_type_naming:string [private]** - Nomenclatura dei tipi di misurazione di umidità.
    - **dtsPm10_measure_type_naming:string [private]** - Nomenclatura dei tipi di misurazione di PM10.
    - **healthScore_measure_type_naming:string [private]** - Nomenclatura dei tipi di misurazione di punteggio di salute.
    - **lock [private]** - Un oggetto lock per gestire l'accesso concorrente.
  - **Metodi:**
    - **generate_new_health_score(): List[MisurazioneSalute] [public]** - Genera e restituisce una nuova lista di punteggi di salute, uno per ogni cella della città di cui sono state fornite misurazioni.
    - **calcola_incremento_tmp(cella: str, lista_misurazioni): int [private]** - Calcola e restituisce l'incremento della temperatura.
    - **calcola_incremento_umd(cella: str, lista_misurazioni): int [private]** - Calcola e restituisce l'incremento dell'umidità.
    - **calcola_incremento_dstPm10(cella: str, lista_misurazioni): int [private]** - Calcola e restituisce l'incremento della polvere PM10.
  - **Note**:
    - La classe implementa l'interfaccia *HealthAlgorithm* e la classe astratta *HealthProcessorBuffer* per calcolare il punteggio di salute tramite la strategia concreta definita in *generate_new_health_score()* che genera una nuova lista di punteggi di salute.

        - Questa classe rappresenta il vero cervello del calcolo del punteggio di salute in quanto utilizzatore di tutti gli incrementatori e delle misurazioni bufferizzate per creare una strategia di calcolo.

- **Classe: *Misurazione***
  - **Attributi**:
    - **timestamp:datetime [private]** - Timestamp della misurazione.
    - **value:T [private]** - Valore della misurazione.
    - **type:str [private]** - Tipo della misurazione.
    - **coordinates:coordinate [private]** - Coordinate della misurazione.
    - **ID_sensore:str [private]** - ID_G del $sensore_G$ che ha effettuato la misurazione.
    - **cella:str [private]** - Cella in cui è stata effettuata la misurazione.
  - **Metodi**:
    - **__eq__(other:Misurazione):bool [public]** - Ridefinizione dell'operatore di uguaglianza per confrontare due oggetti Misurazione.

- **Classe: *coordinate***
  - **Attributi**:
    - **latitude:float [private]** - Latitudine della coordinata.
    - **longitude:float [private]** - Longitudine della coordinata.
  - **Metodi**:
    - **__eq__(other:coordinate):bool [public]** - Ridefinizione dell'operatore di uguaglianza per confrontare due oggetti Coordinate.

- **Classe: *MisurazioneSalute***
  - **Attributi**:
    - **timestamp:datetime [private]** - Il timestamp della misurazione di salute.
    - **value:float [private]** - Il valore della misurazione di salute.
    - **type:string [private]** - Il tipo della misurazione.
    - **cella:string [private]** - La cella della misurazione di salute.
  - **Note**:
    - La classe rappresenta una misurazione di salute. Contiene informazioni sul timestamp, il valore (ovvero il punteggio di salute calcolato), il tipo della misurazione e la cella relativa alla misurazione.

- **Classe: *lista_misurazioni***

  - **Attributi**:
    - **list:List[Misurazione] [private]** - Una lista di oggetti Misurazione.
  - **Metodi:**
    - **add_misurazione(timestamp, value, type_, latitude, longitude, ID_sensore, cella): None [public]** - Aggiunge una nuova misurazione alla lista.
    - **clear_list(): None [public]** - Svuota la lista di misurazioni.
    - **get_list_by_cella_and_type(cella: str, tipo_dato: str): List[Misurazione] [public]** - Restituisce una lista di misurazioni che corrispondono alla cella e al tipo di misurazione specificati (temperatura,umidità,ecc.).
    - **get_unique_celle(): List[str] [public]** - Restituisce la lista di celle presenti nelle misurazioni senza ripetzioni.
  - **Note**:
    - La classe rappresenta una lista di misurazioni. Fornisce metodi per aggiungere misurazioni, svuotare la lista, ottenere misurazioni per cella e tipo di misurazioni, e ottenere le celle di cui si hanno misurazioni.

- **Enumerazione: *SensorTypes***
  - **Costanti**:
    - **TEMPERATURE:str [public]** - Rappresenta la nomenclatura dei $sensore_G$ di temperatura.
    - **HUMIDITY:str [public]** - Rappresenta la nomenclatura dei $sensore_G$ di umidità.
    - **DUST_PM10:str [public]** - Rappresenta la nomenclatura dei $sensore_G$ di "polvere PM10".
    - **CHARGING_STATION:str [public]** - Rappresenta la nomenclatura dei $sensore_G$ di stato delle colonnine di ricarica.
    - **ECOLOGICAL_ISLAND:str [public]** - Rappresenta la nomenclatura dei $sensore_G$ di stato riempimento isole ecologica.
    - **WATER_PRESENCE:str [public]** - Rappresenta la nomenclatura dei $sensore_G$ di presenza d'acqua.
    - **ELECTRICAL_FAULT:str [public]** - Rappresenta la nomenclatura dei $sensore_G$ di guasti elettrici.
  - **Note**:
    - L'enumerazione viene utilizzata per centralizzare la gestione della nomenclatura dei tipi di sensori che verrà salvata nelle misurazioni.

- **Interfaccia: *Incrementer***

  - **Metodi:**
    - **get_incrementation(misurazioni: List[Misurazione]): int [abstractmethod]** - Un metodo astratto che deve essere implementato nelle sottoclassi. Questo metodo calcola e restituire un incremento basato sulla lista di misurazioni fornita.
  - **Note**:
    - L'interfaccia definisce il contratto per un incrementatore. Le sottoclassi devono implementare il metodo *get_incrementation()*.
    - Rappresenta la componente "Strategy" del *pattern*$_G$ omonimo.

- **Classe: *TemperatureIncrementer***
  - **Attributi**:
    - **upper_health_soglia:int [private]** - La soglia superiore di benessere per la temperatura;
    - **under_health_soglia:int [private]** - La soglia inferiore di benessere per la temperatura.
  - **Metodi:**
    - **get_incrementation(misurazioni: List[Misurazione]): int [public]** - Calcola e restituisce un incremento basato sulle sole misurazioni di temperatura della lista fornita.
  - **Note**:
    - La classe implementa l'interfaccia *Incrementer*;
    - I valori di default per le soglie vengono presi dall'enumerazione *HealthConstant* altrimenti sono impostabili alla costruzione.
    - Rappresenta una strategia concreta del *pattern*$_G$ *Strategy* per il calcolo dell'incremento di temperatura.

- **Classe: *HumidityIncrementer***
  - **Attributi**:
    - **upper_health_soglia:int [private]** - La soglia superiore di benessere per l'umidità;
    - **under_health_soglia:int [private]** - La soglia inferiore di benessere per l'umidità.
  - **Metodi:**

        - **get_incrementation(misurazioni: List[Misurazione]): int [public]** - Calcola e restituisce un incremento basato sulle sole misurazioni di umidità della lista fornita.
    - **Note**:
        - La classe implementa l'interfaccia *Incrementer*;
        - I valori di default per le soglie vengono presi dall'enumerazione *HealthConstant* altrimenti sono impostabili alla costruzione.
        - Rappresenta una strategia concreta del *pattern$_G$ Strategy* per il calcolo dell'incremento di umidità.

- **Classe: *DustPM10Incrementer***
    - **Metodi:**
        - **get_incrementation(misurazioni: List[Misurazione]): int [public]** - Calcola e restituisce un incremento basato sulle sole misurazioni di polveri sottili della lista fornita.
    - **Note**:
        - La classe implementa l'interfaccia *Incrementer*;
        - Rappresenta una strategia concreta del *pattern$_G$ Strategy* per il calcolo dell'incremento di polveri sottili PM10.
        - A differenza degli altri *Incrementer*, *DustPM10Incrementer* non definisce soglie di benessere in quanto è scontato che il valore ottimale di inquinamento è zero.

3.4.4 Modulo Writer

Il modulo Writer è lo stesso di quello descritto in **??** e viene nella sua totalità riutilizzato per la scrittura dei punteggi di salute calcolati. Non viene riportata la strategia di scrittura su di una lista poichè non ne è stato ritenuto necessario l'utilizzo.

Classi: metodi e attributi

Tutte le informazioni sono già state esposte in: **??**.

3.4.5 Modulo Threading/Scheduling

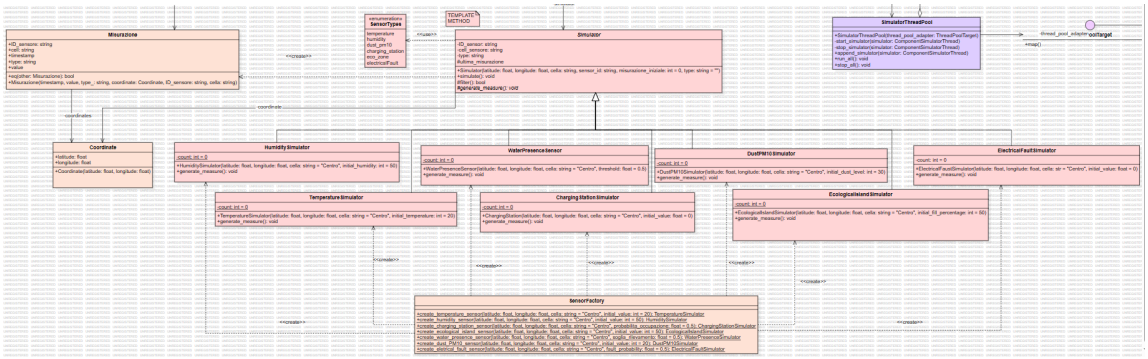

Figure 5: Modulo simulatori sensori - Innovacity

Questo modulo si occupa di integrare la logica di Scheduling e Threading per il calcolo periodico del punteggio di salute della città e quella di scrittura/invio di *Writable*. In particolare, fornisce un'implementazione di un thread che, a intervalli regolari, richiama il calcolo del punteggio di salute della città. Successivamente, utilizzando il modulo "Writer", adatta le misurazioni di salute ottenute all'interfaccia per renderle un oggetto che implementa *Writable*. Pertanto, questo modulo è utilizzatore del modello per il calcolo del punteggio di salute e del modulo di Writer. In sintesi, il modulo:

1. Richiama l'algoritmo per calcolo del punteggio di salute della città a intervalli regolari;
2. Scrive il risultato ottenuto sul topic $Kafka_G$ dedicato.

Dependecy Inversion principle

Il modulo di scheduling e threading dipende dall'astrazione dell'interfaccia del modello per il calcolo del punteggio di salute e del modulo di Writer, invece di dipendere direttamente dalle implementazioni concrete di questi moduli. Ciò consente una maggiore flessibilità e facilità di manutenzione, poiché il modulo di scheduling e threading non è vincolato a implementazioni specifiche, ma può essere facilmente adattato per utilizzare diverse implementazioni che soddisfano lo stesso contratto.
In sostanza, seguendo il principio di Inversione delle Dipendenze, il modulo di scheduling e threading si concentra sull'utilizzo di interfacce o astrazioni, piuttosto che sulle implementazioni concrete, rendendo il $sistema_G$ più modulare, scalabile e facilmente estendibile.

Classi: metodi e attributi

- **Classe: *HealthCalculatorThread***
  - **Attributi**:
    - **healthCalculator: HealthAlgorithm [private]** - Un implementatazione dell'interfaccia *HealthAlgorithm*, ovvero una strategia per il calcolo del punteggio.
    - **frequency: float [private]** - La frequenza con cui il thread genera nuovi punteggi di salute.
    - **is_running: bool [private]** - Un flag che indica se il thread è in esecuzione.
    - **data_to_generate: int [private]** - Il numero di misurazioni di salute da generare.
    - **writers: Writer [private]** - Un oggetto della classe Writer. (Singolo scrittore o albero, Composite *pattern*$_G$)
  - **Metodi:**
    - **run(): None [public]** - Esegue il thread, generando nuovi punteggi di salute a una certa frequenza.
    - **stop(): None [public]** - Ferma l'esecuzione del thread.
  - **Note**:
    - La classe estende la classe *threading.Thread*.
    - Se *data_to_generate* è < 0 genera misurazioni di salute finchè il thread non viene interroto dall'esterno.
    - Grazie al *pattern*$_G$ *Strategy* è possibile cambiare agevolmente l'algoritmo volto al calcolo del punteggio di salute della città.

3.4.6 Modulo Processing

Per garantire un'interfaccia uniforme per i metodi di elaborazione dei dati provenienti da *Kafka*$_G$ tramite Faust e per stabilire un canale di comunicazione con il modello per il calcolo del punteggio di salute, viene sviluppato il modulo di Processing. Questo modulo offre l'interfaccia target denominata *Processor*, e un adapter a *Processor* per l'invio delle misurazioni al modello per il calcolo del punteggio di salute, denominato *HealthModelProcessorAdapter*.

Design Pattern Object Adapter

Nel contesto dell'applicazione Faust, all'interno del ruolo svolto dagli agenti, ogni volta che una misurazione viene ricevuta, viene invocato il metodo *process_measure()* dell'implementazione dell'interfaccia *Processor*, denominata *HealthModelProcessorAdapter*. In

particolare, *HealthModelProcessorAdapter* adatta la classe astratta *HealthModelBuffer*, che rappresenta un buffer di misurazioni utilizzato per eseguire il calcolo periodico del punteggio di salute della città, all'interfaccia *Processor*.
Questo *pattern*$_G$ consente di incapsulare le logiche di elaborazione e di rendere il modello indipendente dall'implementazione specifica dell'applicazione Faust. Allo stesso tempo, facilita la sostituzione dell'operazione di elaborazione eseguita su ogni misurazione dagli agenti grazie al contratto dell'interfaccia *Processor*.

Classi: metodi e attributi

- **Interfaccia: *Processor***
  - **Metodi:**
    - **process(misurazione: FaustMeasurement): None [public, abstract]** - Un metodo astratto che deve essere implementato nelle sottoclassi. Questo metodo elabora una misurazione.
  - **Note**:
    - Le sottoclassi devono implementare il metodo astratto *process()* definendo la propria operazione da effettuare su ogni misurazione ricevuta dai topic di iscrizione.
    - Rappresenta la componente "Target" del *pattern*$_G$ *Object Adapter*.
    - L'interfaccia è stata progettata per garantire un'interfaccia uniforme per i metodi di elaborazione dei dati provenienti da *Kafka*$_G$ tramite Faust.
    - Rappresenta un contratto per l'elaborazione di misurazioni.
    - Gli agenti in ascolto sul topic utilizzeranno un implementatazione di *Processor* per effettuare l'elaborazione delle misurazioni ottenute.
- **Classe: *FaustMeasurement***
  - **Attributi**:
    - **timestamp: str** - Il timestamp della misurazione.
    - **value: float** - Il valore della misurazione.
    - **type: str** - Il tipo della misurazione.
    - **latitude: float** - La latitudine della misurazione.
    - **longitude: float** - La longitudine della misurazione.
    - **ID_sensore: str** - L'*ID*$_G$ del *sensore*$_G$ che ha effettuato la misurazione.
    - **cella: str** - La cella della misurazione.
  - **Note**:

    - La classe *FaustMeasurement* definita utilizzando *faust.Record* rappresenta un singolo record di misurazione proveniente da un *sensore*$_G$ in un'applicazione Faust basata su *Python*$_G$
    - Faust si occupa automaticamente della conversione dei dati in formato JSON in base agli attributi definiti, facilitando la trasmissione e la ricezione dei dati nei topic *Kafka*$_G$.
    - È possibile definire la validazione dei dati in ingresso per garantire l'integrità e la coerenza delle misurazioni.
    - **In sintesi**: Questa classe viene utilizzata in un'applicazione Faust per definire il tipo di dati atteso nei topic *Kafka*$_G$. I dati provenienti dai sensori, contenenti timestamp, valore, tipo, coordinate geografiche, identificativo del *sensore*$_G$ e eventuale cella di appartenenza, verranno convertiti in oggetti di tipo FaustMeasurement prima di essere elaborati dall'applicazione.

- **Classe: *HealthModelProcessorAdapter***
  - **Attributi**:
    - **healthCalculator: HealthProcessorBuffer** - Un implementazione di HealthProcessorBuffer.
  - **Metodi:**
    - **process(misurazione: FaustMeasurement): None [public, async]** - Aggiunge la misurazione all'oggetto *HealthProcessorBuffer* adattando *FaustMeasurement* alla porta di accesso fornita da *HealthProcessorBuffer* per l'elaborazione volta al calcolo del punteggio di salute.
  - **Note**:
    - La classe implementa l'interfaccia *Processor*. Implementa il metodo astratto *process()* per aggiungere/adattare la misurazione del tipo *FaustMeasurement* ad un implementatazione di *HealthProcessorBuffer*.
    - Rappresenta la componente "Adapter" del *pattern*$_G$ *Object Adapter*.

3.5 Configurazione Database

Si è optato per l'utilizzo di ClickHouse per il salvataggio dei dati, le motivazioni sono descritte nella sezione **??**. In particolare, per ogni sensore dei quali si desidera memorizzare i dati, viene creata una tabella che acquisisce i dati dal relativo topic Kafka. Le tipologie di sensori cui misurazioni si vogliono trattare nel progetto sono:

- Sensori di temperatura;

- Sensori di umidità;
- Sensori di rilevamento polveri sottili;
- Sensori stato riempimento isole ecologiche;
- Sensori di stato occupazione colonnine di ricarica;
- Sensori di guasti elettrici;
- Sensori del livello dell'acqua.

La configurazione del database ClickHouse è stata cruciale nella progettazione, poiché un'adeguata ottimizzazione consente di garantire prestazioni ottimali per un sistema orientato al tempo reale e in grado di gestire analisi su enormi volumi di dati.

3.5.1 Funzionalità Clickhouse utilizzate

Materialized Views

Le Materialized Views in ClickHouse sono un meccanismo potente per migliorare le prestazioni delle query e semplificare l'accesso ai dati. Funzionano mantenendo una copia fisica dei risultati di una query di selezione, che viene quindi memorizzata su disco. Questa copia è aggiornata periodicamente in base ai dati sottostanti.

Utilizzi Principali delle Materialized Views

- **Calcolo aggregazioni e popolamento tabelle**:Spesso le delle materialized Views sono state utilizzate per calcolare aggregazioni su dati e quindi popolare altre tabelle con i risultati aggregati. Ad esempio, nel caso specifico in cui una Materialized View calcola la media delle temperature per ogni sensore ogni secondo, i risultati di questa vista possono essere utilizzati per popolare una tabella principale contenente i dati di temperatura aggregati, aggiornando i valori di temperatura medi per ogni sensore ogni secondo;
- **Ottimizzazione delle Prestazioni**: memorizzando i risultati di una query complessa, le Materialized Views consentono di eseguire rapidamente le Query successive senza dover ricalcolare i dati ogni volta. Ciò è particolarmente utile in applicazioni che richiedono interrogazioni frequenti su grandi volumi di dati;
- **Decomposizione delle *Query* Complesse**: le Materialized Views consentono di decomporre query complesse in passaggi più semplici e riutilizzabili, migliorando la leggibilità del codice e semplificando lo sviluppo e la manutenzione delle query.

MergeTree

Link alla documentazione: ClickHouse - MergeTree.
MergeTree è uno dei motori di tabella più potenti e utilizzati in ClickHouse, noto per la sua capacità di gestire e memorizzare grandi volumi di dati in modo efficiente. È una scelta ideale per applicazioni che richiedono l'archiviazione e l'analisi di dati cronologicamente ordinati, come i dati di log o di monitoraggio. L'architettura di MergeTree organizza i dati in parti, ciascuna contenente una serie di punti dati ordinati cronologicamente. Questa organizzazione ottimizzata consente di eseguire rapidamente le query che richiedono l'accesso a dati specifici all'interno di un intervallo di tempo definito, garantendo prestazioni elevate anche su grandi dataset. Oltre alla gestione efficiente dei dati, MergeTree supporta funzionalità avanzate come la compressione dei dati e la gestione automatica delle partizioni. Queste caratteristiche consentono di ottimizzare ulteriormente le prestazioni e la gestione complessiva dei dati, rendendo MergeTree una scelta affidabile per una vasta gamma di scenari di utilizzo in ClickHouse.

Time To Live in ClickHouse

Link alla documentazione: ClickHouse - Implementing a Rollup
In ClickHouse, la funzionalità TTL (Time To Live) è un elemento chiave per gestire grandi volumi di dati in modo efficiente e garantire la pulizia automatica di informazioni obsolete o non più rilevanti.
Quando si specifica il motore Rollup per definire una tabella in ClickHouse, si abilita la creazione di tabelle che supportano il TTL. Questo consente di impostare un periodo temporale dopo il quale i dati saranno eliminati automaticamente dalla tabella. La struttura a Rollup organizza i dati in parti, ciascuna contenente una serie di punti dati ordinati cronologicamente. Il TTL può essere configurato per ciascuna parte dei dati, offrendo un controllo preciso sulla conservazione delle informazioni nel tempo. Questa flessibilità è particolarmente utile per applicazioni che richiedono la conservazione di dati storici per un periodo limitato, come ad esempio i dati di log o di monitoraggio.
Un esempio di come potrebbe può venire utilizzato il motore Rollup per il TTL in ClickHouse è il seguente:

```
TTL toDateTime(timestamp) + INTERVAL 1 MONTH
```

L'uso del TTL di tipo Rollup in questo contesto è cruciale per garantire che la tabella rimanga efficiente e gestibile nel tempo, eliminando automaticamente i dati più vecchi e non più necessari dopo un periodo di tempo specificato. Questo aiuta a ottimizzare le prestazioni complessive del sistema e a gestire in modo efficiente i grandi volumi di dati accumulati nel tempo.

Partition

Link alla documentazione: ClickHouse - Partitioning.
Le partizioni sono una funzionalità fondamentale di ClickHouse che consente di organizzare in modo efficiente e gestire grandi volumi di dati. Questa caratteristica permette di suddividere i dati in gruppi logici in base a criteri specifici, come il valore di una colonna o un intervallo di tempo. Grazie a questa organizzazione ottimizzata, le query che richiedono l'accesso a dati specifici all'interno di una partizione possono essere eseguite rapidamente, garantendo prestazioni elevate anche su dataset di grandi dimensioni.
L'utilizzo delle partizioni nel nostro contesto viene giustificato dall'utilizzo di un TTL (Time To Live), infatti l'utilizzo combinato di queste due funzionalità consente:

- Una gestione efficace dei dati nel tempo;
- Migliori prestazioni del sistema;
- Una semplificazione nella manutenzione del database.

Il partizionamento basato sul timestamp è una pratica comune in ClickHouse, poiché consente di organizzare i dati in partizioni in base al periodo temporale, ad esempio mensilmente. Questo approccio ottimizza l'archiviazione e facilita l'analisi dei dati di serie temporali, come le temperature o i log di eventi. Grazie a questa struttura, le query che coinvolgono dati all'interno di specifici intervalli temporali diventano più efficienti, consentendo un accesso rapido e una migliore analisi dei dati.

Projection

Link alla documentazione:
https://clickhouse.com/docs/en/sql-reference/statements/alter/projection
Le proiezioni memorizzano i dati in un formato che ottimizza l'esecuzione delle *Query*, questa caratteristica è utile per:

- Eseguire *Query* su una colonna che non fa parte della chiave primaria;
- Pre-aggregare colonne, riducendo sia i calcoli che l'I/O.

Puoi definire una o più proiezioni per una tabella e durante l'analisi della *Query* la proiezione con meno dati da esaminare sarà selezionata da ClickHouse senza modificare la *Query* fornita dall'utente.
In generale l'introduzione delle PROJECTIONS produce risultati di notevole importanza, come illustrato di seguito. Consideriamo una tipica query eseguita per l'analisi tramite Grafana:

```
1       SELECT ID_sensore, avgMerge(value) AS value, timestamp
2       FROM innovacity.temperatures
3       WHERE (cella IN ('Arcella')) AND ((timestamp >= toDateTime64
            (1708338633507 / 1000, 3)) AND (timestamp <= toDateTime64
            (1708338933507 / 1000, 3) + INTERVAL 1 DAY))
4       GROUP BY timestamp, ID_sensore
5       HAVING (value >= -100) AND (value <= 100)
6
7       --Query id: 48635435-9b35-4727-b580-9e33a9db92d4
```

Listing 1: Query tipica - Grafana

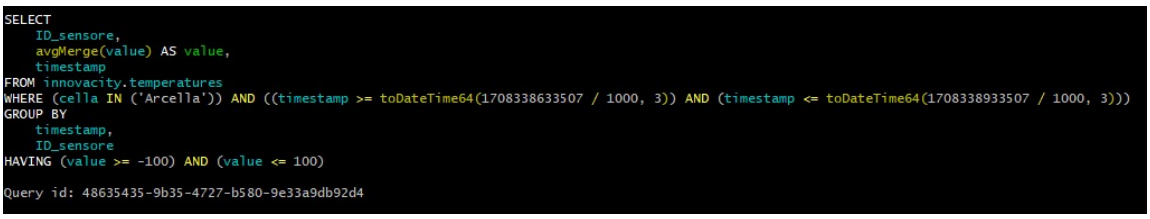

Figure 6: Query tipica - Grafana

Senza l'utilizzo delle PROJECTIONS, il risultato ottenuto è il seguente:

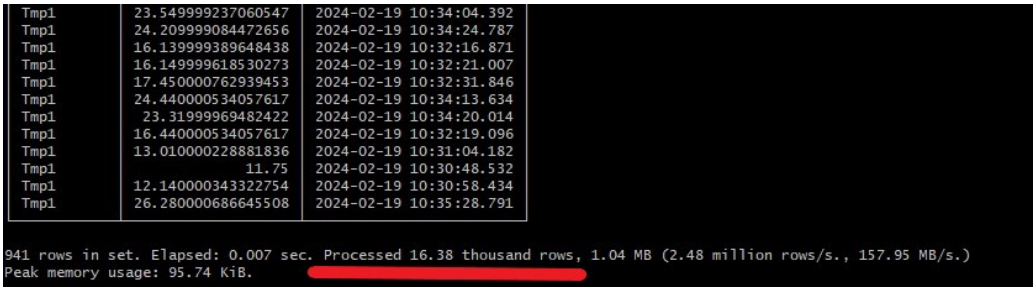

Figure 7: Query tipica risultato senza projections

ovvero sono state processate per ottenere il risultato della query **16,38** migliaia di righe. Invece in seguito all'aggiunta delle PROJECTIONS:

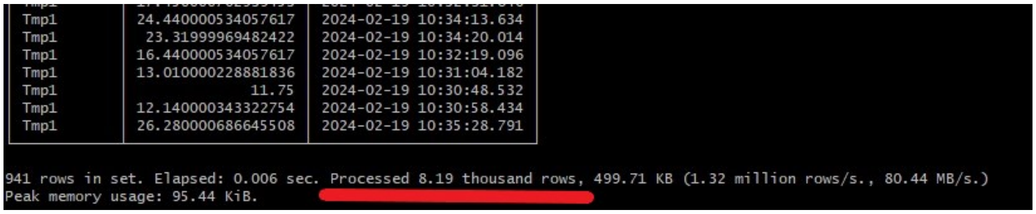

Figure 8: Query tipica risultato con projections

Sono state elaborate approssimativamente 8,19 migliaia di righe per ottenere il risultato della query, circa la metà rispetto al conteggio precedente, evidenziando un miglioramento significativo. Inoltre, mediante un'interrogazione specifica è possibile confermare che le PROJECTIONS sono state effettivamente impiegate per generare il risultato della query in questione.

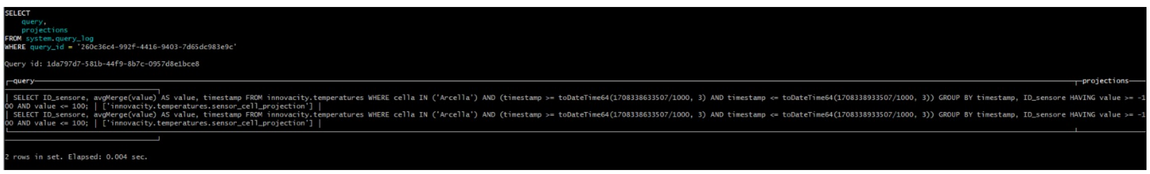

Figure 9: Uso della Projection

Considerando un'altra query eseguita dall'applicativo, che calcola la media globale di **170.000** misurazioni di temperatura, è possibile riconoscere i benefici derivanti dall'utilizzo delle PROJECTIONS. Alla conclusione dell'analisi, è evidente anche il loro effettivo impiego nel calcolo del risultato. Grazie all'adozione delle PROJECTIONS, si ottiene:

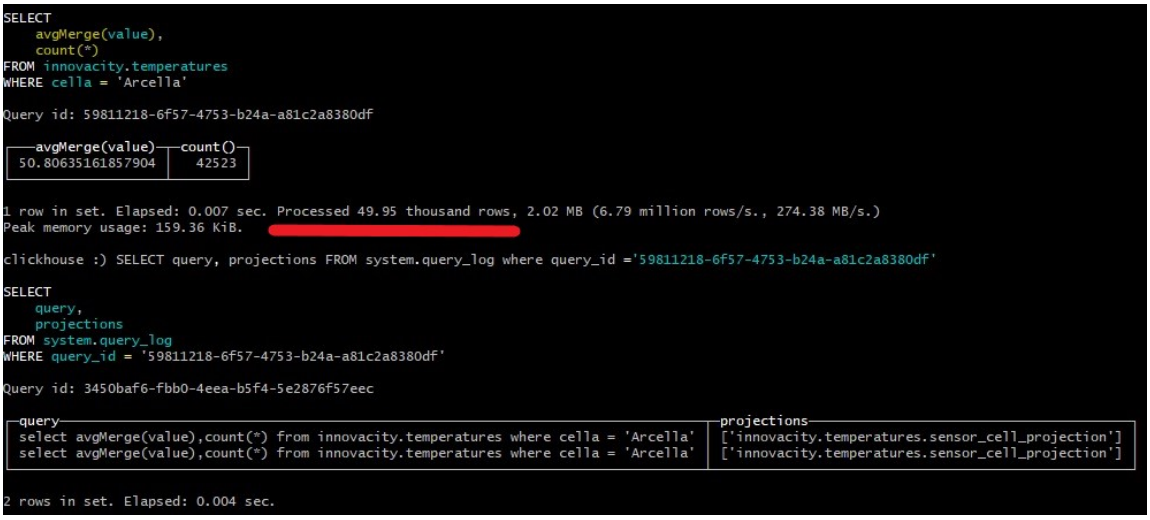

Figure 10: Query esempio Projection 2 - ClickHouse

Ovvero il totale di righe processate per ottenere il risultato è di **49,95 migliaia** con **0,07 secondi** di tempo utilizzati. Si puo notare invece la differenza delle righe processate una volta rimossa la *PROJECTIONS*:

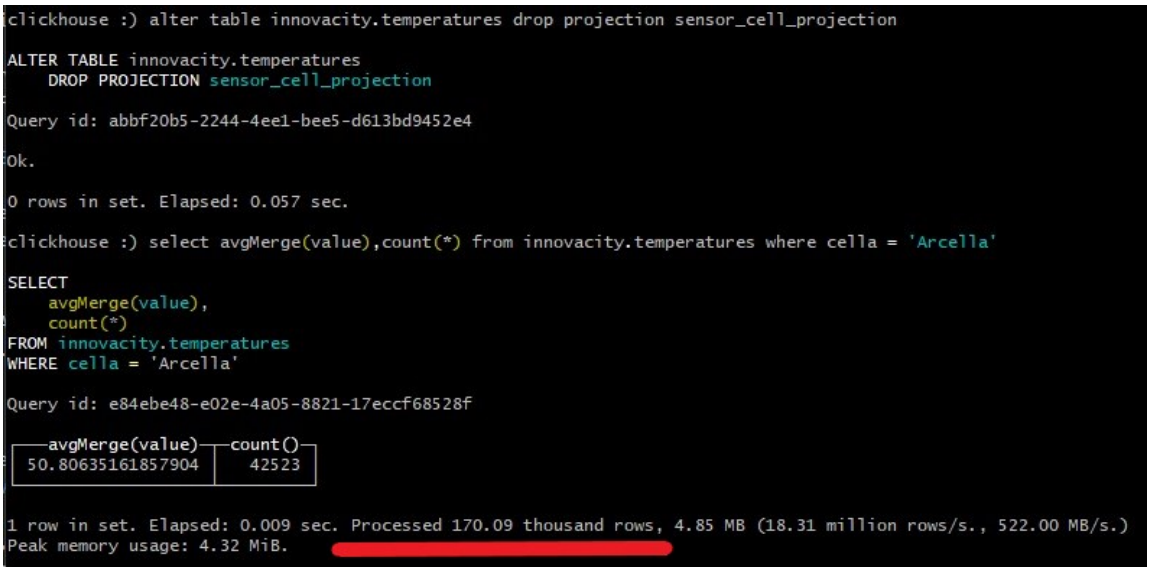

Figure 11: Query esempio senza Projection 2 - ClickHouse

Il totale di righe processate per ottenere il risultato è ora di **170,09 migliaia**, ovvero la totalità delle righe presenti nella tabella, con **0,09 secondi** di tempo utilizzati.

Utilizzo dello spazio su disco

Attenzione: le proiezioni creeranno internamente una nuova tabella nascosta, ciò significa che saranno necessari più I/O e spazio su disco. Ad esempio, se la proiezione ha definito una chiave primaria diversa, tutti i dati dalla tabella originale verranno duplicati.

3.5.2 Integrazione tramite Kafka Engine in ClickHouse

ClickHouse supporta l'integrazione con Kafka tramite Kafka Engine, permettendo la lettura dei dati da un topic Kafka e il loro salvataggio in una tabella ClickHouse. Tale funzionalità riveste un'importanza notevole per applicazioni che richiedono l'elaborazione in tempo reale di dati provenienti da fonti esterne, una necessità frequente nel contesto del monitoraggio urbano. L'integrazione con Kafka consente l'acquisizione e la memorizzazione efficiente dei dati, garantendo prestazioni elevate anche su grandi volumi di dati.
Kafka Engine è progettato per il recupero di dati una sola volta. Ciò significa che una volta che i dati vengono interrogati da una tabella Kafka, vengono considerati consumati dalla coda. Pertanto, non si dovrebbero mai selezionare dati direttamente da una tabella di Kafka Engine, ma utilizzare invece una vista materializzata. Una vista materializzata viene attivata una volta che i dati sono disponibili in una tabella di Kafka Engine. Automaticamente sposta i dati da una tabella Kafka a una tabella di tipo MergeTree o Distributed. Quindi, sono necessarie almeno 3 tabelle:

- La tabella di origine del motore Kafka;
- La tabella di destinazione (famiglia MergeTree o distribuita);
- Vista materializzata per spostare i dati;

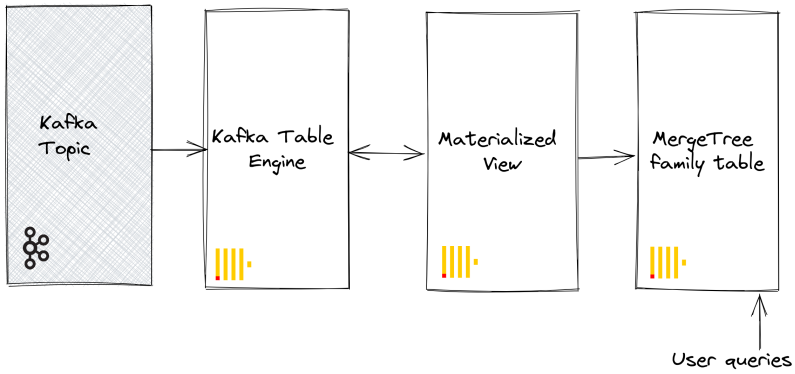

Figure 12: Architettura di Kafka Engine in ClickHouse

3.5.3 Trasferimento dati tramite Materialized View

Una materialized view funge da ponte tra la fonte dei dati (Kafka Engine) e la destinazione dei dati (MergeTree). Quando nuovi dati vengono scritti nella tabella Kafka Engine, la materialized view viene attivata automaticamente.
La materialized view esegue una query sulla tabella Kafka Engine per selezionare i dati più recenti. Una volta selezionati, questi dati vengono inseriti nella tabella di destinazione (ad esempio, una tabella MergeTree). Questo processo avviene in modo automatico e immediato, senza bisogno di intervento manuale.
In pratica, la materialized view si assicura che la tabella di destinazione sia sempre aggiornata con i dati più recenti presenti nella tabella Kafka Engine. Questo offre numerosi vantaggi:

- **Automatizzazione del processo**: Non è necessario eseguire manualmente operazioni di trasferimento dati da una tabella all'altra. La materialized view si occupa di tutto in modo automatico;
- **Efficienza**: Il trasferimento dei dati avviene in tempo reale, garantendo che la tabella di destinazione sia sempre allineata con la fonte dei dati senza ritardi;
- **Ottimizzazione delle risorse**: Il processo di trasferimento dei dati è gestito in modo efficiente, utilizzando al meglio le risorse disponibili e garantendo prestazioni elevate.

Nel contesto specifico, le materialized view sono responsabili di eseguire controlli sui dati, come ad esempio la verifica della loro correttezza ed affidabilità nel contesto di utilizzo, prima di inserirli nella tabella di destinazione. Questo processo assicura che i dati siano sempre affidabili e pronti per l'analisi, senza la necessità di ulteriori operazioni di pulizia o preparazione. Per esempio, nel caso dei dati di umidità raccolti da sensori in un'area urbana, la materialized view potrebbe eseguire controlli per assicurarsi che i valori rientrino all'interno di un intervallo plausibile e che non ci siano discrepanze improbabili. Ciò garantirebbe che i dati di umidità inseriti nella tabella di destinazione siano accurati e affidabili per l'analisi meteorologica o ambientale.

3.5.4 Tabella di origine di Kafka Engine per un sensore generico

Le tabelle del database impiegate per registrare le misurazioni di ciascuna tipologia di sensore presentano una configurazione sostanzialmente simile, differenziandosi principalmente per il tipo di dato della colonna relativa alla misurazione e per il *topic* di riferimento utilizzato per ottenere le misurazioni. Nello specifico per ogni sensore si avrà la seguente tabella Clickhouse:

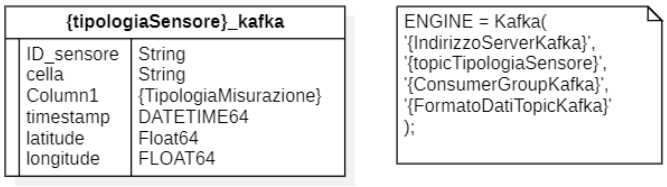

Figure 13: Tabella sensore generico per il reperimento da kafka - ClickHouse

La tabella è configurata con il motore di storage *Kafka*, il che significa che i dati verranno letti da un *topic Kafka*.
I campi sono:

- **ID_sensore**: un campo di tipo *String* che identifica univocamente il sensore che ha effettuato la misurazione;
- **cella**: un campo di tipo *String* che rappresenta la cella della città in cui è stata effettuata la misurazione;
- **value**: un campo di tipo variabile a seconda del tipo di misurazione che contiene il valore della temperatura;
- **timestamp**: campo di tipo *DATETIME64* che rappresenta il timestamp della misurazione della temperatura;
- **latitude**: un campo di tipo *Float64* che rappresenta la latitudine del luogo dove è stata effettuata la misurazione;
- **longitude**: un campo di tipo *Float64* che rappresenta la longitudine del luogo dove è stata effettuata la misurazione.

Mentre i parametri esposti racchiusi da parentesi graffe variano per ogni tipolgia di sensore correlato alla misurazione e sono:

- **tipologiaSensore**: viene sostituito con la tipologia del sensore che effettua le misurazioni salvate nella tabella; (ex. temperatures)
- **TipoDatoMisurazione**: viene sostituito con il tipo del dato che rappresenta la misurazione (ex. Float32, UInt8);
- **IndirizzoServerKafka**: specifica l'indirizzo del server Kafka. Nel nostro caso il server Kafka è in esecuzione su un container *Docker* raggiungibile tramite l'indirizzo: *'kafka:9092'*;
- **topicTipologiaSensore**: specifica il nome del topic Kafka da cui leggere i dati (ex.temperature);

- **ConsumerGroupKafka**: specifica il nome del consumer group Kafka che verrà utilizzato per leggere i messaggi dal topic *Kafka* denominato 'temperature'. Un consumer group in *Kafka* è un gruppo di consumatori che lavorano insieme per consumare i messaggi da uno o più topic. Ogni messaggio inviato a un *topic Kafka* può essere consumato da uno dei consumatori nel gruppo. I consumer all'interno di uno stesso gruppo condividono l'elaborazione dei messaggi all'interno dei topic: ogni messaggio viene elaborato da uno e un solo consumatore all'interno del gruppo. Nel nostro caso sarà sempre '*CG_Clickhouse_1*' per indicare il servizio di salvataggio *Clickhouse*.
- **FormatoDatiTopicKafka**: specifica il formato dei dati nel *topic Kafka*. Nel nostro caso, i dati sono nel formato JSONEachRow, che è un formato di serializzazione JSON di *ClickHouse* che consente di scrivere o leggere record JSON separati da una riga. Quindi avremo che «FormatoDatiTopicKafka» = JSONEachRow.
- **KafkaSkipBrokenMessages**: specifica il numero di errori da tollerare durante il parsing dei messaggi, configurato a livello di tabella, rappresenta la quantità massima di errori accettabili che il sistema può gestire durante il processo di analisi dei messaggi. Questo parametro consente di regolare il livello di tolleranza agli errori a livello di tabella, offrendo la possibilità di controllare quanto il sistema debba essere flessibile nell'interpretazione dei dati.

3.5.5 Misurazioni temperatura

Di seguito viene presentata una configurazione dettagliata per l'archiviazione delle misurazioni di temperatura. Tale configurazione si applica alla tabella 'temperatures_kafka', progettata per acquisire dati da un topic Kafka. La tabella è strutturata per includere l'ID del sensore (String), la posizione della cella (String), il valore della temperatura misurato (Float32), il timestamp della misurazione (DATETIME64), la latitudine (Float64) e la longitudine (Float64) del sensore. Ogni campo è definito con un tipo di dato specifico al fine di garantire la precisione e l'integrità dei dati.

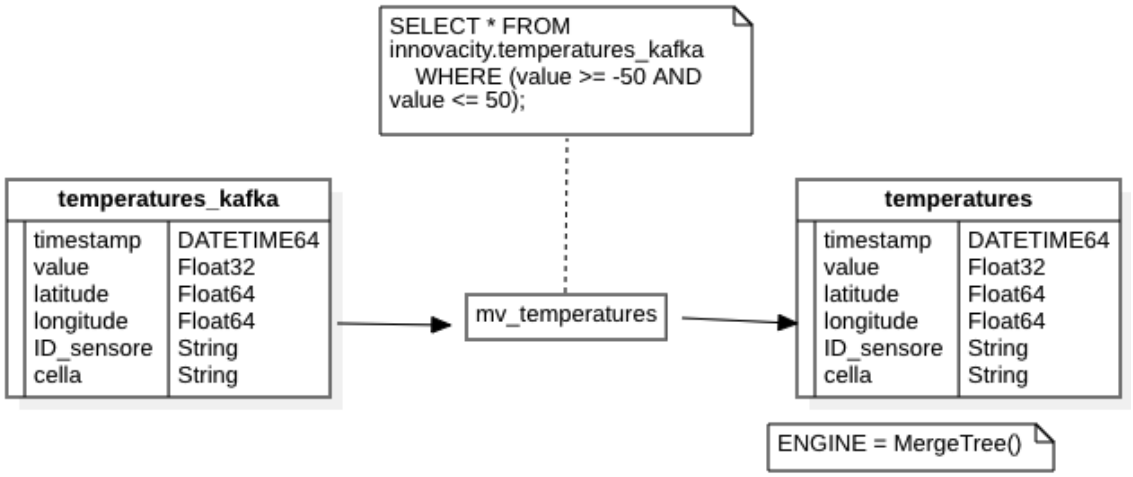

Figure 14: Tabella temperatures_kafka e temperatures

La tabella 'temperatures_kafka' è essenziale nel contesto dell'architettura dei dati, poiché funge da tramite tra un topic Kafka e il sistema di gestione dei dati ClickHouse. Questa tabella agisce come un'interfaccia di origine, trasformando i flussi di dati provenienti dal topic Kafka in un formato comprensibile per ClickHouse. Successivamente, una Materialized View, in questo caso 'mv_temperatures', opera su questa tabella per trasferire i dati ottenuti verso la tabella di destinazione 'temperatures' come spiegato in 3.5.3.

Projections per misurazioni di temperatura

Durante la fase di progettazione, è stata dedicata particolare attenzione all'utilizzo delle tabelle precedentemente descritte e alle richieste che verranno formulate su di esse. È emerso che, considerando il requisito di suddividere la città in una serie di celle e specificare la cella di origine della misurazione, la filtrazione delle misurazioni per celle diventerà una richiesta frequente al database. Di conseguenza, si è optato per l'utilizzo delle PROJECTIONS, le quali sono dettagliatamente descritte nella sezione 3.5.1.

```
1    --Projection per tabella temperatures
2    ALTER TABLE innovacity.temperatures ADD PROJECTION
         tmp_sensor_cell_projection (SELECT * ORDER BY cella);
3    ALTER TABLE innovacity.temperatures MATERIALIZE PROJECTION
         tmp_sensor_cell_projection;
```

Listing 2: Esempio di proiezione e materializzazione in una tabella

La proiezione ci consentirà di effettuare rapidamente filtraggi basati sulle celle, anche se tale attributo non è definito come *PRIMARY_KEY* nella tabella originale.

3.5.6 Misurazioni umidità

Le considerazioni relative al salvataggio delle misurazioni di umidità coincidono con quelle espresse nella sezione 3.5.5 riguardo alle misurazioni di temperatura. In questa situazione, dove le misure riguardano l'umidità, la tabella di destinazione ClickHouse è nominata 'humidity':

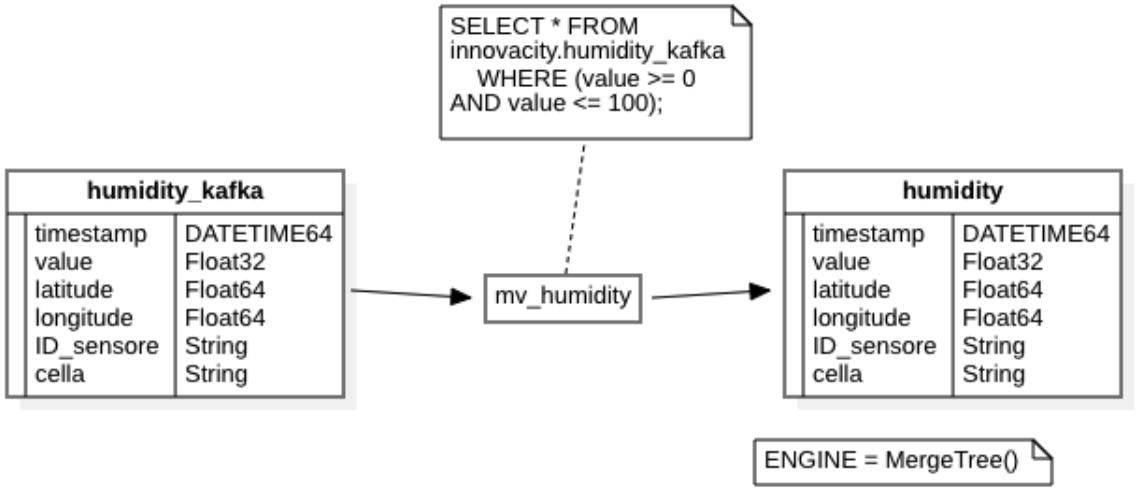

Figure 15: Tabella humidity_kafka e humidity

Projections per misurazioni di umidità

Dopo aver considerato le stesse argomentazioni presentate nella sezione 3.5.5 riguardanti le misurazioni di temperatura, abbiamo deciso di estendere l'utilizzo delle PROJECTION anche alle misurazioni di umidità. I vantaggi ottenuti risultano essere simili a quelli evidenziati per le misurazioni di temperatura, come descritto nella stessa sezione. A seguire, vengono illustrate le configurazioni delle PROJECTION relative alle tabelle delle misurazioni di umidità:

```
1  --Projection per tabella humidity
2  ALTER TABLE innovacity.humidity ADD PROJECTION
       umd_sensor_cell_projection (SELECT * ORDER BY cella);
3  ALTER TABLE innovacity.humidity MATERIALIZE PROJECTION
       umd_sensor_cell_projection;
```

3.5.7 Misurazioni di polveri sottili

Le considerazioni concernenti l'archiviazione delle misurazioni di polveri sottili corrispondono a quelle espresse nella sezione 3.5.5 in merito alle misurazioni di temperatura.

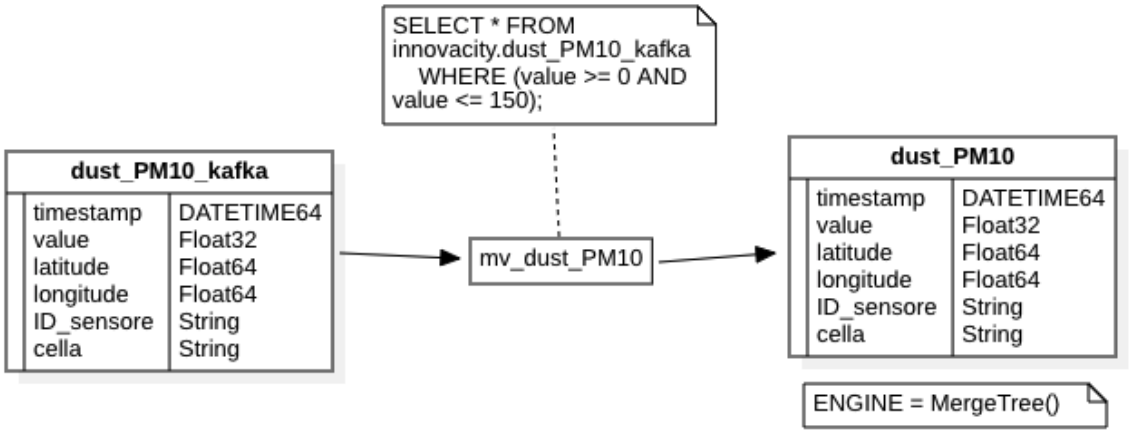

Figure 16: Tabella dustPM10_kafka e dustPM10

Projections per misurazioni di polveri sottili

Dopo aver considerato le stesse argomentazioni presentate nella sezione 3.5.5 riguardanti le misurazioni di temperatura, abbiamo deciso di estendere l'utilizzo delle PROJECTION anche alle misurazioni di polveri sottili. I vantaggi ottenuti risultano essere simili a quelli evidenziati per le misurazioni di temperatura, come descritto nella stessa sezione. A seguire, vengono illustrate le configurazioni delle PROJECTION relative alle tabelle delle misurazioni di polveri sottili:

```
1 --Projection per tabella dust_PM10
2 ALTER TABLE innovacity.dust_PM10 ADD PROJECTION
      dust_sensor_cell_projection (SELECT * ORDER BY cella);
3 ALTER TABLE innovacity.dust_PM10 MATERIALIZE PROJECTION
      dust_sensor_cell_projection;
```

3.5.8 Misurazioni guasti elettrici

Segue una dettagliata configurazione per l'archiviazione delle misurazioni relative ai guasti elettrici, applicabile alla tabella 'electricalFault_kafka' progettata per acquisire dati dal topic Kafka. La struttura della tabella include l'ID del sensore (String), la posizione della cella

(String), il valore misurato (UInt8), il timestamp della misurazione (DATETIME64), la latitudine (Float64) e la longitudine (Float64) del sensore, ciascuno definito con un tipo di dato specifico per garantire la precisione e l'integrità dei dati.

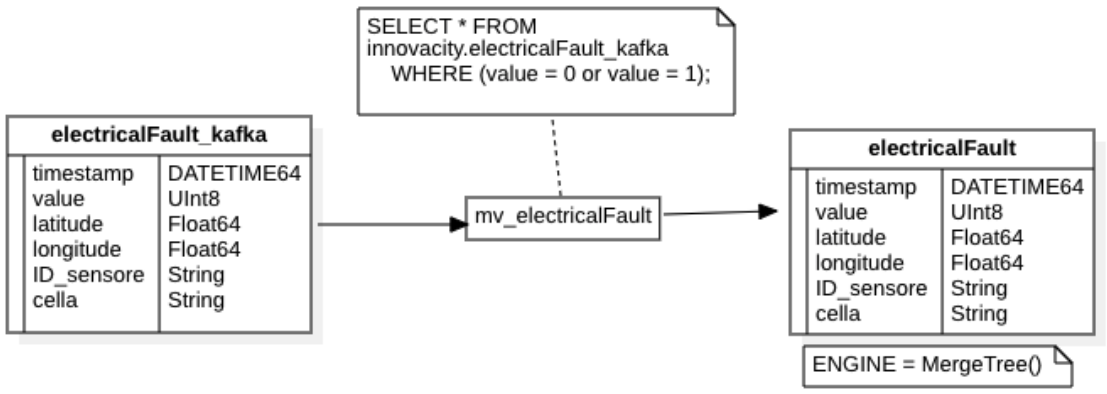

Figure 17: Tabella electricalFault_kafka e electricalFault

Le considerazioni concernenti l'archiviazione delle misurazioni di guasti elettrici corrispondono a quelle espresse nella sezione 3.5.5 in merito alle misurazioni di temperatura.

Projections per misurazioni di guasti elettrici

Dopo aver considerato le stesse argomentazioni presentate nella sezione 3.5.5 riguardanti le misurazioni di temperatura, abbiamo deciso di estendere l'utilizzo delle PROJECTION anche alle misurazioni di guasti elettrici. I vantaggi ottenuti risultano essere simili a quelli evidenziati per le misurazioni di temperatura, come descritto nella stessa sezione. A seguire, vengono illustrate le configurazioni delle PROJECTION relative alle tabelle delle misurazioni di guasti elettrici:

```
1 --Projection per tabella electricalFault
2 ALTER TABLE innovacity.electricalFault ADD PROJECTION
      elctF_sensor_cell_projection (SELECT * ORDER BY cella);
3 ALTER TABLE innovacity.electricalFault MATERIALIZE PROJECTION
      elctF_sensor_cell_projection;
```

3.5.9 Misurazioni stazioni di ricarica

Le considerazioni concernenti l'archiviazione delle misurazioni delle stazioni di ricarica corrispondono a quelle espresse nella sezione 3.5.8 in merito alle misurazioni guasti elettrici.

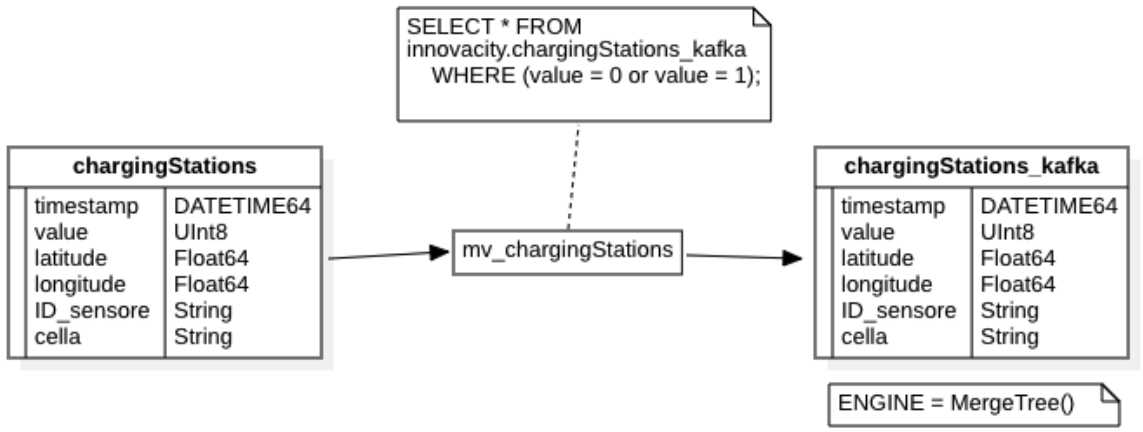

Figure 18: Tabella chargingStation_kafka e chargingStation

Projections per misurazioni delle stazioni di ricarica

Dopo aver considerato le stesse argomentazioni presentate nella sezione 3.5.5 riguardanti le misurazioni di temperatura, abbiamo deciso di estendere l'utilizzo delle PROJECTION anche alle misurazioni delle stazioni di ricarica. I vantaggi ottenuti risultano essere simili a quelli evidenziati per le misurazioni di temperatura, come descritto nella stessa sezione. A seguire, vengono illustrate le configurazioni delle PROJECTION relative alle tabelle delle misurazioni delle stazioni di ricarica:

```
1 --Projection per tabella chargingStations
2 ALTER TABLE innovacity.chargingStations ADD PROJECTION
      chS_sensor_cell_projection (SELECT * ORDER BY cella);
3 ALTER TABLE innovacity.chargingStations MATERIALIZE PROJECTION
      chS_sensor_cell_projection;
```

3.5.10 Misurazioni isole ecologiche

Le considerazioni concernenti l'archiviazione delle misurazioni delle isole ecologiche corrispondono a quelle espresse nella sezione 3.5.5 in merito alle misurazioni di temperatura.

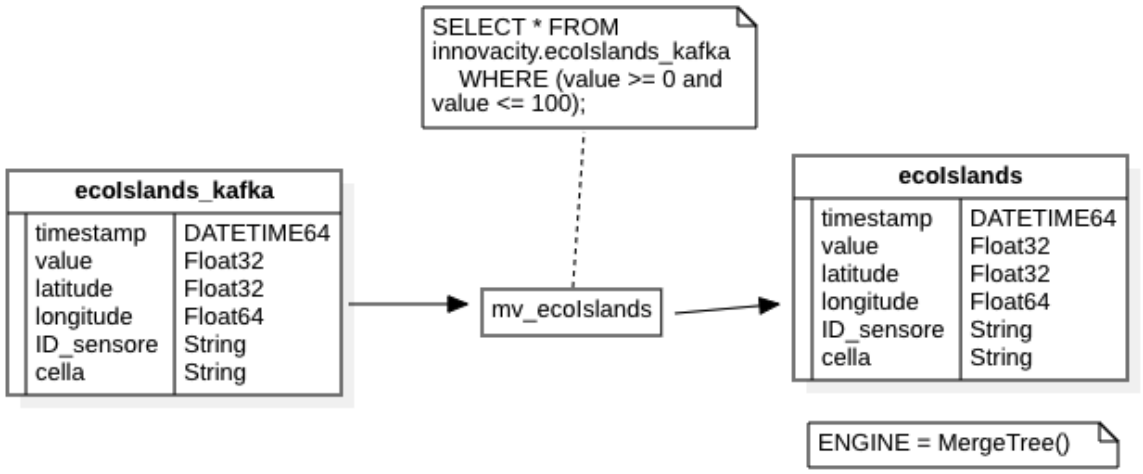

Figure 19: Tabella ecoIslands_kafka e ecoIslands

Projections per misurazioni delle isole ecologiche

Dopo aver considerato le stesse argomentazioni presentate nella sezione 3.5.5 riguardanti le misurazioni di temperatura, abbiamo deciso di estendere l'utilizzo delle PROJECTION anche alle misurazioni delle isole ecologiche. I vantaggi ottenuti risultano essere simili a quelli evidenziati per le misurazioni di temperatura, come descritto nella stessa sezione. A seguire, vengono illustrate le configurazioni delle PROJECTION relative alle tabelle delle misurazioni delle isole ecologiche:

```
1 --Projection per tabella ecoIslands
2 ALTER TABLE innovacity.ecoIslands ADD PROJECTION
      umd_sensor_cell_projection (SELECT * ORDER BY cella);
3 ALTER TABLE innovacity.ecoIslands MATERIALIZE PROJECTION
      umd_sensor_cell_projection;
```

3.5.11 Misurazioni sensori livello dell'acqua

Le considerazioni concernenti l'archiviazione delle misurazioni dei sensori di livello dell'acqua corrispondono a quelle espresse nella sezione 3.5.8 in merito alle misurazioni guasti elettrici.

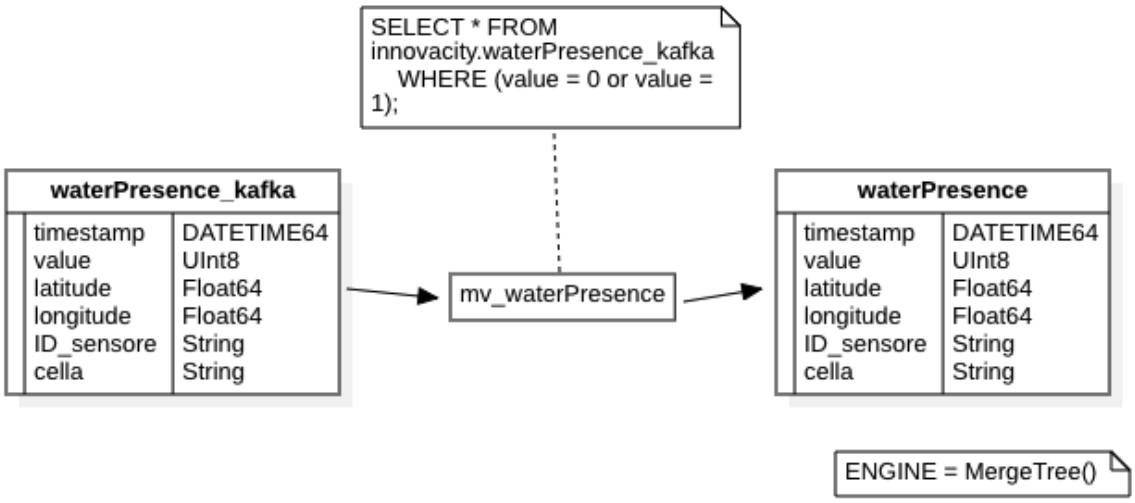

Figure 20: Tabella waterPresence_kafka e waterPresence

Projections per misurazioni del livello dell'acqua

Dopo aver considerato le stesse argomentazioni presentate nella sezione 3.5.5 riguardanti le misurazioni di temperatura, abbiamo deciso di estendere l'utilizzo delle PROJECTION anche alle misurazioni del livello d'acqua. I vantaggi ottenuti risultano essere simili a quelli evidenziati per le misurazioni di temperatura, come descritto nella stessa sezione. A seguire, vengono illustrate le configurazioni delle PROJECTION relative alle tabelle delle misurazioni del livello d'acqua:

```
1 --Projection per tabella waterPresence
2 ALTER TABLE innovacity.waterPresence ADD PROJECTION
      waPr_sensor_cell_projection (SELECT * ORDER BY cella);
3 ALTER TABLE innovacity.waterPresence MATERIALIZE PROJECTION
      waPr_sensor_cell_projection;
```

3.6 Grafana

Grafana è un software open source per la visualizzazione e l'analisi dei dati. È progettato per funzionare con vari database di serie temporali, tra cui Clickhouse. Grafana offre un'interfaccia utente intuitiva e flessibile che consente di creare e condividere dashboard personalizzate per monitorare i dati in tempo reale. È ampiamente utilizzato per monitorare sistemi e applicazioni, nonché per analizzare e visualizzare dati in tempo reale.

3.7 Dashboards

Per soddisfare tutti i requisiti definiti in *Analisi dei requisiti v2.0.0* sono state create due Dashboard:

- **Dashboard Principale:** Questa dashboard fornisce una visualizzazione chiara e intuitiva delle misurazioni provenienti da tutti i sensori, di tutte le tipologie, distribuiti nell'area urbana. La dashboard include una mappa interattiva della città che mostra la posizione geografica di ciascun sensore e la relativa ultima misurazione. Inoltre, viene presentato il punteggio di salute della città o di celle specifiche.
- **Dashboard dedicata:** Mostra le misurazioni di una specifica tipologia di sensore selezionata dall'utente in modo più dettagliato e permette di effettuare le attività di filtraggio e aggregazione definite in *Analisi dei requisiti v2.0.0*.

3.7.1 ClickHouse data source plugin

Documentazione:

https://grafana.com/grafana/plugins/grafana-clickhouse-datasource/
Questo plugin di grafana consente di connettersi a un'istanza di ClickHouse e di visualizzare i dati in tempo reale. È possibile eseguire query SQL personalizzate e visualizzare i risultati in forma di grafici, tabelle e pannelli personalizzati. Il plugin offre anche funzionalità di aggregazione e di calcolo dei dati, consentendo di analizzare e visualizzare i dati in modo flessibile e personalizzato.

Data sources configuration

La configurazione del data source avviene tramite file *yaml* che deve essere presente in *"grafana/provisioning/datasources"*. Il protocollo di trasporto utilizzato è TLS ma puo essere modificato nel file appena citato grazie al parametro di configurazione: "protocol".

Macro utilizzate

Per semplificare la sintassi e consentire operazioni dinamiche, come i filtri dell'intervallo di date, le queries al database Clickhouse possono contenere macro. Quelle utilizzate sono:

- **$__timeFilter(columnName)**: Permette di effettuare il filtro temporale alla query per ottenere le sole misurazioni all'interno dell'intervallo di tempo selezionato dall'utente.
- **$__timeInterval(columnName)**: Permette di modificare il raggruppamento temporale delle misurazioni in automatico sulla base dell'ampiezza dell'intervallo temporale selezionato dall'utente. In questo modo è possibile avere una visione ottimizzate delle misurazioni.

3.7.2 Variabili Grafana

Documentazione:

https://grafana.com/docs/grafana/latest/dashboards/variables/
Le variabili in Grafana sono un potente strumento per rendere le dashboard dinamiche e interattive. Permettono di filtrare i dati visualizzati in base a valori scelti dall'utente, rendendo la dashboard più versatile e adattabile a differenti esigenze.

Utilizzo delle variabili nella dashboard principale:

Nella dashboard principale, le variabili sono:

- **variabile $cella**: per mostrare solo le misurazioni provenienti da determinate celle della città,
- **variabili $<TipoSensore>_sensors_id**: per mostrare le misurazioni di determinati sensori di un certo tipo.

Queste variabili all'interno delle query al database permettono il filtraggio delle misurazioni sulla base di quanto selezionato dall'utente. Un esempio di query per la visualizzazione delle misurazioni time-series di temperatura è:

```
1   SELECT    ID_sensore, avg(value) as value,
2             $__timeInterval(timestamp) as timestamp
3   FROM    innovacity.temperatures
4   WHERE    $__timeFilter(timestamp) AND cella IN ($Cella) AND ID_sensore in (
        ${tmp_sensors_id})
5   GROUP BY ID_sensore, timestamp;
```

La query esposta mostra anche l'utilizzo delle macro esposte in: 3.7.1

Utilizzo delle variabili nella dashboard dedicata:

Nella dashboard dedicata alla visualizzazione spefica delle misurazioni di una sola tipologia di sensori, le variabili sono:

- **variabile $cella**: per mostrare solo le misurazioni provenienti da determinate celle della città,
- **variabili $<TipoSensore>_sensors_id**: per mostrare le misurazioni di determinati sensori di un certo tipo.
- **variabili $tabella**: per selezionare la tipoligia di sensore di cui si vuole visualizzare la dashboard dedicata e quindi la tabella del database da cui ricavare i dati.

- **$aggregazione**: variabile per selezionare l'intervallo temporale di aggregazione delle misurazioni (Automatico, Secondo,Minuto,Ora,Giorno,Mese,Nessuno). Nel caso della selezione della modalità "Automatico" si utilizzza l'intervallo temporale di aggregazione più opportuno sulla base dell'ampiezza dell'intervallo temporale selezionato dall'utente.
- **$Max_value**: variabile ad input numerico per filtrare le misurazioni con valore al di sotto di quello indicato.
- **$Min_value**: variabile ad input numerico per filtrare le misurazioni con valore al di sopra di quello indicato.

Variable Panel plugin

Documentazione: https://volkovlabs.io/plugins/volkovlabs-variable-panel/ Il plugin permette di creare dei pannelli grafana che possono essere posizionati ovunque nella Dashboard e che consentono di selezionare i valori delle variabili. Inoltre permette la visualizzazione ad albero delle variabili utile nel nostro caso dove i sensori sono contenuti all'interno di celle della città.

3.7.3 Grafana alerts

Documentazionehttps://grafana.com/docs/grafana/latest/alerting/
Grafana offre un sistema di alerting completo per monitorare i dati e inviare notifiche quando si verificano determinate condizioni. Le notifiche possono essere inviate tramite diversi canali, tra cui email, Slack, Telegram e Discord.

Alert Rule

Per poter configurare un alert è necessario creare una regola di alert. La regola di alert viene impostata tramite query al data source e fa scattare l'alert quando la query restituisce un risultato che soddisfa le condizioni impostate. Gli alert impostati sono per:

- Quando un sensore di temperatura registra una temperatura superiore ai 40°C;
- Quando un sensore di polveri sottili supera i 50 microgrammi al metro cubo;
- Quando un sensore di guasti elettrici rileva un guasto.

Gli alert attraversano 3 stati:

- **Pending:** indica che un alert è stato attivato, ma la sua valutazione non è ancora stata completata. Quando si è in questo stato è perchè il valore della query di alert è stato valutato e risulta essere vero ma la configurazione della regola di allerta ha impostato

che l'allerta deve essere attiva per un certo periodo di tempo prima di essere considerata vera e quindi inviare la notifica.

- **Firing:** indica che un alert è stato attivato e la sua valutazione ha confermato che la condizione di alert è soddisfatta per il periodo impostato nella regola e quindi viene inviata la notifica ai canali impostati.
- **OK:** indica che un alert è stato disattivato e la sua valutazione ha confermato che la condizione di alert non è più soddisfatta.

  Le regole di allerta sono impostabili tramite l'interfaccia grafica di Grafana e vengono esportate in formato *yaml* ed inserite in "/provisioning/alerting".

Configurazione il canale di notifica

Per configurare i canali di notifica è necessario andare in "Alerting" e selezionare "Notification channels" dall'interfaccia grafica di Grafana.
Per il progetto è stato scelto Discord come unico canale di notifica. Per configurare il canale di notifica è necessario:

- Seleziona Discord come canale di notifica.
- Inserisci il webhook URL del tuo canale Discord.
- Personalizza il messaggio di notifica.

Anche le impostazioni di configurazione del canale di notifica sono esportabili in formato *yaml* e vengono inserite in "/provisioning/alerting".

Notification policies

Le norme di notifica negli Alert di Grafana sono un modo potente per gestire l'invio degli alert a diversi canali di notifica.
Per una spiegazione dettagliata della configurazione si rimanda alla documentazione ufficiale di Grafana:
https://grafana.com/docs/grafana/latest/alerting/alerting-rules/create-notification-policy/
Anche le impostazioni delle notification policies sono esportabili in formato *yaml* e vengono inserite in "/provisioning/alerting".

3.7.4 Altri plugin utilizzati

Orchestra Cities Map plugin

Documentazione: https://grafana.com/grafana/plugins/orchestracities-map-panel/

Il plugin Orchestra Cities Map per Grafana estende il pannello Geomap di Grafana con diverse funzionalità avanzate per la visualizzazione di dati geolocalizzati su mappe:
Funzionalità principali:

- **Supporto per GeoJSON**: Permette di visualizzare dati geoJSON su mappe, come shapefile di città, regioni o stati.
- **Icone personalizzate**: Puoi utilizzare icone personalizzate per rappresentare diversi tipi di dati sui punti mappa.
- **Popup informativi**: Mostra popup con informazioni dettagliate quando si clicca su un punto mappa.
- **Strati multipli**: Permette di creare più strati sovrapposti per visualizzare diversi set di dati sulla stessa mappa.
- **Filtraggio e ricerca**: Puoi filtrare i punti mappa in base a diversi criteri, come proprietà dei dati o valori delle metriche.
- **Colorazione dei punti**: Puoi colorare i punti mappa in base a valori di metriche o ad altri criteri.
- **Legende personalizzate**: Puoi creare legende personalizzate per spiegare il significato dei colori e delle icone utilizzati nella mappa.

Viene utilizzato per poter visualizzare in modo diverse le icone dei diversi tipi di sensori dislocati nella città oltre che l'ultima misurazione effettuata ovvero lo stato attuale del sensore.

4 Architettura di deployment

L'architettura di deployment, detta anche "architettura di rilascio", rappresenta la struttura e la configurazione di un sistema software in fase di esecuzione. Essa definisce come i componenti software, i dati e le risorse di rete sono distribuiti e interconnessi nell'ambiente di produzione.

4.1 Architettura a microservizi

La decisione di adottare un'architettura a microservizi è stata motivata dalla necessità di creare una struttura modulare e scalabile. L'applicazione è stata suddivisa in una suite di microservizi, ciascuno dei quali può essere sviluppato, modificato, deployato e scalato

indipendentemente dagli altri. Docker rappresenta uno standard de facto per sistemi composti da microservizi, offrendo un ambiente uniforme e semplice da gestire.
L'architettura a microservizi si rivela una scelta solida nell'ambito dell'IoT, poiché si prevede che le diverse parti del sistema evolveranno in maniera indipendente nel tempo e poiché la scalabilità e l'isolamento dei guasti sono un aspetto critico. Questa scelta garantisce maggiore flessibilità e prestazioni ottimizzate mediante un utilizzo accurato delle risorse disponibili. Infatti, nel contesto dell'architettura a microservizi, si presenta l'opportunità di assegnare risorse specifiche a ciascun servizio, il che permette loro di scalare in modo differenziato in base alle necessità. Questo è particolarmente vantaggioso in scenari in cui i servizi potrebbero essere sottoscritti a specifici topic e argomenti all'interno di un sistema di streaming come Kafka.
L'allocazione di risorse individualizzate consente ai servizi di adattarsi dinamicamente alla loro attività e al volume di dati con cui devono interagire. Ad esempio, un servizio che riceve un alto flusso di dati da un particolare topic potrebbe richiedere una maggiore capacità di calcolo e di memorizzazione rispetto a un altro servizio che gestisce un carico meno intenso. Inoltre, questa flessibilità nell'allocazione delle risorse consente di ottimizzare l'efficienza complessiva del sistema, garantendo che le risorse siano allocate in modo proporzionale alla richiesta effettiva dei servizi. Ciò contribuisce a migliorare le prestazioni complessive del sistema e a garantire una gestione ottimale delle risorse disponibili.

4.2 Il container deployment

Docker è un software open source sviluppato in Go che facilita il deployment di sistemi software all'interno di container. Questi container contengono l'applicativo stesso e tutte le sue dipendenze, consentendo un'esecuzione flessibile in qualsiasi ambiente. Docker offre un'infrastruttura di deployment leggera e portatile, che consente di distribuire facilmente applicazioni in ambienti di sviluppo, test e produzione.
Per implementare l'intero stack tecnologico e i layer di elaborazione dati in streaming, è stato configurato un ambiente Docker a microservizi che simula la divisione e la distribuzione dei layer e dei componenti. In particolare, sono stati creati container per:

- **Data feed**:
  - **Simulators**: Esegue i **simulatori di sensori** per la raccolta dei dati.
  - Non espone porte all'esterno.
- **Streaming layer**:
  - Esegue **Apache Kafka** per la gestione del flusso di dati in tempo reale.
  - Accessibile agli altri container tramite l'indirizzo ***kafka:9092***.

  - Permette l'invio e il recupero di messaggi attraverso librerie e framework appositi.

- **Processing layer**:
  - Esegue l'app **Faust** per il processing e il calcolo del punteggio di salute.
  - Non espone porte all'esterno.

- **Storage layer**:
  - Esegue **Clickhouse** per lo storage delle misurazioni.
  - La banca dati è accessibile agli altri container tramite l'indirizzo ***clickhouse:8123***.

- **Data Visualization Layer**:
  - Esegue **Grafana** come interfaccia utente per la visualizzazione dei dati.
  - Espone la porta 3000 all'esterno per permettere l'accesso al servizio di dashboarding.

Questa struttura permette una distribuzione modulare e scalabile del sistema, semplificando la gestione e la manutenzione dei componenti e consentendo una rapida scalabilità in risposta alle esigenze emergenti. Grazie all'uso di Docker, si garantisce coerenza e riproducibilità dell'ambiente di esecuzione, semplificando il deployment e garantendo maggiore affidabilità nell'ambiente di produzione nonchè la possibilità di attribuire le risorse necessarie ad ogni servizio in modo mirato.

4.3 Comunicazione tra i componenti

Nel contesto di Docker, la comunicazione tra i container avviene tramite la rete Docker interna, che è una rete virtuale creata automaticamente da Docker per i servizi all'interno di uno stesso file Compose o di un ambiente Docker. Questa rete consente ai container di comunicare tra loro utilizzando i nomi dei servizi come hostnames.
Quando un container viene avviato, Docker assegna un hostname basato sul nome del servizio definito nel file Compose. Ad esempio, nel file Compose fornito, il servizio Kafka ha il nome "kafka" e il servizio ClickHouse ha il nome "clickhouse". Questi nomi sono utilizzati all'interno dei container stessi per identificare gli altri servizi. Quando un container desidera comunicare con un altro container sulla stessa rete Docker, può semplicemente utilizzare il nome del servizio come hostname.
Inoltre, Docker offre una funzionalità chiamata "Discovery", che consente ai container di scoprire automaticamente gli altri container sulla stessa rete Docker senza dover conoscere esplicitamente i loro indirizzi IP. Questo semplifica la configurazione e la gestione della comunicazione tra i container.

È anche possibile specificare dipendenze tra i servizi utilizzando l'attributo "depends_on" nel file Compose. Questo assicura che un servizio venga avviato solo dopo che i servizi di cui dipende sono stati avviati e sono nella condizione desiderata.
Infine, per i servizi che espongono porte, come Kafka, ClickHouse e Grafana, è possibile mappare le porte del container su porte del sistema host utilizzando l'attributo "ports" nel file Compose. Questo consente ad altri componenti esterni al Docker network di comunicare con i container attraverso le porte esposte.
Quando un container invia dati a un altro container tramite la rete Docker, i dati vengono incapsulati in pacchetti di rete utilizzando il protocollo TCP/IP. Questi pacchetti vengono quindi instradati attraverso la rete Docker, che si occupa di consegnarli al destinatario corretto.
In sintesi, Docker fornisce un'infrastruttura di rete integrata che gestisce la comunicazione tra i container all'interno dello stesso ambiente di deployment, semplificando la configurazione e la gestione della comunicazione tra i diversi componenti del sistema.

- **Comunicazione data feed Layer -> Streaming Layer**: Si utilizza libreria *Confluent Kafka* per Python che offre un'API efficiente e flessibile per inviare dati dai simulatori dei sensori a specifici topic Kafka.
- **Comunicazione Processing Layer -> Streaming Layer**: Si utilizza Faust come interfaccia di alto livello per la comunicazione con Kafka. Faust offre un'interfaccia intuitiva e ben documentata per consumare dati da topic con flussi di dati in tempo reale. Ulteriori informazioni in: 3.4
- **Comunicazione Storage Layer -> Streaming Layer**: Per ottenere in tempo reale i dati dai topic kafka viene utilizzato l'engine kafka di Clickhouse. Ulteriori informazioni in: 3.5.2
- **Comunicazione Data Visualization Layer-> Storage Layer**: Grafana si connette a Clickhouse per ottenere i dati da visualizzare in tempo reale tramite lo specifico plugin ClickHouse che permette l'utilizzo di un database Clickhouse come *data source*. Il plugin nasconde alcuni dettagli di implementazione sottostanti, come la gestione della connessione,protocolli di comunicazione e l'esecuzione delle query. Ulteriori informazioni in: 3.7.1

4.3.1 Dipendenze tra i servizi

- **Clickhouse -> kafka**: Il servizio ClickHouse dipende dal servizio Kafka. Questo assicura che il servizio ClickHouse venga avviato solo dopo che il servizio Kafka è stato avviato e è nella condizione desiderata.
- **Simulators -> kafka**: Il servizio Simulators dipende dal servizio Kafka. Questo assicura che il servizio Simulators venga avviato solo dopo che il servizio Kafka è stato avviato e è nella condizione desiderata.

- **Faust app -> kafka**: Il servizio Processor dipende dal servizio Kafka. Questo assicura che il servizio Processor venga avviato solo dopo che il servizio Kafka è stato avviato e è nella condizione desiderata.

5 Tabella dei requisiti soddisfatti

Si riporta ciascun requisito mediante il corrispondente codice, rispetto alla stessa tabella ritrovabile nel documento Analisi dei Requisiti v2.0.0, qui è presente una colonna Stato indicante la soddisfazione di tale requisito.

Codice	Importanza	Descrizione	Stato	
RF0	Obbligatorio	L'accesso al prodotto è vincolato da un *sistema*$_G$ di *login*$_G$, tuttavia, non è necessario che gli utenti si registrino autonomamente. Le credenziali di accesso sono fornite da terze parti o dall'amministratore del *sistema*$_G$.	Soddisfatto	
RF1	Obbligatorio	Il prodotto non deve avere una gestione di amministrazione.	Soddisfatto	
RF2	Obbligatorio	Il *sistema*$_G$ deve integrare simulatori di diverso tipo al fine di generare dati di misurazioni che siano coerenti con l'ambito del *sensore*$_G$ simulato.	Soddisfatto	
RF3	Obbligatorio	Ogni misurazione trasmessa dal simulatore del *sensore*$_G$ deve essere composta dall'id del *sensore*$_G$, il timestamp e la misurazione.	Soddisfatto	
RF4	Obbligatorio	Il *sistema*$_G$ deve essere in grado di simulare almeno un *sensore*$_G$ che rilevi la temperatura espressa in gradi Celsius.	Soddisfatto	

RF5	Obbligatorio	Il *sistema*$_G$ deve essere in grado di simulare almeno un *sensore*$_G$ che misuri l'umidità, espressa in percentuale di umidità nell'aria.	Soddisfatto	
RF6	Obbligatorio	Il *sistema*$_G$ deve essere in grado di simulare almeno un *sensore*$_G$ per la rilevazione delle particelle di polveri sottili presenti nell'aria, espresse in microgrammi per metro cubo.	Soddisfatto	
RF7	Obbligatorio	Il *sistema*$_G$ deve includere la simulazione di almeno un *sensore*$_G$ per individuare guasti elettrici. Questi sensori segnalano interruzioni nella fornitura di energia elettrica tramite un *bit*$_G$ binario, con il valore 0 che indica l'assenza di energia elettrica.	Soddisfatto	
RF8	Obbligatorio	Il *sistema*$_G$ deve essere in grado di simulare almeno un *sensore*$_G$ per monitorare lo stato di riempimento dei diversi conferitori nelle isole ecologiche. L'indicazione fornita sarà una percentuale di riempimento dell'isola ecologica.	Soddisfatto	
RF9	Obbligatorio	Il *sistema*$_G$ deve includere la simulazione di almeno un *sensore*$_G$ per le colonnine di ricarica. Questi sensori indicheranno tramite un *bit*$_G$ binario se la colonnina è occupata (*bit*$_G$ 1) o libera (*bit*$_G$ 0).	Soddisfatto	

RF10	Obbligatorio	Il *sistema*$_G$ deve includere la simulazione di almeno un *sensore*$_G$ per il livello dell'acqua. Questi sensori indicheranno il livello dell'acqua.	Soddisfatto	
RF11	Obbligatorio	Ogni dato generato dai simulatori dei sensori deve essere strettamente correlato al dato successivo, garantendo così una transizione realistica e plausibile tra le misurazioni.	Soddisfatto	
RF12	Obbligatorio	Il *sistema*$_G$ deve essere in grado di memorizzare in modo sicuro e efficiente i dati generati dai sensori. Ciò include la registrazione accurata di ogni misurazione, assicurando l'integrità e la coerenza dei dati.	Soddisfatto	
RF13	Obbligatorio	La *piattaforma*$_G$ deve supportare la visualizzazione di dati provenienti da diversi tipi di sensori.	Soddisfatto	
RF14	Obbligatorio	L'utente deve poter visualizzare una *dashboard*$_G$ con una panoramica completa dello stato della città tramite l'utilizzo di *widget*$_G$ adibiti alla rappresentazione delle misurazioni dei sensori.	Soddisfatto	
RF15	Obbligatorio	L'utente deve avere la possibilità di visualizzare le misurazioni all'interno dei *widget*$_G$ adibiti alla rappresentazione delle rilevazioni dei sensori in formato grafico time series.	Soddisfatto	

RF16	Obbligatorio	L'utente deve avere la possibilità di visualizzare le misurazioni all'interno dei *widget*$_G$ adibiti alla rappresentazione delle rilevazioni dei sensori in formato testuale time series.	Soddisfatto	
RF17	Obbligatorio	La visualizzazione delle misurazioni in formato testuale time series deve presentare le informazioni nel formato: `IDSensore ,TIMESTAMP, Dato`.	Soddisfatto	
RF18	Obbligatorio	L'utente deve essere in grado di visualizzare le ultime misurazioni all'interno dei *widget*$_G$ dedicati alla presentazione dei rilevamenti dei sensori che trasmettono dati binari (ex. Occupato/Libero) attraverso una mappa interattiva. La mappa, tramite etichette adeguate, deve rappresentare chiaramente il valore corrispondente all'ultima misurazione effettuata da ciascun *sensore*$_G$.	Soddisfatto	
RF19	Obbligatorio	La *dashboard*$_G$ richiede un aggiornamento quasi istantaneo per garantire che i dati provenienti dai sensori siano riflessi nel minor tempo possibile, entro un massimo di 10 secondi.	Soddisfatto	

RF20	Obbligatorio	La *dashboard*$_G$ deve mostrare un *widget*$_G$ distinto per ciascun tipo di *sensore*$_G$ attivo che trasmette dati al *sistema*$_G$, contenente le misurazioni in formato grafico, testuale o mappa interattiva.	Soddisfatto	
RF21	Obbligatorio	Ogni *widget*$_G$ che visualizza le misurazioni deve includere, insieme ai dati stessi, informazioni sull'identificativo dei sensori che hanno contribuito a quelle misurazioni.	Soddisfatto	
RF22	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori di temperatura.	Soddisfatto	
RF23	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori di temperatura deve offrire all'utente di default la visualizzazione di tali dati in un formato grafico a linee, con una linea corrispondente a ciascun *sensore*$_G$ coinvolto.	Soddisfatto	
RF24	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori di umidità.	Soddisfatto	

RF25	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori di umidità deve offrire all'utente di default la visualizzazione di tali dati in un formato grafico a linee, con una linea corrispondente a ciascun *sensore*$_G$ coinvolto.	Soddisfatto	
RF26	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori delle polveri sottili.	Soddisfatto	
RF27	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione temporale delle misurazioni effettuate dai sensori di polveri sottili deve offrire all'utente la possibilità di visualizzare tali dati in un formato grafico a linee, con una linea corrispondente a ciascun *sensore*$_G$ coinvolto.	Soddisfatto	
RF28	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori dei guasti elettrici.	Soddisfatto	
RF29	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori dei guasti elettrici deve offrire all'utente di default la visualizzazione di tali dati con una mappa interattiva delle ultime misurazioni.	Soddisfatto	

RF30	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori di soglia delle isole ecologiche.	Soddisfatto	
RF31	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori di soglia delle isole ecologiche deve offrire all'utente la visualizzazione di tali dati con una mappa interattiva delle ultime misurazioni.	Soddisfatto	
RF32	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori delle colonnine di ricarica.	Soddisfatto	
RF33	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori delle colonnine di ricarica deve offrire all'utente la visualizzazione di tali dati con una mappa interattiva delle ultime misurazioni.	Soddisfatto	
RF34	Obbligatorio	La *dashboard*$_G$ deve includere un *widget*$_G$ dedicato alle misurazioni dei sensori del livello dell'acqua.	Soddisfatto	
RF35	Obbligatorio	Il *widget*$_G$ destinato alla rappresentazione delle misurazioni effettuate dai sensori del livello dell'acqua deve offrire all'utente la visualizzazione di tali dati con una mappa interattiva delle ultime misurazioni.	Soddisfatto	

RF36	Obbligatorio	La *dashboard*$_G$ della città deve includere una mappa interattiva che mostra la posizione dei diversi sensori nella città.	Soddisfatto	
RF37	Obbligatorio	I sensori nella mappa devono essere etichettati in modo da consentirne il riconoscimento della tipologia.	Soddisfatto	
RF38	Obbligatorio	I sensori posizionati sulla mappa devono visualizzare l'ultimo valore registrato quando il puntatore del mouse è posizionato sopra di essi.	Decisione interna	Soddisfatto
RF39	Desiderabile	La *dashboard*$_G$ deve fornire un *widget*$_G$ con il punteggio di salute relativo alla città basato sui dati aggregati provenienti dai sensori.	Soddisfatto	
RF40	Obbligatorio	L'utente deve avere la possibilità di selezionare una cella, ovvero un'area specifica della città, al fine di visualizzare una *dashboard*$_G$ dedicata contenente esclusivamente sensori, misurazioni e punteggio di salute correlati a essa.	Soddisfatto	
RF41	Obbligatorio	L'utente deve poter filtrare la visualizzazione delle misurazioni di una specifica tipologia di sensori inserendo uno specifico intervallo temporale.	Soddisfatto	
RF42	Obbligatorio	Il *sistema*$_G$ deve verificare la validità dell'intervallo temporale inserito dall'utente.	Soddisfatto	

RF43	Obbligatorio	In caso di intervallo temporale non valido, il *sistema*$_G$ deve generare una notifica di errore.	Soddisfatto	
RF44	Obbligatorio	La notifica di errore relativa all'inserimento di un intervallo temporale non valido deve richiedere all'utente di reinserire date valide.	Soddisfatto	
RF45	Obbligatorio	La notifica di errore relativa all'inserimento di un intervallo temporale non valido deve essere chiara e informativa, indicando il motivo specifico dell'invalidità dell'intervallo temporale (data fine precedente a data inizio, arco temporale precedente o antecedente all'inizio della trasmissione dati).	Soddisfatto	
RF46	Obbligatorio	L'utente ha la possibilità di selezionare l'intervallo temporale desiderato (secondo, minuto, ora, giorno, mese, anno) per aggregare le misurazioni in base al relativo periodo di registrazione corrispondente.	Soddisfatto	
RF47	Obbligatorio	Il *sistema*$_G$ deve essere in grado di adattare dinamicamente la rappresentazione delle misurazioni secondo un intervallo temporale di aggregazione selezionato dall'utente.	Soddisfatto	

RF48	Obbligatorio	L'utente deve avere la possibilità di definire due valori (un minimo e un massimo) per filtrare le misurazioni dei sensori di una specifica tipologia, utilizzando questi limiti come criterio per visualizzare solo i dati compresi in quei range.	Soddisfatto	
RF49	Obbligatorio	Il *sistema*$_G$ deve verificare la validità dell'intervallo di rilevamento inserito dall'utente.	Soddisfatto	
RF50	Obbligatorio	In caso di intervallo di rilevamento non valido, il *sistema*$_G$ deve generare una notifica di errore.	Soddisfatto	
RF51	Obbligatorio	La notifica di errore relativa all'inserimento di un intervallo di rilevamento non valido deve richiedere all'utente di reinserire valori validi.	Soddisfatto	
RF52	Obbligatorio	La notifica di errore relativa all'inserimento di un intervallo di rilevamento non valido deve essere chiara e informativa, indicando il motivo specifico dell'invalidità dell'intervallo di rilevamento (data fine precedente a data inizio, arco temporale precedente o antecedente all'inizio della trasmissione dati).	Soddisfatto	

RF53	Obbligatorio	L'utente deve avere la possibilità di filtrare le misurazioni selezionando uno o più sensori di una specifica categoria in modo tale da visualizzare esclusivamente le misurazioni corrispondenti ai sensori selezionati.	Soddisfatto	
RF54	Obbligatorio	L'utente deve poter filtrare la visualizzazione delle misurazioni di una specifica tipologia di sensori selezionando una o più specifiche celle come criterio di filtro.	Soddisfatto	
RF55	Obbligatorio	L'utente deve poter applicare più filtri simultaneamente per la visualizzazione delle misurazioni di una specifica tipologia di sensori.	Soddisfatto	
RF56	Obbligatorio	L'utente deve poter rimuovere i filtri applicati e ripristinare la visualizzazione senza tali filtri.	Soddisfatto	
RF57	Opzionale	L'utente deve poter salvare in una lista di misurazioni rilevanti una misurazione trasmessa da un *sensore*$_G$.	Soddisfatto	
RF58	Opzionale	Il *sistema*$_G$ deve effettuare una verifica prima di salvare la misurazione tra le misurazioni rilevanti, assicurandosi che il dato non sia già presente in tale lista.	Soddisfatto	
RF59	Opzionale	L'utente deve poter visualizzare la lista delle misurazioni rilevanti.	Soddisfatto	

RF60	Opzionale	Ogni misurazione nella lista dei rilevanti deve fornire l'identificativo del *sensore*$_G$ che ha effettuato la misurazione.	Soddisfatto	
RF61	Opzionale	Ogni misurazione nella lista dei rilevanti deve fornire la tipologia del *sensore*$_G$ che ha effettuato la misurazione.	Soddisfatto	
RF62	Opzionale	Ogni misurazione nella lista dei rilevanti deve fornire l'orario e la data di misurazione.	Soddisfatto	
RF63	Opzionale	Ogni misurazione nella lista dei rilevanti deve fornire il valore misurato e la relativa unità di misura.	Soddisfatto	
RF64	Opzionale	L'utente deve poter rimuovere una misurazione dalla lista delle misurazioni rilevanti.	Soddisfatto	
RF65	Obbligatorio	L'utente deve essere in grado di ricevere notifiche nel caso in cui i sensori superino determinate soglie di sicurezza.	Soddisfatto	
RF66	Obbligatorio	L'utente deve essere in grado di visualizzare le informazioni dei sensori.	Soddisfatto	
RF67	Obbligatorio	L'utente deve essere in grado di visualizzare l'*ID*$_G$ dei sensori.	Soddisfatto	
RF68	Obbligatorio	L'utente deve essere in grado di visualizzare il tipo dei sensori.	Soddisfatto	
RF69	Obbligatorio	L'utente deve essere in grado di visualizzare la posizione dei sensori in coordinate.	Soddisfatto	

RF70	Obbligatorio	L'utente deve essere in grado di visualizzare la cella in cui è installato il *sensore*$_G$.	Soddisfatto	
RF71	Obbligatorio	L'utente deve essere in grado di visualizzare la data di installazione dei sensori.	Soddisfatto	
RF72	Obbligatorio	L'utente deve essere in grado di visualizzare l'unità di misura associata al *sensore*$_G$.	Soddisfatto	
RF73	Obbligatorio	La *piattaforma*$_G$ deve poter ricevere più rilevazioni in parallelo.	Soddisfatto	

5.1 Grafici requisiti soddisfatti

5.1.1 Requisiti funzionali

Riguardo alla soddisfazione dei vari requisiti funzionali, il gruppo ByteOps Engineering ha soddisfatto ....

5.1.2 Requisiti Obbligatori

Invece per quanto riguarda la copertura dei requisiti obbligatori, la copertura rilevata è del 100%.

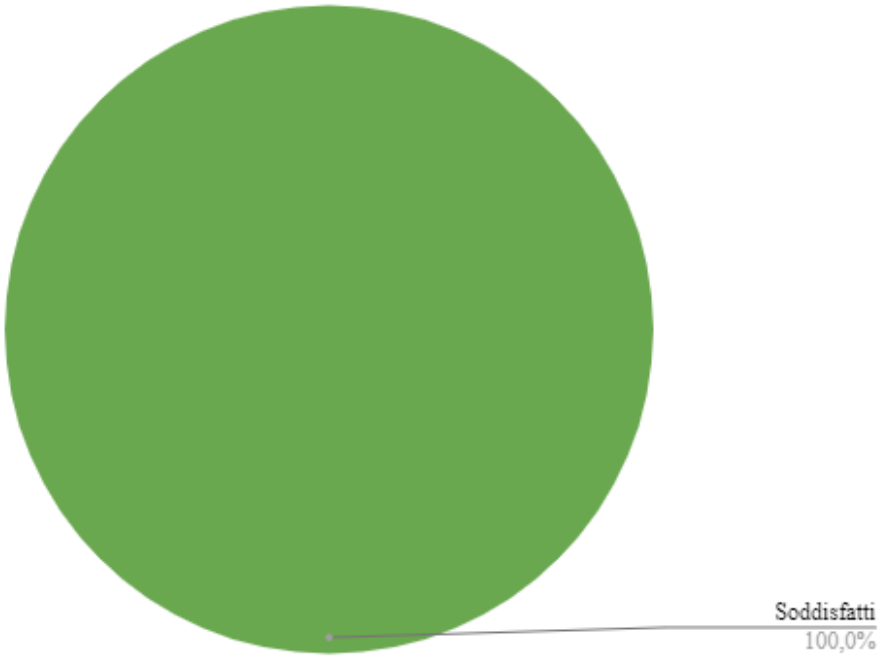

Figure 21: Requisiti obbligatori soddisfatti