RdS
RICERCA DI SISTEMA

ENEA
AGENZIA NAZIONALE PER LE NUOVE TECNOLOGIE, L'ENERGIA E LO SVILUPPO ECONOMICO SOSTENIBILE

MINISTERO DELL'AMBIENTE E DELLA SICUREZZA ENERGETICA

Ricerca di Sistema elettrico

ANALISI ED ESTRAZIONE DI INFORMAZIONI DA DATI ETEROGENEI DI PROPRIETA' ENEA (LA2.17)

Prof. Marco La Cascia

Università degli Studi di Palermo

RdS_PTR22-24_PR1.7_LA2.17_371

LA 2.17 ANALISI ED ESTRAZIONE DI INFORMAZIONI DA DATI ETEROGENEI DI PROPRIETA' ENEA

Prof. Marco La Cascia (Università degli Studi di Palermo)

Dicembre 2024

Report Ricerca di Sistema Elettrico

Accordo di Programma Ministero dell'Ambiente e della Sicurezza Energetica - ENEA Piano Triennale di Realizzazione 2022-2024

Obiettivo: Decarbonizzazione

Progetto: Tema di ricerca 1.7 – Tecnologie per la penetrazione efficiente del vettore elettrico negli usi finali

Linea di attività: *2.17*

Responsabile del Progetto: Claudia Meloni, ENEA

Responsabile del Work Package: Claudia Meloni, ENEA

Responsabile Linea di Attività: Marco La Cascia, Università degli Studi di Palermo

Mese inizio previsto: 19

Mese inizio effettivo: 19

Mese fine previsto: 36

Mese fine effettivo: 36

Il presente documento descrive le attività di ricerca svolte all'interno dell'Accordo di collaborazione n.71 dal titolo "Analisi ed Estrazione di Informazioni da Dati Eterogenei di Proprietà ENEA"

Si ringrazia per la collaborazione alle attività svolte l'Ing. Giuseppe Tomasino (ENEA)

Indice

1 Risultati attesi 4
2 Risultati ottenuti 5
2.1 Ulteriori analisi 5
3 Prodotti attesi 6
3.1 Ulteriori analisi 6
4 Prodotti sviluppati 7
4.1 Ulteriori analisi 7
5 Analisi degli scostamenti su attività e risultati 8
6 Sintesi delle attività svolte 9
7 Dettaglio delle attività svolte 10
7.1 Introduzione 10
7.2 Architettura e strumenti utilizzati 10
7.2.1 MongoDB 13
7.2.2 Spark 14
7.3 Dati ENEA 16
7.3.1 Urban Dataset CounterReading 16
7.4 Analisi struttura dati 16
7.5 Connessione MongoDB - Spark 18
7.5.1 Preparazione dei dati per l'analisi in Spark 19
7.5.2 Spark SQL 20
7.6 Calcolo dei KPI 22
7.7 Esplorazione del dataset 24
8 Pubblicazioni scientifiche 28
9 Eventi di disseminazione 29

Indice delle figure

Figura 1 - Architettura generale del DL 11
Figura 2 - Architettura hardware del sistema 12
Figura 3 - Script di caricamento dati remoti e generazione log di sistema 13
Figura 4 - Architettura hardware di Spark 14
Figura 5 - Architettura software di Spark 15
Figura 6 - Schema del DataFrame dedotto per la Collection datiEnergetici 20
Figura 7 – Schema del DataFrame dopo la trasformazione 22

Indice delle Tabelle

Tabella 1 – Proprietà di interesse per CounterReading 18

1 Risultati attesi

La linea di attività 2.17 del Progetto 1.7 di questo triennio di Ricerca, si propone di estrarre informazioni da dati eterogenei di proprietà ENEA e di tipologia prevalentemente energetica, ma provenienti da fonti di varia natura, applicando le moderne tecniche di Analisi dei Dati.

Nell'esecuzione di tale attività ci si attende di ottenere i seguenti risultati:

- Definizione dell'architettura di un Data Lake (nel seguito **DL**) atto a contenere i dati da analizzare.
- Individuazione di uno o più formati che siano idonei al caricamento dei dati all'interno del DL.
- Definizione delle procedure e delle metodologie da seguire per il caricamento dei dati all'interno del DL.
- Implementazione di procedure per l'analisi dei dati e per l'estrazione di informazioni da essi.
- Definizione di una metodologia per la trasformazione dei dati in formati più facilmente gestibili dalle piattaforme di analisi
- Calcolo di KPI e sviluppo di algoritmi di analisi dei dati.

2 Risultati ottenuti

I risultati ottenuti nel triennio corrente sono elencati di seguito, e saranno descritti con maggiore dettaglio al capitolo 7

- Architettura e implementazione del DL
- Definizione del formato dati e delle metodologie per il caricamento dei dati all'interno del DL.
- Definizione delle sorgenti e dei sottoinsiemi di caratteristiche da utilizzare per le analisi successive.
- Definizione delle tecniche di elaborazione MongoDB – Spark
- Individuazione delle metodologie di analisi dati in SparkSQL.
- Implementazione procedure di calcolo dei KPI e semplici analisi dei dati.

In particolare, una volta analizzato il problema e individuate MongoDB e Spark come piattaforme software maggiormente indicate per l'acquisizione, l'archiviazione e l'analisi dei dati si è affrontato il problema di far comunicare la componente di storage con la componente di elaborazione in maniera efficiente e compatibile con la gestione di grandi volumi di dati. La soluzione è stata trovata nell'impiego di MongoDB Spark Connector, un modulo basato sulle più recenti API di Spark in grado di leggere dati da MongoDB in un DataFrame Spark senza passare dal programma driver. L'impiego di tale modulo e la successiva operazione di trasformazione dei dati da una forma fortemente annidata a una forma tabellare in cui ciascun record rappresenta i dati contestuali e i dati comportamentali di una lettura ha permesso di utilizzare le API di SparkSQL per le successive analisi sfruttando appieno le risorse disponibili.

In generale la metodologia sviluppata, che prevede di estrarre i dati dalla componente MongoDB del data lake e portarli in una tabella virtuale distribuita di SparkSQL, permette di effettuare analisi interattive in tempi ragionevolmente brevi utilizzando uno strumento semplice come il linguaggio SQL. Tramite l'utilizzo di SparkSQL come API di analisi dei dati è stato quindi implementato il calcolo dei KPI richiesti e l'analisi esplorativa dell'intero dataset finalizzata all'individuazione di anomalie nelle letture dei contatori.

Tutta la metodologia di trasformazione e analisi dei dati è stata concepita sin dall'inizio per poter lavorare su piattaforme di calcolo distribuito. Sebbene l'attività sia stata condotta su server stand-alone, va evidenziato che la stessa metodologia e lo stesso codice potrebbero essere portati su un cluster con qualsiasi numero di nodi **senza alcuna modifica**. Configurando Spark in modalità cluster, in maniera del tutto trasparente, i dati letti da MongoDB verrebbero distribuiti sui vari nodi e tutte le operazioni specificate tramite le API di SparkSQL verrebbero eseguite in parallelo garantendo scalabilità orizzontale potenzialmente infinita.

2.1 Ulteriori analisi

La metodologia sviluppata per l'acquisizione dei dati nel DL e la successiva trasformazione dei dati per l'analisi in Spark ha permesso una più semplice esplorazione del dataset e aperto la strada per l'implementazione di ulteriori analisi utilizzando algoritmi e librerie off-the-shelf.

3 Prodotti attesi

I prodotti che ci si attende di ottenere in questo triennio sono i seguenti:

- Un DL di proprietà ENEA contenente i dati da analizzare.
- La definizione di una o più procedure per il caricamento dei dati all'interno del DL
- Il codice di trasformazione dei dati dal formato MongoDB a quello Spark SQL
- Il codice Spark SQL per il calcolo dei KPI
- I moduli software per l'analisi dei dati

3.1 Ulteriori analisi

Se le attività lo renderanno possibile, sarà prodotto il codice per l'individuazione di anomalie, il calcolo della similarità fra serie temporali e il clustering.

4 Prodotti sviluppati

I prodotti che sono stati sviluppati sono i seguenti:

- Un Data Lake ENEA per l'immagazzinamento dei dati. Il DL è accessibile solo da rete ENEA, e per utilizzarlo occorre prima effettuare l'accesso alla rete ENEA tramite credenziali VPN.
- Le specifiche dei formati dei dati che consentono il loro caricamento all'interno del DL.I formati dei file accettati sono il csv ed il json
- Una procedura per il caricamento dei dati nel DL. Il caricamento avviene attraverso l'utilizzo di uno script python, che gira sul server ENEA di accesso al DL, ed il cui codice è descritto al paragrafo 7
- Un sito web che consente l'accesso amministrativo al DL dalla rete intranet ENEA. L'indirizzo di accesso è il seguente: https://apic.palermo.enea.it
- Il codice per la trasformazione dei dati dal formato MongoDB a Spark
- Il codice per il calcolo dei KPI
- Il codice per l'esplorazione del dataset CounterReading

Tutto il codice prodotto è scaricabile dal seguente URL:

https://apic.palermo.enea.it/downloads

4.1 Ulteriori analisi

È stata condotta un'analisi esplorativa per l'individuazione di possibili anomalie nelle letture dei contatori.

5 Analisi degli scostamenti su attività e risultati

Nel DL sono confluiti e sono stati analizzati i dati ottenuti dalla piattaforma ENEA denominata PELL[1]. Si tratta di dati di natura energetica, ottenuti da sensori "smart meter" collocati sugli impianti di illuminazione pubblica cittadina per la misurazione dei consumi energetici. Al momento non è stato possibile recuperare altre tipologie di dati quali immagini da droni, testi descrittivi o dati legati a fenomeni di natura meteorologica sebbene l'architettura del Data Lake progettato ne consentirebbe una semplice integrazione. Questa limitazione non ha comunque influito negativamente sulle attività, al fine di ottenere i risultati previsti.

[1] https://sue.enea.it/product/citta-sicure-e-sostenibili/

6 Sintesi delle attività svolte

Per l'implementazione del DL sono state eseguite le seguenti attività:

- Analisi e definizione dell'architettura da realizzare
- Individuazione ed installazione dei necessari apparati per la realizzazione dell'infrastruttura hardware
- Analisi e individuazione degli strumenti idonei all'acquisizione e alla elaborazione dei dati
- Individuazione ed utilizzo degli strumenti più efficaci per effettuare le operazioni di analisi dei dati

Per l'infrastruttura hardware, è stata utilizzata la configurazione descritta al paragrafo seguente, che prevede l'utilizzo di due Server e di uno storage. Per esigenze di efficienza e scalabilità è stata utilizzata la piattaforma **Hadoop** di Apache.

Definita l'architettura del DL, sono stati individuati gli strumenti per la gestione ed il trattamento dei dati, e quindi quelli per la vera e propria analisi ed estrazione delle informazioni richieste. Ai fini della memorizzazione dei dati, è stato utilizzato il DBMS **MongoDB**, mentre per l'analisi dei dati lo strumento di riferimento è la piattaforma **Spark** di Apache.

E' stata quindi definita una metodologia per la lettura dei dati presenti in MongoDB in Spark e la relativa trasformazione in tabelle temporanee accessibili tramite le API di alto livello SparkSQL.

E' stato infine implementato il calcolo dei KPI utilizzando SparkSQL ed è stata condotta un'analisi esplorativa sul dataset CounterReading al fine di determianre eventuali anomalie.

Tutti gli strumenti utilizzati saranno descritti in dettaglio nei successivi paragrafi.

7 Dettaglio delle attività svolte

7.1 Introduzione

Un DL è un archivio di dati eterogenei, acquisiti e memorizzati nel loro formato originario. Esso è destinato a permanere e a crescere nel tempo e proprio per questa sua caratteristica, deve essere organizzato in modo flessibile e scalabile. Per definire con precisione l'Architettura del DL, occorre individuare i processi che devono essere eseguiti sui dati, e ove possibile, anche le applicazioni che utilizzeranno tali dati.

I processi individuati sono i seguenti:

1. Acquisizione dei dati da sorgenti remote (***data ingestion***): i dati da archiviare sono eterogenei, e provengono da fonti remote di varia natura. I problemi da affrontare in questa fase riguardano l'archiviazione dei dati in maniera sicura ed efficiente, ma occorre anche verificare che il dato immagazzinato sia integro e non vi siano errori durante la trasmissione.
2. Scrematura e filtraggio dei dati grezzi (***data processing***): in genere è utile effettuare una prima scrematura dei dati ancor prima dell'analisi vera e propria, per eliminare fin da subito dati falsati e non coerenti con i processi che si vogliono analizzare.
3. Elaborazione e analisi dei dati ottenuti dalla fase precedente (***data analysis):*** questa fase permette di eseguire delle elaborazioni (anche) di natura statistica sui dati risultanti dalla fase precedente, in modo da estrarre informazioni e KPI utili ai nostri scopi.
4. Accesso ai dati da parte di applicazioni esterne (***data integration)***: ottenute le informazioni utili, e i KPI richiesti, occorre rendere disponibili tali risultati ad applicazioni esterne, e quindi definire dei protocolli di accesso, ed eventualmente realizzare delle interfacce che consentano l'accesso e l'interrogazione delle informazioni in maniera automatica.

7.2 Architettura e strumenti utilizzati

L'architettura da realizzare deve essere il più possibile flessibile e orientata alla specifica applicazione. Il DL deve essere progettato su misura per le esigenze dell'applicazione che si sta realizzando, tenendo conto degli obiettivi perseguiti e delle proprietà dei dati che devono essere trattati. Questo approccio richiede sicuramente un grande sforzo iniziale, ma garantisce flessibilità e controllo sul flusso successivo delle attività. Tenendo conto di tutte queste premesse, l'Architettura generale del DL viene riportata in Fig.1.

Il Sistema DL risulta composto da alcuni blocchi che intervengono durante le varie fasi del processo di trattamento dei dati, e interagiscono tra loro in funzione del relativo compito che esercitano. Il blocco di Acquisizione Dati si occuperà di immagazzinare gli stessi ed inviarli allo storage, mentre il blocco di Filtraggio Dati, effettuerà un pretrattamento degli stessi, per far sì che l'analisi non produca dei valori falsati a causa dell'elaborazione di dati non congruenti al tipo di informazione che si desidera estrapolare.

I dati remoti saranno acquisiti attraverso la rete utilizzando i protocolli supportati, ed in linea di principio, un generico utente potrebbe fare richiesta riguardo ad una specifica elaborazione o ad una classe particolare di KPI (*questa funzionalità al momento non è supportata*)

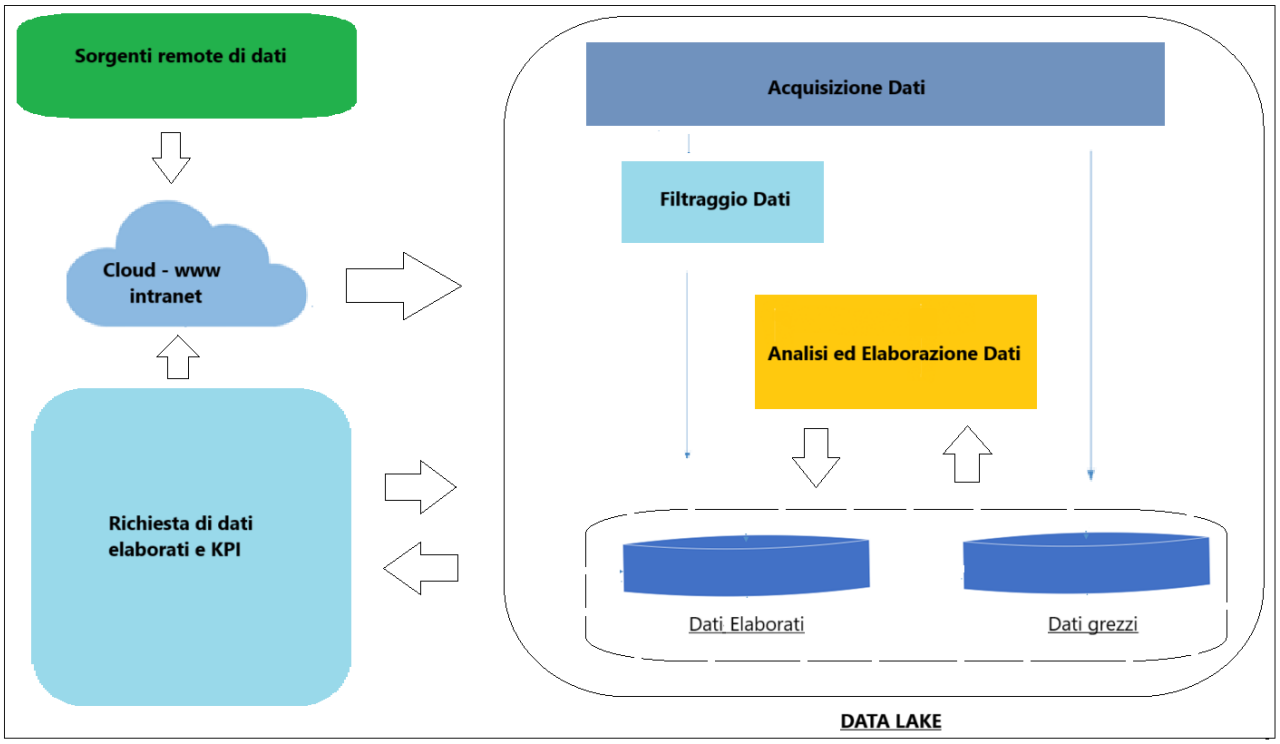

Figura 1 - Architettura generale del DL

La piattaforma Hardware realizzata si compone di due server connessi tra loro, e di uno storage dedicato che ha la funzione di garantire la persistenza dei dati acquisiti. I dischi dello Storage, sono configurati in **RAID 5**. Su ambedue i server utilizzati è stato installato il Sistema Operativo Linux, distribuzione CentOS Stream 9.

Siccome il DL deve immagazzinare i dati così come vengono ricevuti, la scelta ottimale per il DBMS di riferimento è NoSQL, non relazionale. Tra i DBMS di questo tipo attualmente MongoDB[2] è sembrato essere quello più adatto allo scopo, sia per affidabilità, che per facilità d'uso.

Inoltre, considerato che l'implementazione della struttura dati viene spesso effettuata su piattaforme distribuite, è stato installato all'interno del sistema anche Hadoop[3] di Apache, che consente appunto l'elaborazione distribuita di grandi insiemi di dati su cluster di computer, utilizzando semplici modelli di programmazione. Sicuramente nel caso particolare non si sfrutteranno appieno tutte le funzionalità di Hadoop, però questa scelta consente di ottenere un sistema con un elevato grado di robustezza ai guasti ed elevata scalabilità.

La Fig.2 mostra l'Architettura Hardware del sistema.

[2] https://www.mongodb.com/it-it
[3] https://hadoop.apache.org/

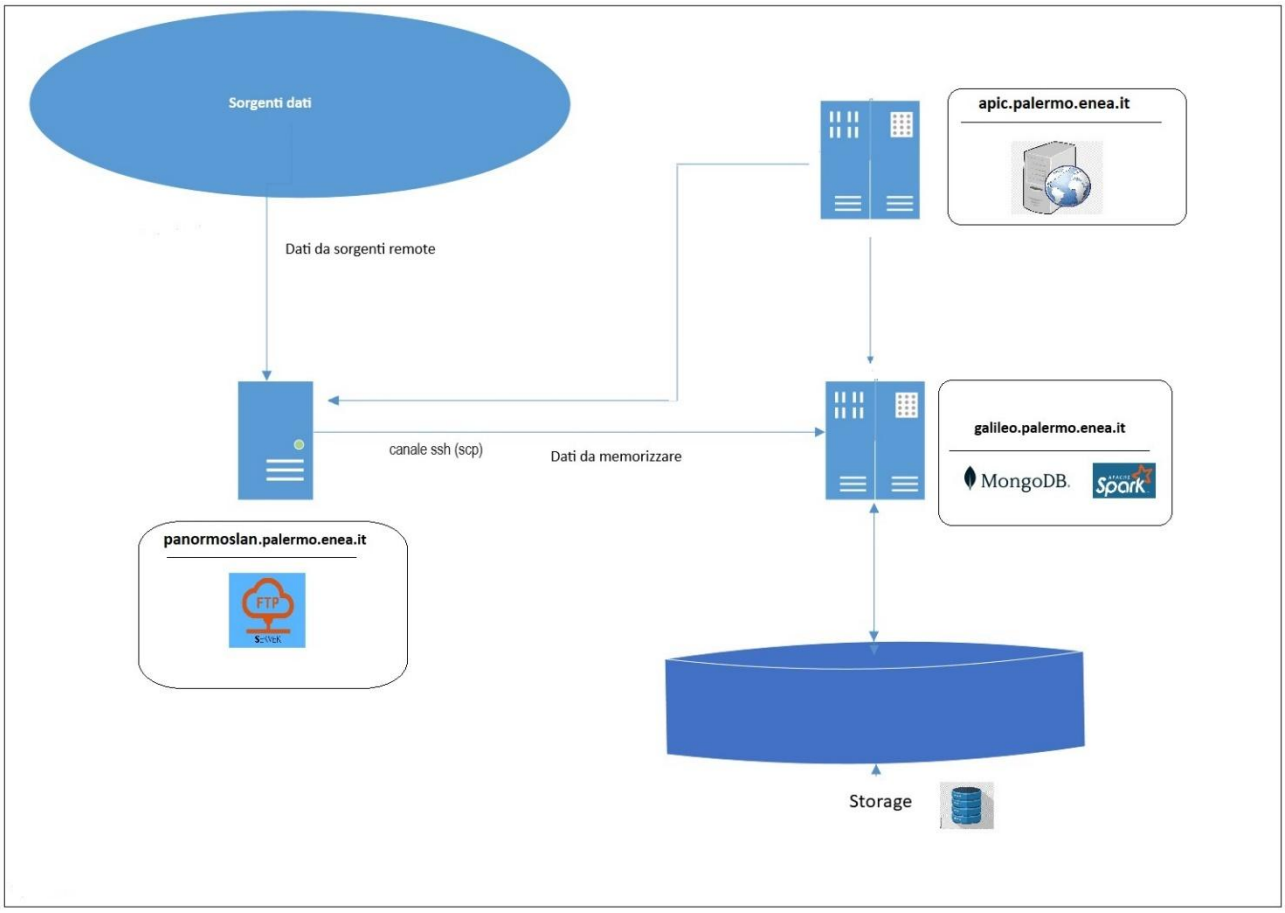

Figura 2 - Architettura hardware del sistema

I due server sono connessi in rete sulla stessa LAN, ed è inoltre attivo tra loro un canale SSH sicuro per la comunicazione. I protocolli attualmente utilizzati per inviare i dati al DL, sono i seguenti:

- FTP con cifratura TLS (implementato con il demone *vsftpd* sul server)
- SSH. In questo caso il dispositivo remoto dovrà essere accoppiato con il server *panormos* tramite scambio di chiavi pubblica e privata, e inviare una richiesta di trasmissione al server utilizzando il comando *scp*

I formati attualmente supportati sono ***csv*** e ***json***. All'indirizzo **panormoslan.*palermo.enea.it*** è attivo un FTP server dotato di cifratura TLS, per gestire le richieste di caricamento dati. Il file acquisito viene elaborato da uno script Python, che si occupa anche delle seguenti attività:

- Memorizzazione dei dati nello storage nel formato corretto: i dati estratti dal file caricato sul server vengono inviati al DBMS utilizzando le librerie ***pymongo*** e ***json*** (nel caso di file json) o **csv** (nel caso di file csv) di Python
- Archiviazione dei file acquisiti (dati originari): in assenza di errori, il file remoto sarà archiviato nello storage, e a tal fine sarà trasmesso al server ***galileo*** utilizzando il canale sicuro SSH (comando *scp*)

Un ulteriore script PHP si occupa invece di archiviare i log di sistema, per mantenere lo storico delle operazioni effettuate. A tal fine, ogni operazione con l'indicazione del relativo esito finale, sarà registrata su un DBMS relazionale (tabella MySQL su *panormos*), sia in caso di esito positivo che negativo. In caso di errore però, il file remoto non sarà archiviato nello storage, ma verrà

memorizzato in locale sul server *panormos*. In caso invece di successo, i dati estratti dal file remoto saranno memorizzati nella Collection datiEnergetici di MongoDB.

Siccome il server di accesso al DL si trova all'interno della rete ENEA, nel caso in cui le sorgenti dei dati si trovassero all'esterno, per utilizzare questi protocolli, i relativi servizi dovranno essere abilitati sul firewall di rete ENEA, ovvero, in alternativa, la trasmissione da parte dei sensori dovrà essere preceduta da una connessione alla rete VPN aziendale ENEA.

Attualmente non è prevista la trasmissione simultanea di dati al DL, che necessiterebbe di sistemi avanzati per la gestione delle code di ingresso al sistema, e quindi non abbiamo implementato questa funzionalità.

L'accesso ai dati su MongoDB è disponibile utilizzando l'applicazione dedicata MongoDB Compass[4], alla seguente URI: ***galileo.palermo.enea.it:27017***

Infine, all'indirizzo ***https://apic.palermo.enea.it*** è presente una pagina web di accesso al DL, che permette di visualizzare i log di sistema ed alcune statistiche di funzionamento del DBMS. A tal fine abbiamo installato un server web Apache su un terzo server denominato ***apic***.

La figura seguente mostra un estratto degli script utilizzati:

```python
import sys
import os

def is_hidden(file): return file.startswith('.')
def is_csv(file): return file.endswith('.csv')
def is_CSV(file): return file.endswith('.CSV')
def is_json(file): return file.endswith('.json')
def is_JSON(file): return file.endswith('.JSON')

ftpPath="/home/userftp/"
proPath="/home/userftp/processed"
bugPath="/home/userftp/bugs"

uploads=[]
filedir= os.listdir(ftpPath)
for file in filedir:
    if not is_hidden(file):
        uploads.append(file)
count= len(uploads)
if count!= 0:
    for f in uploads:
        if (is_json(f) or is_JSON(f)):
            loadCmd= "python loadJson.py " + ftpPath + f
            os.system(loadCmd)
            cpfile= "scp " + ftpPath + f +
                    " gtomasino@galileo.palermo.enea.it:~/fileRemoti/" + f
            os.system(cpfile)
            mvcmd= "mv " + ftpPath + f + " ./processed"
            os.system(cmd)
            logcmd= "php putlog.php " + proPath + f
            os.system(cmd)
        else :
            if (is_csv(f) or is_CSV(f)):
                loadCmd= "python loadCSV.py " + ftpPath + f
                os.system(loadCmd)
                cpfile= "scp " + ftpPath + f +
                        " gtomasino@galileo.palermo.enea.it:~/fileRemoti/" + f
                os.system(cpfile)
                mvcmd= "mv " + ftpPath + f + " ./processed"
                os.system(cmd)
                logcmd= "php putlog.php " + proPath + f
                os.system(cmd)
            else :
                mvcmd= "mv " + ftpPath + f + " ./bugs"
                os.system(cmd)
"list.py" 49L, 1656B written
```

```php
<?php
        require_once 'config.php';

        if ($argc < 2) echo "nessun file specificato";
        else {
                $filename= $argv[1];
                echo $filename;

                // Connessione a MySQL
                $link= mysqli_connect(DBLOC,DBOWN,DBPWD,DBNAME);
                // Check connection
                if (!$link) {
                        die("Connection failed: " . mysqli_connect_error());
                }
                echo "Connected successfully\n";

                $query= "select * from logTB";
                $result= mysqli_query($link, $query);
                $nrow= mysqli_num_rows($result);

                $names= explode("/",$filename);
                $t= date("Y-m-d_H:i:s");
                $codOp= $t . "-" . ($nrow+1);
                $res= $names[3];
                $insquery= "insert into logTB(codOP, datetime,filename,result)
                                values ('$codOp','$t','$names[4]','$res')";
                $result= mysqli_query($link,$insquery);
                mysqli_close($link);
        }
?>
```

Figura 3 - Script di caricamento dati remoti e generazione log di sistema

7.2.1 MongoDB

MongoDB è un DBMS ("Data Base Management System", sistema di gestione della base di dati) categorizzato come NoSQL, un termine usato per indicare i DBMS che propongono soluzioni alternative all'utilizzo del paradigma relazionale. MongoDB offre dei "connettori" che permettono di mettere in comunicazione applicazioni di elaborazione distribuita dei dati con

[4] https://www.mongodb.com/products/tools/compass

l'ambiente MongoDB, traducendo le funzioni scritte in codice sorgente in query MongoDB, comandi e pipeline di aggregazione.

Mentre i database relazionali sono strutturati in tabelle che contengono record omogenei e sono legate tra loro da vincoli di integrità referenziale, in MongoDB i dati sono conservati sotto forma di documenti, organizzati in collezioni indipendenti tra loro. L'uniformità nella struttura dei documenti all'interno di una collezione è facoltativa, ma consigliata per agevolare le operazioni di lettura e scrittura dei documenti. Questo paradigma viene chiamato "schema-on-read". Richiede più tempo per l'interrogazione rispetto al paradigma "schema-on-write", in quanto la struttura di ciascun documento non è nota a priori, bensì va esaminata ed applicata per strutturare i valori del documento stesso ma è più agile, perché prevede l'interrogazione di basi di dati non strutturate.

7.2.2 Spark

Apache Spark è un framework open-source per il calcolo distribuito di grandi moli di dati, pensato per essere più rapido ed efficiente di MapReduce. È un motore di elaborazione che offre delle API di programmazione di facile utilizzo, con capacità di calcolo parallelo e distribuito. Il punto di forza principale di Spark è la sua capacità di effettuare elaborazioni in memoria per poi salvare i risultati intermedi in una cache distribuita. Quest'ultima è un meccanismo che permette di condividere informazioni tra i nodi della rete per elaborare in parallelo i dati.

L'infrastruttura di calcolo distribuito di Apache Spark è costituita da computer, detti "nodi", organizzati in gruppi, detti "cluster". Si basa su un modello master-slave, in cui un singolo nodo distribuisce i compiti agli altri nodi e coordina l'esecuzione del programma, mentre tutti gli altri nodi elaborano i dati in parallelo tra loro e fanno riferimento al nodo coordinatore.

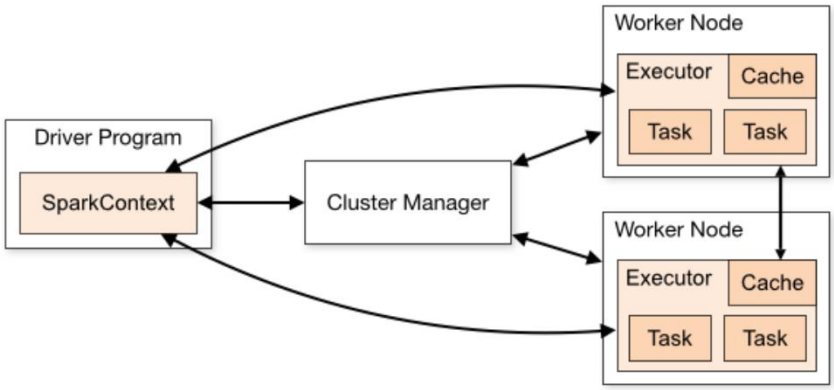

Figura 4 - Architettura hardware di Spark – Fonte Big Data Infrastructure Technologies for Data Analytics. Springer.

Apache Spark offre delle API che possono essere utilizzate in combinazione tra loro per effettuare diversi tipi di operazioni in maniera sequenziale o in parallelo per gestire gli RDD (Resilient Distributed Dataset), un insieme di elementi distribuiti su molti nodi del cluster che possono essere manipolati in parallelo.

Con applicazione Spark si intende qualsiasi applicazione che richiede l'elaborazione distribuita di dati in un cluster Spark. Il computer che lancia una tale applicazione è considerato parte integrante del cluster operativo, e viene definito nodo **driver**, o semplicemente driver. Questo invia al nodo master il compito di elaborare i dati, e a sua volta il master suddivide questo compito in parti più piccole, detti executor. I programmi Spark possono anche essere eseguiti localmente, ma così facendo si perdono i vantaggi offerti dall'elaborazione distribuita dei dati.

Spark SQL è l'interfaccia di Spark che consente di effettuare query su dati strutturati e semi-strutturati con estrema semplicità per chiunque conosca la sintassi del linguaggio SQL. Spark SQL introduce le astrazioni di DataFrame. Un DataFrame è un'astrazione di livello superiore rispetto agli RDD, ha una struttura simile a quella di una tabella in una base di dati tradizionale, con una serie di colonne, ciascuna con un nome e un tipo di dato associato, e righe rappresentanti ognuna un elemento diverso dell'RDD su cui è basato il DataFrame.

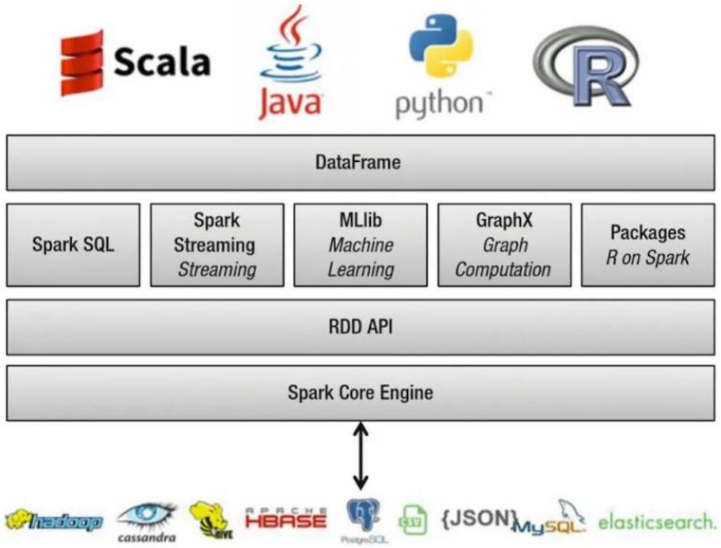

Figura 5 - Architettura software di Spark – Fonte https://medium.com/@thejasbabu/spark-dataframes-10c349de04c

Se si vuole eseguire una query scrivendola in linguaggio SQL, è necessario registrare i DataFrame come viste utilizzando il metodo "*createOrReplaceTempView()*", in questo modo potremo dare un nome al DataFrame da poter utilizzare nelle query SQL successive. Questo è l'approccio che è stato adottato nella presente attività. L'esempio che segue dimostra questa funzionalità di Spark SQL:

```
from pyspark.sql import SparkSession

spark = SparkSession \
    .builder \
    .appName("Python Spark SQL basic example") \
    .config("spark.some.config.option", "some-value") \
    .getOrCreate()

data = [(1, "Alice", 1000), (2, "Bob", 1500), (3, "Charlie", 2000)]
columns = ["id", "name", "salary"]

df = spark.createDataFrame(data, columns)
view = df.createOrReplaceTempView("employees")

# Query sql lanciata con il metodo sql()
result_df = df.sql('''SELECT TOP 2 name, salary
    FROM employees
    WHERE salary > 1200
    ORDER BY salary DESC;''')
```

Nello sviluppo di questa linea di attività, poiché è stato utilizzato Python come linguaggio di programmazione e Apache Spark è scritto in linguaggio Scala, è stato utilizzato il pacchetto PySpark come adattatore. PySpark si appoggia a Py4j, una libreria molto diffusa che permette a Python di interfacciarsi dinamicamente con oggetti JVM, come ad esempio gli RDD, le strutture dati di Apache Spark.

7.3 Dati ENEA

Come già accennato in precedenza, i dati analizzati sono quelli ottenuti dalle rilevazioni del Progetto ENEA PELL, inviati dagli smart meter installati sugli impianti di illuminazione pubblica cittadina.

7.3.1 Urban Dataset CounterReading

La specifica SCPS Semantic 2.0 descrive la semantica comune a cui aderire, al fine di attuare uno scambio di dati non ambiguo, tra una Smart City Platform e i sistemi verticali, e quindi ottenere una comunicazione interoperabile tra i sistemi della Smart City. Per semantica si intende la definizione formale del significato degli elementi che costituiscono il modello, e il corrispondente formato, utilizzato per rappresentare i dati. Tale formato viene chiamato UrbanDataset. Ogni UrbanDataset è suddiviso in tre sezioni principali: Context, Specification e Values.

La sezione Context offre informazioni utili a contestualizzare i dati contenuti nel dataset, come il fuso orario, l'istante di generazione del dataset e le coordinate di rilevamento. La sezione Specification contiene le informazioni che descrivono l'UrbanDataset, tra cui le definizioni di ogni UrbanDatasetProperty contenuta in esso. Tutte le proprietà hanno un nome e una descrizione. La sezione Values contiene proprietà raggruppate per un determinato criterio, solitamente in base a ora e luogo del rilevamento delle misurazioni.

7.4 Analisi struttura dati

L'UrbanDataset oggetto di studio per questa attività è il CounterReading, che contiene i dati rilevati dai contatori che monitorano il consumo elettrico di un sistema di illuminazione pubblica con impianti trifase. Per essere più precisi, il dataset utilizzato è una piccola porzione di CounterReading, contenente tutti i rilevamenti effettuati tra il 20/05/2022 e il 10/04/2023 nei comuni di San Severino Marche (MC) e Comunanza (AP).

Il dataset è stato caricato su un'istanza locale MongoDB. La struttura dei documenti della collezione CounterReading sul database MongoDB è descritta nel seguente schema:

- _id: identificatore univoco del documento all'interno della collezione; tipo ObjectId
- UrbanDataset: oggetto contenente tutte le proprietà definite nella semantica del dataset; tipo Object
  - context: contiene informazioni relative al contesto in cui è stato generato il documento; tipo Object
    - producer: contiene informazioni relative al sistema (Piattaforma/Soluzione) che ha prodotto i dati rilevati; tipo Object
      - id: identificatore univoco del sistema; tipo String
      - schemeID: identificativo dello schema di identificazione secondo il quale è stato definito l'identificatore; tipo String
    - timeZone: fuso orario di qualsiasi marca temporale presente nel documento; tipo String
    - timestamp: istante di generazione di questo UrbanDataset (ovvero il momento in cui i dati sono stati aggregati); tipo String
    - coordinates: coordinate WGS84 del centro geometrico della rete applicativa su cui sono stati rilevati i valori trasmessi; tipo Object

        - format: formato WGS84 in cui sono espresse le coordinate; tipo String
        - latitude: latitudine; tipo Double
        - longitude: longitudine; tipo Double
        - height: altitudine; tipo Double
    - language: lingua in cui sono espressi i campi testo dell'istanza; tipo String
- specification: contiene la versione delle specifiche e la lista delle proprietà che vengono da esso rappresentate; tipo Object
    - version: versione della specifica; tipo String
    - id: contiene le informazioni riguardo l'identificativo della specifica; tipo Object
        - value: identificativo della specifica; tipo String
        - schemeID: identificativo dello schema di identificazione secondo il quale è stato definito l'identificatore; tipo String
    - name: nome associato all' UrbanDataset; tipo String
    - uri: URI che identifica la specifica (può essere un urn indicante il dominio della specifica, o un url); tipo String
    - properties: contiene le definizioni delle proprietà specifiche per il dataset; tipo Object
        - propertyDefinition: array contenente tutte le proprietà specifiche per il dataset sotto forma di oggetti; tipo Array
            - propertyName: nome della proprietà; tipo String
            - propertyDescription: descrizione della proprietà; tipo String
            - subProperties: oggetto che contiene un array di sotto-proprietà; tipo Object
                - propertyName: array di sotto-proprietà; tipo Array
            - dataType: Tipo di dato con cui viene espresso il valore della proprietà; tipo String
            - codeList: riferimento ad una lista di codici contenente il set di valori ammessi per questa proprietà e il relativo significato; tipo String
            - unitOfMeasure: unità di misura in cui è espresso il valore; tipo String
- values: contiene i valori di ciascuna variabile definita per questo dataset; tipo Object
    - line: array contenente i valori di ciascun rilevamento; tipo Array
        - id: identificativo del singolo rilevamento; tipo Int32
        - coordinates: riferimento geografico; tipo Object
            - format: formato WGS84 in cui sono espresse le coordinate; tipo String
            - latitude: latitudine; tipo Double
            - longitude: longitudine; tipo Double
            - height: altitudine; tipo Double
        - period: periodo durante il quale sono stati rilevati i dati riportati nella riga; tipo Object
            - start_ts: marca temporale indicante l'inizio del periodo; tipo String
            - end_ts: marca temporale indicante la fine del periodo; tipo String

- property: array contenente tutti i valori delle proprietà definite; tipo Array
    - name: nome della proprietà; tipo String
    - val: valore della proprietà; tipo variabile

7.5 Connessione MongoDB - Spark

La sezione Values contiene proprietà raggruppate per un determinato criterio, solitamente in base a ora e luogo del rilevamento delle misurazioni. Scopo dell'attività è l'estrazione da ciascuna lettura di un vettore numerico multidimensionale in modo da poter applicare più agevolmente le tecniche di analisi successive. Le proprietà specifiche di interesse definite per il dataset CounterReading sono:

Proprietà	**Descrizione**	**Tipo - unità di misura**
ElectricPanelID	Identificativo del quadro elettrico	String
PODID	Codice POD che identifica univocamente il punto di prelievo	String
TownCode	Codice ISTAT del comune	String
Phase1Voltage	Tensione della fase R	Double - V
Line1Current	Corrente della linea 1	Double - A
Phase1PowerFactor	Fattore di potenza R	Double
Phase1ActivePower	Potenza attiva fase R	Double - kW
Phase1ApparentPower	Potenza apparente R	Double - kVA
Phase1ReactivePower	Potenza reattiva R	Double - kVAR
Phase2Voltage	Tensione della fase S	Double - V
Line2Current	Corrente della linea 2	Double - A
Phase2PowerFactor	Fattore di potenza S	Double
Phase2ActivePower	Potenza attiva fase S	Double - kW
Phase2ApparentPower	Potenza apparente S	Double - kVA
Phase2ReactivePower	Potenza reattiva S	Double - kVAR
Phase3Voltage	Tensione della fase T	Double - V
Line3Current	Corrente della linea 3	Double - A
Phase3PowerFactor	Fattore di potenza T	Double
Phase3ActivePower	Potenza attiva fase T	Double - kW
Phase3ApparentPower	Potenza apparente T	Double - kVA
Phase3ReactivePower	Potenza reattiva T	Double - kVAR
TotalActiveEnergy	Energia attiva totale calcolata sulle tre fasi	Double - kWh
TotalReactiveEnergy	Energia reattiva totale per le tre fasi	Double - kVARh
TotalActivePower	Potenza attiva totale	Double - kW
TotalApparentPower	TotalApparentPower	Double - kVA
TotalReactivePower	Potenza reattiva totale	Double - kVAR

Tabella 1 – Proprietà di interesse per CounterReading.

7.5.1 Preparazione dei dati per l'analisi in Spark

Tutti gli esperimenti sono stati condotti sul server galileo descritto in precedenza; la versione del pacchetto PySpark utilizzato è la 3.5.1, compatibile con il connettore Mongo-Spark utilizzato nella sua versione 10.4.0 per Scala 2.12. Per connettere l'istanza di MongoDB all'ambiente Spark si può configurare la connessione a MongoDB e i parametri di configurazione all'avvio della Pyspark shell:

```
pyspark \
  --driver-memory 16g \
  --executor-memory 32g \
  --conf "spark.mongodb.read.connection.uri=mongodb://apic.palermo.enea.it:27017/" \
  --conf "spark.mongodb.write.connection.uri=mongodb://apic.palermo.enea.it:27017/" \
  --packages org.mongodb.spark:mongo-spark-connector_2.12:10.4.0
```

Si osservi che per la corretta elaborazione della collection datiEnergetici è stato necessario specificare i parametri *driver-memory* ed *executor-memory* in quanto, con i valori di default, veniva lanciata una OutOfMemory exception dalla JVM.

Dopo aver creato la SparkSession, possiamo lanciare il metodo "*read()*" specificando il formato "mongodb", e infine caricare i dati sotto forma di DataFrame.

```
dataFrame = spark.read \
                  .format("mongodb") \
                  .option("database", "RSE2022") \
                  .option("collection", "datiEnergetici") \
                  .load()
```

Al termine di queste operazioni, viene creato un DataFrame con lo schema dedotto a partire dalla collezione MongoDB. Tramite questo DataFrame possiamo utilizzare Spark SQL per fare delle query, come se si trattasse di un database relazionale locale.

Lo schema del DataFrame dedotto per la collection datiEnergetici è riportato in Fig. 6.

```
root
 |-- UrbanDataset: struct (nullable = true)
 |    |-- context: struct (nullable = true)
 |    |    |-- producer: struct (nullable = true)
 |    |    |    |-- id: string (nullable = true)
 |    |    |    |-- schemeID: string (nullable = true)
 |    |    |-- timeZone: string (nullable = true)
 |    |    |-- timestamp: string (nullable = true)
 |    |    |-- coordinates: struct (nullable = true)
 |    |    |    |-- format: string (nullable = true)
 |    |    |    |-- latitude: double (nullable = true)
 |    |    |    |-- longitude: double (nullable = true)
 |    |    |    |-- height: double (nullable = true)
 |    |    |-- language: string (nullable = true)
 |    |-- specification: struct (nullable = true)
 |    |    |-- version: string (nullable = true)
 |    |    |-- id: struct (nullable = true)
 |    |    |    |-- value: string (nullable = true)
 |    |    |    |-- schemeID: string (nullable = true)
 |    |    |-- name: string (nullable = true)
 |    |    |-- uri: string (nullable = true)
 |    |    |-- properties: struct (nullable = true)
 |    |    |    |-- propertyDefinition: array (nullable = true)
 |    |    |    |    |-- element: struct (containsNull = true)
 |    |    |    |    |    |-- codeList: string (nullable = true)
 |    |    |    |    |    |-- dataType: string (nullable = true)
 |    |    |    |    |    |-- propertyDescription: string (nullable = true)
 |    |    |    |    |    |-- propertyName: string (nullable = true)
 |    |    |    |    |    |-- subProperties: struct (nullable = true)
 |    |    |    |    |    |    |-- propertyName: array (nullable = true)
 |    |    |    |    |    |    |    |-- element: string (containsNull = true)
 |    |    |    |    |    |-- unitOfMeasure: string (nullable = true)
 |    |-- values: struct (nullable = true)
 |    |    |-- line: array (nullable = true)
 |    |    |    |-- element: struct (containsNull = true)
 |    |    |    |    |-- id: integer (nullable = true)
 |    |    |    |    |-- coordinates: struct (nullable = true)
 |    |    |    |    |    |-- format: string (nullable = true)
 |    |    |    |    |    |-- latitude: double (nullable = true)
 |    |    |    |    |    |-- longitude: double (nullable = true)
 |    |    |    |    |    |-- height: double (nullable = true)
 |    |    |    |    |-- period: struct (nullable = true)
 |    |    |    |    |    |-- start_ts: string (nullable = true)
 |    |    |    |    |    |-- end_ts: string (nullable = true)
 |    |    |    |    |-- property: array (nullable = true)
 |    |    |    |    |    |-- element: struct (containsNull = true)
 |    |    |    |    |    |    |-- name: string (nullable = true)
 |    |    |    |    |    |    |-- val: string (nullable = true)
 |-- _id: string (nullable = true)
```

Figura 6 - Schema del DataFrame dedotto per la Collection datiEnergetici

7.5.2 Spark SQL

Una volta creato il DataFrame, è possibile interrogarlo lanciando delle query SQL utilizzando il metodo "*SparkSession.sql(query)*", dove query è la stringa che consiste nella query in formato SQL, e il valore di ritorno di questo metodo è un nuovo DataFrame contenente i valori raccolti attraverso la query. È difficile però interrogare il DataFrame df nella sua forma attuale, poiché i documenti sono profondamente innestati e le proprietà che ci interessano non sono

immediatamente fruibili; quindi, effettuiamo alcune trasformazioni per rendere i dati più facilmente accessibili.

Inizialmente utilizziamo la funzione "*explode*()" per estrapolare l'oggetto "line" dalla sezione "values" dell'UrbanDataset, dopodiché a partire da line individuiamo alcune proprietà di contesto ("id", "start_ts", "latitude", "longitude" e "property") e creiamo una colonna per ciascuna di esse. L'ultima di queste colonne contiene tutte le proprietà specifiche del dataset CounterReading relative ai consumi elettrici. Per estrapolare i dati contenuti in essa, utilizziamo ancora una volta la funzione *explode*() per esplodere la colonna "prop" e creiamo due colonne, "name" e "value". Queste colonne contengono rispettivamente i nomi delle proprietà e i loro valori, dunque, effettuata questa operazione, il DataFrame rappresenterà ogni lettura come un insieme di righe che hanno in comune i valori delle colonne "id", "timestamp", "latitude" e "longitude", e differiscono invece per il valore delle colonne "name" e "value", in quanto ciascuna riga rappresenta un singolo valore della lettura del contatore. Per far sì che a una lettura non corrisponda un insieme di righe ma una sola riga con più colonne, prima eliminiamo le colonne che non ci interessano più ("_id", "UrbanDataset", "line", "prop" ed "exploded"), e poi raggruppiamo per le colonne "id", "timestamp", "latitude" e "longitude". Fatto ciò, usiamo la funzione "*pivot*()" per trasformare i valori distinti della colonna "name" in nuove colonne, e con la funzione "*agg*()" riempiamo le colonne così create con i corrispondenti valori che si trovavano nella colonna "value".

Gli attributi del dataset sono adesso atomici, cioè non sono strutture o liste ma valori singoli immediatamente accessibili. Tuttavia, per uniformare i dati e renderli più facili da utilizzare successivamente, vengono effettuate alcune altre operazioni sulla colonna TownCode e sulla colonna "timestamp".

```
import pyspark.sql.functions as f
df1 = dataFrame.withColumn("line", f.explode("UrbanDataset.values.line")) \
        .withColumn("id", f.col("line.id")) \
        .withColumn("timestamp", f.col("line.period.start_ts")) \
        .withColumn("latitude", f.col("line.coordinates.latitude")) \
        .withColumn("longitude", f.col("line.coordinates.longitude")) \
        .withColumn("prop", f.col("line.property"))\
        .withColumn("exploded", f.explode("prop"))\
        .withColumn("name", f.col("exploded").getItem("name"))\
        .withColumn("value", f.col("exploded").getItem("val"))\
        .drop("_id", "UrbanDataset", "line", "prop", "exploded")\
        .groupby(["id", "timestamp", "latitude", "longitude"])\
        .pivot("name").agg(f.max("value"))\
        .withColumn("TownCode", f.lpad("TownCode", 8, "11"))\
        .withColumn("date", f.split("timestamp", "T").getItem(0))\
        .withColumn("time", f.split("timestamp", "T").getItem(1))\
        .withColumn("Year", f.split("date", "-").getItem(0))\
        .withColumn("Month", f.split("date", "-").getItem(1))\
        .withColumn("Day", f.split("date", "-").getItem(2))\
        .withColumn("Hours", f.split("time", ":").getItem(0))\
        .withColumn("Minutes", f.split("time", ":").getItem(1))\
        .withColumn("Seconds", f.split("time", ":").getItem(2))\
        .drop("timestamp", "date", "time")\
        .alias("df1")
```

Con queste trasformazioni, la struttura ad oggetti innestati è stata abbandonata in favore di uno schema piatto, identico a quello utilizzato nelle tabelle dei database relazionali. Lo schema

risultante, così come visualizzato sul terminale attraverso il metodo "*printSchema*()" è riportato in Fig. 7.

```
root
 |-- id: integer (nullable = true)
 |-- latitude: double (nullable = true)
 |-- longitude: double (nullable = true)
 |-- ElectricPanelID: string (nullable = true)
 |-- Line1Current: string (nullable = true)
 |-- Line2Current: string (nullable = true)
 |-- Line3Current: string (nullable = true)
 |-- PODID: string (nullable = true)
 |-- Phase1ActivePower: string (nullable = true)
 |-- Phase1ApparentPower: string (nullable = true)
 |-- Phase1PowerFactor: string (nullable = true)
 |-- Phase1ReactivePower: string (nullable = true)
 |-- Phase1Voltage: string (nullable = true)
 |-- Phase2ActivePower: string (nullable = true)
 |-- Phase2ApparentPower: string (nullable = true)
 |-- Phase2PowerFactor: string (nullable = true)
 |-- Phase2ReactivePower: string (nullable = true)
 |-- Phase2Voltage: string (nullable = true)
 |-- Phase3ActivePower: string (nullable = true)
 |-- Phase3ApparentPower: string (nullable = true)
 |-- Phase3PowerFactor: string (nullable = true)
 |-- Phase3ReactivePower: string (nullable = true)
 |-- Phase3Voltage: string (nullable = true)
 |-- TotalActiveEnergy: string (nullable = true)
 |-- TotalActivePower: string (nullable = true)
 |-- TotalApparentPower: string (nullable = true)
 |-- TotalReactiveEnergy: string (nullable = true)
 |-- TotalReactivePower: string (nullable = true)
 |-- TownCode: string (nullable = true)
 |-- Year: string (nullable = true)
 |-- Month: string (nullable = true)
 |-- Day: string (nullable = true)
 |-- Hours: string (nullable = true)
 |-- Minutes: string (nullable = true)
 |-- Seconds: string (nullable = true)
```

Figura 7 – Schema del DataFrame dopo la trasformazione

Utilizzando il metodo "*createOrReplaceTempView*()", il DataFrame trasformato viene registrato sotto forma di vista mantenuta in memoria e accessibile dalla SparkSession per effettuare le query su di essa.

7.6 Calcolo dei KPI

I KPI, o Key Performance Indicators, sono indicatori chiave di prestazione, cioè indici dell'andamento delle operazioni, solitamente calcolati in un secondo momento rispetto alla raccolta dei dati grezzi.

Il primo KPI esaminato è la potenza attiva media, misurata in kilowatt che si può calcolare e visualizzare utilizzando il seguente codice:

```
query = spark.sql('''SELECT ROUND(AVG(TotalActivePower), 3) as AverageActivePower
                     FROM readings;''')
```

Il risultato di questa prima elaborazione è corrispondente a **1.249 kW** di potenza media.

Il secondo KPI esaminato è la potenza attiva totale misurata, sempre in kilowatt, ogni ora e in ciascun comune. In questo caso la query dipende da due fattori: il comune e il momento di rilevazione (data e ora), che possono essere fatti corrispondere alle colonne TownCode, Year, Month, Day, Hours, Minutes e Seconds del DataFrame. Di seguito è mostrata la query utilizzata e l'output ottenuto dall'elaborazione tramite Spark SQL.

```
q = spark.sql('''SELECT TownCode, Year, Month, Day, (Hours || ':00-' || Hours || ':59')
as Period, TotalActivePowerByTown
    FROM (SELECT TownCode, Year, Month, Day, Hours, ROUND(SUM(TotalActivePower), 3) as
TotalActivePowerByTown
              FROM readings
              GROUP BY TownCode, Year, Month, Day, Hours
              ORDER BY TownCode, Year, Month, Day, Hours);''')
```

L'output di questa query mostra chiaramente un andamento dei consumi elettrici congruo con l'esperienza quotidiana: durante le ore in cui si può approfittare della luce solare per l'illuminazione, i consumi sono estremamente bassi, mentre nelle fasce orarie crepuscolari o notturne i consumi aumentano in risposta all'accensione dei dispositivi di illuminazione pubblica.

Il terzo KPI è il consumo totale giornaliero per ogni singolo impianto, che può essere calcolato con la seguente query:

```
q = spark.sql('''SELECT v1.PODID, v1.Year, v1.Month, v1.Day, ROUND((EndTotalActiveEnergy -
StartTotalActiveEnergy), 3) as DailyEnergyConsumption
    FROM (SELECT r1.PODID, r1.Year, r1.Month, r1.Day, r1.TotalActiveEnergy as StartTotalActiveEnergy
          FROM readings as r1
          JOIN (SELECT PODID, Year, Month, Day, min(Hours || '-' || Minutes || '-' || Seconds) as time
                FROM readings
                GROUP BY PODID, Year, Month, Day) as r2
          ON r1.PODID = r2.PODID
             AND (r1.Year || '-' || r1.Month || '-' || r1.Day) =
                                                  (r2.Year || '-' || r2.Month || '-' || r2.Day)
             AND (r1.Hours || '-' || r1.Minutes || '-' || r1.Seconds) = r2.time) as v1
          JOIN (SELECT r1.PODID, r1.Year, r1.Month, r1.Day, r1.TotalActiveEnergy as EndTotalActiveEnergy
                FROM readings as r1
                JOIN (SELECT PODID, readings.Year, Month, Day,
                                                max(Hours || '-' || Minutes || '-' || Seconds) as time
                      FROM readings
                      GROUP BY PODID, Year, Month, Day) as r2
                ON r1.PODID = r2.PODID
                   AND (r1.Year || '-' || r1.Month || '-' || r1.Day) =
                                                  (r2.Year || '-' || r2.Month || '-' || r2.Day)
                   AND (r1.Hours || '-' || r1.Minutes || '-' || r1.Seconds) = r2.time) as v2
          ON v1.PODID = v2.PODID AND v1.Year = v2.Year AND v1.Month = v2.Month AND v1.Day = v2.Day
     ORDER BY v1.PODID, v1.Year, v1.Month, v1.Day;''')
```

Per calcolare i rimanenti KPI, si può creare una vista *c1* a partire dall'output della query precedente. Ad esempio, il consumo medio giornaliero misurato da ciascun contatore può essere calcolato leggendo direttamente dalla vista c1:

```
q.createOrReplaceTempView("c1")
print('Consumo medo giornaliero')
q = spark.sql('''SELECT ROUND(AVG(DailyEnergyConsumption), 3) as AverageEnergyConsumption
                        FROM c1;''')
```

Questa query dà come risultato **28.799 kWh** di energia elettrica media consumata in un giorno da un impianto.

Infine, l'ultimo KPI è il consumo totale giornaliero per comune, calcolabile con la seguente query.

```
q = spark.sql('''SELECT r1.TownCode, r1.Year, r1.Month, r1.Day,
ROUND(sum(c1.DailyEnergyConsumption), 3) as DailyTownEnergyConsumption
                        FROM readings as r1
                        JOIN c1 ON r1.PODID = c1.PODID
                        GROUP BY r1.TownCode, r1.Year, r1.Month, r1.Day;''')
```

7.7 Esplorazione del dataset

Una volta riportate tutte le letture dei contatori in una tabella virtuale di SparkSQL, in maniera analoga a quanto fatto per il calcolo dei KPI, è immediato effettuare esplorazioni del dataset ai fini di individuare statistiche o anomalie. Sono quindi state eseguite le seguenti interrogazioni:

```
query = spark.sql('''SELECT COUNT(*)
                        FROM readings;''')
```

che indica che nel dataset sono presenti in totale **1.907.479** letture.

La query:

```
query = spark.sql('''SELECT COUNT(DISTINCT townCode)
                        FROM readings;''')
```

indica che le letture si riferiscono a **12** distinti comuni (townCode). Le letture provengono da **328** ElectricPanelID distinti e **330** PODID distinti come rilevato dalle seguenty query:

```
query = spark.sql('''SELECT COUNT(DISTINCT ElectricPanelID)
                        FROM readings;''')
query = spark.sql('''SELECT COUNT(DISTINCT PODID)
                        FROM readings;''')
```

Le seguenti query confermano che le letture presenti nel dataset sono state effettuate fra il 20/05/2022 e il 10/04/2023:

```
lastdate = spark.sql('''SELECT year, month, day
                        FROM readings
                        ORDER BY year DESC, month DESC, day DESC
                        LIMIT 1;''')
firstdate = spark.sql('''SELECT year, month, day
                        FROM readings
                        ORDER BY year, month, day
                        LIMIT 1;''')
```

Si è osservato che un limitato numero di letture riportano il valore 'null' per la proprietà Phase1Voltage, in particolare le letture non nulle risultano **1.889.866** sul totale visto prima di **1.907.479:**

```
query = spark.sql('''SELECT COUNT(*)
                        FROM readings
                        WHERE Phase1Voltage <> 'null';''')
```

Molto maggiore è invece il numero di letture nulle per Phase2Voltage e Phase3Voltage, il numero di letture non nulle risulta infatti rispettivamente **1.281.919** e **1.236.724**:

```
query = spark.sql('''SELECT COUNT(*)
                        FROM readings
                        WHERE Phase2Voltage <> 'null';''')

query = spark.sql('''SELECT COUNT(*)
                        FROM readings
                        WHERE Phase3Voltage <> 'null';''')
```

In breve, solo due terzi delle letture hanno tutte e tre le tensioni di fase non nulle.

Poiché il numero di letture con Phase2Voltage non nulla è risultato leggermente diverso dalle letture con Phase3Voltage non nulla è stata fatta un'ulteriore analisi. La query:

```
query = spark.sql('''SELECT count(*)
                        FROM readings
                        WHERE Phase2Voltage = 'null' AND Phase3Voltage != 'null';
                        ''')
```

ha rilevato che ci sono **12** letture in cui Phase2Voltage è nullo e Phase3Voltage non lo è.

Di contro le letture in cui Phase3Voltage è nullo e Phase2Voltage non lo è sono **45.207**

```
query = spark.sql('''SELECT count(*)
                        FROM readings
                        WHERE Phase3Voltage = 'null' AND Phase2Voltage != 'null';
                        ''')
```

Le letture in cui sia Phase2Voltage che Phase3Voltage sono nulle risultano **625.548**:

```
query = spark.sql('''SELECT count(*)
                        FROM readings
                        WHERE Phase3Voltage = 'null' AND Phase2Voltage = 'null';
                        ''')
```

Si osserva che il numero di ElectricPanel che hanno letture di Phase2Voltage e Phase3Voltage contemporaneamente nulle è pari a **131** sul totale di 328:

```
query = spark.sql('''SELECT count(DISTINCT ElectricPanelID)
                        FROM readings
                        WHERE Phase3Voltage = 'null' AND Phase2Voltage = 'null';
                        ''')
```

Si può immediatamente determinare quali sono questi ElectricPanel con una semplice query:

```
query = spark.sql('''SELECT DISTINCT ElectricPanelID
                        FROM readings
                        WHERE Phase3Voltage = 'null' AND Phase2Voltage = 'null';
                        ''')
```

Come controprova si visualizzano alcune letture di uno di questi:

```
query = spark.sql('''SELECT ElectricPanelID, Phase1Voltage, Phase2Voltage, Phase3Voltage
                        FROM readings
                        WHERE ElectricPanelID='IADZGEAM0400'
                        LIMIT 20;
                        ''')
```

```
+---------------+-------------+-------------+-------------+
|ElectricPanelID|Phase1Voltage|Phase2Voltage|Phase3Voltage|
+---------------+-------------+-------------+-------------+
|   IADZGEAM0400|        230.2|         null|         null|
|   IADZGEAM0400|        229.4|         null|         null|
|   IADZGEAM0400|        227.6|         null|         null|
|   IADZGEAM0400|        226.7|         null|         null|
|   IADZGEAM0400|        229.4|         null|         null|
|   IADZGEAM0400|          228|         null|         null|
|   IADZGEAM0400|          226|         null|         null|
|   IADZGEAM0400|        232.2|         null|         null|
|   IADZGEAM0400|        233.2|         null|         null|
+---------------+-------------+-------------+-------------+
```

Il risultato fa pensare che gli ElectricPanel che hanno fornito letture con Phase2Voltage e Phase3Voltage contemporaneamente nulle abbiano tutte le letture con questa caratteristica. Si può verificare per l'ElectricPanelID 'IADZGEAM0400' con la seguente query:

```
query = spark.sql('''SELECT ElectricPanelID, Phase1Voltage, Phase2Voltage, Phase3Voltage
                        FROM readings
                        WHERE ElectricPanelID='IADZGEAM0400' AND
                                (Phase2Voltage != 'null' OR Phase3Voltage != 'null');
                        ''')
```

Lo stesso test può essere effettuato per gli ElectricPanelID 'IADZGEAM0170' e 'IADZGEAM0289' e si osserva che effettivamente gli ElectricPanel che hanno letture di Phase2Voltage e Phase3Voltage contemporaneamente nulle hanno tutte le letture di fase 2 e fase 3 nulle.

SI è cercato quindi di capire l'origine delle **12** letture con Phase2Voltage nulla e Phase3Voltage non nulla. Questa query trova gli ElectricPanel che avevano letture con Phase2Voltage nulla e Phase3Voltage non nulla:

```
query = spark.sql('''SELECT ElectricPanelID, Phase1Voltage, Phase2Voltage, Phase3Voltage
                        FROM readings
                        WHERE Phase2Voltage = 'null' AND Phase3Voltage != 'null';
                        ''')
```

si è osservato che queste letture si riferiscono tutte al solo ElectricPanel IADZGEAM0180. Le letture totali di questo ElectricPanel sono **5925**:

```
query = spark.sql('''SELECT COUNT(*)
                        FROM readings
                        WHERE ElectricPanelID='IADZGEAM0180';
                        ''')
```

Con una semplice query si osserva anche che non ci sono letture Phase3Voltage nulle. Le 12 letture Phase2Voltage nulle su 5925 saranno presumibilmente dovute a un malfunzionamento temporaneo. Se necessario si potrebbe fare una query per vedere data e ora di queste letture anomale.

Sono stati quindi analizzati gli ElectricPanel che avevano Phase3Voltage nulla e Phase2Voltage non nulla per verificare se sono tutte le letture con questa caratteristica o soltanto alcune

```
query = spark.sql('''SELECT DISTINCT(ElectricPanelID)
                        FROM readings
                        WHERE Phase2Voltage != 'null' AND Phase3Voltage = 'null';
                        ''')
```

Si è osservato che queste letture anomale provengono tutte da 7 ElectricPanel. Si è verificato per uno di essi, IADZGEAM0185, che hanno questa caratteristica 8524 letture su un totale di 8524:

```
query = spark.sql('''SELECT COUNT(*)
                        FROM readings
                        WHERE ElectricPanelID='IADZGEAM0185' AND
                                Phase2Voltage !='null' AND Phase3Voltage = 'null';
                        ''')
```

Facendo la stessa verifica su IADZGEAM0113 si osserva che sono 5509 letture su un totale di 5509. Analogamente per IADZGEAM0175 si trovano 6373 letture su un totale di 6373.

Sembra quindi che questi ElectricPanel abbiano la lettura della Phase3Voltage non corretta, varificando gli altri parametri della fase 3 si osserva che sono tutti mancanti.

```
query = spark.sql('''SELECT *
                        FROM readings
                        WHERE ElectricPanelID='IADZGEAM0175'
                        LIMIT 20;
                        ''')
```

In maniera analoga si potrebbero fare le analisi sui 131 contatori che hanno letture di Phase2Voltage e Phase3Voltage contemporaneamente nulle. Da semplici interrogazioni si osserva che quei 131 ElectricPanel hanno soltanto informazioni relative alla fase 1.

8 Pubblicazioni scientifiche

Non sono state prodotte pubblicazioni scientifiche.

9 Eventi di disseminazione

Nessuno