Specifica Tecnica

v1.0

7Last

Versioni

Ver.	Data	Redattore	Verificatore$_G$	Descrizione
1.0	20/07/2024	Valerio Occhinegro	Leonardo Baldo	Stesura finale del documento
0.8	13/07/2024	Leonardo Baldo	Antonio Benetazzo	Stesura requisiti
0.7	09/07/2024	Matteo Tiozzo	Antonio Benetazzo	Conclusione stesura specifiche
0.6	02/07/2024	Davide Malgarise	Raul Seganfreddo	Inizio stesura specifiche di Grafana
0.5	30/06/2024	Leonardo Baldo	Raul Seganfreddo	Stesura specifiche di Clickhouse
0.4	25/06/2024	Raul Seganfreddo	Valerio Occhinegro	Stesura specifiche di Apache Flink
0.3	18/06/2024	Elena Ferro	Valerio Occhinegro	Stesura specifiche di Redpanda
0.2	15/06/2024	Valerio Occhinegro	Elena Ferro	Stesura architettura dei simulatori
0.1	08/06/2024	Elena Ferro	Matteo Tiozzo	Stesura struttura del documento

Indice

Elenco delle tabelle

Elenco delle figure

Indice dei listati

1 Introduzione

1.1 Scopo della specifica tecnica

Questo documento è rivolto a tutti gli *stakeholder* coinvolti nel progetto ***SyncCity*** - *A smart city monitoring platform*. Esso ha lo scopo di fornire una visione dettagliata riguardo l'architettura del sistema, i *design pattern* utilizzati, le tecnologie adottate e le scelte progettuali effettuate. Inoltre, contiene diagrammi UML delle classi e delle attività.

1.2 Scopo del prodotto

Lo scopo del prodotto è realizzare un prototipo di una piattaforma di monitoraggio per una *Smart City*, la quale permetta di raccogliere e analizzare dati provenienti da sensori IoT posizionati nelle città. Questi dati, una volta elaborati, devono essere visualizzati in maniera chiara e intuitiva, tramite grafici e mappe, per permettere alle autorità locali della città di prendere decisioni tempestive e mirate per migliorare la qualità della vita dei cittadini.

1.3 Glossario

Per evitare qualsiasi ambiguità o malinteso sui termini utilizzati nel documento, verrà adottato un glossario. Questo glossario conterrà varie definizioni. Ogni termine incluso nel glossario sarà indicato applicando uno stile specifico:

- aggiungendo una "G" al pedice della parola;
- fornendo il link al glossario online.

1.4 Riferimenti

1.4.1 Normativi

- Standard **ISO 8601**: `https://www.iso.org/iso-8601-date-and-time-format.html`;
- **Capitolato$_G$ d'appalto C6**: *SyncCity*$_G$ – A smart city$_G$ monitoring platform `https://www.math.unipd.it/~tullio/IS-1/2023/Progetto/C6.pdf`
- **Regolamento di progetto didattico** `https://www.math.unipd.it/~tullio/IS-1/2023/Dispense/PD2.pdf`

- **Norme di Progetto$_G$** v2.0:
  `https://7last.github.io/docs/pb/documentazione-interna/norme-di-progetto`

1.4.2 Informativi

- **Architettura Kappa e Lambda**:
  `https://nexocode.com/blog/posts/lambda-vs-kappa-architecture/` [Ultima consultazione: 2024-07-18];
- **Gestione degli eventi temporali con Apache Flink**: `https://nightlies.apache.org/flink/flink-docs-release-1.18/docs/concepts/time/` [Ultima consultazione: 2024-07-18];
- **Glossario** v2.0: `https://7last.github.io/docs/pb/documentazione-interna/glossario`

2 Tecnologie

Questa sezione si occupa di fornire una panoramica delle tecnologie utilizzate per implementare il sistema software. In particolare, delinea le piattaforme, gli strumenti, i linguaggi di programmazione, i framework e altre risorse tecnologiche che sono state impiegate durante lo sviluppo.

2.1 Docker

È una piattaforma di virtualizzazione leggera che semplifica lo sviluppo, il testing e il rilascio delle applicazioni fornendo un ambiente isolato e riproducibile. È utilizzato per creare ambienti di sviluppo standardizzati, facilitare la scalabilità delle applicazioni e semplificare la gestione delle risorse.

2.1.1 Ambienti

Per lo sviluppo di questo progetto sono stati ipotizzati i due seguenti scenari di esecuzione, separati grazie all'utilizzo di profili diversi di Docker Compose:

- `local`: utilizzato dagli sviluppatori per testare e sviluppare le funzionalità dell'applicazione sui propri computer. Questo ambiente permette di eseguire tutti i componenti del sistema all'interno di un container Docker, ad eccezione del simulatore Python. Esso viene eseguito direttamente sul sistema operativo dell'utente, in modo da facilitare il debugging e il testing delle funzionalità, senza dover necessariamente eseguire la *build* dell'immagine Docker ad ogni modifica del codice;
- `release`: utilizzato quando si desidera simulare un ipotetico ambiente di produzione o non è necessario modificare il codice Python. Consente di non dover manualmente installare le dipendenze o configurare l'ambiente di esecuzione. In questo caso, tutti i componenti del sistema vengono eseguiti all'interno di container Docker.

2.1.2 Immagini Docker

Nello sviluppo di questo progetto abbiamo utilizzato diverse immagini Docker, di seguito elencate. Le porte sono indicate nel formato `<porta_host>:<porta_container>`, dove `<porta_host>` è la porta del sistema host a cui si mappa la porta del container `<porta_container>`.

- **Simulator - Python**
  - **Immagine**: python:3.11.9-alpine;
  - **Riferimento**: Python Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `release`;
- **Redpanda Init**: l'immagine di `alpine` viene utilizzata per creare un container che si occupa di inizializzare il broker Redpanda.
  - **Immagine**: alpine:3.20.1;
  - **Riferimento**: Alpine [Ultima consultazione: 2024-06-25].
  - **Ambiente**: `local`, `release`.
- **Redpanda**
  - **Immagine**: docker.redpanda.com/redpandadata/redpanda:v23.3.11;
  - **Riferimento**: Redpanda Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`;
  - **Porte**:
    - 18081:18081: *schema registry*;
    - 18082:18082: *proxy*;
    - 19092:19092: *broker*.
- **Redpanda console**
  - **Immagine**: docker.redpanda.com/redpandadata/console:v2.4.6;
  - **Riferimento**: Redpanda Console Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`;
  - **Porte**:
    - 8080:8080.
- **Connectors**
  - **Immagine**: docker.redpanda.com/redpandadata/connectors:v1.0.27;

  - **Riferimento**: Redpanda Connectors Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`;
  - **Porte**:
    * 8083:8083.

- **ClickHouse**
  - **Immagine**: clickhouse/clickhouse-server:24-alpine;
  - **Riferimento**: ClickHouse Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`.
  - **Porte**:
    * 8123:8123.

- **Apache Flink**
  - **Immagine**: flink:1.18.1-java17;
  - **Riferimento**: Flink Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`.
  - **Porte**:
    * servizio `jobmanager`: 9001:8081 interfaccia *web*;
    * servizio `taskmanager`: 6121:6121.

- **Grafana**
  - **Immagine**: grafana/grafana-oss:10.3.0;
  - **Riferimento**: Grafana Docker Image [Ultima consultazione: 2024-06-02].
  - **Ambiente**: `local`, `release`.
  - **Porte**:
    * 3000:3000.

2.2 Linguaggi e formato dati

Nome	Versione	Descrizione	Impiego
Python	3.11.9	Linguaggio di programmazione ad alto livello, interpretato e multiparadigma.	Simulatore di sensori, *testing*, *script* per automatizzare il *deployment* dei *job* di Flink.
JSON	-	Formato di dati semplice da interpretare e generare, ampiamente utilizzato per lo scambio di dati tra applicazioni.	Configurazione *dashboard* Grafana, definizione di schemi Avro.
YAML	-	Linguaggio di serializzazione dei dati leggibile sia per gli esseri umani sia per le macchine.	Docker Compose, provisioning Grafana e configurazione *alert*, file di *workflow* per le GitHub Actions.
SQL	Ansi SQL	Linguaggio di programmazione specificamente progettato per la gestione e la manipolazione di dati all'interno di sistemi di gestione di database.	*Query* e gestione database ClickHouse.
TOML	1.0.0	Linguaggio di *markup* progettato per essere più leggibile e facile da scrivere rispetto ad altri formati di configurazione come JSON e YAML.	Configurazione e gestione dei sensori simulati.
Java	17	Linguaggio di programmazione ad alto livello, orientato agli oggetti.	Creazione di *job* per le aggregazioni dei dati in Apache Flink.

Tabella 1: Linguaggi e formato dati

2.3 Librerie

Python		
Nome	**Versione**	**Impiego**
`confluent_avro`	1.8.0	Serializzazione dei dati in formato Confluent Avro.
`coverage`	7.5.1	Strumento per misurare la percentuale di linee di codice e rami coperti dai test.
`isodate`	0.6.1	Libreria per la manipolazione delle date e delle ore in formato ISO8601.
`kafka-python-ng`	2.2.2	Client Kafka per Python.
`ruff`	0.3.5	Libreria per l'analisi statica del codice.
`toml`	0.10.2	Libreria per effettuare il parsing dei file di configurazione in formato TOML.
Java		
`flink-streaming-java`	1.18.0	Utilizzo di DataStream API di Flink.
`flink-connector-kafka`	3.1.0-1.18	Connessione di Flink a Kafka.
`flink-clients`	1.18.0	Creazione di *job* di Flink.
`flink-java`	1.18.0	Creazione di *job* di Flink.
`flink-avro-confluent-registry`	1.18.0	Connessione di Flink a uno *schema registry* che utilizza Avro.
`flink-shaded-guava`	31.1-jre-17.0	Gestione delle dipendenze di Flink.
`slf4j-simple`	1.7.36	Implementazione di SLF4J.
`lombok`	1.18.32	Libreria per la generazione di codice *boilerplate*.
`maven-assembly-plugin`	3.7.1	Plugin Maven per la creazione di un *fat jar*.

Tabella 2: Librerie utilizzate

2.4 Servizi

2.4.1 Redpanda

Redpanda è una piattaforma di streaming sviluppata in C++. Il suo obiettivo è fornire una soluzione leggera, semplice e performante, pensata per essere un'alternativa ad

Apache Kafka. Viene utilizzato per disaccoppiare i dati provenienti dal simulatore.

- **Versione**: v23.3.11;
- **documentazione**: `https://docs.redpanda.com/current/home/` [Ultima consultazione: 2024-06-02].

2.4.1.1 Vantaggi

I vantaggi nell'utilizzo di questo strumento consistono in:

- ***performance***: è scritto in C++ e utilizza il *framework* Seastar, offrendo un'architettura *thread-per-core* ad alte prestazioni. Ciò permette di ottenere un'elevata *throughput* e latenze costantemente basse, evitando cambi di contesto e blocchi. Inoltre, è progettato per sfruttare l'*hardware* moderno, tra cui unità NVMe, processori *multi-core* e interfacce di rete ad alta velocità;
- **semplicità di configurazione**: oltre al *message broker*, contiene anche un *proxy* HTTP e uno *schema registry*;
- **minore richiesta di risorse**: rispetto ad Apache Kafka, richiede meno risorse per l'esecuzione in locale, rendendolo più adatto per l'esecuzione su *hardware* meno potente;
- **compatibilità con le API di Kafka**: è compatibile con le API di Apache Kafka, consentendo di utilizzare le librerie e gli strumenti esistenti;

2.4.1.2 Casi d'uso

Tra i casi d'uso di Redpanda si possono citare:

- ***streaming* di eventi**, permettendo la gestione e l'elaborazione di flussi di dati in tempo reale;
- ***data integration***, agisce come un intermediario flessibile e robusto per l'integrazione dei dati, consentendo la raccolta, il trasporto e la trasformazione dei dati provenienti da diverse sorgenti verso varie destinazioni;
- **elaborazione di *big data***, permette di gestire e processare enormi volumi di dati in modo efficiente e scalabile;

- **messaggistica *real time***, supporta la messaggistica in tempo reale tra applicazioni e sistemi distribuiti.

2.4.1.3 Impiego nel progetto

Il **broker** Redpanda gestisce i dati provenienti dai simulatori e li rende disponibili per i due consumatori. Inoltre, con lo *schema registry* integrato è possibile garantire la compatibilità tra i dati prodotti dai simulatori e i consumatori. I consumatori sono:

- Il **connector sink ClickHouse**, che salva i dati nelle tabelle di ClickHouse;
- **Apache Flink**, che elabora i dati in tempo reale.

2.4.2 ClickHouse

ClickHouse è un sistema di gestione di database colonnare *open-source* progettato per l'analisi dei dati in tempo reale e l'elaborazione di grandi volumi di dati.

- **Versione**: v24-alpine;
- **documentazione**: `https://clickhouse.com/docs/en/intro` [Ultima consultazione: 2024-06-02].

2.4.2.1 Vantaggi

I vantaggi nell'utilizzo di questo strumento consistono in:

- **alte prestazioni**, è progettato per eseguire query analitiche complesse in modo estremamente rapido;
- **scalabilità orizzontale**, può essere scalato orizzontalmente su più nodi, permettendo di gestire grandi volumi di dati;
- **elaborazione in tempo reale**, è in grado di gestire l'ingestione e l'elaborazione dei dati in tempo reale, rendendolo ideale per applicazioni che richiedono l'analisi immediata dei dati appena arrivano;
- **compressione efficiente**, utilizza algoritmi di compressione avanzati per ridurre lo spazio di archiviazione e migliorare l'efficienza I/O;

- **facilità di integrazione**, si integra facilmente con molti strumenti di visualizzazione dei dati e piattaforme di business intelligence come Grafana;
- **partizionamento e indici**, supporta il partizionamento dei dati e l'uso di indici per ottimizzare le query;

2.4.2.2 Casi d'uso

ClickHouse è utilizzato in una varietà di casi d'uso, tra cui:

- **analisi dei *log* e monitoraggio**, utilizzato per l'analisi e il monitoraggio dei log in tempo reale;
- ***business intelligence***, impiegato in applicazioni di BI per eseguire analisi approfondite dei dati aziendali, supportando la presa di decisioni basata sui dati;
- ***data warehousing***, funziona come data warehouse per memorizzare e analizzare grandi volumi di dati.

2.4.2.3 Impiego nel progetto

ClickHouse viene utilizzato per memorizzare i dati grezzi provenienti dai simulatori; attraverso il *connector sink* di Redpanda, i *record* pubblicati nei *topic* vengono salvati in tabelle di ClickHouse. Inoltre, tramite l'utilizzo di Materialized Views, vengono effettuate delle semplici aggregazioni sui dati, come ad esempio la media oraria o giornaliera, le quali sono poi memorizzate in apposite tabelle, in modo da poterne monitorare l'andamento nel tempo.
Le aggregazioni più complesse che coinvolgono dati provenienti da sensori differenti sono invece effettuate utilizzando Apache Flink, come meglio descritto nella sezione 2.4.3.
ClickHouse si integra semplicemente con Grafana, attraverso l'utilizzo del plugin `datasource-clickhouse`, fornito da Grafana Labs.

2.4.3 Apache Flink

Apache Flink è un framework open-source per l'elaborazione dei dati in tempo reale e in *batch*. Sviluppato in Java e Scala, è progettato per gestire *data stream* in modo efficiente, consentendo l'elaborazione di grandi volumi di dati in tempo reale. Flink si

distingue per la sua capacità di fornire elaborazione a bassa latenza, esecuzione *fault-tolerant* e scalabilità orizzontale. Come sistema di *build* e gestione delle dipendenze, è stato utilizzato Maven.

- **Versione**: v1.18.1;
- **documentazione**: `https://flink.apache.org` [Ultima consultazione: 2024-06-25].

2.4.3.1 Vantaggi

- **elaborazione a bassa latenza**: Flink è progettato per elaborare i dati in tempo reale con latenza estremamente bassa, rendendolo ideale per applicazioni che richiedono risposte rapide ai cambiamenti dei dati.
- ***fault tolerance***: Flink utilizza una tecnologia chiamata *Stateful Stream Processing* che garantisce che lo stato dell'applicazione venga memorizzato in modo sicuro e possa essere recuperato in caso di guasti. Questo consente un'elaborazione affidabile e continua anche in presenza di errori hardware o software.
- **scalabilità**: Flink può scalare orizzontalmente su cluster di grandi dimensioni, distribuendo il carico di lavoro tra molteplici nodi per gestire volumi di dati crescenti senza compromettere le prestazioni.
- **modello di programmazione flessibile**: Flink offre due tipologie di API per l'elaborazione dei dati: DataStream API, utilizzato per l'elaborazione di flussi di dati non strutturati in tempo reale, e Table API, un'astrazione di livello superiore per manipolare dati strutturati come tabelle, facilitando l'uso di operazioni simili a SQL;
- **supporto per analisi complesse**: Flink fornisce potenti funzionalità di analisi come aggregazioni, join e *windowing*, che consentono di realizzare analisi complesse sui flussi di dati.

2.4.3.2 Casi d'uso

Tra i principali casi d'uso di Apache Flink si trovano:

- **applicazioni *event-driven***: applicazioni *stateful* che elaborano eventi provenienti da uno o più flussi di eventi e reagiscono agli eventi in ingresso attivando calcoli, aggiornamenti di stato o azioni esterne;

- ***data analytics***: estrazione di informazioni e *insight* a partire dall'elaborazione dei dati grezzi, sia in *real time* che in modalità *batch*;
- ***data pipeline***: ad esempio per la costruzione di *Extract-transform-load* (ETL) o integrazione di dati provenienti da sorgenti differenti.

2.4.3.3 Impiego nel progetto

Per ciascuno degli indici in seguito elencati è stato sviluppato un *job* in Apache Flink; i dettagli implementativi di ciascuno di essi sono meglio discussi nella sezione 3.5.

- ***Heat Index***: una misura che combina la temperatura dell'aria e l'umidità relativa per determinare la temperatura percepita dall'uomo. Questa misura riflette meglio il livello di disagio che una persona potrebbe sperimentare rispetto alla sola temperatura dell'aria;
- ***Charging Efficiency***: combinando i dati provenienti dai sensori di occupazione dei parcheggi e all'utilizzo delle colonnine di ricarica (relative allo stesso parcheggio), si calcola l'efficienza di utilizzo di queste ultime; viene calcolato sia un indice di utilizzo (`utilization_rate`) nel periodo di tempo considerato, sia un indice di efficienza (`efficiency_rate`) che rappresenta quanto tempo la colonnina è stata effettivamente utilizzata rispetto al tempo in cui il parcheggio è stato occupato.

2.4.4 Grafana

È una potente piattaforma di visualizzazione dei dati progettata per creare, esplorare e condividere *dashboard* interattive che visualizzano metriche, *log* e altri dati di monitoraggio in tempo reale.

- **Versione**: v10.3.0;
- **documentazione**: `https://grafana.com/docs/grafana/v10.4/` [Ultima consultazione: 2024-06-02].

2.4.4.1 Vantaggi

- **Facilità d'uso**: possiede un'interfaccia intuitiva che rende facile la creazione e la gestione delle dashboard;

- **flessibilità**: La capacità di integrarsi con molteplici sorgenti dati e l'ampia gamma di plugin disponibili la rendono estremamente flessibile;
- **personalizzazione**: permette una personalizzazione completa delle dashboard, soddisfando ogni possibile necessità di visualizzazione dei dati;
- **gestione degli accessi**: offre funzionalità avanzate di gestione degli accessi e delle autorizzazioni, consentendo di controllare chi può accedere alle *dashboard* e quali azioni possono eseguire.

2.4.4.2 Casi d'uso

- **Monitoraggio delle infrastrutture**: utilizzato per monitorare le prestazioni e la disponibilità delle infrastrutture IT, inclusi server, database, servizi cloud e altro;
- **analisi delle performance delle applicazioni**: utilizzato per monitorare le prestazioni delle applicazioni e identificare eventuali problemi di prestazioni;
- **analisi delle serie temporali**: utilizzato per visualizzare e analizzare dati di serie temporali, come metriche di monitoraggio, log e dati di sensori;
- **business intelligence**: utilizzato per creare *dashboard* personalizzate per l'analisi dei dati aziendali e la visualizzazione delle metriche chiave.

2.4.4.3 Impiego nel progetto

- **Visualizzazione dei dati**: creazione *dashboard* interattive che visualizzano i dati salvati su ClickHouse;
- **notifiche superamento soglie**: invio di notifiche nel caso in cui vengano superate delle soglie prestabilite, che rappresentano situazioni di eventuale pericolo, forte disagio o disservizio per i cittadini.

3 Architettura di sistema

3.1 *Data processing architectures*

Le architetture di tipo *data processing* sono progettate per gestire l'*ingestion*, *processing* e memorizzazione di grandi quantità di dati. Esse permettono di analizzare e ottenere informazioni utili (*insight*) da questi dati, consentendo di ottimizzare i processi decisionali e migliorare le prestazioni aziendali. Esistono diverse architetture, ciascuna con le proprie caratteristiche e vantaggi. Tra le più comuni troviamo l'architettura *lambda* e l'architettura *kappa*.

3.1.1 Architettura *lambda*

L'architettura *lambda* è costituita dalle seguenti quattro componenti:

- **sorgente di dati**: responsabile dell'acquisizione dei dati grezzi da diverse sorgenti;
- ***batch layer***: responsabile dell'elaborazione e persistenza di dati storici in *batch* di grandi dimensioni; il suo scopo è fornire risposte complete e accurate, anche se con una latenza più elevata rispetto allo *speed layer*. Tale componente è tipicamente rappresentata da *framework* come Apache Hadoop o Apache Spark;
- ***speed* (*real-time*) *layer***: responsabile dell'elaborazione e persistenza di dati in tempo reale. I dati vengono elaborati in modo rapido e con una latenza molto bassa, fornendo tuttavia risposte meno elaborate rispetto al *batch layer*. Questa componente è tipicamente rappresentata da *framework* come Apache Storm o Apache Flink;
- ***serving layer***: responsabile della fornitura dei dati elaborati in modo veloce ed affidabile, indipendentemente dal *layer* di elaborazione utilizzato.

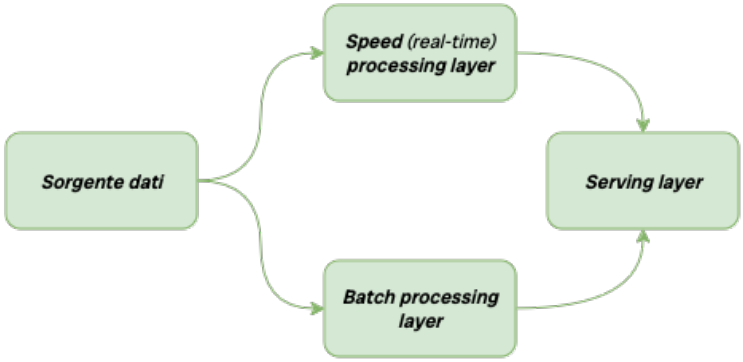

Figura 1: Architettura *lambda*

3.1.1.1 Vantaggi e svantaggi

L'architettura *lambda* offre diversi vantaggi, tra cui la **scalabilità orizzontale**, la **tolleranza ai guasti** e la **flessibilità**. Tuttavia, la presenza di due *layer* di elaborazione separati può portare a problemi di coerenza dei dati, duplicazione della logica di aggregazione e complessità aggiuntiva nella gestione del sistema. Inoltre, rispetto all'architettura *kappa*, l'architettura *lambda* può avere una latenza più elevata.

3.1.1.2 Casi d'uso

L'architettura *lambda* è particolarmente adatta per applicazioni che richiedono sia un'analisi sui dati in tempo reale che un'analisi storica.

3.1.2 Architettura *kappa*

L'architettura *kappa* è stata introdotta per semplificare l'architettura *lambda*, eliminando la necessità di gestire due *layer* di elaborazione separati per il *batch processing* e il *real-time processing*. Essa si divide in tre componenti principali:

- **sorgente di dati**: responsabile dell'acquisizione dei dati grezzi da diverse sorgenti;
- ***processing layer***: responsabile dell'elaborazione dei dati in tempo reale, senza la necessità di separare i dati in *batch* e *real-time*;
- ***serving layer***: responsabile della fornitura dei dati elaborati in modo veloce ed affidabile.

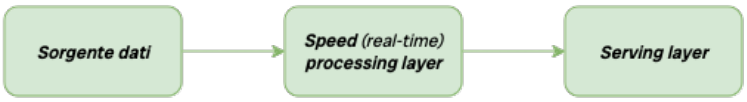

Figura 2: Architettura *kappa*

3.1.2.1 Vantaggi e svantaggi

L'architettura *kappa* offre diversi vantaggi, tra cui la **semplicità**, la **riduzione dei costi** e la **bassa latenza**. Tuttavia, può non essere adatta per applicazioni che richiedono un'analisi storica dei dati.

3.1.2.2 Casi d'uso

L'architettura *kappa* è particolarmente adatta per gli scenari in cui sono critici i dati in tempo reale e l'analisi dei dati storici è meno importante. Inoltre, semplifica notevolmente il processo di sviluppo e manutenzione dei sistemi di elaborazione dei dati.

3.2 Architettura scelta

Nello scenario del capitolato proposto da *SyncLab S.r.L.*, è importante l'analisi in tempo reale, in quanto i dati provenienti dai sensori IoT devono fornire informazioni sempre aggiornate ed eventualmente sollevare allarmi in caso di situazioni critiche. Inoltre, non è richiesta l'aggregazione storica di dati, dunque i vantaggi dell'architettura *lambda* non risultano utili per i nostri fini. Per soddisfare tali requisiti, è stata dunque scelta l'architettura *kappa*.

3.2.1 Componenti di sistema

All'interno del sistema progettato sono dunque presenti le seguenti componenti:

- **sorgenti di dati**: costituite dal simulatore di sensori, il quale genera i dati grezzi che in un contesto reale sarebbero provenienti dai sensori IoT;
- ***streaming layer***: gestisce il flusso di dati in tempo reale provenienti dai sensori. È composto da *Redpanda* e lo *Schema Registry*;
- ***processing layer***: elabora i dati in tempo reale per calcolare metriche e indici. È composto da *Apache Flink*;

- ***storage layer***: memorizza i dati elaborati per l'analisi e la visualizzazione. È composto da *ClickHouse;*
- ***data visualization layer***: fornisce un'interfaccia utente per visualizzare i dati elaborati. È composto da *Grafana*.

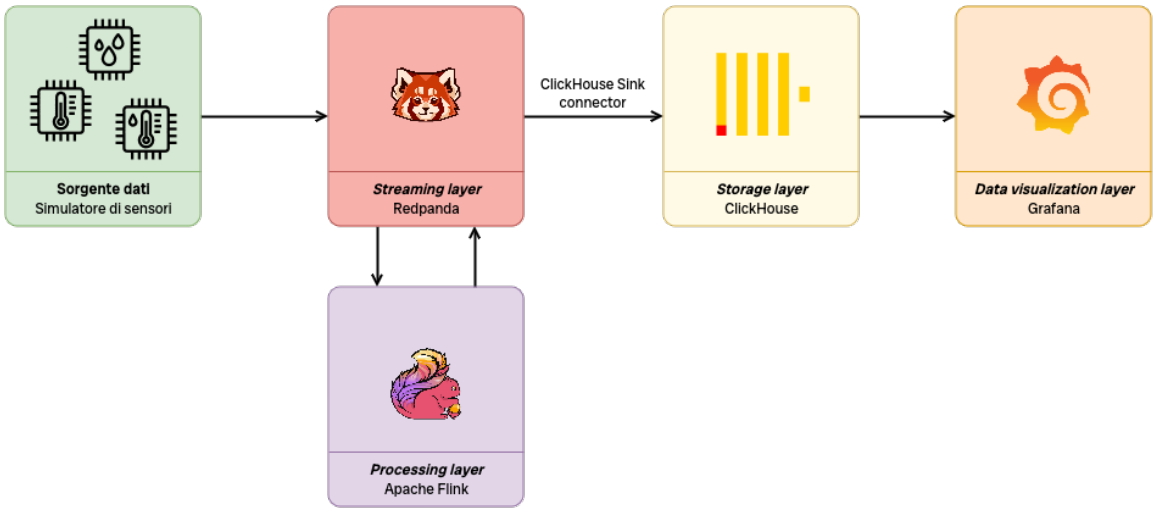

Figura 3: Componenti di sistema ad alto livello

3.2.2 Flusso di dati

Per illustrare il flusso di dati all'interno del sistema, è stato realizzato il seguente diagramma, il quale mostra il percorso che i dati grezzi seguono dal simulatore fino alla visualizzazione tramite Grafana.

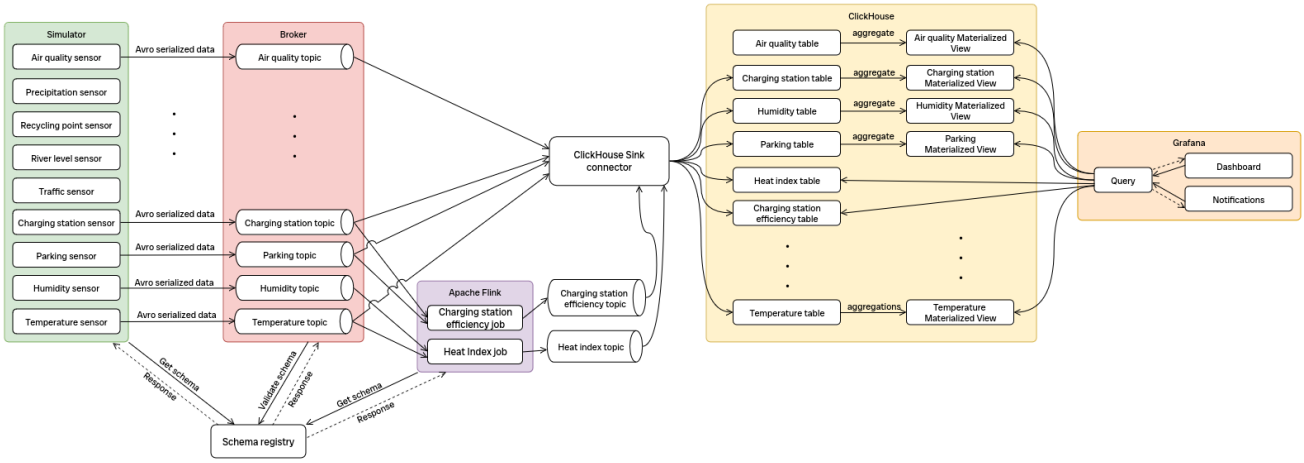

Figura 4: Flusso di dati all'interno del sistema. I sensori di precipitazioni, isole ecologiche, livello dei fiumi e traffico sono stati omessi per chiarezza, ma il percorso di tali dati è analogo a quello dei sensori di qualità dell'aria.

Il flusso seguito dai dati si può riassumere nei seguenti passaggi:

1. **generazione dei dati**: ciascun simulatore emula il comportamento di un singolo sensore IoT, generando ad intervalli periodici o ad eventi (*event-driven*) i dati grezzi relativi alla propria tipologia di dato.

2. **serializzazione e produzione dei dati**: i dati grezzi generati nel punto precedente vengono serializzati utilizzando il formato Confluent Avro e inviati nel *topic* corrispondente al tipo di dato generato;

3. **elaborazione dei dati**: i *topic* contenenti i dati grezzi di temperatura, umidità, occupazione dei parcheggi e colonnine di ricarica vengono consumati da Apache Flink. Due *job* distinti si occupano di calcolare la temperatura percepita e il grado di efficienza delle colonnine di ricarica. Una volta elaborati, i dati vengono inviati rispettivamente nei *topic* `heat_index` e `charging_station_efficiency`;

4. **memorizzazione dei dati**: attraverso il connettore *sink* per ClickHouse, i dati pubblicati in tutti i *topic* vengono memorizzati nel database;

5. **aggregazioni con *materialized view***: attraverso l'utilizzo di *materialized view* in ClickHouse, vengono calcolate le statistiche relative ai dati memorizzati, come ad esempio la media oraria o giornaliera. Tali aggregazioni sono più semplici rispetto a quelle effettuate da Flink, in quanto non richiedono elaborazioni complesse sui dati;

6. **visualizzazione dei dati**: i dati memorizzati in ClickHouse vengono visualizzati tramite Grafana, che permette di creare *dashboard* personalizzate per monitorare i dati in tempo reale;

7. **notifiche**: Grafana esegue periodicamente delle *query* per verificare se sono state superate delle soglie predeterminate. In caso affermativo, vengono inviate notifiche tramite il canale Discord dedicato, in modo tale da poter avvisare l'autorità locale.

3.3 Architettura dei simulatori

I simulatori vengono utilizzati per produrre dati grezzi che sostituiscono le rilevazioni effettuate dai sensori IoT in un contesto reale. Per tale motivo, questa parte del sistema non è ufficialmente parte del prodotto finale, ma è stata sviluppata per scopi di *test* e dimostrativi nell'ambito del progetto didattico; ai fini di quest'ultimo, il gruppo ha deciso di dedicare alcune risorse per la progettazione.
Nei paragrafi successivi verranno descritti i moduli che compongono i simulatori, le classi e metodi principali e i *design pattern* utilizzati.
Sono stati implementati simulatori per i seguenti tipi dato:

- qualità dell'aria;
- precipitazioni;
- isole ecologiche;
- livello dei fiumi;
- traffico;
- colonnine di ricarica;
- parcheggi;
- temperatura;
- umidità.

3.3.1 Modulo `models`

Questo modulo contiene le classi che rappresentano i dati grezzi generati dai sensori (sottomodulo `raw_data`) e la configurazione dei sensori stessi (sottomodulo `config`) letta dal file di configurazione `sensors.toml` e dalle variabili d'ambiente.
Ciascun tipo di dato grezzo è rappresentato da una classe che estende `RawData` (astratta).
La classe `SensorConfig` riceve nel costruttore la configurazione sotto forma di dizionario, effettua *parsing*, validazione, popola con valori di *default* i campi mancanti (nel caso lo prevedano) ed inizializza i propri attributi, corrispondenti ai campi del file di configurazione.
Allo stesso modo, `EnvConfig` legge le variabili d'ambiente ed espone il metodo `bootstrap_server()`

che combina *host* e *port* per formare l'indirizzo del *broker* Kafka.

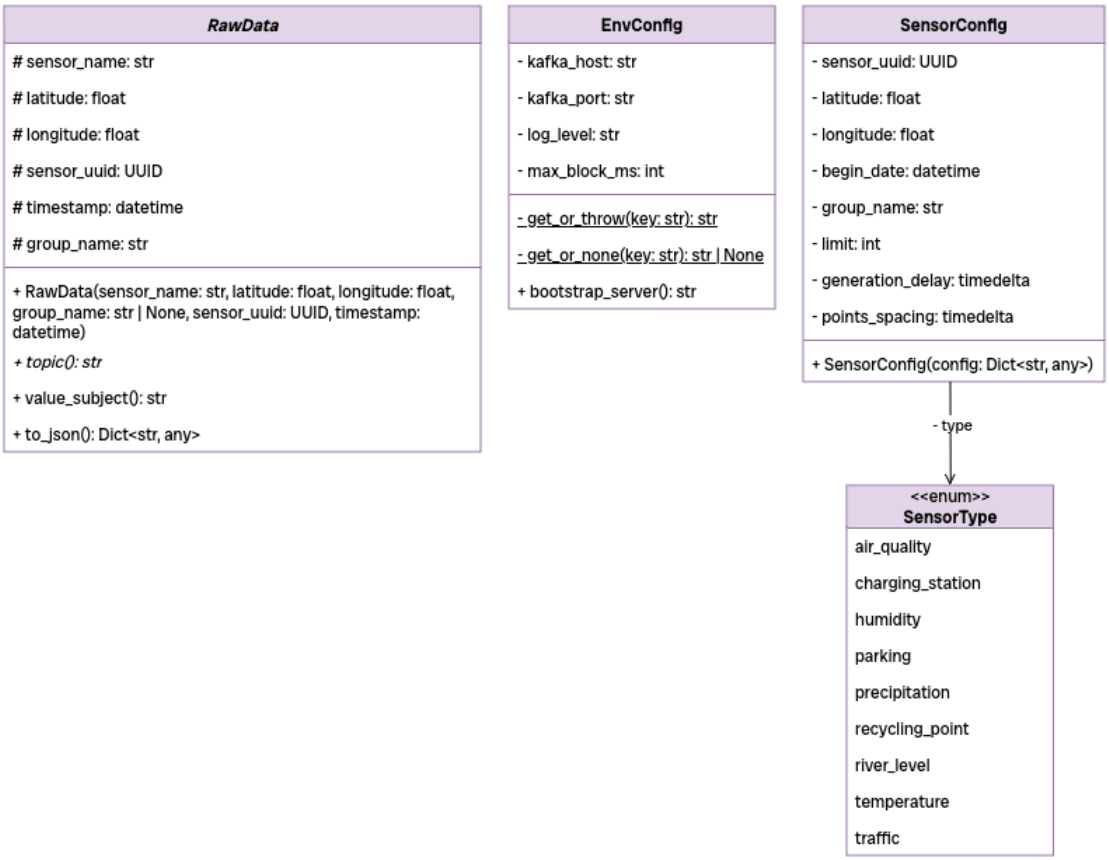

Figura 5: Diagramma delle classi del modulo `models`. Per ragioni di spazio, le implementazioni di `RawData` sono illustrate nel diagramma successivo

3.3.1.1 Classi, interfacce metodi e attributi

- **Classe astratta** `RawData`:
  - **Attributi**
    - `sensor_uuid` string [protected]: identificativo univoco del sensore;
    - `latitude` float [protected]: latitudine del sensore;
    - `longitude` float [protected]: longitudine del sensore;
    - `sensor_name` string [protected]: nome del sensore;
    - `group_name` string [protected]: nome del gruppo di sensori a cui appartiene.

  - **Metodi**
    * `topic()` string [public,abstract]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati;
    * `value_subject()` string [public]: restituisce il nome del campo value del record.

- **Classe EnvConfig**:
  - **Attributi**
    * `kafka_host` string [private]: *host* del *broker* Kafka;
    * `kafka_port` string [private]: porta del *broker* Kafka;
    * `log_level` string [private]: livello di *logging* da utilizzare;
    * `max_block_ms` int [private]: tempo massimo di blocco per la produzione di messaggi.
  - **Metodi**
    * `get_or_throw(key: string)` string [private,static]: restituisce il valore della variable d'ambiente associato alla chiave `key` se presente, altrimenti lancia un'eccezione;
    * `get_or_none(key: string)` string [private,static]: restituisce il valore della variable d'ambiente associato alla chiave `key` se presente, altrimenti `None`;
    * `bootstrap_server()` string [public]: restituisce l'indirizzo del *broker* Kafka.

- **Classe SensorConfig**:
  - **Attributi**
    * `sensor_uuid` string [private]: identificativo univoco del sensore;
    * `limit` int [private]: limite massimo di misurazioni da effettuare;
    * `begin_date` datetime [private]: data e ora di inizio delle misurazioni;
    * `latitude` float [private]: latitudine del sensore;
    * `longitude` float [private]: longitudine del sensore;
    * `group_name` string [private]: nome del gruppo di sensori a cui appartiene;
    * `type` SensorType [private]: tipo di sensore;
    * `points_spacing` timedelta [private]: intervallo temporale tra due misurazioni;

    * `generation_delay` timedelta [private]: ritardo tra la generazione di due misurazioni adiacenti.

- **Enum SensorType**:
  - **Valori**
    * `AIR_QUALITY`
    * `PARKING`
    * `RECYCLING_POINT`
    * `TEMPERATURE`
    * `TRAFFIC`
    * `CHARGING_STATION`
    * `PRECIPITATION`
    * `RIVER_LEVEL`
    * `HUMIDITY`
  - **Metodi**
    * `from_str(value: string)` SensorType [public,static]: restituisce il valore dell'enum corrispondente alla stringa `value`.

- **Classe AirQualityRawData**
  - **Metodi**:
    * `topic()` str [public]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati.

- **Classe ChargingStationRawData**
  - **Metodi**:
    * `topic()` str [public]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati.

- **Classe HumidityRawData**
  - **Metodi**:
    * `topic()` str [public]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati.

- **Classe ParkingRawData**
  - **Metodi**:
    - `topic()` str [public]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati.
- **Classe PrecipitationRawData**
  - **Metodi**:
    - `topic()` str [public]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati.
- **Classe RecyclingPointRawData**
  - **Metodi**:
    - `topic()` str [public]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati.
- **Classe RiverLevelRawData**
  - **Metodi**:
    - `topic()` str [public]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati.
- **Classe TemperatureRawData**
  - **Metodi**:
    - `topic()` str [public]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati.
- **Classe TrafficRawData**
  - **Metodi**:
    - `topic()` str [public]: restituisce il nome del *topic* in cui i dati grezzi vanno pubblicati.

3.3.2 Modulo `simulators`

Il modulo `simulators` contiene la logica per la generazione dei dati grezzi e l'orchestrazione dei simulatori.
L'*entrypoint* di tutto il sistema è la classe `SimulatorExecutor`, che riceve la configurazione dei sensori ed utilizzando il `SimulatorFactory` crea un `SimulatorThread` per ogni sensore. Quest'ultimo, a partire da una `SimulatorStrategy` ed una `ProducerStrategy`, genera i dati grezzi e li invia al `Producer`.
La classe `SimulatorThread` utilizza un `threading.Event`, che contiene un *flag* booleano. Quest'ultimo può essere impostato a `True` per far partire il *thread*; il metodo `wait()` permette di mettere in attesa il *thread* fino a quando il *flag* è impostato a `True`, per un periodo di tempo massimo espresso dal parametro `timeout`.
Tale metodo è utilizzato per attendere un determinato periodo di tempo tra la generazione di due misurazioni.

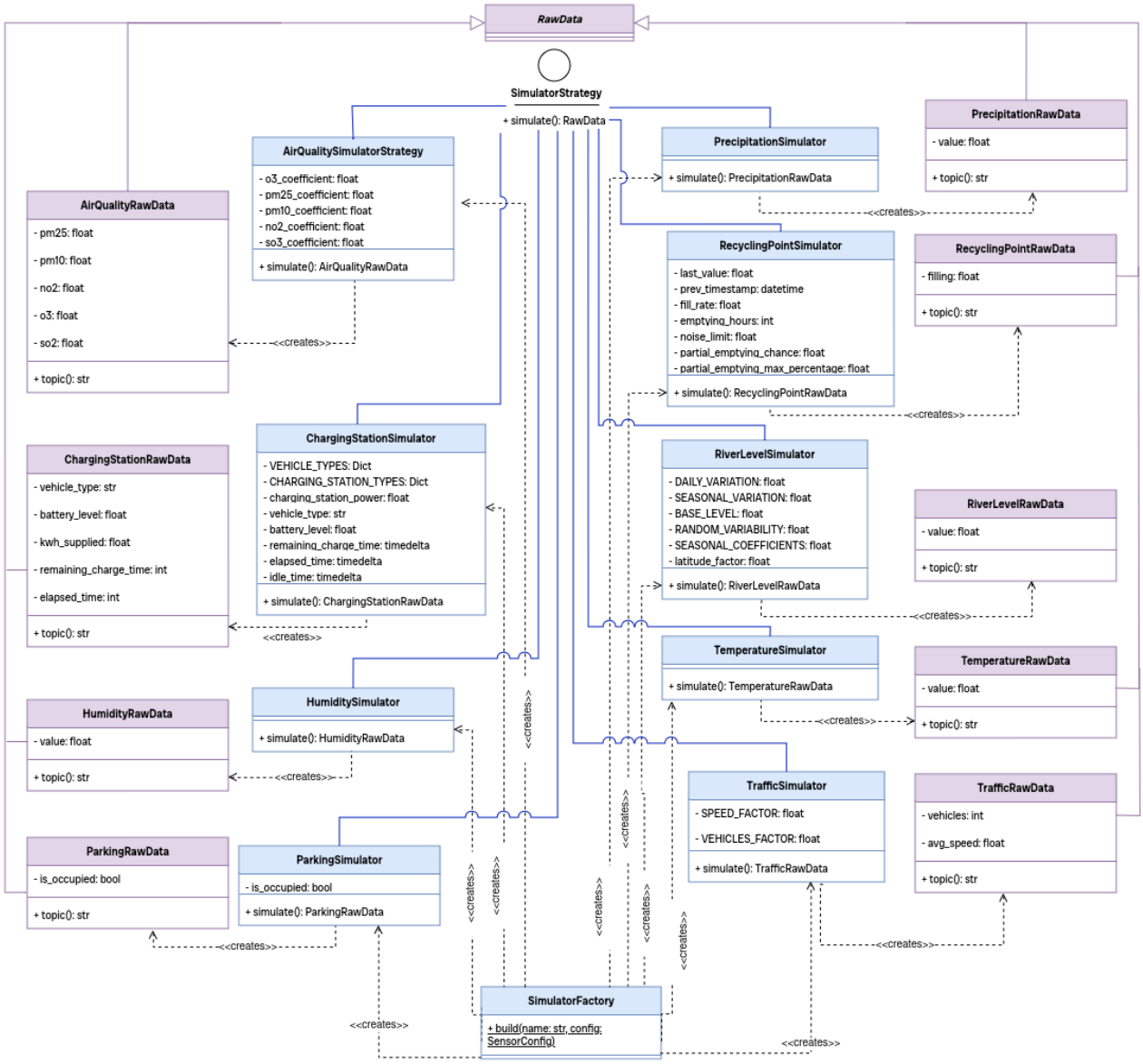

Figura 6: Diagramma delle classi modulo `simulators` e `models`.

3.3.2.1 *Design Pattern*

3.3.2.1.1 *Strategy*

All'interno del modulo `simulators` è stato utilizzato il *design pattern Strategy* per permettere la generazione di dati grezzi di diversi tipi. Ciascuna tipologia simulatore implementa l'interfaccia `SimulatorStrategy` che definisce il metodo `simulate()`. In questo modo, la classe `SimulatorThread` può eseguire il simulatore senza conoscere il tipo di

dato generato, rendendo inoltre semplice l'aggiunta di nuovi tipi di dati grezzi senza dover modificare il codice esistente.

3.3.2.1.2 *Factory*

La classe `SimulatorFactory` implementa il *design pattern Factory*, fornendo un metodo che si occupa della creazione dei simulatori a partire da un valore dell'enum `SensorType`.

3.3.2.2 Classi, interfacce metodi e attributi

- **Classe SimulatorExecutor**
  - **Attributi**:
    - `simulator_threads` List[SimulatorThread] [private]: lista dei simulatori da eseguire;
    - `stop_event`: threading.Event [private]: evento utilizzato per tenere attivo il *thread* principale dopo aver lanciato i *thread* dei simulatori.
  - **Metodi**:
    - `stop_all()` None [public]:
    - `run()` None [public]:
- **Classe SimulatorFactory**
  - **Metodi**:
    - `build(name: str, config: SensorConfig)` SimulatorStrategy [public, static]: restituisce un'istanza del simulatore corrispondente al nome `name`.
- **Interfaccia SimulatorStrategy**
  - **Attributi**:
    - `generation_delay` timedelta [protected]: ritardo tra la generazione di due misurazioni adiacenti.
    - `group_name` str [protected]: nome del gruppo di sensori a cui appartiene il sensore;
    - `latitude` float [protected]: latitudine del sensore;

    * `limit` int [protected]: limite massimo di misurazioni da produrre;
    * `longitude` float [protected]: longitudine del sensore;
    * `points_spacing` timedelta [protected]: intervallo temporale tra due misurazioni;
    * `sensor_name` str [protected]: nome del sensore per cui il simulatore genera dati;
    * `sensor_uuid` str [protected]: identificativo univoco del sensore per cui il simulatore genera dati;
    * `timestamp` datetime [protected]: data e ora dell'ultima misurazione;
  - **Metodi**:
    * `simulate` RawData [public]: metodo che simula la generazione di dati grezzi;

- **Classe SimulatorThread**
  - **Attributi**:
    * `simulator` SimulatorStrategy [private]: simulatore da eseguire;
    * `event` threading.Event [private]: evento utilizzato per fermare, far partire o lasciare in attesa il *thread* del simulatore.
  - **Metodi**:
    * `run()` None [public]: esegue il simulatore;
    * `is_running()` bool [public]: restituisce `True` se il *thread* è in esecuzione, `False` altrimenti;
    * `stop()` None [public]: ferma il *thread* del simulatore.

- **Classe AirQualitySimulatorStrategy**
  - **Attributi**:
    * `o3_coefficient` float [private]: coefficiente randomico utilizzato per generare il valore di `o3`;
    * `pm25_coefficient` float [private]: coefficiente randomico utilizzato per generare il valore di `pm25`;
    * `pm10_coefficient` float [private]: coefficiente randomico utilizzato per generare il valore di `pm10`;

    * `no2_coefficient` float [private]: coefficiente randomico utilizzato per generare il valore di `no2`;
    * `so2_coefficient` float [private]: coefficiente randomico utilizzato per generare il valore di `so2`;
  - **Metodi**:
    * `simulate()` AirQualityRawData [public]: genera un dato di tipo AirQualityRawData;

- **Classe ChargingStationSimulatorStrategy**
  - **Attributi**:
    * `VEHICLE_TYPES`: Dict[str, Dict[str, float]] [private]: dizionario contenente i tipi di veicoli supportati e la capacità minima e massima della batteria;
    * `CHARGING_STATION_TYPES` Dict[str, Dict[str, float]] [private]: dizionario contenente i tipi di colonnine di ricarica supportati e la potenza minima e massima;
    * `charging_station_power` float [private]: potenza della colonnina di ricarica;
    * `vehicle_type` str [private]: tipo di veicolo supportato dalla colonnina di ricarica;
    * `battery_level` float [private]: livello di carica della batteria;
    * `remaining_charge_time` timedelta [private]: tempo rimanente per completare la ricarica;
    * `elapsed_time` timedelta [private]: tempo trascorso dall'inizio della ricarica;
    * `idle_time` timedelta [private]: tempo in cui la colonnina di ricarica è inattiva;
  - **Metodi**:
    * `simulate()` ChargingStationRawData [public]: genera un dato di tipo ChargingStationRawData;

- **Classe HumiditySimulatorStrategy**
  - **Metodi**:
    * `simulate()` HumidityRawData [public]: genera un dato di tipo HumidityRawData;

- **Classe ParkingSimulatorStrategy**

  - **Attributi**:
    * `is_occupied` bool [private]: indica se il parcheggio è occupato;
  - **Metodi**:
    * `simulate()` ParkingRawData [public]: genera un dato di tipo ParkingRawData;

- **Classe PrecipitationSimulatorStrategy**
  - **Metodi**:
    * `simulate()` PrecipitationRawData [public]: genera un dato di tipo PrecipitationRawData;

- **Classe RecyclingPointSimulatorStrategy**
  - **Attributi**:
    * `last_value` float [private]: ultimo valore generato;
    * `prev_timestamp` datetime [private]: data e ora dell'ultima misurazione;
    * `fill_rate` float [private]: tasso di riempimento del contenitore;
    * `emptying_hours` int [private]: ore necessarie per svuotare il contenitore;
    * `noise_limit` float [private]: quantità massima di rumore da aggiungere al valore generato;
    * `partial_emptying_chance` float [private]: probabilità di svuotamento parziale;
    * `partial_emptying_max_percentage` float [private]: percentuale massima di svuotamento parziale;
  - **Metodi**:
    * `simulate()` RecyclingPointRawData [public]: genera un dato di tipo RecyclingPointRawData;

- **Classe RiverLevelSimulatorStrategy**
  - **Attributi**:
    * `DAILY_VARIATION` float [private]: variazione giornaliera massima del livello del fiume;
    * `SEASONAL_VARIATION` float [private]: variazione stagionale massima del livello del fiume;

    * `BASE_LEVEL` float [private]: livello base del fiume;
    * `RANDOM_VARIABILITY` float [private]: massima variazione casuale del livello del fiume;
    * `SEASONAL_COEFFICIENTS` Dict[int, float] [private]: coefficienti stagionali per la variazione del livello del fiume;
    * `latitude_factor` float [private]: fattore moltiplicativo per la latitudine;
  - **Metodi**:
    * `simulate()` RiverLevelRawData [public]: genera un dato di tipo RiverLevelRawData;

- **Classe TemperatureSimulatorStrategy**
  - **Metodi**:
    * `simulate()` TemperatureRawData [public]: genera un dato di tipo TemperatureRawData;

- **Classe TrafficSimulatorStrategy**
  - **Attributi**:
    * `SPEED_FACTOR` float [private]: fattore moltiplicativo per la velocità;
    * `VEHICLES_FACTOR` [private]: fattore moltiplicativo per il numero di veicoli;
  - **Metodi**:
    * `simulate()` TrafficRawData [public]: genera un dato di tipo TrafficRawData;

3.3.3 Modulo `producers`

Il modulo `producers` contiene le classi che si occupano della produzione dei dati grezzi.

3.3.3.1 Design Pattern

3.3.3.1.1 *Strategy*

Analogamente a quanto effettuato nel modulo `simulators`, anche in questo caso è stato utilizzato il *design pattern Strategy* per permettere la produzione di dati grezzi di diversi tipi. Sono stati implementati due produttori: `KafkaProducerAdapter` e `StdOutProducer`, rispettivamente per la produzione di dati su Kafka e su *standard output*.

3.3.3.1.2 *Object Adapter*

Al fine di adattare la classe `KafkaProducer`, contenuta nella libreria `kafka`, abbiamo utilizzato il *design pattern Adapter*, nella sua variante *Object Adapter*. Esso consente di rendere compatibile con l'interfaccia `ProducerStrategy` la classe `KafkaProducer`, la quale potrebbe subire cambiamenti da noi non controllabili. In tale eventualità, il *pattern Adapter* consente di poter continuare ad utilizzare tale classe senza dover modificare altre parti del sistema.

3.3.3.2 Classi, interfacce metodi e attributi

- **Interfaccia ProducerStrategy**
  - **Attributi**:
    - `serialization_strategy` SerializationStrategy [protected]: strategia di serializzazione dei dati grezzi.
  - **Metodi**:
    - `produce(data: RawData)` bool [public]: metodo che produce i dati grezzi in base alla strategia utilizzata, ritornando `True` in caso di successo, `False` altrimenti.
- **Classe KafkaProducerAdapter**
  - **Attributi**:
    - `serialization_strategy` SerializationStrategy [protected]: strategia di serializzazione dei dati grezzi.
    - `adaptee` KafkaProducer [private]: produttore di dati su Kafka.
  - **Metodi**:
    - `produce(data: RawData)` bool [public]: produce i dati grezzi su Kafka, ritornando `True` in caso di successo, `False` altrimenti.
  - **Note**:
    - la classe `KafkaProducerAdapter` si appoggia alla libreria `kafka` per interagire con il *broker*, realizzando il *pattern Adapter* per adattare la classe `KafkaProducer` all'interfaccia `ProducerStrategy`.
- **Classe StdOutProducer**

- **Attributi**:
  - `serialization_strategy` SerializationStrategy [protected]: strategia di serializzazione dei dati grezzi.
- **Metodi**:
  - `produce(data: RawData)` bool [public]: produce i dati grezzi su *standard output*.

3.3.4 Modulo `serializers`

Il modulo `serializers` contiene le classi che si occupano della serializzazione dei dati grezzi. Questi vengono serializzati in due formati: JSON e Confluent Avro. La serializzazione in JSON viene principalmente utilizzata per il *debugging* e la visualizzazione dei dati grezzi su *standard output*, mentre la serializzazione in Avro per l'effettiva produzione dei dati su Kafka.

3.3.4.1 *Design Pattern*

3.3.4.1.1 *Strategy*

Abbiamo deciso di utilizzare il *design pattern Strategy* per serializzare le istanze di `RawData` in *byte* utilizzando diverse codifiche, senza dover modificare il codice che le utilizza; per tale motivo sono state implementate le classi
`JsonSerializationStrategy` e `AvroSerializationStrategy`. `DictSerializable` è un'interfaccia che definisce il metodo `to_dict()`, il quale restituisce un dizionario Python. L'interfaccia `SerializationStrategy` definisce il metodo `serialize(data: DictSerializable)`, che prende in input un `DictSerializable` restituisce i *byte* corrispondenti.
`AvroSerializationStrategy` si appoggia sulla libreria `confluent_avro` per interagire con lo *schema registry* e costruire un oggetto di tipo `AvroValueSerde`, necessario per la serializzazione a partire da uno schema Avro e un dizionario.

3.3.4.1.2 *Object Adapter*

Entrambe le classi `JsonSerializationStrategy` e `AvroSerializationStrategy` necessitano di convertire un oggetto di tipo `RawData` in un dizionario Python prima della serializzazione vera e propria. Nel primo caso poi la conversione in *byte* è direttamente effettuata dal metodo `dumps()` della libreria `json`. Nel secondo caso invece, il dizionario costituisce

l'input per il metodo `serialize()` di `AvroValueSerde`.
Per tale motivo, abbiamo deciso di utilizzare il *design pattern Adapter* nella sua variante *Object Adapter*, al fine di rendere compatibile la classe `RawData` con l'interfaccia `DictSerializable`.

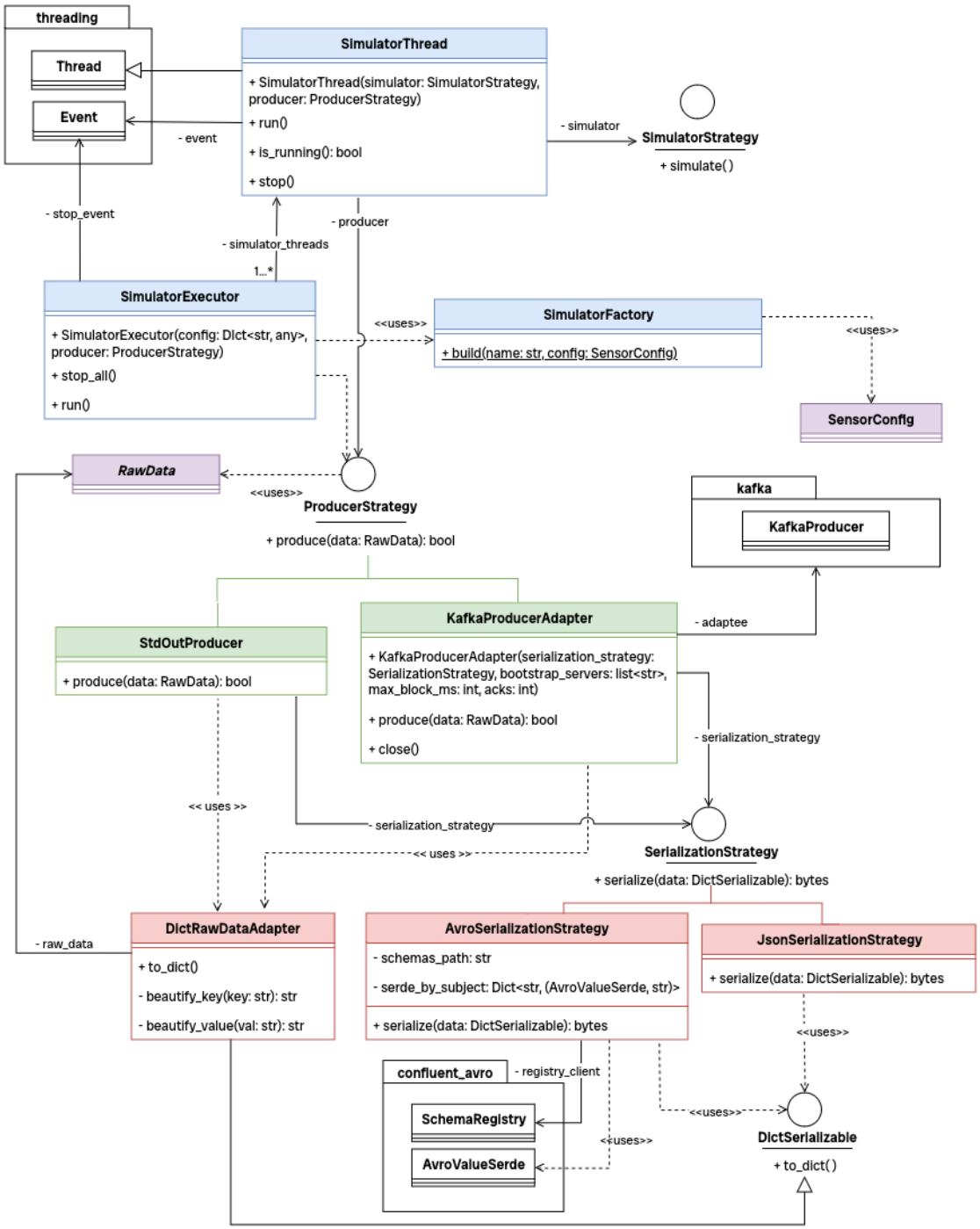

Figura 7: Diagramma delle classi modulo `producers` e `serializers`

3.3.4.2 Classi, interfacce metodi e attributi

- **Interfaccia SerializationStrategy**
  - **Metodi**:
    - `serialize(data: DictSerializable)` bytes [public]: metodo che serializza i dati grezzi in *byte*.
- **Interfaccia DictSerializable**
  - **Metodi**:
    - `to_dict()` dict [public]: metodo che restituisce un dizionario Python.
- **Classe AvroSerializationStrategy**
  - **Attributi**:
    - `schemas_path` str [private]: percorso della cartella contenente gli schemi Avro;
    - `registry_client` SchemaRegistry [private]: client per interagire con lo schema registry, contenuto nella libreria `confluent_avro`;
    - `serde_by_subject` Dict[str, (AvroValueSerde, str)] [private]: *cache* per memorizzare gli oggetti `AvroValueSerde` e gli schemi Avro associati.
  - **Metodi**:
    - `serialize(data: DictSerializable)` bytes [public]: serializza i dati grezzi in *byte* utilizzando il formato Confluent Avro.
- **Classe DictRawDataAdapter**
  - **Metodi**:
    - `to_dict()` dict [public]: restituisce un dizionario Python a partire da un'istanza di `RawData`;
    - `beautify_key(key: str)` str [private]: formatta una chiave del dizionario;
    - `beautify_value(value: object)` str [private]: formatta un valore del dizionario.
- **Classe JsonSerializationStrategy**

- **Metodi**:
  * `serialize(data: DictSerializable)` bytes [public]:

3.4 Redpanda

3.4.1 *Topic*

Nel contesto di Redpanda, un *topic* è una categoria o canale a cui vengono inviati i dati. Essi sono utili per organizzare logicamente i diversi tipi di messaggi o eventi. Nel nostro sistema, i dati grezzi provenienti dai simulatori vengono pubblicati in un *topic* differente per ciascun tipo di dato; ciò consente di elaborare in modo indipendente le varie tipologie di messaggi.

3.4.2 Partizioni e chiavi

I *topic* possono essere suddivisi in più partizioni, le quali consentono la distribuzione del carico di lavoro tra più *broker* Redpanda, allo scopo di migliorare le prestazioni e la scalabilità. Ciascuna partizione di un *topic* viene memorizzata in diversi nodi del *cluster*; la numerosità delle partizioni può essere configurata a seconda delle necessità.
Redpanda garantisce l'ordine degli eventi all'interno della stessa partizione, tuttavia di *default* non è garantito l'ordine tra partizioni diverse. Il partizionamento consente di elaborare i dati in parallelo, infatti i consumatori possono leggere da più partizioni contemporaneamente, distribuendo il carico computazionale e migliorando il *throughput*.
Ogni messaggio pubblicato è detto *record* e ha una chiave, che può essere utilizzata per determinare la partizione a cui il messaggio verrà assegnato, ed un valore, che costituisce il vero e proprio *payload*; eventi con la stessa chiave vengono inviati alla stessa partizione.
Nel caso del nostro progetto, abbiamo deciso di utilizzare come chiave il `sensor_uuid`, un identificativo univoco globale per ciascun sensore, affinché i dati siano inviati alla stessa partizione e conseguentemente elaborati nell'ordine in cui sono stati prodotti.

3.4.3 Redpanda *schema registry*

Lo *schema registry* offre un archivio centralizzato per gestire e convalidare gli schemi associati ai messaggi Kafka, facilitandone la serializzazione e deserializzazione. I produttori e consumatori dei *topic* Kafka possono utilizzare questi schemi per garantire coerenza e compatibilità dei dati durante la loro evoluzione nel tempo.

3.4.3.1 *Compatibility mode*

Le modalità di compatibilità di uno schema sono delle regole che determinano come i cambiamenti ad uno schema influiscono sulla capacità dei dati serializzati con versioni precedenti di essere letti da versioni successive e viceversa; sono essenziali per garantire che i dati rimangano compatibili durante l'evoluzione degli schemi. Di seguito sono descritte le principali modalità di compatibilità supportate dallo *schema registry* di Redpanda:

- `BACKWARD`: i consumatori che utilizzano lo schema più recente possono leggere i dati prodotti con lo schema precedente;
- `BACKWARD_TRANSITIVE`: i consumatori che utilizzano lo schema più recente possono leggere i dati prodotti con tutti gli schemi precedenti;
- `FORWARD`: i consumatori che utilizzano lo schema precedente possono leggere i dati prodotti con lo schema più recente;
- `FORWARD_TRANSITIVE`: i consumatori che utilizzano uno qualsiasi degli schemi precedenti possono leggere i dati prodotti con lo schema più recente;
- `FULL`: i dati prodotti con lo schema più recente possono essere letti da consumatori che utilizzano lo schema precedente e viceversa;
- `FULL_TRANSITIVE`: i dati prodotti con uno qualsiasi degli schemi possono essere letti da consumatori che utilizzano qualsiasi altro schema;
- `NONE`: nessun controllo di compatibilità viene effettuato.

Nel progetto proposto da *SyncLab S.r.L.* l'obiettivo principale è l'elaborazione dei dati in tempo reale piuttosto che di quelli storici, pertanto è importante che i consumatori possano sempre ricevere i messaggi più recenti, anche se prodotti con un nuovo schema. Pertanto questo tipo di applicazioni beneficiano della modalità `FORWARD`, ovvero quella che abbiamo scelto di utilizzare.

3.4.3.2 Serializzazione dei dati

3.4.3.2.1 Chiavi

Come menzionato in precedenza, si utilizza il `sensor_uuid` per popolare il campo chiave dei *record*; tale identificativo viene, prima di essere pubblicato nel *topic*, convertito a stringa e codificato in UTF-8.

3.4.3.2.2 Valori

Il formato Avro consente di definire attraverso JSON uno schema che descriva la struttura dei dati, permettendo di serializzare e deserializzarli in modo affidabile; la serializzazione tramite Avro produce dati binari compatti, che consentono di ridurre l'*overhead* di rete e migliore le prestazioni di trasmissione dei dati. Solamente i dati che rispettano lo schema definito possono essere inviati nel *topic*, garantendo la coerenza dei dati e facilitando la gestione delle evoluzioni dello schema.
Per la serializzazione dei valori abbiamo stabilito di utilizzare il formato Confluent Avro; la principale differenza rispetto al formato Avro standard è l'inclusione di un *magic byte* e dell'ID dello schema all'inizio del messaggio, seguiti dal *payload* vero e proprio. Ciò consente di evitare di includere lo schema all'interno di ogni messaggio, riducendo la dimensione dei dati trasmessi.
Il produttore consulta lo *schema registry* per ottenere l'ID corretto da utilizzare quando invia un messaggio, mentre il consumatore lo utilizza per ottenere lo schema con cui deserializzare il messaggio.

3.4.3.3 Formato dei messaggi

3.4.3.3.1 Dati grezzi prodotti dai simulatori

Per ciascun *topic* è stato definito uno schema Avro che descrive la struttura dei dati grezzi generati dai simulatori. Rispetto all'utilizzo di uno schema comune per tutti i *topic*, questa scelta consente di:

- rendere **indipendenti** i vari tipi di messaggi. Se una tipologia sensore dovesse cambiare il formato dei dati, sarebbe sufficiente modificare lo schema relativo al *topic* corrispondente;

- non dover stabilire a priori il **numero di misurazioni** che un sensore può effettuare. Se si utilizzasse uno schema comune, sarebbe necessario prevedere un numero massimo di campi, anche se non tutti i sensori potrebbero utilizzarli;
- far conoscere al consumatore il **tipo esatto** del dato che riceverà, senza dover utilizzare un campo di tipo *union*. Le misurazioni effettuate dai sensori possono essere numeri interi, decimali, stringhe o booleani;

I sensori inviano, oltre alle misurazioni relative alla propria tipologia, i campi contenuti nella seguente tabella:

Campo	Tipo	Descrizione
`sensor_uuid`	string	Identificativo univoco del sensore.
`sensor_name`	string	Nome del sensore.
`latitude`	double	Latitudine del sensore.
`longitude`	double	Longitudine del sensore.
`timestamp`	string	Data e ora della misurazione in formato ISO 8601.
`group_name`	string	Nome (opzionale) del gruppo di sensori a cui appartiene.

Un esempio di schema Avro per il tipo di dato `Temperature` è il seguente:

```
{
  "type": "record",
  "name": "Temperature",
  "fields": [
    { "name": "sensor_uuid", "type": "string" },
    { "name": "sensor_name", "type": "string" },
    { "name": "latitude", "type": "double" },
    { "name": "longitude", "type": "double" },
    { "name": "timestamp", "type": "string" },
    { "name": "value", "type": "float" },
    { "name": "group_name", "type": [ "string", "null" ] }
  ]
}
```

Listing 1: Esempio di schema Avro per il tipo di dato `Temperature`

3.4.3.3.2 Dati elaborati da Apache Flink

Per quanto riguarda invece i dati aggregati da Apache Flink, è stato definito uno schema Avro per ciascuno di essi, il quale viene pubblicato in un *topic* dedicato. Lo schema Avro per il tipo di dato `HeatIndex` è il seguente:

```
{
  "type": "record",
  "name": "Heat_Index",
```

```
  "fields": [
    {"name":"sensor_names", "type": {"type":"array", "items":"string"}},
    {"name":"group_name", "type": "string"},
    {"name":"heat_index", "type": "float"},
    {"name":"avg_temperature", "type": "float"},
    {"name":"avg_humidity", "type": "float"},
    {"name":"center_of_mass_latitude", "type": "float"},
    {"name":"center_of_mass_longitude", "type": "float"},
    {"name":"radius_in_km", "type": "float"},
    {"name":"timestamp", "type": "string"}
  ]
}
```

Listing 2: Schema Avro per il tipo di dato `HeatIndex`

```
{
  "type": "record",
  "name": "Charging_Efficiency",
  "fields": [
    { "name": "sensor_uuid", "type": "string" },
    { "name": "utilization_rate", "type": "double" },
    { "name": "efficiency_rate", "type": "double" },
    { "name": "timestamp", "type": "string" },
    { "name": "group_name", "type": "string" },
    { "name":"sensor_names", "type": { "type": "array", "items": "string" } }
  ]
}
```

Listing 3: Schema Avro per il tipo di dato `ChargingEfficiency`

3.4.3.4 Altre configurazioni

- **Numero di partizioni**: il numero di partizioni di un *topic* è stato configurato con *default* a 3;
- *Subject name strategy*, ovvero la strategia per la generazione del nome dello schema all'interno dello *schema registry*, è stata impostata a `TopicNameStrategy`, che

prevede che il nome dello schema sia composto dal nome del *topic* seguito da `-value` per i valori e `-key` per le chiavi.

3.4.4 Inizializzazione e configurazione

Apache Flink necessita che i *topic* da cui consuma, ovvero temperatura, umidità, colonnine di ricarica e parcheggi, e quelli in cui pubblica, ovvero `heat_index` e `charging_efficiency`, siano creati precedentemente all'esecuzione del *job*, con i rispettivi schemi. A tale scopo, è stata realizzata un'immagine Docker basata su Alpine Linux [Ultima consultazione 2024-07-18], al cui interno viene scaricato il binario rpk [Ultima consultazione 2024-07-18], strumento messo a disposizione da Redpanda per interagire e configurare un *cluster* Redpanda, anche remoto. Al suo interno inoltre è presente uno *script* Bash che dato il nome di un *topic* lo crea se non esiste e ne registra lo schema all'interno dello *schema registry*.

3.4.5 Redpanda Connect

Redpanda Connect è una piattaforma integrata nel sistema Redpanda, progettata per facilitare l'integrazione e il trasferimento dei dati tra Redpanda e altre fonti o destinazioni. Esso consente di gestire dei connettori, componenti *software* che si occupano automatizzare lo spostamento dei dati da e verso Redpanda. Tali connettori si dividono in due categorie:

- ***source connector***: si occupano di trasferire i dati da una sorgente esterna a Redpanda;
- ***sink connector***: si occupano di trasferire i dati da Redpanda a una destinazione esterna.

3.4.5.1 *Sink connector* per ClickHouse

All'interno del progetto abbiamo utilizzato Redpanda Connect per persistere su ClickHouse i dati provenienti dai sensori pubblicati nei differenti *topic*. Per poter effettuare questa operazione è stato necessario utilizzare un *sink connector*, che si occupasse di deserializzare i messaggi in formato Confluent Avro, effettuare il *parsing* dei campi di tipo `DateTime` (pubblicati come stringhe in formato ISO 8601) ed infine salvare i dati in ClickHouse.

La documentazione relativa è consultabile al seguente url [Ultima consultazione 2024-07-10].
La configurazione di tale connettore è disponibile all'interno del *repository* del progetto al percorso `redpanda/connectors/configs/clickhouse.json`. Al fine di effettuare il *parsing* delle date è stato necessario definire all'interno di tale file un *transformer*, il quale si occupa di leggere il campo `timestamp` e convertirlo in `DateTime`, tipo riconosciuto da ClickHouse.
La versione utilizzata è la 1.1.1, scaricabile dal seguente url [Ultima consultazione 2024-07-10].
La configurazione del *transformer* è la seguente:

```
{
    //...
    "transforms": "TimestampConverter",
    "transforms.TimestampConverter.type": "org.apache.kafka.connect.transforms.
       TimestampConverter$Value",
    "transforms.TimestampConverter.format": "yyyy-MM-dd'T'HH:mm:ss",
    "transforms.TimestampConverter.field": "timestamp",
    "transforms.TimestampConverter.target.type": "Timestamp"
    //...
}
```

Listing 4: Configurazione del *transformer* all'interno del file `clickhouse.json`

La creazione del connettore viene effettuata eseguendo il seguente comando nella radice del progetto:

```
curl "localhost:8083/connectors" -H 'Content-Type: application/json' \
    -d @./redpanda/connectors/configs/clickhouse.json
```

3.4.5.2 Avro converter

È stato inoltre necessario utilizzare un ulteriore *plugin*, `avro-converter` (versione 7.6.1), scaricabile dal seguente url [Ultima consultazione 2024-07-10], il quale consente di effettuare la deserializzazione dei messaggi in formato Confluent Avro.
Il *connector sink* è stato configurato per utilizzarlo come segue:

```
{
    //...
```

```
    "value.converter": "io.confluent.connect.avro.AvroConverter",
    "value.converter.schemas.enable": "true",
    "value.converter.schema.registry.url": "http://redpanda:8081",
    //...
}
```

Listing 5: Utilizzo del *plugin* `avro-converter` all'interno del file `clickhouse.json`

3.4.6 Redpanda Console

Redpanda Console è un'applicazione web che consente di gestire e effettuare *debug* di un'istanza Redpanda. Essa ha diverse funzionalità, tra cui:

- **visualizzazione dei messaggi**: consente di esplorare i messaggi dei *topic* attraverso *query* ad-hoc e filtri dinamici, scritti con semplici funzioni JavaScript;
- **gruppi di consumatori**: permette di visualizzare tutti i gruppi di consumatori attivi, insieme ai relativi offset, modificarli o eliminarli;
- **panoramica dei *topic***: permette di visualizzare la lista dei *topic*, controllarne la configurazione, lo spazio utilizzato, la lista dei consumatori e i dettagli delle partizioni;
- **panoramica del *cluster***: permette di visualizzare le ACL, i *broker* disponibili, il loro spazio utilizzato, l'ID del rack e altre informazioni per ottenere una panoramica del *cluster*;
- ***schema registry***: permette di visualizzare tutti gli schemi Avro, Protobuf o JSON all'interno del registro degli schemi;
- **Kafka Connect**: permette di gestire i connettori da più *cluster* di connessione, modificare le configurazioni, visualizzare lo stato corrente o riavviare i task.

3.5 Apache Flink - *Processing layer*

Nel contesto di Apache Flink, un *job* è un'applicazione che definisce una serie di operazioni di trasformazione su un flusso di dati. Essi possono essere eseguiti su *cluster* distribuiti per sfruttare la scalabilità e la potenza di calcolo necessaria per elaborare grandi quantità di dati in tempo reale. Tipicamente, un *job* consiste di tre componenti principali:

- **sorgente di dati (*source*)**: il punto di ingresso del flusso di dati, ad esempio un *topic* Kafka o un file di log;
- **transformations**: operazioni come map, filter, aggregate, join, che trasformano i dati in ingresso;
- ***sink***: il punto di uscita dove i dati elaborati vengono scritti, ad esempio un database o un altro *topic* Kafka.

Flink offre due API per la definizione dei *job*: *DataStream API* e *Table API* Il primo consente di lavorare con *stream* di dati, avendo un controllo più fine sul flusso di dati, mentre il secondo permette di lavorare con tabelle, offrendo una sintassi più simile a SQL.
Per questo progetto abbiamo utilizzato le *DataStream API* per la realizzazione di due *job* indipendenti tra loro, che tuttavia condividono alcune classi di utilità.

3.5.1 *Watermark*

I *watermark* sono un meccanismo che Flink utilizza per tracciare l'avanzamento del tempo degli eventi (il tempo in cui gli eventi si sono effettivamente verificati nel mondo reale, rispetto al tempo in cui gli eventi vengono elaborati dal sistema) per un flusso di dati. I *watermark* non sono necessari in tutti gli scenari di elaborazione dei flussi, ma quando vengono utilizzati sono emessi da Flink nel flusso di dati di origine. Non si tratta di eventi di dati veri e propri, ma di marcatori di metadati utilizzati per tracciare l'aspetto temporale del flusso durante l'elaborazione degli eventi.
In un flusso di dati, un *watermark* indica che non devono arrivare altri eventi più vecchi del timestamp del *watermark* stesso, trasmettendo l'informazione di completezza dei dati fino a quel momento. I *watermark* hanno diversi scopi fondamentali in Apache Flink:

- consentono a Flink di elaborare i flussi di dati in base all'ora di produzione dell'evento piuttosto che quella di elaborazione e consentono a Flink di tracciare la progressione dell'ora dell'evento per ogni sorgente di dati. Questo è particolarmente importante quando gli eventi arrivano fuori ordine, poiché i *watermark* assicurano che Flink elabori gli eventi nell'ordine corretto in base ai loro timestamp;
- sono fondamentali per le operazioni basate sul tempo, come l'utilizzo di funzioni di *window*, che consentono di raggruppare gli eventi in finestre temporali e calcolare risultati basati su tali finestre. Flink utilizza i *watermark* per determinare quando attivare i calcoli delle finestre ed emettere i risultati delle stesse;

- aiutano Flink a rilevare gli eventi in ritardo, cioè quelli che arrivano dopo un *watermark*, in termini di tempo. Tracciando i *watermark*, Flink può permettere all'applicazione di decidere se ignorare o elaborare tali eventi e per quanto tempo continuare ad accettare dati in ritardo.

All'interno dei due *job* implementati abbiamo assegnato i *watermark* ammettendo un ritardo massimo di 10 secondi. Il *timestamp* considerato per ciascun evento è quello relativo al campo `timestamp` presente nei dati grezzi prodotti dai simulatori e dunque non considera il tempo di elaborazione ma quello di produzione del dato.

3.5.1.1 *Heat Index*

A partire dai dati rilevati dai sensori di temperatura e umidità relativa lo *Heat Index* consente di stimare la percezione della temperatura da parte dell'essere umano. Nella configurazione di ciascun simulatore, oltre a posizione e identificativo dello stesso, è possibile specificare un `group_name`, ovvero una stringa che identifica il gruppo o zona di appartenenza; si suppone che sensori situati in posizioni geografiche vicine abbiano lo stesso `group_name`. Il *job* calcola prima separatamente la temperatura e l'umidità media per finestre di un'ora, aggregando i dati provenienti da sensori dello stesso gruppo. Successivamente con i valori ottenuti computa lo *Heat Index*, utilizzando la formula empirica ideata da Blazejczyk [Ultima consultazione 2024-06-25]. Nel risultato finale, oltre al valore dello *Heat Index*, vengono restituiti anche i valori di temperatura e umidità medi, il centro di massa del gruppo di sensori (utilizzando la formula Haversine [Ultima consultazione 2024-06-25] per il calcolo della distanza) e la distanza dal centro di massa al sensore più lontano. Questi ultimi due dati sono impiegati in una mappa interattiva su Grafana per poter disegnare un cerchio, rappresentante la zona di influenza del gruppo di sensori.

3.5.1.1.1 Modello di calcolo

3.5.1.1.1.1 Heat Index

Lo *Heat Index* viene calcolato con la seguente formula, dove T è la temperatura in gradi Celsius, R l'umidità relativa in percentuale:

$$HI = c_1 + c_2T + c_3R + c_4TR + c_5T^2 + c_6R^2 + c_7T^2R + c_8TR^2 + c_9T^2R^2$$

e i coefficienti c_i sono:

$$c_1 = -8.78469475556 \qquad c_2 = 1.61139411 \qquad c_3 = 2.33854883889$$
$$c_4 = -0.14611605 \qquad c_5 = -0.012308094 \qquad c_6 = -0.0164248277778$$
$$c_7 = 2.211732 \times 10^{-3} \qquad c_8 = 7.2546 \times 10^{-4} \qquad c_9 = -3.582 \times 10^{-6}$$

3.5.1.1.1.2 Centro di massa o centroide

Il calcolo del centro di massa tiene in considerazione la sfericità della Terra. Sia dato un insieme di punti $P = \{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\}$, dove a_i rappresenta la latitudine del punto i-esimo e b_i la longitudine. Per calcolare il centro di massa si procede, per ciascun punto, a convertire in radianti le coordinate:

$$lat_i = \frac{\pi}{180} \cdot a_i, \quad lon_i = \frac{\pi}{180} \cdot b_i$$

Successivamente si calcolano i coefficienti x_i, y_i e z_i:

$$x_i = \cos(lat_i) \cdot \cos(lon_i), \quad y_i = \cos(lat_i) \cdot \sin(lon_i), \qquad z_i = \sin(lat_i)$$

Una volta ottenuti i coefficienti per tutti i punti, si calcola la media di tutti i coefficienti:

$$x = \frac{1}{n} \sum_{i=1}^{n} x_i, \qquad y = \frac{1}{n} \sum_{i=1}^{n} y_i, \quad z = \frac{1}{n} \sum_{i=1}^{n} z_i$$

Infine si calcolano lat (latitudine in radianti), lon (longitudine in radianti) e hyp (ipotenusa ne piano cartesiano, che rappresenta la distanza dall'origine alla proiezione del punto sul piano lat_i, lon_i):

$$lon = atan2(y, x)$$
$$hyp = \sqrt{x^2 + y^2}$$
$$lat = atan2(z, hyp)$$

Una volta ottenuti lat e lon, si convertono in gradi e si ottiene il centro di massa $CM = (c_a, c_b)$.

$$c_a = \frac{180}{\pi} \cdot lat, \quad c_b = \frac{180}{\pi} \cdot lon$$

3.5.1.1.1.3 Raggio del cerchio

Il raggio del cerchio viene calcolato come la distanza dal centro di massa al punto più lontano. Sia $P = \{(a_1, b_1), (a_2, b_2), \ldots, (a_n, b_n)\}$ l'insieme di punti, $CM = (c_a, c_b)$, il raggio r è dato da:

$$r = \max_{i=1}^{n} haversine(c_a, c_b, a_i, b_i)$$

3.5.1.1.2 Flusso di dati

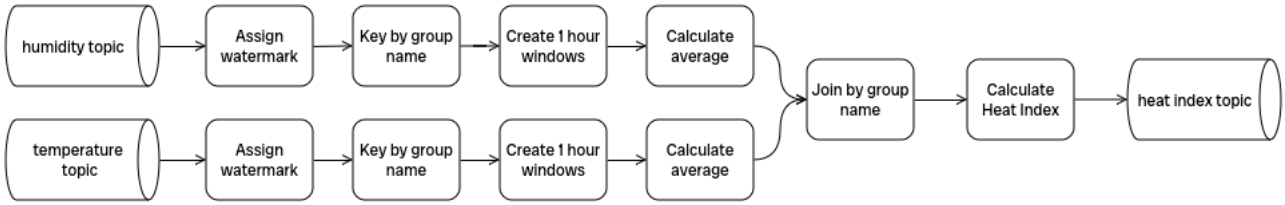

Figura 8: Flusso di dati del *job Heat Index*

I seguenti passaggi vengono eseguiti indipentendemente ed in parallelo per i dati provenienti dai *topic* di temperatura ed umidità:

1. lettura dei dati grezzi dai *topic* relativi alla temperatura e all'umidità, attraverso la classe `KafkaSource`;
2. assegnazione di un *watermark* ai dati, come precedentemente descritto nel paragrafo *watermark*;
3. raggruppamento di dati utilizzando come chiave il `group_name`, tramite la funzione `keyBy` fornita da Flink;
4. creazione di una finestra temporale di un'ora, utilizzando la funzione `window` di Flink;
5. calcolo della media della temperatura e dell'umidità, utilizzando la classe `AverageWindowFunction`;

Successivamente, attraverso l'utilizzo dell'operatore `join` si uniscono i due *stream* di dati, calcolando attraverso la classe `HeatIndexJoinFunction` il valore dello *Heat Index*, la latitudine e longitudine del centro di massa e il raggio del cerchio. Infine, i dati vengono pubblicati nel *topic* dedicato allo *Heat Index*.

3.5.1.1.3 Architettura

Al fine di realizzare un *job*, è necessario implementare una classe dotata di un metodo `main` che si occupi di inizializzare l'ambiente di esecuzione, definire sorgente e *sink* dei dati e infine di far partire il *job* vero e proprio. La classe che svolge tale ruolo per il *job Heat Index* è `HeatIndexJob`, che definisce come sorgente di dati i *topic* relativi alla temperatura e all'umidità, leggendo le variabili d'ambiente relative all'indirizzo del *bootstrap server* e *schema registry*, applica le trasformazioni illustrate nel paragrafo precedente ed infine pubblica i risultati nel *topic* dedicato allo *Heat Index*. In tale classe viene definito un metodo `execute`, il quale applica le trasformazioni a partire dai *source* definiti nel metodo *main*. Separare la logica di esecuzione del *job* dalla sua definizione consente di rendere il codice più modulare e rende possibile la realizzazione di test di integrazione sull'esecuzione dell'intero *job*, fornendo una versione *mock* di *source* e *sink* di dati.
Come illustrato nel diagramma seguente, per ciascuna fase di elaborazione dei dati viene definita una classe che estende la funzione appropriata per il tipo di operazione che si vuole effettuare. Per i casi più semplici sarebbe possibile utilizzare direttamente delle classi anonime o funzioni *lambda*, tuttavia definendo classi separate per ciascuna trasformazione consente di sottoporre ciascuna di esse a test di unità e riutilizzarle se necessario; la funzione `AverageWindowFunction` viene ad esempio utilizzata sia per calcolare la media della temperatura che dell'umidità.

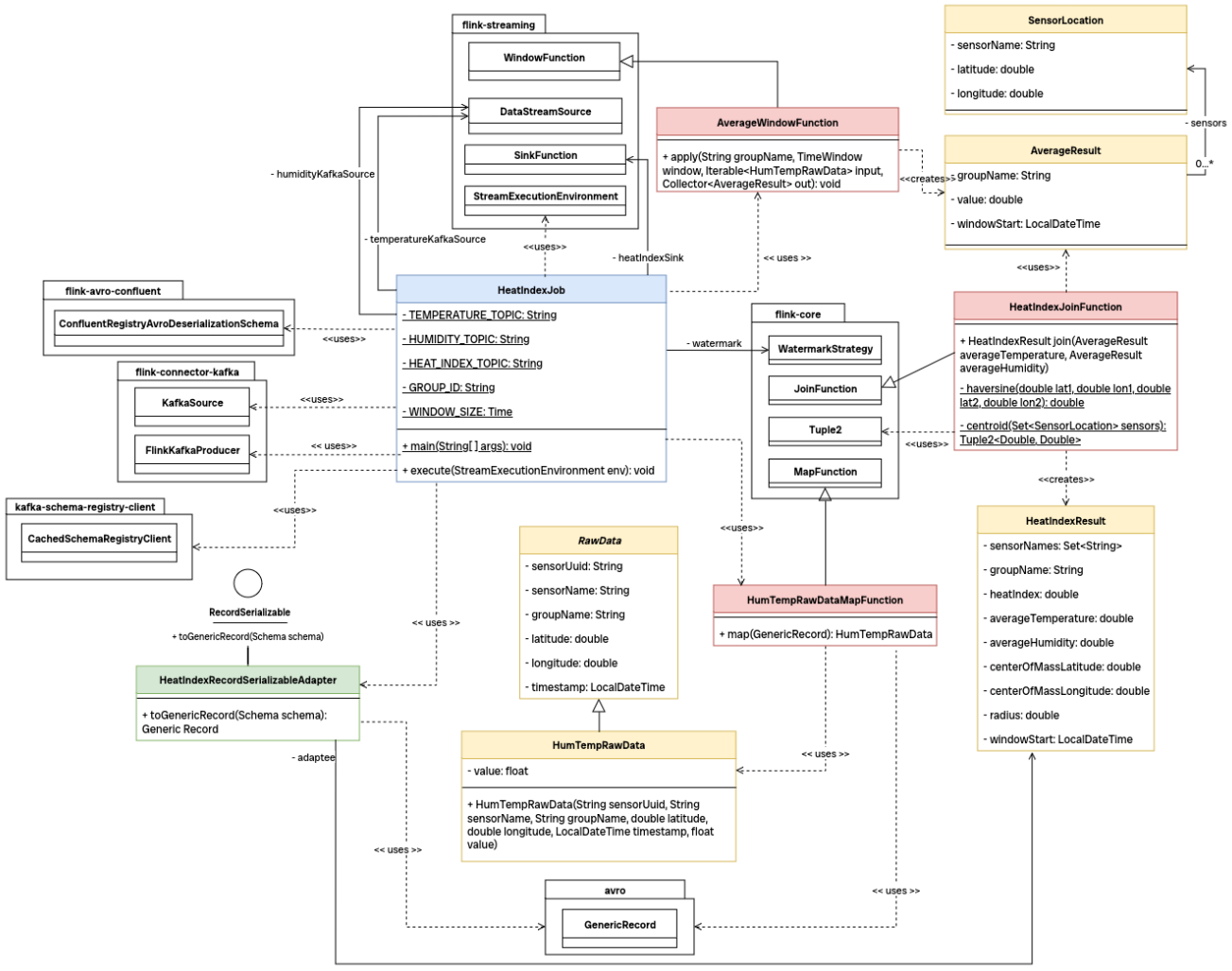

Figura 9: Architettura del *job Heat Index*

3.5.1.1.3.1 *Object adapter*

Data la necessità di convertire la classe `HeatIndexResult`, prodotto delle aggregazioni, in un oggetto di tipo `GenericRecord` (contenuto nella libreria `avro`), abbiamo implementato il *pattern object adapter*, che si occupa di effettuare tale conversione.
A tale scopo, è stata definita un'interfaccia `RecordSerializable` che espone il metodo `toGenericRecord`, il quale restituisce un oggetto di tipo `GenericRecord`.
`HeatIndexRecordSerializableAdapter` implementa tale interfaccia e ha un campo `adaptee` di tipo `HeatIndexResult`. Prima di effettuare dunque il *sink* dei dati, viene invocato il metodo `toGenericRecord` per ottenere il *record* da pubblicare nel *topic* dedicato

allo *Heat Index*.

3.5.1.1.3.2 Classi, interfacce metodi e attributi

- **Classe HeatIndexJob**
  - **Attributi**
    * `TEMPERATURE_TOPIC` str [private,final,static]: nome del *topic* relativo alla temperatura;
    * `HUMIDITY_TOPIC` str [private,final,static]: nome del *topic* relativo all'umidità;
    * `HEAT_INDEX_TOPIC` str [private,final,static]: nome del *topic* relativo allo *Heat Index*;
    * `GROUP_ID` str [private,final,static]: identificativo del gruppo di consumatori;
    * `WINDOW_SIZE` int [private,final,static]: dimensione della finestra temporale per cui calcolare le aggregazioni.
  - **Metodi**
    * `main(String[] args)` [public,static]: metodo principale che si occupa di inizializzare l'ambiente di esecuzione, definire sorgente e *sink* dei dati ed eseguire il *job* vero e proprio;
    * `execute(StreamExecutionEnvironment env)` [public]: metodo che applica le trasformazioni a partire dai *source* definiti nel metodo *main*.
- **Classe AverageWindowFunction**
  - **Metodi**
    * `apply(String groupName, TimeWindow window, Iterable<HumTempRawData> input, Collector<AverageResult> out)` void [public]: metodo che calcola la media delle misurazioni di temperatura e umidità per una certa finestra temporale.
- **Classe HeatIndexJoinFunction**
  - **Metodi**
    * `join(AverageResult averageTemperature, AverageResult averageHumidity)` HeatIndexResult [public]: metodo che calcola lo *Heat Index*, il centro di massa e il raggio del cerchio a partire dai valori di temperatura e umidità medi e i sensori che li hanno prodotti.

- **Classe HumTempRawDataMapFunction**
  - **Metodi**
    - `map(GenericRecord record)` HumTempRawData [public]: metodo che converte un `GenericRecord` in un oggetto di tipo `HumTempRawData`.
- **Interfaccia RecordSerializable**
  - **Metodi**
    - `toGenericRecord()` GenericRecord [public]: metodo che restituisce un oggetto di tipo `GenericRecord`.
- **Classe HeatIndexRecordSerializableAdapter**
  - **Attributi**
    - `adaptee` HeatIndexResult [private, final]: oggetto da adattare.
  - **Metodi**
    - `toGenericRecord()` GenericRecord [public]: metodo che restituisce un oggetto di tipo `GenericRecord` a partire da `adaptee`.
- **Classe AverageResult**
  - **Attributi**
    - `groupName` String [private]: nome del gruppo di sensori che ha prodotto la misurazione;
    - `sensors` Set <String> [private]: insieme degli identificativi dei sensori che hanno prodotto la misurazione;
    - `value` double [private]: valore medio di tutte le misurazioni considerate;
    - `windowStart` LocalDateTime [private]: data e ora di inizio della finestra temporale in cui è stata calcolata la media.
- **Classe SensorLocation**
  - **Attributi**
    - `sensorName` String [private]: nome del sensore;
    - `latitude` double [private]: latitudine del sensore;
    - `longitude` double [private]: longitudine del sensore.

- **Classe HeatIndexResult**
  - **Attributi**
    - `sensors` Set <String> [private]: insieme degli identificativi dei sensori che hanno prodotto le misurazioni;
    - `groupName` String [private]: nome del gruppo di sensori che ha prodotto la misurazione;
    - `heatIndex` double [private]: valore dello *Heat Index*;
    - `averageTemperature` double [private]: valore medio della temperatura;
    - `averageHumidity` double [private]: valore medio dell'umidità;
    - `centerOfMassLatitude` double [private]: latitudine del centro di massa;
    - `centerOfMassLongitude` double [private]: longitudine del centro di massa;
    - `radius` double [private]: raggio del cerchio;
    - `windowStart` LocalDateTime [private]: data e ora di inizio della finestra temporale in cui è stato calcolato lo *Heat Index*.
- **Classe astratta RawData**
  - **Attributi**
    - `sensorUuid` String [protected]: identificativo univoco del sensore;
    - `sensorName` String [protected]: nome del sensore;
    - `groupName` String [protected]: nome del gruppo di sensori a cui appartiene;
    - `latitude` double [protected]: latitudine del sensore;
    - `longitude` double [protected]: longitudine del sensore;
    - `timestamp` LocalDateTime [protected]: data e ora della misurazione.
- **Classe HumTempRawData**
  - **Attributi**
    - `value` double [private]: valore della misurazione.

3.5.1.1.4 *Deployment*

Per poter effettuare il *deployment* dei due *job* sviluppati è stato necessario creare un *fat jar* contenente tutte le dipendenze necessarie per l'esecuzione. Tale operazione è stata effettuata utilizzando il *plugin Maven* `maven-assembly-plugin`, con la seguente configurazione:

```
<plugin>
    <groupId>org.apache.maven.plugins</groupId>
    <artifactId>maven-assembly-plugin</artifactId>
    <version>3.7.1</version>
    <configuration>
        <descriptorRefs>
            <descriptorRef>jar-with-dependencies</descriptorRef>
        </descriptorRefs>
    </configuration>
    <executions>
        <execution>
            <id>assemble-all</id>
            <phase>package</phase>
            <goals>
                <goal>single</goal>
            </goals>
        </execution>
    </executions>
</plugin>
```

Listing 6: Configurazione del *plugin* `maven-assembly-plugin` per la creazione del *fat jar*

Al fine di poter caricare ed eseguire i *job* all'interno di un *cluster* Flink, è necessario effettuare una chiamata `POST` all'*endpoint* `/jars/upload` esposto dal *job manager* di Flink. A tale scopo, è stata realizzata un'immagine Docker chiamata `deployer` con all'interno uno *script* Python che si occupa di effettuare tale operazione. Essendo lo *script* parametrico, è sufficiente ridefinire `command` all'interno del `docker-compose.yaml`, montando la cartella contenente il *fat jar* all'interno del *container* e passando come argomento il percorso del file e le *entry class* dei due *job*.

3.5.1.2 *Charging Efficiency* (efficienza delle colonnine elettriche)

A partire dai dati rilevati dai sensori di occupazione dei parcheggi e delle colonnine elettriche, questo *job* calcola giornalmente per ciascun sensore i seguenti valori:

- `utilization_rate`: percentuale di tempo in cui le colonnine sono utilizzate rispetto al tempo totale considerato;

- `efficiency_rate`: percentuale di tempo in cui le colonnine sono utilizzate rispetto al tempo totale in cui il parcheggio è occupato.

3.5.1.2.1 Modello di calcolo

Siano dati P e C, due insiemi di misurazioni rispettivamente di parcheggi e colonnine elettriche, tali che:

- $P = \{(tp_1, o_1), (tp_2, o_2) \dots (tp_n, o_n)\}$, dove:
  - tp_i è il *timestamp* della misurazione;
  - $tp_i < tp_{i+1} \forall i \in \{1 \dots n\}$, ovvero le misurazioni sono ordinate cronologicamente;
  - o_i è un intero pari a 1 se il parcheggio è occupato, 0 altrimenti.
- $C = \{(tc_1, k_1), (tc_2, k_2) \dots (tc_m, k_m)\}$, dove:
  - tc_i è il *timestamp* della misurazione;
  - $tc_i < tc_{i+1} \forall i \in \{1 \dots m\}$, ovvero le misurazioni sono ordinate cronologicamente;
  - k_i è il numero di kwh erogati dalla colonnina elettrica al momento della misurazione.

3.5.1.2.1.1 *Utilization rate*

L' *utilization rate* è calcolato come la percentuale di tempo in cui le colonnine sono utilizzate rispetto al tempo totale considerato. Sia $T = tp_m - tp_1$ il tempo totale considerato. Il tempo in cui le colonnine sono utilizzate è dato dalla somma delle differenze tra due misurazioni consecutive in cui $k_i > 0$, ovvero

$$S = \sum_{i=1}^{m-1} (tc_{i+1} - tc_i) \cdot 1_{k_i > 0}$$

Il *utilization rate* è dunque ottenuto nel seguente modo:

$$\textit{utilization rate} = \frac{S}{T} \cdot 100$$

3.5.1.2.1.2 *Efficiency rate*

L' *efficiency rate* è calcolato come la percentuale di tempo in cui le colonnine sono utilizzate rispetto al tempo totale in cui il parcheggio è occupato. Il tempo totale in cui il parcheggio è occupato è dato dalla somma delle differenze tra due misurazioni consecutive in cui $o_i > 0$, ovvero:

$$O = \sum_{i=1}^{n-1} (tp_{i+1} - tp_i) \cdot 1_{o_i > 0}$$

Si utilizza quindi S calcolato nel paragrafo precedente per ottenere l' *efficiency rate*:

$$\textit{efficiency rate} = \frac{S}{O} \cdot 100$$

3.5.1.2.2 Flusso di dati

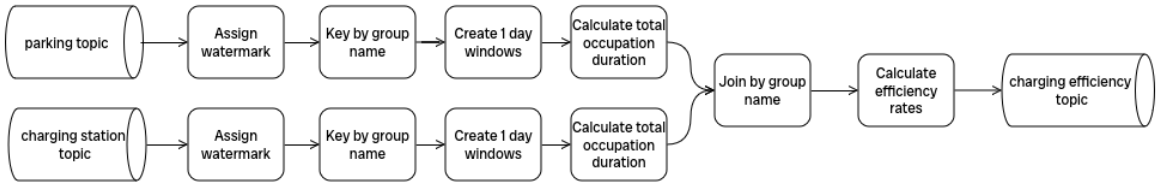

Figura 10: Flusso di dati del *job Charging Efficiency*

Le seguenti operazioni vengono eseguite indipendentemente per i dati provenienti dai *topic* relativi ai parcheggi e alle colonnine elettriche:

1. lettura dei dati grezzi dai *topic* relativi ai parcheggi e alle colonnine elettriche, attraverso la classe `KafkaSource`;
2. assegnazione di un *watermark* ai dati, come precedentemente descritto nel paragrafo *watermark*;
3. raggruppamento di dati utilizzando come chiave il `group_name`, tramite la funzione `keyBy` fornita da Flink;
4. creazione di una finestra temporale di un giorno, utilizzando la funzione `window` di Flink;
5. calcolo del tempo totale in cui le colonnine sono in uso e del tempo totale in cui i parcheggi sono occupati, utilizzando rispettivamente le classi `ChargingStationTimeDifferenceWindowFunction` e `ParkingTimeDifferenceWindowFunction`.

Successivamente, attraverso l'utilizzo dell'operatore `join` si uniscono i due *stream* di dati, calcolando attraverso la classe `ChargingEfficiencyJoinFunction` il valore dell'*utilization rate* e dell'*efficiency rate*. Infine, i risultati vengono pubblicati nel *topic* dedicato all'efficienza delle colonnine elettriche.

3.5.1.2.3 Architettura

Analogamente a quanto svolto per il *job Heat Index*, anche per il *job Charging Efficiency* è stata definita la classe `ChargingEfficiencyJob`, la quale funge da punto di ingresso per l'esecuzione del *job*. Essa definisce come sorgente di dati i *topic* relativi all'occupazione dei parcheggi e delle colonnine elettriche, prepara l'esecuzione del *job* e infine lo avvia attraverso il metodo `execute`.

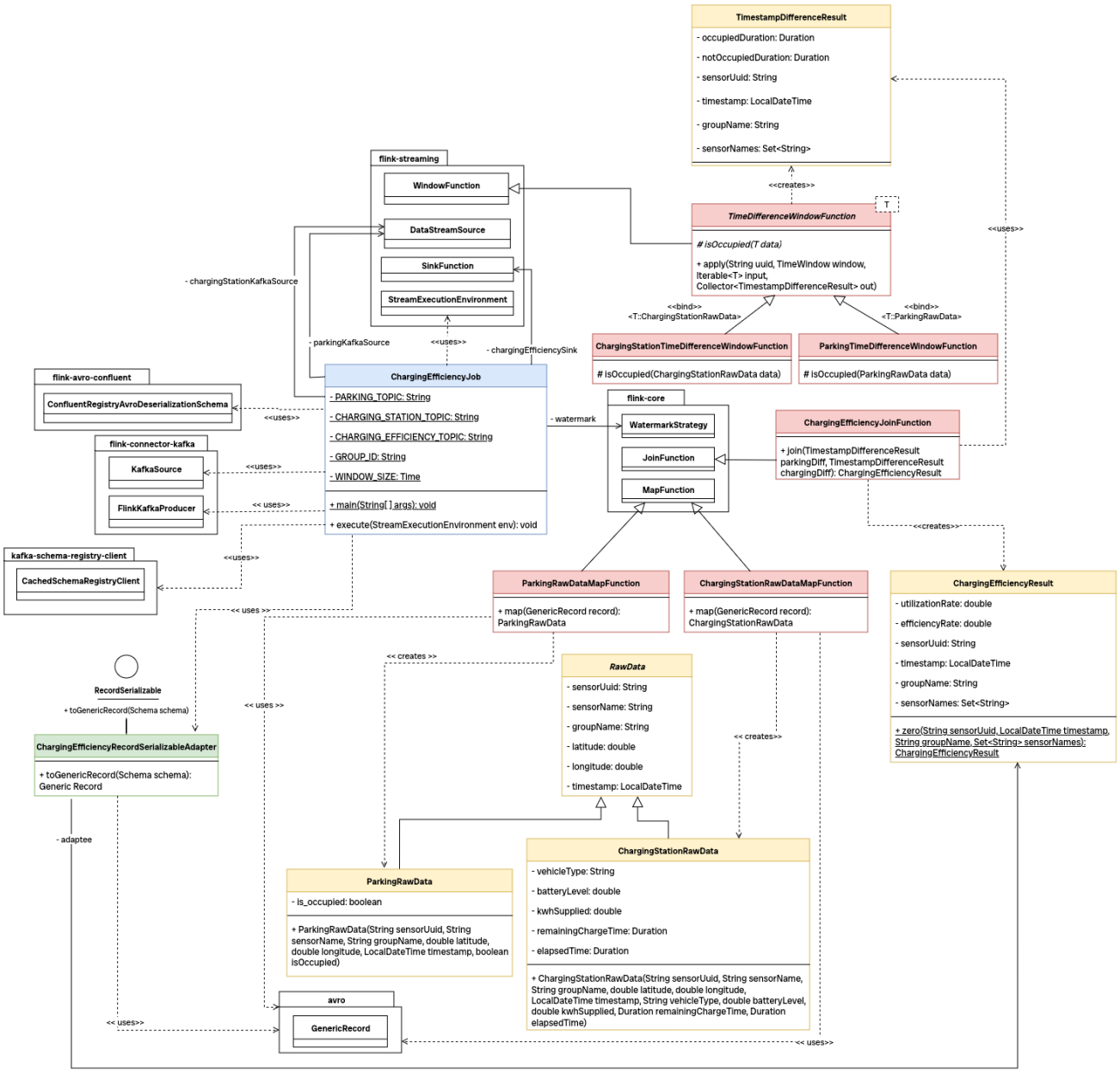

Figura 11: Architettura del *job Charging Efficiency*

3.5.1.2.3.1 *Object adapter*

La classe `ChargingEfficiencyResult` è stata adattata in un oggetto di tipo `GenericRecord` attraverso la classe `ChargingEfficiencyRecordSerializableAdapter`, analogamente a quanto descritto nella sezione *Object adapter*.

3.5.1.2.3.2 *Template method*

Per poter ottenere il tempo totale in cui le colonnine e i parcheggi sono in uso in un certo periodo di tempo è necessario utilizzare una `WindowFunction`, funzione di Flink che si occupa di calcolare una determinata aggregazione in una finestra temporale. In questo caso, si occupa calcolare la somma delle differenze tra due misurazioni consecutive in cui i kilowattora erogati dalle colonnine sono maggiori di 0 o i parcheggi sono occupati.
Risulta quindi evidente che le due funzioni di aggregazione sono molto simili tra loro, differendo solo per il campo utilizzato per determinare se la misurazione è relativa ad un momento in cui le colonnine sono in uso o i parcheggi sono occupati. Per evitare di duplicare il codice, abbiamo deciso di utilizzare il *pattern template method*, definendo una classe astratta `TimeDifferenceWindowFunction` che implementa l'interfaccia `WindowFunction` e utilizza un *generic type* `T extends RawData` per rappresentare il tipo di dato su cui effettuare l'aggregazione. `TimeDifferenceWindowFunction` definisce un metodo astratto `isOccupied(T data)` che restituisce `true` se il dato passato come argomento è relativo ad una misurazione in cui l'entità è in uso, `false` altrimenti. Inoltre, siccome implementa `WindowFunction`, deve ridefinire il metodo `apply`. Riassumendo:

- `isOccupied` è un metodo astratto e nel contesto del *pattern template method* è un metodo primitivo;
- `apply` è un metodo concreto e `final` e costituisce il *template method*;
- non sono presenti *hook*, ovvero metodi che possono opzionalmente essere sovrascritti dalle sottoclassi, definiti con implementazione di *default* nella superclasse.

3.5.1.2.3.3 Classi, interfacce metodi e attributi

- **Classe astratta RawData**: precedentemente descritta nella sezione relativa al *job Heat Index*;
- **Classe ParkingRawData**
  - **Attributi**
    - `is_occupied` boolean [private]: valore che indica se il parcheggio è occupato o meno.
- **Classe ChargingStationRawData**

  - **Attributi**
    * `vehicleType` String [private]: tipo di veicolo che la colonnina ricarica;
    * `batteryLevel` double [private]: livello di batteria del veicolo collegato alla colonnina;
    * `kwhSupplied` double [private]: kilowattora che la colonnina sta erogando al momento della misurazione;
    * `remainingChargeTime` Duration [private]: tempo di ricarica rimanente;
    * `elapsedTime` Duration [private]: tempo di ricarica trascorso;

- **Classe ChargingEfficiencyJob**
  - **Attributi**
    * `PARKING_TOPIC` str [private,final,static]: nome del *topic* relativo all'occupazione dei parcheggi;
    * `CHARGING_STATION_TOPIC` str [private,final,static]: nome del *topic* relativo all'utilizzo delle colonnine di ricarica;
    * `CHARGING_EFFICIENCY_TOPIC` str [private,final,static]: nome del *topic* relativo all'efficienza delle colonnine;
    * `GROUP_ID` str [private,final,static]: identificativo del gruppo di consumatori;
    * `WINDOW_SIZE` int [private,final,static]: dimensione della finestra temporale per cui calcolare le aggregazioni.
  - **Metodi**
    * `main(String[] args)` [public,static]: metodo principale che si occupa di inizializzare l'ambiente di esecuzione, definire sorgente e *sink* dei dati ed eseguire il *job* vero e proprio;
    * `execute(StreamExecutionEnvironment env)` [public]: metodo che applica le trasformazioni a partire dai *source* definiti nel metodo *main*.

- **Classe astratta TimeDifferenceWindowFunction**
  - **Metodi**
    * `isOccupied(T data)` boolean [protected,abstract]: metodo che restituisce `true` se il dato passato come argomento è relativo ad una misurazione in cui l'entità è in uso, `false` altrimenti;

    * `apply(String uuid, TimeWindow window, Iterable<T> input, Collector<TimestampDifferenceResult> out)` void [public]: metodo che calcola il tempo totale in cui una data entità (colonnine o parcheggi) sono in uso in un certo periodo di tempo.

- **Classe ChargingStationTimeDifferenceWindowFunction**
  - **Metodi**
    * `isOccupied(ChargingStationRawData data)` boolean [protected]: metodo che restituisce `true` se il dato passato come argomento è relativo ad una misurazione in cui le colonnine sono in uso, `false` altrimenti.
  - **Note**
    * `ChargingStationTimeDifferenceWindowFunction` estende la classe `TimeDifferenceWindowFunction` e implementa il metodo `isOccupied`.
- **Classe ParkingTimeDifferenceWindowFunction**
  - **Metodi**
    * `isOccupied(ParkingTimeDifferenceWindowFunction data)` boolean [protected]: metodo che restituisce `true` se il dato passato come argomento è relativo ad una misurazione in cui i parcheggi sono occupati, `false` altrimenti.
  - **Note**
    * `ParkingTimeDifferenceWindowFunction` estende la classe `TimeDifferenceWindowFunction` e implementa il metodo `isOccupied`.
- **Classe TimestampDifferenceResult**
  - **Attributi**
    * `occupiedDuration` Duration [private]: durata in cui l'entità è in uso;
    * `notOccupiedDuration` Duration [private]: durata in cui l'entità non è in uso;
    * `sensorUuid` String [private]: identificativo univoco del sensore;
    * `timestamp` LocalDateTime [private]: data e ora di inizio della finestra temporale in cui sono state calcolate le durate di occupazione e non occupazione;
    * `groupName` String [private]: nome del gruppo di sensori a cui appartiene il sensore;

    * `sensorNames` Set¡String¿ [private]: insieme dei nomi dei sensori che hanno prodotto le misurazioni.

- **Classe ChargingEfficiencyJoinFunction**
  - `join(TimestampDifferenceResult parkingDiff, TimestampDifferenceResult chargingDiff)`: calcola *efficiency* e *utilization rate* a partire da `TimestampDifferenceResult` di parcheggi e colonnine.
- **Classe ChargingEfficiencyResult**
  - **Attributi**:
    * `utilizationRate` double [private]: tasso di utilizzo della colonnina di ricarica;
    * `efficiencyRate` double [private]: tasso di efficienza della colonnina di ricarica;
    * `sensorUuid` String [private]: identificativo univoco del sensore;
    * `timestamp` LocalDateTime [private]: data e ora di inizio della finestra temporale in cui sono state calcolate le durate di occupazione e non occupazione;
    * `groupName` String [private]: nome del gruppo di sensori a cui appartiene il sensore;
    * `sensorNames` Set¡String¿ [private]: insieme dei nomi dei sensori che hanno prodotto le misurazioni.
  - **Metodi**:
    * `zero(String sensorUuid, LocalDateTime timestamp, String groupName, Set<String> sensorNames)` ChargingEfficiencyResult [public,static]: metodo che costruisce e ritorna un'istanza di ChargingEfficiencyResult con `utilizationRate` e `efficiencyRate` posti a 0.
- **Classe ParkingRawDataMapFunction**
  - **Metodi**
    * `map(GenericRecord record)` ParkingRawData [public]: metodo che converte un `GenericRecord` in un oggetto di tipo `ParkingRawData`.
- **Classe ChargingStationRawDataMapFunction**

  - **Metodi**
    * `map(GenericRecord record)` ChargingStationRawData [public]: metodo che converte un `GenericRecord` in un oggetto di tipo `ChargingStationRawData`.

- **Interfaccia RecordSerializable**: precedentemente descritta nella sezione relativa al *job Heat Index*;

- **Classe ChargingEfficiencyRecordSerializableAdapter**
  - **Attributi**
    * `adaptee` ChargingEfficiencyResult [private, final]: oggetto da adattare.
  - **Metodi**
    * `toGenericRecord()` GenericRecord [public]: metodo che restituisce un oggetto di tipo `GenericRecord` a partire da `adaptee`.

3.6 ClickHouse

ClickHouse viene utilizzato per memorizzare i dati grezzi provenienti dai sensori, i risultati delle elaborazioni effettuate da Apache Flink e i dati aggregati tramite *Materialized View*.

3.6.1 Funzionalità utilizzate

3.6.1.1 *Materialized View*

Le *Materialized View* in ClickHouse sono un meccanismo per memorizzare fisicamente i risultati di una *query* specifica di selezione, che viene periodicamente aggiornata in base ai dati sottostanti. Questo meccanismo consente di migliorare le prestazioni delle *query* complesse e di semplificare l'architettura del sistema, riducendo la necessità di eseguire *query* costose e complesse ogni volta che si accede ai dati.

All'interno del progetto sono state utilizzate ai seguenti scopi:

- **calcolo aggregazioni**: per calcolare la media aritmetica delle misurazioni per un certo periodo di tempo;

- **miglioramento prestazioni**: per rendere più efficienti le *query* utilizzate più frequentemente per il popolamento di grafici e dashboard;

La documentazione è disponibile al seguente link:

`https://clickhouse.com/docs/en/guides/developer/cascading-materialized-views`
[Ultima consultazione 2024-06-05]

3.6.1.2 MergeTree

MergeTree è uno dei principali motori di archiviazione di ClickHouse, progettato per gestire grandi volumi di dati e fornire elevate prestazioni di lettura e scrittura. È particolarmente adatto per applicazioni in cui i dati vengono aggiunti in modo incrementale e le *query* vengono eseguite su intervalli di tempo specifici. Le caratteristiche principali sono:

- **partizionamento**, in cui i dati vengono partizionati in base a una colonna di data o di tempo, in modo che i dati più recenti siano memorizzati in partizioni separate e possano essere facilmente eliminati o archiviati;
- **ordine dei dati**, dove i dati vengono ordinati in base a una colonna di ordinamento, in modo che i dati siano memorizzati in modo sequenziale e possano essere letti in modo efficiente;
- **indice primario**, tramite il quale i dati vengono indicizzati in base a una colonna di chiave primaria, in modo che le *query* di ricerca e di *join* siano veloci ed efficienti;
- ***merging* dei dati**, in questo modo i dati vengono uniti in modo incrementale in background, in modo che le *query* di aggregazione e di analisi siano veloci ed efficienti;
- **compressione**, i dati vengono compressi in modo efficiente per ridurre lo spazio di archiviazione e migliorare le prestazioni di lettura e scrittura;
- **replica e distribuzione**, i dati possono essere replicati e distribuiti su più nodi per garantire l'affidabilità e la disponibilità del sistema.

La documentazione è disponibile al seguente link:

`https://clickhouse.com/docs/en/engines/table-engines/mergetree-family/mergetree`
[Ultima consultazione 2024-06-05]

3.6.2 Struttura

Per ciascuna tipologia di dati o aggregazioni è stata definita una tabella all'interno del database `sensors`, mantenendo il nome esatto del *topic* da cui provengono i dati.
Le tabelle di questo tipo vengono popolate da Redpanda Connect, come precedentemente descritto nella sezione apposita. Per alcune tipologie di dati viene inoltre definita una *Materialized View* che calcola aggregazioni sui dati grezzi, in modo da rendere più efficienti le *query* che necessitano di tali elaborazioni. In particolare, viene calcolata la media aritmetica delle misurazioni per un certo periodo di tempo (differente a seconda del tipo di sensore e meglio descritto nelle sezioni seguenti), dato frequentemente utilizzato per il popolamento di grafici e *dashboard* in Grafana e che dunque necessita di essere calcolato in modo efficiente.

3.6.2.1 Misurazioni *air quality*

Di seguito viene rappresentata la tabella che contiene le misurazioni relative alla qualità dell'aria. Essa viene popolata da Redpanda Connect ed utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente.

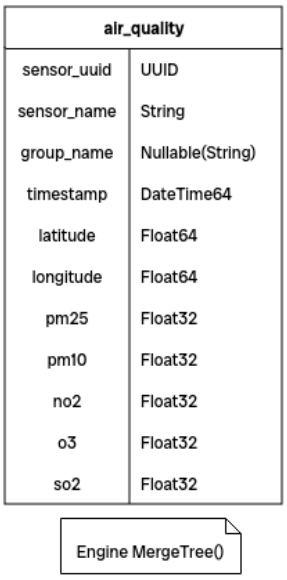

Figura 12: Tabella `air_quality`

3.6.2.2 Misurazioni *parking*

Di seguito viene rappresentata la tabella che contiene le misurazioni relative all'occupazione dei parcheggi. Essa viene popolata da Redpanda Connect ed utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente.

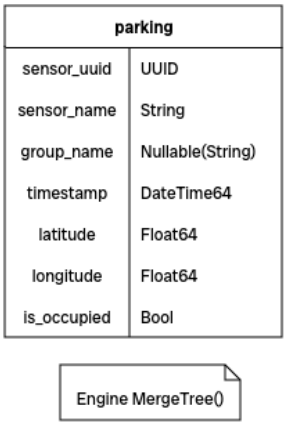

Figura 13: Tabella `parking`

3.6.2.3 Misurazioni *recycling point*

Di seguito viene rappresentata la tabella che contiene le misurazioni relative al riempimento delle isole ecologiche. Essa viene popolata da Redpanda Connect ed utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente.

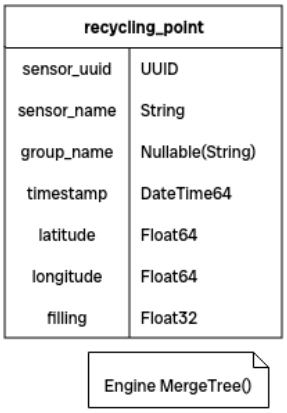

Figura 14: Tabella `recycling_point`

3.6.2.4 Misurazioni *temperature*

Di seguito viene rappresentata la configurazione per l'archiviazione delle misurazioni di temperatura. La tabella `temperature` utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente. Tramite l'utilizzo delle *Materialized View* `temperature_5m_mv`, `temperature_weekly_mv` e `temperature_daily_mv` è possibile aggregare e trasferire i dati sulle tabelle `temperature_5m`, `temperature_weekly` e `temperature_daily`, che contengono rispettivamente la media delle misurazioni per 5 minuti, giornaliere e settimanali.

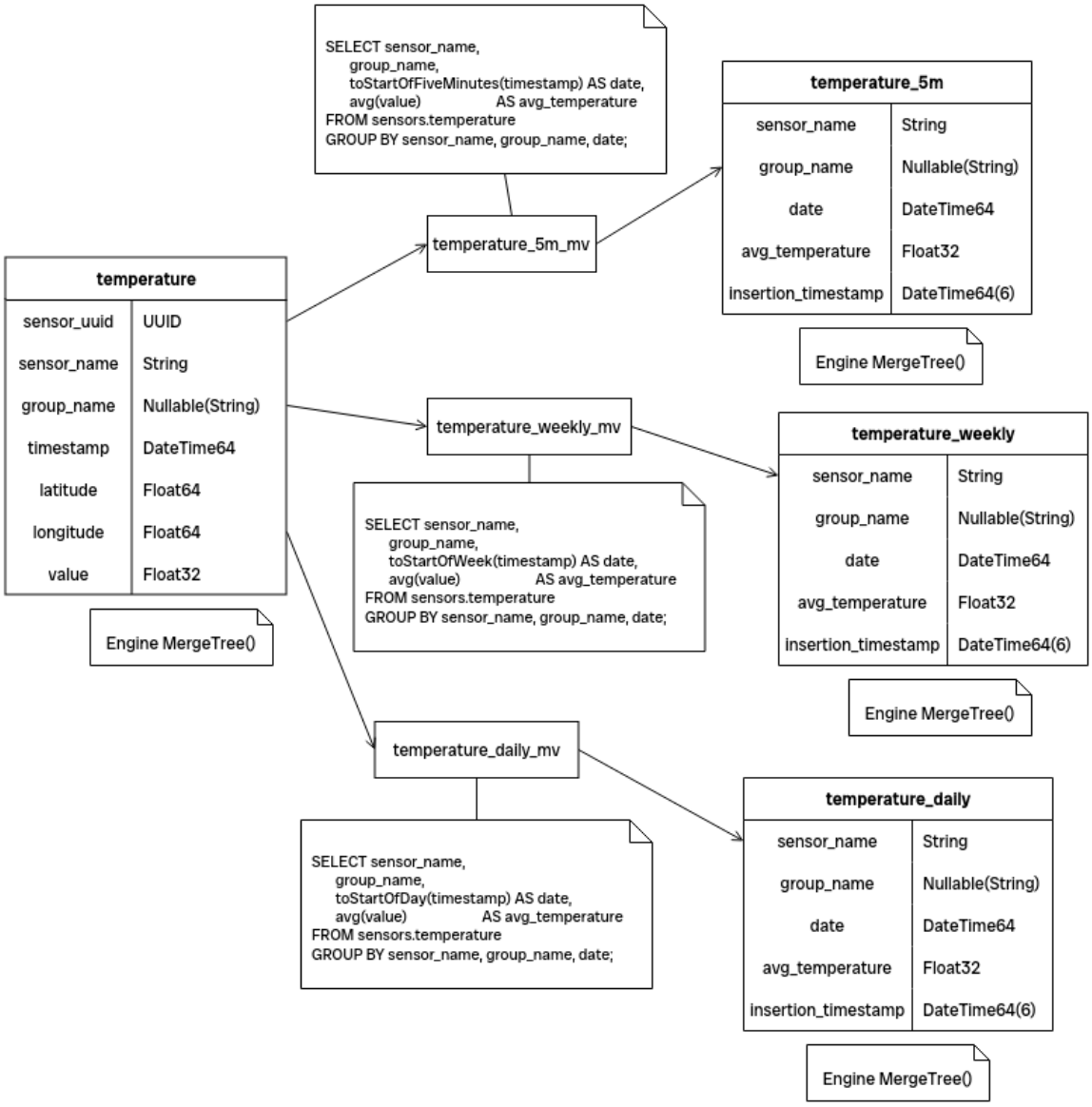

Figura 15: Tabelle e *materialized view* `temperature`

3.6.2.5 Misurazioni *traffic*

Di seguito viene rappresentata la configurazione per l'archiviazione delle misurazioni di temperatura. La tabella `traffic` utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente. Tramite l'utilizzo della *Materialized View* `traffic_1h_mv` è possibile

aggregare e trasferire i dati sulla tabella `traffic_1h`, che contiene la media del numero di veicoli e velocità per ogni ora.

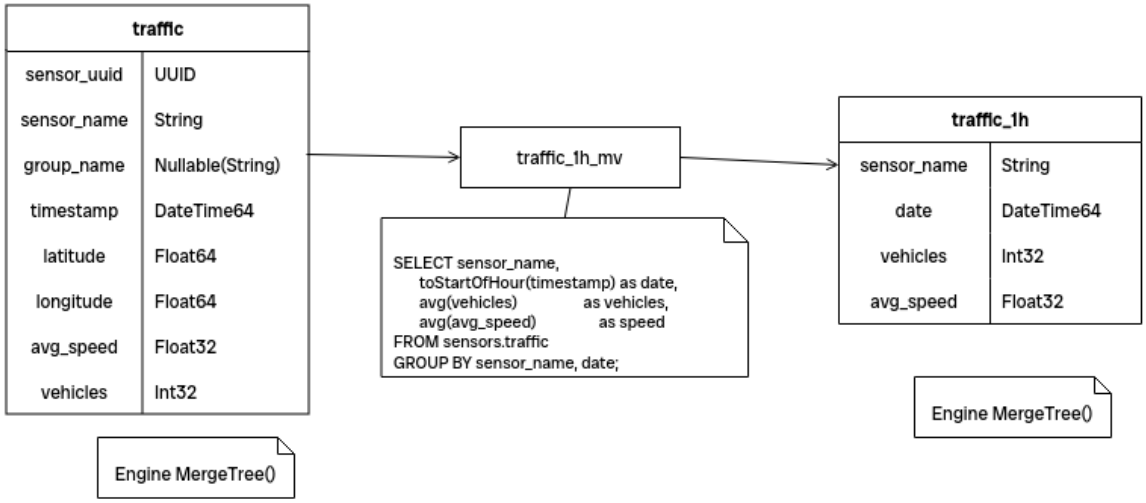

Figura 16: Tabelle e *materialized view* `traffic`

3.6.2.6 Misurazioni *charging station*

Di seguito viene rappresentata la configurazione per l'archiviazione delle misurazioni riguardanti l'occupazione delle colonnine di ricarica. La tabella `charging_station` utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente. Tramite l'utilizzo delle *Materialized View* `charging_station_5m_mv`, `charging_station_weekly_mv` e `charging_station_daily_mv` è possibile aggregare e trasferire i dati sulle tabelle `charging_station_5m`, `charging_station_weekly` e `charging_station_daily`, che contengono rispettivamente la media dei kilowattora erogati per 5 minuti, giornalmente e settimanalmente.

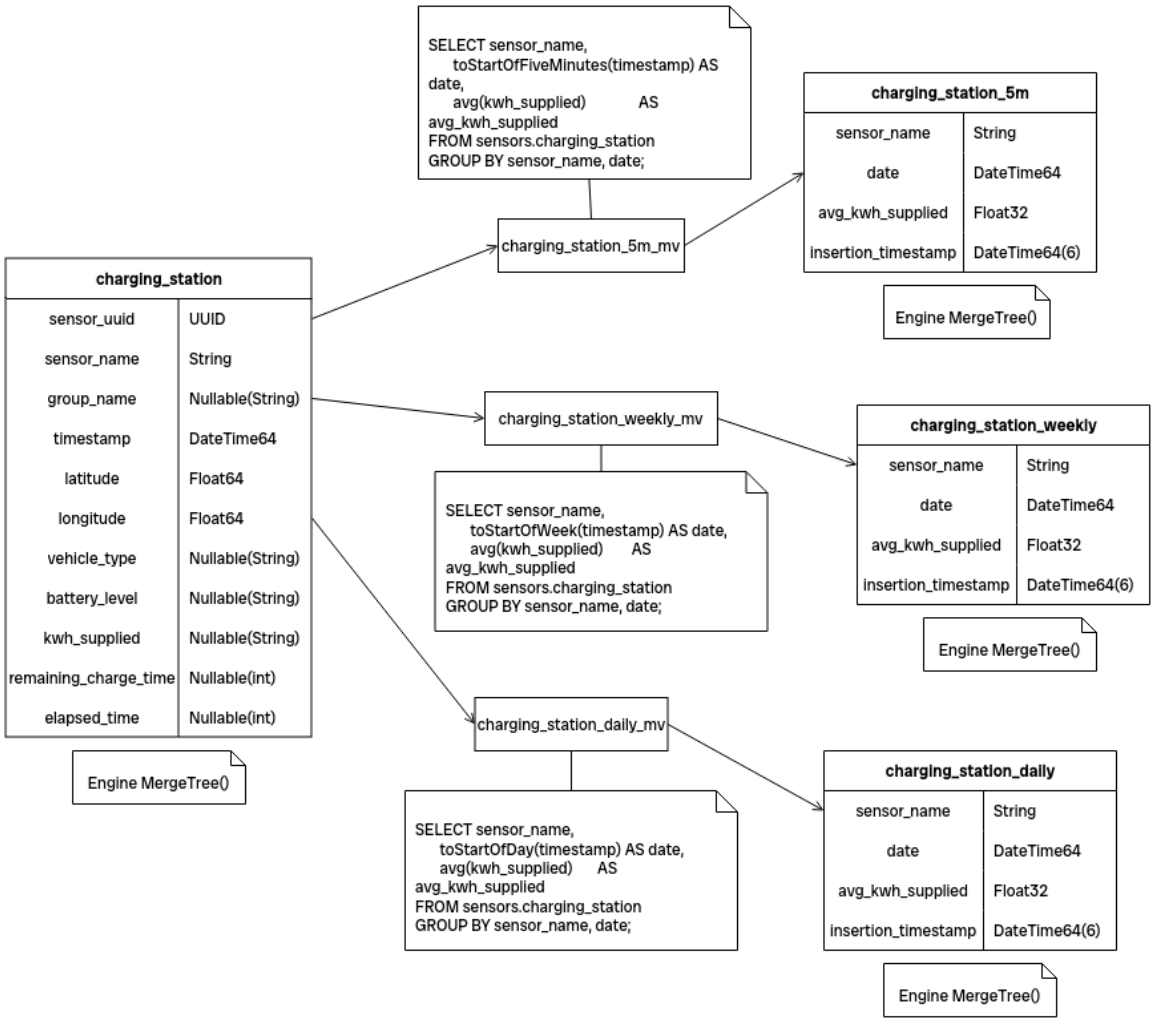

Figura 17: Tabelle e *materialized view* `charging_station`

3.6.2.7 Misurazioni *precipitation*

Di seguito viene rappresentata la configurazione per l'archiviazione delle misurazioni riguardanti le precipitazioni. La tabella `precipitation` utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente. Tramite l'utilizzo delle *Materialized View* `precipitation_1h_mv`, `precipitation_daily_mv`, `precipitation_weekly_mv` e `precipitation_yearly_mv` è possibile aggregare e trasferire i dati sulle tabelle `precipitation_1h`, `precipitation_daily`, `precipitation_weekly` e `precipitation_yearly` che

contengono la media dei millimetri di acqua caduti rispettivamente per ogni ora, giorno, settimana e anno.

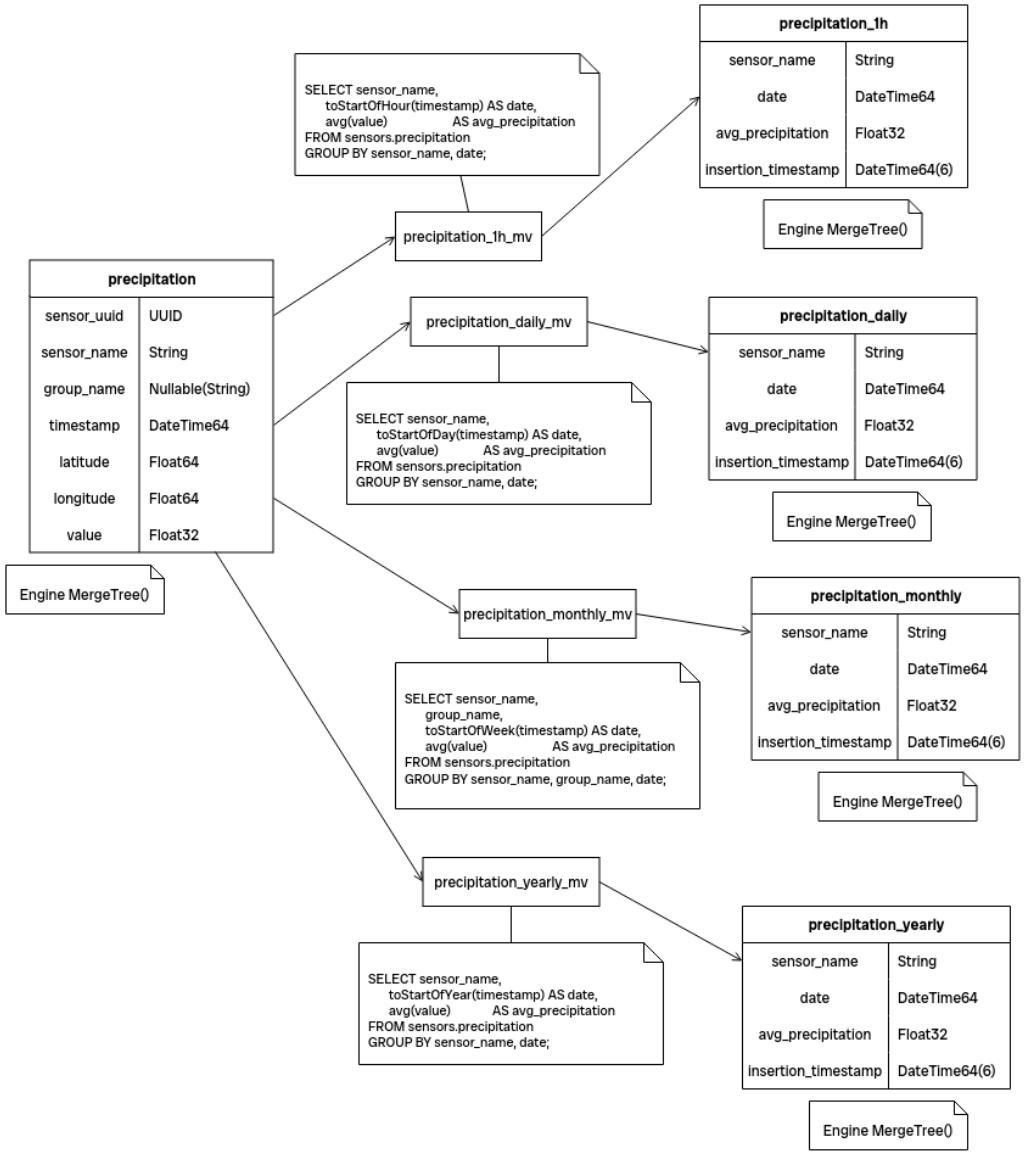

Figura 18: Tabelle e *materialized view* `precipitation`

3.6.2.8 Misurazioni *river level*

Di seguito viene rappresentata la configurazione per l'archiviazione delle misurazioni riguardanti il livello dei fiumi. La tabella `river_level` utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamen-

te, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente. Tramite l'utilizzo delle *Materialized View* `river_level_1h_mv`, `river_level_daily_mv`, `river_level_weekly_mv` e `river_level_yearly_mv` è possibile aggregare e trasferire i dati sulle tabelle `river_level_1h`,`river_level_daily`, `river_level_weekly` e `river_level_yearly` che contengono rispettivamente la media oraria, giornaliera, mensile e annuale del livello dei fiumi (misurato in metri).

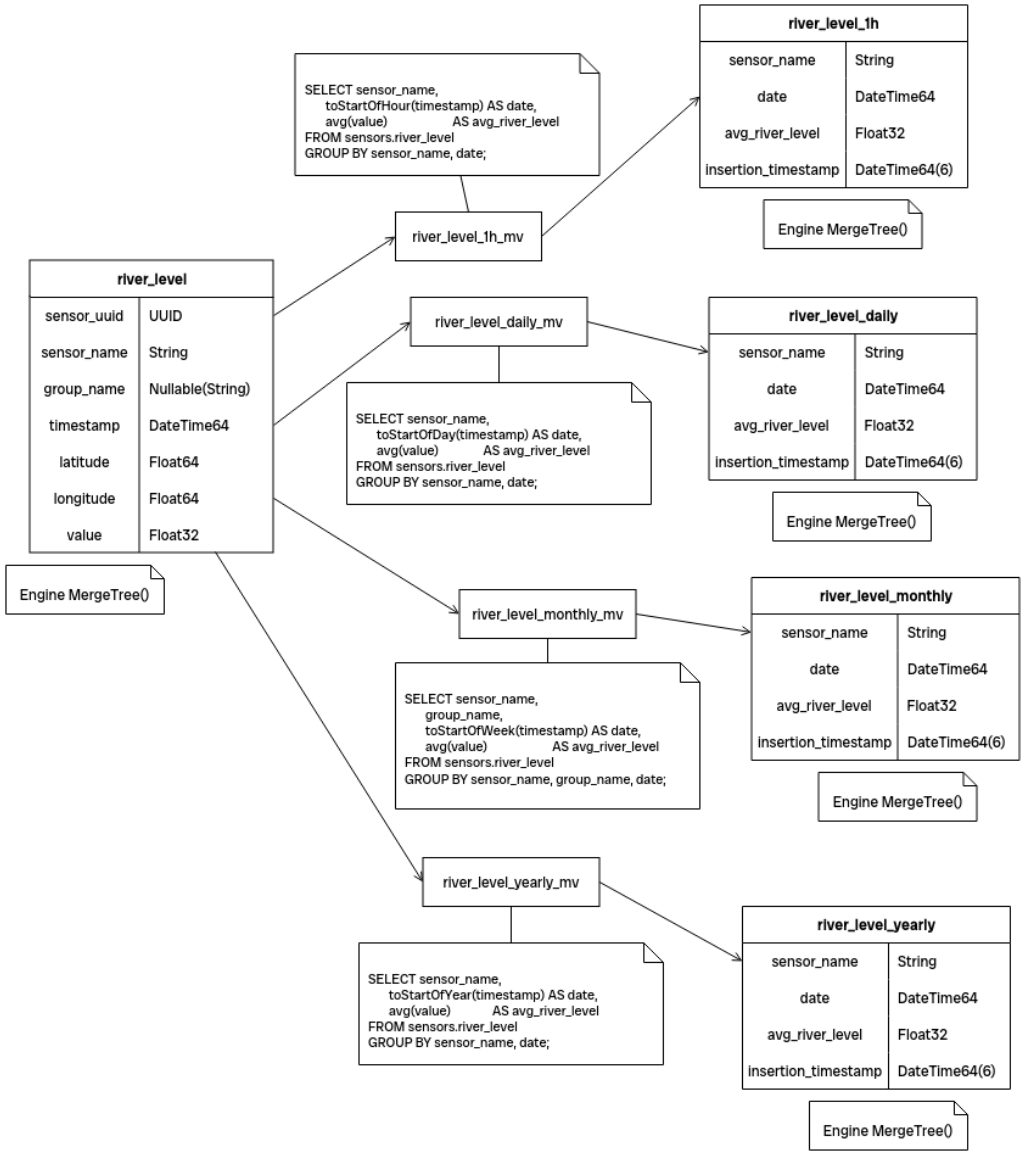

Figura 19: Tabelle e *materialized view* `river_level`

3.6.2.9 Misurazioni *humidity*

Di seguito viene rappresentata la configurazione per l'archiviazione delle misurazioni di umidità. La tabella `humidity` utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente. Tramite l'utilizzo delle *Materialized View* `humidity_5m_mv`, `humidity_weekly_mv` e `humidity_daily_mv` è possibile aggregare e trasferire i dati sulle tabelle `humidity_5m`, `humidity_weekly` e `humidity_daily`, che contengono rispettivamente la media delle misurazioni di umidità relativa per 5 minuti, giornaliere e settimanali.

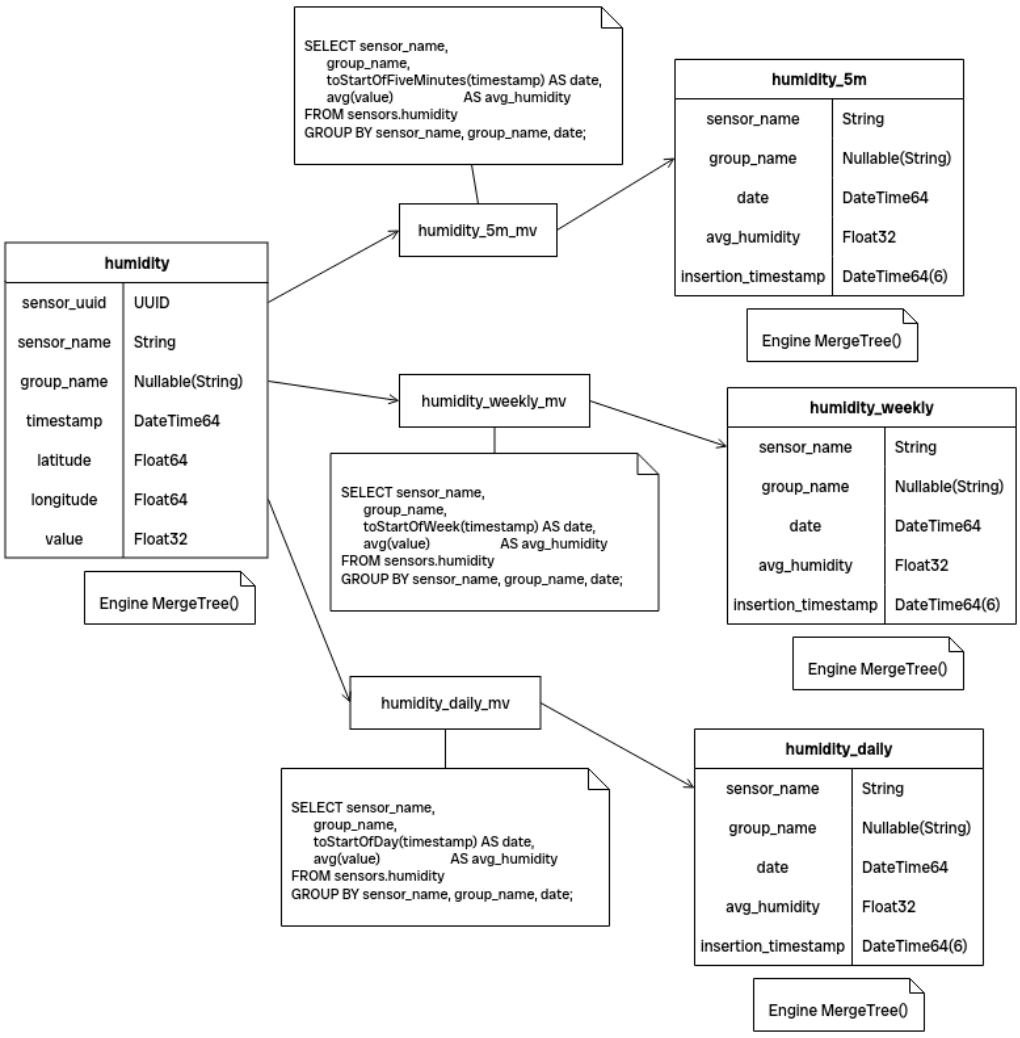

Figura 20: Tabelle e *materialized view* `humidity`

3.6.2.10 Misurazioni *heat index*

Di seguito viene rappresentata la configurazione per l'archiviazione delle misurazioni dello *heat_index*. La tabella `humidity` utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente. Tramite l'utilizzo della *Materialized View* `heat_index_daily_mv` è possibile aggregare e trasferire i dati sulla tabella `heat_index_daily` che contiene la media giornaliera delle misurazioni.

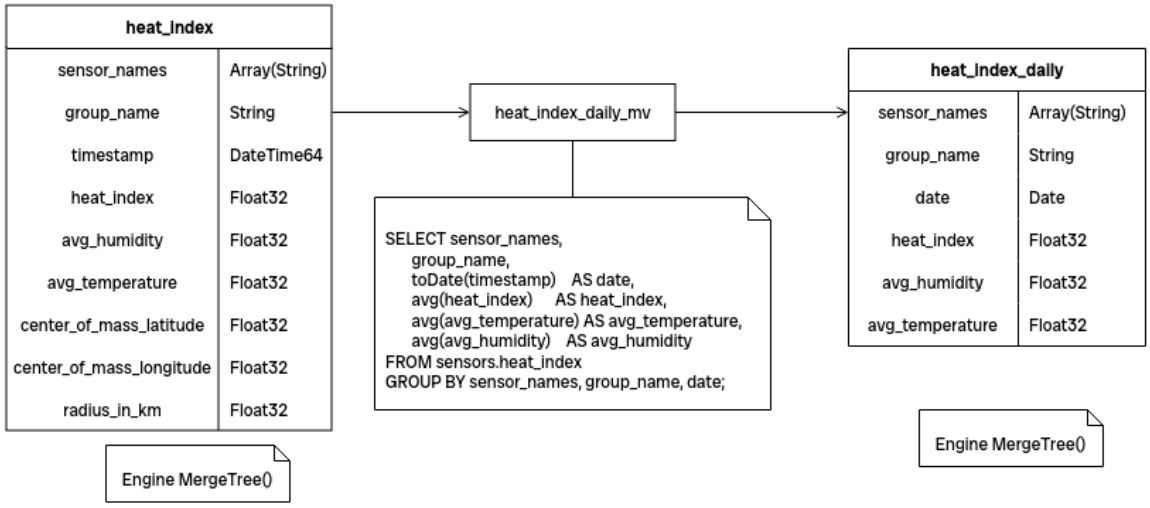

Figura 21: Tabelle e *materialized view* `heat_index`

3.6.2.11 Misurazioni *charging efficiency*

Di seguito viene rappresentata la tabella che contiene le misurazioni relative all'efficienza delle colonnine elettriche. Essa viene popolata da Redpanda Connect ed utilizza il motore di *storage* MergeTree, il quale è ottimizzato per archiviare ed analizzare dati ordinati cronologicamente, in modalità *append-only*; il suo utilizzo è motivato dal fatto che tali misurazioni sono generalmente ordinate cronologicamente.

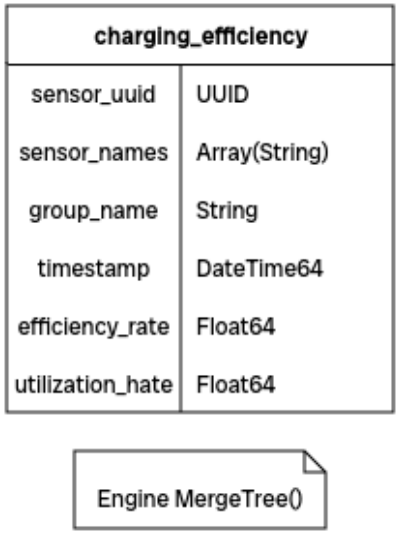

Figura 22: Tabella `charging_efficiency`

3.6.2.12 Tabella sensors

In diversi scenari all'interno di Grafana è necessario poter ottenere i dati di tutti i sensori, indipendentemente dalla loro tipologia, per mostrarli ad esempio in una mappa complessiva o in una tabella contenente la data di ricezione dell'ultimo messaggio da parte di un determinato dispositivo o ancora per la costruzione dei filtri su tipo e nome di ciascun sensore. Al fine di rendere maggiormente efficiente queste operazioni abbiamo realizzato una tabella `sensors` popolata attraverso delle *Materialized View*, che si occupano di trasferire i dati a partire dalle tabelle dei dati grezzi (come `temperature`, `humidity`...). All'interno del commento contenuto nel diagramma seguente è presente un esempio generico di *query* utilizzata per la definizione delle *Materialized View*; al posto del *placeholder* `type` è sufficiente sostituire il nome di ciascuna tabella di dati grezzi (`temperature`, `humidity`...). I campi di quest'ultime sono omessi per semplicità del diagramma e sono stati precedentemente illustrati nella sezione dedicata alle misurazioni di ciascuna tipologia.

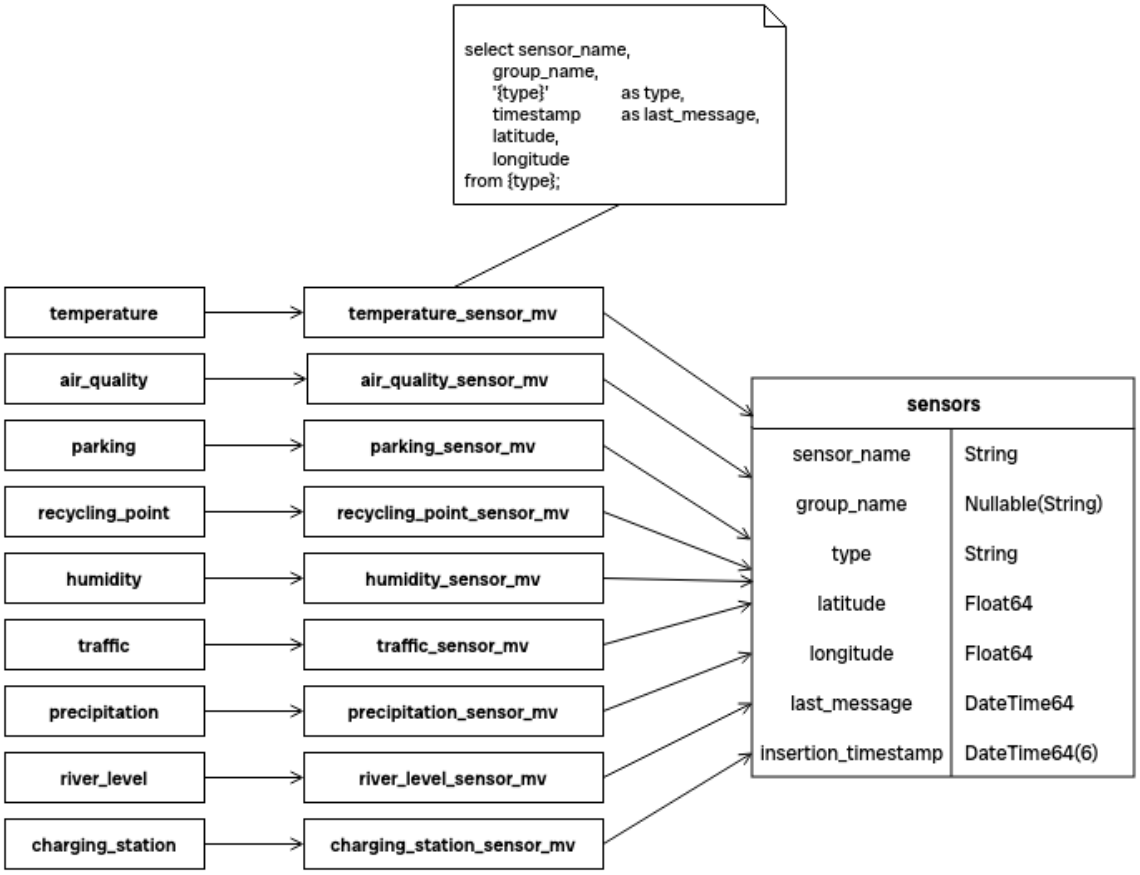

Figura 23: Tabelle e *materialized view* `sensors`

3.7 Grafana

Grafana è uno strumento di analisi e monitoraggio che permette di visualizzare dati provenienti da una varietà di fonti. È sviluppato principalmente in Go e Typescript e si è diffuso notevolmente in quanto offre funzionalità per la creazione di *dashboard* personalizzabili e intuitive.

3.7.1 Dashboard

Sono state realizzate tre *dashboard* distinte, ciascuna per adempiere a uno specifico compito:

- ***Raw data***: mostra i dati grezzi provenienti dai sensori in delle tabelle filtrabili. Inoltre contiene dei pannelli che consentono di visualizzare informazioni generali sui sensori, come una mappa con la loro posizione e una tabella con l'ultimo messaggio ricevuto da ciascuno di essi.
- ***Urban data***: mostra i dati aggregati relativi a traffico, isole ecologiche, parcheggi e colonnine di ricarica. In particolare, sono presenti grafici che mostrano l'andamento temporale delle misurazioni e delle aggregazioni calcolate.
- ***Environmental data***: mostra i dati aggregati relativi a qualità dell'aria, temperatura, umidità, precipitazioni, e livello dei fiumi. Anche in questo caso sono presenti grafici che mostrano l'andamento temporale delle misurazioni e delle aggregazioni calcolate.

La suddivisione è stata progettata in questo modo in quanto, oltre al fatto che la natura dei dati sia molto differente, le varie *dashboard* potrebbero venire utilizzate dagli utenti con esigenze diverse: i dati ambientali possono essere adoperati al fine di prevenire situazioni di emergenza correlate a condizioni climatiche avverse, mentre quelli urbanistici possono essere utilizzati con fini economici o per migliorare i servizi offerti ai cittadini.

3.7.2 ClickHouse datasource plugin

Il plugin ClickHouse per Grafana è un'implementazione che consente di utilizzare ClickHouse come fonte di dati per Grafana. Questo plugin facilita la connessione e l'interrogazione dei dati archiviati in ClickHouse direttamente da Grafana. La documentazione è disponibile al seguente link:

https://grafana.com/grafana/plugins/grafana-clickhouse-datasource/ [Ultima consultazione 2024-06-05]

La configurazione di tale *datasource* è contenuta nella cartella `grafana/provisioning/datasources/default.yaml` della repository del progetto.

3.7.3 Variabili Grafana

Al fine di rendere le *dashboard* più flessibili e personalizzabili, sono state utilizzate le variabili di Grafana. Esse consentono di definire parametri dinamici che possono essere utilizzati per filtrare, raggruppare o personalizzare i dati visualizzati nei pannelli delle dashboard. La documentazione è disponibile al seguente link:

https://grafana.com/docs/grafana/latest/dashboards/variables/ [Ultima consultazione 2024-06-05]

3.7.3.1 Variabili nella *dashboard Raw data*

- `sensor_type`: permette di filtrare i dati visualizzati in base al tipo di sensore;
- `sensor_name`: dipendente dalla variabile `sensor_type` e permette di filtrare i dati visualizzati in base al nome del sensore;

3.7.3.2 Variabili nella *dashboard Urban data*

- `group_name_charging`: permette di filtrare i dati visualizzati in base al `group_name` di colonnine di ricarica;
- `group_name_parking`: permette di filtrare i dati visualizzati in base al `group_name` di parcheggi;
- `sensor_name_recycling_point`: permette di filtrare i dati visualizzati in base al nome del sensore di isole ecologiche;
- `sensor_name_traffic`: permette di filtrare i dati visualizzati in base al nome del sensore di traffico;

3.7.3.3 Variabili nella *dashboard Environmental data*

- `group_name_temperature`: permette di filtrare i dati visualizzati in base al `group_name` di sensori di temperatura;
- `sensor_name_humidity`: permette di filtrare i dati visualizzati in base al `group_name` di sensori di umidità;
- `sensor_name_precipitation`: permette di filtrare i dati visualizzati in base al `group_name` di sensori di precipitazioni;
- `sensor_name_river_level`: permette di filtrare i dati visualizzati in base al `group_name` di sensori di livello dei fiumi;
- `sensor_name_air_quality`: permette di filtrare i dati visualizzati in base al `group_name` di sensori di qualità dell'aria;

3.7.4 Grafana Alerts

Gli *alert* di Grafana sono una funzionalità che permettono di definire, configurare e gestire avvisi basati su condizioni specifiche rilevate nei dati monitorati. Questi avvisi consentono agli utenti di essere informati tempestivamente su eventuali problemi o cambiamenti critici nei loro sistemi, applicazioni o infrastrutture.
La documentazione è disponibile al seguente link:

`https://grafana.com/docs/grafana/latest/alerting/` [Ultima consultazione 2024-06-05]

3.7.4.1 Configurazione delle regole di alert

Definiscono le condizioni che devono essere soddisfatte per attivare un alert. Gli eventi che generano un alert sono:

- temperatura maggiore di 40°C per più di 30 minuti;
- isola ecologica piena al 100% per più di 24 ore;
- superamento dell'indice 3 dell'EAQI (indice di qualità dell'aria);
- livello di precipitazioni superiore a 10 mm in 1 ora.

Gli alert possono possedere tre diversi tipi di stati:

- ***normal***, indica che l'alert non è attivo perché le condizioni definite per l'attivazione dell'avviso non sono soddisfatte;
- ***pending***, indica che le metriche monitorate stanno iniziando a deviare dalle condizioni normali ma non hanno ancora soddisfatto completamente le condizioni per attivare l'alert;
- ***firing***, significa che le condizioni definite per l'avviso sono state soddisfatte e l'alert è attivo.

3.7.4.2 Configurazione canale di notifica

Per configurare un canale di notifica è necessario:

1. nel menù di sinistra, cliccare sull'icona "Alerting";
2. selezionare la voce "Notification channels";
3. cliccare sul pulsante "Add channel" per aggiungere un nuovo canale di notifica;
4. selezionare il tipo di canale di notifica desiderato tra quelli disponibili;
5. configurare le impostazioni del canale di notifica in base alle proprie esigenze;
6. cliccare sul pulsante "Save" per salvare le impostazioni del canale di notifica.

3.7.5 Altri plugin

3.7.5.1 Orchestra Cities Map plugin

Progettato per facilitare la visualizzazione e l'analisi dei dati geospaziali all'interno di piattaforme di pianificazione urbana e sviluppo territoriale.
Le principali funzionalità offerte da questo plugin sono:

- **visualizzazione dei dati geospaziali**: consente agli utenti di visualizzare dati geografici, come mappe, strati di dati GIS (*Geographic Information System*), punti di interesse e altre informazioni territoriali;
- **interfaccia interattiva**: offre un'interfaccia utente intuitiva e interattiva che consente agli utenti di esplorare e interagire con i dati geospaziali in modo dinamico;

- **personalizzazione**: offre opzioni di personalizzazione per adattarsi alle esigenze specifiche dell'utente o dell'applicazione;
- **analisi dei dati**: oltre alla semplice visualizzazione dei dati geospaziali, il plugin può anche supportare funzionalità avanzate di analisi dei dati, come l'identificazione di cluster, la creazione di heatmap e l'esecuzione di analisi spaziali per identificare tendenze o pattern significativi nei dati territoriali;
- **integrazione**: è progettato per integrarsi facilmente con altre componenti dell'ecosistema Orchestra Cities e con altre piattaforme software di pianificazione urbana e sviluppo territoriale.

La documentazione è disponibile al seguente link:

`https://grafana.com/grafana/plugins/orchestracities-map-panel/?tab=installation`
[Ultima consultazione 2024-06-05]

4 Requisiti

4.1 Requisiti funzionali

Codice	Importanza	Stato	Descrizione
RF-1	Obbligatorio	Soddisfatto	La parte *IoT* dovrà essere simulata attraverso tool di generazione di dati casuali che tuttavia siano verosimili.
RF-2	Obbligatorio	Soddisfatto	Il sistema dovrà permettere la visualizzazione dei dati in tempo reale.
RF-3	Obbligatorio	Soddisfatto	Il sistema dovrà permettere la visualizzazione dei dati storici.
RF-4	Obbligatorio	Soddisfatto	L'utente deve poter accedere all'applicativo senza bisogno di autenticazione.
RF-5	Obbligatorio	Soddisfatto	L'utente dovrà poter visualizzare su una mappa la posizione geografica dei sensori.
RF-6	Obbligatorio	Soddisfatto	I tipi di dati che il sistema dovrà visualizzare sono: temperatura, umidità, qualità dell'aria, precipitazioni, traffico, stato delle colonnine di ricarica, stato di occupazione dei parcheggi, stato di riempimento delle isole ecologiche e livello di acqua.
RF-7	Obbligatorio	Soddisfatto	I dati dovranno essere salvati su un database OLAP.
RF-8	Obbligatorio	Soddisfatto	I sensori di temperatura rilevano i dati in gradi Celsius
RF-9	Obbligatorio	Soddisfatto	I sensori di umidità rilevano la percentuale di umidità nell'aria.

Codice	Importanza	Stato	Descrizione
RF-10	Obbligatorio	Soddisfatto	I sensori livello acqua rilevano il livello di acqua nella zona di installazione
RF-11	Obbligatorio	Soddisfatto	I dati provenienti dai sensori dovranno contenere i seguenti dati: id sensore$_G$, data, ora e valore.
RF-12	Obbligatorio	Soddisfatto	Sviluppo di componenti quali widget$_G$ e grafici per la visualizzazione dei dati nelle dashboard$_G$.
RF-13	Obbligatorio	Soddisfatto	Il sistema deve permettere di visualizzare una dashboard$_G$ generale con tutti i dati dei sensori.
RF-14	Obbligatorio	Soddisfatto	Il sistema deve permettere di visualizzare una dashboard$_G$ contenente tutti i dati dei sensori che monitorano l'ambiente.
RF-15	Obbligatorio	Soddisfatto	Il sistema deve permettere di visualizzare una dashboard$_G$ contenente tutti i dati dei sensori che monitorano gli aspetti urbani.
RF-16	Obbligatorio	Soddisfatto	Il sistema deve permettere di visualizzare una sezione specifica per ciascuna categoria di sensori.

Codice	Importanza	Stato	Descrizione
RF-17	Obbligatorio	Soddisfatto	Nella dashboard$_G$ dei dati grezzi dovranno essere presenti: una mappa interattiva, un widget$_G$ con il conteggio totale dei sensori divisi per tipo, una tabella contente tutti i sensori e la data in cui essi hanno trasmesso l'ultima volta. Inoltre verranno mostrate delle tabelle con i dati filtrabili suddivisi per sensore$_G$ e un grafico time series$_G$ con tutti i dati grezzi.
RF-18	Obbligatorio	Soddisfatto	Nella dashboard$_G$ dei dati ambientali dovranno essere presenti delle sezioni contenenti i panel relativi ai sensori di temperatura, umidità, precipitazioni, livello dei fiumi e qualità dell'aria.
RF-19	Obbligatorio	Soddisfatto	Nella dashboard$_G$ dei dati legati agli aspetti urbani dovranno essere presenti delle sezioni contenenti i panel relativi ai sensori di parcheggio, traffico, isole ecologiche e colonnine di ricarica.
RF-20	Obbligatorio	Soddisfatto	Nella sezione della temperatura dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, la temperatura media, minima e massima di un certo periodo di tempo, la temperatura in tempo reale e la temperatura media per settimana e mese.

Codice	Importanza	Stato	Descrizione
RF-21	Obbligatorio	Soddisfatto	Nella sezione dell'umidità dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, l'umidità media, minima e massima di un certo periodo di tempo e l'umidità in tempo reale.
RF-22	Obbligatorio	Soddisfatto	Nella sezione della qualità dell'aria dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, la qualità media dell'aria in un certo periodo e in tempo reale, i giorni con la qualità dell'aria migliore e peggiore in un certo periodo di tempo.
RF-23	Obbligatorio	Soddisfatto	Nella sezione delle precipitazioni dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, la quantità media di precipitazioni in un certo periodo e in tempo reale, i giorni con la quantità di precipitazioni maggiore e minore in un certo periodo di tempo.
RF-24	Obbligatorio	Soddisfatto	Nella sezione del livello di acqua dovranno essere visualizzati: un grafico time series$_G$, una mappa interattiva, il livello medio di acqua in un certo periodo e in tempo reale.

Codice	Importanza	Stato	Descrizione
RF-25	Obbligatorio	Soddisfatto	Nella sezione delle isole ecologiche dovranno essere visualizzati: una mappa interattiva con il rispettivo stato di riempimento e il conteggio di isole ecologiche suddivise per stato di riempimento in tempo reale.
RF-26	Obbligatorio	Soddisfatto	Nella sezione dei parcheggi dovranno essere visualizzati: una mappa interattiva con il rispettivo stato di occupazione e il conteggio di parcheggi suddivisi per stato di occupazione in tempo reale.
RF-27	Obbligatorio	Soddisfatto	Nella sezione delle colonnine di ricarica dovranno essere visualizzati: una mappa interattiva contenente anche lo stato e il numero di colonnine di ricarica suddivise per stato in tempo reale.
RF-28	Obbligatorio	Soddisfatto	Nella sezione del traffico dovranno essere visualizzati: un grafico time series$_G$, il numero di veicoli e la velocità media in tempo reale, il calcolo dell'ora di punta sulla base del numero di veicoli e velocità media.
RF-29	Obbligatorio	Soddisfatto	Nel caso in cui non ci siano dati visualizzabili, il sistema deve notificare l'utente mostrando un opportuno messaggio.
RF-30	Obbligatorio	Soddisfatto	I sensori di qualità dell'aria inviano i seguenti dati: *PM10*, *PM2.5*, *NO2*, *CO*, *O3*, *SO2* in $\mu g/m^3$.

Codice	Importanza	Stato	Descrizione
RF-31	Obbligatorio	Soddisfatto	I sensori di precipitazioni inviano la quantità di pioggia caduta in mm.
RF-32	Obbligatorio	Soddisfatto	I sensori di traffico inviano il numero di veicoli rilevati e la velocità in km/h.
RF-33	Obbligatorio	Soddisfatto	Le colonnine di ricarica inviano lo stato di occupazione e il tempo mancante alla fine della ricarica (se occupate) o il tempo passato dalla fine dell'ultima ricarica (se libere).
RF-34	Obbligatorio	Soddisfatto	I sensori di parcheggio inviano lo stato di occupazione del parcheggio (1 se occupato, 0 se libero) e il timestamp dell'ultimo cambiamento di stato.
RF-35	Obbligatorio	Soddisfatto	Le isole ecologiche inviano lo stato di riempimento come percentuale.
RF-36	Obbligatorio	Soddisfatto	I sensori di livello di acqua inviano il livello di acqua in cm.
RF-37	Obbligatorio	Soddisfatto	Il sistema deve permettere di filtrare i dati visualizzati in base a un intervallo di tempo.
RF-38	Obbligatorio	Soddisfatto	Il sistema deve permettere di filtrare i dati visualizzati in base al sensore$_G$ che li ha generati.
RF-39	Desiderabile	Soddisfatto	Devono essere messe in relazione più sorgenti di dati.
RF-40	Desiderabile	Soddisfatto	Nei grafici time series$_G$ i dati devono essere aggregati calcolando la media di 5 minuti, in modo da risultare più leggibili.

Codice	Importanza	Stato	Descrizione
RF-41	Obbligatorio	Soddisfatto	Deve essere implementato almeno un simulatore di dati.
RF-42	Desiderabile	Soddisfatto	Devono essere implementati più simulatori di dati.
RF-43	Obbligatorio	Soddisfatto	I simulatori devono produrre dei dati verosimili.
RF-44	Obbligatorio	Soddisfatto	Per ciascuna tipologia di sensore$_G$ dev'essere sviluppata almeno una sezione.
RF-45	Opzionale	Non soddisfatto	Deve essere implementata una funzionalità di previsione di dati futuri della temperature, basandosi sui dati dell'anno e della settimana precedente.
RF-46	Desiderabile	Soddisfatto	Deve esistere una dashboard$_G$ per la visualizzazione della posizione geografica dei sensori su una mappa.
RF-47	Opzionale	Soddisfatto	Deve essere presente un sistema di notifiche che allerti l'utente nel caso in cui la temperatura superi i 40°C per più di 30 minuti.
RF-48	Opzionale	Soddisfatto	Deve essere presente un sistema di notifiche che allerti l'utente se un'isola ecologica rimane al 100% di riempimento per più di 24 ore.
RF-49	Opzionale	Soddisfatto	Deve essere presente un sistema di notifiche che allerti l'utente se la qualità dell'aria supera l'indice 3 dell'EAQI.
RF-50	Opzionale	Soddisfatto	Deve essere presente un sistema di notifiche che allerti l'utente se la quantità di precipitazioni supera i 10mm in un'ora.

Codice	Importanza	Stato	Descrizione
RF-51	Opzionale	Soddisfatto	Deve essere implementato il calcolo dell'indice di qualità dell'aria EAQI.
RF-52	Opzionale	Soddisfatto	Deve essere implementato il calcolo dell'indice di temperatura percepita Heat Index, combinando i dati provenienti dai sensori di temperatura e umidità.
RF-53	Opzionale	Soddisfatto	Devono essere combinati i dati provenienti dalle colonnine di ricarica e dai parcheggi per calcolare quanti parcheggi sono stati utilizzati da veicoli elettrici e se il parcheggio ha fruttato abbastanza per coprire i costi di installazione.
RF-54	Obbligatorio	Soddisfatto	Il sistema deve permettere di filtrare i dati visualizzati in base al tipo di sensore che li ha prodotti.

Tabella 3: Requisiti funzionali

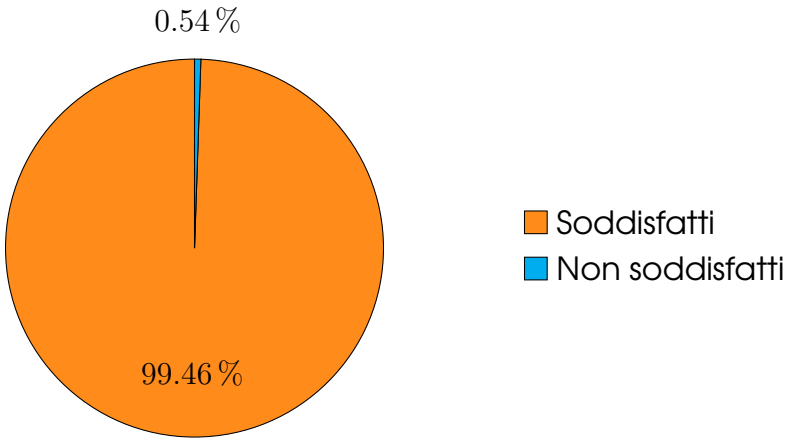

Figura 24: Percentuale di soddisfacimento dei requisiti funzionali

4.2 Requisiti qualitativi

Codice	Importanza	Stato	Descrizione
RQ-55	Obbligatorio	Soddisfatto	Sviluppo di test che dimostrino il corretto funzionamento dei servizi e delle funzionalità previste. Viene richiesta una copertura dell'80% corredata di report.
RQ-56	Obbligatorio	Soddisfatto	Il progetto deve essere corredato di documentazione riguardo scelte implementative e progettuali effettuate e relative motivazioni.
RQ-57	Obbligatorio	Soddisfatto	Il progetto deve essere corredato di documentazione riguardo problemi aperti e eventuali soluzioni proposte da esplorare.
RQ-58	Obbligatorio	Soddisfatto	Tutte le componenti del sistema devono essere testate con *test end-to-end*$_{G}$.
RQ-59	Obbligatorio	Soddisfatto	Il sistema sarà corredato di un Manuale Utente che spieghi le funzionalità del sistema e come utilizzarle.
RQ-60	Obbligatorio	Soddisfatto	Il sistema sarà corredato di un documento di Specifica Tecnica che spieghi le scelte progettuali effettuate.

Tabella 4: Requisiti qualitativi

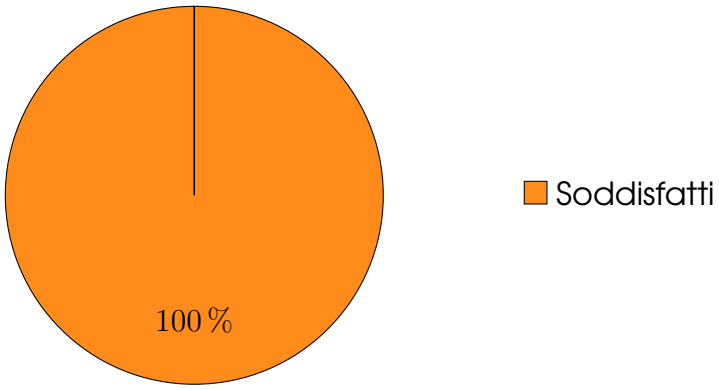

Figura 25: Percentuale di soddisfacimento dei requisiti qualitativi

4.3 Requisiti di vincolo

Codice	Importanza	Stato	Descrizione
RV-61	Obbligatorio	Soddisfatto	Il simulatore di dati deve pubblicare messaggi in una piattaforma di *data streaming*.
RV-62	Obbligatorio	Soddisfatto	La piattaforma di *data streaming* utilizzata è *Redpanda*$_G$.
RV-63	Obbligatorio	Soddisfatto	I dati pubblicati nella piattaforma di *data streaming* devono essere salvati in un database OLAP.
RV-64	Obbligatorio	Soddisfatto	I dati devono poter essere visualizzati dall'utente finale in delle *dashboard*$_G$, sviluppate con un *tool* apposito, ad esempio *Grafana*$_G$.
RV-65	Opzionale	Soddisfatto	I dati pubblicati nei *topic*$_G$ di *Redpanda*$_G$ sono serializzati in formato Confluent Avro.
RV-66	Obbligatorio	Soddisfatto	Il sistema deve essere sviluppato con *Docker*$_G$ *Compose*$_G$, utilizzando la versione 3.8 della specifica.

Codice	Importanza	Stato	Descrizione
RV-67	Obbligatorio	Soddisfatto	Il sistema deve poter essere usufruito dalle versioni più recenti dei browser web più diffusi. Al momento della stesura del presente documento, le versioni supportate sono: *Google Chrome* v124, *Safari* v17.4, *Microsoft Edge* v123, *Firefox* v125.
RV-68	Obbligatorio	Soddisfatto	Il sistema deve poter funzionare su sistema operativo *Linux*, con CPU a 64 bit, almeno 4GB di RAM e una delle seguenti distribuzioni e versioni minime: *Ubuntu* 22.04, *Debian* 12, *Fedora* 38, *Red Hat Enterprise Linux* 8.
RV-69	Obbligatorio	Soddisfatto	Il sistema deve poter funzionare su sistema operativo *Windows* con versione 10 o 11, CPU a 64 bit, almeno 4GB di RAM e la funzionalità WSL2 abilitata.
RV-70	Obbligatorio	Soddisfatto	Il sistema deve poter funzionare su sistema operativo *MacOs* con versione 12 o superiore, CPU *Intel* o *Apple Silicon* a 64bit e almeno 4GB di RAM.

Tabella 5: Requisiti di vincolo

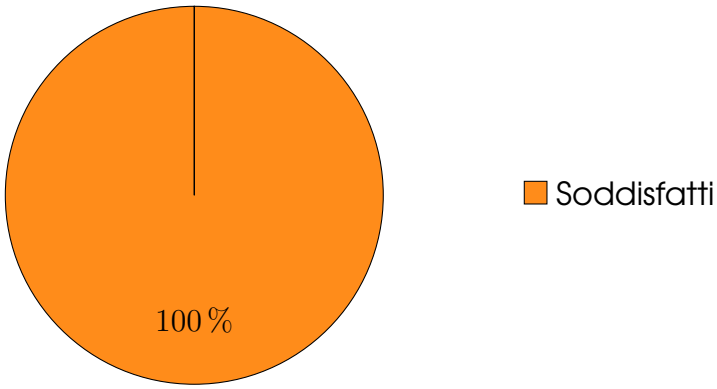

Figura 26: Percentuale di soddisfacimento dei requisiti di vincolo

4.4 Requisiti prestazionali

Codice	Importanza	Stato	Descrizione
RP-71	Obbligatorio	Soddisfatto	Il sistema deve garantire che la visualizzazione dei dati in tempo reale avvenga entro 5 secondi dalla ricezione dei dati.

Tabella 6: Requisiti prestazionali

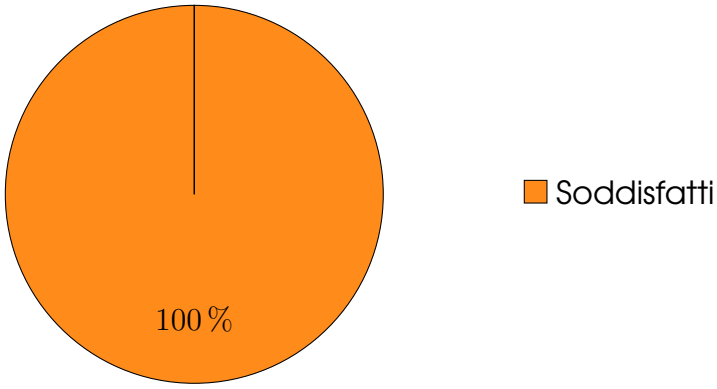

Figura 27: Percentuale di soddisfacimento dei requisiti prestazionali

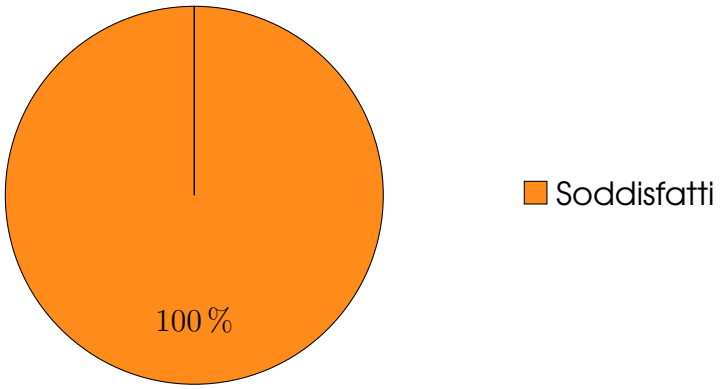

Figura 28: Percentuale di soddisfacimento dei requisiti obbligatori

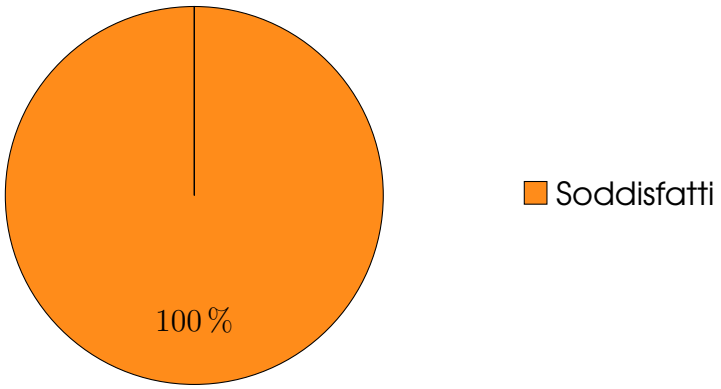

Figura 29: Percentuale di soddisfacimento dei requisiti desiderabili

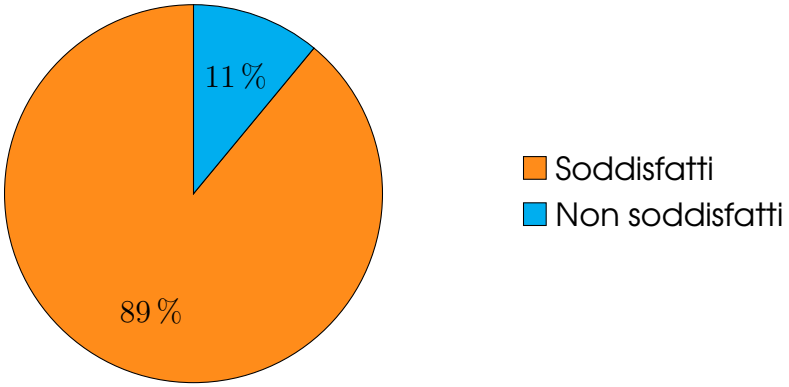

Figura 30: Percentuale di soddisfacimento dei requisiti opzionali

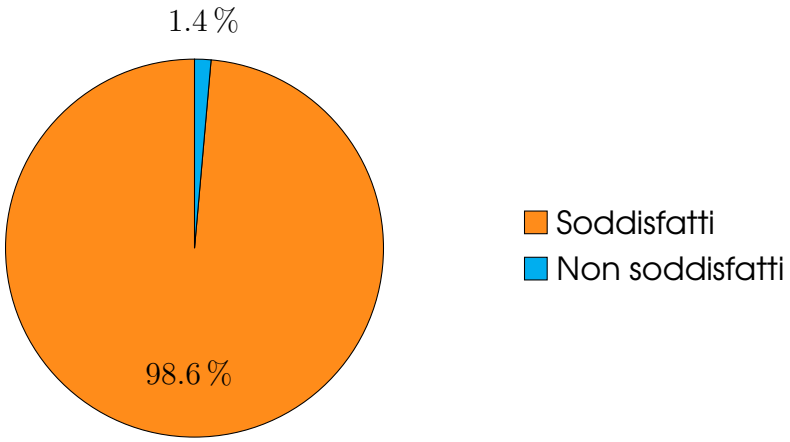

Figura 31: Percentuale di soddisfacimento dei requisiti totale