



UNIVERSITÀ DEGLI STUDI DI SALERNO

Relazione Progettuale

Photovoltaic Panels Recognition and Geospatial Analysis using
MongoDB and Neo4j

Antony Storti

a.storti2@studenti.unisa.it

Matricola: 0622702353

DIEM

Paola Saggiomo

p.saggiomo1@studenti.unisa.it

Matricola: 0622702380

DIEM

30 novembre 2024

Abstract

Il progetto si incentra sull'uso di tecniche di *Artificial Vision* per identificare i pannelli fotovoltaici presenti nelle immagini satellitari, associando a ciascun pannello le relative coordinate geospaziali. I risultati del riconoscimento vengono memorizzati e gestiti tramite due diversi DBMS NoSQL: *MongoDB*, che offre una gestione efficiente dei dati geospaziali in formato documentale, e, *Neo4j*, un database a grafo utilizzato per analizzare le relazioni spaziali tra i pannelli solari. MongoDB si distingue per la sua velocità nelle query geospaziali e la gestione di grandi volumi di dati, mentre Neo4j eccelle nell'analisi delle connessioni e delle interazioni spaziali tra entità. Il progetto include un confronto prestazionale tra i due sistemi, esaminando criteri come: la velocità di query, l'efficienza nella gestione dei dati geospaziali e la scalabilità; con l'obiettivo finale di determinare quale piattaforma offre le migliori prestazioni in un contesto di analisi massiva di dati.

Indice

1	Introduzione	1
2	Training della Rete Neurale	3
2.1	Le immagini satellitari	3
2.2	Dataset di Addestramento	3
2.3	Approcci Risolutivi	5
2.3.1	Approccio di estrazione delle Features ^[3]	5
2.3.2	Approccio R-CNN ^[4]	7
2.3.3	Approccio di Segmentazione ^[5]	8
2.4	<i>Image Segmentation</i>	10
2.4.1	UNET++	10
2.5	<i>Object Detection</i>	13
2.5.1	YOLOv5	13
2.5.2	Perché 256x256 pixel?	15
2.5.3	Risultati e benchmarks	16
2.6	Sintesi sull'approccio utilizzato	19
3	QGIS: Ottenimento delle immagini satellitari	20
3.1	Zone OMI, ShapeFile e Layers	21
3.2	Scelta delle città	22
3.3	Uso di QGIS	22
4	MongoDB	24
4.1	Dominio di Interesse	25
4.1.1	Perchè questa scelta?	25
4.2	Cluster MongoDB in Docker	26
4.2.1	Il File System XFS	27
4.2.2	Scelta delle "Shard Keys" e della "Chunk size"	28
4.3	Strutturazione delle <i>Collections</i>	32
4.3.1	I Redditi	33
4.3.2	Le Zone OMI	34
4.3.3	Gli Inquinanti Atmosferici	35
4.3.4	I Consumi Elettrici	37
4.3.5	Le informazioni anagrafiche dei <i>Nuclei Familiari</i>	38
4.3.6	I Pannelli	39
4.4	Relazioni tra le collezioni	40
4.5	Schemi di Validazione	41
4.5.1	Schema per i <i>Consumi Energetici</i>	41
4.5.2	Schema per gli <i>Indirizzi</i>	41
4.5.3	Schema per gli <i>Inquinanti Aria</i>	42
4.5.4	Schema per i <i>Pannelli</i>	42
4.5.5	Schema per i <i>Redditi</i>	42
4.6	Scelta degli Indici	44
4.7	L'API "PyMongo"	45
4.8	Le Query	46
4.8.1	QUERY 1: Reddito medio per città	46
4.8.2	QUERY 2: Tasso di pannelli per Zona OMI	48

4.8.3	QUERY 3: Consumi per Zona OMI	50
4.8.4	QUERY 4: Inquinanti per Zona OMI	52
4.9	Osservazioni	54
5	Neo4j	55
5.1	Strutturazione dei Dati	56
5.1.1	I Redditi	57
5.1.2	Le Zone OMI	57
5.1.3	Gli Inquinanti Atmosferici	58
5.1.4	I Consumi Elettrici	58
5.1.5	Le informazioni anagrafiche dei <i>Nuclei Familiari</i>	59
5.1.6	I Pannelli	59
5.2	Relazioni tra i Nodi	60
5.3	Il Plugin " <i>Neo4j-Spatial</i> " [13]	62
5.3.1	Esempi di utilizzo	63
5.3.2	Funzionamento interno del Plugin: gli " <i>R-Tree</i> " [14]	65
5.4	Indici e Vincoli	69
5.4.1	Nodo <i>Soggetto</i>	70
5.4.2	Nodo <i>Consumo</i>	71
5.4.3	Nodo <i>Indirizzo</i>	71
5.4.4	Nodo <i>Zona OMI</i>	72
5.5	Le Query	73
5.5.1	QUERY 1: Reddito medio per città	73
5.5.2	QUERY 2: Tasso di pannelli per Zona OMI	75
5.5.3	QUERY 3: Consumi per Zona OMI	77
5.5.4	QUERY 4: Inquinanti per Zona OMI	79
5.6	Osservazioni	82
5.6.1	Le differenze tra Neo4j e MongoDB	82
5.6.2	Impatto delle differenze sulle performance di caricamento dati	84
6	Confronto Prestazionale	85
7	Conclusioni e Sviluppi Futuri	87
Indice delle Figure		90
Bibliografia		91

1 Introduzione

In questa sezione verrà fornita una panoramica generale sul progetto, esplicitando gli obiettivi, le motivazioni, e, le ragioni alla base della realizzazione del presente lavoro.

Questo documento è da intendersi come un mero resoconto dell'attività progettuale svolta, pertanto, non ci sarà una rigorosa esposizione teorica delle tematiche trattate.

Per il lettore interessato ad approfondire, si rimanda ai riferimenti bibliografici.

Inoltre, tutto il codice sorgente prodotto è reperibile al seguente [link](#).

Esso può essere liberamente consultato ed utilizzato, con le limitazioni della licenza MIT.

La parte di *Artificial Vision* del progetto si è incentrata sulla ricerca ed annesso addestramento della migliore "Rete Neurale" che fosse in grado di effettuare il riconoscimento e la geolocalizzazione dei pannelli fotovoltaici a partire da delle immagini satellitari di media risoluzione; un compito arduo che ha necessitato di un sinergico connubio di tecniche avanzate di *image segmentation* ed *object detection*, implementate tramite algoritmi di *Deep Learning*. A causa dell'indisponibilità di una GPU o di hardware prestante, ci siamo scontrati con limitazioni significative in termini di risorse computazionali.

Di conseguenza, abbiamo optato per reti neurali meno profonde e computazionalmente meno onerose. Tuttavia, per massimizzare l'efficacia del modello, è stato effettuato un tuning accurato e dettagliato degli iperparametri, adattando l'architettura e il processo di training per ottenere risultati ottimali nonostante le restrizioni hardware.

Questa scelta ha inevitabilmente esteso i tempi di addestramento, ma, ci ha permesso di bilanciare precisione ed efficienza: garantendo il completamento del task che ci eravamo prefissati. Le reti neurali utilizzate sono di tipo CNN, poiché, particolarmente efficaci nel riconoscimento di pattern visivi. Inoltre, esse non sono troppo profonde (ergo, esose di risorse computazionali) se confrontate con Transformers, GANs e RNNs.

Le reti neurali convoluzionali (CNNs) sono reti che vengono addestrate con la tecnica del *Supervised Learning*: è un tipo di apprendimento automatico in cui un modello viene addestrato utilizzando un dataset etichettato: cioè, un insieme di dati in cui ogni esempio ha un'etichetta corretta associata. L'obiettivo del modello è imparare una mappatura tra gli input (dati) e le etichette (risultati desiderati), in modo da poter fare previsioni accurate su nuovi dati mai visti prima dal modello.

Il dataset utilizzato per l'addestramento delle CNNs comprendeva immagini satellitari: sia contenenti pannelli fotovoltaici, accuratamente annotati con *bounding boxes* e relative coordinate geospaziali, sia immagini senza pannelli, utili per addestrare il modello a distinguere correttamente i pannelli dagli altri oggetti o superfici ad essi verosimiglianti. Questa diversificazione ha garantito un training bilanciato, migliorando la capacità del modello di riconoscere pannelli in ambienti complessi e riducendo al minimo i falsi positivi. Questi risultati sono stati successivamente utilizzati per effettuare analisi geospaziali, siccome, ogni pannello solare individuato ha un collegamento diretto con le sue coordinate geografiche (latitudine e longitudine).

Sebbene le CNNs possono identificare correttamente i pannelli fotovoltaici nelle immagini satellitari ed estrapolare le relative coordinate geografiche, queste informazioni, da sole, non sono sufficienti per effettuare alcuna analisi geospaziale. La rete neurale ci ha fornito i dati, ma, non ci dà nessun ausilio inerentemente alla semantica degli stessi!

Infatti, i DBMS (Data Management Systems), nel nostro caso, MongoDB e Neo4j, sono strumenti essenziali per gestire ed analizzare i dati raccolti. Senza l'uso di un sistema di gestione dei dati adeguato, non sarebbe possibile organizzare, archiviare né tantomeno sfruttare in modo efficace le informazioni spaziali derivanti dai risultati del modello.

I DBMS permettono non solo di immagazzinare i dati, ma anche di effettuare analisi avanzate come: la ricerca di aree densamente popolate da pannelli solari, il calcolo di distanze tra pannelli e l'integrazione con altre informazioni geospaziali; ad esempio, per l'analisi del potenziale solare di una zona nell'ottica di ottimizzare le nuove installazioni. Nell'era dell'exploit dell'intelligenza artificiale, è doveroso ammettere che senza i progressi avuti nel campo della gestione dei dati nulla sarebbe stato possibile!

L'intelligenza artificiale si nutre di dati per addestrare modelli sempre più sofisticati, e senza l'evoluzione dei DBMS, che oggi supportano enormi volumi di dati distribuiti, query ad alte prestazioni e funzionalità integrate per il machine learning, sarebbe stato impossibile raggiungere gli attuali livelli di innovazione.

I moderni database, sia relazionali che NoSQL, hanno reso i dati più accessibili, scalabili e analizzabili, consentendo all'IA di sfruttare appieno il proprio potenziale. In questo contesto, la gestione efficace dei dati è diventata il pilastro su cui si fondano tutte le applicazioni avanzate di AI, dal riconoscimento di immagini e linguaggio, alla guida autonoma, all'analisi predittiva.

Nel nostro scenario applicativo:

- MongoDB ci offre una base robusta per l'archiviazione, il recupero e l'analisi di grandi quantità di dati geospaziali, creando un ambiente adatto a gestire ed interrogare le informazioni relative ai pannelli fotovoltaici in modo efficiente e scalabile.
- Nel caso di Neo4j, il suo approccio basato su grafi ci consente di analizzare in modo efficiente le relazioni spaziali tra i pannelli fotovoltaici, mappando la loro distribuzione geografica. Grazie alla capacità di effettuare ricerche di prossimità ed analizzare percorsi complessi tra nodi, Neo4j può essere utilizzato per ottimizzare la distribuzione dell'energia, individuando la rete più efficiente per il trasporto dell'elettricità prodotta dai pannelli fotovoltaici. In futuro, queste capacità potrebbero essere sfruttate per ottimizzare la rete di distribuzione elettrica nazionale, migliorando l'efficienza e riducendo le dispersioni energetiche.

Al contrario, i tradizionali database relazionali non sono adatti per la gestione di grandi volumi di dati. La loro struttura rigida non si adatta facilmente ai dati non strutturati o complessi, e l'implementazione di operazioni spaziali avanzate richiede estensioni aggiuntive, come PostGIS in PostgreSQL, che possono risultare più complesse e sicuramente meno performanti rispetto alle soluzioni native offerte da MongoDB e Neo4j. Inoltre, tali sistemi non offrono una gestione ottimale delle relazioni spaziali e non sono progettati per scalare orizzontalmente con facilità, rendendoli inadatti per analisi geospaziali su larga scala. Infatti, qui si denota l'importanza e la flessibilità offerta dai NoSQL.

Risulta essere proprio questa una delle motivazioni tecniche che ci ha spinto a scegliere tale tematica per il nostro progetto, unitamente alla nostra volontà di dare un contributo, seppur piccolo, nella direzione di un mondo più ecosostenibile e prospero.

Perchè affermiamo ciò? Sarà chiaro nel seguito della trattazione...

2 Training della Rete Neurale

In questo capitolo descriveremo come abbiamo ottenuto una rete neurale funzionante, ossia, in grado di riconoscere e geolocalizzare, con un certo tasso di confidenza ($\approx 89\%$), gli impianti fotovoltaici presenti in date immagini satellitari.

Non avendo conoscenze di *Artificial Vision*, al di là di pochi nozionistici concetti appresi in un precedente corso, abbiamo dovuto accumulare un po' di conoscenze al riguardo; competenze che abbiamo usato come solidi puntatori per arrivare alla soluzione.

I vari articoli scientifici^[1] sono stati preziosissimi, seppur non fornissero una soluzione *take-away*, né la ben che minima riga di codice, ci hanno fornito la capacità di "saper mettere le mani", e, dunque, tirar fuori il giusto insieme di iperparametri.

La trattazione di questa parte del progetto non sarà troppo dispersiva, non essendo essa una parte rilevante ai fini del corso in oggetto; si rimanda il lettore interessato ad avere una visione di dettaglio della tematica in essere a consultare i riferimenti bibliografici sovraindicati.

2.1 Le immagini satellitari

L'idea iniziale che abbiamo avuto per ottenere le immagini è stata quella di usare l'API di Copernicus, che offre accesso libero e gratuito ai dati dei satelliti Sentinel.

Nelle fasi iniziali del progetto, è stato utilizzato il software *SNAP Toolbox*¹ per elaborare i dati provenienti dal satellite Sentinel-2. I dati risultanti includevano più immagini a diverse risoluzioni spaziali con le relative informazioni dello spettro elettromagnetico.

Il problema principale riscontrato nell'uso di questa fonte di dati è stata la mancanza di campioni di riferimento di pannelli solari nei dati geospaziali. Per addestrare un modello al riconoscimento degli oggetti, è necessario disporre di numerosi esempi per caratterizzare le caratteristiche comuni nel modello e garantire un rilevamento corretto.

Inoltre, la risoluzione dei dati elaborati non era sufficientemente elevata per garantire un rilevamento accurato in questo tipo di immagini. Pertanto, a causa del periodo limitato per l'implementazione del progetto e della specificità del compito, era fondamentale trovare un'alternativa migliore per ottenere i dati satellitari. Tale alternativa, *free* a causa di una falla nel sistema API di Google, si basa sull'uso combinato di QGIS e Google Maps. Per spiegazioni sul funzionamento dettagliato, si rimanda il lettore al Capitolo 3.

2.2 Dataset di Addestramento

Per addestrare un modello in grado di rilevare i pannelli solari è necessario un campione relativamente ampio di immagini. Tuttavia, vi è una carenza di dataset che contengano immagini di pannelli solari ripresi da un punto di vista satellitare.

Nell'agosto 2021, però, la *School of Engineering of China* ha pubblicato uno dei dataset più completi di immagini satellitari ($\approx 7GB$) contenenti celle fotovoltaiche^[2].

¹SNAP Toolbox, un software sviluppato dall'Agenzia Spaziale Europea, è stato progettato specificamente solo per analizzare i dati dei satelliti Sentinel.

Il dataset include immagini aeree di pannelli solari con risoluzioni spaziali di 0,8 m, 0,3 m e 0,1 m, per un totale di 3716 campioni di pannelli fotovoltaici raggruppati in base al terreno circostante (pannelli solari a terra e pannelli su tetto).

I terreni includono vari tipi di superficie, come aree arbustive, praterie, terreni coltivati, saline-alcaline e superfici acquatiche, mentre i tetti comprendono cemento piatto, tegole in acciaio e tetti in mattoni. Uno dei principali vantaggi di questo dataset rispetto ai dati satellitari grezzi è la presenza di **maschere binarie** che rappresentano ogni pannello solare, oltre al formato delle immagini (.bmp), che risulta più adatto per l'addestramento di un modello.

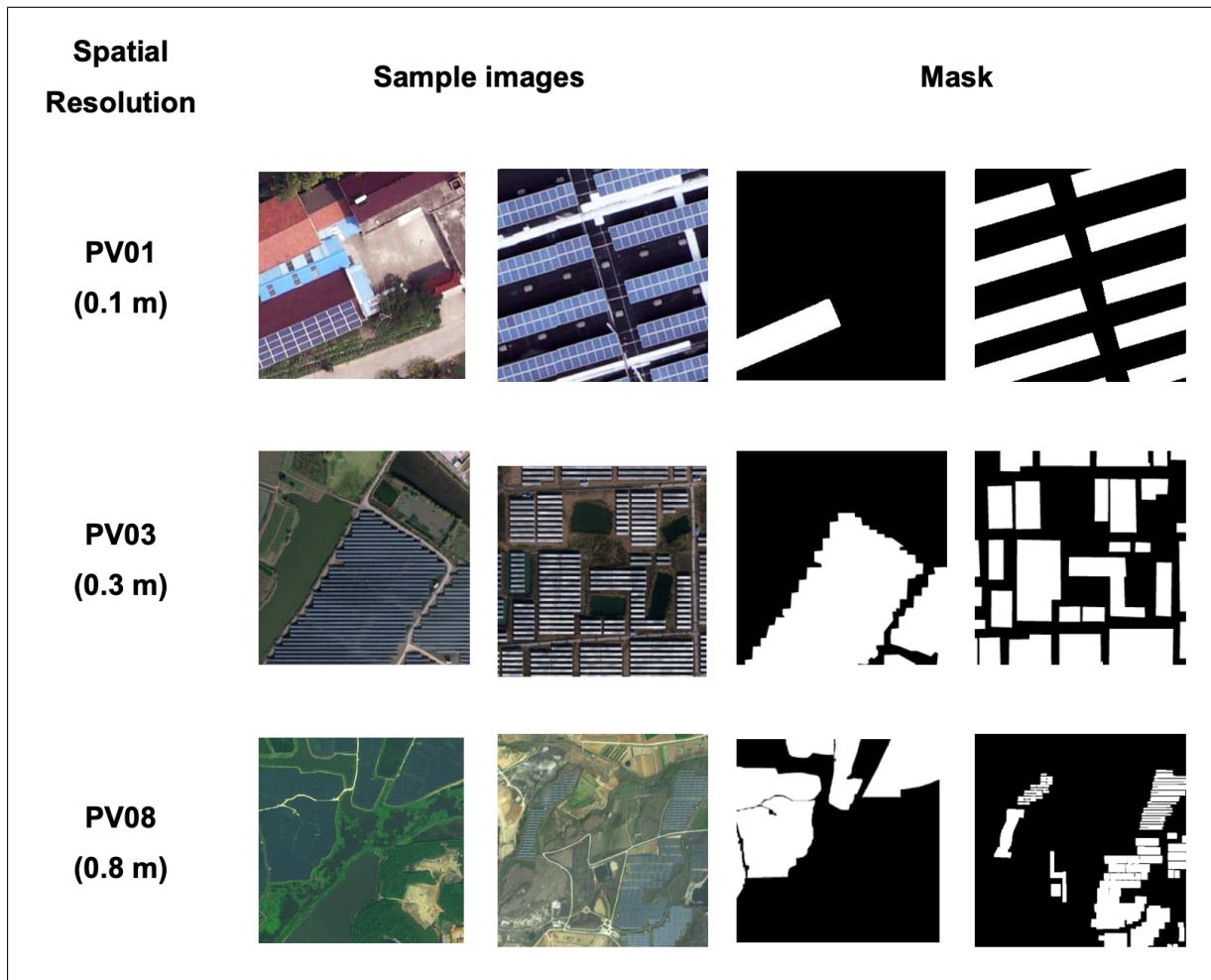


Figura 1: Contenuto del Dataset

Spatial Resolution	Ground	Rooftop	TOTAL
PV01 (0.1m)	0	645	645
PV03 (0.3m)	2122	186	2308
PV08 (0.8m)	673	90	763
			3716

Figura 2: Suddivisione per tipologia

2.3 Approcci Risolutivi

Sono stati esaminati diversi articoli e sono stati studiati i rispettivi metodi usati per rilevare i pannelli solari. La nostra soluzione è principalmente un **ibrido** basato su tre articoli che forniscono una panoramica delle tecniche più recenti e dei benchmark nel settore.

2.3.1 Approccio di estrazione delle Features [3]

Per il primo articolo, viene esaminato un approccio leggermente più datato che utilizza l'estrazione classica delle caratteristiche per elaborare le immagini ed estrarre i modelli, e una *Random Forest* (RF) per prevedere i livelli di confidenza. Un team di annotatori umani è incaricato di annotare manualmente gli array fotovoltaici (PV). I dati utilizzati provengono dall'Aerial Dataset (Fresno, USA).

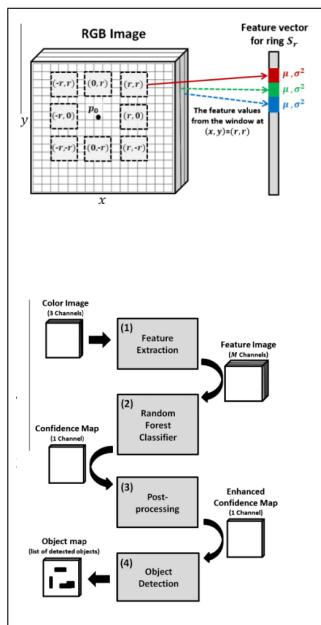


Figura 3: Estrazione delle Features con (*Random Forest*)

Il processo può essere riassunto come segue:

1. Estrazione delle caratteristiche dalle immagini attorno a ciascun pixel che caratterizzano tutti i modelli.
2. La RF assegna una probabilità o confidenza a ciascun pixel di appartenere a un array fotovoltaico.
3. Post-elaborazione per identificare i pixel ad alta confidenza (massimi locali).
4. Rilevamento degli oggetti: identificazione di gruppi di pixel ad alta confidenza contigui.

Per collegare le rilevazioni automatizzate alle annotazioni umane, viene utilizzato l' *indice di Jaccard*. A seconda dell'obiettivo del problema, è necessario un valore molto alto (quasi corrispondenza perfetta) o un valore maggiore di 0 (il modello sa dove si trova l'oggetto vero, ma non riesce a rilevarlo correttamente).

Per la performance basata sui pixel, cioè nel rilevare se un pixel appartiene a un array di pannelli solari, gli autori hanno provato a usare solo RF con e senza post-elaborazione sia nel set di addestramento che in quello di test.

I risultati hanno mostrato che RF da sola tende a sovraddattarsi nel set di addestramento e la sua performance diminuisce significativamente nel set di test. D'altra parte, l'uso di una tecnica di post-elaborazione risolve questo problema e rende il modello più capace di generalizzare le previsioni future.

Per il caso basato sugli oggetti, l'obiettivo è identificare la forma o la dimensione dei singoli array fotovoltaici. La performance in questo caso può essere valutata variando l'indice di Jaccard, poiché un valore inferiore richiede meno precisione esatta e quindi fornisce una maggiore accuratezza. I risultati mostrano che per $J=0.1$ la precisione è 0.9 (il 90% delle rilevazioni totali sono rilevazioni corrette). Tuttavia, per un indice J più elevato, la performance diminuisce, poiché diventa un compito molto più difficile.

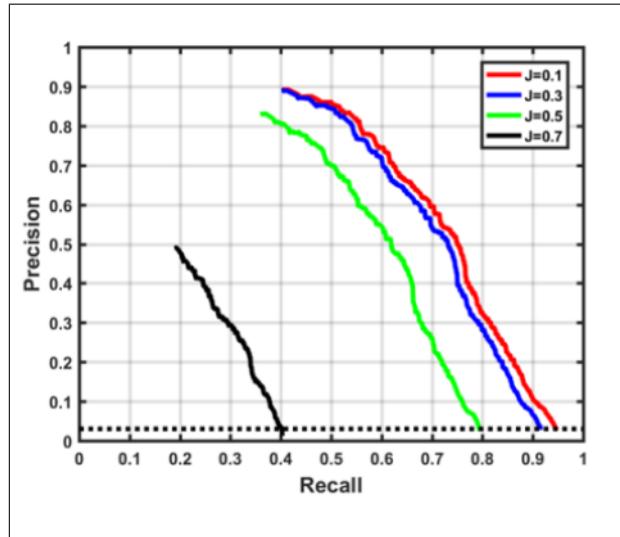


Figura 4: Risultati

In conclusione, questo articolo rappresenta un punto di partenza nel riconoscimento degli oggetti utilizzando metodi tradizionali. Non è il metodo più efficiente, ma uno dei più veloci. Per il nostro problema, non ci preoccupiamo della posizione esatta del pannello solare sul tetto, ma piuttosto del rilevamento all'interno di un edificio.

2.3.2 Approccio R-CNN [4]

Questo articolo introduce uno schema di miglioramento basato su *Faster R-CNN* per il rilevamento di piccoli oggetti nelle immagini di telerilevamento, cercando di minimizzare il tempo di elaborazione. Il modello è stato testato per il rilevamento di veicoli e aerei.

Il funzionamento è il seguente:

1. **Uso di un modello pre-addestrato:** viene utilizzato un modello pre-addestrato (VGG16) per addestrare un numero limitato di dati etichettati.
2. **Online Hard Example Mining (OHEM):** questa tecnica viene utilizzata per selezionare i campioni difficili da classificare, basandosi sulla loro perdita, accumulando i gradienti e passando questi al network convoluzionale.
3. **Modifica della risoluzione:** per mantenere la risoluzione durante l'espansione del campo ricettivo, viene rimossa la layer pool4 dal modello pre-addestrato e viene estesa la dimensione dei filtri in Conv5 a 2.
4. **Rappresentazione multi-scala:** invece di utilizzare mappe di caratteristiche di dimensioni fisse dell'ultimo layer della parte CNN per estrarre le regioni candidate, vengono utilizzate le concatenazioni dei risultati da Conv3, Conv4 e Conv5 per generare un nuovo layer convoluzionale. In questo processo vengono applicati metodi di inizializzazione chiamati Xavier, Batch Normalization e ReLU. Il diagramma di flusso dettagliato è mostrato qui sotto.
5. **Ottimizzazione delle connessioni complete:** le due layer completamente connesse per eseguire la classificazione e la regressione vengono sostituite da una layer convoluzionale e una layer di pooling al fine di ridurre il tempo computazionale, diminuendo il numero di parametri generati.

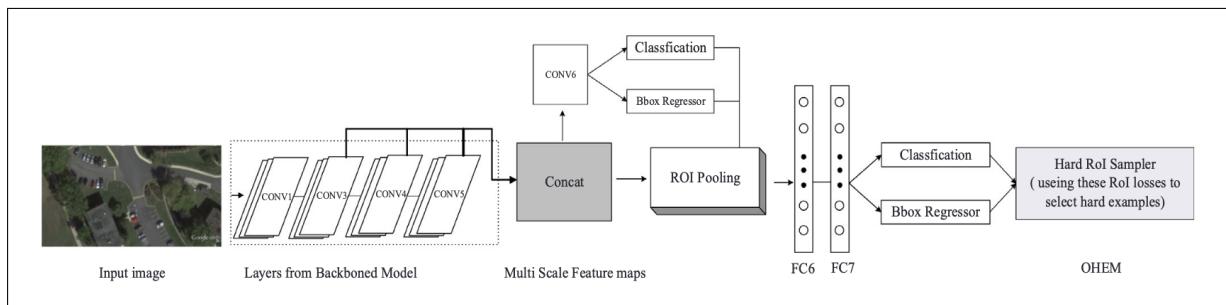


Figura 5: Funzionamento della R-CNN

Le principali metriche utilizzate per valutare la performance del modello sono precisione media e recall. Con tutte le procedure precedentemente descritte, 1000 immagini di aerei sono state addestrate, ottenendo una precisione media di 0.907 e un recall di 0.9685. Su un dataset di auto, la precisione media è risultata essere 0.879 e il recall 0.8846. Tutti i risultati sono stati migliorati rispetto a quelli ottenuti utilizzando solo Faster R-CNN con VGG16-Net.

2.3.3 Approccio di Segmentazione [5]

Questo articolo introduce un nuovo approccio al problema che vogliamo risolvere nella nostra ricerca. Esamina anche la segmentazione istantanea come primo passo per fornire una conoscenza affidabile sulla produzione di energia che una determinata area potrebbe avere. Troviamo quest'ultimo articolo di grande importanza per la quantità di diverse architetture che implementa, al fine di comprendere meglio i risultati ottenuti con l'approccio proposto.

Il funzionamento è il seguente:

- 1. Implementazione di modelli con diversi encoder:** Lo studio implementa diversi modelli con vari encoder per trovare il metodo più affidabile, verificando anche l'efficienza in termini di tempo. Alcuni dei modelli testati sono Unet, Segnet, DilatedNet, PspNet, DeepLab v3+, Dilated Resnet, mentre gli encoder scelti sono diverse variazioni di VGGNet e Resnet. L'approccio proposto consiste nell'uso di Unet-Mobilenet con Mobilenet come encoder.
- 2. Dati di input:** Vengono utilizzate 601 immagini RGB di alta qualità da 5000x5000 pixel, segmentate a 224x224 per catturare meglio i dettagli nelle immagini. Contenutualmente, vengono ottenute le maschere binarie per utilizzarle come input.
- 3. Implementazione della *dice loss* del layer:** Una delle caratteristiche chiave di questo articolo è l'implementazione di una *dice loss* layer. Questa layer affronta le bounding box in modo diverso rispetto a metodi più semplici come la cross entropy loss. Poiché questa layer è di grande importanza, la spiegheremo ulteriormente dopo aver mostrato i risultati dei vari modelli utilizzati.

Model	Encoder	Recall	Precision	
Unet	Vggnet 16	0.8695	0.8773	
Segnet	Vggnet 16	0.816	0.8252	
DilatedNet	Vggnet 16	0.6427	0.6813	
PspNet	Resnet 50	0.7002	0.6113	
DeepLab v3+	Resnet 50	0.7536	0.7319	
Dilated Resnet	Resnet 18	0.6615	0.6557	
Our other Implementations	Unet-Vggnet-BN	Vggnet 16 (with batch normalization)	0.9266	0.9227
	Unet-Vggnet-DWS	Vggnet 16 (with DWS convolution)	0.9115	0.9045
	Unet_Resnet	Resnet Blocks	0.9078	0.9198
	Fully Unet-Resnet	Resnet Blocks	0.8994	0.9421
Proposed	Unet-Mobilenet	Encoder (Mobilenet)	0.8498	0.9595

Figura 6: Risultati

$$D = \frac{2 \sum_i^N p_i g_i}{\sum_i^N p_i^2 + \sum_i^N g_i^2}$$

Figura 7: Formula della *dice loss*

Come spiegato prima, l'approccio *dice loss* è implementato in questo articolo. La sua utilità va ben oltre alcune delle ragioni menzionate nel paper, e cercheremo di spiegare perché è stato scelto. In questa formula, Pi e Gi rappresentano le coppie di valori rispettivi dei pixel nella previsione e nella verità di terra (*ground truth*).

In uno scenario di rilevamento dei bordi, i loro valori sono 0 o 1, rappresentando se il pixel è un bordo (1) o no (0). Di conseguenza, la formula rappresenta la somma dei pixel di bordo correttamente classificati divisa per la somma totale dei pixel di bordo nella previsione e nella verità a terra. Confrontando questo approccio con metodi più tradizionali, vediamo che il nostro metodo è migliore nel considerare la vicinanza ai pixel adiacenti.

La *dice loss* si comporta anche meglio con dataset sbilanciati, dove non si trovano molti pixel di bordo, come nel nostro caso, relativo al rilevamento dei pannelli fotovoltaici.

Paper Number	Type of Approach	Number of Images in Train/Test	Type of Problem	Techniques	Evaluation Criteria	Performance
1	Classical approach	Training: 1780 (90km ²), Testing: 1014 (45km ²) + Annotations	Supervised (manually annotated)	Classical feature extraction + Random Forest	PR curves (precision - recall balance) + Jaccard index (for object detection)	Object (array) detection: Precision: 0.9 (J=0.1)
2	Object detection	1000 images with about 7000 aircrafts, 500 images with about 7000 cars	Supervised pre-trained model + Domain-specific	Faster R-CNN + OHEM + Multi-scale representation	Average Precision (AP) + Recall Score	Aircraft dataset: AP: 0.907, Recall: 0.9685 Car dataset: AP: 0.879, Recall: 0.8846
3	Instance segmentation	601 images of size 5000x5000 in subsets of 224x224	Supervised learning with masks	Unet-Mobilenet + Dice loss	Recall + Precision + Time	Solar panel dataset: Precision: 0.8498, Recall: 0.9595

Figura 8: Confronto fra i tre approcci

Analizzando i risultati in Figura 8, abbiamo deciso di usare una Rete Neurale Convolutzionale (CNN) come base architetturale. Questo ci è stato di grande aiuto visto il gran numero di reti esistenti che potevano adattarsi al nostro scenario applicativo.

La seconda considerazione che abbiamo dovuto fare riguardava il tipo di approccio che il nostro modello avrebbe implementato; dopo un'attenta riflessione abbiamo deciso di implementare i due approcci più promettenti presenti in letteratura, ossia, *Object Detection* ed *Image Segmentation*.

In sintesi, entrambi gli approcci riescono a risolvere il nostro problema, ma, con tempistiche e carichi computazionali significativamente differenti! Infatti proprio tali motivazioni ci hanno spinto ad usare in produzione solo uno dei due modelli addestrati, che seppur meno preciso, era drasticamente più veloce e meno esoso di risorse...

2.4 *Image Segmentation*

L' *Image Segmentation* è un processo in cui un'immagine viene suddivisa in diverse regioni, chiamate segmenti, che corrispondono a particolari aree con caratteristiche omogenee, come colore, intensità o texture. La segmentazione è un compito più fine rispetto al rilevamento, poiché mira a identificare la forma e la posizione di ogni oggetto in modo dettagliato, piuttosto che limitarne il riconoscimento a una semplice classificazione. È particolarmente utile quando è necessario analizzare la forma esatta degli oggetti, ad esempio per il calcolo dell'area di un pannello solare o per distinguere tra diverse tipologie di terreni. Mentre il rilevamento degli oggetti fornisce una visione generale, la segmentazione offre un'analisi più precisa e localizzata. Entrambi i metodi sono complementari, con l'uso di tecniche avanzate come le CNN che permettono di ottenere ottimi risultati in entrambi i casi.

Noi abbiamo deciso di utilizzare come modello base per la rete a segmentazione: **UNET++**.

2.4.1 UNET++

UNet++ è una variante avanzata del modello U-Net, progettato specificamente per migliorare le prestazioni nella segmentazione delle immagini, un compito cruciale in molti ambiti, come la medicina, l'analisi satellitare e il rilevamento degli oggetti. UNet++ è stato introdotto per affrontare alcune delle limitazioni di U-Net, migliorando la sua capacità di segmentare oggetti complessi e di gestire meglio le informazioni spaziali a diverse risoluzioni.

Come nel caso di U-Net, UNet++ è un modello di tipo encoder-decoder, ma con alcune modifiche significative. La struttura di base di UNet++ prevede:

- Encoder (Contrazione):** Questa parte del modello si occupa di estrarre le caratteristiche a diverse risoluzioni, riducendo progressivamente la dimensione spaziale delle immagini. Ogni livello dell'encoder applica convoluzioni, normalizzazione e attivazioni per estrarre rappresentazioni sempre più astratte dell'immagine.
- Bottleneck:** Una volta che l'immagine è stata ridotta a una dimensione molto più piccola, il modello arriva al bottleneck, che è il punto centrale in cui le informazioni vengono codificate con la massima astrazione.

3. Decoder (Espansione): Dopo il bottleneck, il decoder riporta le dimensioni spaziali dell'immagine alle dimensioni originali. Qui, le informazioni estratte durante l'encoding vengono "decodificate" e ripristinate nella forma originale.

La novità principale di UNet++ sta nella sua architettura modificata, che introduce una *skip connection* più complessa tra i vari livelli del modello. Piuttosto che semplicemente collegare il livello dell'encoder con il livello corrispondente del decoder come avviene in UNet, UNet++ utilizza più connessioni a diverse profondità per ogni livello. Questo aiuta a ridurre la perdita di informazioni durante il passaggio da un livello all'altro, consentendo al modello di utilizzare più efficacemente le informazioni estratte.

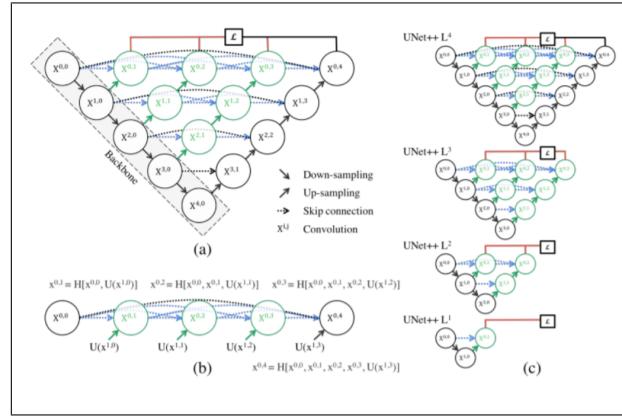


Figura 9: Architettura di UNET++

Nel nostro processo di *preprocessing* dei dati, una delle modifiche più rilevanti che abbiamo effettuato sulle immagini è stata quella di enfatizzare il canale blu dell'immagine.

Questo è stato fatto applicando **un filtro RGB** che ha accentuato il colore **blu**, con l'obiettivo di migliorare la visibilità dei pannelli solari. La scelta di usare il blu è stata dettata dal fatto che, in determinate condizioni di illuminazione e per la specifica configurazione del nostro dataset, i pannelli solari risultavano più evidenti quando visualizzati in questa componente cromatica.

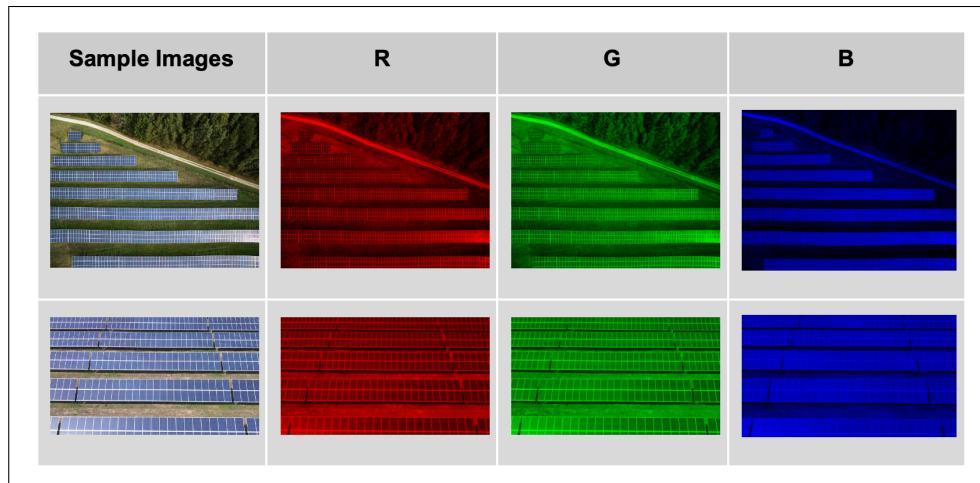


Figura 10: Preprocessing del dataset

L'applicazione di questo filtro ha permesso ai pannelli solari di emergere in modo più marcato rispetto allo sfondo, rendendo la loro rilevazione più facile per il modello. In molte immagini originali, i pannelli solari erano poco distintivi, soprattutto in presenza di altri elementi visivi come edifici o vegetazione, che potevano confondere il modello. Aumentando l'intensità del blu, siamo riusciti a ridurre l'interferenza di questi altri elementi e a rendere i pannelli più visibili, facilitando così il compito del sistema di deep learning di identificarli correttamente. Questa modifica si è rivelata particolarmente utile, poiché ha migliorato la qualità delle immagini per l'addestramento, rendendo più chiari i dettagli e facendo risaltare le caratteristiche dei pannelli solari, che sono spesso più evidenti quando visualizzati attraverso una predominanza del blu.

In questo modo, il modello ha potuto concentrarsi meglio sugli oggetti di interesse, migliorando l'efficacia nel rilevamento dei pannelli solari.

Questa è la rete che ha ottenuto in assoluto le prestazioni migliori, sia sul set di train che su quello di test, ma, è troppo lenta ed esosa di risorse computazionali da renderla inutilizzabile per l'hardware ed il tempo che avevamo a disposizione...

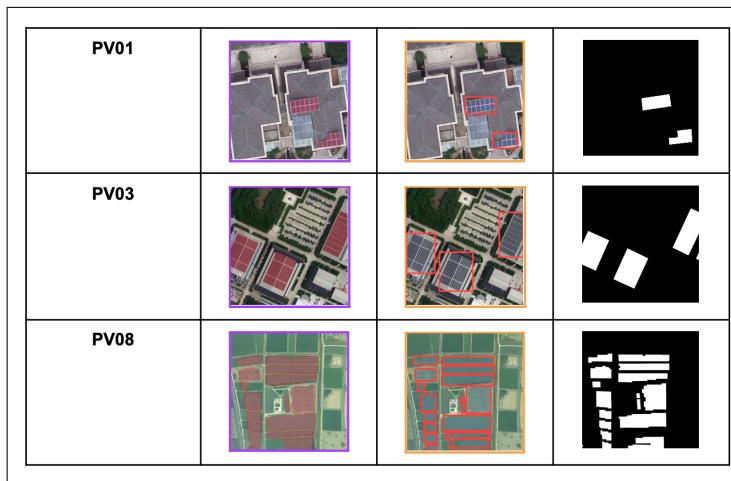


Figura 11: Funzionamento di UNET++

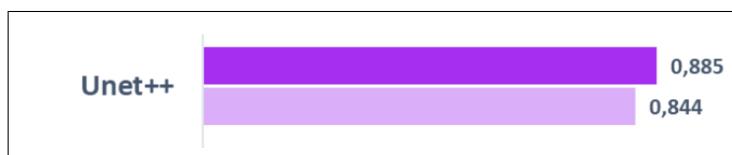


Figura 12: Accuracy di UNET++

In sintesi, UNet++ è un'evoluzione potente di U-Net, progettata per migliorare la segmentazione delle immagini, soprattutto in contesti complessi. Le sue connessioni dense consentono una gestione più efficiente delle informazioni provenienti da diverse risoluzioni, migliorando l'accuratezza del modello. Tuttavia, la sua maggiore complessità computazionale e i requisiti di tempo di addestramento sono fattori da considerare, soprattutto in ambienti con risorse limitate. Per compiti di segmentazione complessi, come il rilevamento dei pannelli solari da immagini satellitari, UNet++ rappresenta un'opzione molto promettente.

2.5 Object Detection

L' *Object Detection* è il processo attraverso cui un sistema automatizzato identifica e classifica specifici oggetti all'interno di un'immagine. Questo compito implica non solo l'individuazione della presenza di un oggetto, ma anche la sua localizzazione precisa, solitamente tramite un **bounding box** che ne delinea i confini. Il rilevamento degli oggetti è particolarmente utile in applicazioni come il monitoraggio di veicoli, persone, o, come nel caso del nostro progetto, i pannelli solari. Grazie all'uso di CNN, è possibile ottenere risultati molto accurati, anche in ambienti complessi, in quanto queste reti sono in grado di apprendere autonomamente le caratteristiche distinctive degli oggetti da grandi quantità di dati.

2.5.1 YOLOv5

YOLOv5 è una delle versioni più avanzate ed utilizzate della famiglia di algoritmi YOLO (*You Only Look Once*), progettata specificamente per il compito di rilevamento degli oggetti. Questo modello si distingue per la sua capacità di combinare alta velocità e precisione, rendendolo particolarmente adatto ad applicazioni in tempo reale. Introdotto nel 2020 da Ultralytics, YOLOv5 è diventato rapidamente una delle scelte preferite da sviluppatori e ricercatori grazie alla sua semplicità d'uso e alle sue prestazioni ottimizzate. Una delle caratteristiche principali di YOLOv5 è la sua scalabilità: il modello è disponibile in diverse dimensioni, come YOLOv5s (small), YOLOv5m (medium), YOLOv5l (large) e YOLOv5x (extra large). Questo significa che può essere adattato a diversi scenari, bilanciando le risorse hardware disponibili e le esigenze di precisione o velocità. Inoltre, essendo sviluppato in PyTorch, YOLOv5 è altamente personalizzabile, il che lo rende accessibile anche a chi ha una conoscenza avanzata di deep learning.

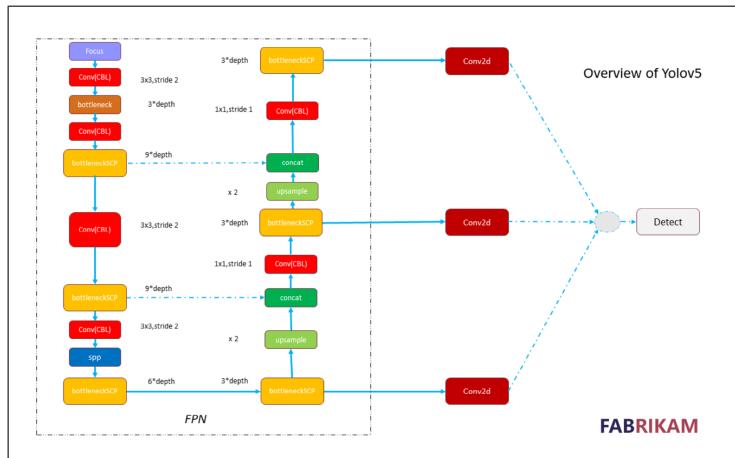


Figura 13: Architettura di YOLOv5

Dal punto di vista architettonico, YOLOv5 segue una struttura a tre componenti: il **backbone**, che estrae le caratteristiche principali delle immagini; il **neck**, che combina queste informazioni per migliorarne l'interpretazione; e l' **head**, che genera le previsioni finali, come le classi degli oggetti, le coordinate dei bounding box e i livelli di confidenza. Grazie a innovazioni come l'uso di CSPDarknet come backbone e di una Path Aggregation Network (PANet) come neck, il modello riesce a bilanciare efficienza ed accuratezza in modo ottimale.

Un altro punto di forza di YOLOv5 è l'integrazione di tecniche avanzate di preprocessing e data augmentation, come il mosaic, che permette di combinare più immagini in un'unica rappresentazione. Questo migliora la capacità del modello di generalizzare a nuovi dati, riducendo il rischio di overfitting anche con dataset limitati.

Siccome YOLOv5 vuole in input solo immagini in formato PNG, abbiamo dovuto effettuare una fase di conversione dal nostro GeoTIFF iniziale; come illustrato nell'immagine sottostante:

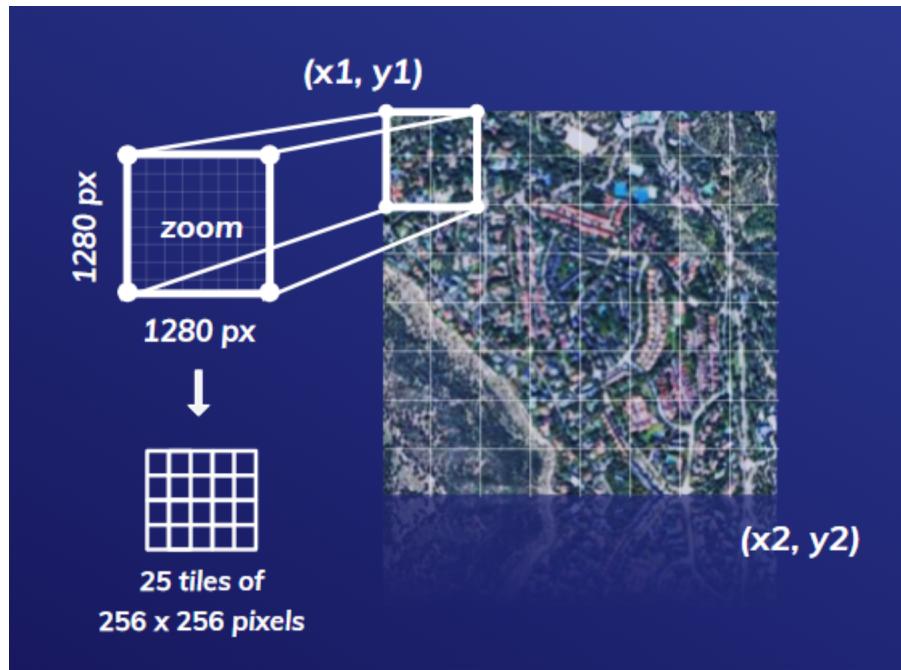


Figura 14: Processo di conversione in PNG

Abbiamo deciso di suddividere il file GeoTIFF in tiles PNH di dimensioni 256x256 pixel, un formato ideale per garantire che il modello di deep learning potesse concentrarsi sui dettagli delle immagini e rilevare con maggiore precisione i pannelli solari. Questo approccio, combinato con l'utilizzo di file associati contenenti informazioni geografiche, ha rappresentato un punto chiave del nostro processo di preprocessing. Ogni tile generato non era un'entità isolata; abbiamo creato file associati per ciascun blocco. Questi file contenevano informazioni essenziali come:

- Le coordinate geografiche del tile.
- La proiezione e il sistema di riferimento del file GeoTIFF originale.
- I bordi (*bounding box*) del tile nel sistema di coordinate geografico.

Questa struttura ha garantito che ogni immagine mantenesse il suo contesto geospaziale originale. In pratica, i file associati ci hanno permesso di:

1. Mantenere la mappatura geografica: Le predizioni del modello, come la posizione dei pannelli solari, potevano essere ricollegate facilmente al mondo reale.
2. Garantire la replicabilità: Ogni tile e i suoi dati geografici erano tracciabili, consentendo verifiche o integrazioni successive.

2.5.2 Perché 256x256 pixel?

La scelta delle dimensioni dei tile è stata strategica, in quanto, tile più grandi, come 512x512 o 1024x1024, avrebbero potuto sembrare più efficienti in termini di gestione del dataset, ma avrebbero ridotto la capacità del modello di concentrarsi su dettagli specifici. I pannelli solari, spesso piccoli e con confini poco distinti rispetto all'ambiente circostante, risultano più difficili da individuare su tile più grandi, dove altri elementi del paesaggio possono "distrarre" il modello.

Le immagini sottostanti renderanno chiara la cosa:

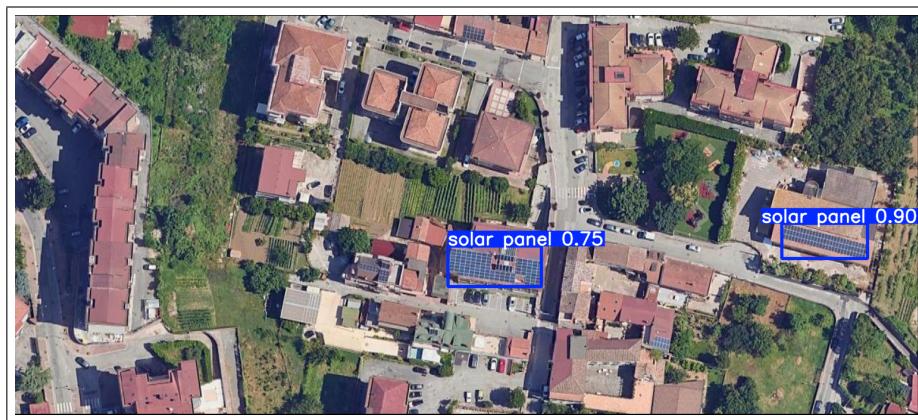


Figura 15: Tile di 1024x1024



Figura 16: Tile di 256x256

2.5.3 Risultati e benchmarks

L'ottimizzazione degli iperparametri è una parte cruciale del processo di addestramento di modelli di deep learning, in particolare quando si applicano tecniche di *augmenting* dei dati. L'augmenting dei dati è utilizzata per arricchire il dataset originale, creando nuove varianti delle immagini che il modello può utilizzare per migliorare la sua capacità di generalizzare a nuovi dati.

Le tecniche di augmenting dei dati includono modifiche come HSV (Hue, Saturation, Value), che altera i colori e la saturazione delle immagini, aiutando il modello a diventare più robusto a variazioni di illuminazione o condizioni atmosferiche. La traduzione sposta l'immagine lungo gli assi orizzontale e verticale, consentendo al modello di essere più resilienti rispetto alla posizione degli oggetti nelle immagini. Il ridimensionamento (scaling) cambia la dimensione dell'immagine per simulare la variazione delle dimensioni degli oggetti, mentre la deformazione (shearing) altera l'orientamento dell'immagine, migliorando la capacità del modello di riconoscere oggetti inclinati. L'inversione (flipping) è una trasformazione semplice che ribalta orizzontalmente o verticalmente le immagini, aumentando la varietà dei dati di addestramento. Infine, la tecnica del mosaico crea nuove immagini combinando più immagini più piccole, simulando diverse scene che potrebbero essere utili per addestrare il modello in ambienti complessi.

Tutti questi metodi sono gestiti regolando specifici iperparametri nel modello, che determinano l'intensità o la frequenza di ciascuna trasformazione applicata alle immagini durante il processo di addestramento. L'ottimizzazione accurata di questi iperparametri aiuta il modello a diventare più robusto, migliorando le sue performance in scenari reali dove le condizioni possono variare notevolmente.

Nella figura sottostante sono riportati gli iperparametri di default:

DEFAULT HYPERPARAMETERS					
lr0	0.01	lrf	0.1	momentum	0.937
weight_decay	0.0005	warmup_epochs	3.0	warmup_momentum	0.8
warmup_bias_lr	0.1	box	0.05	cls	0.5
cls_pw:	1.0	obj	1.0	obj_pw	1.0
iou_t:	0.2	anchor_t:	4.0	fl_gamma:	0.0
hsv_h	0.015	hsv_s	0.7	hsv_v	0.4
degrees	0.0	translate	0.1	scale	0.5
shear	0.0	perspective	0.0		
flipud	0.0	fliplr	0.5		
mosaic	1.0	mixup	0.0	copy_paste	0.0

Figura 17: Iperparametri di default

Di seguito i benchmarks dei vari risultati ottenuti variando gli iperparametri:

Batch size: 16	Yolov5s				Yolov5m			
	EPOCHS	Precision	Recall	mAP	F1 score	Precision	Recall	mAP
25	0.869	0.778	0.825	0.821	0.879	0.772	0.821	0.813
50	0.893	0.794	0.836	0.841	0.904	0.763	0.829	0.827
100	0.918	0.786	0.841	0.847	0.918	0.786	0.841	0.847

Batch size: 16	Yolov5l				Yolov5x			
	EPOCHS	Precision	Recall	mAP	F1 score	Precision	Recall	mAP
25	0.895	0.782	0.831	0.834	0.895	0.782	0.831	0.834
50	0.899	0.775	0.828	0.832	0.893	0.794	0.836	0.841
100	0.922	0.775	0.836	0.842	0.921	0.784	0.840	0.847

Figura 18: Benchmarks delle reti

	Initial learning rate	Momentum	Weight_decay	hsv-h	hsv-s	hsv-v
For Baseline	0.01	0.937	0.0005	0.015	0.7	0.4
After Evolution	0.01162	0.9334	0.00035	0.01002	0.55308	0.31491

	Translate	Scale	Mosaic	Mixup	Copy-paste	Anchors
For Baseline	0.1	0.9	1	0.1	0.1	3
After Evolution	0.129	0.77614	0.82128	0.09367	0.0979	3.556

Figura 19: Iperparametri utilizzati

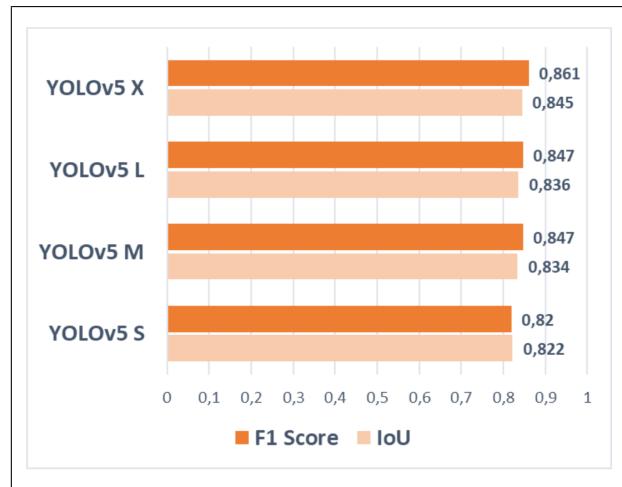


Figura 20: Risultati Complessivi

In definitiva, YOLOv5X, tra le quattro famiglie YOLO, rappresenta la soluzione potente e versatile per il rilevamento degli oggetti. La sua combinazione di semplicità, velocità e precisione lo rende uno strumento ideale per affrontare sfide complesse in vari settori, senza sacrificare l'efficienza o la flessibilità. Motivi per cui abbiamo scelto di usare in produzione questa rete e di scartare le altre.

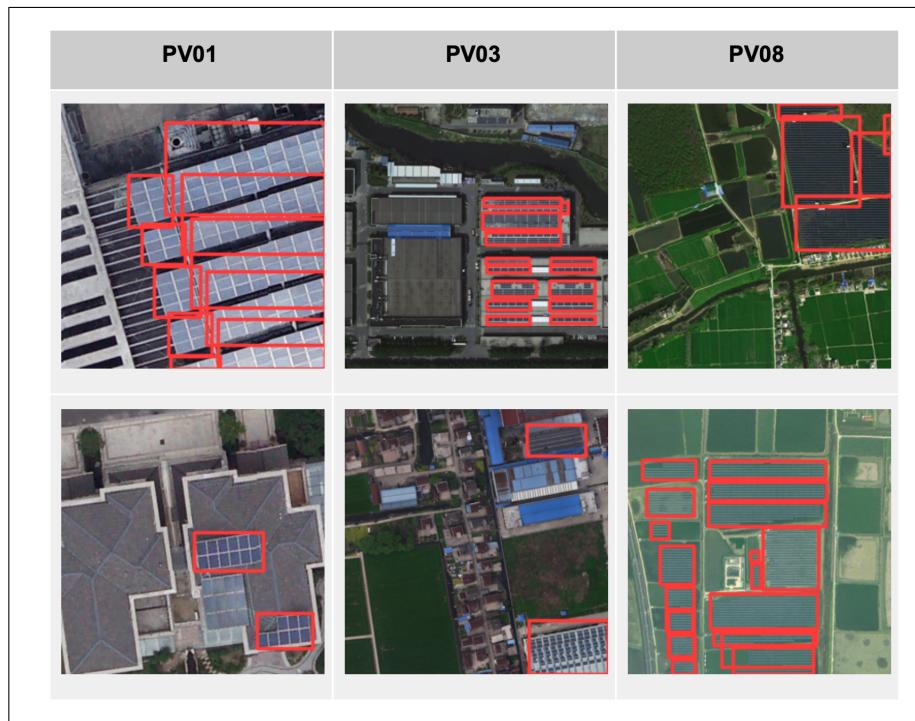


Figura 21: Funzionamento di YOLOv5

Possiamo vedere che i risultati ottenuti sono abbastanza buoni per una prima implementazione, ma l'output che l'algoritmo produce è ancor più promettente!

2.6 Sintesi sull'approccio utilizzato

L'immagine seguente sintetizza l'approccio generale che abbiamo utilizzato per addestrare entrambi i modelli (UNET++, e, YOLOv5):

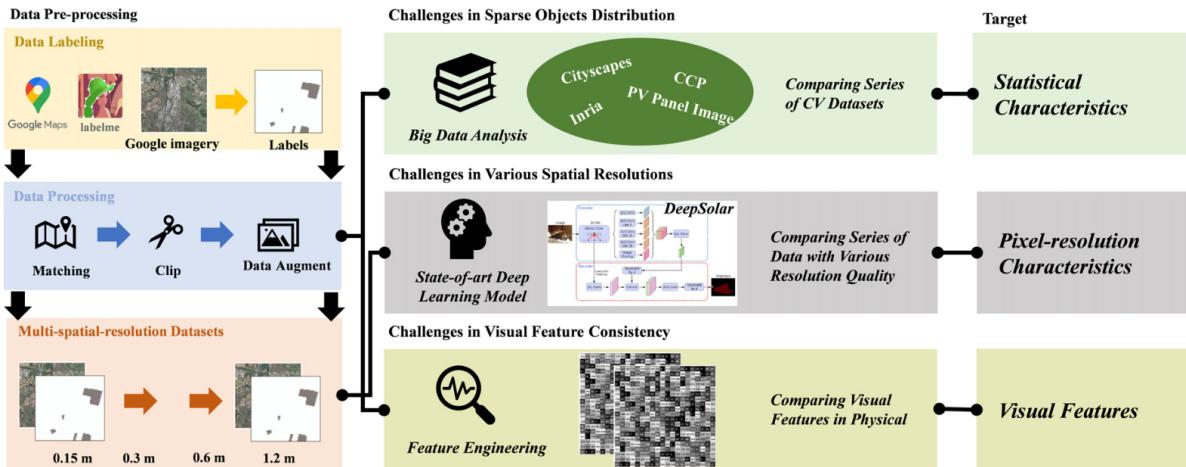


Figura 22: Processo di Addestramento

Siamo partiti dalle immagini satellitare in formato GeoTIFF, le abbiamo ”spezzettate” e suddivise in tiles PNG di 256x256, siccome, abbiamo notato empiricamente che era la dimensione ideale per non sovraccaricare la rete con eccessive informazioni visive. Abbiamo addestrato due reti neurali che approcciavano il problema in maniere totalmente differenti: YOLOv5 e UNET++. Nonostante quest’ultima fosse quella con *accuracy* maggiore, essa era troppo lenta ed esosa di risorse computazionali; pertanto, abbiamo optato per usare la rete YOLOv5, che sfrutta il paradigma dell’*Object Detection*.

Durante tutti i nostri esperimenti, i nostri dati sono stati un set di immagini con le rispettive etichette annotate, che sono state inserite nel modello. A parte le tecniche di aumento dei dati applicate durante il caricamento dei batch per addestrare i modelli, non sono state apportate altre modifiche prima di estrarre le mappe delle caratteristiche (nel caso di YOLO) con i layer convoluzionali.

Questo approccio semplice è stata l’idea iniziale degli sviluppatori di YOLO. Non volevano usare algoritmi complessi programmati manualmente per estrarre caratteristiche rilevanti dalle immagini di input. Volevano che la CNN imparasse da sola quelle caratteristiche rilevanti.

3 QGIS: Ottenimento delle immagini satellitari

L'acquisizione di immagini satellitari è stato chiaramente il primo passo, e quello più cruciale, dell'intero progetto: senza avere immagini di discreta qualità e risoluzione, sarebbe stato impensabile ottenere un modello neurale in grado di effettuare la geolocalizzazione dei pannelli fotovoltaici. Per questo progetto, è stato utilizzato QGIS, una piattaforma GIS open-source, che consente di accedere ad un'ampia gamma di dati satellitari, sia gratuiti che provenienti da satelliti *enterprise*, pertanto, a pagamento. Grazie alla sua versatilità, QGIS permette di selezionare con precisione le aree geografiche di interesse, configurare connessioni a servizi WMS/WMTS e scaricare immagini ad alta risoluzione necessarie per l'elaborazione tramite gli algoritmi di visione artificiale.

L'acquisizione delle immagini satellitari è stata effettuata utilizzando il plugin di QGIS **QuickMapServices** [6], che ha permesso di accedere rapidamente alle immagini satellitari provenienti dal servizio *Google Maps*. Tramite tale plugin siamo riusciti ad aggirare la necessità di avere un'API Google (chiaramente a pagamento!)

Infatti, quando, e se, Big G noterà questa falla, il plugin non sarà più funzionante... Le immagini satellitari di Google Maps sono considerate tra le migliori al mondo principalmente per la loro alta qualità visiva, gli aggiornamenti frequenti e la capacità di presentare dettagli in modo chiaro e accessibile. Questo è possibile grazie a una combinazione di satelliti commerciali avanzati, come quelli forniti da DigitalGlobe, ed un'ottimizzazione tecnologica che permette di ottenere immagini filtrate da ogni tipo di disturbo con una risoluzione molto alta, che può arrivare fino a 30 cm per pixel, o, anche meno.

Questo livello di dettaglio consente di visualizzare con precisione edifici, strade e anche veicoli. D'altra parte, i satelliti del programma Sentinel dell'ESA, come Sentinel-2, hanno una risoluzione spaziale inferiore, che varia tra i 10 e i 60 metri a seconda della banda spettrale utilizzata. Sebbene Sentinel fornisca immagini utili per analisi ambientali, non raggiunge lo stesso livello di dettaglio che Google Maps è in grado di offrire.

Inoltre, Google Maps si distingue per gli aggiornamenti frequenti delle sue immagini. Grazie alla combinazione di satelliti e di veicoli come le auto di Google Street View, è in grado di offrire una copertura regolare delle aree più densamente popolate e di grande interesse, mantenendo le immagini sempre fresche e accurate.

Al contrario, i satelliti Sentinel-2 acquisiscono immagini ogni cinque giorni per la stessa area, ma la loro capacità di monitorare in modo continuo e regolare le aree urbane e specifiche non è paragonabile a quella di Google Maps, che aggiorna molto più spesso le immagini nelle zone più popolose. Un altro punto di forza di Google Maps è la sua focalizzazione sulla qualità visiva delle immagini. Le immagini satellitari di Google sono progettate per essere visivamente piacevoli, con colori naturali e dettagli ben definiti, ottimizzati per la visualizzazione su piattaforme online.

Questo le rende particolarmente adatte all'uso quotidiano da parte di utenti che cercano un'esperienza visiva chiara e precisa. Le immagini di Sentinel, invece, sono ottimizzate per applicazioni scientifiche e ambientali. Utilizzano immagini multispettrali che permettono di monitorare fenomeni come la vegetazione o l'umidità del suolo, ma non sono pensate per fornire dettagli visivi di alta qualità per l'utente medio.

Infine, Google Maps si distingue anche per la sua estrema usabilità. La piattaforma è progettata per un pubblico ampio e non richiede competenze tecniche per essere utilizzata. Gli utenti possono facilmente navigare, ingrandire e esplorare le immagini satellitari in modo intuitivo. Al contrario, le immagini di Sentinel sono accessibili principalmente a

ricercatori, scienziati e professionisti attraverso piattaforme come il *Copernicus Open Access Hub*, che richiede una certa conoscenza tecnica per essere sfruttato appieno. Inoltre, mentre le immagini di Sentinel sono più adatte a studi scientifici, di natura totalmente diversa rispetto allo scopo del nostro progetto! Noi avevamo solo bisogno di immagini di alta risoluzione, senza nuvole, e rapidamente convertibili in GeoTIFF.

3.1 Zone OMI, ShapeFile e Layers

Il primo passo è stato scaricare gli ShapeFile (*.shp*) contenenti i confini OMI delle città di interesse. Successivamente, è stata eseguita un'intersezione tra le aree geografiche definite dallo ShapeFile e le immagini satellitari, al fine di estrarre le immagini satellitari solo delle zone pertinenti. Per ottenere tali file abbiamo usato il servizio web gratuito messo a disposizione dall'*Agenzia delle Entrate* [7].

Attraverso questa operazione, è stato possibile ottenere un GeoTIFF contenente le immagini satellitari delle aree selezionate, pronte per essere utilizzate nel successivo processo di riconoscimento e geolocalizzazione dei pannelli fotovoltaici.

In questo capitolo, vengono descritti i passaggi chiave del processo: dalla configurazione iniziale di QGIS ed il collegamento ai provider di dati satellitari, alla selezione e al download delle immagini, fino alla loro preparazione per il *preprocessing*.

Gli ShapeFile sono un formato di dati spaziali usato principalmente nei GIS per rappresentare oggetti geografici e relativi attributi. Sviluppati da ESRI nel 1998, essi sono composti da diversi file interconnessi, di cui almeno tre sono essenziali:

- **.shp**: contiene le geometrie degli oggetti (punti, linee, poligoni).
- **.shx**: è l'indice spaziale, che facilita l'accesso ai dati geometrici.
- **.dbf**: è un file in formato tabellare che contiene gli attributi associati alle geometrie (simile ad un foglio di calcolo).

Oltre a questi, possono esserci file opzionali come:

- **.prj**: definisce il sistema di coordinate e la proiezione.
- **.cpg**: specifica la codifica dei caratteri.

Gli ShapeFile sono ampiamente usati grazie alla loro semplicità e compatibilità con diversi software GIS, ma hanno alcune limitazioni, come il supporto per un solo sistema di coordinate e la mancanza di supporto nativo per le geometrie 3D od attributi complessi. Tali problematiche hanno portato alla nascita del formato *GeoJSON*, ma, si sa che la P.A. è sempre restia ai cambiamenti. Infatti, non sono reperibili dati ufficiali in tale formato, anche se il web è pieno di tool di conversione, noi abbiamo preferito usare uno script Python da noi scritto.

3.2 Scelta delle città

Abbiamo scelto di mappare la densità di pannelli fotovoltaici installati in sei città italiane:

- Avellino
- Napoli
- Roma
- Milano
- Genova
- Venezia

Sono state scelte queste, tra le innumerevoli possibili, poiché solo di esse avevamo dati certi sul numero di impianti fotovoltaici presenti. Questo aspetto è fondamentale: solo confrontando questi dati con quelli che ci restituiva la rete neurale potevamo essere certe che quest'ultima fosse funzionante! [8]

3.3 Uso di QGIS

Di seguito si riportano due esempi d'uso del software:

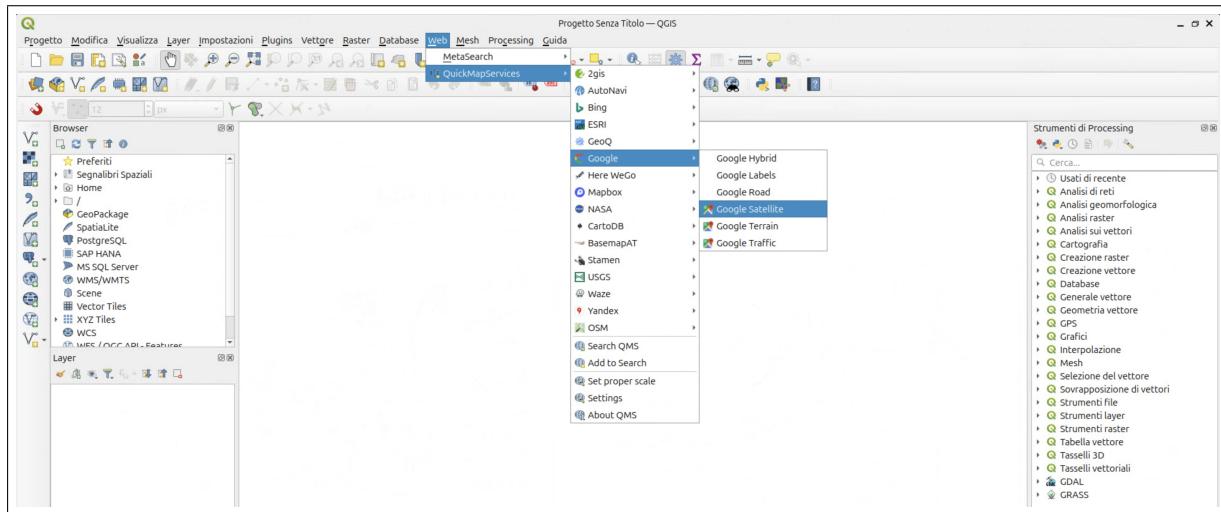


Figura 23: QuickMapServices

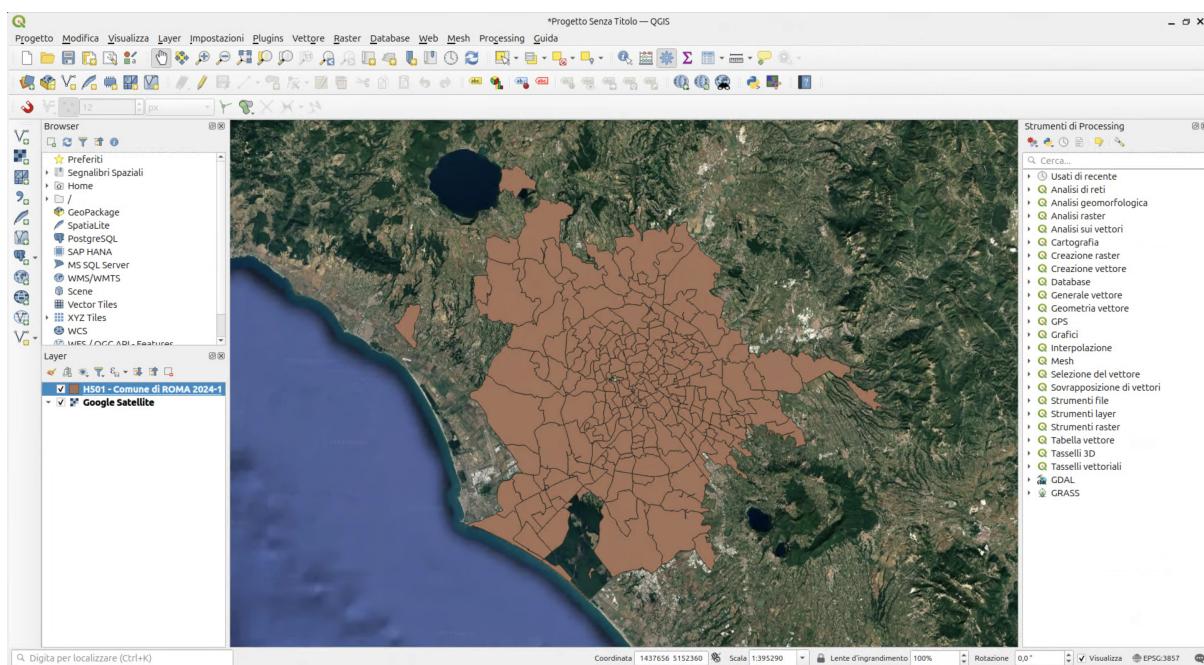


Figura 24: Intersezione dei Layers

4 MongoDB

I sistemi di gestione dei dati (DBMS), nel nostro caso, MongoDB e Neo4j, rappresentano strumenti fondamentali per la gestione e l'analisi dei dati raccolti. Senza un sistema di gestione adeguato, sarebbe impossibile organizzare, archiviare e sfruttare efficacemente le informazioni spaziali derivanti dai risultati del modello.

I DBMS consentono non solo di archiviare i dati, ma anche di eseguire analisi avanzate, come identificare aree densamente popolate da pannelli solari, calcolare le distanze tra pannelli e integrare altre informazioni geospaziali, ad esempio per valutare il potenziale solare di una zona e ottimizzare nuove installazioni.

Nell'era apicale dell'intelligenza artificiale, è evidente che i progressi nella gestione dei dati hanno avuto un ruolo cruciale. L'IA si basa sui dati per addestrare modelli sempre più sofisticati, e senza l'evoluzione dei DBMS – che oggi supportano enormi volumi di dati distribuiti, query ad alte prestazioni e funzionalità integrate per il machine learning – sarebbe stato impossibile raggiungere i livelli di innovazione attuali.

I database moderni, sia relazionali che NoSQL, hanno reso i dati più accessibili, scalabili e analizzabili, permettendo all'intelligenza artificiale di esprimere appieno il proprio potenziale.

Nell'ambito del nostro progetto, una volta geolocalizzati gli impianti fotovoltaici presenti nelle sei città scelte (vedi § 3.2), era fondamentale sia conservare tali informazioni, che, ovviamente, renderle utili ad un qualche scopo operativo concreto.

Per fare ciò, abbiamo pensato di sfruttare le enormi potenzialità offerte da MongoDB nella gestione di dati spaziali. MongoDB si è rivelato una scelta eccellente per diverse ragioni: in primo luogo, il supporto nativo per i dati geospaziali ci ha consentito di archiviare e interrogare le coordinate GPS degli impianti con grande efficienza. Grazie agli indici geospaziali di MongoDB, è stato possibile effettuare query complesse, come l'individuazione di impianti fotovoltaici situati in specifiche aree urbane o il calcolo della densità di pannelli in una determinata regione.

Inoltre, l'integrazione con strumenti di analisi e visualizzazione è stata immediata grazie all'integrazione col linguaggio Python, permettendoci di rappresentare le elaborazioni dei dati geolocalizzati su mappe interattive. Questo approccio ha aperto nuove prospettive operative, come la possibilità di pianificare interventi di manutenzione o di stimare la capacità produttiva degli impianti fotovoltaici in base alla loro distribuzione territoriale. Un altro aspetto fondamentale è stata la capacità di MongoDB di gestire dati non strutturati. Abbiamo potuto associare ai dati spaziali informazioni aggiuntive, come le caratteristiche tecniche dei pannelli o dati temporali relativi alla loro installazione, senza essere vincolati a una struttura rigida. Questa flessibilità si è dimostrata cruciale per un progetto in continua evoluzione come il nostro.

In sintesi, MongoDB ha permesso non solo di conservare le informazioni relative alla geolocalizzazione degli impianti fotovoltaici, ma anche di trasformarle in un asset strategico, utile per analisi approfondite e decisioni operative mirate.

4.1 Dominio di Interesse

Essendo nell'ambito di un corso incentrato sui sistemi NoSQL, vi era la necessità di disporre di un grande quantitativo di dati (possibilmente non strutturati); motivo per cui abbiamo deciso di espandere quelli relativi ai pannelli con altri dati: semanticamente affini ed a quest'ultimi relazionabili. [Così abbiamo circa 10 milioni di documenti!]

Nello specifico abbiamo deciso di raccogliere, per ognuna delle sei città, le seguenti informazioni:

- Informazioni anagrafiche e reddituali degli abitanti [9].
- Valori degli inquinanti atmosferici (per Zona OMI)
- Consumi elettrici (per Zona OMI)

Purtroppo avere dati reddituali vecchi di vent'anni rende il tutto vano, ma, per questioni di privacy, non vi è modo di recuperare dati recenti!

4.1.1 Perchè questa scelta?

Questa scelta non è casuale, ma risponde a precise motivazioni legate agli obiettivi del nostro progetto e alle esigenze di comprensione dei fenomeni correlati alla diffusione e all'uso degli impianti fotovoltaici. Il nostro obiettivo non è solo monitorare la distribuzione degli impianti fotovoltaici, ma anche comprendere il loro impatto socio-economico ed ambientale. Raccogliere dati eterogenei e correlati permette di individuare le aree più bisognose di interventi.

- Le *informazioni anagrafiche e reddituali* degli abitanti sono fondamentali per comprendere il contesto socio-economico delle diverse zone urbane. Ad esempio, il reddito medio può influire sulla probabilità che una famiglia o un'impresa scelga di installare un impianto fotovoltaico. Le aree con redditi più elevati potrebbero avere una maggiore capacità di investimento iniziale, mentre le aree a reddito medio-basso potrebbero richiedere incentivi o finanziamenti per adottare tecnologie green. Questi dati ci permettono di formulare analisi di fattibilità e proposte mirate per favorire l'adozione delle energie rinnovabili.
- I *valori degli inquinanti atmosferici*, raccolti per Zona OMI (una suddivisione standardizzata delle città in aree omogenee), sono cruciali per valutare l'impatto ambientale degli impianti fotovoltaici. Una riduzione delle emissioni locali di CO₂ e altri inquinanti può essere correlata alla diffusione delle energie rinnovabili. Inoltre, comprendere quali aree soffrono di maggiore inquinamento può aiutare a individuare le zone dove l'installazione di impianti fotovoltaici potrebbe avere il maggior impatto positivo sulla qualità dell'aria e, di conseguenza, sulla salute pubblica.
- I *dati sui consumi elettrici*, anch'essi organizzati per Zona OMI, sono essenziali per calcolare il fabbisogno energetico delle diverse aree urbane. Questi dati ci permettono di stimare la capacità di produzione necessaria degli impianti fotovoltaici per soddisfare le esigenze locali. Confrontare i consumi con la distribuzione degli impianti installati ci consente inoltre di valutare il grado di autosufficienza energetica di ciascuna zona e di proporre soluzioni di miglioramento mirate.

4.2 Cluster MongoDB in Docker

MongoDB è un sistema di gestione di database NoSQL progettato per funzionare in modo nativo su cluster, sfruttando al massimo i vantaggi offerti da un'architettura distribuita. Questa caratteristica si rivela particolarmente utile per progetti come il nostro, che prevedono la gestione di grandi volumi di dati. Tuttavia, poiché non avevamo accesso a un'infrastruttura hardware dedicata, abbiamo simulato un cluster utilizzando Docker. Questa scelta ci ha permesso di replicare un'architettura distribuita con tutti i vantaggi di scalabilità e resilienza, senza i costi e le complessità associate all'implementazione nativa su hardware fisico.

Abbiamo creato una configurazione Docker composta da:

- Replica set per i *server di configurazione*: Tre nodi configurati per gestire la metadata e coordinare la distribuzione dei dati tra gli shard. Ogni nodo è stato eseguito in un contenitore separato, connesso a una rete Docker dedicata.
- Replica set per gli *shard*: Tre shard, ognuno con tre nodi replicati per garantire ridondanza e affidabilità dei dati. Gli shard sono stati configurati per suddividere i dati su più nodi fisici simulati.
- Router (*mongos*): Un componente centrale per la gestione delle query e la connessione agli shard. È stato configurato per comunicare con i server di configurazione e coordinare le operazioni di lettura e scrittura.

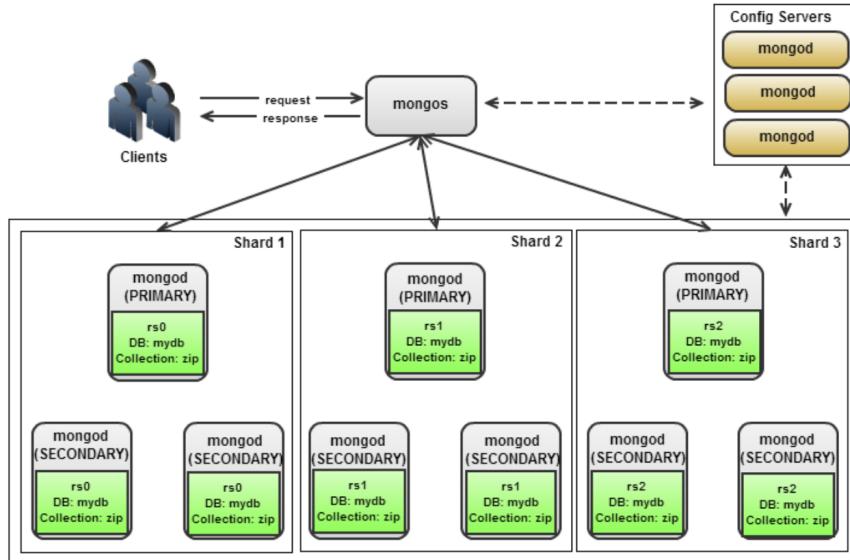


Figura 25: Struttura di un Cluster MongoDB

Ogni componente è stato definito utilizzando un *Docker Compose*, che ha facilitato la gestione dei contenitori e la simulazione dell'infrastruttura distribuita. I dettagli principali includono:

- *Port mapping*: Per ogni nodo, abbiamo mappato porte specifiche sulla macchina host per consentire il debug e l'accesso diretto.
- *Persistenza dei dati*: I volumi Docker sono stati utilizzati per salvare i dati in modo permanente, consentendo di preservare il contenuto tra i riavvii dei contenitori.
- *Rete dedicata*: Una rete Docker bridge è stata creata per garantire che tutti i contenitori potessero comunicare tra loro in modo isolato.

In ogni modo, la gestione di un cluster, seppur solo in simulazione, è stata un'esperienza molto formativa ed interessante!

4.2.1 Il File System XFS

Vista la grande mole di dati movimentati è stato fondamentale, per il corretto funzionamento del cluster, garantire che tutti i containers avessero accesso a volumi formattati col File System XFS. Quest'ultimo è un file system **journaled**: è una tecnica utilizzata da molti file system moderni per preservare l'integrità dei dati da eventuali cadute di tensione. Derivata dal mondo dei database, il journaling si basa infatti sul concetto di transazione dove ogni scrittura su disco è interpretata dal file system come tale.

Quando un applicativo invia dei dati al file system per memorizzarli su disco, questo prima memorizza le operazioni che intende fare su un file di log e in seguito provvede a effettuare le scritture sul disco rigido, quindi registra sul file di log le operazioni che sono state effettuate. In caso di caduta di tensione durante la scrittura del disco rigido, al riavvio del sistema operativo il file system non dovrà far altro che analizzare il file di log per determinare quali sono le operazioni che non sono state terminate e quindi sarà in grado di correggere gli errori presenti nella struttura del file system.

Poiché nel file di log vengono memorizzate solo le informazioni che riguardano la struttura del disco (metadati), un'eventuale caduta di tensione elimina i dati che si stavano salvando, ma non rende incoerente il file system.

XFS è stato sviluppato da Silicon Graphics nel 2005, è progettato per gestire file di grandi dimensioni (fino ad 8 exabyte!) e operazioni di I/O ad alta intensità, il che lo rende ideale per sistemi che devono archiviare e gestire grandi quantità di dati, come database o sistemi di archiviazione. È particolarmente utile in contesti in cui le dimensioni dei singoli file o dei volumi di dati sono considerevoli, come nel caso di MongoDB o altri sistemi che gestiscono file di grandi dimensioni.

Inoltre, è progettato per funzionare in ambienti multithreaded, ottimizzando l'uso delle risorse del sistema. MongoDB, che supporta operazioni parallele tramite replica e distribuzione dei dati, beneficia enormemente di questa caratteristica di XFS. Le operazioni di lettura/scrittura concorrenti su più thread vengono gestite in modo molto più efficiente rispetto a filesystem meno ottimizzati per il multithreading.

Infine, XFS supporta la creazione di *snapshot*, che consente di fare copie istantanee dei dati senza interruzioni. Questo è particolarmente utile in un contesto MongoDB, dove i backup frequenti e coerenti sono essenziali per la protezione dei dati. Gli snapshot

consentono di creare copie di backup mentre il database è ancora attivo e in esecuzione, senza compromettere l'accessibilità o la consistenza dei dati.

In sintesi, l'uso di XFS con MongoDB offre vantaggi significativi in termini di prestazioni, scalabilità, affidabilità e gestione dei dati, soprattutto in ambienti di produzione con grandi volumi di dati e carichi di lavoro complessi. L'alta efficienza di XFS nell'elaborazione di file di grandi dimensioni e nell'ottimizzazione delle operazioni parallele si combina perfettamente con le esigenze di MongoDB, creando un sistema robusto e ad alte prestazioni.

4.2.2 Scelta delle *"Shard Keys"* e della *"Chunk size"*

Per quanto concerne la gestione del cluster, gli aspetti meno tecnici e più rilevanti dal punto di vista ingegneristico con cui ci siamo confrontati sono stati: la scelta della chiave di *shard* e della dimensione dei *chunks*.

L'immagine sottostante mostra la configurazione del nostro cluster:

```
[direct: mongos] test> sh.status()
{
  shardingVersion: 1,
  _id: ObjectId('672b7de2350a14f13b3ffde6'),
  shards: [
    {
      id: 'shard-1 Replica Set',
      host: 'shard-1 Replica Set/shard-1-node-a:27019, shard-1-node-b:27019, shard-1-node-c:27019',
      state: 1,
      topologyTime: Timestamp({ t: 1730903560, i: 11 }),
      replSetConfigVersion: Long('1')
    },
    {
      id: 'shard-2 Replica Set',
      host: 'shard-2 Replica Set/shard-2-node-a:27019, shard-2-node-b:27019, shard-2-node-c:27019',
      state: 1,
      topologyTime: Timestamp({ t: 1730903561, i: 8 }),
      replSetConfigVersion: Long('1')
    },
    {
      id: 'shard-3 Replica Set',
      host: 'shard-3 Replica Set/shard-3-node-a:27019, shard-3-node-b:27019, shard-3-node-c:27019',
      state: 1,
      topologyTime: Timestamp({ t: 1730903566, i: 9 }),
      replSetConfigVersion: Long('1')
    }
  ],
  activeMongoses: 1,
  autosplit: true,
  balancer: {
    currentlyRunning: false,
    currentlyEnabled: true,
    failedBalancerRoundsInLast5Attempts: 0,
    migrationResultsForTheLast24Hours: 'No recent migrations'
  }
}
```

Figura 26: Stato dello *"Sharded Cluster"*

Tutta la conoscenza sulla gestione del cluster è stata appresa consultando un manuale dedicato; il lettore interessato ad approfondire l'argomento può consultare il seguente riferimento bibliografico [10].

La scelta della **Sharding Key** in MongoDB è una fase fondamentale per garantire prestazioni ottimali e una distribuzione bilanciata dei dati in un cluster shardato. Di seguito sono elencati i criteri principali da considerare e i motivi per cui ciascuno è importante:

- **Bilanciamento della distribuzione dei dati** La chiave di sharding deve garantire una distribuzione uniforme dei documenti tra gli shard. Ciò riduce il rischio che uno o più shard diventino sovraccarichi rispetto agli altri. Campi con **valori altamente variabili** (alta cardinalità) sono ideali, poiché permettono una suddivisione efficace.
- **Minimizzazione delle query distribuite ("scatter-gather")** I dati correlati che vengono frequentemente interrogati insieme dovrebbero trovarsi nello stesso shard. Scegliere una chiave che spesso appare nei filtri delle query consente alle richieste di essere indirizzate direttamente al singolo shard pertinente, migliorando significativamente la latenza.
- **Immutabilità dei valori della chiave** Una volta che un documento è inserito nel cluster, MongoDB non consente di modificare la chiave di sharding. Per questo motivo, è importante scegliere un campo i cui valori rimangano stabili per l'intero ciclo di vita del documento.
- **Scalabilità futura** La chiave di sharding deve supportare la crescita dei dati e l'aggiunta di nuovi shard nel tempo. Evitare chiavi con distribuzioni fortemente sbilanciate o che causano accessi concentrati su pochi shard (ad esempio, campi monotonicamente crescenti come `timestamp` o `ID incrementale`) è essenziale per mantenere un bilanciamento efficace.
- **Ottimizzazione delle query frequenti** Analizzare i modelli di accesso è cruciale. La chiave di sharding dovrebbe essere scelta tra i campi più utilizzati nei filtri delle query, consentendo al sistema di ottimizzare le operazioni con indirizzamenti precisi.
- **Cardinalità e unicità** La chiave ideale presenta un alto livello di variabilità, con valori unici o quasi unici per ciascun documento. Campi come `userId` o `orderId` sono spesso buone scelte, a condizione che i valori siano distribuiti in modo uniforme.
- **Chiavi composte** In alcuni scenari, una chiave composta, formata da più campi (ad esempio `{region, userId}`), può bilanciare distribuzione e ottimizzazione delle query. Questo approccio è utile quando la segmentazione geografica o funzionale dei dati è significativa.
- **Analisi e simulazione preliminare** È essenziale analizzare i dati esistenti per identificare campi con una buona distribuzione. Strumenti di aggregazione e simulazioni di carico possono aiutare a verificare come una chiave candidata influenza il bilanciamento e l'efficienza delle query.

Immaginiamo un'applicazione che gestisce utenti distribuiti globalmente. Una chiave di sharding basata su `userId` potrebbe garantire una distribuzione uniforme dei dati. In alternativa, per un'applicazione che segmenta i dati su base regionale, una chiave composta come `{region, itemId}` consente di ottimizzare sia la distribuzione che le query locali.

Scegliere con attenzione la chiave di sharding permette di migliorare significativamente le prestazioni del cluster MongoDB, ridurre i tempi di risposta delle query e garantire una scalabilità efficace. Una scelta sbagliata, invece, può portare a inefficienze difficili da correggere a posteriori.

```
[{"Immatricolazione.Nome Redditi": {
    shardKey: { $oggetto: 1, "Data di Nascita": 1 },
    unique: false,
    balanced: true,
    chunkMetadata: [
        { shard: "shard-1 replica-set", nchunks: 1 },
        { shard: "shard-2 replica-set", nchunks: 1 },
        { shard: "shard-3 replica-set", nchunks: 1 }
    ],
    chunks: [
        min: { $oggetto: MinKey(), "Data di Nascita": MinKey() }, max: { $oggetto: "D'ALESSIO MANOLO", "Data di Nascita": ISODate("1977-08-22T00:00:00.000Z") }, 'on shard': 'shard-1 replica-set', 'last modified': Timestamp({ t: 2, i: 0 }) },
        { min: { $oggetto: "D'ALESSIO MANOLO", "Data di Nascita": ISODate("1977-08-22T00:00:00.000Z") }, max: { $oggetto: "FUSCO ANNA MARIA", "Data di Nascita": ISODate("1964-04-06T00:00:00.000Z") }, 'on shard': 'shard-2 replica-set', 'last modified': Timestamp({ t: 3, i: 0 }) },
        { min: { $oggetto: "FUSCO ANNA MARIA", "Data di Nascita": ISODate("1964-04-06T00:00:00.000Z") }, max: { $oggetto: MaxKey(), "Data di Nascita": MaxKey() }, 'on shard': 'shard-3 replica-set', 'last modified': Timestamp({ t: 3, i: 1 }) }
    ]
}}
```

Figura 27: Esempio di chiave di Shard

Nel nostro progetto, abbiamo scelto di impostare la dimensione dei chunk a 64MB, che si è dimostrata essere la soluzione migliore, in grado di affrontare e risolvere alcune problematiche legate alla distribuzione dei dati nel cluster.

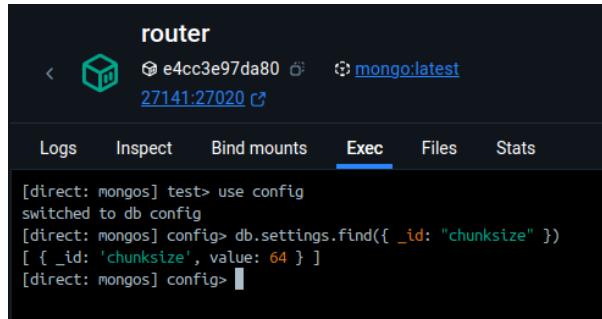


Figura 28: Dimensione dei Chunk

Questa scelta ha generato diversi vantaggi:

- *Bilanciamento dei dati tra gli shard*: garantisce una distribuzione equilibrata dei dati tra gli shard. Senza un bilanciamento adeguato, alcuni shard potrebbero accumulare troppi dati, portando a sovraccarichi e rallentamenti nelle operazioni. Con chunk di dimensioni più piccole, MongoDB può distribuire meglio i dati e migliorare l'efficienza complessiva del sistema.
- *Gestione efficace delle operazioni di split*: Impostando la dimensione dei chunk a 64 MB, siamo riusciti a ridurre il numero di operazioni di split e merge. Se i chunk fossero stati troppo piccoli, MongoDB avrebbe dovuto eseguire frequenti operazioni di split, aumentando il traffico di rete e i tempi di elaborazione.
- *Migliore prestazione nelle operazioni di lettura e scrittura*: La scelta ha contribuito a garantire una migliore gestione delle letture e scritture. Con chunk troppo grandi, si rischiava di rallentare le operazioni di lettura per via della quantità elevata di dati da elaborare in un singolo chunk. Con i chunk da 64 MB, le operazioni risultano più rapide e scalabili, senza compromettere le performance.

- *Scalabilità e prestazioni del cluster:* MongoDB bilancia automaticamente i chunk tra gli shard quando uno di essi diventa troppo grande, ma una dimensione di chunk appropriata riduce il rischio di sbilanciamento dei dati. L'impostazione a questo valore si è rivelata ideale per garantire una distribuzione dei dati efficiente e una scalabilità lineare del sistema, anche con l'aumentare del volume dei dati.

Considerazioni sul Bilanciamento

- **Chunk troppo piccoli:** Se i chunk sono troppo piccoli, potrebbero causare un numero elevato di operazioni di split e merge. Questo potrebbe aumentare il traffico di rete e il carico sui metadati, riducendo le prestazioni generali del cluster.
- **Chunk troppo grandi:** D'altra parte, chunk troppo grandi potrebbero provocare uno squilibrio tra gli shard, poiché un singolo shard potrebbe accumulare un carico eccessivo di dati, limitando la scalabilità e aumentando i tempi di risposta.

In sostanza, la scelta empirica di impostare la dimensione dei chunk a 64 MB ha permesso di affrontare le problematiche relative al bilanciamento dei dati, ottimizzando il flusso di informazioni tra gli shard e migliorando le prestazioni complessive del cluster MongoDB.

4.3 Strutturazione delle *Collections*

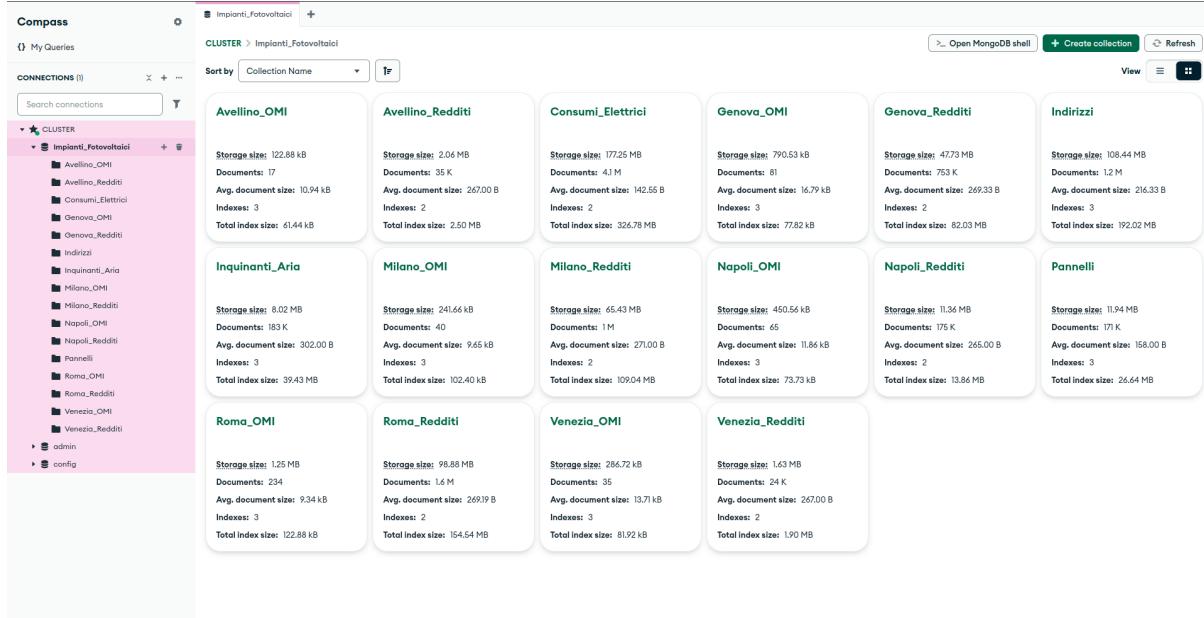


Figura 29: Visione complessiva del DB

L’immagine sopra riportata riassume la struttura globale del nostro database.

Abbiamo scelto, tramite l’uso di differenti collezioni, di separare i dati relativi a ciascuna delle sei città. Questa non è stata una mera scelta stilistica, ma una necessità funzionale. Infatti, considerando l’enorme numero di documenti, ad esempio, se avessimo centralizzato tutti i redditi in un’unica grande collezione, una semplice query per ottenere il reddito medio di Milano sarebbe stata notevolmente rallentata. Questo perché la query avrebbe dovuto eseguire un’operazione di *find* che avrebbe estratto solo i documenti relativi alla città di Milano, creando un carico di lavoro extra per il sistema.

Al contrario, per le collezioni relative ai *Consumi Elettrici* e quelle relative agli *Inquinanti Atmosferici*, abbiamo optato per una struttura più compatta. Essendo il numero di dati contenuti in queste collezioni relativamente ridotto, la fase di *find* non comporta un carico significativo per il sistema. Pertanto, in questi casi, abbiamo preferito un’unica collezione per tutte le città, dando priorità alla compattezza e alla facilità di accesso ai dati piuttosto che alla separazione semantica.

Di seguito, dettaglieremo il contenuto di ogni singola collezione e il modo in cui queste sono interconnesse, evidenziando l’approccio che abbiamo adottato per ottimizzare l’efficienza delle operazioni e la gestione dei dati all’interno del database.

4.3.1 I Redditi

Questa collezione rappresenta l'entità del mondo reale corrispondente agli abitanti.

```
_id: ObjectId('672c8fb99dc395b37ad8c8c4')
Soggetto : "ABATE ALFONSO"
Data di Nascita : 1977-10-04T00:00:00.000+00:00
Categoria di Reddito : "C"
Codice Attività : 4.33
Reddito Imponibile : 2
Imposta Netta : 582
Reddito d'Impresa / Lavoro Autonomo : 0
Volume d'Affari : 0
Tipo Modello : "730"
```

Figura 30: Document esemplificativo di *Redditi*

Il documento rappresenta un elemento all'interno della collezione **Redditi**, contenente informazioni relative a un soggetto, come dettagli fiscali, dati anagrafici e altre informazioni rilevanti per l'analisi e la gestione del reddito e delle imposte. I campi principali del documento includono:

- **Soggetto:** Nome del soggetto a cui si riferiscono i dati, che potrebbe trattarsi di un individuo, un'entità legale o un altro tipo di partecipante economico.
- **Data di Nascita:** La data di nascita del soggetto, formattata secondo lo standard ISO 8601, utile per l'identificazione anagrafica e per la categorizzazione in base all'età.
- **Categoria di Reddito:** Una classificazione che indica la tipologia di reddito dichiarato dal soggetto. Può rappresentare categorie fiscali, come lavoro dipendente, autonomo, pensione, ecc.
- **Codice Attività:** Un codice che rappresenta l'attività economica o professionale svolta dal soggetto, secondo un sistema di classificazione standard (ad esempio ATECO o altre normative fiscali).
- **Reddito Imponibile:** L'importo su cui vengono calcolate le imposte. Rappresenta il reddito soggetto a tassazione, al netto di eventuali deduzioni e esenzioni.
- **Imposta Netta:** L'importo complessivo delle imposte che il soggetto deve pagare, dopo aver applicato eventuali crediti fiscali o riduzioni. Indica l'effettivo onere fiscale del soggetto.
- **Reddito d'Impresa / Lavoro Autonomo:** Un campo che indica i redditi derivanti da attività d'impresa o lavoro autonomo. Può essere valorizzato o meno a seconda della situazione fiscale del soggetto.
- **Volume d'Affari:** Rappresenta l'ammontare complessivo delle transazioni commerciali o economiche effettuate dal soggetto, se applicabile. È un dato importante per la categorizzazione di soggetti imprenditoriali.

- **Tipo Modello:** Indica il tipo di modello fiscale o di dichiarazione utilizzato dal soggetto (ad esempio, modello 730, modello Unico, ecc.), se presente.

4.3.2 Le Zone OMI

Ogni documento di questa collezione è espressione di una Zona OMI, con annessa rappresentazione della geometria della stessa come Poligono.

```

_id: ObjectId('672ce4c07ba33fd6ec8f2eac')
geometry: Object
  type: "Polygon"
  coordinates: Array (1)
    0: Array (263)
properties: Object
  name: "AVELLINO - Zona OMI D8"
  description: Object
    @type: "html"
    value: "<table border='1'><tr><td><b>Cod. Naz. Comune</b></td><td>A509</td></tr>" styleUrl: "#style_160_0-0-153_0-0-0_1"
    fill-opacity: 0.6274509803921569
    fill: "#000099"
    stroke-opacity: 0.6274509803921569
    stroke: "#000000"
    stroke-width: 1
    LINKZONA: ""
    CODCOM: "A509"
    CODZONA: "D8"
  
```

Figura 31: Document esemplificativo di *OMI*

- **Geometry:** Un oggetto che descrive la geometria spaziale dell'area.
È un **Poligono** che rappresenta i confini della zona geografica, il che significa che l'area è definita da una serie di coordinate.
 - **type:** Il tipo di geometria (in questo caso, un "Poligono").
 - **coordinates:** Un array che contiene le coordinate geografiche del poligono. Queste coordinate definiscono i punti che delineano i confini dell'area, in un formato che può essere facilmente utilizzato da software GIS.
- **properties:** Un oggetto che contiene le proprietà associate alla zona geografica. Queste informazioni potrebbero includere metadati come nome, descrizione e altre caratteristiche rilevanti della zona.
 - **name:** Il nome della zona, in questo caso, "AVELLINO - Zona OMI D8".
 - **description:** Una descrizione della zona, formattata come HTML, che potrebbe contenere ulteriori dettagli sulla zona stessa.
 - * **@type:** Indica il tipo di contenuto della descrizione (ad esempio, HTML).
 - * **value:** Il valore della descrizione, che potrebbe essere un frammento HTML (es. una tabella con dettagli specifici sulla zona, come il codice nazionale del comune).
 - **styleUrl:** Riferimento a uno stile visivo per rappresentare questa zona (può essere utilizzato da un'applicazione GIS per applicare uno stile predefinito, come un colore o un pattern).

- **fill-opacity**: La trasparenza del riempimento della zona, indicata come una frazione tra 0 (completamente trasparente) e 1 (completamente opaco).
- **fill**: Il colore di riempimento della zona, espresso in formato esadecimale (in questo caso, #000099, che corrisponde a un blu).
- **stroke-opacity**: La trasparenza del contorno (bordi) del poligono.
- **stroke**: Il colore del contorno del poligono, espresso in formato esadecimale (in questo caso, #000000 per il nero).
- **stroke-width**: Lo spessore del contorno del poligono (in pixel).
- **LINKZONA**: Un campo che potrebbe contenere un URL o un riferimento a un'altra risorsa che descrive ulteriori dettagli sulla zona. Non è completamente definito nel frammento di dati fornito.
- **CODCOM**: Un codice identificativo per il comune, in questo caso, "A509", che è il codice nazionale del comune di Avellino.
- **CODZONA**: Probabilmente un altro codice che identifica in modo univoco la zona OMI, ma il valore non è completato nel frammento di dati fornito.

4.3.3 Gli Inquinanti Atmosferici

Questa collezione concretizza i dati degli inquinanti atmosferici rilevati in un anno solare dalle centraline ARPAC installate in taluni punti di ogni città. I dati sono stati estratti dalle banche dati pubbliche disponibili all'indirizzo link.

```

_id: ObjectId('67362a521959afae48fcda07')
▼ geometry : Object
  type : "Point"
  ▼ coordinates : Array (2)
    0: 9.231667
    1: 45.478347
▼ properties : Object
  Idstazione : 705
  Quota : 122
  Provincia : "MI"
  valore : 11.9
  Name : "Milano Pascal Città Studi"
  inquinante : "Ammoniaca"
  um : "µg/m³"
  data_ora : 2018-01-01T08:00:00.000+00:00
  DataStart : "2007-03-30T00:00:00"
  COMUNE : "Milano"

```

Figura 32: Document esemplificativo di *Inquinanti Aria*

- **geometry:** Un oggetto che descrive la geometria spaziale del punto in cui è ubicata la stazione di monitoraggio.
 - **type:** Il tipo di geometria, che in questo caso è "Point", indicando che la stazione di monitoraggio è rappresentata come un singolo punto geografico.
 - **coordinates:** Un array che contiene le coordinate geografiche del punto. Le coordinate sono espresse in longitudine e latitudine:
 - * 0: Longitudine (9.231667).
 - * 1: Latitudine (45.478347).
- **properties:** Un oggetto che contiene le proprietà associate alla stazione di monitoraggio, ovvero informazioni specifiche sulla stazione e i dati rilevati.
 - **Idstazione:** Identificativo univoco della stazione di monitoraggio, in questo caso "705".
 - **Quota:** L'altitudine della stazione in metri sopra il livello del mare, in questo caso "122".
 - **Provincia:** La provincia in cui si trova la stazione di monitoraggio, in questo caso "MI" (Milano).
 - **valore:** Il valore della concentrazione di inquinante misurata dalla stazione, in questo caso "11.9", che rappresenta la concentrazione dell'inquinante rilevato.
 - **Name:** Il nome della stazione di monitoraggio, in questo caso "Milano Pascal Città Studi".
 - **inquinante:** Il tipo di inquinante monitorato dalla stazione, in questo caso "Ammoniaca".
 - **um:** L'unità di misura della concentrazione dell'inquinante, che in questo caso è "ug/m³" (microgrammi per metro cubo).
 - **data_ora:** La data e l'ora in cui è stata effettuata la misurazione, espressa come timestamp ISO 8601 (2018-01-01T08:00:00.000+00:00).
 - **DataStart:** La data di inizio delle rilevazioni presso questa stazione, espressa anch'essa come timestamp ISO 8601 ("2007-03-30T00:00:00").
 - **COMUNE:** Probabile campo che contiene il nome del comune o altre informazioni geografiche relative alla stazione. Il valore non è stato completato nel frammento fornito.

4.3.4 I Consumi Elettrici

Questa collezione conserva i consumi giornalieri, divisi in fasce orarie, di ogni Zona OMI. Tali dati sono stati ottenuti interrogando le banche dati pubbliche messe a disposizione dalla società statale che si occupa della distribuzione della rete elettrica su tutto il territorio nazionale: *Terna*.

```
_id: ObjectId('673b600d8c9d2b997ffe6a89')
codzona : "D1"
città : "Avellino"
consumo_energetico : 1.99574
Unità Misura : "MWh"
data_ora : 2018-01-01T00:00:00.000+00:00
OMI : ObjectId('672ce4c07ba33fd6ec8f2eae')
```

Figura 33: Document esemplificativo di *Consumi Elettrici*

- **codzona:** Codice identificativo della zona OMI (Osservatorio del Mercato Immobiliare) in cui è stato misurato il consumo energetico. Il codice è "D1".
- **città:** Nome della città a cui appartiene la zona specificata. In questo caso, è "Avellino".
- **consumo_energetico:** Valore numerico che rappresenta il consumo energetico totale registrato nella zona. Il consumo è di 1.99574 MWh.
- **Unità Misura:** L'unità di misura per il valore del consumo energetico. Qui è "MWh" (megawattora).
- **data_ora:** Data e ora della misurazione del consumo energetico, espresse nel formato ISO 8601. Il valore è "2018-01-01T00:00:00.000+00:00", che indica che la misurazione è avvenuta il primo gennaio 2018 a mezzanotte.
- **OMI:** Collegamento a un altro documento nella collezione relativa alle zone OMI. Questo campo contiene un riferimento a un altro oggetto con l'`ObjectId` che ha la associata Zona OMI nella relativa collezione referenziata. Questo legame permette di accedere ai dettagli completi della zona OMI a cui si riferisce il consumo energetico.

4.3.5 Le informazioni anagrafiche dei *Nuclei Familiari*

Questa collezione conserva le informazioni della residenza e della rispettiva composizione dei nuclei familiari; partendo dai dati relativi ai *Redditii*, tramite uno script Python abbiamo ricomposto le relative composizioni dei nuclei in essere al 31/12/2005.

```

_id: ObjectId('67334e6ee02f8025b7b9dab9')
  ▼ Soggetto : Array (4)
    0: ObjectId('672c97fd9dc395b37aec8394')
    1: ObjectId('672c98509dc395b37aee43ed')
    2: ObjectId('672c9c7b9dc395b37a03da9d')
    3: ObjectId('672c9ad39dc395b37afb579d')
  Indirizzo : "Sentiero uliveto"
  Civico : 31
  Città : "Roma"
  ▼ location : Object
    type : "Point"
    ▼ coordinates : Array (2)
      0: 12.538325
      1: 41.8422703

```

Figura 34: Document esemplificativo di *Indirizzi*

- **Soggetto:** Una lista di riferimenti ad altri documenti, rappresentati dagli `ObjectId`. Ogni ID corrisponde ad un soggetto che appartiene allo stesso nucleo familiare e risiede:
- **Indirizzo:** Nome della via o percorso, specificato come "Sentiero uliveto".
- **Civico:** Numero civico dell'indirizzo, in questo caso 31.
- **Città:** Nome della città a cui appartiene l'indirizzo. In questo caso, è "Roma".
- **location:** Oggetto che rappresenta la posizione geografica dell'indirizzo:
 - **type:** Tipo di geometria, in questo caso "Point", che indica un punto specifico.
 - **coordinates:** Una lista di coordinate geografiche (*longitudine*, *latitudine*):
 - * Longitudine
 - * Latitudine

4.3.6 I Pannelli

Questa collezione, nonostante la semplicità della sua struttura documentale, rappresenta il nucleo fondamentale del progetto. Grazie ad essa, è stato possibile mettere in relazione i dati geografici dei pannelli solari rilevati dalla rete neurale con informazioni anagrafiche e di proprietà. A partire dalle coordinate geografiche generate dal modello, abbiamo sviluppato uno script in Python che utilizzava le API di OpenStreetMap per ottenere l'indirizzo completo corrispondente a ciascun punto individuato.

Questo passaggio si è rivelato cruciale, poiché ha permesso di trasformare semplici coordinate in informazioni comprensibili e utilizzabili nel contesto del progetto.

Con l'indirizzo ricavato, è stato poi possibile incrociare tali dati con le informazioni delle residenze anagrafiche contenute in altre collezioni del database. Questo ha consentito di identificare i proprietari degli impianti fotovoltaici rilevati, fornendo uno strumento potente per analisi, studi di distribuzione territoriale, o, possibili applicazioni amministrative.

```
_id: ObjectId('6745e0d5c8f82b1ef0dad455')
Indirizzo : "Via della Meccanica"
Civico : 12
Città : "Venezia"
▼ location : Object
  type : "Point"
  ▼ coordinates : Array (2)
    0: 12.2125064
    1: 45.4373791
```

Figura 35: Document esemplificativo di *Pannelli*

La semplicità di questa collezione, apparentemente modesta, cela in realtà la complessità e la rilevanza delle operazioni che ha abilitato, confermandone il ruolo cruciale come cuore operativo dell'intero progetto.

4.4 Relazioni tra le collezioni

In MongoDB, le collezioni possono essere collegate tramite due approcci principali: *embedding* (dati incorporati) e *riferimenti esterni*. Nel nostro progetto, a causa della complessità e della natura relazionale dei dati, abbiamo scelto di utilizzare **riferimenti esterni**. Questa decisione ci ha consentito di mantenere una struttura più modulare e scalabile, evitando la duplicazione dei dati e facilitando la manutenzione del database.

Motivo della Scelta

L'uso di riferimenti esterni è stato dettato dalla necessità di gestire un'elevata quantità di dati strutturati, in cui le relazioni tra entità giocano un ruolo centrale. Incorporare direttamente i dati (*embedding*) avrebbe potuto complicare le operazioni di aggiornamento e sovraccaricare la memoria del database, soprattutto nel caso di relazioni uno-a-molti o molti-a-molti.

Relazioni Implementate

- **Soggetti e Indirizzi:** Abbiamo collegato i documenti relativi ai soggetti (persone fisiche) agli indirizzi attraverso riferimenti esterni. Questa relazione ci ha permesso di ricostruire con precisione i nuclei familiari e le loro rispettive residenze. L'associazione è stata utilizzata anche per correlare i dati demografici ai consumi energetici e agli impianti fotovoltaici, rendendo possibili analisi di tipo socio-economico.
- **Consumi e Zone OMI:** I dati sui consumi energetici sono stati associati alle relative zone OMI (Osservatorio del Mercato Immobiliare). Questa relazione ha facilitato l'analisi georeferenziata dei consumi, consentendo di valutare variazioni e trend energetici in base alla posizione geografica e alle caratteristiche delle aree urbane.

Benefici delle Relazioni Esterne

1. **Scalabilità:** Separando i dati in collezioni correlate, abbiamo garantito una struttura flessibile, adatta a crescere con l'aumento del volume di dati.
2. **Manutenzione Facilitata:** L'aggiornamento delle informazioni in una collezione non ha richiesto modifiche ridondanti in altre collezioni, semplificando la gestione del database.
3. **Prestazioni Ottimizzate:** Le query, seppur più complesse a causa dell'uso dei riferimenti, sono state progettate per ottenere risultati mirati senza sovraccaricare il sistema.
4. **Modularità:** La separazione logica dei dati in collezioni distinte ha permesso di gestire in modo più ordinato i vari tipi di informazioni, mantenendo una chiara struttura semantica.

4.5 Schemi di Validazione

Nel nostro progetto, abbiamo creato 5 schemi di validazione per le principali collezioni di dati presenti nel database. Questi schemi, definiti tramite **JSON Schema**, sono stati progettati per garantire l'integrità dei dati e evitare errori o incoerenze durante l'inserimento e l'aggiornamento dei record. A ciascun schema sono stati assegnati campi obbligatori e specificati i tipi di dato e le restrizioni, al fine di mantenere una struttura coerente e valida. Ecco una panoramica di ciascun schema:

4.5.1 Schema per i *Consumi Energetici*

La collezione dei *Consumi Energetici* raccoglie dati relativi ai consumi energetici nelle varie zone OMI. Lo schema include:

- **Campi obbligatori:** _id, ID, codzona, città, consumo_energetico, Unità Misura, data_ora.
- **Validazione del tipo di dato:**
 - consumo_energetico deve essere un numero (intero o decimale), maggiore o uguale a zero.
 - Unità Misura deve essere una delle seguenti: "MWh", "kWh", "GWh".
 - data_ora deve essere un timestamp ISO 8601 valido.
- **Validazioni sui valori:**
 - Il campo città è limitato a un insieme specifico di città (Avellino, Napoli, Roma, Milano, Genova, Venezia).
 - Il campo codzona è una stringa alfanumerica.

4.5.2 Schema per gli *Indirizzi*

La collezione degli *Indirizzi* è strutturata per raccogliere informazioni precise su ogni località, inclusa la geolocalizzazione. Le validazioni includono:

- **Campi obbligatori:** Soggetto, Indirizzo, Civico, Città, location.
- **Validazione del tipo di dato:**
 - Soggetto è un array di almeno un identificatore `ObjectId`, rappresentante i soggetti associati.
 - Indirizzo è una stringa non vuota che rappresenta l'indirizzo fisico.
 - Civico è un numero intero che rappresenta il numero civico.
 - location deve essere un oggetto con proprietà `type` (che deve essere "Point") e `coordinates` (un array di due valori numerici rappresentanti latitudine e longitudine).
- Il campo Città deve essere una delle città definite (Avellino, Napoli, Roma, Milano, Genova, Venezia).

4.5.3 Schema per gli *Inquinanti Aria*

Lo schema per la collezione degli *Inquinanti Aria* è simile a quello degli indirizzi, ma con validazioni aggiuntive per i dati legati agli inquinanti atmosferici. I principali dettagli includono:

- **Campi obbligatori:** Soggetto, Indirizzo, Civico, Città, location.
- **Validazione del tipo di dato:**
 - Soggetto deve essere un array di identificatori `ObjectId` che collegano l'inquinante a uno o più soggetti.
 - Indirizzo e Civico sono validati come stringa e intero, rispettivamente.
 - location è un oggetto con coordinate geografiche (longitudine e latitudine) e tipo `Point`.
- Città è un valore che deve appartenere a un insieme predefinito (Avellino, Napoli, Roma, Milano, Genova, Venezia).

4.5.4 Schema per i *Pannelli*

Lo schema per la collezione dei *Pannelli* include:

- **Campi obbligatori:** Soggetto, Indirizzo, Civico, Città, location.
- **Validazione del tipo di dato:**
 - Soggetto deve essere un array contenente almeno un `ObjectId` per identificare i soggetti associati ai pannelli.
 - Indirizzo e Civico sono validati come stringa e intero, rispettivamente.
 - location è un oggetto che definisce la posizione geografica come un punto, con `coordinates` come array di latitudine e longitudine.
- Città è una stringa che deve essere uno dei valori predefiniti (Avellino, Napoli, Roma, Milano, Genova, Venezia).

4.5.5 Schema per i *Redditi*

La collezione dei *Redditi* si occupa di registrare i dati relativi ai redditi dei soggetti, con i seguenti dettagli:

- **Campi obbligatori:** Soggetto, Indirizzo, Civico, Città, location.
- **Validazione del tipo di dato:**
 - Soggetto deve essere un array contenente almeno un identificatore `ObjectId`.
 - Indirizzo e Civico devono rispettare i formati stringa e intero, rispettivamente.
 - location deve contenere informazioni geografiche valide (tipo `Point` e coordinate).
- Città deve essere uno dei valori prestabiliti (Avellino, Napoli, Roma, Milano, Genova, Venezia).

Gli schemi di validazione implementati per ciascuna delle cinque principali collezioni (consumi, indirizzi, inquinanti, pannelli e redditi) sono strumenti fondamentali per garantire la qualità e l'integrità dei dati nel nostro progetto.

Le validazioni hanno lo scopo di:

- **Garantire l'integrità dei dati:** Solo i dati che soddisfano le condizioni predefinite (ad esempio, valori numerici positivi, date in formato corretto, città valide) vengono accettati.
- **Migliorare l'affidabilità del sistema:** Riducendo il rischio di errori durante l'inserimento o la modifica dei dati, specialmente quando più fonti esterne sono coinvolte.
- **Ottimizzare l'analisi e il processamento dei dati:** Poiché ogni collezione è garantita da un formato coerente, le operazioni sui dati diventano più efficienti e affidabili.

Per quanto riguarda i dati delle Zone OMI, non è stato necessario implementare uno schema di validazione, in quanto i dati sono stati estratti direttamente dal sito ufficiale dell'*Agenzia delle Entrate*. Poiché il sito dell'AdE fornisce dati ufficiali e aggiornati, possiamo essere certi della loro integrità e correttezza, senza necessità di ulteriori verifiche o validazioni. In altre parole, i dati relativi alle zone OMI sono considerati "affidabili al 100%" in quanto provengono da una fonte ufficiale e sono già correttamente strutturati. Pertanto, non è stato necessario eseguire ulteriori operazioni di validazione come per le altre collezioni del progetto.

4.6 Scelta degli Indici

Nel progettare la struttura del database e la gestione delle query future, è stato fondamentale definire gli indici più appropriati per ottimizzare le prestazioni.

La scelta di dove applicare gli indici dipendeva dalle **tipologie di query** che si prevedeva di eseguire più frequentemente.

Chiavi di Shard su campi d'interesse

Le chiavi di shard sono state scelte tenendo conto non solo della distribuzione dei dati, ma anche dei campi che avrebbero probabilmente interessato le query future. Pertanto, oltre ad essere utilizzati come chiavi di shard, questi stessi campi sono stati indicizzati per ottimizzare le performance delle query. In particolare, abbiamo scelto di shardare le collezioni in modo da ridurre al minimo i dati da trasferire tra i nodi durante l'esecuzione di query complesse. Questa strategia ha permesso di avere una struttura coerente per le operazioni di ricerca e distribuzione dei dati, riducendo il tempo necessario per l'esecuzione di query complesse. Inoltre, la scelta di usare gli stessi campi sia come chiavi di shard che come indici ha contribuito a garantire che le operazioni di lettura e scrittura sui dati fossero efficienti, evitando costosi passaggi di riorganizzazione o ribilanciamento del cluster.

Indici "2dsphere" per le Coordinate

Un altro aspetto importante riguardava l'indicizzazione delle **coordinate geografiche**. Poiché molte delle query avrebbero coinvolto la ricerca e la gestione di posizioni geografiche, abbiamo scelto di creare indici 2dsphere per tutte le collezioni che contenevano coordinate geografiche. Tali indici sono ottimizzati per eseguire query su dati geografici, permettendo di eseguire operazioni come la ricerca di punti all'interno di una determinata area geografica o la ricerca di punti più vicini a una determinata posizione. Utilizzando questi indici, è stato possibile garantire che le query spaziali venissero eseguite in modo rapido ed efficiente.

In sintesi, questi indici sono essenziali per ridurre il tempo di risposta delle query, in quanto consentono a MongoDB di localizzare rapidamente i documenti pertinenti senza dover eseguire una scansione completa della collezione. La scelta degli indici è stata fatta con l'obiettivo di ottimizzare l'esecuzione delle query più comuni, ridurre il carico sul sistema e migliorare le performance complessive del database.

4.7 L'API "*PyMongo*"

Nel nostro progetto, abbiamo utilizzato l'API **PyMongo** ^[11], un driver Python nativo per MongoDB, che si è rivelata una scelta strategica per gestire e interrogare il nostro database. Questa decisione è stata motivata dalla maggiore flessibilità e semplicità offerta da PyMongo rispetto all'utilizzo diretto dell'interfaccia Compass, il tool grafico ufficiale per MongoDB.

Vantaggi dell'uso di PyMongo

1. Automazione delle Query:

Con PyMongo è stato possibile scrivere script Python per automatizzare l'esecuzione di query complesse e ripetitive. Ciò ha eliminato la necessità di eseguire manualmente ogni query tramite Compass, risparmiando tempo e riducendo il rischio di errori umani.

2. Flessibilità Operativa:

L'integrazione di PyMongo con il nostro codice ci ha consentito di gestire dinamicamente le operazioni sul database, come l'inserimento, l'aggiornamento e l'eliminazione dei documenti, direttamente dal flusso logico delle applicazioni Python.

3. Supporto per Query Avanzate:

PyMongo offre il pieno supporto per le funzionalità avanzate di MongoDB, come l'aggregazione, la gestione di indici e la pipeline di elaborazione dei dati. Questo ci ha permesso di sfruttare appieno le potenzialità del database senza dover ricorrere a strumenti aggiuntivi.

4. Gestione del Workflow di Progetto:

Utilizzare PyMongo ha facilitato l'integrazione delle query con il resto delle operazioni previste nel progetto, come il preprocessing dei dati, le analisi statistiche e la visualizzazione dei risultati, creando un workflow fluido e ben integrato.

Perché non usare *Compass*?

Compass è uno strumento eccellente per la gestione manuale dei dati e per le analisi preliminari del database. Tuttavia, per un progetto come il nostro, che richiedeva una gestione programmata e ripetitiva di grandi volumi di dati, PyMongo si è dimostrato indispensabile. La possibilità di lavorare direttamente dal codice ha reso il processo non solo più veloce, ma anche completamente automatizzato e riproducibile.

Perché non usare *Motor*?

Abbiamo scelto PyMongo per la sua semplicità, stabilità e compatibilità con il nostro flusso di lavoro sincrono. Non avendo esigenze specifiche per la gestione asincrona delle operazioni, l'adozione di PyMongo ha reso il nostro sviluppo più efficiente, riducendo la complessità del progetto. Motor sarebbe stato utile se avessimo avuto un carico di lavoro I/O molto pesante o se avessimo dovuto gestire un gran numero di operazioni simultanee, ma nel nostro caso, PyMongo ha soddisfatto perfettamente le necessità del progetto.

4.8 Le Query

Di seguito presentiamo le query che abbiamo ideato per sfruttare al massimo le potenzialità offerte da MongoDB in termini di interrogazioni geospatiali, le quali permettono di gestire e analizzare i dati geografici in modo molto efficiente. In particolare, ci siamo concentrati sull'uso degli indici geospatiali, come gli indici *2dsphere*, che consentono di eseguire operazioni complesse su coordinate geografiche, come la ricerca di punti vicini, la misurazione delle distanze e l'interrogazione di aree specifiche.

Queste tecniche sono fondamentali per rispondere a domande e analisi basate sulla posizione, come la localizzazione di impianti fotovoltaici in relazione a determinati indirizzi o la valutazione di zone con particolare densità di impianti.

Inoltre, le query presentate evidenziano come gli indici ottimizzano significativamente le prestazioni, riducendo i tempi di risposta e migliorando l'efficienza del database, soprattutto quando si lavora con grandi volumi di dati geospatiali.

4.8.1 QUERY 1: Reddito medio per città

```

Stage 1 $group
1 { 
2   _id: null,
3   media_reddito: {
4     $avg: "$Reddito Imponibile"
5   }
6 }

Output after $group stage (Sample of 1 document)
{
  "_id": null,
  "media_reddito": 2558.61396
}

```

Figura 36: QUERY 1: Struttura

Obiettivo

Questa query ha l'obiettivo di ottenere una media globale del reddito imponibile per tutti i soggetti presenti nella collezione *Roma_Reddi*. Non vengono utilizzati gruppi specifici (ad esempio, per città, anno o altre caratteristiche) in quanto l'intento è calcolare la media complessiva di tutti i dati nella collezione.

Descrizione della query

- **Operazione \$group:** L'operazione *\$group* in MongoDB è utilizzata per raggruppare i documenti della collezione in base a un campo specifico. In questo caso, non viene specificata una chiave di raggruppamento, quindi *_id* è impostato su *null*. Questo significa che non ci sarà un vero e proprio raggruppamento per chiavi specifiche, ma verranno trattati tutti i documenti come un singolo gruppo. L'obiettivo, in questo caso, è calcolare un valore aggregato globale (la media).
- **_id: null:** Utilizzando *null* come valore per *_id*, la query calcolerà un'aggregazione su tutti i documenti della collezione. Non c'è una divisione per chiavi specifiche,

quindi otteniamo un solo documento di output con la media di tutti i valori del campo richiesto.

- **\$avg: "\$Reddito Imponibile"**: La funzione di aggregazione \$avg calcola la media di un campo numerico. In questo caso, la query calcola la media del campo **Reddito Imponibile** per tutti i documenti nella collezione. Questo significa che la query prende tutti i valori del campo **Reddito Imponibile** dei documenti presenti nella collezione **Roma_Reddit**, li somma e li divide per il numero totale di documenti, ottenendo così il valore medio di quel campo. Il risultato finale sarà un singolo documento contenente il campo **media_reddito**, che rappresenta la media di tutti i valori di **Reddito Imponibile**.

Il risultato finale sarà un singolo documento contenente il campo **media_reddito**, che rappresenta la media di tutti i valori di **Reddito Imponibile**.

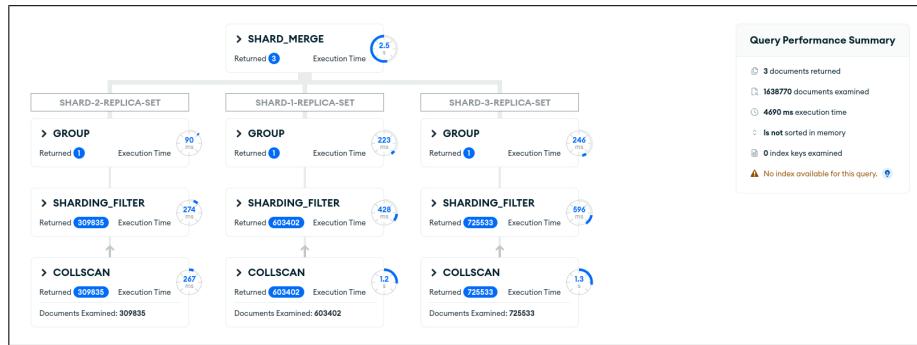


Figura 37: QUERY 1: Esecuzione senza indice

Di seguito è possibile osservare i benefici derivanti dalla creazione di un indice sul campo **Reddito Imponibile**.

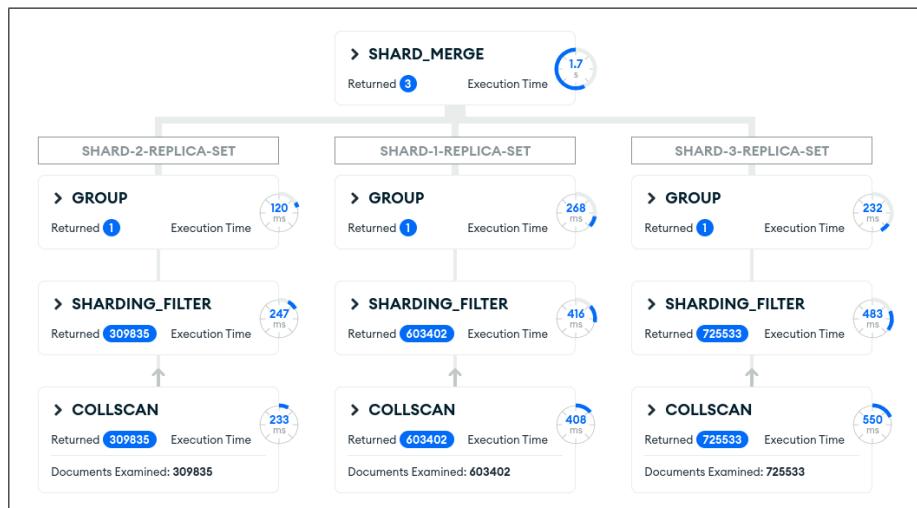


Figura 38: QUERY 1: Esecuzione con indice

4.8.2 QUERY 2: Tasso di pannelli per Zona OMI

```

46 # Iteriamo attraverso tutti i pannelli
47 for pannello in pannelli:
48     # Controlliamo in quale zona OMI si trova il pannello
49     omi_zones = db.Roma_OMI.find({
50         "geometry": {
51             "$geoIntersects": {
52                 "$geometry": pannello["location"]
53             }
54         }
55     })
56
57
58     # Per ogni zona OMI trovata, incrementiamo il conteggio dei pannelli in quella zona
59     for omi_zone in omi_zones:
60         zona_name = omi_zone["properties"]["name"]
61         if zona_name not in zona_pannelli:
62             zona_pannelli[zona_name] = 0
63             zona_pannelli[zona_name] += 1
64

```

Figura 39: QUERY 2: Struttura

Obiettivo

Questa query ha l'obiettivo di determinare il numero di pannelli fotovoltaici presenti in ogni zona OMI della città di Roma. I pannelli sono mappati rispetto alle zone utilizzando un'interrogazione geospaziale basata sulla relazione `$geoIntersects`, che verifica quali geometrie (i pannelli) si trovano all'interno delle geometrie delle zone OMI definite nel database. Questo tipo di analisi è utile per comprendere la distribuzione degli impianti nelle diverse aree territoriali.

Descrizione della Query

- Filtraggio iniziale:** La funzione `find` viene utilizzata per estrarre tutti i pannelli fotovoltaici appartenenti alla città di **Roma** dalla collezione **Pannelli**. Questa operazione seleziona i documenti con il campo **Città** impostato su "Avellino". È un filtro iniziale necessario per restringere il campo di analisi.
- Interrogazione geospaziale:** Per ogni pannello estratto, la query verifica in quale zona OMI si trova, sfruttando l'operatore `$geoIntersects`. Questo operatore confronta la posizione geografica del pannello (definita dal campo `location`) con le geometrie delle zone OMI memorizzate nella collezione **Roma_OMI**. Se il pannello si trova all'interno di una zona, vengono restituiti i dati relativi a quella zona.
- Conteggio dei pannelli per zona:** Una volta individuata la zona OMI a cui appartiene un pannello, viene incrementato un contatore in un dizionario Python chiamato `zona_pannelli`. Ogni chiave del dizionario rappresenta il nome di una zona OMI, mentre il valore associato indica il numero totale di pannelli presenti in quella zona.
- Esportazione dei risultati:** I risultati vengono salvati in un file CSV per facilitare le analisi successive. Ogni riga del file CSV contiene il nome della zona OMI e il relativo numero di pannelli fotovoltaici. Inoltre, i dettagli di `explain` (che forniscono informazioni sull'ottimizzazione della query) vengono salvati in un file JSON per monitorare le performance e verificare che gli indici vengano utilizzati correttamente.

Il risultato della query è un elenco di zone OMI con il rispettivo numero di pannelli fotovoltaici presenti in ciascuna zona. Questo permette di analizzare la densità e la distribuzione dei pannelli all'interno delle diverse aree della città, fornendo una visione chiara e dettagliata della diffusione degli impianti fotovoltaici a Roma.

Tale elenco viene successivamente usato come input per uno script Python che genera una rappresentazione visiva del risultato atteso.

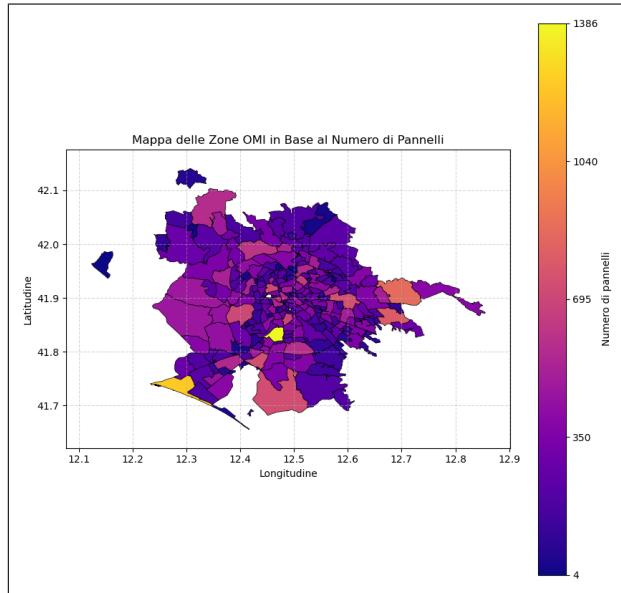


Figura 40: QUERY 2: Risultato

Perchè manca l'explain di Compass?

L'*explain* di Compass non è disponibile per questa query poiché l'operatore `$geoIntersects` non può essere utilizzato direttamente all'interno di una pipeline di aggregazione in MongoDB. Questo vincolo impedisce di generare un *execution plan* completo tramite gli strumenti integrati di Compass per l'analisi delle performance.

Per aggirare questa limitazione, la logica dell'interrogazione geospaziale è stata implementata in Python. Ogni documento della collezione dei pannelli viene processato singolarmente in modo iterativo: ogni pannello è confrontato con le zone OMI tramite query individuali basate sull'operatore `$geoIntersects`. Questa soluzione consente di ottenere il risultato desiderato, ma a discapito della possibilità di analizzare direttamente il piano di esecuzione tramite MongoDB.

Totale documenti esaminati: 113775
Totale tempo di esecuzione (ms): 25754

Figura 41: QUERY 2: Esecuzione

4.8.3 QUERY 3: Consumi per Zona OMI

```

1  [
2   {
3     $match: {
4       città: "Roma"
5     }
6   },
7   {
8     $lookup: {
9       from: "Roma_OMI",
10      localField: "OMI",
11      foreignField: "_id",
12      as: "zona_info"
13    }
14  },
15  {
16    $unwind: "$zona_info"
17  },
18  {
19    $group: {
20      _id: "zona_info.properties.name",
21      media_consumo: {
22        $avg: "$consumo_energetico"
23      }
24    }
25  },
26  {
27    $project: {
28      _id: 0,
29      name: "$_id",
30      media_consumo: 1
31    }
32  }
33 ]
34 .

```

Figura 42: QUERY 3: Struttura

Obiettivo

La query ha l'obiettivo di calcolare la media del consumo energetico per ciascuna zona OMI (Osservatorio del Mercato Immobiliare) della città di Roma, raggruppando i dati per zona e associandoli al consumo medio registrato nella collezione dei consumi energetici.

Descrizione della Query

- Filtraggio iniziale (\$match):** Il primo stadio della query seleziona soltanto i documenti della collezione `Consumi_Elettrici` che si riferiscono alla città di Roma, limitando l'elaborazione ai dati pertinenti.
- Join tra collezioni (\$lookup):** Effettua un join con la collezione `Avellino_OMI` (che rappresenta le zone OMI). Il campo `OMI` della collezione `Consumi_Elettrici` viene confrontato con il campo `_id` della collezione `Avellino_OMI` per associare ogni consumo energetico alla relativa zona OMI. Il risultato del join è un array di documenti OMI che corrispondono a ciascun consumo energetico.
- Decomposizione del risultato (\$unwind):** Poiché il join restituisce un array per ciascun documento, l'operazione `$unwind` scomponete l'array in documenti singoli. Ogni documento rappresenta un consumo energetico associato a una zona OMI specifica.
- Raggruppamento per zona (\$group):** La query raggruppa i documenti per il nome della zona OMI (`zona_info.properties.name`), utilizzando questo campo come chiave di raggruppamento. Durante il raggruppamento, viene calcolata la media del campo `consumo_energetico` per ciascuna zona.
- Proiezione dei risultati (\$project):** Viene creata una struttura finale dei documenti di output, dove il nome della zona OMI (`_id`) viene rinominato in `name` e la media calcolata del consumo energetico (`media_consumo`) viene mantenuta.

Il risultato della query sarà una lista di documenti, ognuno contenente:

- **name**: il nome di una zona OMI di Roma.
- **media_consumo**: la media dei consumi energetici (in MWh) per quella zona.

Tale risultato viene successivamente usato come input per uno script Python che genera una rappresentazione visiva del risultato atteso.

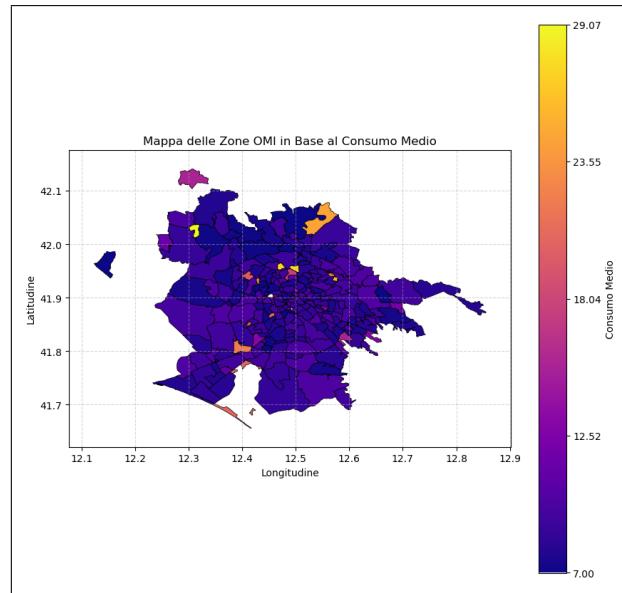


Figura 43: QUERY 3: Risultato

Come mostra l'immagine sottostante, la query a 5 stage è molto complessa e vista la grande mole di documenti da processare impiega poco più di 25 min.
La stessa, senza indici sui campi di ricerca, ne impiegava 45 !!!

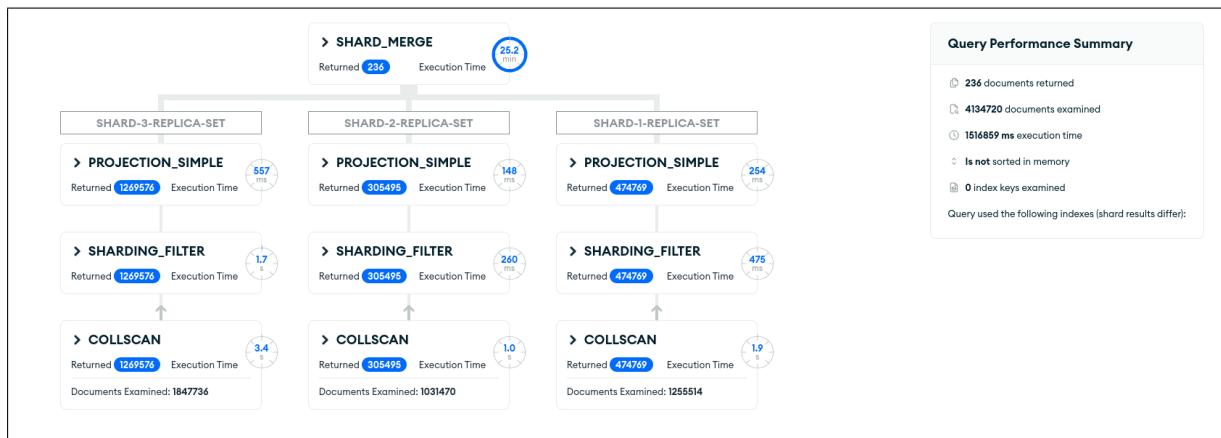


Figura 44: QUERY 3: Esecuzione

4.8.4 QUERY 4: Inquinanti per Zona OMI

```

# Iteriamo attraverso tutti gli inquinanti
for inquinante in inquinanti:
    # Controlliamo in quale zona OMI si trova l'inquinante
    omi_zones = db.Avellino_OMI.find({
        "geometry": {
            "$geoIntersects": {
                "$geometry": inquinante["geometry"]
            }
        }
    })

    # Per ogni zona OMI trovata, aggiungiamo il valore dell'inquinante (e sommiamo i valori)
    for omi_zone in omi_zones:
        zona_name = omi_zone["properties"]["name"]
        if zona_name not in zona_inquinanti:
            zona_inquinanti[zona_name] = {} # Dizionario per contenere gli inquinanti per zona

        # Ottieni il valore dell'inquinante e aggiorna il dizionario
        inquinante_name = inquinante["properties"]["inquinante"]
        valore_inquinante = inquinante["properties"].get("valore", 0)

        if inquinante_name not in zona_inquinanti[zona_name]:
            zona_inquinanti[zona_name][inquinante_name] = []

        zona_inquinanti[zona_name][inquinante_name].append(valore_inquinante)

```

Figura 45: QUERY 4: Struttura

Obiettivo

La query ha l'obiettivo di recuperare i dati relativi agli inquinanti atmosferici presenti nella città di Avellino, associandoli alle zone OMI (Osservatorio del Mercato Immobiliare). Inoltre, calcola la media dei valori degli inquinanti per ciascuna zona OMI, raccogliendo anche informazioni sulle performance della query, come il tempo di esecuzione e il numero di documenti esaminati, tramite l'operazione `explain`.

Descrizione della Query

- **Filtraggio dei documenti inquinanti per la città di Roma:** La query inizia estraendo tutti i documenti dalla collezione `Inquinanti_Aria` che appartengono alla città di Roma. Questo filtro avviene utilizzando il campo `properties.COMUNE`, che contiene il nome del comune, per selezionare i documenti con valore "Roma".
- **Intersezione geografica con le zone OMI:** Per ogni inquinante estratto, la query esegue una ricerca nella collezione `Avellino_OMI` per determinare le zone OMI che intersecano la geometria dell'inquinante. L'operatore `$geoIntersects` viene utilizzato per verificare in quale zona OMI si trova ogni inquinante, confrontando le geometrie.
- **Uso dell'operazione explain:** Per analizzare le performance della query, viene eseguito il comando `explain`. Questo fornisce informazioni dettagliate sul piano di esecuzione della query, come il numero di documenti esaminati e il tempo di

esecuzione. Tali informazioni sono utili per monitorare l'efficienza della query e garantire che gli indici siano utilizzati correttamente.

- **Calcolo delle medie degli inquinanti per zona:** La query raccoglie i valori degli inquinanti per ciascuna zona OMI e calcola la media dei valori per ogni tipo di inquinante. Questo calcolo avviene sommando i valori degli inquinanti e dividendoli per il numero totale di valori presenti per ciascuna zona.
- **Salvataggio dei risultati:** I risultati della query, che includono la zona OMI, l'inquinante e la media dei valori per ciascun inquinante, vengono esportati in un file CSV. Questo file permette una facile consultazione e analisi dei dati. Inoltre, l'output dell'operazione `explain` viene salvato in un file JSON, offrendo una panoramica dettagliata sulla performance della query e sul piano di esecuzione, utile per ottimizzare ulteriori esecuzioni.

Il risultato della query sarà un file CSV contenente:

- **Zona OMI:** il nome della zona OMI.
- **Inquinante:** il tipo di inquinante atmosferico.
- **Media:** la media dei valori registrati per ciascun inquinante in quella zona OMI.

Tale risultato viene successivamente usato come input per uno script Python che genera una rappresentazione visiva del risultato atteso. Come in precedenza, anche per questa query, non è possibile ottenere da *Compass* l'explain grafico ma ci può essere restituito solo in formato testuale invocandolo da PyMongo.

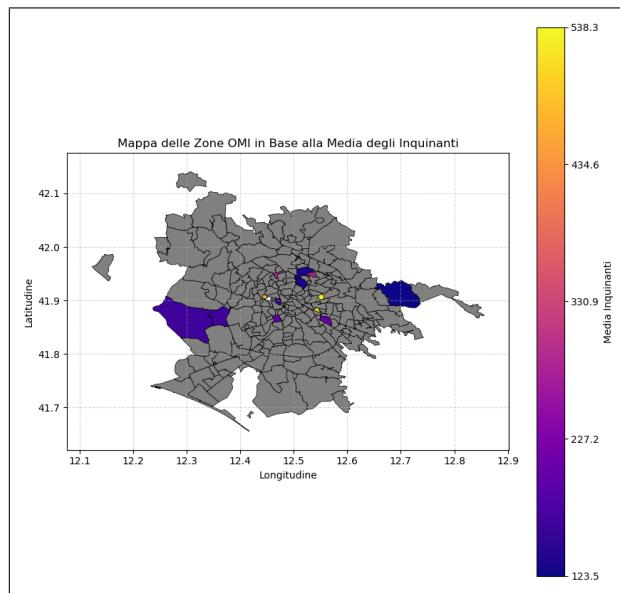


Figura 46: QUERY 4: Risultato

Totale documenti esaminati: 99820
Totale tempo di esecuzione (ms): 20335

Figura 47: QUERY 4: Esecuzione

4.9 Osservazioni

Nel contesto di questa query, è evidente il grande potenziale che MongoDB offre per le operazioni geospatiali. L'uso di operatori come `$geoIntersects` consente di eseguire interrogazioni efficienti su dati geospatiali, come nel caso in cui si desideri determinare quali inquinanti si trovano in specifiche zone geografiche (zone OMI).

MongoDB è particolarmente utile in scenari in cui le informazioni geografiche devono essere combinate con altri dati (ad esempio, la concentrazione di inquinanti nelle diverse aree), permettendo query molto potenti e flessibili.

Un altro vantaggio significativo è rappresentato dagli indici geospatiali: MongoDB è in grado di creare indici spaziali per velocizzare le ricerche su dati geospatiali, riducendo notevolmente il tempo di esecuzione delle query. Gli indici, infatti, consentono di ridurre la quantità di dati da esaminare, migliorando l'efficienza e riducendo i tempi di risposta, soprattutto quando si lavora con grandi volumi di dati.

Tuttavia, come evidenziato dagli explain, quando la query coinvolge **operazioni di merge su sharded data** (cioè quando si uniscono dati provenienti da shard diversi), le prestazioni possono rallentare significativamente. In MongoDB, quando i dati provengono da **shard diversi** e devono essere combinati, il processo di **shard merging** può essere lungo se le operazioni richiedono un'elaborazione complessa. In questi casi, MongoDB deve eseguire un'ulteriore fase di coordinamento per raccogliere i dati da vari shard, (potenzialmente localizzati anche in punti diversi del globo) un processo che può aumentare il tempo di esecuzione complessivo della query.

5 Neo4j

La migrazione da MongoDB a Neo4j rappresenta un cambiamento significativo nella rappresentazione dei dati, poiché, richiede non solo il trasferimento delle informazioni ma una vera e propria riprogettazione del **modello concettuale**.

MongoDB è un database NoSQL orientato ai documenti, ideale per dati gerarchici o non strutturati. Neo4j, invece, è un database a grafo, pensato per rappresentare entità e relazioni in modo esplicito e diretto. Questo passaggio offre vantaggi considerevoli per alcuni tipi di applicazioni, ma comporta anche un ripensamento del modo in cui i dati sono organizzati ed interrogati.

In MongoDB, i dati vengono archiviati in documenti JSON-like organizzati in collezioni. Ogni documento è un'unità autonoma, che può contenere informazioni strutturate in modo gerarchico. Le relazioni tra i documenti, tuttavia, non sono native e devono essere simulate tramite riferimenti o mediante l'embedding di informazioni correlate.

Con Neo4j, invece, si passa a una rappresentazione a grafo, dove i nodi rappresentano entità (come inquinanti, zone OMI, ecc.), mentre le relazioni tra nodi descrivono i legami tra di essi. Ad esempio, si avrà un nodo che rappresenta un inquinante connesso a un nodo che rappresenta una zona OMI attraverso una relazione che indica dove l'inquinante è stato misurato. Questa struttura permette di interrogare i dati in modo più naturale quando si tratta di analizzare connessioni complesse, ad esempio per identificare la distribuzione geografica degli inquinanti all'interno di una zona.

La migrazione richiede anche un adattamento delle logiche di interrogazione. MongoDB utilizza il proprio linguaggio di query (MQL), con un focus sulle aggregazioni e sui filtri basati sui campi. Neo4j, invece, adotta **Cypher**, un linguaggio specifico per i grafi, che permette di esprimere query relazionali in modo intuitivo, come trovare tutti gli inquinanti misurati in una determinata area o esplorare connessioni tra zone vicine.

Uno dei punti di forza di Neo4j rispetto a MongoDB è la gestione nativa delle relazioni. In MongoDB, le query che coinvolgono relazioni complesse richiedono spesso più join o pipeline di aggregazione, che possono risultare costose in termini di prestazioni su dataset di grandi dimensioni. Neo4j, invece, eccelle in questi scenari, grazie alla sua struttura a grafo che ottimizza l'attraversamento delle relazioni.

Per quanto riguarda la gestione geospaziale, entrambi i database offrono funzionalità avanzate, ma con approcci diversi. MongoDB supporta molteplici operatori per eseguire query geospaziali, utili per calcolare distanze o verificare se un punto si trova all'interno di una determinata area. Neo4j, con il **plugin Spatial**, consente di integrare direttamente dati geografici nei nodi e di effettuare interrogazioni avanzate sfruttando la semantica dei grafi. Ad esempio, è possibile trovare tutti gli inquinanti situati entro un certo raggio da un punto specifico con query che combinano distanza e relazioni tra nodi.

Un altro aspetto da considerare è l'uso degli indici e dei vincoli. In MongoDB, gli indici vengono creati sui campi dei documenti per velocizzare le operazioni di ricerca, come quelle geospaziali. In Neo4j, gli **indici** possono essere applicati non solo alle proprietà dei nodi, ma anche alle relazioni. Inoltre, è possibile definire **vincoli** per garantire l'unicità o la presenza di dati, migliorando l'integrità del grafo.

Dal punto di vista delle prestazioni, Neo4j offre un vantaggio significativo per le query che attraversano molte relazioni, come l'identificazione di tutti i dati connessi a una determinata zona OMI e ai suoi inquinanti. Tuttavia, MongoDB rimane più performante per aggregazioni e filtri su documenti autonomi, grazie alla sua natura schemaless.

In conclusione, la migrazione da MongoDB a Neo4j permette di sfruttare al meglio i vantaggi di un modello a grafo, soprattutto per applicazioni che richiedono un'analisi intensiva delle relazioni e connessioni complesse. Tuttavia, questa transizione richiede uno sforzo iniziale per riprogettare i dati e adattare le logiche di query, compensato poi da prestazioni superiori in scenari che richiedono analisi relazionali o geospaziali avanzate.

5.1 Strutturazione dei Dati

La migrazione dei dati da MongoDB a Neo4j è stata affrontata come un processo manuale, senza utilizzare strumenti automatici come APOC (*A Procedure On Cypher*) [12].

Questa scelta è stata dettata dalla necessità di preservare e ricostruire la struttura complessa e le relazioni annidate che caratterizzavano il nostro database da migrare.

In MongoDB, le informazioni erano organizzate in documenti JSON-like, spesso con relazioni implicite o gerarchiche tra i dati. Ad esempio, un documento relativo ad un inquinante poteva contenere direttamente informazioni sulla zona OMI associata o un riferimento a un altro documento correlato. Questa struttura non è facilmente traducibile in un grafo senza un'analisi accurata della semantica stessa che vi è nei dati!

In Neo4j, abbiamo adottato un approccio che mette al centro il grafo come modello concettuale. I dati sono stati completamente riorganizzati, in nodi e corrispondenti relazioni.

La differenza principale che salta all'occhio è quella di aver elevato ad entità, quello che in MongoDB siamo riusciti a modellare col costrutto della *Collections*, ovvero: le **Città**. E' partendo da questa entità che siamo riuscito ad aver un unico grande grafo interconnesso, un esempio è visibile nelle immagini sottostanti:

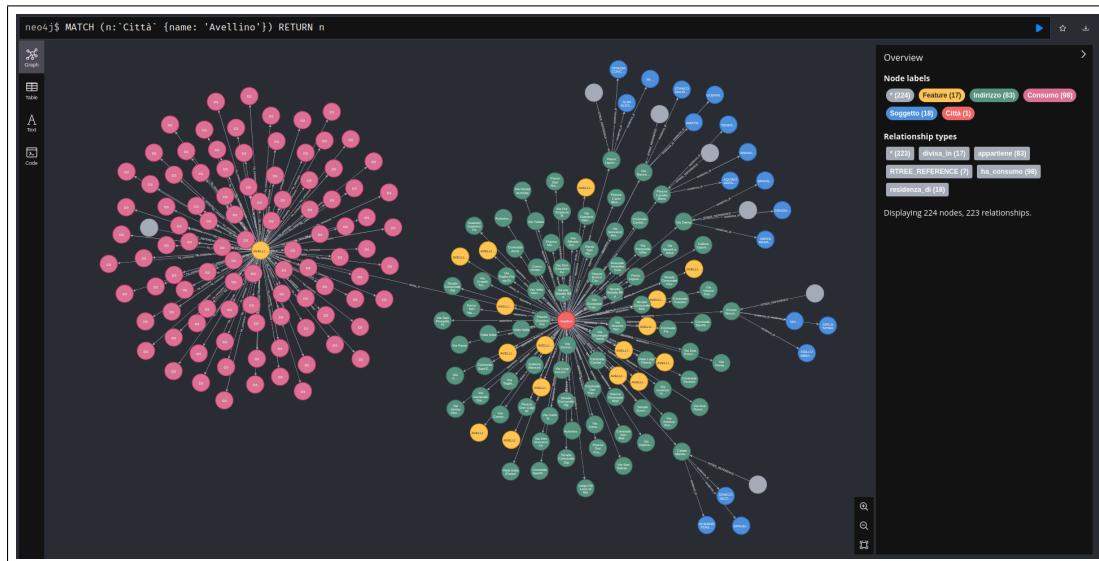


Figura 48: Visione d'insieme della città di Avellino

5.1.1 I Redditi

Il *focus* si è spostato sull'entità del mondo reale corrispondente.

Si è creato un nodo per ogni abitante, con i relativi dati anagrafici e reddituali come proprietà dello stesso.

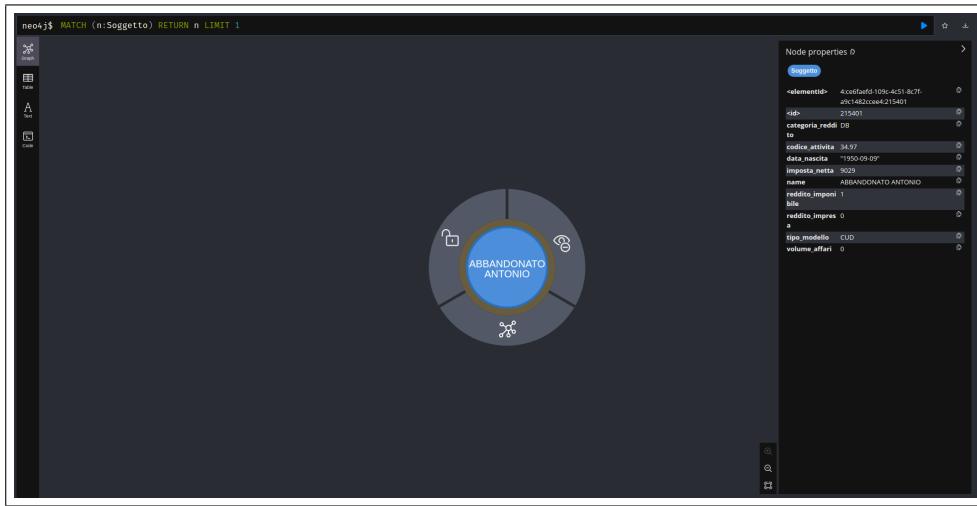


Figura 49: Il Nodo *Soggetto*

5.1.2 Le Zone OMI

Questo nodo rappresenta la *Zona OMI* nella sua rispettiva forma poligonale.

Qui abbiamo avuto la difficoltà maggior nell'intero processo migrazionale: in quanto, Neo4j nativamente non offre alcun supporto per memorizzare, né tantomeno effettuare query su dati geospatiali! Per operare con questo tipo di dati abbiamo utilizzato un plugin ad-hoc, ne discuteremo nel dettaglio nel seguito...

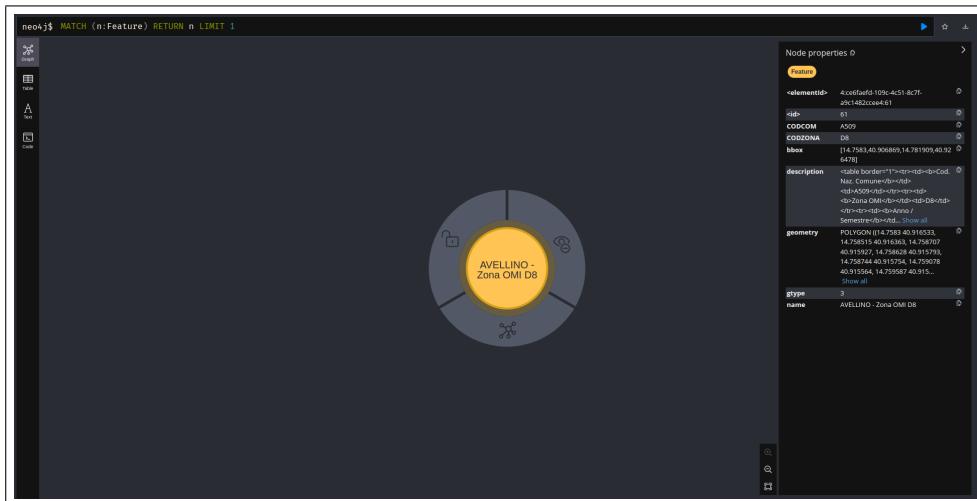


Figura 50: Il Nodo *Zona OMI*

5.1.3 Gli Inquinanti Atmosferici

Questo nodo rappresenta l'inquinante misurato e ha come attributo la stazione che lo ha misurato. La stazione di monitoraggio è stata elevata ad entità autonoma ed il rispettivo valore misurato è divenuto attributo della relazione fra quest'ultimi.

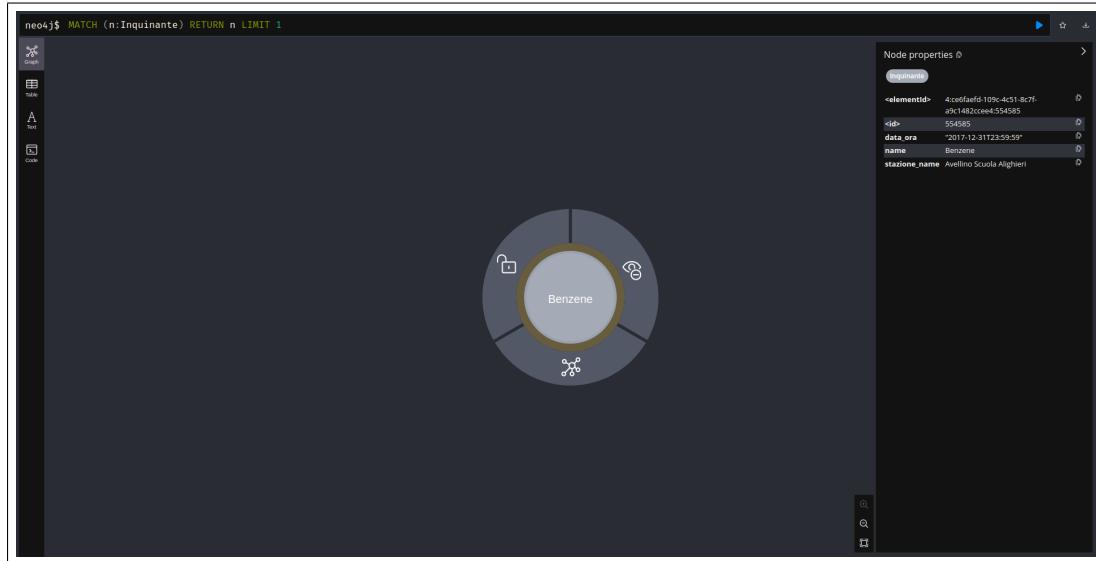


Figura 51: Il Nodo *Inquinanti Atmosferici*

5.1.4 I Consumi Elettrici

Analogo discorso del punto precedente ma la relazione ora sussiste tra Zona OMI e il nodo del Consumo.

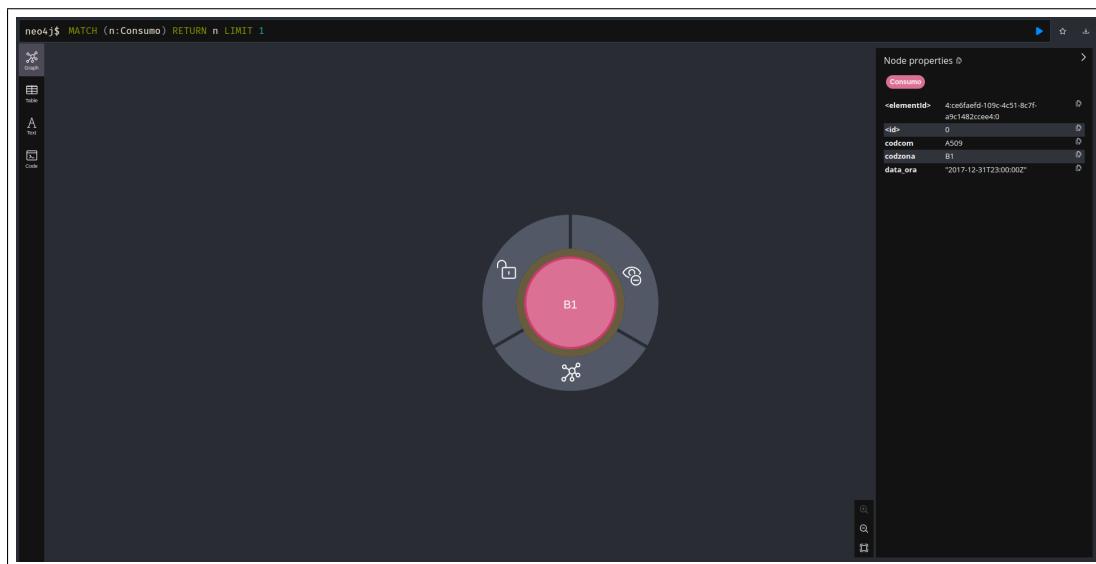


Figura 52: Il Nodo *Consumi Elettrici*

5.1.5 Le informazioni anagrafiche dei *Nuclei Familiari*

Questa è l'entità che rappresenta le informazioni relative alla residenza. Collegando questo nodo alle persone è possibile ricostruire il nucleo familiare.

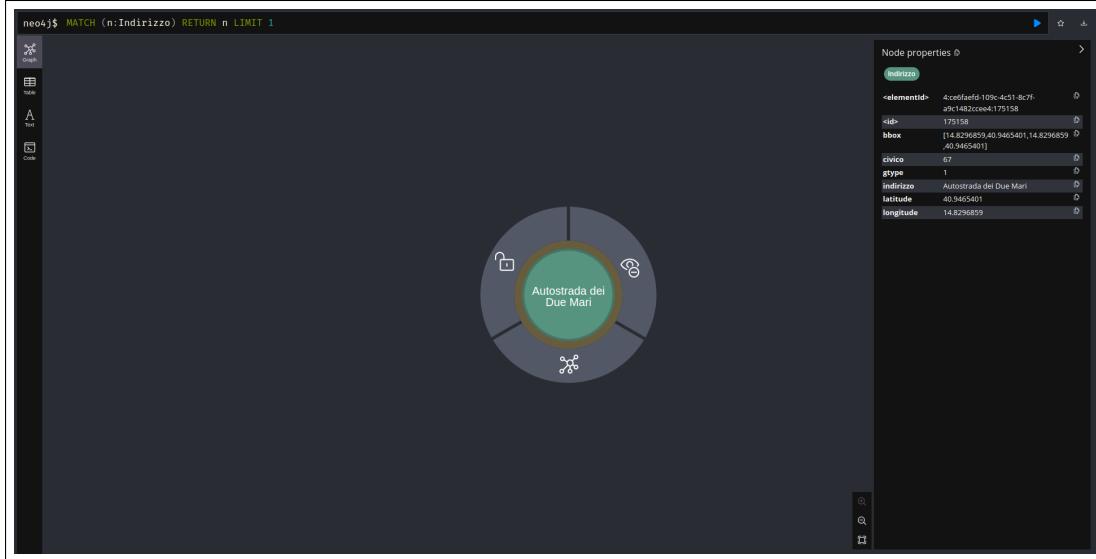


Figura 53: Il Nodo *Indirizzo*

5.1.6 I Pannelli

Questo nodo non contiene alcun dato (*Empty Node*) poiché la sua semantica è restituita solo se relato all'indirizzo corrispondente alla sua locazione spaziale.

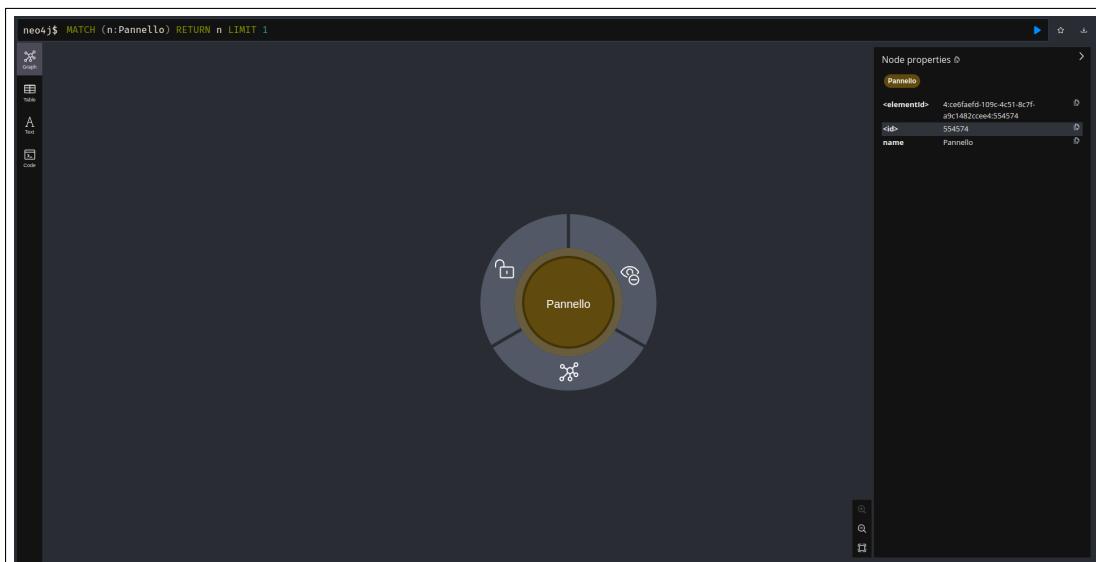


Figura 54: Il Nodo *Pannello*

5.2 Relazioni tra i Nodi

L'immagine rappresenta un sottografo completamente connesso, che funge da esempio per illustrare la struttura e le relazioni presenti nel nostro grafo globale migrato in Neo4j. Questo sottografo evidenzia la modellazione gerarchica e relazionale dei dati, con un focus sulle connessioni significative tra nodi distinti e le proprietà associate alle relazioni.

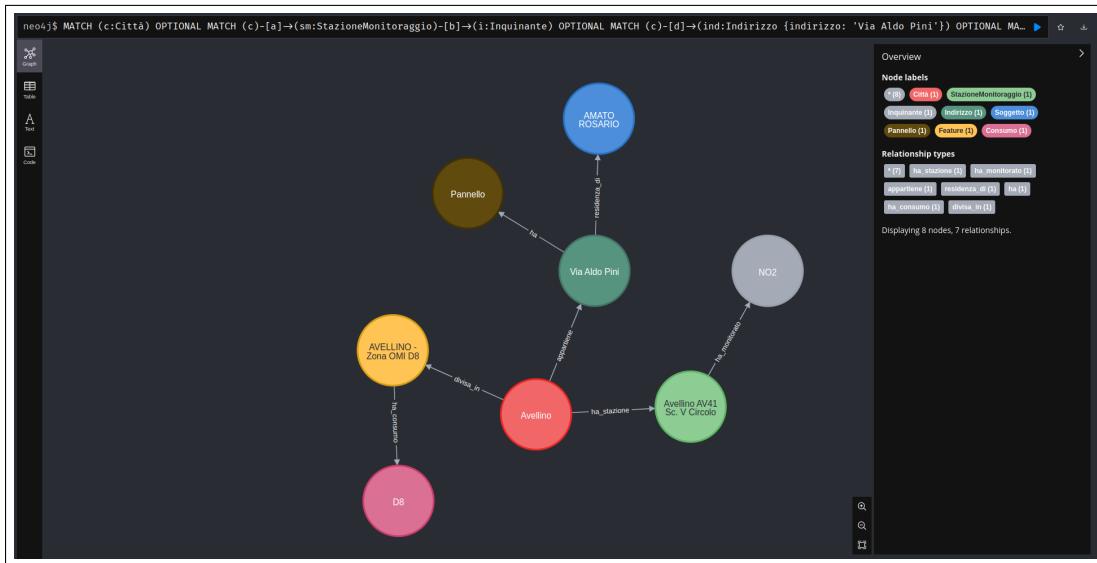


Figura 55: Visione di dettaglio del grafo

Ora analizziamo nel dettaglio ciascuna relazione e la sua semantica nella strutturazione generale dei dati:

ha_stazione (Nodo Avellino → Nodo Avellino AV41 Sc. V Circolo):

- **Significato:** Rappresenta il legame tra una città e le sue infrastrutture di monitoraggio ambientale. La città di Avellino è collegata alla stazione di monitoraggio AV41 Sc. V Circolo, che raccoglie dati relativi all'ambiente urbano.
- **Utilità:** Questa relazione consente di associare i dati ambientali raccolti a specifiche aree urbane, utile per analisi di inquinamento e pianificazione ecologica.

ha_monitorato (Nodo Avellino AV41 Sc. V Circolo → Nodo NO2):

- **Significato:** Specifica che la stazione di monitoraggio osserva un particolare inquinante, in questo caso il biossido di azoto (NO2).
- **Proprietà della relazione:**
 - *UM (Unità di Misura)*: Specifica il formato dei dati rilevati (es. $\mu\text{g}/\text{m}^3$).
 - *Valore*: Contiene il livello quantitativo dell'inquinante rilevato (es. $40 \mu\text{g}/\text{m}^3$).
- **Utilità:** Permette di tracciare la qualità dell'aria e i livelli di specifici inquinanti in base alla posizione della stazione.

appartiene (Nodo Via Aldo Pini → Nodo Avellino):

- **Significato:** Collega l'indirizzo Via Aldo Pini alla città di Avellino, stabilendo una relazione spaziale tra un punto specifico e il contesto urbano generale.
- **Utilità:** Utile per analisi spaziali che coinvolgono zone urbane specifiche, come la localizzazione di infrastrutture o residenze.

residenza_di (Nodo Via Aldo Pini → Nodo Amato Rosario):

- **Significato:** Indica che una persona, Amato Rosario, risiede presso l'indirizzo Via Aldo Pini.
- **Utilità:** Consente di collegare persone a luoghi specifici, utile per studi demografici, analisi di mobilità, o altre applicazioni legate alla residenza.

ha (Nodo Via Aldo Pini → Nodo Pannello):

- **Significato:** Collega l'indirizzo Via Aldo Pini a un pannello, che può rappresentare un dispositivo tecnologico (ad esempio, un pannello solare o un sensore ambientale).
- **Utilità:** Abilita l'analisi dell'integrazione tra le infrastrutture tecnologiche e gli spazi urbani.

divisa_in (Nodo Avellino → Nodo AVELLINO – Zona OMI D8):

- **Significato:** La città di Avellino è suddivisa in zone OMI (Osservatorio del Mercato Immobiliare). Una di queste è la Zona D8, che fornisce un livello di dettaglio maggiore del territorio.
- **Utilità:** Permette di effettuare analisi localizzate su sotto-zone urbane, utile per valutazioni immobiliari, studi urbanistici, o suddivisioni amministrative.

ha_consumo (Nodo AVELLINO – Zona OMI D8 → Nodo D8):

- **Significato:** Collega una specifica zona OMI (D8) a un dato di consumo energetico.
- **Proprietà della relazione:**
 - *UM (Unità di Misura)*: Indica il formato del dato (es. kWh).
 - *Consumo Energetico*: Contiene il valore del consumo registrato (es. 1200 kWh).
- **Utilità:** Consente di monitorare e analizzare i consumi energetici per specifiche aree urbane, essenziale per studi di sostenibilità o efficienza energetica.

5.3 Il Plugin "*Neo4j-Spatial*" [13]

Il plugin **Neo4j-Spatial** è una libreria che estende le funzionalità di Neo4j per permettere di lavorare con **dati geospaziali** in modo efficiente. Con esso, è possibile memorizzare e interrogare dati geografici come punti, linee, poligoni, e utilizzare operazioni spaziali per eseguire ricerche e analisi geospaziali direttamente nel grafo.

Cosa fa Neo4j-Spatial?

Neo4j-Spatial consente di memorizzare dati spaziali nel grafo e di eseguire query spaziali come:

- **Punti (Point)**: Rappresentazione di coordinate geografiche in un sistema di riferimento spaziale, ad esempio, latitudine e longitudine.
- **Poligoni (Polygon)**: Geometrie definite da un insieme di coordinate, come aree geografiche delimitate.
- **Linee (LineString)**: Rappresentazione di linee geografiche, ad esempio per tracciare strade o percorsi.

Il plugin rende anche possibile eseguire operazioni come la ricerca di nodi o poligoni all'interno di una determinata area geografica, calcolare distanze tra punti, identificare l'intersezione tra geometrie, e molto altro.

Caratteristiche principali del plugin

1. Tipi di dati geospaziali:

- **Point**: Consente di rappresentare la posizione geografica di un oggetto. Un punto è definito da una coppia di coordinate (latitudine, longitudine).
- **Polygon**: Una figura geometrica chiusa composta da un insieme di punti che definiscono i bordi del poligono. È utile per rappresentare aree geografiche complesse, come confini di città o aree di interesse.
- **LineString**: Una serie di punti connessi che formano una linea. Viene utilizzato per tracciare percorsi, strade o fiumi.

2. Operazioni spaziali:

- **Intersezione**: Verifica se due geometrie (come due poligoni o un punto e un poligono) si sovrappongono o si intersecano. Ad esempio, puoi determinare se un punto (una città) si trova all'interno di un poligono (un'area geografica).
- **Contenimento**: Verifica se una geometria (ad esempio un punto) è contenuta all'interno di un'altra geometria (ad esempio un poligono).
- **Distanza**: Calcola la distanza tra due punti geospaziali. Questa operazione è utile per trovare oggetti vicini, come negozi o stazioni di monitoraggio.
- **Vicino (Near)**: Restituisce gli oggetti spaziali che sono più vicini a una determinata posizione.

3. Indici spaziali:

- Neo4j-Spatial sfrutta indici spaziali per migliorare le prestazioni delle query geospatiali. Gli indici spaziali rendono le operazioni di ricerca e intersezione molto più veloci, specialmente quando ci sono molti nodi o poligoni.
- Questi indici possono essere utilizzati per cercare rapidamente punti, poligoni e altre geometrie nel grafo.

4. Supporto per vari formati:

- Neo4j-Spatial supporta diversi formati di dati spaziali, come **GeoJSON**, che è uno dei formati più comuni per la rappresentazione di dati geospatiali in formato JSON.

5. Funzionalità di ricerca spaziale avanzata:

- Neo4j-Spatial consente di eseguire ricerche avanzate come la ricerca di nodi che si trovano all'interno di una determinata area (ad esempio una regione geografica) o di nodi che si trovano a una certa distanza da un altro nodo.

Come funziona Neo4j-Spatial?

1. **Aggiungere dati spaziali al grafo:** Puoi memorizzare dati spaziali come proprietà su nodi o relazioni. Ad esempio, puoi avere un nodo di tipo **Città** con una proprietà **location** che rappresenta la latitudine e la longitudine della città come punto geospatiale.
2. **Creare indici spaziali:** Per ottimizzare le query geospatiali, puoi creare indici spaziali sui nodi che contengono dati geospatiali. Questo accelera notevolmente le ricerche spaziali.
3. **Query spaziali con Cypher:** Neo4j-Spatial si integra perfettamente con il linguaggio di query **Cypher**, il che significa che puoi utilizzare le funzionalità spaziali direttamente nelle tue query. Ad esempio, puoi usare la funzione **spatial.intersects()** per trovare nodi che si trovano all'interno di un poligono, o **spatial.distance()** per calcolare la distanza tra due punti.

5.3.1 Esempi di utilizzo

1. **Trova tutte le città che si trovano all'interno di una certa area geografica:** Se hai un poligono che rappresenta una zona (ad esempio, una regione geografica) e vuoi trovare tutte le città che si trovano all'interno di questa zona, puoi utilizzare una query di tipo:

```
MATCH (c:Città)
WHERE spatial.intersects(c.location, polygon)
RETURN c
```

2. **Calcolare la distanza tra due città:** Se vuoi calcolare la distanza tra due città, puoi utilizzare la funzione `spatial.distance()` per ottenere la distanza tra i punti di due città:

```
MATCH (c1:Città {name: 'Roma'}), (c2:Città {name: 'Milano'})  
RETURN spatial.distance(c1.location, c2.location) AS distanza
```

3. **Ricerca di punti di interesse più vicini a una determinata posizione:** Se hai un nodo rappresentante una stazione di monitoraggio con una posizione geospaziale, puoi trovare le stazioni più vicine a un determinato punto utilizzando la funzione `spatial.near()`:

```
MATCH (s:StazioneMonitoraggio)  
WHERE spatial.near(s.location, point({latitude: 41.9028, \\  
longitude: 12.4964}), 1000)  
RETURN s
```

In questo esempio, la query restituisce tutte le stazioni di monitoraggio che si trovano a meno di 1000 metri da un punto specificato (in questo caso, le coordinate di Roma).

Vantaggi di Neo4j-Spatial

- **Facilità di integrazione con Neo4j:** Neo4j-Spatial è un plugin progettato per integrarsi perfettamente con Neo4j. Le query spaziali vengono eseguite direttamente in Cypher, lo che facilita l'integrazione con il resto del modello di dati grafico.
- **Indici spaziali:** L'uso di indici spaziali consente di eseguire query geospaziali molto più velocemente, anche su grandi dataset di dati geospaziali.
- **Potente supporto per le analisi geospaziali:** Con Neo4j-Spatial, è possibile eseguire analisi avanzate come la ricerca di oggetti in un'area, il calcolo delle distanze, o la verifica delle intersezioni tra diverse geometrie.
- **Versatilità:** Supporta una vasta gamma di formati e operazioni spaziali, il che rende Neo4j-Spatial utile in una varietà di casi d'uso, dall'analisi di percorsi alla pianificazione urbana, fino alla gestione di dati geospaziali complessi.

5.3.2 Funzionamento interno del Plugin: gli "R-Tree" [14]

L'R-Tree è una struttura dati fondamentale per gestire e indicizzare dati spaziali, ed è utilizzata nel plugin Neo4j-Spatial per ottimizzare l'accesso e le operazioni su geometrie complesse come poligoni, linee e rettangoli. Esploriamo nel dettaglio cos'è un R-Tree, come funziona e perché viene utilizzato in Neo4j-Spatial.

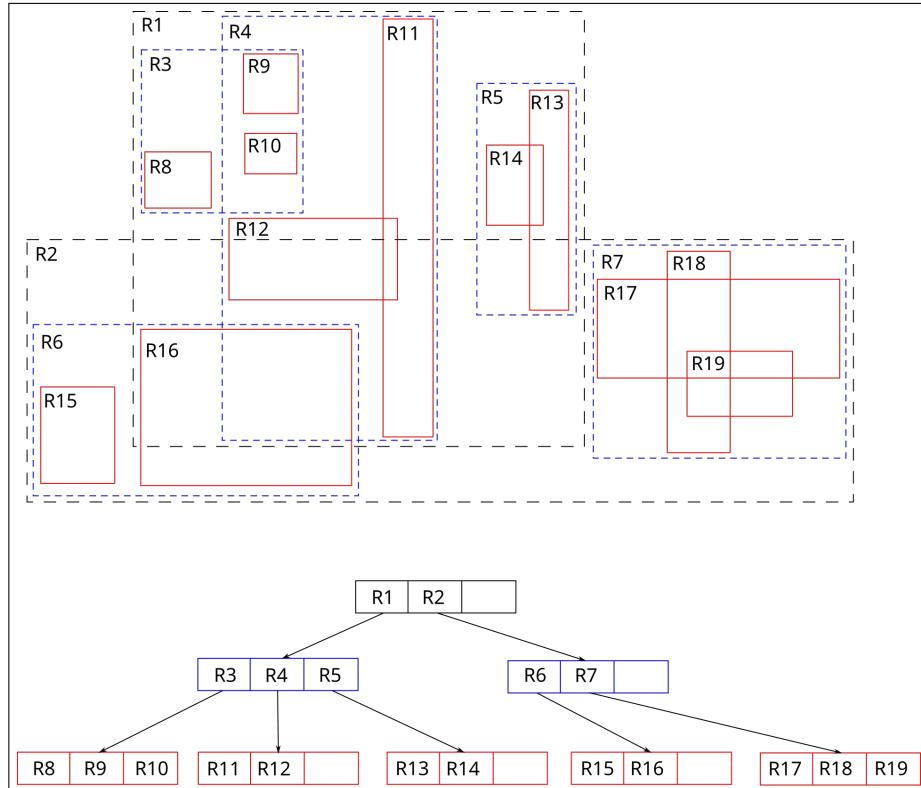


Figura 56: La struttura dati "R-Tree"

Cos'è un R-Tree?

Un R-Tree è un albero bilanciato progettato per l'indicizzazione di oggetti spaziali multidimensionali, come rettangoli, poligoni, cerchi e altre forme geometriche. La struttura dell'R-Tree è particolarmente adatta per applicazioni geospaziali, dove è fondamentale lavorare con oggetti che occupano aree nello spazio e la loro posizione deve essere determinata rispetto ad altre geometrie.

In termini semplici, un R-Tree è un albero dove:

- Ogni nodo contiene un rettangolo (spesso chiamato "bounding box") che rappresenta l'area geografica coperta dal nodo.
- Ogni foglia dell'albero rappresenta un oggetto spaziale effettivo, come un punto, una linea o un poligono.
- I nodi non fogliare rappresentano aree "raggruppate" di oggetti spaziali, e sono utili per la ricerca efficiente, riducendo il numero di confronti tra oggetti spaziali.

Struttura dell'R-Tree

Un R-Tree è costituito da vari livelli di nodi:

- **Nodi foglia:** Contengono i dati spaziali effettivi (ad esempio, i poligoni o i punti) e le loro proprietà. In pratica, ogni foglia ha un bounding box che rappresenta l'area spaziale dell'oggetto.
- **Nodi non foglia:** Contengono bounding boxes che racchiudono i bounding box dei nodi figli. Ogni nodo non foglia rappresenta una “superficie” che racchiude altre superfici.
- **Radice:** È il nodo più alto dell'albero e contiene il bounding box che racchiude tutti gli oggetti nell'albero. La radice è il punto di partenza per la ricerca spaziale.

Ogni nodo dell'albero è generalmente bilanciato in modo che l'albero non diventi troppo profondo, migliorando così le prestazioni nelle operazioni di ricerca e aggiornamento.

Funzionamento dell'R-Tree

L'R-Tree è progettato per risolvere il problema di come trovare oggetti spaziali che si trovano in determinate aree o che interagiscono con altre geometrie. Questo viene fatto organizzando i dati in modo da ridurre al minimo il numero di confronti necessari durante le ricerche.

Inserimento di nuovi oggetti

Quando un nuovo oggetto spaziale (ad esempio, un nuovo poligono) viene inserito nell'R-Tree, il sistema:

- Identifica il nodo in cui il nuovo oggetto dovrebbe essere inserito, cercando il nodo con il bounding box che può ospitare il nuovo oggetto senza far crescere troppo le dimensioni del bounding box (per evitare un eccessivo allargamento delle aree).
- Se il nodo è pieno, viene diviso in due nodi, creando una struttura bilanciata.

Operazioni di ricerca

Quando si eseguono operazioni di ricerca, come la ricerca di intersezioni spaziali o la determinazione di distanze tra oggetti, l'R-Tree sfrutta il principio di “propagazione” delle intersezioni tra i nodi:

- Inizia la ricerca dalla radice dell'albero e naviga verso il basso, visitando solo i nodi che potrebbero contenere oggetti che soddisfano la condizione di ricerca.
- I nodi vengono esplorati in base alla coincidenza dei loro bounding box con l'area di ricerca. Poiché i nodi sono organizzati in modo spaziale, questo approccio riduce notevolmente il numero di confronti da fare, limitando le ricerche solo ai nodi che effettivamente potrebbero essere rilevanti.
- Quando si arriva ai nodi foglia, viene eseguita la ricerca effettiva sugli oggetti spaziali (ad esempio, calcolando l'intersezione tra il poligono ricercato e gli altri oggetti).

Vantaggi dell'R-Tree

- Efficienza nelle ricerche spaziali:** L'R-Tree riduce significativamente il numero di comparazioni necessarie durante le ricerche spaziali. Poiché i nodi non foglia contengono informazioni sugli oggetti che racchiudono, è possibile eliminare ampie porzioni del grafo senza doverle esplorare nel dettaglio.
- Adattabilità a geometrie complesse:** L'R-Tree è molto flessibile e si adatta bene a dati spaziali di vario tipo e complessità, come poligoni, linee e punti. È particolarmente utile per applicazioni geospaziali che richiedono operazioni di ricerca su grandi quantità di dati.
- Bilanciamento automatico:** Il bilanciamento dell'albero evita che la struttura diventi troppo sbilanciata, il che potrebbe rallentare le ricerche. In pratica, l'albero cresce e si adatta in modo dinamico ai dati, mantenendo buone prestazioni.
- Ottimizzazione per l'area:** Gli oggetti spaziali che sono vicini tra loro fisicamente sono facilmente raggruppati insieme, riducendo la necessità di fare confronti tra oggetti distanti tra loro.

Utilizzo in Neo4j-Spatial

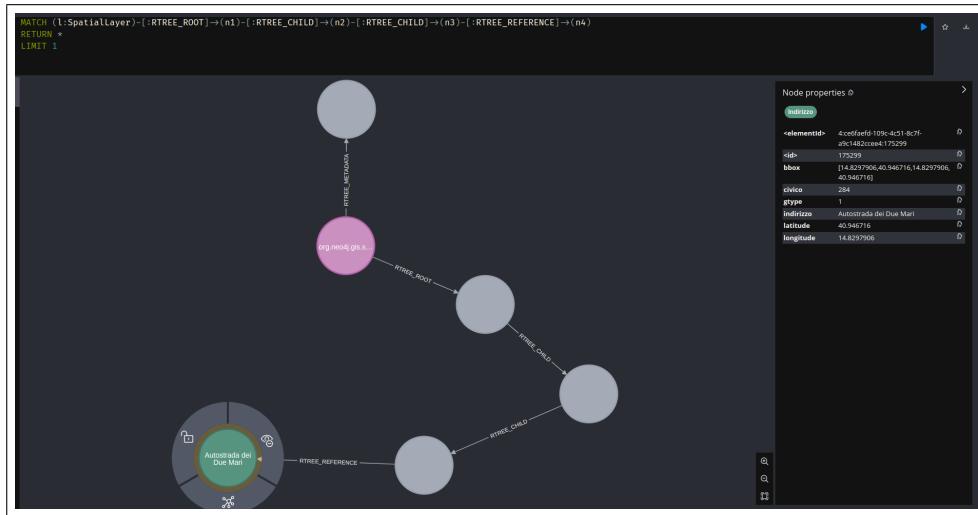


Figura 57: Struttura del "Layer Spaziale" in Neo4j

In Neo4j-Spatial, l'R-Tree viene utilizzato come uno degli indici spaziali per memorizzare e ottimizzare l'accesso ai dati spaziali memorizzati nel grafo. L'uso di questo tipo di indice consente di eseguire operazioni come:

- Ricerche di vicinanza: Identificare rapidamente quali nodi (ad esempio, stazioni meteo, punti di interesse) si trovano entro una certa distanza da un punto dato.
- Intersezione tra geometrie: Verificare se due geometrie (ad esempio, un poligono e una linea) si intersecano, utile in operazioni di analisi geospaziale come l'analisi del rischio di inondazione o la pianificazione territoriale.
- Calcolo delle distanze: Determinare la distanza tra oggetti spaziali, per esempio tra una stazione di monitoraggio e un punto di interesse.

Considerazioni sulle prestazioni

- **Scalabilità:** L'R-Tree è molto efficace quando si gestiscono grandi quantità di dati spaziali. Man mano che i dati aumentano, l'albero si espande in modo bilanciato, mantenendo la ricerca efficiente anche con grafi complessi.
- **Aggiornamenti e inserimenti:** L'R-Tree è ottimizzato per operazioni di lettura, ma le operazioni di inserimento e aggiornamento possono richiedere una riorganizzazione dei nodi, che potrebbe comportare un sovraccarico. Tuttavia, questo è un compromesso che consente di mantenere alta l'efficienza nelle operazioni di ricerca.

In conclusione, Neo4j-Spatial è uno strumento potente per lavorare con dati spaziali all'interno del modello grafico di Neo4j. Consente di integrare senza problemi operazioni geospatiali in un grafo, sfruttando le potenti funzionalità di Neo4j per interrogare, analizzare e visualizzare dati geografici. Con il supporto per operazioni spaziali avanzate, indici geospatiali, e una facile integrazione con Cypher, Neo4j-Spatial è ideale per applicazioni che richiedono analisi geospatiali su grandi quantità di dati.

5.4 Indici e Vincoli

In Neo4j, i **vincoli** (constraints) e gli **indici** (indexes) sono strumenti cruciali per migliorare la gestione dei dati e le prestazioni delle query nel grafo, ma servono a scopi diversi.

Vincoli (*Constraints*)

I vincoli sono regole applicate sui dati per garantire l'integrità e la consistenza del grafo. In altre parole, i vincoli definiscono restrizioni su come i dati devono essere organizzati e interconnessi. Essi impediscono che i dati non validi vengano inseriti o modificati nel grafo, assicurando che vengano rispettate determinate regole.

Ci sono vari tipi di vincoli in Neo4j:

- **Vincoli di unicità:** Questi vincoli garantiscono che una proprietà su un nodo o una relazione abbia un valore unico all'interno di un determinato contesto. Ad esempio, un vincolo di unicità potrebbe essere applicato su una proprietà come l'ID di un nodo, in modo che non possano esistere due nodi con lo stesso ID.
- **Vincoli di esistenza:** Questi vincoli assicurano che una proprietà esista su un nodo o una relazione. In pratica, garantiscono che determinati dati siano sempre presenti per ogni entità del grafo, migliorando la qualità dei dati e la loro coerenza.
- **Vincoli di tipo:** Questi vincoli possono essere applicati per assicurarsi che un nodo o una relazione sia sempre di un certo tipo, rafforzando la struttura del modello del grafo.

L'applicazione di vincoli in Neo4j è importante perché, oltre a mantenere i dati consistenti e validi, permette di evitare errori e garantire che la logica di business sia rispettata. I vincoli sono particolarmente utili quando è necessario mantenere un alto grado di integrità, come nel caso di dati sensibili o critici.

Indici (*Indexes*)

Gli indici, al contrario, sono strumenti progettati per **migliorare le prestazioni delle query**. In Neo4j, gli indici velocizzano la ricerca dei nodi e delle relazioni basati su determinati valori di proprietà. Essi creano strutture di dati ottimizzate per le ricerche rapide, evitando di dover eseguire una scansione completa del grafo ogni volta che una query viene eseguita.

Gli indici sono particolarmente utili quando si lavora con grafi di grandi dimensioni, dove la ricerca di specifici nodi o relazioni senza un indice potrebbe diventare estremamente lenta. Ad esempio, se una query cerca frequentemente nodi basati su una determinata proprietà (come un nome o un codice identificativo), l'indice consente di localizzare rapidamente quei nodi senza esaminare ogni singolo elemento del grafo.

Oltre agli indici tradizionali basati su proprietà, Neo4j offre anche **indici geospatiali** per ottimizzare le ricerche su dati geolocalizzati, come coordinate geografiche o poligoni. Questi indici sono particolarmente utili quando si lavora con dati spaziali, come nel caso di applicazioni di mappatura o di analisi geografica.

Differenze

Mentre i **vincoli** sono utilizzati per **garantire l'integrità dei dati** e evitare l'inserimento di dati errati o duplicati, gli **indici** sono progettati per **migliorare le prestazioni delle query**, rendendo il recupero dei dati più veloce ed efficiente. I vincoli sono legati alla qualità dei dati e alla loro consistenza, mentre gli indici si concentrano sull'efficienza nell'accesso ai dati.

In termini pratici:

- **I vincoli** limitano ciò che può essere inserito nel grafo, facendo rispettare le regole di validità, e possono prevenire situazioni in cui i dati siano incoerenti o non corretti.
- **Gli indici** ottimizzano l'accesso ai dati, migliorando le performance, specialmente quando si eseguono query che richiedono di trovare rapidamente nodi o relazioni con determinati valori di proprietà.

L'uso combinato di vincoli ed indici ci ha permesso di ottenere un grafo più affidabile, coerente e performante. I vincoli assicurano che i dati siano corretti, mentre gli indici migliorano la velocità di accesso e ricerca nel grafo.

Di seguito descriviamo nel dettaglio i vincoli e gli indici che abbiamo scelto di utilizzare; ricordiamo che gli indici geospatiali sono gestiti internamente dal plugin *Spatial*.

5.4.1 Nodo *Soggetto*

```
CREATE INDEX cat_reddito FOR (n:Soggetto) ON (n.categoria_reddito);
CREATE INDEX modello FOR (n:Soggetto) ON (n.tipo_modello);
CREATE INDEX red_imp FOR (n:Soggetto) ON (n.reddito_imponibile);
```

Figura 58: Nodo *Soggetto*: Indici

```
CREATE CONSTRAINT soggetto_unico
IF NOT EXISTS FOR (n:Soggetto)
REQUIRE (n.name, n.data_nascita, n.reddito_imponibile) IS UNIQUE;

CREATE CONSTRAINT name_not_null
IF NOT EXISTS FOR (n:Soggetto)
REQUIRE n.name IS NOT NULL;
CREATE CONSTRAINT data_not_null
IF NOT EXISTS FOR (n:Soggetto)
REQUIRE n.data_nascita IS NOT NULL;
CREATE CONSTRAINT red_imp_not_null
IF NOT EXISTS FOR (n:Soggetto)
REQUIRE n.reddito_imponibile IS NOT NULL;
CREATE CONSTRAINT modello_not_null
IF NOT EXISTS FOR (n:Soggetto)
REQUIRE n.tipo_modello IS NOT NULL;
CREATE CONSTRAINT imp_net_not_null
IF NOT EXISTS FOR (n:Soggetto)
REQUIRE n.imposta_netta IS NOT NULL;
```

Figura 59: Nodo *Soggetto*: Vincoli

5.4.2 Nodo *Consumo*

```
CREATE INDEX codcom_cons FOR (n:Consumo) ON (n.codcom);
CREATE INDEX codzona_cons FOR (n:Consumo) ON (n.codzona);
CREATE INDEX data_ora FOR (n:Consumo) ON (n.data_ora);
```

Figura 60: Nodo *Consumo*: Indici

```
CREATE CONSTRAINT consumo_unico
IF NOT EXISTS FOR (n:Consumo)
REQUIRE (n.codcom, n.codzona, n.data_ora) IS UNIQUE;

CREATE CONSTRAINT codcom_not_null
IF NOT EXISTS FOR (n:Consumo)
REQUIRE n.codcom IS NOT NULL;
CREATE CONSTRAINT codzona_not_null
IF NOT EXISTS FOR (n:Consumo)
REQUIRE n.codzona IS NOT NULL;
CREATE CONSTRAINT data_not_null
IF NOT EXISTS FOR (n:Consumo)
REQUIRE n.data_ora IS NOT NULL;
```

Figura 61: Nodo *Consumo*: Vincoli

5.4.3 Nodo *Indirizzo*

```
CREATE CONSTRAINT indirizzo_unico
IF NOT EXISTS FOR (n:Indirizzo)
REQUIRE (n.indirizzo, n.civico, n.latitude, n.longitude) IS UNIQUE;

CREATE CONSTRAINT ind_not_null
IF NOT EXISTS FOR (n:Indirizzo)
REQUIRE n.indirizzo IS NOT NULL;
CREATE CONSTRAINT civ_not_null
IF NOT EXISTS FOR (n:Indirizzo)
REQUIRE n.civico IS NOT NULL;
CREATE CONSTRAINT lat_not_null
IF NOT EXISTS FOR (n:Indirizzo)
REQUIRE n.latitude IS NOT NULL;
CREATE CONSTRAINT lon_not_null
IF NOT EXISTS FOR (n:Indirizzo)
REQUIRE n.longitude IS NOT NULL;
```

Figura 62: Nodo *Indirizzo*: Vincoli

5.4.4 Nodo Zona *OMI*

```
CREATE INDEX codcom FOR (n:Feature) ON (n.CODCOM);
CREATE INDEX codzona FOR (n:Feature) ON (n.CODZONA);
CREATE INDEX name_zona FOR (n:Feature) ON (n.name);
```

Figura 63: Nodo *Zona OMI*: Indici

```
CREATE CONSTRAINT codcom_codzona_unico
IF NOT EXISTS FOR (n:Feature)
REQUIRE (n.CODCOM, n.CODZONA) IS UNIQUE;
CREATE CONSTRAINT name_unico
IF NOT EXISTS FOR (n:Feature)
REQUIRE n.name IS UNIQUE;

CREATE CONSTRAINT name_not_null
IF NOT EXISTS FOR (n:Feature)
REQUIRE n.name IS NOT NULL;
CREATE CONSTRAINT codcom_not_null
IF NOT EXISTS FOR (n:Feature)
REQUIRE n.CODCOM IS NOT NULL;
CREATE CONSTRAINT codzona_not_null
IF NOT EXISTS FOR (n:Feature)
REQUIRE n.CODZONA IS NOT NULL;
CREATE CONSTRAINT geom_not_null
IF NOT EXISTS FOR (n:Feature)
REQUIRE n.geometry IS NOT NULL;
```

Figura 64: Nodo *Zona OMI*: Vincoli

5.5 Le Query

Di seguito proponiamo le stesse query viste per MongoDB (vedi § 4.8) nella loro versione nel linguaggio di interrogazione *Cypher*. Analogamente a quanto fatto in precedenza, seguirà una descrizione della query ed un'analisi prestazionale della sua esecuzione.

5.5.1 QUERY 1: Reddito medio per città



```

1 MATCH (c:Città)-[:appartiene]->(i:Indirizzo)-[:residenza_di]->(s:Soggetto)
2 RETURN c.name AS città, AVG(s.reddito_imponibile) AS media_reddito
3 ORDER BY media_reddito DESC
  
```

città	media_reddito
"Avellino"	2094.110570659258
"Venezia"	1726.9606016896805

Figura 65: QUERY 1: Struttura

Obiettivo

Questa query è progettata per calcolare la media del reddito imponibile dei soggetti (Soggetto) in ciascuna città (Città), restituendo le città ordinate in ordine decrescente rispetto alla media del reddito.

Descrizione della Query

- `MATCH (c:Città)-[:appartiene]->(i:Indirizzo)-[:residenza_di]->(s:Soggetto):`
 - Si cerca di navigare nel grafo partendo da nodi etichettati come **Città** (`c`).
 - La relazione `[:appartiene]` collega una città a un nodo **Indirizzo** (`i`). Questa relazione indica che un determinato indirizzo appartiene a una città.
 - Da ogni nodo **Indirizzo**, si segue la relazione `[:residenza_di]` per raggiungere i nodi etichettati come **Soggetto** (`s`), che rappresentano le persone che risiedono in quell'indirizzo.
- `RETURN c.name AS città, AVG(s.reddito_imponibile) AS media_reddito:`
 - La query restituisce due colonne:
 - * `c.name`: Il nome della città.
 - * `AVG(s.reddito_imponibile)`: La media dei redditi imponibili di tutti i soggetti collegati agli indirizzi della città.
- `ORDER BY media_reddito DESC:`
 - I risultati sono ordinati in ordine decrescente in base alla media dei redditi imponibili (`media_reddito`), con le città con redditi medi più alti visualizzate per prime.

Relazioni e Struttura del Grafo

- **Nodo Città (c):**
 - Rappresenta le città nel grafo.
 - Collegato agli indirizzi attraverso la relazione [:appartiene].
 - **Nodo Indirizzo (i):**
 - Rappresenta indirizzi o aree specifiche associate alle città.
 - Ha una relazione [:residenza_di] verso il nodo Soggetto.
 - **Nodo Soggetto (s):**
 - Rappresenta persone che vivono negli indirizzi.
 - Ogni nodo Soggetto ha un attributo `reddito_imponibile`, calcolato per calcolare la media.

Questa query può essere utile per:

- **Calcolo della media del reddito imponibile per città:**
 - Permette di analizzare il livello economico medio degli abitanti di diverse città.
 - Può essere utile in studi statistici, analisi socio-economiche e pianificazione politica o aziendale.
 - **Classificazione delle città per reddito medio:**
 - Ordinando le città per reddito medio in ordine decrescente, la query evidenzia le città con il livello economico più alto in cima alla lista.

Figura 66: QUERY 1: Esecuzione

5.5.2 QUERY 2: Tasso di pannelli per Zona OMI

```

1 MATCH (c:Città {name: 'Avellino'})-[r]-(i:Indirizzo)-[ha]-(p:Pannello)
2 WITH i.latitude AS lat, i.longitude AS lon, i
3 WITH point({latitude: lat, longitude: lon}) AS indirizzo_point, i
4 CALL spatial.intersects('zone_omi', indirizzo_point) YIELD node AS polygon
5 WITH polygon.name AS zone_name, COUNT(i) AS pannelli_count
6 RETURN zone_name, pannelli_count

```

Figura 67: QUERY 2: Struttura

Obiettivo

Questa query è concepita per determinare il numero di pannelli solari installati in ciascuna zona OMI della città di Avellino. Utilizza funzionalità geospaziali avanzate di Neo4j Spatial per verificare l'intersezione tra i punti geografici degli indirizzi e i poligoni che rappresentano le zone OMI.

Descrizione della Query

- MATCH (c:Città {name: 'Avellino'})-[r]-(i:Indirizzo)-[ha]-(p:Pannello):
 - Filtra i nodi **Città** per selezionare solo la città denominata "Avellino".
 - Naviga attraverso la relazione generica **[r]** per raggiungere i nodi **Indirizzo**.
 - Dall'indirizzo, segue la relazione **[ha]** per identificare i pannelli solari (**Pannello**) associati.
- WITH i.latitude AS lat, i.longitude AS lon, i:
 - Estraie le coordinate geografiche (**latitude** e **longitude**) dai nodi **Indirizzo** per prepararli alla creazione di un punto geografico.
- WITH point({latitude: lat, longitude: lon}) AS indirizzo_point:
 - Crea un oggetto **point** (struttura dati geospaziale di Neo4j) utilizzando le coordinate di latitudine e longitudine degli indirizzi.
- CALL spatial.intersects('zone_omi', indirizzo_point) YIELD node:
 - Usa la funzione **spatial.intersects** per verificare se il punto geografico **indirizzo_point** si trova all'interno di uno dei poligoni definiti nello strato **zone_omi**.
 - Restituisce i nodi dei poligoni con cui il punto ha un'intersezione.
- WITH polygon.name AS zone_name, COUNT(i) AS pannelli_count:
 - Aggrega i risultati per nome della zona OMI (**polygon.name**), conteggiando il numero di indirizzi che contengono pannelli solari in ciascuna zona.

- RETURN `zone_name`, `pannelli_count`:
 - Restituisce per ciascuna zona OMI il nome della zona e il numero di pannelli solari presenti.

Relazioni e Struttura del Grafo

- **Nodo Città:**
 - Contiene la città di interesse (in questo caso, "Avellino").
- **Nodo Indirizzo:**
 - Rappresenta indirizzi fisici con informazioni geografiche (`latitudine` e `longitudine`).
- **Nodo Pannello:**
 - Rappresenta pannelli solari associati agli indirizzi.
- **Zona OMI:**
 - È rappresentata da un poligono nello strato geospaziale `zone_omi`.

Uso di Neo4j Spatial

- **Intersezione geospaziale:**
 - La funzione `spatial.intersects` consente di verificare se un punto si trova all'interno di un poligono, rendendo possibile associare indirizzi con una posizione geografica alle zone OMI.
- **Strati geospaziali (`zone_omi`):**
 - Lo strato `zone_omi` è una collezione di poligoni che rappresentano le zone OMI, utilizzato per calcoli geospaziali.
- **Efficienza tramite indici:**
 - L'uso di indici geospaziali accelera la ricerca e le intersezioni.

Operator	Id	Details	Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)	Pipeline
+ProduceResults	0	zone_name, pannelli_count	117	17	0	0	0/0	0,700	In Pipeline 1
+EagerAggregation	1	polygon.name AS zone_name, COUNT(i) AS pannelli_count	117	17	6188	4120			
+ProcedureCall	2	spatial.intersects\$autostring_1, indirizzo_point :: (polygon :: NODE)	13645	3094		0			
+Projection	3	point([latitude: lat, longitude: lon]) AS indirizzo_point	1365	3257					
+Projection	4	cache[i.latitude] AS lat, cache[i.longitude] AS lon	1365	3257					
+CacheProperties	5	cache[i.latitude], cache[i.longitude]	1365	3257	6510				
+Filter	6	NOT ha = r AND p:Pannello	1365	3257	75588				
+Expand(All)	7	(i)-[ha]->(p)	4323	37796	53180				
+Filter	8	i:indirizzo	1986	15388	38610				
+Expand(All)	9	(c)-[r]->(i)	1986	15407	15404				
+Filter	10	c.name = \$autostring_0	1	1	6				
+NodeByLabelScan	11	c:città	10	2		376		257989/0	7776,401 Fused in Pipeline 0

Total database accesses: 187686 + ?, total allocated memory: 4592
17 rows
ready to start consuming query after 1376 ms, results consumed after another 7815 ms

Figura 68: QUERY 2: Esecuzione

5.5.3 QUERY 3: Consumi per Zona OMI

```

1 MATCH (c:Città)-[:divisa_in]→(z:Feature)-[r:ha_consumo]→()
2 WITH z, AVG(r.consumo_energetico) AS media_consumo
3 RETURN z.name AS zona, media_consumo

```

Figura 69: QUERY 3: Struttura

Obiettivo

Questa query in Neo4j ha lo scopo di calcolare il consumo energetico medio per ogni zona (rappresentata da un nodo di tipo Feature (alias "Zona OMI")) all'interno delle diverse città.

Descrizione della Query

- MATCH (c: Città)-[:divisa_in]→(z:Feature)-[r:ha_consumo]→():
 - (**c: Città**): Identifica i nodi di tipo Città, che rappresentano città nel grafo.
 - [:divisa_in]: Rappresenta la relazione che collega ogni città alle sue zone, modellate come nodi Feature.
 - (**z:Feature**): Rappresenta le zone o le unità territoriali all'interno della città.
 - [r:ha_consumo]: Identifica la relazione che associa una zona (Feature) a un valore di consumo energetico.
 - -i(): Il nodo di destinazione della relazione [ha_consumo] non è specificato nella query (può essere generico), ma il consumo è memorizzato come proprietà sulla relazione stessa (r.consumo_energetico).

- WITH z , AVG($r.consumo_energetico$) AS $media_consumo$:
 - Aggrega i dati per ciascuna zona (z).
 - Calcola la media dei consumi energetici utilizzando la proprietà `consumo_energetico` della relazione [`ha_consumo`].
- RETURN $z.name$ AS $zona$, $media_consumo$:
 - Restituisce:
 - * Il nome della zona ($z.name$).
 - * La media del consumo energetico calcolata per quella zona ($media_consumo$).

Relazioni e Struttura del Grafo

- **Nodo Città:**
 - Contiene informazioni sulle città incluse nell'analisi.
 - Collegato alle zone tramite la relazione [:divisa_in].
- **Nodo Feature:**
 - Rappresenta le zone territoriali o altre unità associate a una città.
 - Identificato dalla proprietà `name`, che rappresenta il nome della zona.
- **Relazione :ha_consumo:**
 - Memorizza dati specifici di consumo energetico associati a una zona.
 - La proprietà `consumo_energetico` sulla relazione contiene il valore di consumo.

```
Cypher 5
Planner COST
Runtime PIPELINED
Runtime version 5.25
Batch size 1024
+-----+-----+-----+-----+-----+-----+-----+-----+
| Operator | Id | Details | Estimated Rows | Rows | DB Hits | Memory (Bytes) | Page Cache Hits/Misses | Time (ms) | Pipeline |
+-----+-----+-----+-----+-----+-----+-----+-----+
| +ProduceResults | 0 | zona, media_consumo | 675 | 52 | 0 | 0 | 0/0 | 1,000 | |
| +Projection | 1 | z.name AS zona | 675 | 52 | 104 | 1 | 11/0 | 1,675 | In Pipeline 1 |
| +EagerAggregation | 2 | z, AVG(r.consumo_energetico) AS media_consumo | 675 | 52 | 455583 | 11384 | 1 | 1 |
| +Expand(All) | 3 | (z)-[r:ha_consumo]-(anon_1) | 455583 | 455583 | 455556 | 1 | 1 |
| +Filter | 4 | z:Feature | 53 | 53 | 994 | 1 | 1 |
| +Expand(All) | 5 | (c)-(anon_0:idvisa_ln)->(z) | 53 | 53 | 55 | 1 | 1 |
| +NodeByLabelScan | 6 | c:Città | 10 | 2 | 3 | 376 | 3623/0 | 305,320 | Fused in Pipeline 0 |
+-----+-----+-----+-----+-----+-----+-----+-----+
Total database accesses: 912215, total allocated memory: 11840
52 rows
ready to start consuming query after 688 ms, results consumed after another 355 ms
```

Figura 70: QUERY 3: Esecuzione

5.5.4 QUERY 4: Inquinanti per Zona OMI

```

1 MATCH (c:Città)-[:ha_stazione]→(s:StazioneMonitoraggio)-[r:ha_monitorato]→(i:Inquinante)
2 WITH s, i, r.valore AS valore_inquinante, s.name AS station_name, s.latitude AS latitude, s.longitude AS longitude
3 WITH point({latitude: latitude, longitude: longitude}) AS stazione_coord, s, i, valore_inquinante
4 CALL spatial.intersects('zone_omi', stazione_coord) YIELD node
5 WITH node.name AS zona_name, i.name AS inquinante_name, valore_inquinante
6 WITH zona_name, inquinante_name, collect(valore_inquinante) AS valori_inquinante
7 UNWIND valori_inquinante AS valore
8 RETURN zona_name, inquinante_name, avg(valore) AS media_inquinante

```

Figura 71: QUERY 4: Struttura

Obiettivo

L'obiettivo di questa query è calcolare la media dei valori degli inquinanti monitorati in diverse stazioni di monitoraggio, per ogni zona OMI, utilizzando informazioni geospatiali per determinare in quale zona OMI si trovano le stazioni.

Descrizione della Query

Questa query evidenzia la potenza di Neo4j nell'analisi di dati spaziali e relazionali, permettendo di aggregare informazioni geografiche e ambientali per calcolare medie degli inquinanti in base alle zone OMI. L'approccio consente di monitorare in modo accurato la qualità dell'aria in una città, con l'opportunità di visualizzare e analizzare l'inquinamento a livello geografico.

- MATCH (c: Città)-[:ha_stazione]->(s: StazioneMonitoraggio)-[r:ha_monitorato]->(i: Inquinante):
 - (**c: Città**): Identifica i nodi di tipo **Città**, che rappresentano le città in cui avviene il monitoraggio dell'inquinamento.
 - [**:ha_stazione**]: Rappresenta la relazione che collega la città alle stazioni di monitoraggio.
 - (**s: StazioneMonitoraggio**): Rappresenta le stazioni di monitoraggio, nodi che registrano i valori degli inquinanti.
 - [**r:ha_monitorato**]: Questa relazione collega ogni stazione di monitoraggio con i nodi **Inquinante**, che rappresentano le diverse tipologie di inquinamento monitorate.
 - (**i: Inquinante**): Rappresenta un inquinante specifico, come ad esempio il PM10, il NO2, ecc.
- WITH s, i, r.valore AS valore_inquinante, s.name AS station_name, s.latitude AS latitude, s.longitude AS longitude:
 - Viene selezionato il valore di inquinante misurato dalla stazione, insieme alle informazioni sul nome della stazione e sulle sue coordinate geografiche (latitudine e longitudine).

- La proprietà `valore` della relazione `ha_monitorato` contiene il valore misurato per l'inquinante, che verrà utilizzato per il calcolo della media.
- `WITH point(latitude: latitude, longitude: longitude) AS stazione_coord, s, i, valore_inquinante:`
 - La query crea un punto geospaziale a partire dalle coordinate di latitudine e longitudine della stazione di monitoraggio. Questo punto sarà utilizzato per verificare la posizione geografica della stazione all'interno di una zona OMI.
- `CALL spatial.intersects('zone_omi', stazione_coord) YIELD node:`
 - La funzione `spatial.intersects` viene utilizzata per verificare se il punto geospaziale della stazione (creato precedentemente) interseca una delle zone OMI.
 - Le zone OMI sono nodi di tipo `Feature` che rappresentano aree geografiche definite, e vengono memorizzate come poligoni nel sistema spaziale di Neo4j.
 - Se la stazione interseca una zona, viene restituito il nodo della zona, che rappresenta una zona OMI specifica.
- `WITH node.name AS zona_name, i.name AS inquinante_name, valore_inquinante:`
 - Dopo aver identificato la zona OMI in cui si trova la stazione di monitoraggio, la query seleziona il nome della zona (`zona_name`), il nome dell'inquinante (`inquinante_name`), e il valore misurato dell'inquinante (`valore_inquinante`).
- `WITH zona_name, inquinante_name, collect(valore_inquinante) AS valori_inquinante:`
 - I valori misurati per ogni inquinante nelle diverse stazioni che appartengono alla stessa zona vengono raccolti in una lista (`valori_inquinante`).
 - Questo passaggio è essenziale per poter successivamente calcolare la media di questi valori per ciascun inquinante e zona.
- `UNWIND valori_inquinante AS valore:`
 - La funzione `UNWIND` permette di "sfogliare" la lista di valori raccolti precedentemente, in modo che ogni singolo valore possa essere trattato separatamente per il calcolo della media.
- `RETURN zona_name, inquinante_name, avg(valore) AS media_inquinante:`
 - Infine, la query restituisce il nome della zona (`zona_name`), il nome dell'inquinante (`inquinante_name`), e la media dei valori misurati dell'inquinante per quella zona.

Relazioni e Struttura del Grafo

- **Nodo Città:**

- Contiene informazioni sulle città monitorate per l'inquinamento. Ogni città è collegata alle stazioni di monitoraggio attraverso la relazione [:ha_stazione].

- **Nodo StazioneMonitoraggio:**

- Rappresenta le stazioni che monitorano gli inquinanti. Ogni stazione è associata a uno o più valori di inquinamento.

- **Nodo Inquinante:**

- Rappresenta i diversi tipi di inquinamento, come PM10, CO2, NO2, ecc. Le stazioni monitorano questi inquinanti, e i valori sono memorizzati sulle relazioni [:ha_monitorato].

- **Relazione [:ha_stazione]:**

- Collega una città alle stazioni di monitoraggio. Ogni stazione è legata a una città, e ogni città può avere più stazioni.

- **Relazione [:ha_monitorato]:**

- Collega una stazione a un inquinante. La relazione contiene i valori misurati dell'inquinante dalla stazione di monitoraggio.

- **Funzione Geospaziale spatial.intersects:**

- Questa funzione è utilizzata per determinare se una stazione di monitoraggio si trova all'interno di una zona OMI, confrontando la posizione della stazione con la geometria della zona OMI.

Cypher 5								
Planner COST								
Runtime PIPELINED								
Runtime version 5.25								
Batch size 1024								
Operator Id details								
+ProduceResults 0 zona_name, inquinante_name, media_inquinante			Estimated Rows	Rows	DB Hits	Memory (Bytes)	Page Cache Hits/Misses	Time (ms)
+EagerAggregation 1 zona_name, inquinante_name, avg(valore) AS media_inquinante			68	41	0	0	0/0	1,111 In Pipeline 3
+Unwind 2 valore_inquinante AS valore			4008	30235	0	893740		
+EagerAggregation 3 zona_name, inquinante_name, collect(valore_inquinante) AS valori_inquinante			468	41	0	871736		
+Projection 4 node.name AS zona_name, cache[i.name] AS inquinante_name			211645	30235	60478			
+Proceduralcall 5 spatial.intersects(\$autostring_0, stazione.coord) :: (node :: NODE)			211645	30235		0		
+Projection 6 point((latitude, longitude, longitude)) AS stazione.coord			21165	30235				
+Projection 7 cache[s.latitude] AS latitude, cache[s.longitude] AS longitude, cache[s.name] AS station_name,			21165	30235	30205			
+Projection 8 cache[i.name]			21165	30235	66440			
+Filter 9 :Inquinante			21165	30235	66440			
+Expand(All) 10 (s)-[r:ha_monitorato]->(c)			21165	30235	30205			
+CacheProperties 11 cache[s.latitude], cache[s.longitude], cache[s.name]				7	7	78		
+Filter 12 c:citta				7	7	42		
+Expand(All) 13 (s)-[:ha_stazione]->(c)				7	7			
+NodeByLabelScan 14 s:StazioneMonitoraggio				18	7	376	2133914/0	82786,995 Fused in Pipeline 0

Total database accesses: 241880 + 2, total allocated memory: 911132
41 rows ready to start consuming query after 997 ms, results consumed after another 82786 ms

Figura 72: QUERY 4: Esecuzione

5.6 Osservazioni

Anche se abbiamo aumentato i parametri di memoria di Neo4j, non siamo riusciti a caricare tutti i dati che avevamo in MongoDB, e questa difficoltà può essere attribuita a diversi fattori legati alle differenze architetturali e implementative tra i due database, in particolare, per quanto riguarda la base su cui ciascuno è costruito: **Neo4j** è scritto in **Java**, mentre **MongoDB** è scritto in **C++**.

5.6.1 Le differenze tra Neo4j e MongoDB

1. Linguaggi di Programmazione e Performance:

- **MongoDB (C++)**: MongoDB è scritto in **C++**, un linguaggio compilato noto per le sue alte performance, specialmente in operazioni ad alta intensità di I/O e gestione dei dati. C++ permette una gestione diretta della memoria e una gestione più efficiente delle risorse hardware a livello di basso livello. Questo si traduce in una gestione molto performante di grandi volumi di dati, specialmente quando si tratta di scritture e letture non strutturate.
- **Neo4j (Java)**: Neo4j, d'altra parte, è scritto in **Java**, che è un linguaggio di programmazione più astratto rispetto al C++. Java è gestito dalla Java Virtual Machine (JVM), che introduce una certa quantità di overhead in termini di performance. La JVM gestisce la memoria dinamicamente attraverso il garbage collector, il che può comportare rallentamenti nelle operazioni di lettura e scrittura rispetto a MongoDB. Anche se Java è molto ottimizzato e offre buone performance, non raggiunge la stessa efficienza di C++ per operazioni ad alte prestazioni e gestione della memoria a basso livello.

2. Architettura e Tipologia di Dati:

- **MongoDB**: MongoDB è un database **NoSQL** orientato ai documenti e progettato per essere altamente scalabile, specialmente per archiviare e accedere a **dati non strutturati**. I suoi meccanismi di memorizzazione e indicizzazione sono particolarmente efficienti per dati che non richiedono una struttura complessa, come documenti JSON o BSON.
- **Neo4j**: Neo4j, essendo un database **grafico**, è progettato per rappresentare e gestire relazioni complesse tra i dati. Ogni nodo e relazione deve essere rappresentato come un'entità separata nel grafo. Sebbene Neo4j sia estremamente potente nell'eseguire **query complesse sui grafi**, la scrittura e la lettura massiva di nodi e relazioni, in particolare quando questi sono interconnessi in modo complesso, richiede un overhead maggiore rispetto alla gestione di documenti indipendenti.

3. Ottimizzazione per il Caricamento Dati:

- **MongoDB:** MongoDB è ottimizzato per caricare grandi volumi di dati rapidamente, soprattutto grazie alla sua architettura basata su **documenti JSON-like**. La struttura dei dati in MongoDB è altamente flessibile, e il caricamento massivo di dati è un'operazione che il sistema gestisce con molta efficienza.
- **Neo4j:** In Neo4j, invece, il caricamento di dati su una struttura a grafo (dove i dati sono rappresentati come nodi, relazioni e proprietà) richiede più risorse computazionali. Le operazioni di **traversamento del grafo** e la gestione delle relazioni tra i nodi sono complesse rispetto a un modello documentale come quello di MongoDB. Inoltre, la **scrittura delle relazioni** può comportare più overhead, soprattutto se si sta caricando una quantità massiccia di dati che devono essere correttamente relazionati tra loro.

4. Gestione della Memoria:

- **MongoDB:** MongoDB è progettato per essere molto efficiente in operazioni di lettura e scrittura su grandi set di dati. L'uso di C++ consente un controllo diretto della memoria, riducendo l'overhead e permettendo l'elaborazione più rapida dei dati. Inoltre, MongoDB usa il sistema di **memory-mapped files**, che migliora ulteriormente l'efficienza di accesso ai dati.
- **Neo4j:** Pur avendo configurazioni avanzate di gestione della memoria, Neo4j può comunque subire un certo overhead rispetto a MongoDB. Java è progettato per la portabilità e la gestione automatica della memoria, ma questo viene a scapito di un controllo più diretto. Il garbage collector della JVM, che gestisce la memoria, può introdurre pause occasionali durante l'elaborazione, rallentando operazioni particolarmente intensive come il caricamento massivo di dati. Inoltre, la gestione della memoria in Neo4j dipende molto dalla configurazione di **heap memory** e **page cache**, e se queste non sono ottimizzate correttamente, le prestazioni possono degradare, soprattutto su set di dati molto grandi.

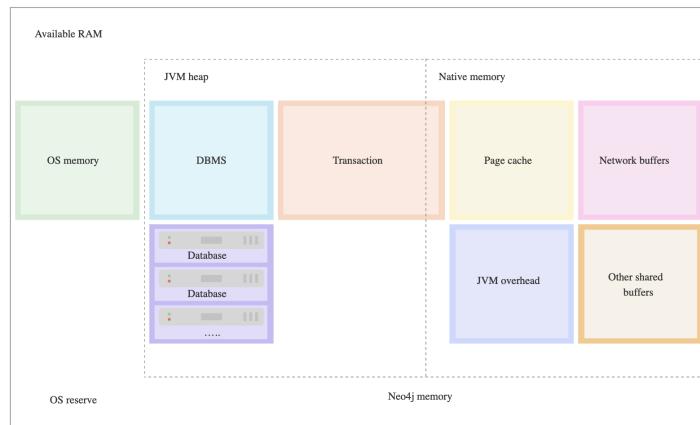


Figura 73: La gestione della memoria in *Neo4j*

5.6.2 Impatto delle differenze sulle performance di caricamento dati

Nonostante l'ottimizzazione apportata alla memoria della JVM, abbiamo: aumentato la dimensione della **page cache**, della **heap memory** e fatto puntare il DBMS in una partizione formattata con XFS [vedi Figura 74], la differenza intrinseca tra i due sistemi ha un impatto significativo sul processo di caricamento dei dati:

- **Neo4j:** Con la sua struttura grafica e la gestione complessa delle relazioni tra i nodi, Neo4j richiede una gestione accurata della memoria per evitare che il sistema venga rallentato. Inoltre, i **garbage collector di Java** possono introdurre pause nel flusso di lavoro, impedendo al sistema di gestire un volume molto elevato di dati in modo fluido.
- **MongoDB:** Grazie al suo modello più semplice e alla scrittura ottimizzata per set di dati non relazionati, MongoDB riesce a caricare grandi quantità di dati con minor overhead. In più, la gestione diretta della memoria in **C++** consente di sfruttare meglio le risorse hardware, risultando più veloce durante l'importazione massiva di dati.

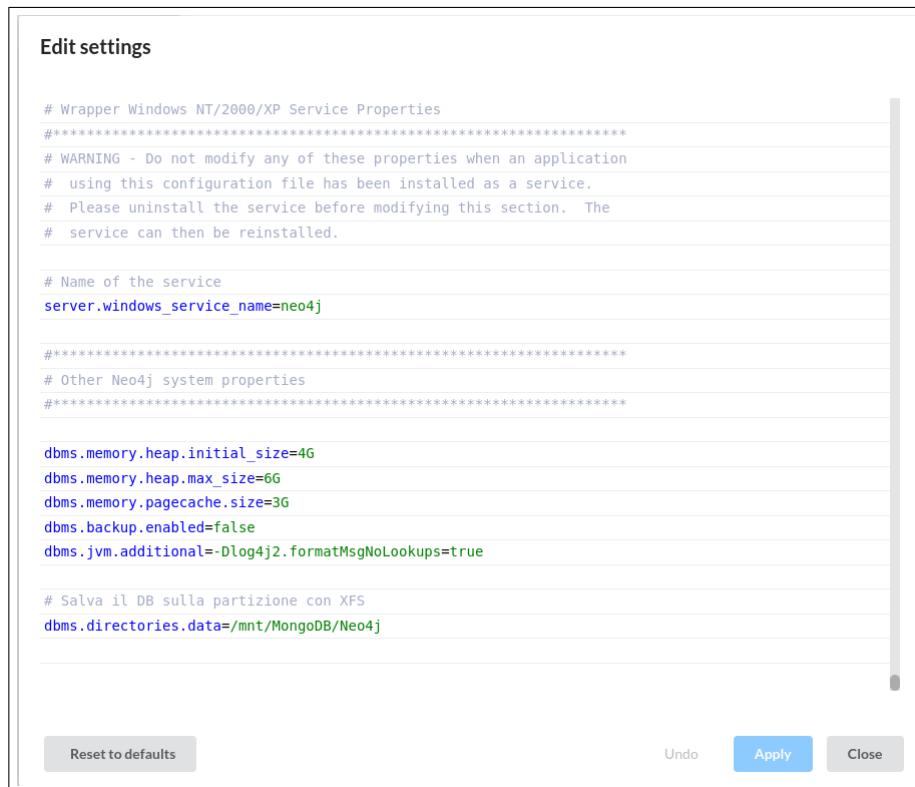


Figura 74: Personalizzazione dei parametri della JVM

In sintesi, nonostante l'aumento dei parametri di memoria in Neo4j, non siamo riusciti a caricare tutti i dati provenienti da MongoDB principalmente a causa delle differenze nell'architettura sottostante dei due sistemi. MongoDB, scritto in C++, è più adatto per operazioni di caricamento rapido di dati non strutturati e riesce a sfruttare meglio le risorse hardware. Neo4j, pur essendo estremamente potente per l'analisi dei dati grafici, soffre un po' di più quando si tratta di caricare grandi volumi di dati complessi a causa della gestione della memoria in Java e della sua struttura di dati grafici più complessa.

6 Confronto Prestazionale

Uno dei punti di forza di Neo4j rispetto a MongoDB è la gestione nativa delle relazioni. In MongoDB, le query che coinvolgono relazioni complesse richiedono spesso l'uso di più join o pipeline di aggregazione, che possono risultare costosi in termini di prestazioni, specialmente quando si lavora con dataset di grandi dimensioni. Questi approcci, infatti, non sono ottimizzati per gestire la navigazione tra dati fortemente correlati.

Al contrario, Neo4j è un database a grafo progettato proprio per gestire e ottimizzare le relazioni tra i dati. La sua struttura a grafo permette di attraversare facilmente le relazioni, riducendo significativamente il costo delle query. In Neo4j, le relazioni tra i nodi (che rappresentano entità) sono trattate come primi cittadini, il che significa che l'accesso e la navigazione attraverso queste relazioni è molto più rapido ed efficiente rispetto a database basati su tabelle o documenti come MongoDB.

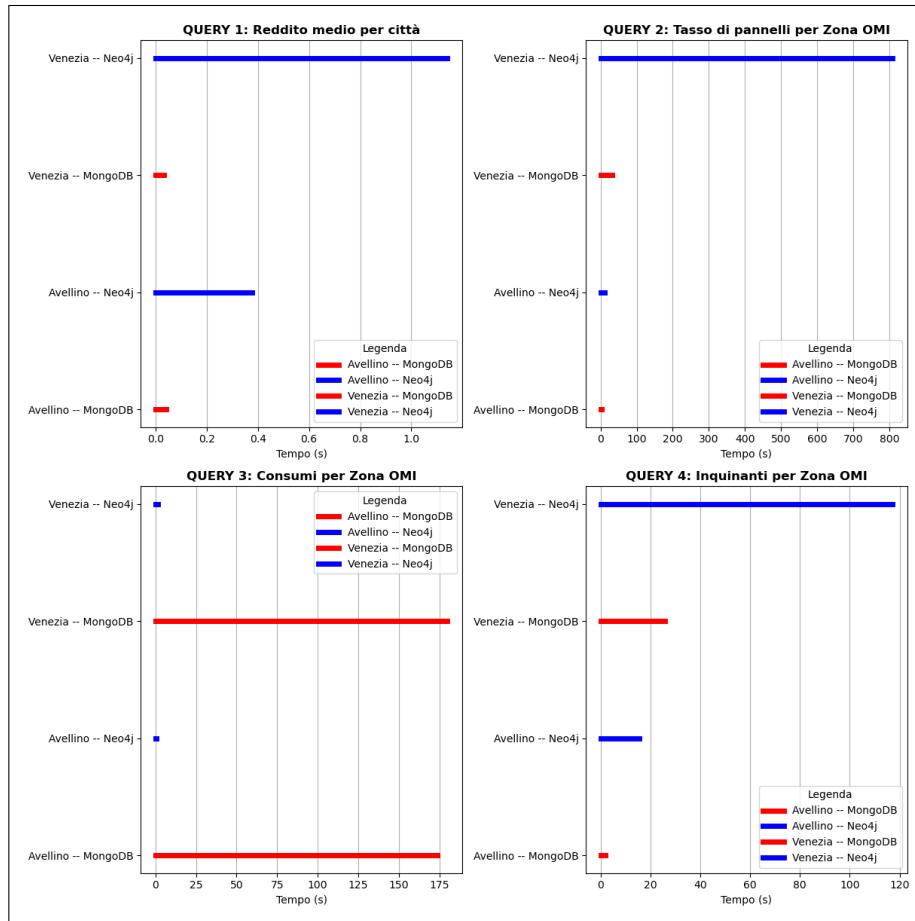


Figura 75: Prestazioni nell'esecuzione delle query: *MongoDB vs Neo4j*

In un confronto diretto, **MongoDB** è risultato **ordini di grandezza più veloce rispetto a Neo4j su 3 query su 4**, grazie alla sua ottimizzazione per operazioni di lettura su grandi volumi di dati non correlati.

Tuttavia, in una query particolarmente complessa che richiedeva **5 stage e aggregazioni tra collezioni diverse**, MongoDB è stato terribilmente più lento. Questo perché MongoDB doveva eseguire numerosi passaggi di aggregazione, ognuno dei quali richiedeva l'elaborazione di collezioni separate e la combinazione dei risultati. In Neo4j, invece, questo stesso processo è stato eseguito in un istante grazie alla sua natura di database a grafo, dove le relazioni tra i nodi sono già intrinsecamente connesse, e il traversamento dei dati è molto più veloce ed efficiente.

In quella particolare query dove MongoDB ha mostrato le sue difficoltà, una possibile ottimizzazione avrebbe potuto consistere nell' **accorpare le collezioni utilizzando documenti embedded**. Questo approccio avrebbe ridotto la necessità di effettuare più aggregazioni tra collezioni diverse, migliorando la velocità delle query.

Tuttavia, anche con questa ottimizzazione, l'efficienza del traversamento delle relazioni in Neo4j avrebbe continuato a renderlo significativamente più veloce ed efficiente per scenari complessi, dove la gestione delle relazioni è cruciale.

In breve, mentre MongoDB offre ottime prestazioni su query semplici e operazioni che non richiedono la gestione di complesse relazioni tra i dati, Neo4j eccelle in scenari complessi in cui le relazioni tra le entità sono un aspetto centrale, riducendo drasticamente il tempo necessario per operazioni che in un database documentale richiederebbero passaggi multipli e costosi.

7 Conclusioni e Sviluppi Futuri

L'adozione massiva dei **pannelli fotovoltaici** rappresenta una delle risposte più promettenti alla crescente necessità di ridurre le emissioni di gas serra e mitigare i cambiamenti climatici. Negli ultimi anni, infatti, la diffusione di questa tecnologia è stata alimentata dal continuo miglioramento delle sue performance e dalla significativa riduzione dei costi di produzione e installazione. Questi progressi hanno reso i pannelli fotovoltaici una scelta sempre più accessibile per privati, aziende e istituzioni, dando un forte impulso alla transizione verso un sistema energetico basato su fonti rinnovabili. Questa evoluzione non solo contribuisce alla sostenibilità ambientale, ma porta anche a benefici tangibili come il miglioramento della qualità dell'aria e dell'acqua, risorse fondamentali per la salute del pianeta e degli esseri viventi.

Tuttavia, nonostante i benefici evidenti, la rapida diffusione dei pannelli solari sta comportando nuove sfide, soprattutto per quanto riguarda la gestione delle **reti elettriche** e il coordinamento tra le autorità di regolamentazione. Una delle difficoltà principali è la **natura intermittente della produzione solare**. I pannelli fotovoltaici generano energia in modo variabile, influenzata dalle condizioni meteorologiche e dal ciclo giorno-notte. Questo rende difficile mantenere un equilibrio tra domanda e offerta energetica, creando una sfida per le reti di distribuzione. In aggiunta, il crescente numero di installazioni solari in punti decentralizzati può destabilizzare le infrastrutture esistenti, che non sono state progettate per gestire l'energia prodotta da una moltitudine di piccole fonti distribuite. A complicare ulteriormente la situazione, le aziende di distribuzione (es. Terna) spesso non dispongono di informazioni sufficienti sulla posizione, lo stato e le caratteristiche dei pannelli fotovoltaici installati. La mancanza di dati accurati e in tempo reale rende difficile pianificare in modo efficace l'integrazione di queste nuove risorse nel sistema elettrico, con conseguente inefficienza nella gestione della rete e rischi per la sua stabilità.

In risposta a queste sfide, il progetto mira a sviluppare un **framework basato su apprendimento automatico** per affrontare questi problemi. In particolare, uno degli obiettivi principali è il rilevamento automatico dei pannelli solari tramite l'analisi di immagini aeree ad alta risoluzione. Questa tecnologia permetterebbe di identificare e localizzare con precisione i pannelli fotovoltaici, fornendo dati essenziali per una gestione più efficiente della rete elettrica. Inoltre, il progetto prevede la raccolta di informazioni dettagliate, come le coordinate geografiche, l'area occupata dai pannelli e il numero di celle presenti, offrendo così una panoramica completa delle installazioni. Questi dati potrebbero essere utilizzati per monitorare lo stato dei pannelli e ottimizzare la gestione della rete, con benefici diretti sia per le aziende di distribuzione che per i cittadini.

Un altro aspetto positivo dell'approccio proposto riguarda l'**ottimizzazione della rete elettrica**. Con informazioni precise sulla posizione e sulla capacità di generazione dei pannelli, le aziende di distribuzione potrebbero bilanciare meglio la domanda e l'offerta di energia. Ad esempio, se un'area presenta una concentrazione elevata di pannelli solari, i gestori della rete potrebbero prevedere una maggiore produzione di energia durante le ore diurne e adattare di conseguenza la distribuzione di energia nelle altre zone. Inoltre, la **pianificazione urbana** potrebbe beneficiare di questa mappatura, permettendo di identificare le aree con alta densità di pannelli solari e pianificare future espansioni o aggiornamenti infrastrutturali in modo più mirato e efficiente.

Un altro beneficio importante riguarda la **promozione della sostenibilità**. Migliorare la gestione delle risorse energetiche rinnovabili consente di ridurre ulteriormente le emissioni di gas serra, ottimizzando la produzione di energia e riducendo gli sprechi. La gestione intelligente delle risorse solari, unita alla riduzione delle inefficienze nella distribuzione dell'energia, rappresenta un passo fondamentale verso un futuro a basse emissioni di carbonio.

Guardando al futuro, ci sono diversi sviluppi che potrebbero migliorare ulteriormente l'efficacia di questo progetto:

- **Integrazione dei pannelli fotovoltaici con sistemi IoT (Internet of Things)**, che consentirebbe di ottenere dati in tempo reale sulle prestazioni dei pannelli, permettendo un monitoraggio costante dello stato di ciascun impianto e una gestione più tempestiva e accurata delle risorse.
- **Analisi predittive avanzate**, basate su modelli di machine learning, che potrebbero prevedere la produzione energetica futura in base a variabili come le condizioni meteorologiche, la stagione e altre dinamiche ambientali, ottimizzando la gestione delle risorse energetiche.
- **Espansione del database geospaziale** per raccogliere e integrare informazioni su pannelli solari, reti di distribuzione e impatti ambientali. Un archivio globale di questi dati sarebbe utile per le politiche energetiche e la pianificazione a livello locale, nazionale e internazionale.

In conclusione, mentre l'espansione dei pannelli fotovoltaici rappresenta una componente fondamentale della transizione energetica globale, è essenziale affrontare le sfide operative e infrastrutturali che questa evoluzione comporta. Un framework tecnologico avanzato, supportato dall'intelligenza artificiale e dalle tecnologie geospaziali, non solo migliorerà l'efficienza delle reti elettriche, ma contribuirà a creare un futuro più sostenibile, rispondendo così alle necessità di un mondo in rapido cambiamento.

Indice delle Figure

1	Contenuto del Dataset	4
2	Suddivisione per tipologia	4
3	Estrazione delle Features con (<i>Random Forest</i>)	5
4	Risultati	6
5	Funzionamento della R-CNN	7
6	Risultati	8
7	Formula della <i>dice loss</i>	9
8	Confronto fra i tre approcci	9
9	Architettura di UNET++	11
10	Preprocessing del dataset	11
11	Funzionamento di UNET++	12
12	Accuracy di UNET++	12
13	Architettura di YOLOv5	13
14	Processo di conversione in PNG	14
15	Tile di 1024x1024	15
16	Tile di 256x256	15
17	Iperparametri di default	16
18	Benchmarks delle reti	17
19	Iperparametri utilizzati	17
20	Risultati Complessivi	18
21	Funzionamento di YOLOv5	18
22	Processo di Addestramento	19
23	QuickMapServices	22
24	Intersezione dei Layers	23
25	Struttura di un Cluster MongoDB	26
26	Stato dello "Sharded Cluster"	28
27	Esempio di chiave di Shard	30
28	Dimensione dei Ckunk	30
29	Visione complessiva del DB	32
30	Document esemplificativo di <i>Reddit</i> i	33
31	Document esemplificativo di <i>OMI</i>	34
32	Document esemplificativo di <i>Inquinanti Aria</i>	35
33	Document esemplificativo di <i>Consumi Elettrici</i>	37
34	Document esemplificativo di <i>Indirizzi</i>	38
35	Document esemplificativo di <i>Pannelli</i>	39
36	QUERY 1: Struttura	46
37	QUERY 1: Esecuzione senza indice	47
38	QUERY 1: Esecuzione con indice	47
39	QUERY 2: Struttura	48
40	QUERY 2: Risultato	49
41	QUERY 2: Esecuzione	49
42	QUERY 3: Struttura	50
43	QUERY 3: Risultato	51
44	QUERY 3: Esecuzione	51
45	QUERY 4: Struttura	52
46	QUERY 4: Risultato	53

47	QUERY 4: Esecuzione	53
48	Visione d'insieme della città di Avellino	56
49	Il Nodo <i>Soggetto</i>	57
50	Il Nodo <i>Zona OMI</i>	57
51	Il Nodo <i>Inquinanti Atmosferici</i>	58
52	Il Nodo <i>Consumi Elettrici</i>	58
53	Il Nodo <i>Indirizzo</i>	59
54	Il Nodo <i>Pannello</i>	59
55	Visione di dettaglio del grafo	60
56	La struttura dati " <i>R-Tree</i> "	65
57	Struttura del " <i>Layer Spaziale</i> " in Neo4j	67
58	Nodo <i>Soggetto</i> : Indici	70
59	Nodo <i>Soggetto</i> : Vincoli	70
60	Nodo <i>Consumo</i> : Indici	71
61	Nodo <i>Consumo</i> : Vincoli	71
62	Nodo <i>Indirizzo</i> : Vincoli	71
63	Nodo <i>Zona OMI</i> : Indici	72
64	Nodo <i>Zona OMI</i> : Vincoli	72
65	QUERY 1: Struttura	73
66	QUERY 1: Esecuzione	74
67	QUERY 2: Struttura	75
68	QUERY 2: Esecuzione	77
69	QUERY 3: Struttura	77
70	QUERY 3: Esecuzione	78
71	QUERY 4: Struttura	79
72	QUERY 4: Esecuzione	81
73	La gestione della memoria in <i>Neo4j</i>	83
74	Personalizzazione dei parametri della JVM	84
75	Prestazioni nell'esecuzione delle query: <i>MongoDB vs Neo4j</i>	85

Bibliografia

- [1] Vari Autori. *Articoli accademici usati per il training della rete.*
[https://github.com/AnthonyStorti/Data-Management-Systems/tree/main/Articoli%20Accademici](https://github.com/AntonyStorti/Data-Management-Systems/tree/main/Articoli%20Accademici).
- [2] Yujun Liu Hou Jiang Ling Yao. *Multi-resolution dataset for photovoltaic panel segmentation from satellite and aerial imagery* (Zenodo, 2021).
<https://zenodo.org/records/5171712>.
- [3] Malof et al. *Automatic detection of solar photovoltaic arrays in high resolution aerial imagery* (Applied Energy, 2016).
<https://www.sciencedirect.com/science/article/pii/S0306261916313009>.
- [4] Ding et al. *A light and faster regional convolutional neural network for object detection in optical remote sensing images* (ISPRS, 2018).
<https://www.sciencedirect.com/science/article/pii/S0924271618301382>.
- [5] Ding et al. *Segmentation of Satellite Images of Solar Panels Using Fast Deep Learning Model* (IJRER, 2020).
<https://www.ijrer.org/ijrer/index.php/ijrer/article/view/11607/pdf>.
- [6] QuickMapServices. *Plugin per l'acquisizione di immagini satellitari.*
<https://github.com/nextgis/quickmapservices>.
- [7] Agenzia delle Entrate. *Osservatorio del Mercato Immobiliare (Zone OMI).*
https://www1.agenziaentrate.gov.it/servizi/geopoi_omi/index.php.
- [8] Rinnovabili.it. *Le città italiane con più pannelli fotovoltaici.*
<https://www.rinnovabili.it/mercato/le-aziende-informano/fotovoltaico-piu-350-000-nuovi-impianti-ultimo-anno/>.
- [9] Il Post. *Il giorno in cui in Italia furono consultabili le dichiarazioni dei redditi di tutti (1 Maggio 2019).*
<https://www.ilpost.it/2019/05/01/dichiarazioni-redditi-pubbliche-italia/>.
- [10] Kristina Chodorow. *50 Tips and Tricks for MongoDB Developers* (Aprile 2011).
<https://www.oreilly.com/library/view/50-tips-and/9781449306779/>.
- [11] MongoDB Inc. *MongoDB with Python.*
<https://www.mongodb.com/docs/languages/python/>.
- [12] Neo4j Inc. *From MongoDB to Neo4j.*
<https://neo4j.com/labs/apoc/4.0/database-integration/mongodb/>.
- [13] MongoDB Inc. *Neo4j-Spatial.*
<https://neo4j.com/labs/spatial/>.
- [14] Wikipedia Inc. *The R-Tree.*
<https://en.wikipedia.org/wiki/R-tree>.