UNIVERSITÀ DEGLI STUDI DI MODENA E REGGIO EMILIA

Corso di Laurea in Scienze dell'Informazione

Tesi di Laurea Triennale

Progetto e sviluppo di un'applicazione Big Data per la gestione e l'analisi dei dati provenienti da Modena Automotive Smart Area

Relatore
prof. Riccardo Martoglia

Correlatore:
prof. Roberto Cavicchioli

Laureando
Francesco Barbanti

Anno accademico 2019/2020

Ringraziamenti

In primis, un ringraziamento speciale è rivolto al Professore Riccardo Martoglia, relatore di questa tesi, per la competenza e la disponibilità mostrata in questi mesi.
Un grazie sincero al correlatore di questo elaborato, il Professore Roberto Cavicchioli, per le indicazioni e i consigli pratici senza i quali questo progetto non sarebbe stato possibile.
Una dedica speciale alla mia famiglia e i miei amici, che ogni giorno mi hanno sostenuto e incoraggiato al fine di raggiungere questo traguardo.

Parole chiave

Apache Spark
Stream processing
Reverse geocoding
Big data analytics
Big data management

Indice

Elenco delle figure

Elenco delle tabelle

Elenco dei listati

Introduzione

Nel corso degli ultimi anni, l'evoluzione tecnologica ha introdotto sul mercato sempre più dispositivi in grado di generare e richiedere analisi di grandi quantità di dati. Se da un lato il valore dei dati aumenta sempre più, dall'altro accresce anche la difficoltà nella gestione degli stessi; per questo motivo, il mondo dei Big Data ha assunto un ruolo dominante all'interno dell'informatica moderna.
Progetti di ricerca e sviluppo di tecnologie Big Data sono diventati sempre più comuni, e di conseguenza, anche i campi applicativi di essi sono aumentati sempre più.
Il presente elaborato ha l'obiettivo di progettare ed implementare un applicativo per la gestione, la manipolazione e l'analisi di Big Data. Infatti, a partire dall'enorme quantità di dati generata da una serie di infrastrutture presenti all'interno di *Smart Model Area* e grazie ad un motore di elaborazione streaming, si vuole mantenere in opportune strutture il flusso dati al fine di visualizzare su di una mappa interattiva il traffico relativo alla zona predisposta.
Questa trattazione è articolata in cinque capitoli, ognuno dei quali descrive un modulo diverso del progetto. Più nello specifico, il primo capitolo ha il compito di descrivere in maniera approfondita Apache Spark, il motore di elaborazione streaming adottato in questo elaborato. Il secondo capitolo espone i principali strumenti utili alla costruzione di un reverse geocoder performante, mentre il terzo capitolo si sofferma sulla parte progettuale dell'applicativo. La quarta sezione espone la fase di implementazione con il relativo codice più significativo. A conclusione, il quinto capitolo si compone di una breve analisi delle performance del modulo di reverse geocoding ed espone una serie di screenshot relativi ai risultati ottenuti.

Parte I

Studio delle tecnologie utili per la gestione, l'analisi e l'elaborazione di Big Data

Capitolo 1

Apache Spark

In questo primo capitolo si analizzerà Apache Spark, una piattaforma di cluster computing open source per l'elaborazione di dati su larga scala. In particolare, oltre ad una breve descrizione di tutto il core di Spark, saranno descritti approfonditamente i moduli per l'elaborazione di flussi dati in tempo reale e per la manipolazione di dati strutturati tramite linguaggio SQL.

1.1 Big Data

Ai nostri giorni, la ricerca e lo sviluppo in ambito Big Data sono diventati di primaria importanza per molte aziende e università. La mole di dati generati e manipolati sta aumentando sempre più; infatti si stima che, ogni giorno, vengano generati oltre 2.5 exabyte di dati con la previsione di raggiungere i 59 zettabytes di dati creati a fine anno 2020 [1]. Parlando quindi di Big Data si fa riferimento all'enorme volume di dati, strutturati e non strutturati, che devono essere mantenuti, interrogati, analizzati e manipolati. Come si può immaginare, la gestione di una grande quantità di dati introduce non poche complicazioni, raccolte nella cosiddetta regola delle "quattro V's", che fa riferimento all'enorme volume dei dati, la velocità con cui le informazioni devono essere elaborate, la varietà dei dati e la loro veridicità (cioè il valore informativo del dato). Negli ultimi anni molte aziende hanno compreso come i Big Data rappresentino una vera e propria risorsa, infatti estraendo e analizzando le informazioni opportune da essi è possibile trarne vantaggio sia in ambito economico sia in ambito sociale. Un esempio lampante è rappresentato dalla pandemia globale causata dal virus *SARS-CoV-2*, la gestione e lo studio dei dati provenienti da tutto il mondo ha permesso di monitorare e, in qualche modo, prevedere l'andamento della curva epidemiologica per mettere in atto le dovute misure restrittive. Data la dimensione, la struttura e la complessità, i Big Data non possono essere gestiti tramite gli strumenti tradizionali, come per esempio i DBMS relazionali, ma è necessaria l'introduzione di nuove tecnologie che permettano di mantenere,

interrogare e manipolare una grande mole di dati in maniera efficiente. I principali framework che permettono la computazione distribuita di dati su cluster di computer sono MapReduce introdotto da Google e la controparte open source Apache Hadoop, descritto brevemente in seguito.

1.2 Apache Hadoop

In contemporanea con la nascita del motore di ricerca Google, Mike Cafarella e Doug Cutting danno vita al progetto Nutch, un motore di ricerca web in grado di elaborare richieste in maniera efficiente grazie alla distribuzione del calcolo e dei dati su una rete di computer. Dal momento in cui Google, nel 2004, presentò il paper "MapReduce: Simplified Data Processing on Large Clusters", il motore di ricerca Nutch introdusse la propria implementazione del modello di programmazione MapReduce e un file system distribuito esposti dalla controparte Google. Nel 2006 Doug Cutting si unì al team di Yahoo!, suddividendo così la parte di web crawling di Nutch dalla parte di distribuzione del calcolo e dei dati, che diventò così il progetto Hadoop. Apache Hadoop è un framework open source che permette l'archiviazione e l'elaborazione distribuita di una grande mole di dati su clusters di computers tramite un semplice modello di programmazione.
Le caratteristiche che hanno permesso ad Hadoop di diventare la piattaforma di riferimento nell'ambito della gestione dei Big Data sono:

- *Affidabilità*: lavorando con numerosi nodi, è probabile che uno o più di essi si guastino. Per ovviare a questo problema, Hadoop replica automaticamente ogni blocco dati su vari nodi del cluster. Nel caso in cui nodo del cluster si guasti, è possibile recuperare i dati accedendo al nodo che contiene la copia del corrispettivo blocco dati.

- *Flessibilità*: il file system su cui questo framework è implementato permette di archiviare e analizzare qualsiasi tipo di dato strutturato, semi-strutturati e non-strutturato. Infatti i dati sono semplicemente copiati all'interno del file system per poi, in futuro, essere analizzati e manipolati nella maniera più opportuna; non è necessario elaborare i dati prima di memorizzarli.

- *Scalabilità*: è possibile aggiungere o rimuovere nodi al cluster al fine da aumentarne le prestazioni senza interrompere il flusso di lavoro. A favore di ciò, questa piattaforma può essere eseguita su un cluster di commodity hardware, ovvero dispositivi a basso costo, facili da reperire e capaci di integrarsi facilmente con dispositivi diversi al fine da creare la maggior potenza di calcolo al minore costo possibile.

- *Facilità di utilizzo*: questa caratteristica ha permesso ad Hadoop di riscuotere successo anche nei confronti di utenti meno esperti. Questo perché, i framework contenuti all'interno dell'ecosistema Hadoop, non richiedono all'utente di preoccuparsi delle problematiche derivanti dal calcolo distribuito.

1.2.1 I componenti di Hadoop

Il nucleo centrale di Hadoop è composto da:

- **HDFS (Hadoop Distributed File System)**: un file system distribuito con lo scopo di archiviare una grande quantità di informazioni su un cluster di commodity hardware.
- **Apache YARN**: un framework in grado di gestire le risorse del cluster e schedulare i vari job, rappresenta il gestore delle risorse nell'ecosistema Hadoop.
- **Hadoop MapReduce**: un framework per l'elaborazione parallela di una grande quantità di dati.

Grazie alla crescente popolarità, attorno al nucleo di Hadoop sono stati incorporati molti nuovi framework, ognuno con funzionalità differenti, che hanno dato origine a un vero e proprio ecosistema, la cui struttura è descritta dalla Figura 1.1.

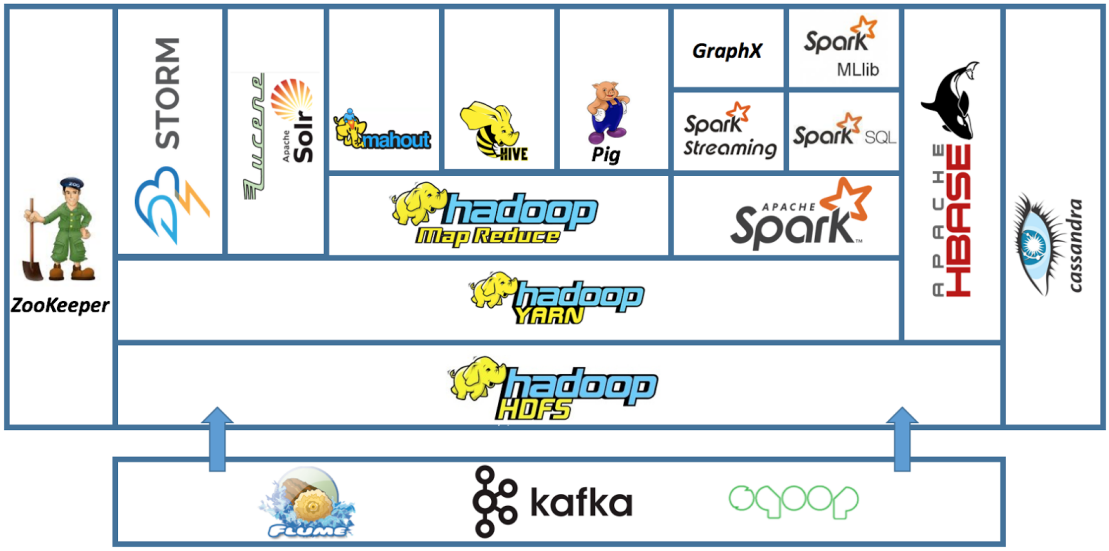

Figura 1.1: Ecosistema Hadoop

Questo elaborato si concentrerà sulla parte di elaborazione dei dati tramite il framework Apache Spark, ponendo in secondo piano la parte di archiviazione sul file system distribuito e la gestione delle risorse del cluster.

1.3 Apache Spark

Apache Spark è una piattaforma di cluster computing pensato per elaborare una grande quantità di dati in maniera efficiente, semplice e veloce. Grazie a queste caratteristiche, Spark è diventato popolare e molto utilizzato non solo in ambito di ricerca, ma anche in contesti industriali. A differenza di tutti gli altri framework presenti all'interno dell'ecosistema Hadoop, Spark non usa Hadoop MapReduce come il proprio motore di esecuzione; infatti, sfruttando la funzionalità di *in-memory processing*, è in grado di avere performance fino a 100 volte più elevate rispetto a MapReduce. Se le prestazioni di quest'ultima dipendono fortemente dalla velocità di lettura e scrittura sul disco, Spark prova quando possibile a mantenere i dati in memoria principale, con la conseguenza di abbassare notevolmente i costi di accesso ai dati. Entrambi i framework mantengono le caratteristiche di tolleranza e scalabilità introdotte dall'ecosistema Hadoop, con la differenza che Spark presenta un'API più semplice e completa rispetto al paradigma MapReduce. Inoltre, tramite la creazione di *DAG* (Directed Acyclic Graph) [2], Spark è in grado di eseguire sequenze di operazioni in un singolo job, aumentando così le performance e diminuendo la complessità del codice applicativo.
Data la diversa natura dei due framework, essi si rivolgono ad ambiti applicativi differenti. MapReduce è indicato per eseguire operazioni su dataset di dimensioni maggiori della memoria centrale a disposizione e per contesti in cui non è richiesta un'elaborazione dei dati in tempi brevi. Grazie alla presenza di più moduli all'interno di esso, Apache Spark è indicato per numerosi contesti applicativi come machine learning, graph processing, real-time processing e iterative processing. Infatti, questa piattaforma si rivolge ad applicazioni in cui i risultati intermedi sono sfruttati in più computazioni successive, come per esempio algoritmi iterativi o query interattive sullo stesso sottoinsieme di dati. La Figura 1.2 e Figura 1.3 mostrano rispettivamente la modalità di esecuzione di query interattive in Apache Spark e MapReduce, mettendo in evidenza il diverso uso della memoria.

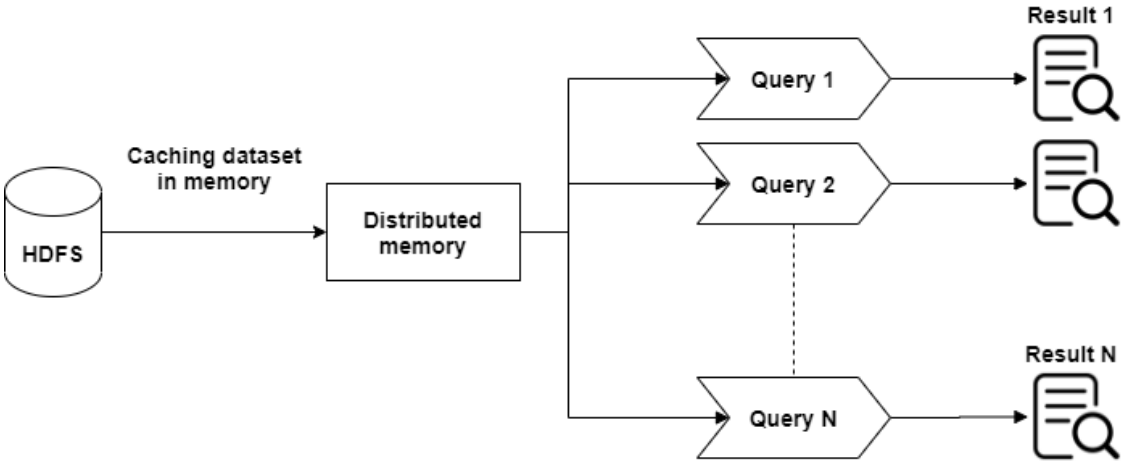

Figura 1.2: Interactive query su Apache Spark

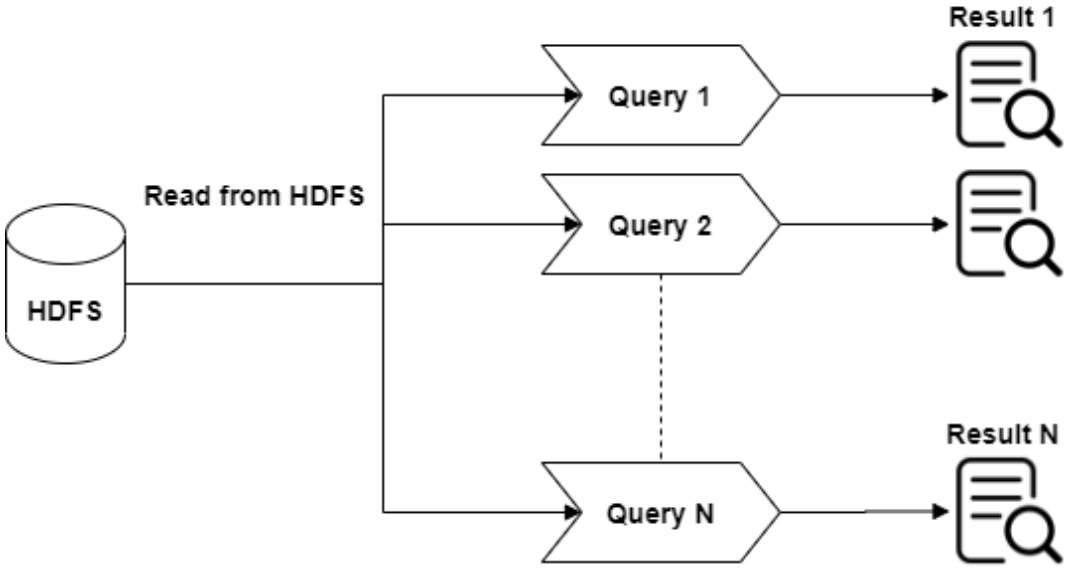

Figura 1.3: Interactive query su MapReduce

1.3.1 Storia di Spark

Come già anticipato, il paradigma di programmazione MapReduce introduce notevoli limitazioni nel caso di algoritmi iterativi e query interattive. Per ovviare a questo problema, nel 2009 un gruppo di ricercatori dell'Università della California Berkeley AMPLab sviluppò il core del progetto Spark, per poi donarlo nel 2013 all'Apache Software Foundation. Attualmente il progetto è distribuito sotto la licenza Apache 2.0, dopo che nel 2010 fu rilasciato in modalità open source con licenza BSD. In concomitanza con il successo di questo framework, una gruppo di ricercatori che già lavoravano sul progetto Spark fondò l'ormai famosa Databricks, una compagnia che fa di Apache Spark il proprio business.

1.3.2 Ecosistema di Spark

Come mostrato in Figura 1.4, Spark presenta al suo interno vari moduli come Spark SQL, Spark Streaming, Mlib, GraphX e Spark Core, ma non possiede un proprio file system e un proprio cluster manager. Questo significa che si può integrare con vari tipi di file system distribuiti come quello proprio di Hadoop (HDFS), Cassandra, Amazon S3 (servizio di memorizzazione offerto da Amazon) e molti altri, oltre a supportare diversi tipi di cluster manager (come Apache Mesos o Apache Yarn).

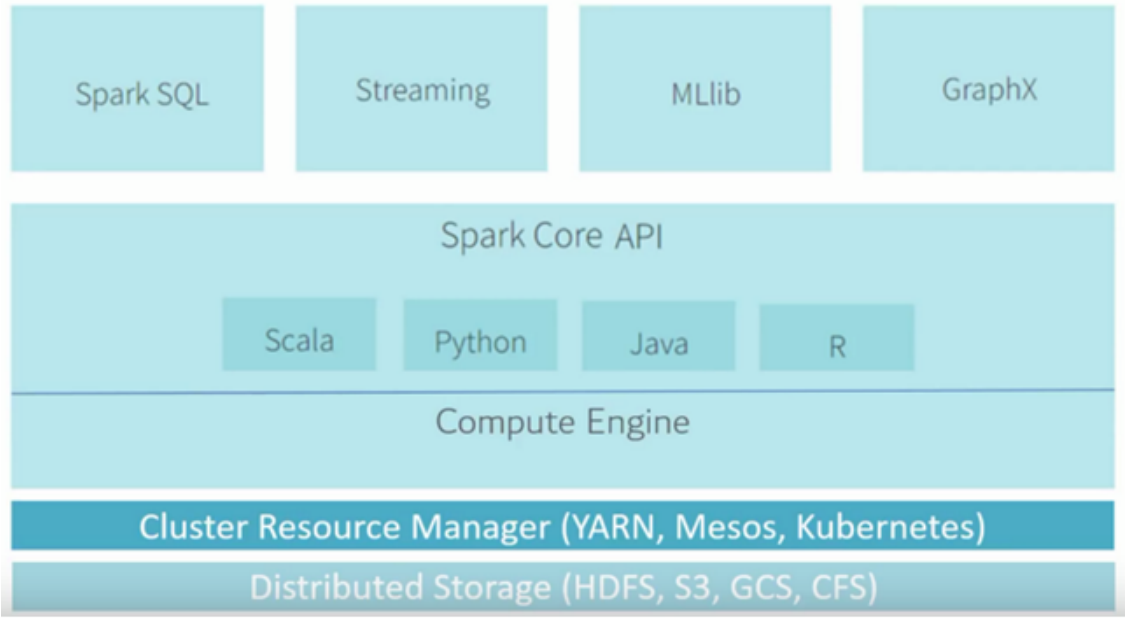

Figura 1.4: Ecosistema Apache Spark

1.3.3 Spark Unified Stack

A differenza di molti altri motori di elaborazione dati, Spark mantiene i propri componenti all'interno di un unico stack, chiamato *Spark unified stack*, costruito intorno al modulo Spark Core. Grazie a questa organizzazione, è possibile sfruttare i risultati di uno o più componenti dello stesso stack per creare un numero di combinazioni teoricamente infinite senza l'ausilio di motori di elaborazioni esterni. Per esempio, non si ha difficoltà ad elaborare un flusso dati in tempo reale tramite il modulo Spark Streaming per poi eseguire algoritmi di machine learning sul flusso dati elaborato precedentemente grazie alla libreria Mllib. Eseguendo su un solo motore di elaborazione, è quindi molto facile implementare applicazioni che sfruttino più moduli dello stesso stack.
Lo stack di Spark è composto dai seguenti componenti:

- **Spark Core**: rappresenta il cuore di Spark e ha il compito di creare una piattaforma di esecuzione per le applicazioni Spark. Fornisce gli strumenti per il controllo e la distribuzione dei task alle diverse macchine del cluster oltre che per la gestione di tutte le interazioni con il file system distribuito, ponendo particolare attenzione al mantenimento dei dati in memoria centrale (quando possibile). Introducendo un sistema di calcolo distribuito, è necessario considerare le varie complicazioni che in un sistema non distribuito non si presentavano; Spark Core si occupa anche del fenomeno di *data shuffling*, ovvero è in grado di ridistribuire dati alle macchine del cluster in caso di fallimento di un nodo. Inoltre, questo modulo è in grado di rappresentare l'unità di dato di Spark tramite l'API Resilient Distributed Datasets (RDDs), che verrà descritta in seguito.

- **Spark SQL**: un modulo che permette di elaborare ed eseguire interrogazioni su dati strutturati scrivendo codice in linguaggio SQL (Structured Query Language). Per aumentare le performance, gli sviluppatori di Spark hanno introdotto un query optimizer di nome *Catalyst optimizer* in grado di eseguire la query dell'utente nel modo più efficiente possibile. Al fine da ottenere un'esperienza utente migliore, Spark SQL introduce altre due astrazioni, DataFrames e Datasets, contenuti all'interno dell'API DataFrames. Inoltre questo modulo supporta nativamente operazioni di scrittura e lettura su vari tipi di file, tra cui JSON, CSV, ORC, Parquet e molti altri.

- **Spark Streaming**: un modulo che permette di elaborare flussi di dati in tempo reale. In particolare, per rappresentare il flusso dati introduce una nuova astrazione chiamata DStream (discretized stream), la quale rappresenta semplicemente una sequenza di RDDs. Internamente Spark Streaming suddivide il flusso di dati in ingresso in vari lotti (*batches* in inglese), che poi verranno trattati singolarmente come istanze di RDD e distribuiti ai vari nodi del cluster per elaborali. Come si evince dalla documentazione ufficiale di Spark [3],

questo componente supporta vari tipi di sorgenti data, quali HDFS, S3, Kafka, Twitter, Flume, Kinesis e sockets TCP. In questo elaborato si analizzerà anche Spark Structured Streaming, un modulo introdotto da Spark 2.1 che permette, oltre alle funzioni di Spark Streaming, di processare il flusso dati in ingresso basandosi su eventi temporali. In questo elaborato si analizzerà anche Spark Structured Streaming, un modulo introdotto da Spark 2.1 che permette, oltre alle funzioni di Spark Streaming, di processare il flusso dati in ingresso basandosi su eventi temporali.

- **Mllib**: un modulo che sfrutta l'API DataFrames per eseguire algoritmi di machine learning. Sfruttando il Catalyst optimizer proprio di Spark SQL e il mantenimento dei dati in memoria centrale, gli algoritmi di machine learning (che di natura sono iterativi) hanno tempi di risposta ridotti.
- **GraphX**: un modulo che permette di manipolare e analizzare grafi di grandi dimensioni su un sistema di calcolo distribuito tramite una computazione parallela. Al suo interno include i classici algoritmi di analisi dei grafi, come algoritmi per la ricerca dei cammini minimi o per il calcolo delle componenti connesse.
- **SparkR**: un modulo che permette di eseguire operazioni su un grande volume di dati tramite il linguaggio di programmazione R.

1.3.4 Esecuzione di un'applicazione Spark

Lanciando un programma Spark è possibile scegliere fra due modalità di esecuzione: *Client Mode* e *Cluster Mode*. La modalità Client permette di eseguire Spark in locale, non servendosi quindi del cluster ma delle risorse di una singola macchina. Molto più interessante è l'esecuzione in Cluster Mode (Figura 1.5), nella quale Spark fa uso di un cluster manager per gestire i nodi della rete. Il gestore di risorse è in grado di suddividere il carico di lavoro ai vari nodi della rete, chiamati *worker*, conoscendone la locazione e le risorse che possono offrire.
Spark segue un'architettura *master-slave*, in cui il master è rappresentato dal *driver* (coordinatore) e gli slave dagli *worker* (lavoratori). Il driver è un processo che ha il compito di suddividere la computazione in una serie di task, che poi verranno assegnati ai vari nodi della rete. Inoltre, tramite il cluster manager, lancia su ogni worker del cluster un processo chiamato *executor*. Ogni esecutore è un processo di una JVM (Java Virtual Machine), il quale ha il compito di eseguire il codice inviato dal driver e memorizzare in memoria centrale o secondaria i dati dell'applicazione. Come riportato nella documentazione ufficiale di Spark, ogni applicazione possiede i propri processi esecutori. La conseguenza è che diverse applicazioni non possono condividere lo stesso esecutore, e questi ultimi possono condividere dati solo scrivendoli prima sul disco. In questo modo il comportamento scorretto di un'applicazione non influisce sulle altre applicazioni Spark.

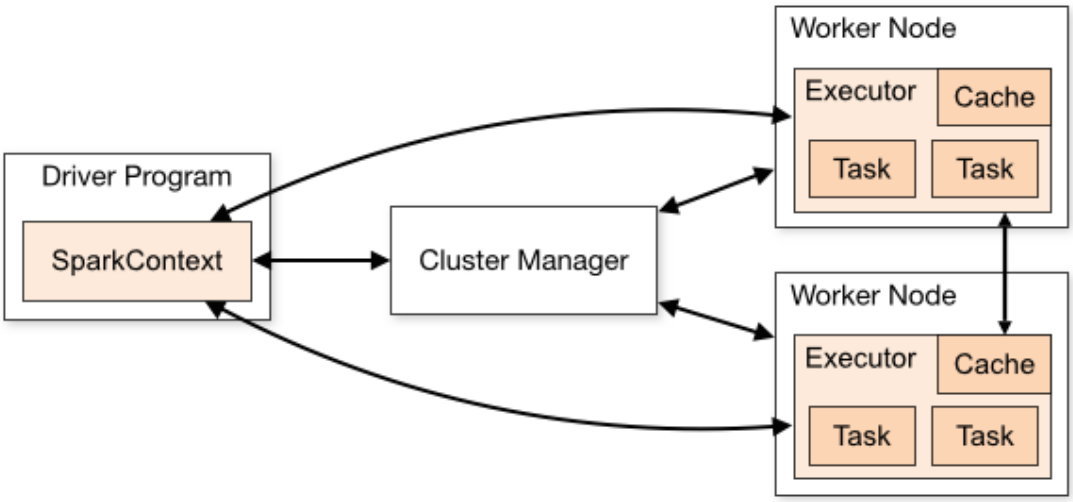

Figura 1.5: Spark in Cluster Mode

Come si può notare nella Figura 1.5, il driver è responsabile di un'istanza della classe di nome *SparkContext*, la quale ha il compito di comunicare con il cluster manager e i vari esecutori.

1.3.5 Resilient Distributed Datasets

Come già anticipato, l'obiettivo di Spark è quello di offrire elevate performance di calcolo per algoritmi iterativi e per analisi interattive su un grandi volumi di dati, grazie al mantenimento dei dati in memoria centrale durante le operazioni su di essi. Per raggiungere questo obiettivo, mantenendo le caratteristiche di flessibilità, semplicità e affidabilità, Spark fa uso dell'astrazione RDDs (Resilient Distributed Datasets). Essi costituiscono l'unità di dato di Spark, dove ogni RDD è una collezione di elementi distribuiti sui nodi di un cluster. Rappresentano la struttura dati fondamentale di Spark, grazie alla quale è possibile distribuire i dati ai vari nodi del cluster ed Figure 5: Spark in Cluster Mode eseguire operazioni parallele su di essi. Per distribuire un singolo RDD alle macchine della rete, la piattaforma suddivide l'astrazione in una serie di partizioni per poi memorizzarla nei nodi worker. Per natura gli RDDs sono strutture dati immutabili e tolleranti agli errori. Sono immutabili in quanto, ogni volta che un RDD viene alterato, Spark restituisce un nuovo Resilient Distributed Dataset contenente le modifiche effettuate senza alterare l'RDD di partenza. Questa caratteristica degli RDDs ne permette una condivisione sicura tra i vari processi in esecuzione.
Come riportato nel paper "*Spark: Cluster Computing with Working Sets*" rilasciato dall'Università della California [4], grazie a questa astrazione i tempi di risposta a

interrogazioni effettuate su dataset di 39 GB sono frazioni del secondo e le performance su algoritmi di machine learning sono fino a 10 volte più elevate rispetto al tradizionale Hadoop.

1.3.6 Creazione di RDD

Esistono due modi per istanziare un RDD: parallelizzare una collezione di oggetti presenti nel driver, ovvero trasformare la collezione in un dataset distribuito, oppure creare un RDD a partire dalle informazioni contenute in un dataset presente su uno storage esterno. Il codice 1.1 mostra la creazione di un RDD a partire da una collezione di oggetti. Si noti l'uso del metodo *parallelize* della classe *SparkContext*, il quale permette di distribuire una copia della collezione ai vari nodi del cluster.

```
1 object = [10, 20, 30]
2 rdd = sc.parallelize(object)
```

Listato 1.1: Creazione di un RDD a partire da una lista di oggetti

1.3.7 Operazioni sugli RDD

Le operazioni che Spark offre per la manipolazione degli RDDs operano tutte a livello di grana grossa (*coarse-grained level*). Questo significa che sono consentite solamente trasformazioni che applicano le stesse operazioni ad un insieme di dati, non permettendo di l'esecuzione di operazioni su una specifica riga del dataset. In questo modo Spark è in grado di costruire la cosiddetta *lineage* (in italiano stirpe) di ogni RDD, ovvero uno storico delle operazioni eseguite sullo stesso con le relative dipendenze (Figura 1.6).

La lineage è composta da:

- Le funzioni applicate al particolare RDD
- Un grafo contente tutte le dipendenze dello stesso RDD

Grazie alle informazioni contenute nella lineage, Spark è in grado di rendere gli RDDs tolleranti agli errori e determinarne l'ordine di esecuzione.
Spark offre un'API molto ricca per la manipolazione dei RDDs, in particolare si suddividono le operazioni su di essi in due categorie:

- *Actions*: restituiscono un risultato al programma driver o lo memorizzano nel file system distribuito dopo aver eseguito un'operazione sul dataset.
- *Transformations*: creano un nuovo RDD a partire da RDD preesistente. Un esempio di trasformazione è il metodo *filter()*, il quale ritorna un nuova struttura dati a partire da una selezione di elementi di un RDD di partenza.

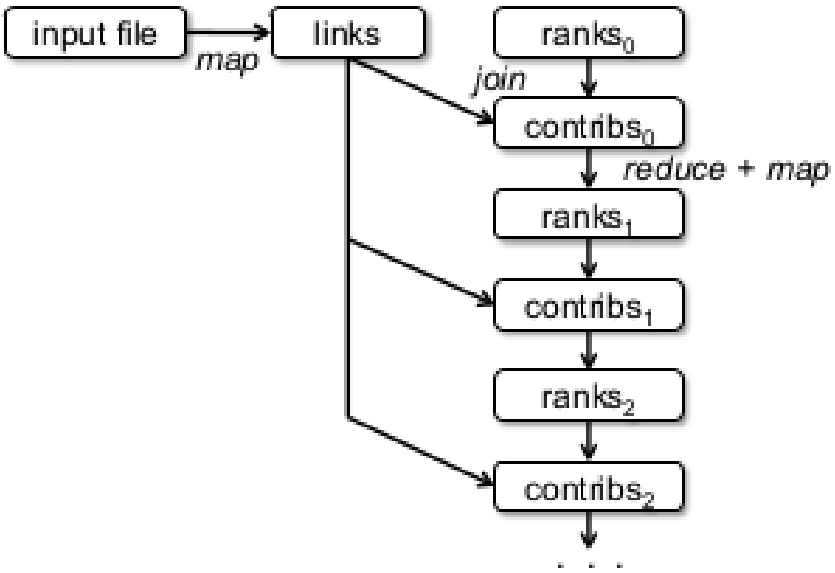

Figura 1.6: Grafo della Lineage per un dataset nell'algoritmo di PageRank

La vera differenza tra i due tipi di operazioni è che le trasformazioni sono processate in modalità *lazy*, ovvero Spark ritarda la trasformazione fino a che non è chiamata un'azione sugli RDD interessati, mentre le azioni sono processate in modalità *eager*, in quanto l'invocazione di una di esse attiverà immediatamente l'esecuzione di tutte le trasformazioni in sospeso. Questo comportamento permette a Spark di introdurre ottimizzazioni e di conseguenza ridurre il tempo di esecuzione delle trasformazioni. Per esempio, Spark considera la funzione *map()* come una trasformazione, mentre *reduce()* è un'azione. Questo perché, dopo aver creato un dataset con la funzione map e manipolato con la funzione reduce, al programma driver viene restituito solamente il dataset modificato e non quello di partenza. La tabella 1.1 mostra le trasformazioni più comuni, mentre la tabella 1.2 mostra le azioni più comuni presenti nell'API RDD.

Tabella 1.1: Trasformazioni RDD

Transormation	**Significato**
map*(func)*	Ritorna un nuovo RDD mappando ogni elemento della sorgente tramite la funzione *func()*

Tabella 1.1: Trasformazioni RDD

Transormation	**Significato**
filter*(func)*	Ritorna un nuovo dataset selezionando dalla sorgente dati gli elementi che soddisfano la condizione espressa nella funzione *func()*
flatMap*(func)*	Simile a *map*, ma ogni elemento viene mappato in zero o più elementi sul dataset di uscita
mapPartitions*(func)*	Simile a *map*, con la differenza che la funzione *func* viene eseguita su ogni partizione dell'RDD di partenza
union*(otherDataset)*	Ritorna un nuovo dataset generato dall'unione di due RDD di partenza
sample*(withReplacement, fraction, seed)*	Ritorna un sottoinsieme dell'RDD di partenza usando una funzione casuale avente come seme l'attributo *seed*
intersection*(otherDataset)*	Ritorna un nuovo dataset generato dall'intersezione di due RDD di partenza
distinct*([numPartitions])*	Ritorna un nuovo dataset eliminando gli elementi duplicati da un RDD di partenza
cartesian*(otherDataset)*	Ritorna un nuovo dataset contenente il prodotto cartesiano di due RDD di partenza
join*(otherDataset, [numPartitions])*	Dati due dataset di coppie chiave-valore (K,V), viene restituito un nuovo dataset di coppie chiave-valore generato tramite l'operazione di join tra i due dataset di partenza
reduceByKey*((func, [numPartitions])*	Dato un dataset di coppie chiave-valore (K,V) viene restituito un nuovo dataset di coppie chiave-valore (K,V) in cui i valori con la stessa chiave vengono aggregati tramite la funzione *func*

Tabella 1.1: Trasformazioni RDD

Transormation	Significato
groupByKey*([numPartitions])*	Dato un dataset di coppie chiave-valore (K,V) viene restituito un nuovo dataset di coppie chiave-valore in cui vengono raggruppati i valori con la stessa chiave in una lista

Tabella 1.2: Azioni RDD

Action	Significato
reduce*(func)*	Data una funzione *func* associativa e commutativa, restituisce un nuovo dataset creato aggregando gli elementi del dataset di partenza
collect	Ritorna tutti gli elementi di un dataset sotto forma di array al programma driver
count	Restituisce il numero di elementi in un dataset
first	Restituisce il primo elemento di un dataset
take*(n)*	Restituisce un array contenente i primi n elementi del dataset di partenza
foreach(func)	Esegue la funzione *func* su ogni elemento del dataset
saveAsTextFile*(path)*	Salva gli elementi del dataset in un uno o più file testuali nel file system locale o un file system distribuito supportato da Hadoop
saveAsSequenceFile*(path)*	Salva il contenuto del dataset come *SequenceFiles*, cioè un file di tipo binario utilizzato nell'ecosistema Hadoop

1.3.8 Persistenza degli RDD

Per aumentare le performance degli algoritmi iterativi, Spark dà la possibilità di rendere gli RDD persistenti in memoria a tutti gli esecutori del cluster. Questo meccanismo permette di memorizzare i risultati intermedi delle operazioni sugli stessi, evitando così di doverli ricalcolare in futuro. Esistono due metodi rendere persistente un RDD: *cache()* e *persist()*. Il primo metodo è una scorciatoia che consente di salvare la struttura dati in memoria (*MEMORY_ONLY*), mentre con

il secondo l'utente ha la possibilità di specificare il livello di memoria in cui salvare le partizioni. I livelli di memorizzazione che un utente può specificare sono:

- *DISK_ONLY*: la struttura dati viene memorizzata nel disco
- *MEMORY_ONLY*: la struttura dati viene salvata in memoria. Nell'eventualità in cui la dimensione dell'RDD superi quella a disposizione dal nodo della rete, alcune partizioni non saranno *cachate* in memoria ma ricalcolate in caso di bisogno
- *MEMORY_ONLY_SER*: la struttura dati viene serializzata e poi memorizzata in memoria. Serializzando un RDD si riduce lo spazio occupato a scapito di un costo di lettura dello stesso più elevato. Questo livello di memoria è disponibile solo per il linguaggio Java e Scala, ma non per Python.
- *MEMORY_AND_DISK*: la struttura dati viene salvata in memoria. Se la dimensione dell'RDD supera quella della memoria a disposizione, le partizioni in eccesso vengono salvate sul disco fisso.
- *MEMORY_AND_DISK_SER*: la struttura dati viene serializzata e poi salvata in memoria. Se la dimensione dell'RDD supera quella della memoria a disposizione, le partizioni in eccesso vengono salvate sul disco fisso.

Poichè i metodi *cache()* e *persist()* sono delle trasformazioni, un RDD non viene reso persistente in memoria fin tanto che non è richiamata un'azione sullo stesso. Per evitare di terminare la quantità di memoria libera nel cluster, Spark rimuove automaticamente le partizioni degli RDD usati meno di recente. In particolare, se il livello di memoria comprende la possibilità di salvataggio su disco (MEMORY_AND_DISK, MEMORY_AND_DISK_SER), le partizioni vengono cancellate dalla memoria e salvate sul disco fisso, evitando quindi di ricalcolarle in futuro. Questa operazione può essere effettuata manualmente tramite il metodo *unpersist()*, il quale rimuove dalla memoria i dati cachati.

1.4 Spark SQL

Spark SQL è un modulo che si appoggia a Spark Core con lo scopo di elaborare grandi quantità di dati strutturati. Offre un supporto per l'analisi di dati sia in linguaggio SQL sia nel linguaggio di interrogazione proprio di Hive, chiamato HiveQL. Inoltre permette di importare i dati strutturati da diverse sorgenti, come file Parquet e file JSON, senza alcuna difficoltà. Le astrazioni chiave di Spark SQL sono l'API DataFrame e l'API Dataset, le quali forniscono primitive in grado di rappresentare e lavorare con i dati strutturati senza aggiungere ulteriore complessità. In aggiunta, Spark SQL include al suo interno un ottimizzatore di nome *Catalyst*, che permette di analizzare e ottimizzare i piani logici, al fine da generare il migliore codice bytecode della query di partenza.

1.4.1 DataFrames

Un DataFrame è una collezione distribuita di dati organizzati in colonne, dove ad ogni colonna è associato un tipo e un nome. Concettualmente è equivalente ad una tabella in un tradizionale database relazionale, con la differenza che le performance del primo sono molto più elevate in caso di grandi mole di dati. A differenza degli RDD, ogni DataFrame presenta uno schema, il quale ne definisce la struttura. Tuttavia, fornendo uno schema al metodo *toDF()*, è possibile trasformare facilmente un RDD in un DataFrame con lo schema desiderato. Oltre a questa modalità, è possibile creare un DataFrame a partire da vari tipi di file, come CSV, JSON, XML, AVRO, ORC, Parquet e da tabelle Hive. Come per l'astrazione RDD, le operazioni supportate dai DataFrame si suddividono in due categorie: azioni e trasformazioni.

1.4.2 Datasets

L'API Dataset è stata sviluppata con l'idea di creare una nuova struttura dati avente le caratteristiche di flessibilità e sicurezza rispetto al tipo di dato in fase di compilazione proprie degli RDDs unite all'efficienza introdotta dai DataFrames. Un dataset è una collezione di oggetti JVM fortemente tipizzata e immutabile mappati su uno schema relazionale. Per supportare la caratteristica di *compile type-safety*, ogni record di un dataset è rappresentato da un oggetto definito dall'utente; in questo modo un errore rispetto al tipo di dato viene rilevato in fase di compilazione e non in fase di esecuzione. Ovviamente, nel mondo dei Big Data, è preferibile rilevare un errore in fase di compilazione al fine di evitare di rieseguire operazioni su un volume di dati enorme. La tabella 1.3 riassume come vengono rilevati gli errori sintattici e di analisi nelle API DataFrame e Dataset.

Tabella 1.3: Rilevazione errori di sintassi e analisi su Spark

	DataFrames	**Datasets**
Syntax errors	Compile time	Compile time
Analysis errors	Runtime	Compile time

L'API Datasets fa uso di un *encoder*, uno strumento in grado di convertire i dati contenuti all'interno di oggetti JVM creati dall'utente in un formato binario compatto chiamato *Tungsten*. Questa traduzione ha permesso di ridurre il quantitativo di memoria usata per cachare un dataset di un fattore 4.5 rispetto al tradizionale RDD e ridurre il numero di byte da distribuire ai nodi del cluster durante l'operazione di *shuffling* [5] (Figura 1.7). Inoltre, gli encoders supportati da Spark sono

in grado di generare un bytecode personalizzato che rende le operazioni di serializzazione e deserializzazione maggiormente ottimizzate rispetto alle performance offerte dalla serializzazione con Java o Kyro (Figura 1.8).

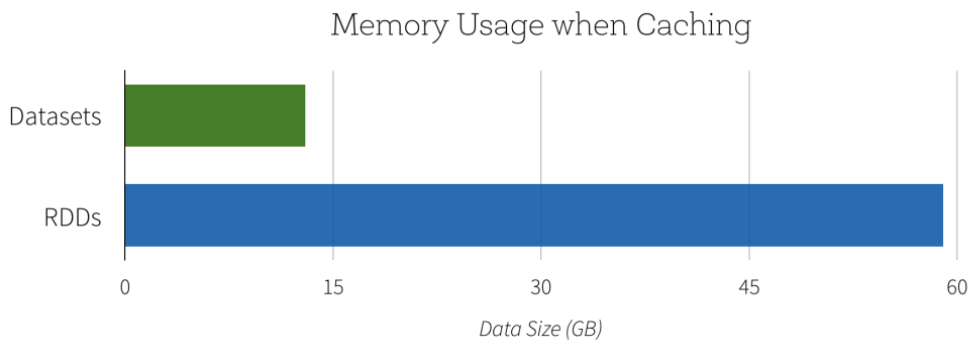

Figura 1.7: Confronto fra l'uso della memoria durante il caching di un dataset e di un RDD

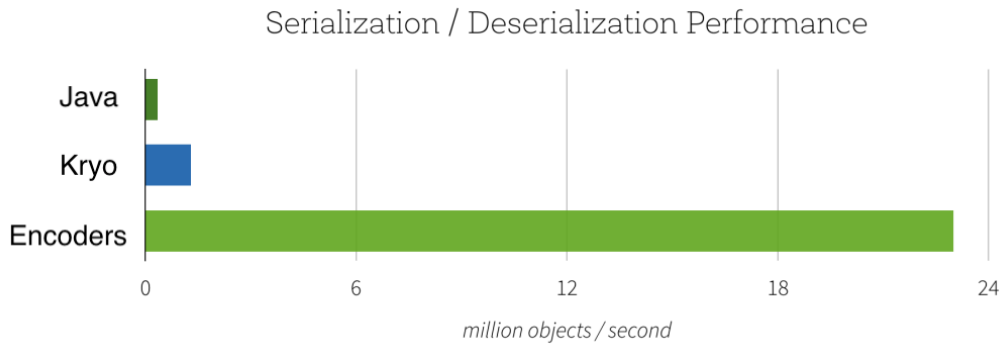

Figura 1.8: Performance durante la serializzazione e deserializzazione di un dataset

Dal momento che non è un linguaggio fortemente tipizzato, Python non supporta l'API Datasets. Grazie però alla sua natura dinamica, molti dei vantaggi introdotti dai datasets sono già intrinsecamente disponibili. E' supportata invece dai linguaggi Scala e Java.

1.4.3 Sorgenti dati di un DataFrame

Un DataFrame può essere creato a partire da un RDD preesistente, una tabella Hive o da una sorgente dati Spark. Grazie alla natura dinamica del modulo sorgente dati di Spark SQL, è possibile implementare una sorgente dati personalizzata. Le

sorgenti dati Spark supportate nativamente dall'API DataFrames sono mostrate in tabella 1.4.

Tabella 1.4: Sorgenti dati di un DataFrame

Nome	**Formato**	**Tipologia**
TXT	Testuale	Non strutturato
JSON	Testuale	Semi-strutturato
CSV	Testuale	Non strutturato
ORC	Binario	Strutturato
Parquet	Binario	Strutturato
JDBC	Binario	Strutturato

La Figura 1.9 mostra la differenza principale tra il formato *row-based* e il formato *column-based*.

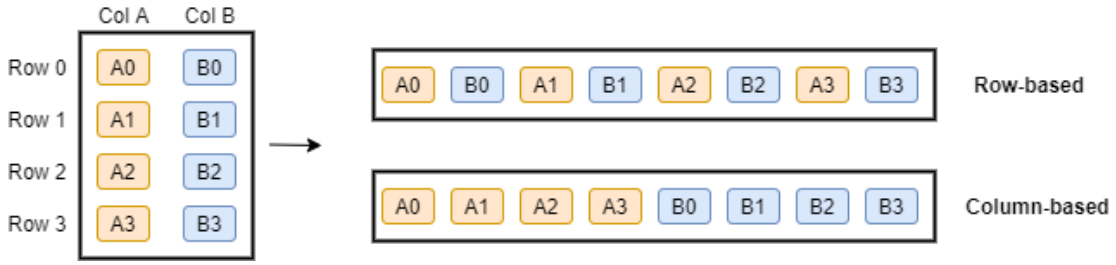

Figura 1.9: Differenza tra Row-based e Column-based

Si noti come, nel formato colonnare, i valori di ogni colonna risiedono in zone contigue di memoria.
Questa caratteristica porta diversi vantaggi:

- consumo di memoria ridotto di circa un terzo rispetto ai formati row-based [6] grazie ad una compressione più efficiente;
- riduzione delle operazioni di input/output, in quanto è possibile scansionare sottoinsiemi delle colonne durante la lettura dei dati;
- aumento di performance nel caso di interrogazioni su grandi volumi di dati grazie a ricerche selettive su colonne;
- possibilità di eseguire algoritmi di compressione specifici nel caso in cui i tipi di dato per colonna sono uguali;

La sorgente dati di default è Parquet, un formato di file binario *open source* e colonnare presente nell'ecosistema Hadoop. Ad alto livello, un file Parquet si compone di tre elementi:

- *header*: contiene il cosiddetto *magic number*, ovvero una stringa che identifica univocamente il tipo di file. Nel caso di file Parquet il magic number ha dimensione pari a 4 byte e valore "PAR1";
- *uno o più blocchi dati*: corrispondono alle partizioni fisiche dei dati memorizzate nel file system distribuito. Ogni blocco dati suddivide le partizioni orizzontali in una serie di *row group*. A loro volta questi gruppi di righe sono suddivisi in frammenti di colonne, che prendono il nome di *column chunk*. Essendo il Parquet un formato di file colonnare, è garantito che i column chunk di ogni row group siano memorizzati in zone di memoria contigue e che, ogni gruppo di righe, contenga un frammento di colonna per ogni colonna del dataset. Infine, ogni column chunk viene memorizzato in una o più *pages*;
- *footer*: contiene i metadati del file, oltre che un campo che identifica la propria dimensione e un campo che mantiene una copia del magic number. In particolare, grazie ai metadati memorizzati nel footer, è possibile ricavare le informazioni relative alle colonne del dataset, come per esempio il tipo, il percorso, la dimensione, l'offset delle pagine e così via.

Una rappresentazione visiva della struttura dei file Parquet è fornita dalla Figura 1.10, nella quale è evidenziata la suddivisione dei dati in più gruppi di righe, ognuno dei quali suddiviso in column chunk.

1.4.4 Manipolazione dei DataFrames

Allo stesso modo dell'API RDD, le operazioni supportate dall'API DataFrame si suddividono in due categorie: azioni e trasformazioni. Le trasformazioni sono processate in modalità *lazy*, ovvero solamente quando il risultato di esse è necessario per il proseguimento del processo, mentre le azioni sono valutate in modalità *eagerly*, cioè la chiamata di una di esse viene processata istantaneamente. A differenza delle operazioni sugli RDD, le operazioni supportate dall'API DataFrame sono per lo più di tipo strutturato; infatti permettono all'utente di analizzare e manipolare DataFrame grazie ad un insieme di operazioni che derivano dal mondo relazionale, quali per esempio *select()*, *filter()*, *groupBy()*, *orderBy()*, *join()* e molte altre.

1.4.5 Persistenza dei DataFrames

Allo stesso modo degli RDDs, anche i DataFrames possono essere resi persistenti in memoria tramite i metodi *persist()* e *cache()*, oltre che rimossi dalla memoria tramite la primitiva *unpersist()*. Grazie alle informazioni contenute all'interno dello

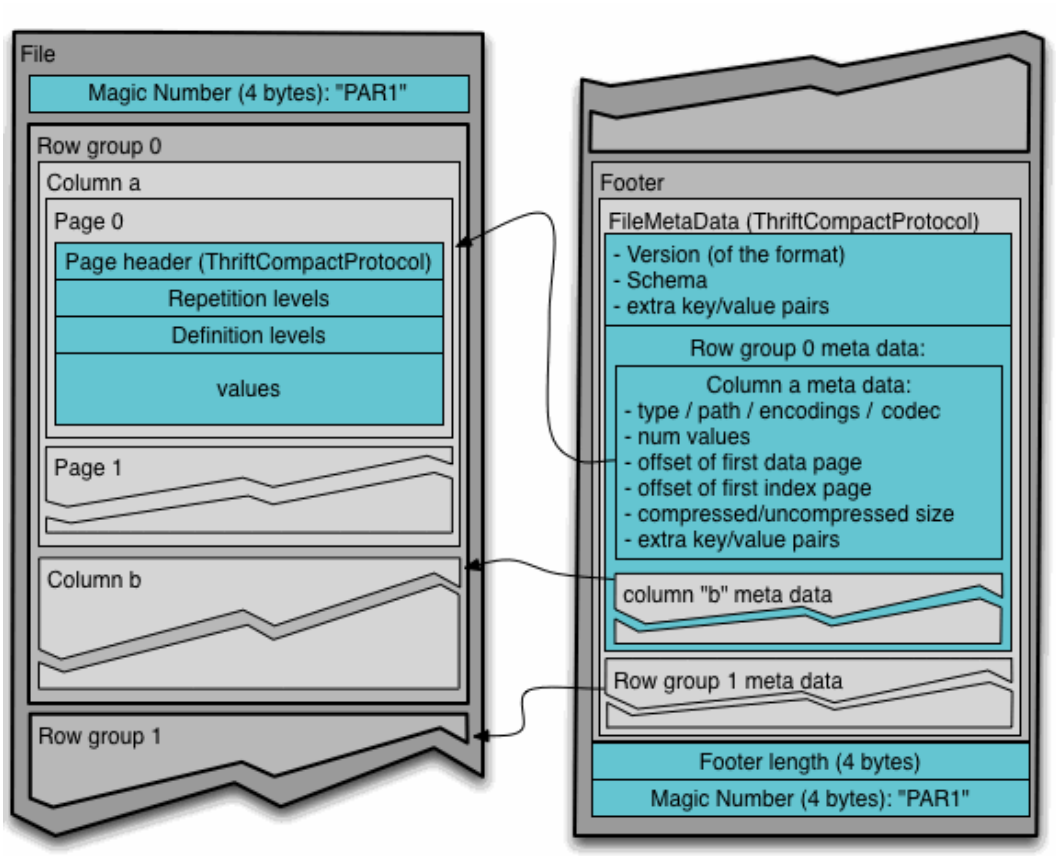

Figura 1.10: Struttura di un file Parquet

schema, Spark SQL è in grado di organizzare il DataFrame cachato in un formato colonnare, in modo da ridurre notevolmente lo spazio occupato. I livelli di memoria supportati dal metodo *persist()* sono equivalenti a quelli descritti nel paragrafo 1.3.8.

1.4.6 Join

Nella maggior parte delle applicazioni reali, studiare singolarmente le tabelle presenti nel sistema non porta ad analisi interessanti; per questo le operazioni di join fra diverse tabelle sono molto comuni. Le tipologie di join supportate da Spark SQL sono:

- Inner join
- Outer join
- Left join

- Right join
- Cross join
- Left semi join
- Left anti join

Per elaborare questo tipo di operazioni è necessario scorrere tutti i record delle tabelle coinvolte. Dal momento che Spark partiziona le strutture dati su diversi nodi del cluster, per completare un join è molto probabile che il motore di elaborazione esegua lo *shuffleling* dei dati, ovvero un meccanismo che permette di redistribuire i dati tra i vari nodi del cluster. L'operazione di shuffle porta tipicamente alla copia di una grandissima quantità dati tra i vari nodi della rete; è infatti considerata l'operazione più onerosa di tutte. Le performance di quest'ultima dipendono principalmente dalla capacità trasmissiva della rete e dalla quantità di dati da trasferire. All'interno di Spark SQL sono presenti due algoritmi per l'esecuzione di join:

- ***Broadcast Hash Join***(Figura 1.11): questo algoritmo può essere applicato nel caso in cui è possibile memorizzare una delle due tabelle in memoria centrale di ogni executor. L'esecuzione di questo tipo di join può essere suddivisa in due fasi. In un primo momento il driver inoltra interamente la tabella con dimensioni minori a tutti gli executor incaricati di eseguire il join. In seguito, ogni executor carica la partizione della tabella di dimensioni maggiori e confronta ogni record della stessa con il dataset caricato precedentemente in memoria centrale.
- ***Shuffle Hash Join*** (Figura 1.12): a differenza del broadcast hash join, questo algoritmo può essere applicato indipendentemente dalla dimensione dei due datasets. L'idea alla base di questo tipo di join è simile al modello di programmazione MapReduce, dove la prima fase prevede l'esecuzione di due operazioni: hash e shuffle. In particolare ogni executor, dopo aver caricato in memoria centrale i record opportuni, applica alla colonna specificata nelle condizioni di join una funzione di hash. In seguito, per distribuire i vari record ai nodi del cluster, ogni executor effettua un'operazione di shuffeling. Questa prima fase ha come obiettivo di collocare sulla stessa partizione i record dei due datasets con le stesse chiavi di join. La seconda fase prevede la combinazione dei dati presenti in memoria su ogni executor tramite un classico hash join.

L'algoritmo di Broadcast Hash Join evita di eseguire l'operazione di shuffle per entrambi i dataset, infatti distribuisce agli executor solamente la tabella con dimensioni minori. Per questo motivo, se la dimensione di una delle due tabelle è minore di un parametro chiamato *autoBroadcastJoinThreshold* (di default 10MB), Spark sfrutta il Broadcast Hash Join come tecnica di join. È inoltre possibile specificare il tipo di join da utilizzare grazie ai cosiddetti hint (consigli), ma non è garantito che Spark utilizzi la strategia di join indicata.

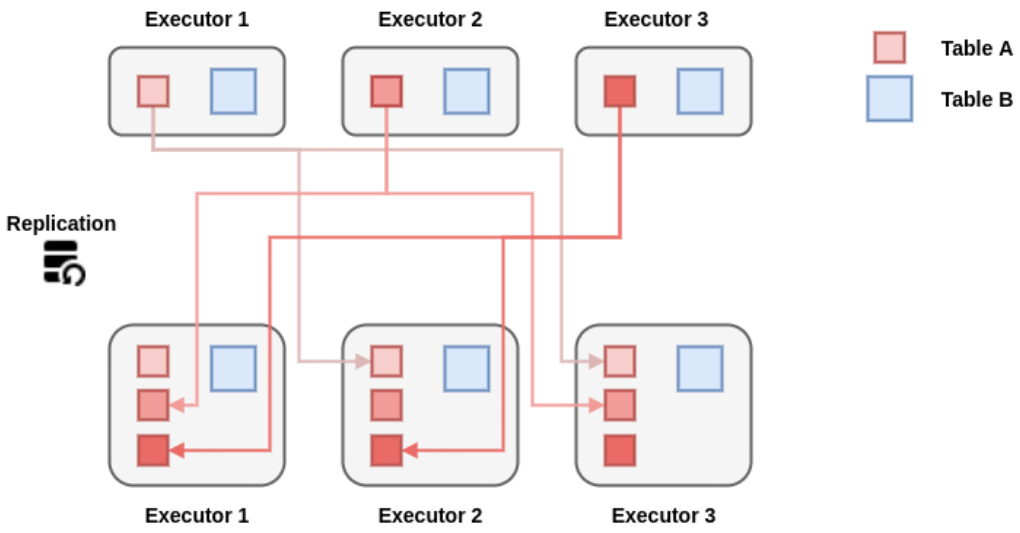

Figura 1.11: Broadcast Hash Joint

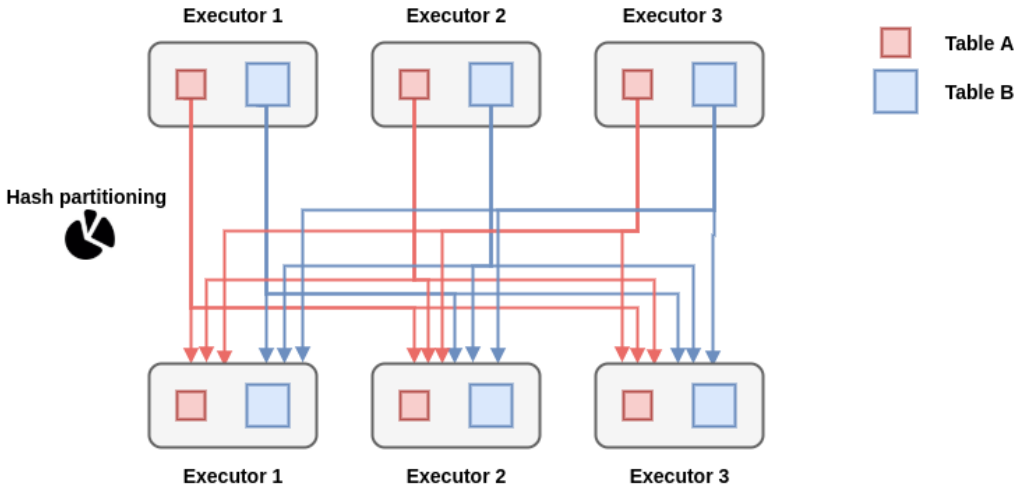

Figura 1.12: Shuffle Hash Join

1.4.7 Catalyst

Al fine di ridurre i tempi di risposta alle interrogazioni, Spark SQL introduce un modulo di ottimizzazione basato su costrutti di programmazione funzionale in Scala chiamato *Catalyst*. A differenza di altri *query optimizer* reperibili sul mercato, è possibile aggiungere ad esso nuove funzionalità e tecniche di ottimizzazione grazie alla sua estensibilità. Per recuperare il piano di esecuzione più efficiente per ogni operazione specificata dall'utente, questo modulo fa uso fa uso di una libreria per la

rappresentazione di alberi. Ogni albero è composto da uno o più nodi oggetto, i quali sono caratterizzati da un tipo e da zero o più figli. Queste strutture dati possono essere manipolate tramite le cosiddette *rules*, ovvero funzioni che permettono di creare nuovi alberi a partire da alberi preesistenti grazie alla natura immutabile degli stessi.
È possibile suddividere il flusso di lavoro di Catalyst (Figura 1.13) in quattro fasi:

- *Analysis*: in questa prima fase si genera il piano logico a partire da un piano logico non risolto rappresentato da un AST (*Abstract Syntax Tree*) o da un oggetto DataFrame. In particolare Spark SQL usa un'istanza della classe *Catalog* e le regole di Catalyst per calcolare i riferimenti agli eventuali attributi non risolti. Il piano logico prodotto è descritto mediante un albero in cui i nodi corrispondono alle operazioni da eseguire.

- *Logical Optimizations*: in questa fase si applicano ottimizzazioni al piano logico generato precedentemente. In aggiunta alle regole standard, come per esempio la semplificazione delle espressioni booleane o il folding delle costanti, è possibile aggiungere regole personalizzate.

- *Physical Planning*: in questa fase si trasforma il piano logico in uno o più piani fisici. Successivamente, grazie ad un semplice modello di costo, viene selezionato il piano fisico migliore per poi applicare ad esso ulteriori ottimizzazioni.

- *Code Generation*: in questa ultima fase Spark SQL genera bytecode Java a partire dal piano fisico migliore.

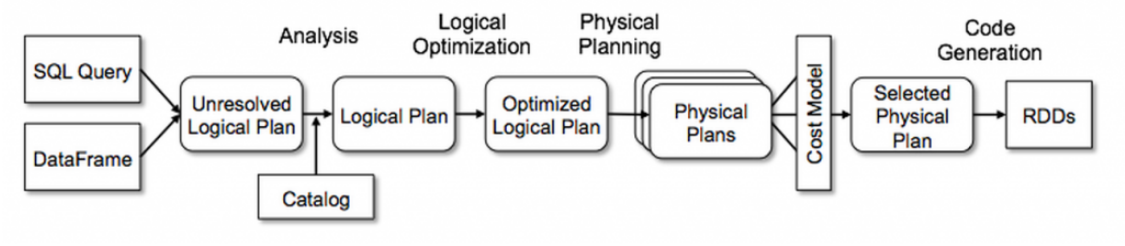

Figura 1.13: Flusso di lavoro catalyst

1.5 Spark Streaming

Spark Streaming è un'estensione di Spark Core che permette di implementare applicazioni scalabili, tolleranti agli errori e con elevato throughput per l'analisi di flussi dati in tempo reale. In particolare consente di elaborare flussi dati provenienti da

varie fonti grazie a funzioni di alto livello, per poi successivamente memorizzare i dati manipolati in databases, filesystems o live dashboards. A differenza di altri framework per l'elaborazione di dati in real time, Spark Streaming sfrutta il modello di elaborazione chiamato *microbatch*, il quale prevede di attendere per un intervallo di tempo prefissato i dati in ingresso con lo scopo di suddividerli in *batch* e memorizzarli nel cluster in strutture dati fault-tolerant. Successivamente vengono generati Spark jobs per elaborare i batch in parallelo. Trattando lo stream in ingresso come una sequenza di batch è possibile ridurre il costo di elaborazione per ogni singolo dato, con la diretta conseguenza di un thoughput più elevato ma con una latenza maggiore rispetto ai framework che sfruttano un modello di elaborazione *element-at-time*, in cui ogni elemento è processato nel momento in cui è rilevato dal sistema.

1.5.1 DStream

Oltre alle astrazioni fornite da Spark Core, Spark Streaming per descrivere un flusso continuo di dati introduce una nuova astrazione chiamata *discretized stream*, o più brevemente *DStream*. Questa astrazione è rappresentata internamente come una sequenza continua di RDDs, dove al suo interno ogni Resilient Distribuited Dataset contiene dati di una particolare finestra temporale. Essendo composta da oggetti immutabili, DStream è un'astrazione immutabile a sua volta. Ogni operazione eseguita su un DStream è riportata a tutti gli RDD di cui è composto, ma è anche possibile accedere singolarmente agli RDD per eseguire trasformazioni e azioni selettive. Come accade per gli RDD, anche l'astrazione DStream mantiene la propria lineage per rappresentare le operazioni effettuate su di esso. In questo modo, in caso di guasto ad un nodo, è possibile ricalcolare la partizione persa eseguendo le trasformazioni mappate nella lineage sull'input originale memorizzato nel cluster.

1.5.2 Trasformazioni su DStream

Spark Streaming supporta due tipi di operazioni su un DStream:

- *Transformations*: sono operazioni che creano un nuovo DStream modificando un DStream preesistente. Oltre a supportare la maggior parte delle trasformazioni stateless introdotte dall'API RDD, come *map*, *reduce*, *gropuBy* e molte altre, Spark Streaming fornisce operazioni stateful per elaborare flussi di dati di intervalli diversi. Tra le operazioni stateful introdotte sono da evidenziare le cosiddette *window transformation*, ovvero operazioni che permettono di applicare una determinata trasformazione ai dati presenti all'interno di una particolare finestra scorrevole. Specificando la durata della finestra (in secondi) e l'intervallo della finestra in cui eseguire l'operazione (in secondi), è possibile analizzare e modificare più dataset alla volta. Le principali trasformazioni supportate dalla finestra temporale sono mostrate nella tabella 1.5.

- *Output operations*: sono operazioni che permettono di scrivere i dati in un sistema esterno. Ogni applicazione implementata tramite Spark Streaming deve contenere almeno un'operazione di output al proprio interno. Le principali operazioni di output sono descritte nella tabella 1.6.

Tabella 1.5: Le principali window trasformation

Transormation	**Significato**
window*(windowLength, slideInterval)*	Restituisce un nuovo DStream a partire dalla finestra temporale definita
countByWindow*((windowLength, slideInterval)*	Restituisce il numero di elementi dello stream presenti all'interno della finestra temporale definita
reduceByWindow*(func, windowLength, slideInterval)*	Data una funzione *func* associativa e commutativa, *reduceByWindow* restituisce un nuovo stream creato aggregando gli elementi dello stream sull'intervallo *slideInterval* grazie alla funzione *func*
reduceByKeyAndWindow*(func,*	

Tabella 1.5: Le principali window trasformation

Transormation	**Significato**
windowLength, slideInterval, [numTasks])	Dato un DStream di coppie chiave-valore (K,V) viene restituito un nuovo DStream di coppie chiave-valore (K,V) in cui i valori con la stessa chiave vengono aggregati tramite la funzione *func* sulla finestra temporale definita. È possibile specificare anche il numero di task da creare tramite l'attributo *numTasks*, che di default è 2
countByValueAndWindow*((windowLength, slideInterval, [numTasks])*	Dato un DStream di coppie chiave-valore (K,V) viene restituito un nuovo DStream di coppie chiave-valore, dove il valore di ogni chiave corrisponde alla sua frequenza all'interno della finestra temporale definita. Come per la trasformazione *reduceByKeyAndWindow()*, anche in questo caso è possibile modificare il numero di thread creati tramite l'attributo *numTasks*

Tabella 1.6: Le principali operazioni di output

Output operation	Significato
print*()*	Stampa i primi dieci elementi di ogni batch di dati di un DStream sul nodo driver che sta eseguendo l'applicazione
saveAsTextFiles*(prefix, [suffix])*	Salva il contenuto del DStream in file di testo. Il nome del file in ogni intervallo è generato in base all'attributo *prefix*
saveAsObjectFiles*(prefix, [suffix])*	Salva il contenuto del DStream come *SequenceFiles* di oggetti Java serializzati, dove SequenceFile è un file di tipo binario utilizzato nell'ecosistema Hadoop. Il nome del file in ogni intervallo è generato in base all'attributo *prefix*
saveAsHadoopFiles*(prefix, [suffix])*	Salva il contenuto del DStream in file Hadopp. Il nome del file in ogni intervallo è generato in base all'attributo *prefix*
foreachRDD*(func)*	Esegue la funzione *func* ad ogni RDD generati dallo stream. È l'operatore di output più generico e flessibile; in base alla funzione *func* è possibile, per esempio, memorizzare gli RDD in un file di testo o in un database esterno

1.6 Structured Streaming

Structured Streaming rappresenta la seconda generazione di motori di elaborazione streaming introdotti nell'ecosistema di Spark. Costruito sul framework Spark SQL, Structured Streaming è definito come un motore di elaborazione streaming scalabile e tollerante agli errori. Permette di implementare applicazioni *end-to-end* per l'analisi e la manipolazione di flussi di dati in tempo reale con prestazioni più elevate rispetto al precedente framework Spark Streaming. A differenza di molti altri motori di elaborazione, Structured Streaming è responsabile dell'aggiornamento della tabella dei risultati in presenzi di nuovi dati in ingresso, evitando che l'utente si preoccupi della coerenza dei dati. A differenza dei primi motori di elaborazione rilasciati, i framework di nuova generazione forniscono una garanzia di consegna dei dati che può essere di tre tipi:

- *At-most-once*: è garantito che ogni record venga inoltrato in uscita zero o al più una volta. Questo comporta che, nel caso in cui il nodo ricevente non sia stato in grado di elaborare il messaggio, non è possibile inviarlo nuovamente.

- *At-least-once*: è garantito che ogni record venga inoltrato in uscita almeno una volta. Questo comporta che i messaggi possono essere duplicati ma non persi.

- *Exactly-once*: è garantito che ogni record venga inoltrato in uscita esattamente una volta. Come è facile prevedere, questo meccanismo è il più costoso e complesso da implementare ma si preoccupa della consistenza dei dati.

Oltre alle caratteristiche end-to-end e fault-tolerance, Structured Streaming garantisce un'elaborazione exactly one, privando l'utente dei problemi quali la perdita e duplicazione dei dati inoltrati dall'applicazione Spark ad un sistema di archiviazione esterno. L'idea chiave su cui si basa questo framework è quella di trattare il flusso dati in ingresso come una tabella che viene continuamente aggiornata; ogni nuovo dato in ingresso rappresenta un nuovo record della tabella. Questa idea permette di sfruttare le APIs DataFrame e Dataset per manipolare e analizzare il flusso di dati in ingresso, lasciando al motore Structured Streaming il compito di aggiornare continuamente la tabella dei risultati. In questo modo l'utente non trova alcuna differenza tra *streaming processing* e *batch processing*, per tale motivo questo framework è considerato un motore di elaborazione streaming *user-friendly*. Inoltre, per generare ed elaborare piani logici, Structured Streaming si serve dello stesso ottimizzatore di Spark SQL chiamato Catalyst.

1.6.1 Modello di programmazione

Come già anticipato, Structured Streaming tratta il flusso dati in ingresso come una tabella che viene continuamente aggiornata, chiamata *input table*. Ad ogni evento generato da un *trigger* il motore di elaborazione controlla la presenza di nuovi dati in ingresso e, in caso affermativo, aggiorna la tabella. Per ogni query definita dall'utente Spark genera una tabella contenente il risultato dell'interrogazione, chiamata *result table*, che viene eventualmente aggiornata ad ogni intervallo temporale definito dal trigger (Figura 1.14). Dovendo elaborare un flusso continuo di dati in ingresso pur mantenendo le tabelle dei risultati consistenti, Spark converte le interrogazioni di tipo batch in un piano di esecuzione streaming. Questa fase è chiamata *incrementalization phase*, e consente a Spark di individuare i record delle tabelle che devono essere aggiornati. Ad ogni aggiornamento della result table, Structured Streaming è in grado di memorizzare le informazioni elaborate su un sistema di archiviazione esterno in tre diverse modalità:

- ***Complete mode***: l'intera tabella dei risultati viene scritta nel sistema di archiviazione.

- ***Append mode***: vengono scritte nel sistema di archiviazione solamente le righe aggiunte alla tabella dei risultati nell'ultimo intervallo. Questa modalità è applicabile unicamente per interrogazioni nelle quali non sono previsti aggiornamenti ai record della result table.
- ***Update mode***: vengono scritte nel sistema di archiviazione solamente le righe aggiornate alla tabella dei risultati nell'ultimo intervallo.

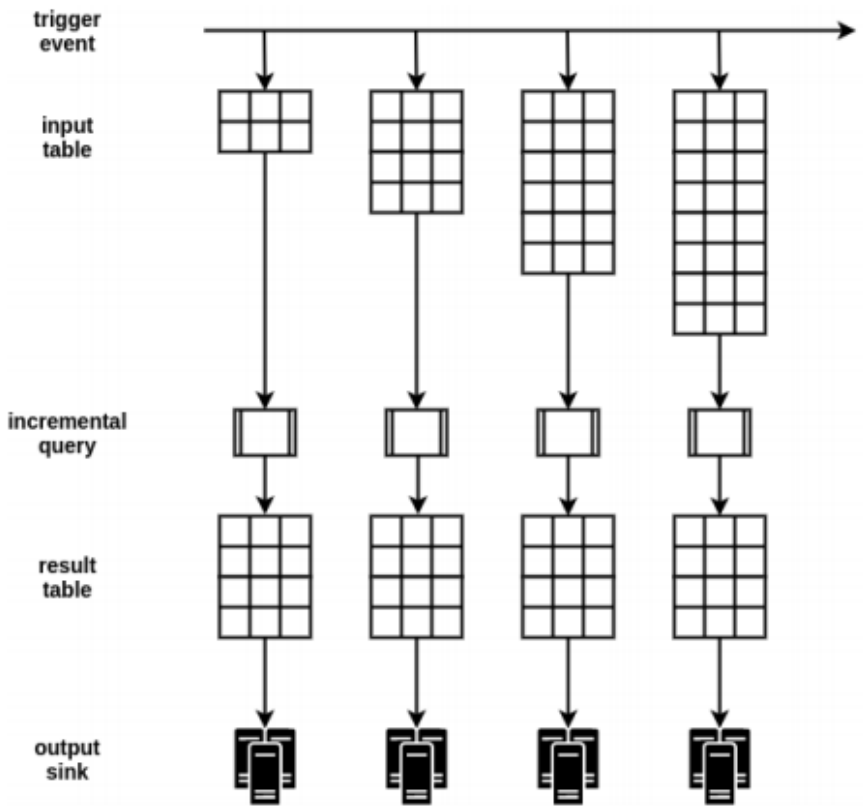

Figura 1.14: Modello di programmazione di Structured Streaming

Di default Structured Streaming sfrutta un modello di esecuzione *micro-batch*, grazie al quale è in grado di suddividere i flussi dati in ingresso in una serie di batch per poi elaborarli ad ogni evento generato da un trigger. Al fine di garantire una semantica end-to-end, si possono verificare casi in cui un record in ingresso deve aspettare l'elaborazione di un batch corrente prima di memorizzare il proprio offset e la successiva manipolazione. Questo fenomeno comporta latenze end-to-end nell'ordine dei 100 millisecondi tra il momento in cui un evento è disponibile in ingresso e quando l'output viene memorizzato nel sistema di archiviazione esterno [7]. Tuttavia, esistono alcuni casi d'uso in cui sono necessarie latenze di ordine minore. Per

questo motivo da Spark 2.3 è stato introdotto un nuovo modello di esecuzione chiamato *continuous processing*, il quale garantisce latenze di circa 1 millisecondo [8], mantenendo le caratteristiche end-to-end e fault-tolerance introdotte nelle versioni precedenti. Questo nuovo modello di esecuzione si basa sull'idea di processare i dati in entrata e memorizzarli sul sistema di archiviazione il prima possibile, evitando di attendere gli eventi generati dai trigger. Essendo ancora in fase sperimentale, questo modello di esecuzione presenta varie limitazioni; attualmente non supporta tutti i tipi di query che sono invece disponibili con il modello micro-batch.

1.6.2 Tolleranza agli errori

Lavorando con sistemi distribuiti, non è raro che una o più macchine del cluster falliscano. Grazie all'uso di *checkpoint* e *write-ahead logs*, Structured Streaming è in grado di recuperare lo stato di un' applicazione al momento del fallimento. Per raggiungere questo obiettivo è necessario che ogni sorgente dati supporti una serie di offset per identificare la posizione di lettura del flusso dati, come per esempio l'offset iniziale e finale di un particolare intervallo temporale, per mantenere informazioni riguardo lo stato di avanzamento del programma. Automaticamente il motore di elaborazione memorizza nel sistema una serie di checkpoint e write-ahead logs che, in caso di errore di un nodo del cluster, permettono all'utente di riavviare il sistema (specificando il checkpoint desiderato) per recuperare lo stato dell'applicazione al momento del fallimento.

1.6.3 Tipi di trigger

Il motore di elaborazione di Structured Streaming sfrutta gli eventi generati dai triggers per determinare gli istanti in cui controllare se sono presenti nuovi dati in ingresso e per aggiornare le tabelle dei risultati. Spark supporta quattro diversi tipi di trigger (descritti in tabella 1.7), i quali determinano il modello di esecuzione delle query (micro-batch o continuous processing).

Tabella 1.7: Tipologie di trigger di Structured Streaming

Tipo	**Descrizione**
Not specified (default)	Rappresenta il tipo di trigger di default di Spark. In questo caso le query sono processate in modalità *micro-batch*, dove ogni lotto di dati è elaborato subito dopo aver terminato di processare il lotto di dati precedente

Tabella 1.7: Tipologie di trigger di Structured Streaming

Tipo	Descrizione
One-time micro-batch	In questo caso le query eseguono un solo batch per processare tutti i dati disponibili. Una volta terminata l'elaborazione del flusso dati Spark arresta l'applicazione. Questo tipo di trigger è utile nel caso in cui la frequenza dei dati in ingresso è bassa, infatti è possibile avviare periodicamente il cluster per processare i dati e successivamente arrestarlo in attesa di altri flussi. Permette di non sovraccaricare di continui aggiornamenti il sistema di archiviazione di uscita
Fixed intervall micro-batch	In questo caso le query sono processate in modalità *micro-batch*, dove ogni lotto di dati è elaborato all'interno di un intervallo temporale specificato dall'utente. Nel caso in cui l'elaborazione di un batch richiede più tempo di quello specificato dall'utente, il batch successivo verrà processato non appena il lotto corrente è stato completato. Nel caso in cui l'elaborazione di un batch viene completata all'interno dell'intervallo, il motore non inizia l'esecuzione del prossimo batch ma attende il termine dell'intervallo
Continuous	In questo caso le query sono processate in modalità *continuous*, ottenendo latenze di circa 1 millisecondo

1.6.4 Sorgenti e ricevitori dati

Structured Streaming supporta diversi tipi di sorgenti dati e altrettanti tipi di sistemi di archiviazione esterni. In particolare, le sorgenti dati nella tabella 1.8 forniscono un flusso continuo di informazioni che devono essere elaborate dal motore di Spark per poi essere memorizzate in un secondo momento su un sistema di archiviazione descritto in tabella 1.9. Al fine di garantire la caratteristica di tolleranza ai guasti, le sorgenti dati supportate da Structured Streaming devono essere riproducibili, ovvero supportano la lettura multipla in caso si verificasse un errore, mentre i sistemi di archiviazione in uscita devono supportare aggiornamenti transazionali in modo che Spark possa scrivere in uscita un insieme di record atomicamente.

Tabella 1.8: Sorgenti dati supportati da Structured Streaming

Tipo	Descrizione	Tolleranza agli errori
File	Rappresenta il tipo di sorgente dati più elementare. Spark valuta i file contenuti all'interno di una cartella come un flusso di dati in ingresso. I formati di file supportati sono: testuale, CSV, JSON, ORC e Parquet	Si
Socket	E' un tipo di sorgente dati utilizzato per testare l'applicativo, infatti non garantisce trasferimenti end-to-end e tolleranti agli errori. Legge dati in formato UTF8 provenienti da un socket in ascolto su una porta del driver. prefissata	No
Rate	E' un tipo di sorgente dati utilizzato per testare l'applicativo. In particolare genera un numero di righe per secondo prefissato dall'utente, in cui ogni riga contiene un timestamp ed un valore	Si
Kafka	E' possibile leggere dati inoltrati da Apache Kafka, un piattaforma di streaming distribuita e open-source [9]	Si

Tabella 1.9: Sistemi di archiviazione esterni supportati da Structured Streaming

Archiviazione	Descrizione	Tolleranza agli errori	Modalità di output supportate
File	Memorizza l'output in una directory del sistema	Si	Append

Tabella 1.9: Sistemi di archiviazione esterni supportati da Structured Streaming

Archiviazione	**Descrizione**	**Tolleranza agli errori**	**Modalità di output supportate**
Foreach	Permette di esegue calcoli arbitrari sui record dell'output in parallelo. Non sempre mantiene la caratteristica di tolleranza agli errori, essa infatti dipende dalle operazioni da eseguire sul flusso dati	Non sempre	Append, Update, Complete
Console	E' un tipo di archiviazione utile per il testing e il debugging del codice. Ad ogni evento generato dal trigger, Structured Streaming stampa l'output sulla console del driver	No	Append, Update, Complete

Tabella 1.9: Sistemi di archiviazione esterni supportati da Structured Streaming

Archiviazione	**Descrizione**	**Tolleranza agli errori**	**Modalità di output supportate**
Memory	E' un tipo di archiviazione utile per il testing e il debugging del codice. L'output generato dal framework viene salvato nella memoria del driver. Ovviamente è un tipo di archiviazione adatta a contesti in cui si lavora con volumi di dati non elevati	No	Append, Complete
Kafka	E' possibile memorizzate l'output in uno o più topic in Apache Kafka	Si	Append, Update, Complete

1.6.5 Aggregazione con finestre temporali sulla base dell'event time

Nelle applicazioni che trattano flussi di dati in tempo reale è di fondamentale importanza minimizzare le latenze, infatti più aumenta il tempo di trasferimento dei dati più l'intero processo di elaborazione degli stessi sarà lento. In un mondo ideale il trasferimento di dati non introduce ritardi ma nella realtà questo non accade; seppur diminuendo le latenze è impossibile annullarle del tutto. Per questo motivo molte applicazioni streaming hanno la necessità di manipolare i dati in base all'istante di tempo in cui sono stati generati e non in base all'instante in cui sono stati ricevuti. Formalmente si differenzia fra *event time* e *processing time*, dove il primo si riferisce al momento in cui il dato è stato generato mentre il secondo identifica

l'istante di tempo in cui il dato è stato ricevuto. A differenza di Spark Streaming, Structured Streaming permette all'utente di manipolare lo stream sulla base dell'event time. In particolare è possibile aggregare i dati tramite finestre temporali sulla base dell'event time degli stessi; Spark infatti si occuperà di mantenere gli aggregati aggiornati e in stati fault-tolerant. Structured Streaming supporta l'aggregazione sia tramite *fixed window* (Figura 1.15) sia tramite *slidind window* (Figura 1.16). Il primo tipo di finestra suddivide il flusso dati in blocchi non sovrapponibili aventi dimensione prefissata in base al proprio event time, mentre il secondo tipo di finestra raggruppa i dati all'interno di una finestra che scorre lungo il flusso dati in base ad un intervallo prefissato.

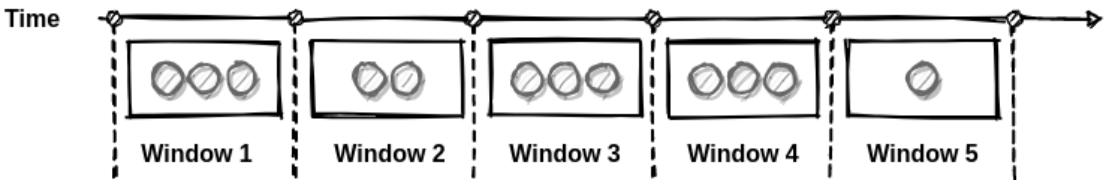

Figura 1.15: Fixed Window

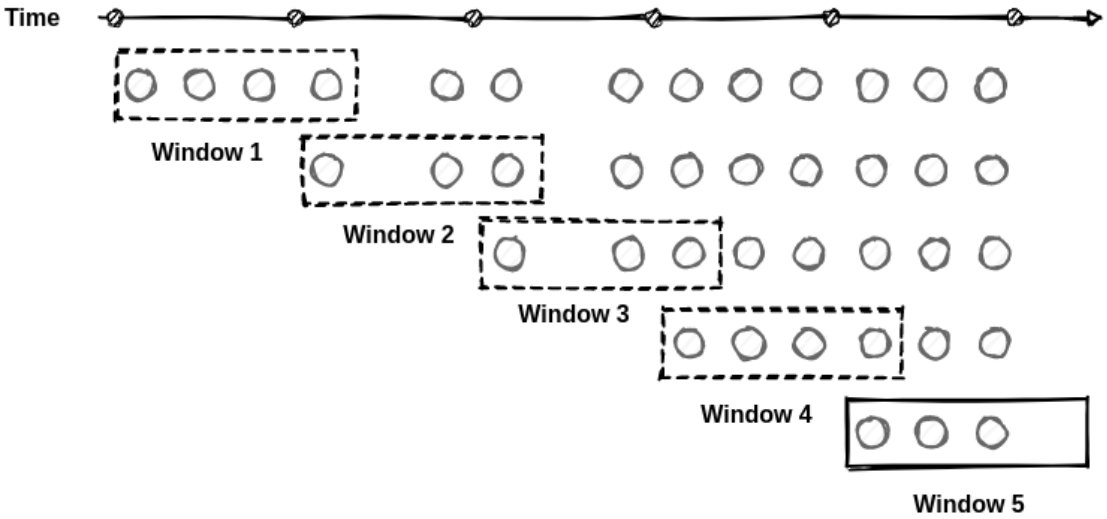

Figura 1.16: Sliding Window

Nelle applicazioni reali, a causa di problemi di congestione e latenze, si possono verificare situazioni in cui i dati in ingresso al motore di elaborazione si presentano fuori ordine. È quindi opportuno mantenere in memoria gli stati intermedi delle aggregazioni per attendere eventuali dati in ritardo. Come si può immaginare, è

impensabile che Spark mantenga le aggregazioni in memoria per un tempo illimitato, infatti il cluster dovrebbe avere a disposizione una quantità di memoria infinita. Per ovviare a questo problema, con Spark 2.1 è stato introdotto il *watermarking*, un meccanismo che permette di rimuovere dalla memoria stati intermedi dopo un determinato periodo di tempo. In questo modo è possibile specificare quanto tempo mantenere gli stati intermedi in memoria prima di considerarli come stati finali, ovvero stati che non devono essere più modificati e sono pronti per essere memorizzati in un sistema esterno. In particolare, per una specifica finestra che termina all'istante T, Spark mantiene gli stati in memoria fino a che la formula seguente sarà verificata:

$$MaxEventTimeSeenByEngine - LateThreshold > T \tag{1.1}$$

Senza questo tipo di funzionalità, il sistema dovrebbe dover tenere traccia dello stato di tutte le finestre, aumentando notevolmente il consumo di memoria durante l'esecuzione del programma.

1.7 Sorgente del flusso dati

Modena Automotive Smart Area (MASA) è un progetto di ricerca nato nel 2016 da una collaborazione fra Comune di Modena, Università di Modena e Reggio Emilia e Maserati S.p.a con il supporto della Regione Emilia Romagna per lo sviluppo della mobilità smart e dei veicoli autonomi. L'obiettivo dichiarato del progetto è quello di essere il primo laboratorio urbano nazionale "a cielo aperto" per la sperimentazione, la verifica e la ricerca delle tecnologie di guida autonoma. L'infrastruttura di MASA si compone di tre elementi: *Smart Model Area*, *Smart Dynamic Area* e il *Research & Development Lab*. In questo elaborato si prende in considerazione Smart Model Area, un'area urbana di circa 1 km^2 nei pressi della stazione dei treni di Modena dotata una serie di tecnologie per la sperimentazione della comunicazione bidirezionale tra veicoli connessi ed elementi della mobilità e della città e per la sperimentazione di veicoli attrezzati con dispositivi ADAS[1] fino a livello 3 e 4 (Figura 1.17).

In particolare, le tecnologie installate internamente alla Smart Model Area comprendono una serie di sensori interconnessi in rete, telecamere per il riconoscimento di ostacoli, segnaletica digitale e quattro semafori interconnessi ad un sistema cloud. Dopo questa breve introduzione, è facile comprendere come la quantità di dati generati dalle infrastrutture sia enorme e senza sosta.

[1] ADAS(Advanced Driver Assistance Systems): sistemi elettronici che supportano il guidatore di un veicolo in situazioni di pericolo o emergenza, con lo scopo di prevenire o limitare le possibilità di un incidente

Figura 1.17: Smart Model Area

Anche se Structured Streaming supporta diversi tipi di sorgenti dati, come per esempio Kafka e socket, per semplicità si è assunto che i dati provenienti da Modena Smart Area siano forniti in file testuali aventi struttura ben definita. In particolare, essi sono mantenuti all'interno di una gerarchia di cartelle, in cui ogni directory rappresenta un particolare giorno dell'anno e contiene 24 sottocartelle che indicano l'ora del giorno. Inoltre, ogni cartella che presenta informazioni orarie mantiene al suo interno 60 file testuali, ognuno dei quali è assegnato ad un minuto specifico. Ogni file testuale presenta i seguenti campi:

- *timestamp*;
- *id*;
- *latitude*;
- *longitude*;
- *speed*;
- *yaw*.

Più nello specifico, il campo *id* permette di rilevare il tipo di veicolo che ha generato il particolare record. I valori che tale campo può assumere sono stati estratti da dataset VOC [2] e sono mostrati in tabella 1.10.
Anche se, attualmente, i campi *speed* e *yaw* sono sempre nulli, è stato deciso di mantenere ugualmente questi attributi nei dati aggregati al fine di supportare eventuali futuri sviluppi.

Tabella 1.10: Lista di veicoli ed i loro identificatori

Id	**Oggetto**
1	bicycle
2	bird
3	boat
4	bottle
5	bus
6	car
7	cat
8	chair
9	cow
10	diningtable
11	dog
12	horse
13	motorbike
14	person
15	pottedplant
16	sheep
17	sofa
18	train
19	tvmonitor

[2]Pascal VOC Dataset: dataset molto popolare per la creazione e la valutazione di algoritmi per la classificazione delle immagini, il rilevamento di oggetti e la segmentazione.

Capitolo 2

Reverse geocoding

Con l'avvento dell'Internet of Things (IoT) molti più dispositivi elettronici generano volontariamente o involontariamente dati spaziali, ovvero dati che possiedono informazioni topologiche, metriche e direzionali. Per questo motivo, negli ultimi anni molte aziende hanno introdotto sul mercato strumenti per geo localizzare le informazioni attraverso due processi: *geocoding* e *reverse geocoding*. Come evidenziato dal nome, il reverse geocoding rappresenta l'algoritmo inverso del geocoding; se il primo converte una coppia latitudine-longitudine in un indirizzo, il secondo processo restituisce una coppia di coordinate a partire da un indirizzo. Il presente capitolo ha l'obiettivo di illustrare le tecnologie utilizzate per implementare un algoritmo di reverse geocoding efficiente sfruttato nell'applicazione reale descritta nella seconda parte dell'elaborato.

2.1 Perché un Reverse geocoder personalizzato?

Il reverse geocoding è il processo che permette di convertire una coppia di coordinate in un indirizzo. Essendo una delle operazioni più comuni richieste in ambito geospaziale, esistono molte API disponibili sul mercato in grado di eseguire operazioni di reverse geocoding, come per esempio *Google Maps*, *Bing Maps Geocode*, *Nominatim*, *Yahoo PlaceFinder* e molte altre. Tutte queste API non sono però implementate con lo scopo di lavorare con grandi quantità di dati in ingresso, hanno tempi di risposta che dipendono da fattori quali la banda di rete e il carico dei server. Lavorando con API gratuite, il numero di query che si possono effettuare non è illimitato; per esempio, senza una chiave, Google Maps limita le richieste giornaliere a 2500. Inoltre, da uno studio preliminare, si è notato come le API gratuite siano poco precise e non molto affidabili. Per queste ragioni, disponendo di una grande quantità di dati in ingresso, è stato necessario implementare un reverse geocoder personalizzato. Nei paragrafi successivi vengono descritti i principali strumenti utilizzati quali *OpenStreetMap* come sorgente dati, *QGIS* come software

di elaborazione di dati geospaziali e *k-d tree* come strutture dati su cui eseguire l'algoritmo di ricerca dei *nearest neighbor*.

2.2 OpenStreetMap

Figura 2.1: Logo di OpenStreetMap

Nel 2004, un imprenditore di nome Steve Coast dà vita a OpenStreetMap (OSM), un progetto collaborativo finalizzato alla creazione e condivisione gratuita di mappe e di dati geospaziali. Grazie al contributo di una comunità di utenti, OpenStreetMap distribuisce le informazioni seguendo la licenza ODbL (Open Database License), la quale permette di usufruire, modificare e condividere i dati con il solo vincolo di citare la fonte e usare la stessa licenza per eventuali progetti futuri derivati. Si manifesta come alternativa al servizio Internet collaborativo offerto da *Google Map Maker*, un progetto che permetteva di correggere e espandere le mappe di Google Maps e nel quale ogni contributo fornito volontariamente dagli utenti sarebbe diventato di proprietà di Google, impedendone riutilizzi futuri. OpenStreetMap prende ispirazione da Wikipedia, dove la conoscenza del singolo individuo è messa a disposizione della comunità. Infatti, ogni utente registrato ha la possibilità di caricare dati geospaziali e apportare modifiche alle mappe preesistenti, in cui ogni correzione è memorizzata in un archivio storico mantenuto dalla comunità di OpenStreetMap. I dati per la realizzazione delle mappe derivano principalmente da traccie GPS caricate da volontari, fotografie aeree e dataset geospaziali donati da fonti governative e compagnie commerciali. Questa idea ha reso OpenStreetMap la banca dati cartografica libera più grande disponibile in rete; attualmente conta più di sei milioni di utenti registrati e più di quattro milioni di modifiche alla mappa ogni giorno [10].

2.2.1 Modello dei dati

Il modello dei dati di OpenStreetMap è rappresentato da *elements*, i quali si differenziano in:

- ***Node***: sono l'unico tipo di elemento che presenta come attributi la latitudine e longitudine, oltre che un campo identificativo. Rappresentano singoli punti di interesse o oggetti avente estensione molto piccola. Grazie alla presenza di coordinate, essi si rivelano necessari al fine di definire un percorso.

- ***Way***: una lista ordinata di almeno due nodi che presenta, oltre ad eventuali *tag*, anche un campo identificativo. In questo modo è possibile ottenere una geometria utilizzando le coordinate dei nodi di cui è composto. Grazie alla struttura della lista, è possibile identificare due tipi di percorsi; se il primo nodo non coincide con l'ultimo della lista allora l'elemento rappresenta una strada aperta (*open way*), se invece il primo nodo coincide con l'ultimo si parla di strada chiusa (*closed way*).

- ***Relation***: consiste in un gruppo di elementi, un campo identificativo e facoltativamente un insieme di tag. In particolare, ogni elemento della relazione è una coppia composta da un elemento, che può essere un node, way o relation, e opzionalmente un ruolo, ovvero una stringa di testo che descrive la funzione che l'elemento assume nella relazione. Sono state introdotte con lo scopo di modellare le relazioni logiche e geografiche tra gli elementi della mappa

- ***Area***: è il tipo di elemento riservato alla rappresentazione di aree. In realtà, in OpenStreetMap le aree non hanno una struttura dati esplicita. Infatti esse sono modellate a partire da elementi già preesistenti all'interno della mappa; nel caso in cui l'area da rappresentare è continua e non presenta buchi viene utilizzata una closed way e viene chiamata *simple area*, mentre qualora l'area presenta discontinuità viene chiamata *multipolygon area* ed è necessario l'ausilio di relations. In Figura 2.2 si mostra la differenza fra area semplice e area multipoligonale.

Il successo di OpenStreetMap è determinato, almeno in parte, dalla quantità e dalla specificità delle informazioni presenti all'interno delle mappe. Infatti, a differenza di come si potrebbe immaginare, a livello teorico è possibile aggiungere al database tutto ciò che è osservabile sul terreno. Per raggiungere questo obiettivo OpenStreetMap fa uso dei *tag*, ovvero coppie del tipo chiave=valore che specificano le caratteristiche di un particolare elemento. Per specificare meglio un elemento, è anche possibile creare una lista di tag. Una visione completa dei tag è mantenuta nella wiki ufficiale di OSM [11].

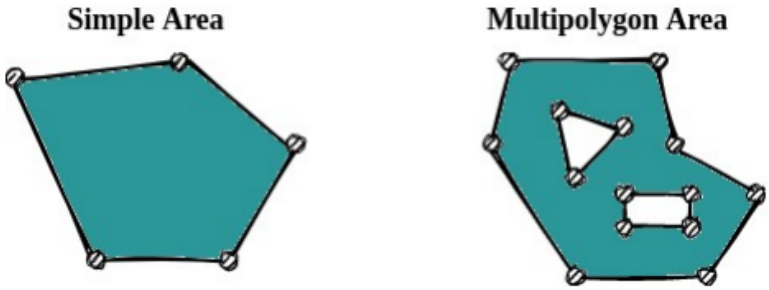

Figura 2.2: Differenza fra area semplice e area multidimensionale

2.2.2 Overpass API

I dati presenti all'interno del database di OpenStreetMap possono essere scaricati attraverso vari strumenti. La modalità più intuitiva è rappresentata dalla funzione Esporta presente nella pagina principale del sito web *openstreetamp.org*, grazie alla quale è possibile scaricare i dati geospaziali relativi ad un'area geografica selezionata. Da citare anche il sito web *Geofabrik* [12] per recuperare mappe di dimensioni elevate come, per esempio, dati relativi ad interi continenti quali Africa e Europa. Inoltre OpenStreetMap dispone di una serie di API per recuperare, modificare e caricare dataset di dati spaziali. In questo progetto di tesi è stata utilizzata *Overpass API*, un' *Application Programming Interface* di sola lettura che permette il recupero dei dati dalle mappe di OpenStreetMap. È stata progettata per rispondere a richieste via Internet di software esterni, infatti il suo funzionamento è simile ad un database raggiungibile via web. Dopo che gli utenti definiscono le query in uno tra i due linguaggi di interrogazione supportati, ovvero *Overpass QL* e *Overpass XML* [13], vengono effettuate richieste *HTTP GET* all'API. Overpass recupera le informazioni richieste dai server di OpenStreetMap e li restituisce al client in formato XML o GeoJSON. I linguaggi di interrogazione supportati da questa API permettono la scrittura di query elaborate, infatti è possibile selezionare elementi con determinate caratteristiche all'interno di un'area geografica circoscritta. Tuttavia può risultare complesso scrivere interrogazioni in linguaggio Overpass QL. Per ovviare a questo problema Martin Raifer ha reso disponibile *Overpass Turbo*, un'applicazione web-based in grado di tradurre una richiesta dell'utente in una query in linguaggio comprensibile alle Overpass API [14].

2.3 QGIS

QuantumGIS (QGIS) è un sistema informativo geografico open source e multipiattaforma che permette di analizzare, mantenere e visualizzare dati geografici.

Figura 2.3: Logo di QGIS

Questo progetto nasce nel Maggio 2002 per poi essere rilasciato nella sua versione 1.0 sotto licenza GNU General Public License nel Gennaio 2009. Grazie alle numerose funzionalità che implementa, questo software è considerato uno dei sistemi informativi geografici più popolari disponibili in commercio. Il cuore di QGIS è sviluppato in C++ e fa uso della libreria Qt per fornire all'utente un'interfaccia semplice e reattiva. Inoltre supporta nativamente Python 3, grazie al quale è possibile creare Plugin personalizzati oltre che implementare i propri script ed eseguire comandi dalla console di QGIS Desktop. Oltre alla semplicità d'uso, una caratteristica fondamentale che ha reso QGIS uno dei principali sistemi informativi geografici è la possibilità di interfacciarsi facilmente con altri software di gestione dati geografici e di elaborare i dati in moltissimi formati raster e vettoriali, cioè i due modelli di dato geospaziali più comuni.

2.3.1 Funzionalità di QGIS

QGIS offre molte delle più comuni funzionalità GIS (geographic information system) grazie alle sue caratteristiche di base e ai plugin. Come riporta la documentazione ufficiale di QGIS [15], è possibile suddividere le funzionalità di base in sei categorie:

- visualizzazione dei dati;
- esplorazione dei dati e composizione di mappe;
- creazione, modifica, gestione ed esportazione dei dati;
- analisi dei dati;
- pubblicazione di mappe su Internet;
- estensione delle funzionalità di QGIS attraverso plugin;

2.3.2 Plugin di QGIS

QGIS presenta un'architettura a plugin. Questo significa che, per sopperire alle esigenze degli utenti, il software permette agli utilizzatori di estendere il programma con funzionalità personalizzate. Grazie a questa modularità, QuantumGIS si presta ad un numero praticamente infinito di casi d'uso. Si possono suddividere i plugin in due tipologie:

- ***Plugin core***: rappresentano i plugin ufficiali di QGIS e sono mantenuti dal team di sviluppo. Grazie alla loro utilità, questi plugin sono inclusi nell'installazione di base del software. Il linguaggio di programmazione utilizzato per tali plugin è C++ o Python. La tabella 2.1 mostra una lista dei plugin core più importanti attualmente disponibili.
- ***Plugin esterni***: rappresentano le funzionalità aggiuntive che possono essere aggiunte a QGIS. Sono memorizzati in repository esterni e mantenuti dai singoli autori. Grazie al plugin core chiamato *Plugin Installer* è possibile installare, disinstallare e attivare questo tipo di plugin. Il linguaggio di programmazione usato è Python.

Tabella 2.1: Plugin core di base di QGIS

Icona	Plugin	Descrizione
	Coordinate Capture	Acquisisce le coordinate di puntamento del mouse usando un sistema di coordinate preimpostato
	DB Manager	Permette di gestire e integrare i vari tipi di database spaziali supportati da QGIS, quali PostGIS, SpatiaLite, GeoPackage, Oracle Spatial e layers virtuali. Inoltre è possibile eseguire query SQL su un insieme di database spaziali

Tabella 2.1: Plugin core di base di QGIS

Icona	Plugin	Descrizione
	eVis	Consente agli utenti di collegare dati geocodificati quali fotografie e altri documenti di supporto ai dati vettoriali in QGIS. È quindi in grado di visualizzare immagini associate agli elementi di un vettore
	Geometry Checker	Controlla e corregge la geometria di un layer. Esempi di errori rilevati sono: nodi duplicati, eventuali buchi nei poligoni, elementi duplicati e molti altri [16]
	Georeferencer GDAL	Genera file di georeferenziazione per i raster
	GPS Tools	Contiene strumenti per importare e caricare dati GPS su una serie di dispositivi esterni
	GRASS	Integra gli strumenti di GRASS [17], un sistema informativo geografico libero
	MetaSearch Catalog Client	Permette di effettuare ricerche in cataloghi di metadati in modo semplice e veloce
	Offline Editing	Consente di modificare il contenuto di una sorgente dati non essendo collegati alla rete. Una volta che si dispone della connessione, è possibile applicare le modifiche al database originario
	Processing	Fornisce l'ambiente di elaborazione dei dati spaziali

Tabella 2.1: Plugin core di base di QGIS

Icona	Plugin	Descrizione
	Topology Checker	Attraverso un elenco di regole, questo plugin rileva errori topologici nei layer vettoriali

Lo studio preliminare delle funzionalità di QGIS mi ha permesso di selezionare i plugin utili alla realizzazione del reverse geocoder dell'applicazione reale. In particolare, le funzionalità base e i plugin esterni utilizzati sono:

- ***Points Along Geometry***: grazie ad una semplice interfaccia (Figura 2.4), QGIS permette di creare punti equidistanti su una linea o un qualsiasi poligono. Un'importante caratteristica di questa funzione è che i punti creati ereditano gli attributi dell'elemento su cui sono stati costruiti. Questo permette all'utente di sfruttare le caratteristiche dell'elemento di partenza per future analisi. Inoltre, i punti creati possiedono due ulteriori attributi che identificano la distanza dalla geometria e l'angolo della linea nel punto. È anche possibile specificare un offset grazie al quale l'algoritmo determina la distanza tra l'inizio e la fine della geometria entro cui creare i punti.

- ***Add Geometry Attributes***: consente di aggiungere una o più proprietà geometriche a elementi di un layer vettoriale in ingresso. Nello specifico, ritorna un nuovo layer vettoriale con lo stesso contenuto del layer in entrata, con l'aggiunta di attributi addizionali contenenti misure geometriche basate sul sistema di riferimento selezionato. Gli attributi aggiunti al vettore in ingresso dipendono dalle dimensioni e il tipo di geometria del layer selezionato e sono espressi dalla Tabella **??**.

- ***QuickOSM***: è un plugin esterno che permette di scaricare e importare agevolmente dati di OpenStreetMap nel progetto QGIS. QuickOSM dispone di un insieme di opzioni e filtri che danno la possibilità all'utilizzatore di interrogare il database geografico in maniera approfondita. Grazie ad un'interfaccia semplice ed intuitiva (Figura 2.5), l'utente ha la possibilità di selezionare le *feature* degli elementi geografici tramite coppie di tipo chiave-valore, oltre che il tipo di oggetto e l'area geografica su cui eseguire l'interrogazione. Inoltre l'utente ha la possibilità di definire la modalità di salvataggio dei layer scaricati. Al fine di garantire queste funzionalità, QuickOSM genera automaticamente la query personalizzata sulla base del valore dei filtri selezionati dall'utente (Figura 2.6) ed effettua il download dei dati tramite le Overpass API.

- ***OpenLayerPlugin***: è un plugin esterno costruito sulle funzionalità di *OpenLayers*, una libreria modulare ad elevate performance per la visualizzazione e

interazione con mappe e dati geospaziali [18]. Permette all'utente di caricare e visualizzare in layer separati mappe derivanti da varie sorgenti dati, quali OpenStreetMap, Google Maps, Wikimedia Maps, Bing Maps e Apple Maps.

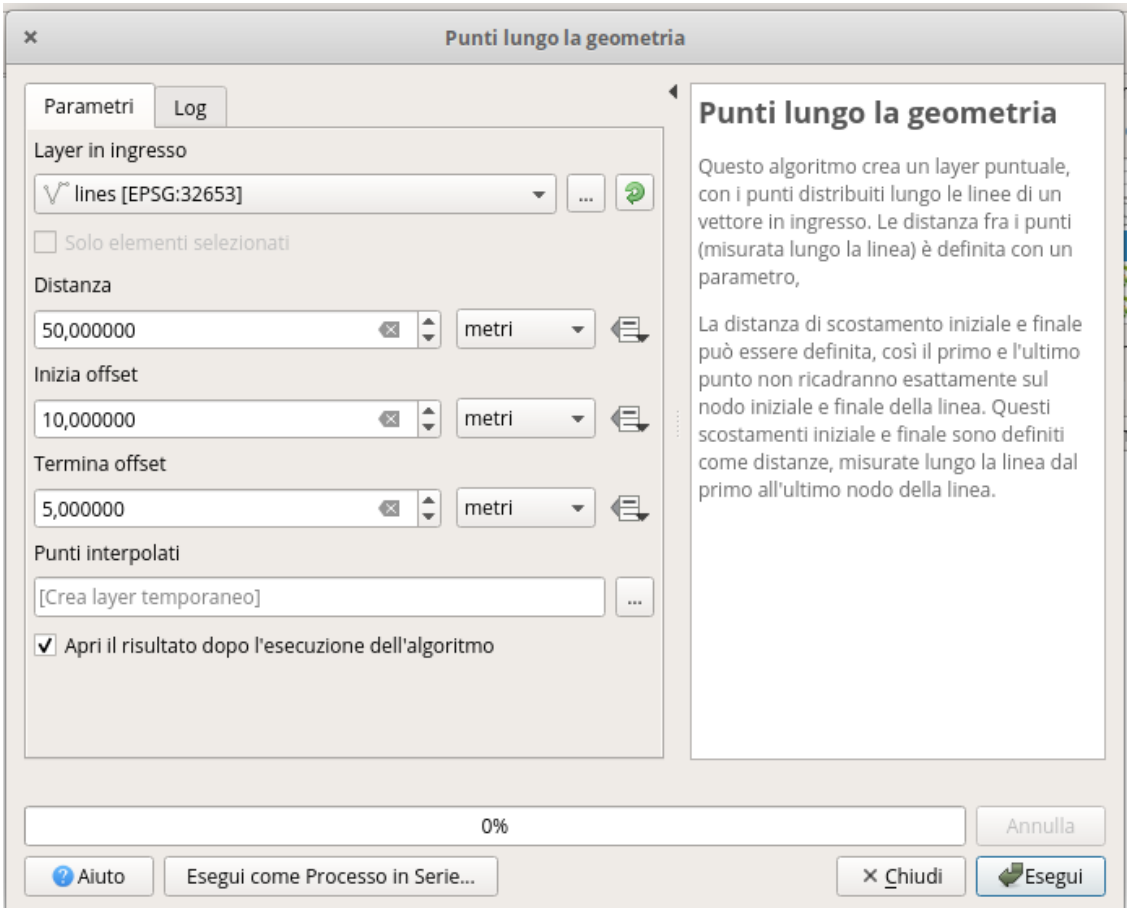

Figura 2.4: Points along geometry

Tabella 2.2: Attributi aggiuntivi per i diversi tipi di layer vettoriali in QGIS

Tipo di layer vettoriali in ingresso	Attributi aggiuntivi

Puntuali	• Coordinata X • Coordinata Y • Coordinata Z • M Value, un parametro opzionale che può essere utilizzato per registrare valori non coordinati in vari punti all'interno di una geometria. Permette di rappresentare una figura geometrica in quattro dimensioni (X,Y,Z,M)
Lineari	• Lunghezza della linea • Sinuosità della linea (solo per particolari tipi geometrici) • Distanza rettilinea (solo per particolari tipi geometrici)
Poligonali	• Perimetro • Area

2.4 K-d tree

Il K-d tree è un albero di ricerca multi-dimensionale in cui ogni nodo indicizza un punto in k dimensioni. È una struttura dati in cui ciascun elemento interno genera un iperpiano che suddivide lo spazio in due parti distinte, chiamate semispazi. Ogni livello dell'albero presenta la cosiddetta "dimensione dominante", la quale determina la direzione dell'iperpiano attraverso cui creare i semispazi. La dimensione dominante è individuata grazie ad una *splitting rule* (descritta al paragrafo 2.4.1), mentre la direzione dell'iperpiano risulta perpendicolare alla dimensione scelta. I punti che appartengono al lato sinistro del piano sono rappresentati dal sotto-albero di sinistra, mentre i punti che appartengono al lato destro del piano sono rappresentati dal sotto-albero di destra. Ad esempio, se la dimensione dominante è rappresentata dall'asse x, allora l'iperpiano creato sarà perpendicolare all'asse x. Inoltre, tutti i punti del sotto-albero avente un valore x minore saranno inseriti nel sotto-albero di sinistra, mentre tutti i punti con valore x maggiore saranno inseriti nel sotto-albero di destra.
Si può dire quindi che la struttura dati del k-d tree è basata su una divisione ricorsiva dello spazio in semispazi. Per effettuare una divisione ricorsiva dello spazio in semispazi è necessario definire due parametri:

- ***Splitting rules***: precetto a cui attenersi per selezionare il corretto iperpiano per dividere lo spazio in regioni rettangolari. Dato in ingresso un semipiano

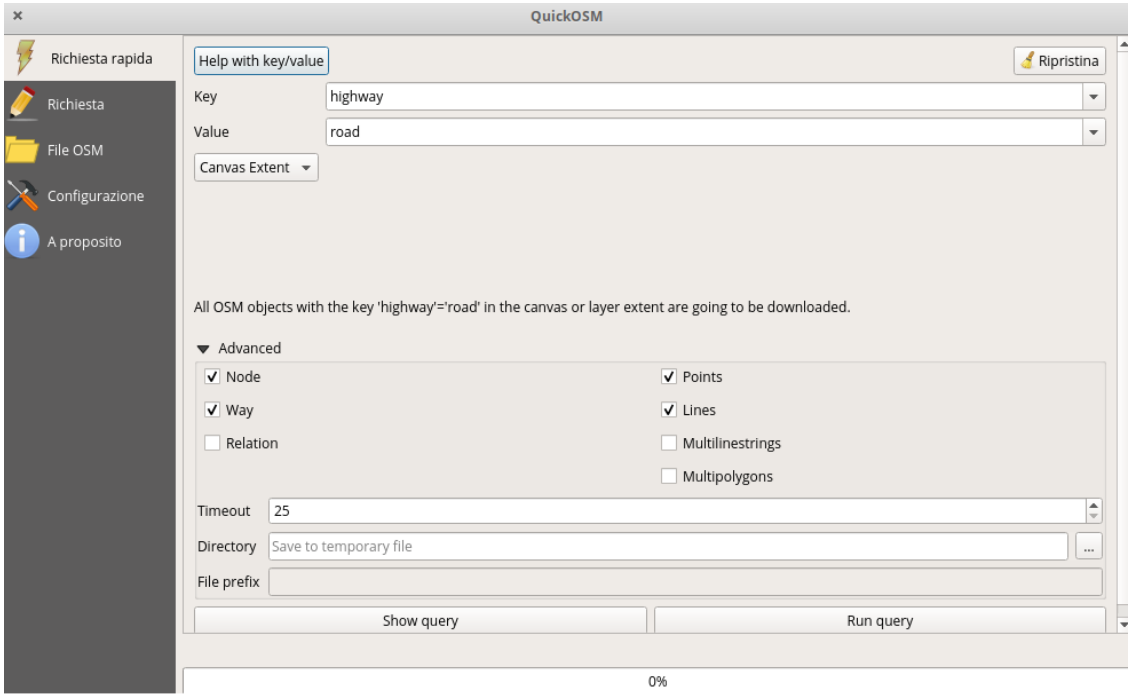

Figura 2.5: Interfaccia plugin QuickOSM

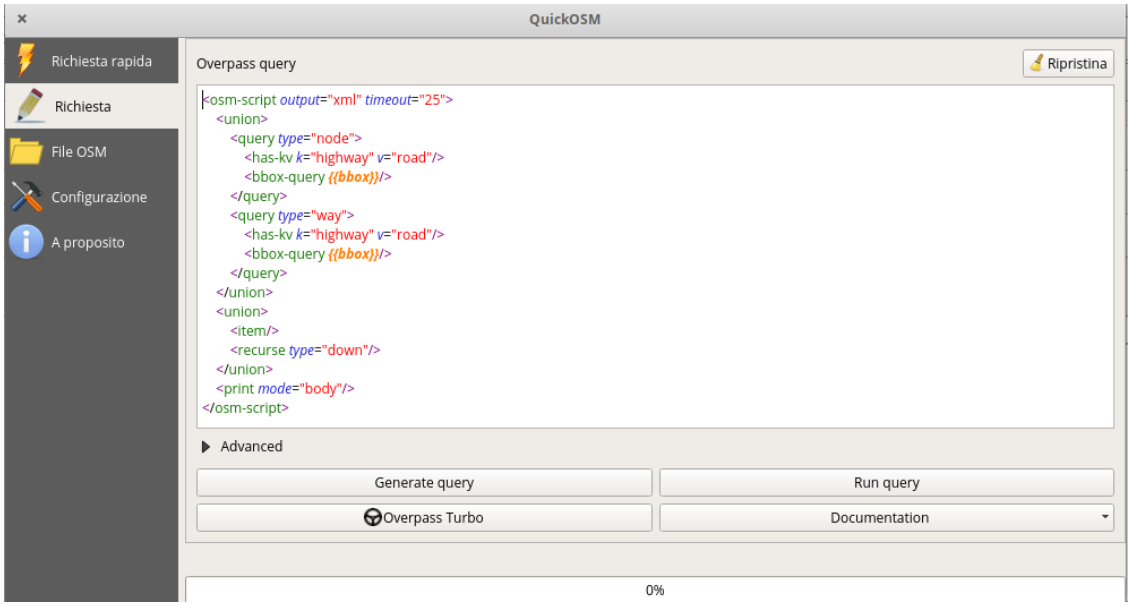

Figura 2.6: Query generata dal plugin QuickOSM

associato ad un nodo e i punti all'interno di esso restituisce la direzione e la posizione dell'iperpiano di taglio. La scelta della splitting rule determina la struttura e le performance dell'albero.

- ***Bucket size***: indica la densità dell'albero, ovvero il numero di punti massimo che possono essere presenti in un singolo semispazio. Se il numero di punti associati ad un nodo interno, anche detto *splitting node*, è maggiore del bucket size allora il semispazio viene suddiviso in due parti da un piano ortogonale agli assi selezionato dalla splitting rule.

2.4.1 Splitting rule

Dal momento che esistono diverse splitting rule, ci sono altrettanti modi differenti per creare un k-d tree. Ovviamente, la scelta della splitting rule influisce pesantemente sul bilanciamento e sulla struttura del k-d tree. Si definisce *splitting dimension* un numero intero che indica l'asse ortogonale all'iperpiano di taglio, mentre con il termine *splitting value* si fa riferimento alla posizione in cui l'iperpiano intercetta l'asse indicato dalla splitting dimension. Al fine di descrivere le principali splitting rules si introduce il seguente formalismo: sia Y il sottoinsieme di punti contenuti nel semipiano corrente B e sia lo *spread di* Y la differenza tra la coordinata maggiore e quella minore tra tutti i punti di Y lungo una dimensione scelta. Le regole di taglio più comuni sono:

- ***Standard splitting rule***: rappresenta la regola di divisione più conosciuta e comune, in cui la splitting dimension è la dimensione del massimo spread di Y, mentre lo splitting value è la mediana delle coordinate di Y lungo questa dimensione.

- ***Cyclic splitting rule***: in questa regola lo splitting value è analogo al caso precedente, mentre la splitting dimension è scelta in modo ciclico. Per esempio, in tre dimensioni quest'ultima. assumerà in ordine la dimensione dell'asse x, y, z, x, y e cosi via.

- ***Midpoint splitting rule***: la splitting dimension è la dimensione del lato più lungo di B, mentre lo splitting value è rappresentato dal punto medio di B. Nel caso in cui sono presenti più dimensioni aventi lato maggiore, viene selezionata la dimensione con spread maggiore.

- ***Sliding midpoint splitting rule***: come nel caso precedente, questa regola tenta di posizionare il piano di taglio nel punto medio del lato più lungo del semispazio. Tuttavia, se i punti di Y si trovano solamente su un solo lato del piano di taglio, l'iperpiano viene fatto scorrere fino ad incontrare il primo punto di Y. Formalmente lo splitting value diventa la coordinata del punto più vicino al piano di taglio lungo la splitting dimension.

La valutazione delle singole regole di taglio si basa sui seguenti parametri:

- dimensione dell'albero;
- profondità dell'albero;
- bilanciamento dell'albero;
- *Aspect ratio*: rapporto tra l'estensione nelle varie dimensioni di una cella. Più questo parametro è elevato più la cella si estende in una sola direzione, assumendo una forma lunga e stretta (o il contrario). Come dimostrato nel paper "*An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions*" [19], le performance dell'algoritmo per la ricerca approssimata di nearest neighbour dipende fortemente da questo parametro. Infatti, più l'aspect ratio è elevato più aumenta il numero di celle che devono essere analizzate e di conseguenza l'algoritmo ha costo maggiore. La figura 2.7 confronta il numero di celle da analizzare per ricercare il nearest neighbour nel caso di aspect ratio alto (albero di sinistra) e aspect ratio basso (albero di destra); il k-d tree di sinistra richiede l'analisi di sei celle mentre nel k-d tree di destra solo tre.

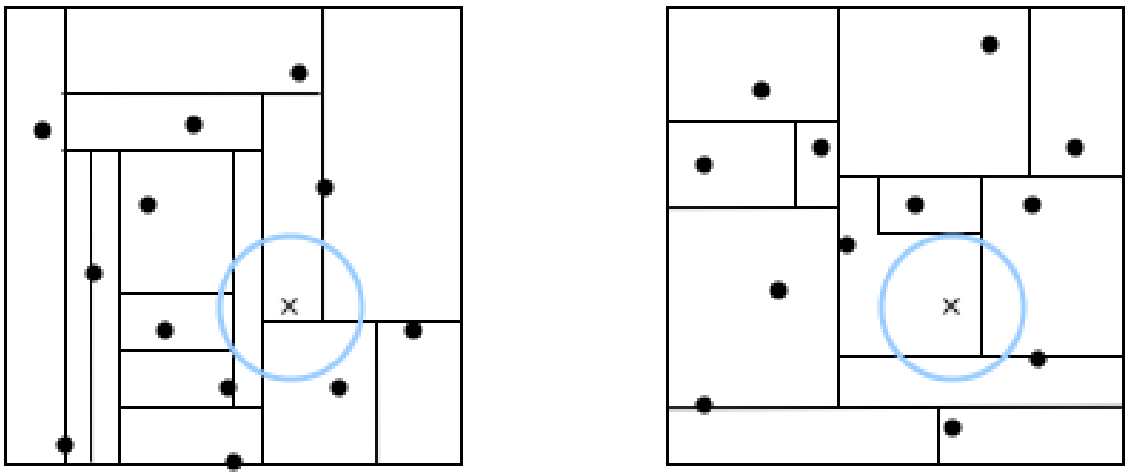

Figura 2.7: Confronto per la ricerca di nearest neighbour in caso di aspect ratio alto (a sinistra) e aspect ratio basso (a destra)

Si può dimostrare come, la standard splitting rule generi k-d tree bilanciati con profondità logaritmica *O(log n)* e con un numero di nodi lineare *O(n)*. Tuttavia, la costruzione degli alberi risulta costosa a causa del calcolo della mediana. Inoltre, nella maggior parte dei casi, i piani di taglio generati portano a celle con aspect ratio elevato. Al contrario, la midpoint splitting rule genera semispazi con aspect

ratio limitata ma può produrre piano di taglio banali, ovvero divisioni che creano celle vuote. Per questo motivo, la profondità e la dimensione dei k-d tree generati da questa regola possono superare anche complessità *O(n)*. Un buon compromesso è rappresento dalla sliding midpoint splitting rule, la quale non produce piano di taglio banali e limita la dimensione dell'albero a complessità *O(n)*. Dal momento che non è garantito il bilanciamento dell'albero, è possibile che la profondità dell'albero superi valori logaritmici *O(log n)* fino a raggiungere costi lineari *O(n)*. Inoltre è possibile rilevare poco frequentemente celle con elevato aspect ratio causate dallo slittamento del piano di taglio al primo punto rilevato, anche se in questo modo si evita l'esistenza di celle vuote.

2.4.2 Ricerca del nearest neighbour

La ricerca del *nearest neighbour*, ovvero "il vicino più vicino", rappresenta uno dei problemi più importante nel campo delle strutture dati geometriche. L'obiettivo di questo algoritmo è restituire in uscita il punto (o i punti nel caso di ricerca dei primi k nearest neighbour) che più sono vicini ad un punto dato. Al fine di ridurre il costo computazionale della ricerca, memorizzare i punti all'interno di un k-d tree può risultare molto utile; sfruttando le caratteristiche intrinseche del k-d tree è possibile eliminare velocemente una grande porzione di spazio di ricerca.
Semplificando, si può suddividere l'algoritmo in tre idee principali:

1. Si parte dalla radice. Visita il figlio destro se il punto in ingresso è maggiore del nodo attuale nella dimensione dominante, altrimenti visita il figlio sinistro.

2. Ogni volta che raggiungi un nodo foglia controlla la sua distanza con il nodo in ingresso e, se la distanza è minore di quella attuale, memorizzala in una variabile.

3. Quando l'algoritmo termina questo processo per il nodo radice, la ricerca del nearest neighbour è completa.

Una possibile implementazione di questo algoritmo è fornita dal seguente pseudocodice: algorithm algorithmic

Algorithm 1 Ricerca del nearest neighbour

Require: Punto x e radice del kd-tree
Ensure: Il punto xj X più vicino ad x.

Inizializza la distanza dist al valore infinito e crea la variabile nn per contenere il punto attuale più vicino a x
if then
è una foglia Calcola la distanza da x di ogni punto salvato in **if** l **then** a distanza corrente è minore di dist
Aggiorna dist ed nn
end if
end if
if then
è un nodo interno Visita il figlio di il cui semispazio associato contiene x. L'altro figlio dev'essere visitato solo se il semispazio ad esso associato dista da x meno della distanza corrente dist.
end if
return nn

Parte II

Progetto e sviluppo di un'applicazione reale

Capitolo 3

Progettazione

3.1 Introduzione

Il presente capitolo ha lo scopo di descrivere la progettazione dell'applicazione reale, analizzando i requisiti funzionali del sistema e motivando le scelte progettuali. In particolare, l'applicazione deve essere in grado di elaborare efficacemente, in tempo reale, stream di dati in ingresso, prodotti da una serie di infrastrutture contenute all'interno della *Smart Model Area* di Modena definita dal progetto *MASA* (Modena Automotive Smart Area) [20]. I dati devono essere mantenuti nel sistema di archiviazione nel loro formato grezzo, ovvero non elaborati; inoltre è necessario programmare un algoritmo per l'aggregazione temporale gerarchica dei dati al fine di mantenere in ulteriori strutture versioni dei dati a vari livelli di archiviazione. Si descriverà anche la progettazione di un reverse geocoder, implementato per trasformare le coordinate geografiche dei dati in ingresso in coppie del tipo (via, tratto) e utilizzare tale coppia come campo di aggregazione. Si studierà anche la realizzazione di un visualizzatore dei dati aggregati, grazie al quale l'utente è in grado di monitorare su una mappa interattiva il traffico all'interno della Smart Model Area di Modena a diversi livelli di aggregazione.

3.2 Strategia progettuale

La progettazione di un'applicazione complessa necessita di una strategia progettuale ben definita. Per questo motivo, in prima battuta, è stato necessario riassumere ad alto livello le fasi della progettazione evidenziate nello schema di Figura 3.1.

Dal momento che i dati sono generati e inviati al sistema, il framework di elaborazione streaming memorizza il flusso grezzo dei dati nel sistema di archiviazione. Successivamente, tramite un algoritmo di reverse geocoding, trasforma le coordinate X e Y dei dati in coppie del tipo (via, tratto), che saranno sfruttate nella successiva fase di aggregazione temporale gerarchica. Il visualizzatore dei dati è un

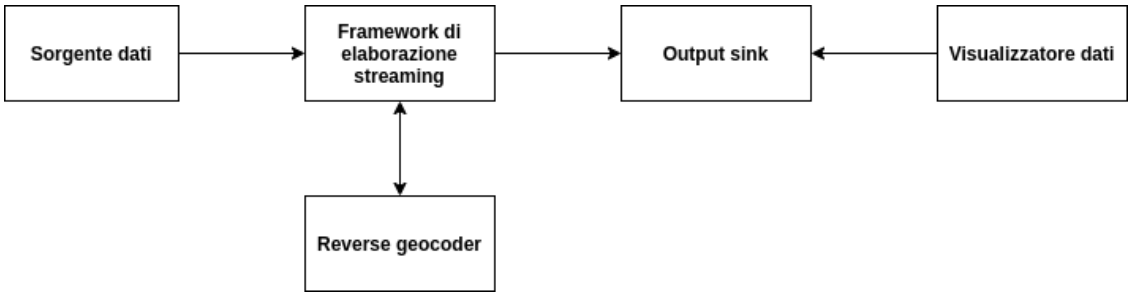

Figura 3.1: Pipeline concettuale

modulo esterno in grado di mostrare i dati aggregati all'utente su una mappa interattiva, dando la possibilità di selezionare il tipo di veicolo, la data e l'intervallo di aggregazione. Nei paragrafi successivi verranno descritti in dettaglio la progettazione dei punti chiave dell'applicativo: il framework di elaborazione streaming (sezione 3.3), il reverse geocoder (sezione 3.4) e il visualizzatore dei dati (sezione 3.5), mentre la sorgente dati è già stata esposta in sezione 1.7.

3.3 Framework di elaborazione streaming

Il panorama tecnologico attuale propone svariate tecnologie per il supporto dello *streaming computing*, ognuna delle quali è maggiormente indicata per ambiti applicativi diversi. Il framework utilizzato per elaborare lo stream di dati in tempo reale dell'applicazione è Apache Spark, il quale è stato descritto approfonditamente nel capitolo 1. Tuttavia, esistono molte alternative a questo framework, come Apache Storm, Apache Flink e Apache Samza; il seguente paragrafo ha il compito di sostenere la scelta di Apache Spark tramite una comparazione con i framework appena citati.
In primo luogo è possibile confrontare i due motori di elaborazione streaming presenti all'interno dell'ecosistema di Apache Spark: Spark Streaming e Structured Streaming. La prima grande differenza tra i due engine è il modello di esecuzione; Spark Streaming sfrutta un modello a micro-batch, mentre Structured Streaming introduce un nuovo modello di esecuzione chiamato continuous processing, il quale garantisce latenze di circa 1 millisecondo. Per manipolare il flusso dati Spark Streaming introduce l'astrazione DStream, la quale internamente è rappresentata come una sequenza di RDD, mentre Structured Streaming sfrutta le APIs DataFrame e Dataset. Come riporta la documentazione di Spark, i DataFrame forniscono prestazioni più elevate rispetto agli RDD e supportano le più comuni funzioni complesse per l'elaborazione del flusso dati, come per esempio l'aggregazione tramite finestre temporali. A conclusione di tali considerazioni, la scelta del motore di elaborazione streaming di Spark, che sarà comparato con altri framework, ricade su Structured

Streaming.
I criteri di valutazione sui quali si basa il confronto tra varie tecnologie di elaborazione streaming sono stati determinati in base alle caratteristiche dell'applicativo e sono i seguenti:

- performance in termini di capacità di trasmissione (thoughput) e latenza;
- complessità del framework;
- supporto al linguaggio Python;
- aspettative di crescita del framework;
- supporto a funzioni complesse di elaborazione streaming, come per esempio watermarking e aggregazione.

3.3.1 Valutazione delle performance

Il processo di elaborazione dati prevede tre diversi approcci: batch processing, micro-batch processing e stream processing. Ogni paradigma presenta vantaggi e svantaggi che influiscono sulla latenza e il throughput dell'applicativo. L'elaborazione batch è indicata per contesti applicativi nei quali non sono richieste latenze minime e il volume di dati da elaborare è molto elevato. L'elaborazione micro-batch è molto simile al batch processing, infatti la differenza maggiore risiede nella dimensione minore dei batch con la conseguenza di latenze inferiori. In ultimo, nello stream processing ogni record viene elaborato appena arriva in ingresso al sistema, consentendo al framework di raggiungere la latenza minima possibile. Tuttavia, tramite questo paradigma è difficile raggiungere la tolleranza agli errori senza compromettere il throughput, poiché è necessario controllare ogni record una volta che viene elaborato. Apache Spark e Apache Flink supportano sia il micro-batch processing sia lo stream processing, anche se il supporto di Spark Structured Streaming nei confronti dello stream processing è ancora acerbo e in fase di sviluppo, e quindi non considerato come una possibile scelta, mentre Apache Storm e Apache Samza si basano solamente sullo stream processing. Un riassunto delle performance di tali framework è mostrata in tabella 3.1, dove il rosso indica prestazioni mediocri, buone l'arancione e ottime il verde.

Tabella 3.1: Confronto delle performance tra i framework Apache Spark, Apache Storm, Apache Flink e Apache Samza

	Apache Spark	**Apache Storm**	**Apache Flink**	**Apache Samza**
Modello di elaborazione	Micro-batch processing e stream processing	Stream processing	Micro-batch processing e stream processing	Stream processing
Latenza				
Throughput				

3.3.2 Supporto al linguaggio Python

Apache Spark supporta diversi tipi di linguaggi di programmazione, quali Scala, Java, Python e R. A partire dalla versione 1.9, anche Apache Flink fornisce un supporto per Python, con la differenza che quest'ultimo framework provvede un sostegno più acerbo e ancora in evoluzione. Come riportato nella documentazione ufficiale [21], Apache Storm è un framework multi-linguaggio e quindi, teoricamente, è possibile utilizzarlo con qualsiasi linguaggio di programmazione. Tuttavia, per eseguire codice in linguaggio non JVM, è necessario scrivere alcune classi Java per invocare tali istruzioni. Il framework Apache Samza supporta solamente Java e Scala.

3.3.3 Aspettative di crescita

Come riportato dalla società Databricks, Spark rappresenta la piattaforma di elaborazione streaming open source più popolare sul mercato. Infatti possiede una community con più di 1000 collaboratori da oltre 250 organizzazioni [22]. Questo dato conferma quanto le ottime aspettative di crescita di Apache Spark.

3.3.4 Supporto a funzioni complesse di elaborazione streaming

Apache Storm è indicato per contesti di elaborazione streaming non complessi, infatti non supporta la maggior parte delle funzioni di elaborazione avanzate. Anche Apache Samza non permette alcune importanti funzioni, come per esempio il watermark e i trigger. Al contrario, Apache Spark e Apache Flink supportano pienamente molte funzionalità di streaming avanzate.

3.3.5 Conclusioni

A conclusione di questo breve confronto, i framework che più risultano indicati allo scenario dell'applicativo sono Apache Spark e Apache Flink. Anche se quest'ultimo ha prestazioni più elevate rispetto a Spark in termini di latenza a parità di throughput, Spark è preferibile in quanto è più promettente, e dispone di una documentazione Python più dettagliata. Inoltre l'API Python di Apache Spark è stabile e consolidata, mentre quella di Flink risulta ancora acerba.

3.4 Reverse geocoder

In questo breve paragrafo si descrive a grandi linee le idee lo schema concettuale del reverse geocoder, rappresentato in Figura 3.2.

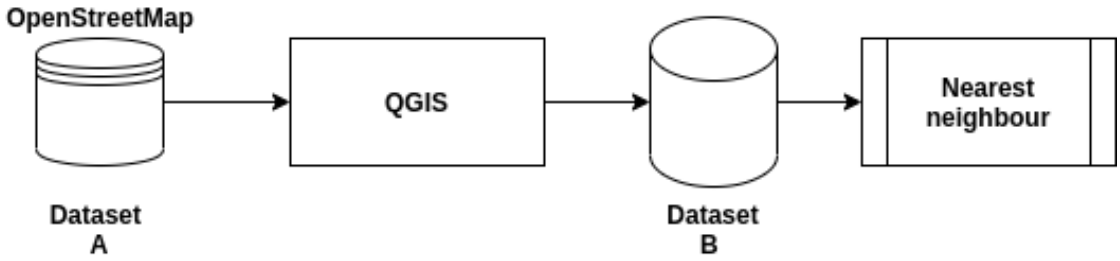

Figura 3.2: Schema concettuale del reverse geocoder

Il Dataset fornito da OpenStreetMap viene elaborato dal sistema informativo geografico QGIS, descritto al paragrafo 2.3, al fine di ritornare un nuovo dataset in cui ogni via è stata suddivisa in più tratti diversi aventi una lunghezza media di 50 metri.
Successivamente tale dataset viene caricato su un k-d tree fornito dalla libreria scipy [23], grazie al quale il framework di elaborazione esegue l'algoritmo di ricerca dei nearest neighbour per associare ad ogni record in entrata la coppia (via, tratto) più vicina.

3.5 Visualizzatore dati

L'obbiettivo di questa fase è implementare un modulo per la visualizzazione degli aggregati ottenuti dalle elaborazioni precedenti. Infatti, si vuole fornire all'utente le risorse per monitorare il traffico ai vari aggregati temporali, con la possibilità di differenziare il traffico in base al tipo di veicolo.
Dopo uno studio preliminare, al fine di sviluppare il reverse geocoder, sono stati selezionati i seguenti strumenti:

- Python 3;
- PySpark: API Python di Apache Spark;
- Jupyter Notebook: una *Open-Source Web Application* che permette di creare e condividere documenti in formato Web. È costituita da due componenti principali: un *kernel* e una *dashboard*. Il kernel è un processo specifico del linguaggio di programmazione scelto che interagisce con le applicazioni Jupyter e le loro interfacce utente per fornire le relative risposte. Il kernel Jupyter di riferimento è rappresentato da *IPython*, un interprete della riga di comando che permette di lavorare con Python. Inoltre, grazie al sostegno di molti altri kernel [24], Jupyter supporta numerosi linguaggi di programmazione come per esempio C++, R e JavaScript. In ultimo, oltre all'analisi dei dati e a modelli di machine learning, Jupyter Notebook dà la possibilità di creare dei *widget* in modo semplice e veloce, rendendo così il programma interattivo;
- Folium: una libreria Python con lo scopo di visualizzare dati geospaziali identificati da coordinate geografiche su una mappa interattiva;
- Numpy: una libreria Python oper source che fornisce strutture dati ad alte prestazioni quali array multidimensionali e matrici, insieme a numerosi strumenti per operare efficacemente su questi elementi.

La Figura 3.3 mostra il mockup del visualizzatore, comprendente filtri per selezionare la data, il tipo di veicolo e la granularità degli aggregati oltre che uno slider per impostare l'orario e la mappa interattiva.
In particolare, la mappa interattiva ha il compito di mostrare il traffico presente grazie una *palette* composta dai seguenti colori: verde, arancione e rosso. Il criterio di valutazione del traffico, ovvero la regola che determina quando esso è scorrevole o inteso, deriva direttamente dai flussi di traffico resi disponibili dalla Regione Emilia-Romagna [25]. Si è notato come, fino ad un transito di 8000 veicoli al giorno, la regione valuta il traffico come scorrevole, da 8000 a 18.000 rallentamenti e da 18.000 in poi traffico intenso. In sezione 4.3.3 è descritta la corrispondenza tra densità di veicolo e colore del traffico sulla mappa interattiva.

3.6 Aggregazione temporale gerarchica

Il sistema, oltre al flusso dei dati grezzi, vuole mantenere anche il flusso dei dati aggregati a varie granularità. Per fare questo è stata progettata un'aggregazione gerarchica, la quale permette di calcolare gli aggregati a partire da risultati ottenuti precedentemente.
Il primo passo nel processo di aggregazione consiste nell'aggregare i dati degli eventi nella granularità più fine richiesta. Successivamente, è possibile utilizzare questa

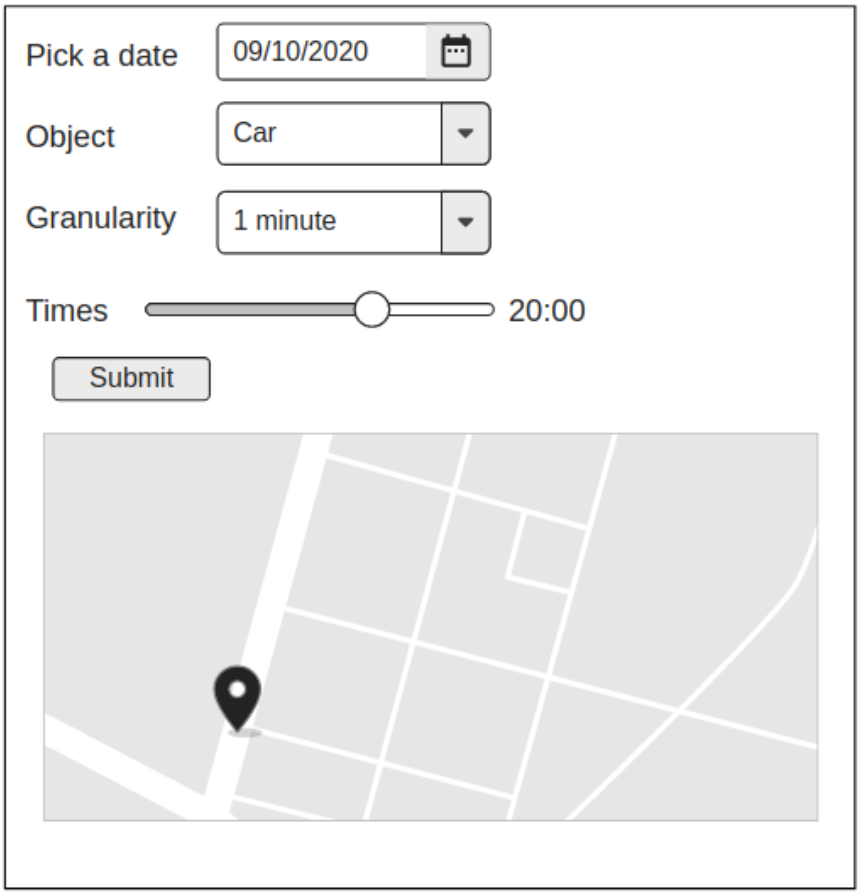

Figura 3.3: Mockup del visualizzatore

aggregazione per generare la granularità del livello successivo, ripetendo questo processo fino a quando non sono state generate tutte le viste richieste.
In particolare, il sistema mantiene le seguenti granularità:

- 1 minuto
- 15 minuti
- 1 ora
- 1 giorno

L'idea iniziale è rappresentata dalla Figura 3.4: dopo aver calcolato l'aggregato con granularità più fine (1 minuto), si sfrutta questo risultato per generare il successivo aggregato (15 minuti) fino ad arrivare all'aggregato con granularità maggiore (1 giorno).

Figura 3.4: Idea iniziale dell'aggregazione temporale gerarchica

Tuttavia, dopo uno studio approfondito delle funzionalità di Structured Streaming, si è notato come, ad oggi, Spark non supporta aggregazioni a catena su DataFrame e Dataset. Per ovviare a questa limitazione si è pensato a due possibili alternative:

- abbandonare l'idea di aggregazione gerarchica e calcolare ogni aggregato a partire dallo stesso streaming DataFrame (Figura 3.5). Non avendo una gerarchia, questa soluzione ha come unico vantaggio la semplicità di implementazione non usufruendo dei benefici introdotti dall'aggregazione gerarchica;
- mantenere l'idea di gerarchia pur con qualche complicazione. In particolare, dopo aver calcolato l'aggregato con granularità più fine, esso verrà memorizzato nel sistema. Quindi, per calcolare l'aggregato successivo, si caricherà in un DataFrame i risultati appena ottenuti, si effettuerà l'aggregazione desiderata e si memorizzerà nel sistema (Figura 3.6).

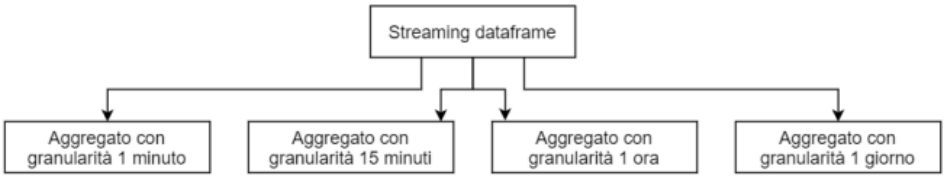

Figura 3.5: Alternativa all'aggregazione gerarchica temporale

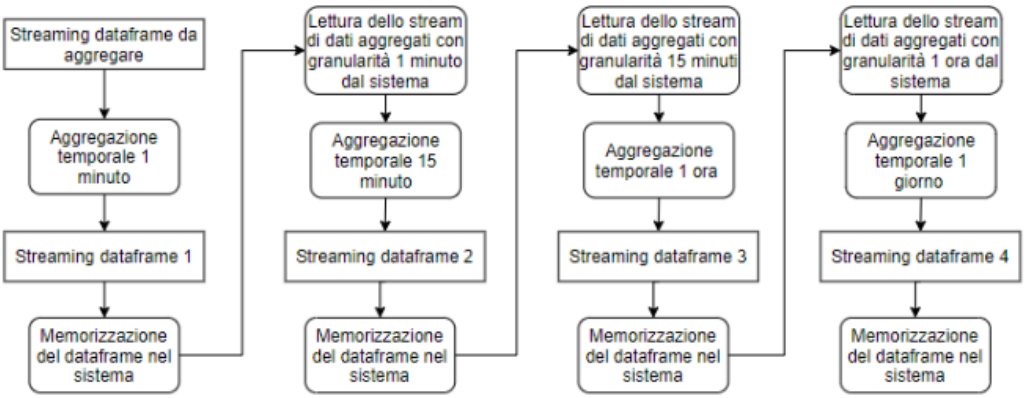

Figura 3.6: Schema dell'aggregazione gerarchica temporale adottata nell'applicativo

Tra le due soluzioni proposte la scelta ricade sulla seconda, anche se le continue scritture e letture dal sistema avranno ripercussioni sulle performance. La speranza è che Spark fornisca in futuro un supporto per aggregazioni a catena su streaming dataframe, in modo da evitare continue letture dal sistema. In aggiunta, ad ogni aggregazione, il volume di dati da leggere e scrivere sarà notevolmente inferiore rispetto al livello precedente.

Capitolo 4

Implementazione

Come già anticipato, la progettazione dell'applicativo ha permesso di suddividere in moduli indipendenti le varie funzionalità richieste, quali il reverse geocoder, l'algoritmo di elaborazione del flusso dati e il visualizzatore. Questo approccio ha semplificato la fase di implementazione ed è stato possibile concentrarsi su un solo modulo alla volta. Il presente capitolo ha l'obiettivo di commentare le principali funzioni implementate con il relativo codice applicativo.

4.1 Implementazione del Reverse geocoder

L'implementazione del reverse geocoder può essere suddivisa in due parti: una parte atta alla creazione del dataset contenente informazioni geografiche relative ai tratti delle vie, e una seconda parte predisposta all'implementazione dell'algoritmo per la ricerca del nearest neighbor.

4.1.1 Elaborazione dei dati geospaziali

Il software scelto per scaricare ed elaborare i dati di OpenStreetMap è QGIS. In particolare si è usufruito dei seguenti plugin (descritti in sezione 2.3.1):

- *OpenLayerPlugin*
- *QuickOSM*
- *Points Along Geometry*
- *Add Geometry Attributes*

In primis, tramite il plugin *OpenLayerPlugin* è stato possibile caricare la mappa di OpenStreetMap su un layer di QGIS, così da visualizzare i dati su un prospetto geografico. Questo plugin è raggiungibile dall'interfaccia di QGIS selezionando *Web*

→ *OpenLayers plugin* → *OpenStreetMap* → *OpenStreetMap*. Dopo un opportuno zoom per evidenziare Smart Model Area, il risultato di questa operazione è mostrato in Figura 4.1.

Figura 4.1: Mappa di Smart Model Area

In seguito, il processo di esportazione dei dati geospaziali dal database di OpenStreetMap è semplificato dal plugin QuickOSM, un programma non autonomo in grado di tradurre le richieste dell'utente in una query da presentare al database di OSM. Per accedere a questo plugin è necessario cliccare sulla voce *Vettore* → *QuickOSM* → *QuickOSM*. L'interfaccia presentata all'utente dispone di una serie di filtri, grazie ai quali è stato possibile esportare tutti i percorsi aventi come chiave *highway* ed un qualsiasi valore (dove highway è la chiave usata per identificare il tipo di strada) e, con l'opzione "*canvas extend*" specificare l'area su cui estrarre i dati. In questo modo è stato possibile caricare ogni tipo di percorso relativo alla Smart Model Area sulla mappa di OpenStreetMap (Figura 4.2).

Ogni percorso mantiene i seguenti attributi:

- *full_id*: stringa che identifica univocamente un percorso;
- *osm_id*: identico a *full_id* privato del primo carattere;
- *osm_type*: campo che identifica il tipo di elemento;
- *highway*: stringa che specifica il tipo di percorso;

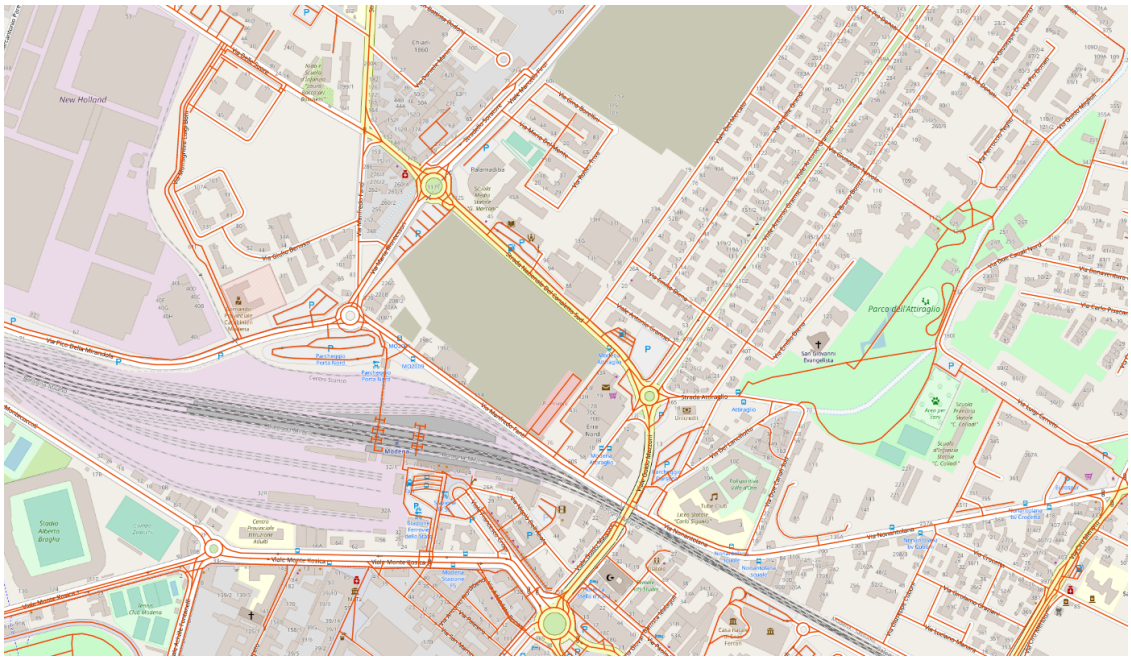

Figura 4.2: Mappa di Smart Model Area con i relativi percorsi

- *name*: nome della via;
- *postal_code*: codice postale
- *service*: se il campo highway ha valore "service", questo campo contiene il tipo di servizio che offre il percorso. Un esempio è il valore "bus", che specifica che il percorso è riservato gli autobus;
- *lanes*: numero di corsie;
- *oneway*: ha valore "yes" se il percorso è a senso unico;
- *maxspeed*: numero intero che indica la velocità massima;
- *surface*: specifica le condizioni della superficie del percorso, esempio "*paved*" o "*asphalt*";
- *bridge*: ha valore "yes" se il percorso transita sotto un ponte;
- *layer*: descrive le relazioni verticali degli elementi;
- *lit*: ha valore "yes" se il percorso è illuminato.

Molti di questi campi non hanno utilità nel progetto, e per questo scartati. In particolare, gli attributi ritenuti importanti sono: *full_id*, *highway*, *name*, *lanes*, *bridge* e *lit*.
Al fine di suddividere le vie in diversi tratti si è sfruttato il plugin di nome *Points Along Geometry*, accessibile da *Processing* → *Strumenti* → *Geometria vettore* → *Punti lungo la geometria*. Dopo aver impostato una lunghezza di 50 metri ed avviato l'elaborazione, il risultato è un nuovo layer contenente un insieme di percorsi suddivisi in più tratti di una lunghezza prestabilita (Figura 4.3).

Figura 4.3: Suddivisione di un percorso in tratti

Una volta terminata questa elaborazione, sono stati evidenziati una serie di punti critici in prossimità di rotonde o in occasioni di nodi interstradali. Se i primi punti critici erano pochi e potevano essere corretti manualmente, il grande numero dei secondi ha costretto alla ricerca di una soluzione alternativa. Per fortuna il plugin *Points Along Geometry* dispone di una funzionalità di offset, la quale permette di slittare di uno specifico valore il punto iniziale e finale creato su una geometria.
La Figura 4.4 mostra il problema dei nodi interstradali; il primo punto creato sul percorso di nome "*Via del Mercato*" è posizionato nel punto di intersezione con il percorso "*Strada Nazionale Canaletto Sud*", e quindi sulla via sbagliata. Dopo aver applicato un offset di 7 metri, il punto iniziale del percorso "*Via del Mercato*" risulta corretto (Figura 4.5).

I punti appena creati ereditano tutti gli attributi dell'elemento padre su cui sono

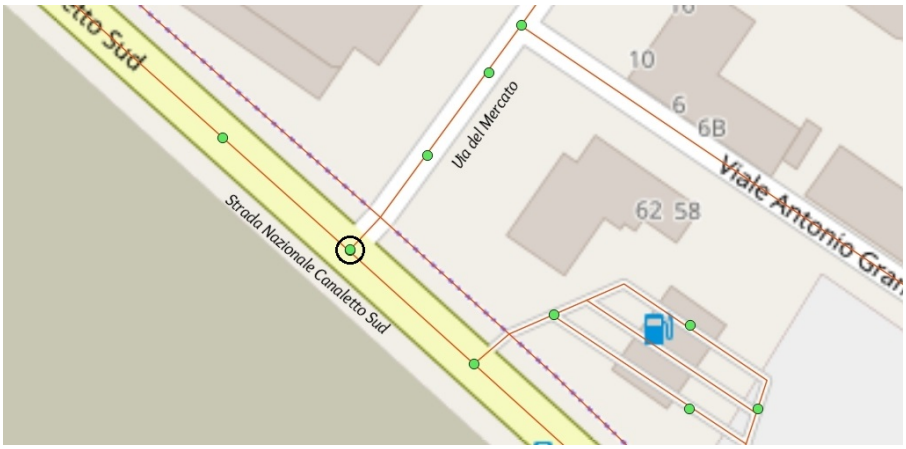

Figura 4.4: Criticità dei nodi interstradali

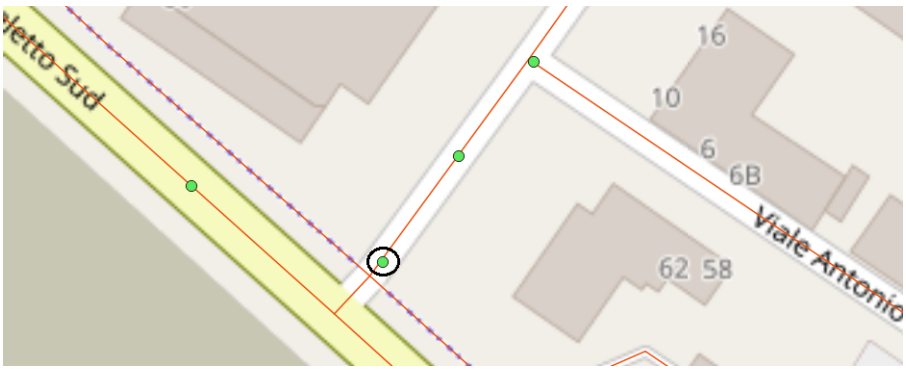

Figura 4.5: Applicazione di un offset ai nodi interstradali

stati creati: questo significa che, ogni punto di una via, ha lo stesso identificatore del percorso stesso. Al fine di identificare univocamente ogni punto di una via è stato aggiunto un attributo "*unique_id*" che contiene un numero intero univoco per ogni via: pertanto ogni punto è identificato dalla coppia (*id_via, unique_id*).
In ultimo, per aggiungere latitudine e longitudine ai punti si è utilizzato il plugin *Add Geometry Attributes*, accessibile da *Processing* → *Strumenti* → *Geometria vettore* → *Aggiungi attributi della geometria*.
La Figura 4.6 mostra il dataset appena elaborato sulla mappa di OpenStreetMap.

4.1.2 Costruzione del k-d tree

Per la realizzazione del k-d tree si è usufruito della libreria Python *SciPy*. Questa libreria, nella costruzione dell'albero, utilizza la *sliding midpoint rule* (descritta in

Figura 4.6: Dataset corretto visualizzato sulla mappa di OpenStreetMap

sezione 2.4.1) ed implementa un algoritmo efficiente per la ricerca del nearest neighbors.
Il listato 4.1 mostra la struttura principale della classe *GeocodeData*, la quale contiene due metodi: *read()* e *query()*. La funzione *read()* è invocata nell'*init* e, dopo aver caricato il file csv contenente i dati geospaziali elaborati precedentemente, ritorna le liste *coordinates* e *locations* , le quali mantengono rispettivamente le coordinate e le informazioni geografiche dei record del file. In seguito, grazie alla prima lista è possibile creare l'albero di ricerca. La funzione *query()* ha il compito di ricercare il "vicino più vicino" ad una coppia di coordinate in ingresso. Questo è reso possibile dal metodo *query()* proprio di *Scipy*, il quale ritorna l'indice relativo al punto più vicino e, di conseguenza, *locations[indices]* ritorna le informazioni geografiche del punto recuperato.

```
1  class GeocodeData:
2      def __init__(self, geocode_filename):
3          self.coordinates, self.locations =
               self.read(geocode_filename)
4          self.tree = KDTree(self.coordinates)
5
6      def read(self, geocode_filename):
7          coordinates, locations = [], []
8          if os.path.exists(geocode_filename):
9              rows = csv.reader(open(geocode_filename))
10         else:
11             print("File non trovato")
12             return
13         for
               full_id,highway,name,lanes,bridge,lit,unique_id,x,y
               in rows:
14             """Inserisco la tupla lat, lon come coordinata
                   """
15             coordinates.append((y, x))
16             """ Inserisco via, numero_civio, quartiere e id
                   in un dizionario, poi aggiungo il dizionario
                   """
17             locations.append(dict(name=name,
                   highway=highway, lanes=lanes, bridge =
                   bridge, lit=lit, id=full_id,
                   unique_id=unique_id))
18         return coordinates, locations
19
20     def query(self, coordinates):
21         """Find closest match to this list of coordinates"""
22         try:
23             """k = numero di vicini da ritornare"""
24             distances, indices =
                   self.tree.query(coordinates, k=1)
25         except ValueError as e:
26             logging.info('Unable to parse coordinates:
                   {}'.format(coordinates))
27             raise e
28         else:
29             #results = [self.locations[indices]]
30             results = self.locations[indices].values()
31             return results
```

Listato 4.1: Reverse Geocoder

4.2 Elaborazione del flusso dati

4.2.1 Configurazione dell'applicazione

Per accedere alle funzionalità offerte da Spark è necessario creare un'istanza della classe *SparkSession* tramite il metodo *SparkSession.builder()* (listato 4.2).

```
1 # Creazione della SparkSession
2     spark = SparkSession \
3         .builder \
4         .appName("Streaming application) \
5         .getOrCreate()
```

Listato 4.2: Creazione della SparkSession

Successivamente, in accordo con la struttura del flusso dati in ingresso, è stato definito uno schema tramite la classe *StructType*, la quale è composta da una serie di campi definiti attraverso la classe *StructField* (listato 4.3).

```
1  # Definizione dello schema
2     UserSchema = StructType([
3         StructField("timestamp", LongType(), True),
4         StructField("id_object", IntegerType(), True),
5         StructField("latitude", FloatType(), True),
6         StructField("longitude", FloatType(), True),
7         StructField("speed", FloatType(), True),
8         StructField("yaw", FloatType(), True),
9     ])
```

Listato 4.3: Definizione dello schema

Nello specifico, lo schema creato è composto dai campi descritti in tabella 4.1. Può sembrare strano creare un campo per identificare il timestamp di tipo Long e non di tipo Timestamp. Questa scelta è motivata dal fatto che la classe *TimestampType* di Apache Spark è di tipo *datetime.datetime*, la quale estende la classe datetime di Python per rappresentare date nel formato yyyy-MM-dd'T'HH:mm:ss[.SSS][XXX]. Osservando i dati in ingresso, il timestamp è fornito nel formato *unix timestamp*, ovvero un valore che rappresenta un offset rispetto alla mezzanotte del primo gennaio 1970. Per questo motivo, in un primo momento il timestamp del flusso viene memorizzato in un campo LongType, per poi essere manipolato successivamente al fine di convertirlo nel tipo TimestampType.

Tabella 4.1: Schema del DataFrame predisposto alla lettura dello stream row

Nome del campo	Tipo
timestamp	Long
id_object	Integer
latitude	Float
longitude	Float
speed	Float
yaw	Float

4.2.2 Lettura del flusso dati

Con il codice 4.4 è possibile caricare nel DataFrame di nome *df* il flusso dati in ingresso rappresentato da file CSV presenti nella cartella "*stream*" grazie all'interfaccia fornita dall'oggetto *DataStreamReader* ritornato dall'attributo *readStream* di SparkSession.

```
1   # Caricamento dello stream
2     df = spark \
3         .readStream \
4         .option("sep", " ") \
5         .schema(UserSchema) \
6         .format("csv") \
7         .load("stream")
```

Listato 4.4: Lettura dello stream row

4.2.3 Scrittura dello stream row nel sistema di archiviazione

Per mantenere i dati nel sistema di archiviazione in formato grezzo è necessario memorizzare il DataFrame df precedentemente creato in file Parquet. Per fare ciò si fa uso dell'interfaccia fornita da *DataStreamWriter* ritornata dall'attributo writeStream del DataFrame *df*. Le opzioni specificate hanno il seguente significato:

- *path*: il percorso della cartella di destinazione;
- *checkpointLocation*: percorso della cartella contenente i checkpoint, obbligatoria al fine di garantire un'elaborazione fault-tolerant.

```
1     df.writeStream \
2         .queryName("Row stream") \
3         .format("parquet") \
4         .option("path", os.path.join(os.getcwd(), 'sink',
             'sink_stream_raw')) \
5         .option("checkpointLocation",
             os.path.join(os.getcwd(), 'checkpoint',
             'checkpoint_stream_raw')) \
6         .start()
```

Listato 4.5: Scrittura dello stream row nel sistema di archiviazione

Si noti come, nel listato 4.5 non sia da specificare l'output mode (descritte al capitolo 1.6.1), in quanto la scrittura su file supporta solamente la modalità *Append*.

4.2.4 Applicazione del reverse geocoder

Dopo aver importato il modulo *tree*, contenente il reverse geocoder, si istanzia un oggetto della classe *GeocodeData* passando come parametro il file contenente i dati geospaziali manipolati con il software QGIS (listato 4.6).

```
1     # Creazione dell'albero per il reverse geocoding
2     rev_tree = tree.GeocodeData(os.path.join(os.getcwd(),
         'reverse_geocoder', 'point.csv'))
```

Listato 4.6: Creazione del k-d tree

Successivamente si esplicita una *user definition function* (listato 4.7), ovvero una funzione column-based personalizzata che estende le funzionalità di Spark SQL. In particolare, *query_udf* contiene una funzione lambda (anche detta anonima) che, data una coppia di coordinate, restituisce una stringa contente le informazioni della via e del tratto più vicino alle coordinate di ingresso.

```
1     # Definizione della User Definition Function per il
         reverse geocoding
2     query_udf = F.udf(lambda a,b :
         ',',.join(rev_tree.query((a,b))), StringType() )
```

Listato 4.7: Creazione di una user definition function

In seguito sarà applicata questa user definition function alle colonne dello stream contenente le coordinate geografiche.

4.2.5 Creazione del campo TimestampType

Come già anticipato, il campo di nome timestamp non è il realtà di tipo TimestampType, bensì LongType. Questa è una limitazione, infatti Spark effettua aggregazioni temporali solamente su campi di tipo TimestampType. Per ovviare a questo problema si fa uso della funzione *from_unixtime()*, la quale converte un unix timestamp espresso in secondi in una stringa nel formato specificato. Disponendo di un timestamp con la precisione dei millisecondi, esso non è compatibile con questa funzione. Allora, l'idea è quella di creare una nuova colonna nel dataframe *df_modified* chiamata *timestamp_modified* di tipo TimestampType in cui inserire il timestamp nel formato yyyy-MM-dd HH:mm:ss, per poi concatenarla con le ultime tre cifre della colonna di nome timestamp, le quali rappresentano i millisecondi del marcatore temporale. In questo modo si ottiene una colonna di nome *timestamp_millisecond* di tipo TimestampType contenente il timestamp in millisecondi espresso nel formato yyyy-MM-dd HH:mm:ss.SSS (listato 4.8).

```
1  df_modified = df.withColumn("timestamp_modified",
2                          F.from_unixtime(df["timestamp"]
                              /1000, format='yyyy-MM-dd
                              HH:mm:ss').cast(TimestampType()))\
3          .withColumn("location",
4                      query_udf(df["latitude"],df["longitude"]))
5
6     df_modified= df_modified.withColumn("timestamp",
         F.regexp_extract(df["timestamp"], ".{3}$", 0))
7     df_modified =
         df_modified.withColumn("timestamp_millisecond",
         F.concat(df_modified["timestamp_modified"],
         F.lit('.'),
         df_modified["timestamp"]).cast(TimestampType()))
```

Listato 4.8: Creazione del campo TimeStampType

4.2.6 Aggregazione temporale gerarchica

Riassumendo, il risultato delle precedenti elaborazioni è uno streaming dataframe di nome *df_modified* il quale contiene i campi mostrati in tabella 4.2.

Tabella 4.2: Schema del DataFrame dopo una prima manipolazione

Nome del campo	**Tipo**	**Descrizione**
timestamp	string	Stringa contenente le ultime tre cifre del timestamp originario, di cui rappresenta i millisecondi
id_object	integer	Numero intero che identifica il tipo di oggetto rilevato
latitude	float	Latitudine
longitude	float	Longitudine
speed	float	Velocità
yaw	float	Imbardata
timestamp_modified	timestamp	Timestamp nella forma yyyy- MM-dd HH:mm:ss
timestamp_millisecond	timestamp	Timestamp nella forma yyyy- MM-dd HH:mm:ss.SSS
route_name	string	Nome della via
highway	string	Tipo di strada. Una lista dei possibili valori è mantenuta nella documentazione di OpenStreetMap [26]
lanes	string	Numero di corsie
bridge	string	Campo che identifica la presenza di un ponte sulla strada [27]
lit	string	Campo che identifica l'illuminazione della strada [28]
route_id	string	Stringa che identifica univocamente la strada

Tabella 4.2: Schema del DataFrame dopo una prima manipolazione

Nome del campo	Tipo	Descrizione
unique_id	string	Stringa che identifica il tratto di strada

L'obbiettivo è quello di ottenere, per ogni tratto di ogni via, la densità dei diversi tipi di veicoli ai vari aggregati temporali. Per fare ciò, lo Streaming DataFrame calcolato in precedenza viene manipolato tramite il codice 4.9. Dopo aver impostato un watermark di due minuti, si raggruppano i record aventi stesso timestamp, nome della via, tratto della via e generati dallo stesso oggetto. Successivamente, viene creata la colonna di nome *count* comprendente il numero di record raggruppati e la colonna *average_speed* contenente la media delle velocità (per adesso sempre con valore zero). Un esempio di funzionamento è mostrato in figura 4.7.

```
1     query = df_modified \
2         .withWatermark("timestamp_millisecond", "2 minutes")
             \
3         .groupBy("timestamp_millisecond", "route_name",
             "unique_id", "id_object")\
4         .agg(
5             F.count("*").alias("count"),
6             F.avg("speed").alias("average_speed"),
7             )
```

Listato 4.9: Calcolo della densità per veicolo

timestamp_millisecond	vehicle_type	speed	route_name	highway	lanes	route_id	section_id
2018-11-19 12:15:00.013	5	0	Strada Nazionale Del Canaletto Sud	secondary	1	w622677478	20
2018-11-19 12:15:00.013	6	0	Strada Nazionale Del Canaletto Sud	secondary	1	w622677478	20
2018-11-19 12:15:00.013	6	0	Strada Nazionale Del Canaletto Sud	secondary	1	w622677478	19
2018-11-19 12:15:00.044	6	0	Strada Nazionale Del Canaletto Sud	secondary	1	w622677478	19
2018-11-19 12:15:00.044	6	0	Strada Nazionale Del Canaletto Sud	secondary	1	w622677478	19

timestamp_millisecond	vehicle_type	route_name	section_id	count	average_speed
2018-11-19 12:15:00.013	5	Strada Nazionale Del Canaletto Sud	20	1	0
2018-11-19 12:15:00.013	6	Strada Nazionale Del Canaletto Sud	19	1	0
2018-11-19 12:15:00.013	6	Strada Nazionale Del Canaletto Sud	20	1	0
2018-11-19 12:15:00.044	6	Strada Nazionale Del Canaletto Sud	19	2	0

Figura 4.7: Esempio del calcolo della densità per veicolo

Dopodiché si memorizza il DataFrame *df_modified* in una serie di Parquet file nel sistema, con lo scopo di caricarlo successivamente su un ulteriore Streaming DataFrame per effettuare le aggregazioni temporali descritte al paragrafo 4.2.6.

In seguito (listato 4.10) si mostra il codice della prima aggregazione temporale gerarchica (granularità 1 minuto), che si compone di una parte di lettura dei dati dal sistema, una di aggregazione e una di scrittura del flusso manipolato in memoria (come descritto in sezione 4.2.6).

```
1    df_read = spark \
2         .readStream \
3         .schema(query.schema) \
4         .parquet(os.path.join(os.getcwd(), 'sink',
             'sink_stream_modified'))
5
6
7     df_oneminute = df_read\
8         .withWatermark("timestamp_millisecond", "2 minutes")\
9         .groupBy(F.window("timestamp_millisecond", "1
             minute"),
10                "route_name",
11                "unique_id",
12                "id_object")\
```

```
13          .agg(F.avg("count").alias("average_count"),
14               F.avg("average_speed").alias("average_speed"),)
15
16
17
18      df_oneminute = df_oneminute.withColumn("window_start",
           df_oneminute["window.start"]).withColumn("window_end",
           df_oneminute["window.end"])
19      df_oneminute = df_oneminute.drop(df_oneminute["window"])
20
21      df_oneminute\
22          .writeStream\
23          .queryName("Aggregazione 1 minuto") \
24          .format("parquet") \
25          .option("path", os.path.join(os.getcwd(), 'sink',
             'sink_one_minute')) \
26          .option("checkpointLocation",
             os.path.join(os.getcwd(), 'checkpoint',
             'checkpoint_one_minute')) \
27          .start()
```

Listato 4.10: Calcolo aggregato a granularità 1 minuto

4.3 Implementazione del visualizzatore dati

Il visualizzatore dei dati è stato implementato su Jupyter Notebook, una Open-Source Web Application che permette di creare e condividere documenti in formato Web. Questo modulo si serve dei file Parquet contente i dati aggregati generati dal motore di elaborazione streaming e il file che mantiene le informazioni sui dati geografici manipolati con il software QGIS.

4.3.1 Creazione della mappa interattiva

Il primo passo per implementare il visualizzatore dati è la creazione della mappa interattiva su cui mostrare il traffico. Per fare ciò si è creata un'istanza della classe *Map* resa disponibile dalla libreria *folium* con i seguenti parametri (listato 4.11):

- *location*: una coppia di coordinate che identificano il focus iniziale della mappa. Nell'applicativo corrispondono ai valori [44.655482, 10.930833], i quali rappresentano un punto all'interno della Smart Model Area;
- *zoom_start*: il valore dello zoom iniziale;
- *tiles*: stile della mappa.

```
map = folium.Map(location=[44.655482, 10.930833],
    zoom_start=15, tiles="CartoDb dark_matter")
```

Listato 4.11: Creazione della mappa interattiva

4.3.2 Creazione dei widget

Jupyter Notebook consente la creazione di *widget*, oggetti Python in grado di generare eventi e di conseguenza di costruire GUI interattive. Per questo modulo sono stati pensati cinque widget:

- un datepicker per selezionare la data;
- un dropdown per selezionare il tipo di oggetto;
- un dropdown per selezionare la granularità degli aaggregati;
- uno slider per impostare l'orario;
- un button per avviare la visualizzazione.

Il codice sorgente è mostrato al listato 4.12.

```
# Variabili globali
hours = '00'
minutes = '00'

def vehicle(x):
    return {
        'car': 6,
        'person': 14,
        'bus': 5,
        'motorbike': 13,
        'bicycle': 1,
    }[x]


def granularity(x):
    return {
        '1 minute': 'sink_one_minute',
        '15 minutes': 'sink_fifteen_minute',
        '1 hour': 'sink_one_hour',
        '1 day': 'sink_one_day',
    }[x]

```

```
23
24 def freq(x):
25     return {
26         '1 minute': '1min',
27         '15 minutes': '15min',
28         '1 hour': '1h',
29         '1 day': '1d',
30     }[x]
31
32 # Definizione dei widget
33 date_picker = DatePicker(description = "Pick a Date:")
34 display(date_picker)
35
36 mezzo = Dropdown(
37     options=['person', 'car', 'bus', 'motorbike', 'bicycle'],
38     value='car',
39     description='Object:',
40     disabled=False,
41 )
42
43 display(mezzo)
44
45 granularit  = Dropdown(
46     options= ['1 minute', '15 minutes', '1 hour', '1 day'],
47     value='1 minute',
48     description='Granularity:',
49     disabled=False,
50 )
51
52 selection_range_slider1 = widgets.Text()
53 btn = Button(description = "Go")
```

Listato 4.12: Creazione dei widget

Si noti come, ogni widget di tipo dropdown, necessita di una lista di valori che rappresentano le opzioni selezionabili nell'attributo *options* e un valore di default nell'attributo *value*.
Il widget di tipo slider richiede qualche accortezza in più. Esso infatti deve essere dinamico, in base alla granularità selezionata deve presentare intervalli differenti. Per questo motivo, è stata associata al widget che rappresenta la granularità la funzione del listato 4.13.

```
1 def on_change(change):
2     global hours
3     global minutes
4     global selection_range_slider1
```

```
5
6       start_time = time(0,0)
7       end_time = time(23, 59)
8
9       # Cambia scala dello slider e imposta il valore 00:00
10      new_times = pd.date_range(str(start_time),
           str(end_time),
           freq=freq(granularit .value)).strftime('%H:%M')
11      new_options = [date for date in new_times]
12      selection_range_slider1.options = new_options
13      hours = '00'
14      minutes = '00'
```

Listato 4.13: Definzione della funzione che modifica la ganularità dello slider

4.3.3 Visualizzazione del traffico

Dopo aver definito le funzioni che gestiscono gli eventi legati ai widget, viene implementata la funzione che aggiunge dei marcatori circolari sulla mappa interattiva al fine da identificare i tratti di ogni strada. Per fare questo viene caricato il file contenente i dati geospaziali e, tramite la funzione *CircleMarker*, si generano marcatori circolari sulla mappa con colore di default verde (listato 4.14).

```
1 locs =  zip(df[8], df[7])
2 for location in locs:
3     traffic_color = "#00FF00"
4     folium.CircleMarker(location=(location[0], location[1]),
         color = traffic_color, radius=2).add_to(map)
```

Listato 4.14: Aggiunta dei marcatori circolari sulla mappa

Dopodiché, se l'utente ha selezionato una data, si carica dal sistema i file parquet contenenti i dati aggregati con la granularità richiesta su un opportuno dataframe (4.15).

```
1  df2 = sqlContext.read.parquet(os.path.join(os.getcwd(),
      "sink", type_granularity , "*"))
```

Listato 4.15: Caricamento dei file parquet dal sistema

Poi, grazie alle funzionalità di Spark SQL, si ritornano tutti i record del file dataframe appena creato che hanno le caratteristiche richieste dall'utente; si controlla quindi il tipo di oggetto, la data e la granularità (listato 4.16).

```
1  # Ritorna tutte le righe che hanno veicolo, data e
     granularit  corrette
2 base_df = df2.select('average_count', 'name', 'unique_id',
    'window_start')\
3     .where(df2.id_object == type_vehicle)\
4     .where(dayofmonth(df2.window_start) == type_date.day)\
5     .where(month(df2.window_start) == type_date.month)\
6     .where(hour(df2.window_start) == hours)\
7     .where(minute(df2.window_start) == minutes)
```

Listato 4.16: Esecuzione di una query sul dataframe caricato dal sistema

Il risultato di questa query è passato come parametro alla funzione *customFunction()*, la quale ha il compito di colorare i tratti di via sulla mappa in base alla densità di veicoli presenti in un particolare intervallo di tempo. In particolare, se la densità di oggetti è:

- < 6 il traffico è scorrevole e il colore corrispondete è verde
- $>= 6$ e < 13 il traffico è scorrevole con qualche rallentamento e il colore corrispondete è arancione
- $>= 13$ il traffico è intenso e il colore corrispondete è rosso

Capitolo 5

Valutazione delle performance e screenshot dei risultati ottenuti

Il presente capitolo ha il compito di valutare le performance dell'applicativo e mostrare una serie di immagini relative ai risultati ottenuti.

5.1 Prestazioni e accuratezza del reverse geocoder

5.1.1 Valutazione delle performance del modulo di reverse geocoding

Per rendere quanto più performante l'elaborazione del flusso dati, è stato necessario implementare un reverse geocoder personalizzato. Le prestazioni di esso sono state confrontate con il reverse geocoder fornito da *Nominatim*, un tool che sfrutta le mappe di OpenStreetMap per ricercare dati a partire da nomi e coordinate [29]. Il primo passo per il testing delle performance è rappresentato dal listato 5.1, con il quale è stato possibile generare randomicamente 100 punti con coordinate appartenenti all'area di Smart Model Area. Poi, sono state eseguite le query relative ai 100 punti e monitorato i tempi di risposta. Tutto questo processo è stato ripetuto per 10 volte, in modo da ottenere una media delle prestazioni.

```
1        coords = [(str(random.uniform(44.6519,44.6598))[:9],
2        str(random.uniform(10.9255, 10.9408))[:9]) for _ in
            range(100)]
```

Listato 5.1: Generazione di 100 punti randomici con coordinate contenute in Smart Model Area

La tabella 5.1 espone i tempi di risposta dei due algoritmi. Dai risultati ottenuti si nota come, a partire dallo stesso numero di query, il reverse geocoder personalizzato ha prestazioni molto più elevate. Questo comportamento è derivato dai seguenti presupposti:

- Nominatim è un servizio gratuito, e per questo i server offerti hanno prestazioni limitate;
- il k-d tree creato con il reverse geocoder personalizzato ha dimensioni molto più limitate rispetto a quello sfruttato da Nominatim, e quindi la ricerca dei punti è notevolmente più veloce.

Tabella 5.1: Tempi di risposta dei reverse geocoder

Iterazione	**Reverse Geocoder Personalizzato**	**Nominatim**
1	0:00:00.014039	0:00:50.063994
2	0:00:00.011592	0:00:49.980574
3	0:00:00.009790	0:00:50.042449
4	0:00:00.009848	0:00:50.066510
5	0:00:00.009708	0:00:49.996321
6	0:00:00.009707	0:00:49.996112
7	0:00:00.009792	0:00:49.989532
8	0:00:00.008598	0:00:49.919067
9	0:00:00.007434	0:00:50.044599
10	0:00:00.007439	0:00:49.980206
Media	**0.0097947**	**50.0079364**

Questa verifica ha dimostrato la "bontà" del modulo di reverse geocoding; infatti, pur eseguendo il programma su un laptop con un hardware modesto (processore: AMD Ryzen 5 3500U con 8 GB di RAM), i tempi per elaborare 100 query sono inferiori ai 10 millisecondi.

5.1.2 Precisione del modulo di reverse geocoding

Oltre alle prestazioni, è importante che il reverse geocoder geolocalizzi senza errori e con buona precisione il flusso dati in ingresso. Per questo motivo, è stato ritenuto opportuno effettuare un breve studio sull'accuratezza dell'algoritmo implementato. Quindi, a partire da un insieme di coordinate in ingresso, tramite la funzione descritta al listato 5.2 e opportune ricerche all'interno del k-d tree, è stato possibile calcolare la distanza fra le coordinate in ingresso e le coordinate corrispondenti ai tratti stradali definiti in sezione 4.1.2.

```
1  def haversine(lon1, lat1, lon2, lat2):
2      # convert decimal degrees to radians
3      lon1, lat1, lon2, lat2 = map(radians, [lon1, lat1, lon2,
          lat2])
4      # haversine formula
5      dlon = lon2 - lon1
6      dlat = lat2 - lat1
7      a = sin(dlat/2)**2 + cos(lat1) * cos(lat2) *
          sin(dlon/2)**2
8      c = 2 * asin(sqrt(a))
9      # Radius of earth in kilometers is 6371
10     km = 6371* c
11     m = km * 1000
12     return m
```

Listato 5.2: Distanza tra due punti geografici

In Figura 5.1 è mostrato il grafico ricavato dai dati generati da tale analisi. Più nello specifico, la linea rossa rappresenta la media della distanza, e di conseguenza la precisione media; essa possiede un valore pari a 27.0159848 metri. Dal momento che i tratti di strada hanno una lunghezza prefissata uguale a 50 metri (sezione 4.1.1), tale valore è più che accettabile.

5.2 Prestazioni complessive dell'applicativo

La seguente sezione ha il compito di analizzare e valutare le prestazioni complessive del sistema, in termini di consumo di memoria, di CPU e di tempo di elaborazione. In questo modo sarà possibile rintracciare gli eventuali punti critici dell'applicativo per future considerazioni.
Per fare ciò, è stato caricato un flusso dati generato in un periodo di tempo di un'ora ed è stato monitorato il consumo di memoria e CPU delle singole aggregazioni temporali, oltre che dell'elaborazione completa. In particolare, la Figura 5.2 mostra le prestazioni della prima aggregazione temporale, la quale ha il compito di effettuare l'operazione di reverse geocoding al flusso dati in ingresso e calcolare la

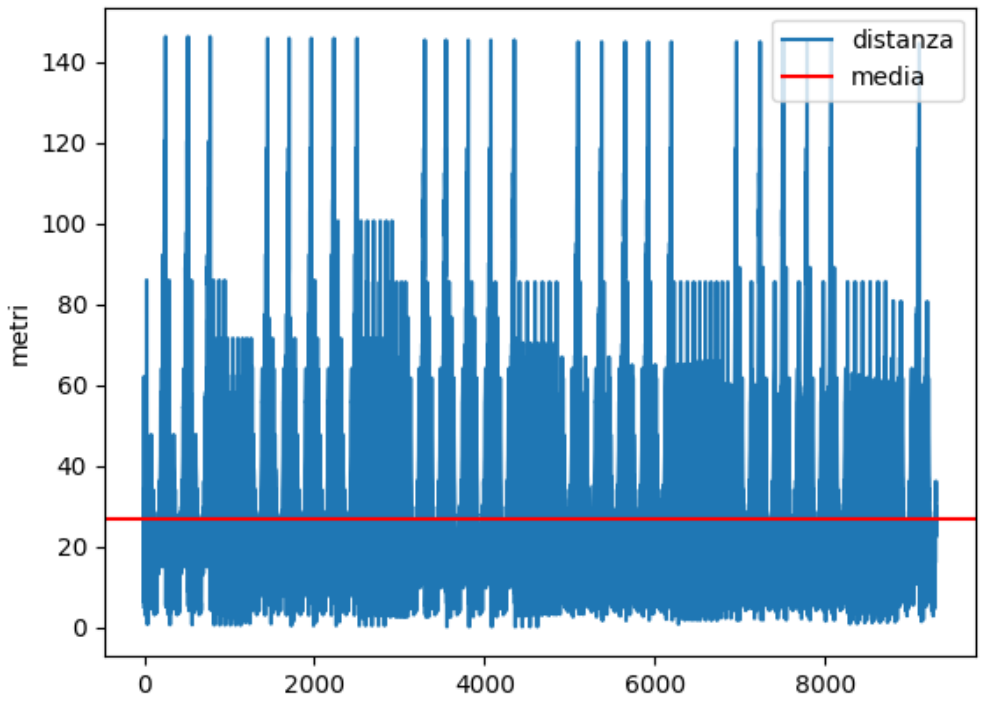

Figura 5.1: Grafico precisione del reverse geocoder

densità dei veicoli per ogni istante di tempo. Come ci si poteva aspettare, gran parte delle risorse utilizzate dall'applicativo sono richieste in questa fase di aggregazione; infatti, a partire dall'aggregazione successiva (mostrata in Figura 5.3), il tempo di elaborazione, di memoria e di CPU calano significativamente.
Le Figure 5.4 e 5.5 mostrano rispettivamente le prestazioni della fase di aggregazione temporale con granularità di quindici minuti e di un'ora, mentre la Figura 5.6 presenta le performance dell'intera elaborazione.
La tabella 5.2 riassume le prestazioni dell'applicativo. Analizzando i grafici e le informazioni fornite dall' interfaccia Web di Apache Spark, si è notato come il collo di bottiglia di tale applicativo è rappresentato dall'elevato numero di operazioni di serializzazione e deserializzazione effettuate durante le fasi di aggregazione gerarchica. D'altra parte, dopo la prima aggregazione di durata 8 minuti e 12 secondi, il numero di scritture e letture dal sistema diminuisce sempre più, grazie alla riduzione del numero dei record (la seconda aggregazione ha una durata di 40 secondi). Inoltre, a causa del watermark che ogni aggregazione presenta, la scrittura sul sistema (e quindi anche la lettura per il prossimo step di aggregazione, vedi sezione

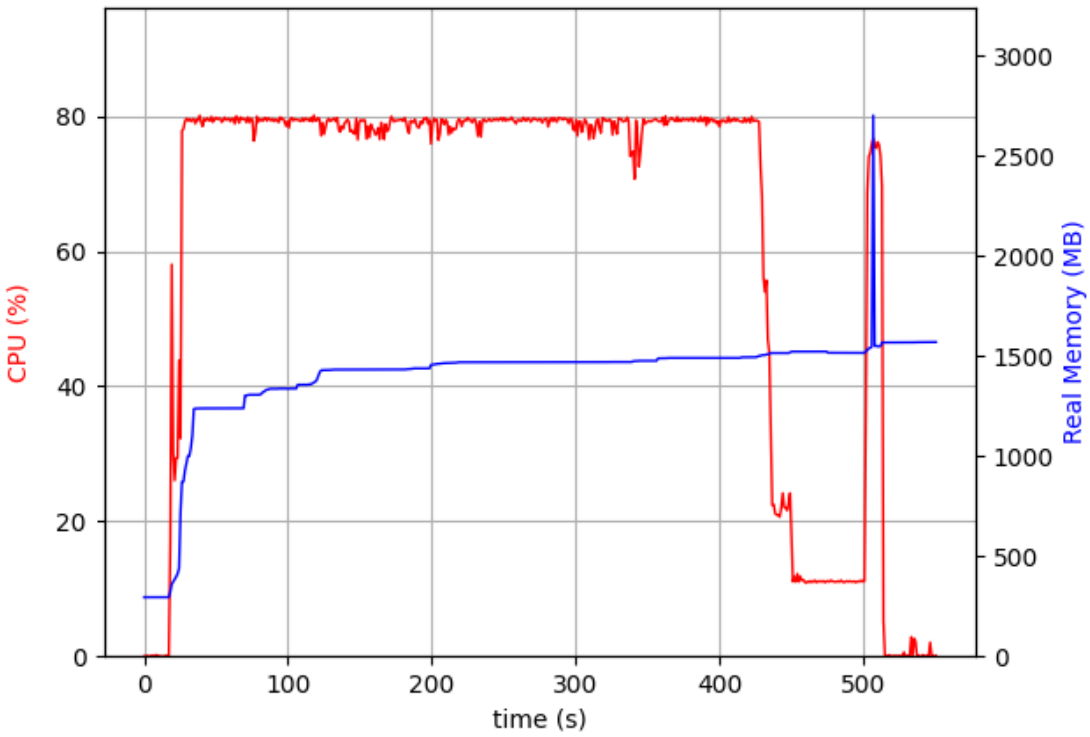

Figura 5.2: Grafico delle prestazioni della prima aggregazione temporale

4.2.6) risulta ritardato.
A conclusione di tale analisi, si può affermare che l'applicativo è in grado di rispondere correttamente al flusso dati in ingresso preso in considerazione in questo elaborato. Infatti, dopo le verifiche effettuate, si è dimostrato come sia in grado di elaborare un flusso contente i dati generati in un'ora in poco meno di quindici minuti, sfruttando circa il 70% della CPU e circa 2GB di memoria.

Tabella 5.2: Performance dell'applicativo

Operazione	**CPU (%)**	**Memory (MB)**	**Tempo**
Calcolo della densità e operazione di reverse geocoding	78.4	1430.20	8 minuti e 12 secondi
Calcolo aggregato con			

Tabella 5.2: Performance dell'applicativo

Operazione	**CPU (%)**	**Memory (MB)**	**Tempo**
granularità 1 minuto	47.2	985.51	40 secondi
Calcolo aggregato con granularità 15 minuti	41.6	780.69	32 secondi
Calcolo aggregato con granularità 1 ora	43.4	528.59	21 secondi
Elaborazione completa	73.6	1901.26	14 minuti e 29 secondi

5.3 Screenshot dei risultati ottenuti

Dopo aver caricato ed elaborato il flusso dati relativo al giorno 24/06/2020, sono stati effettuati una serie di screenshot per mostrare il funzionamento dell'applicativo su un caso d'uso reale.
Le Figure 5.7, 5.8 e 5.9 presentano i dati aggregati con granularità di un minuto mentre le Figure 5.10 ed 5.11 hanno granularità di quindici minuti. In ultimo, le Figure 5.12 ed 5.13 hanno granularità rispettivamente di un' ora e di un giorno.

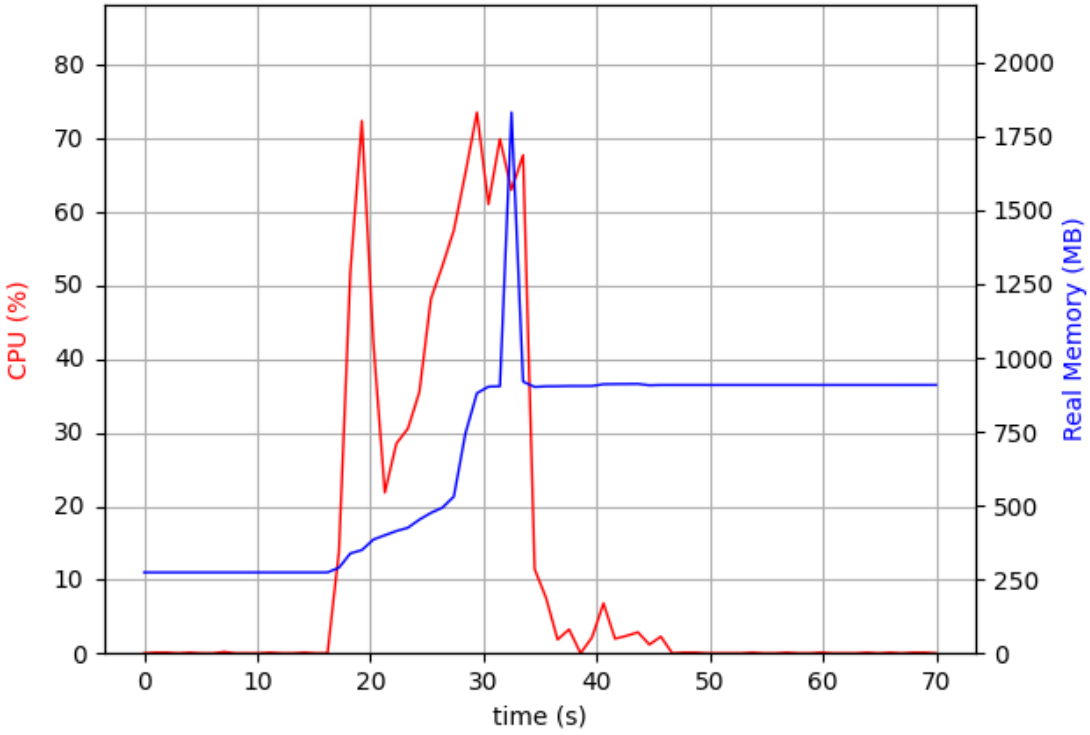

Figura 5.3: Grafico delle prestazioni per aggregazione temporale con granularità 1 minuto

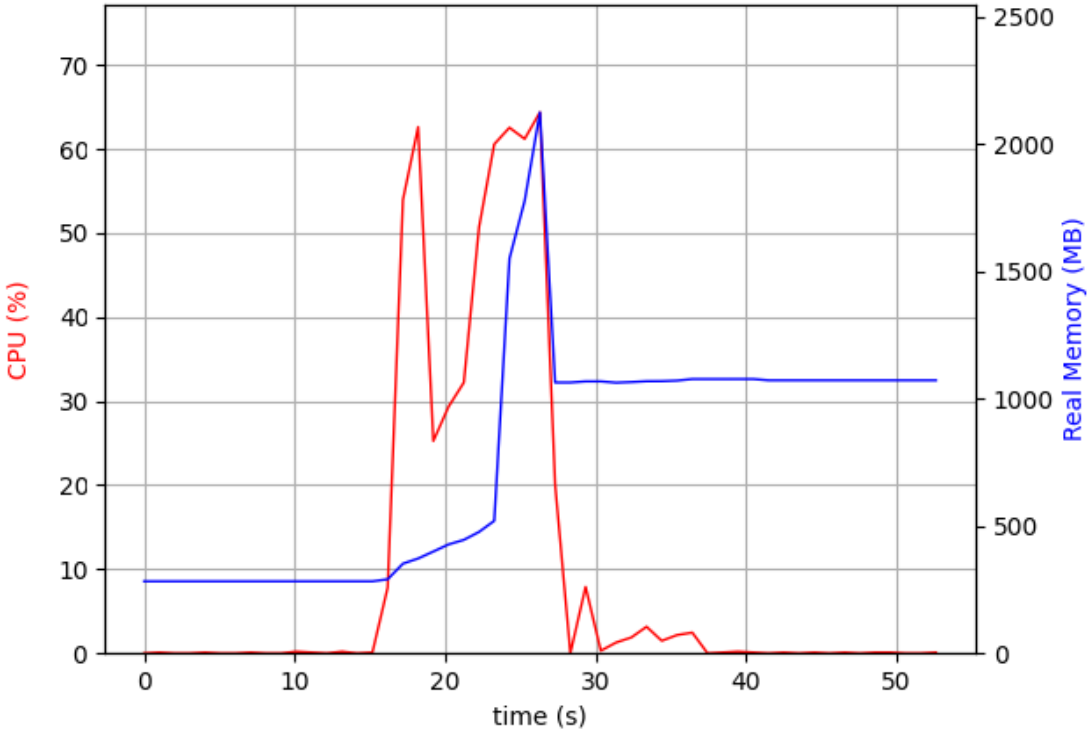

Figura 5.4: Grafico delle prestazioni per aggregazione temporale con granularità 15 minuti

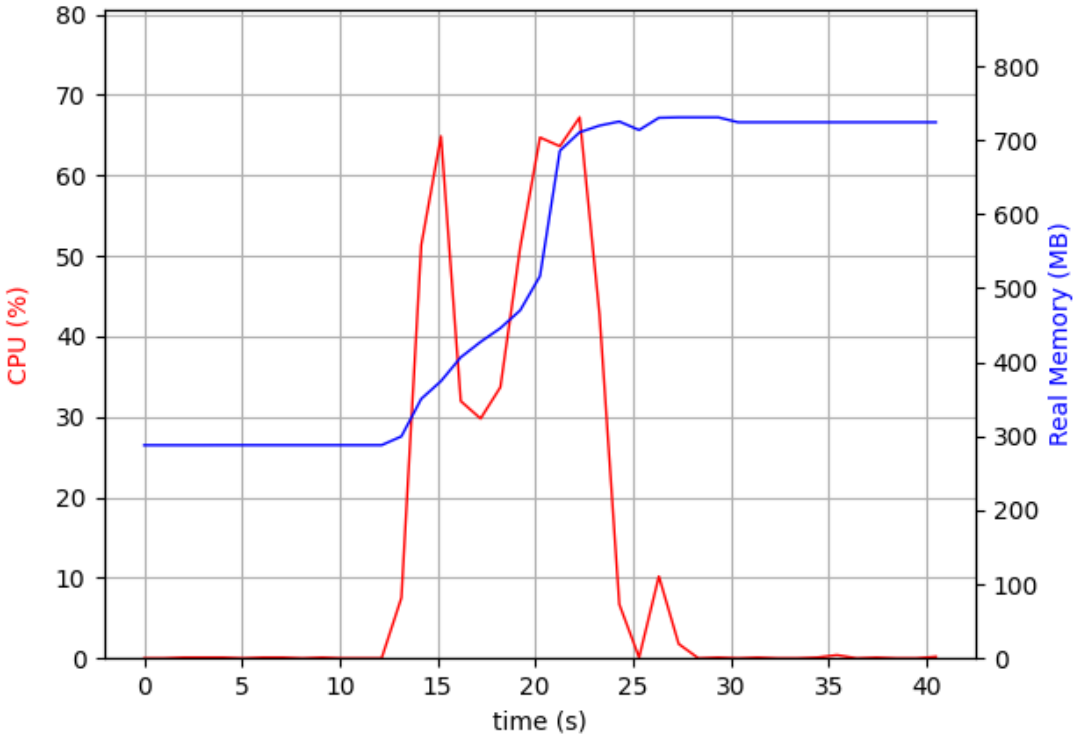

Figura 5.5: Grafico delle prestazioni per aggregazione temporale con granularità 1 ora

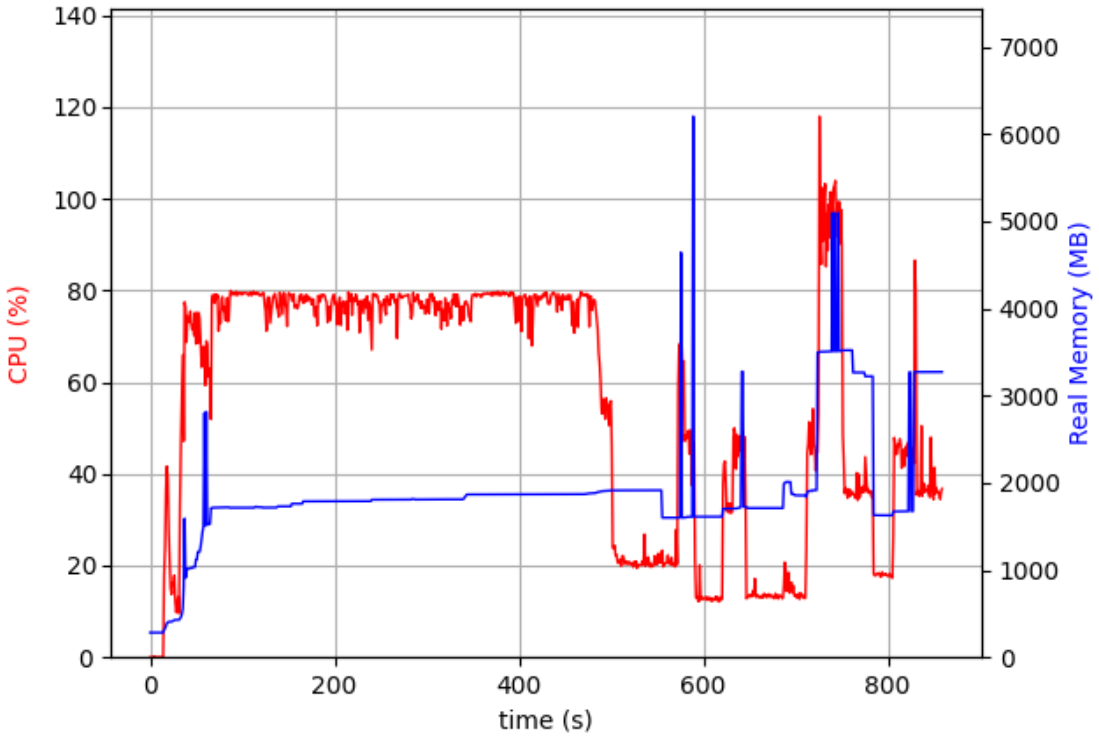

Figura 5.6: Grafico delle prestazioni dell'applicativo

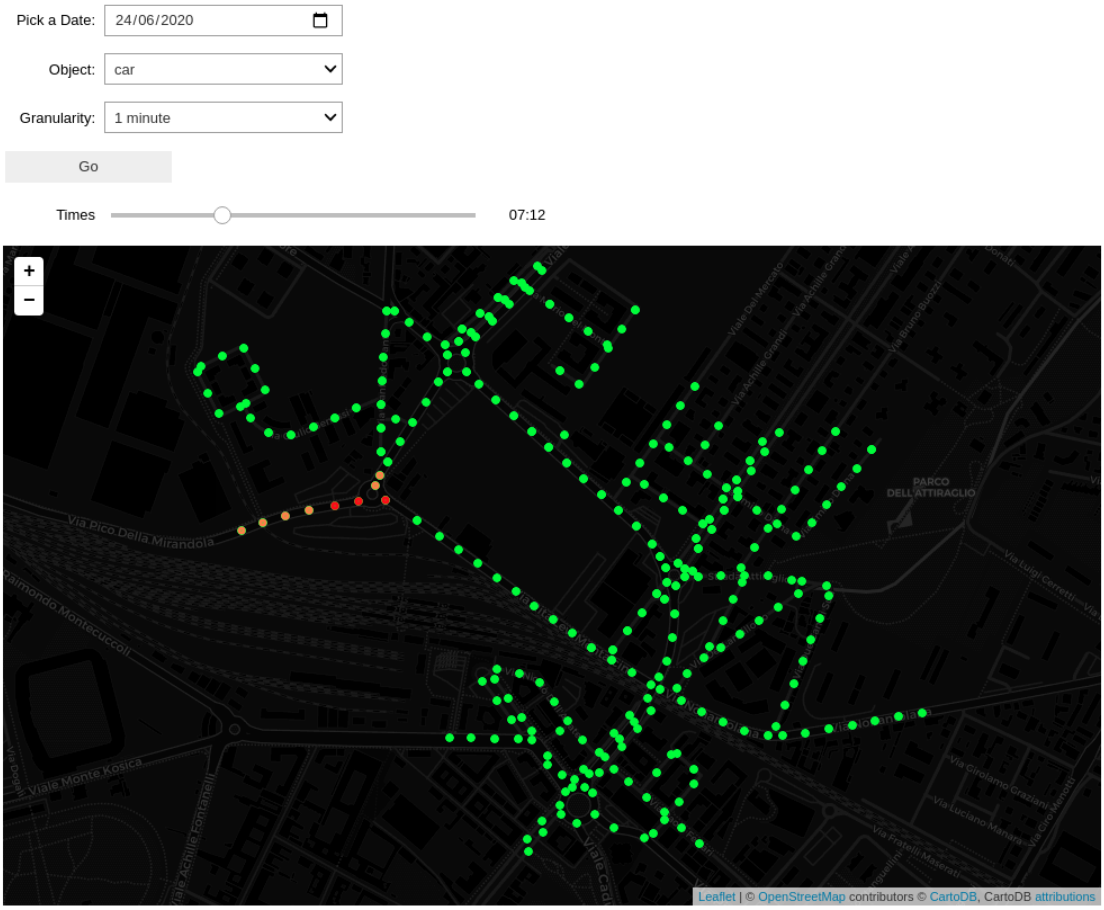

Figura 5.7: Traffico alle ore 07:12 con granularità 1 minuto

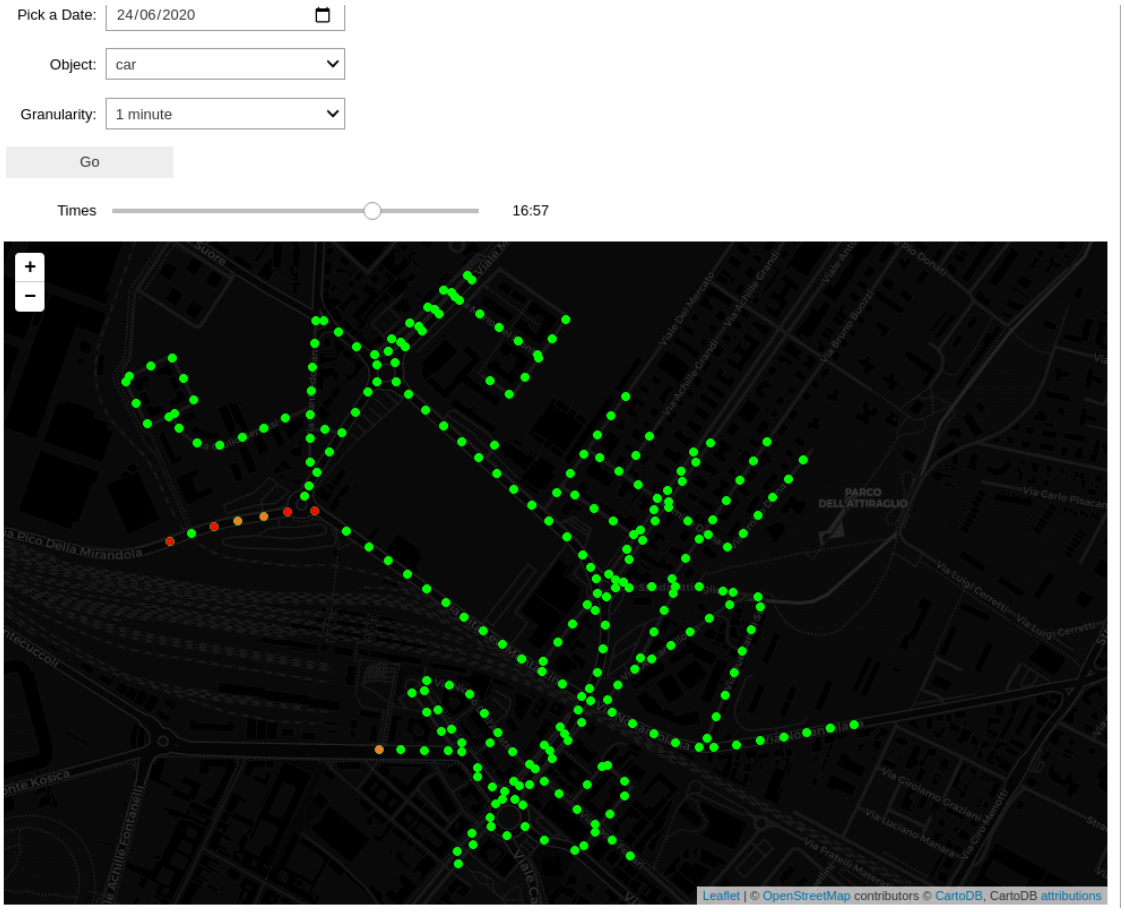

Figura 5.8: Traffico alle ore 16:57 con granularità 1 minuto

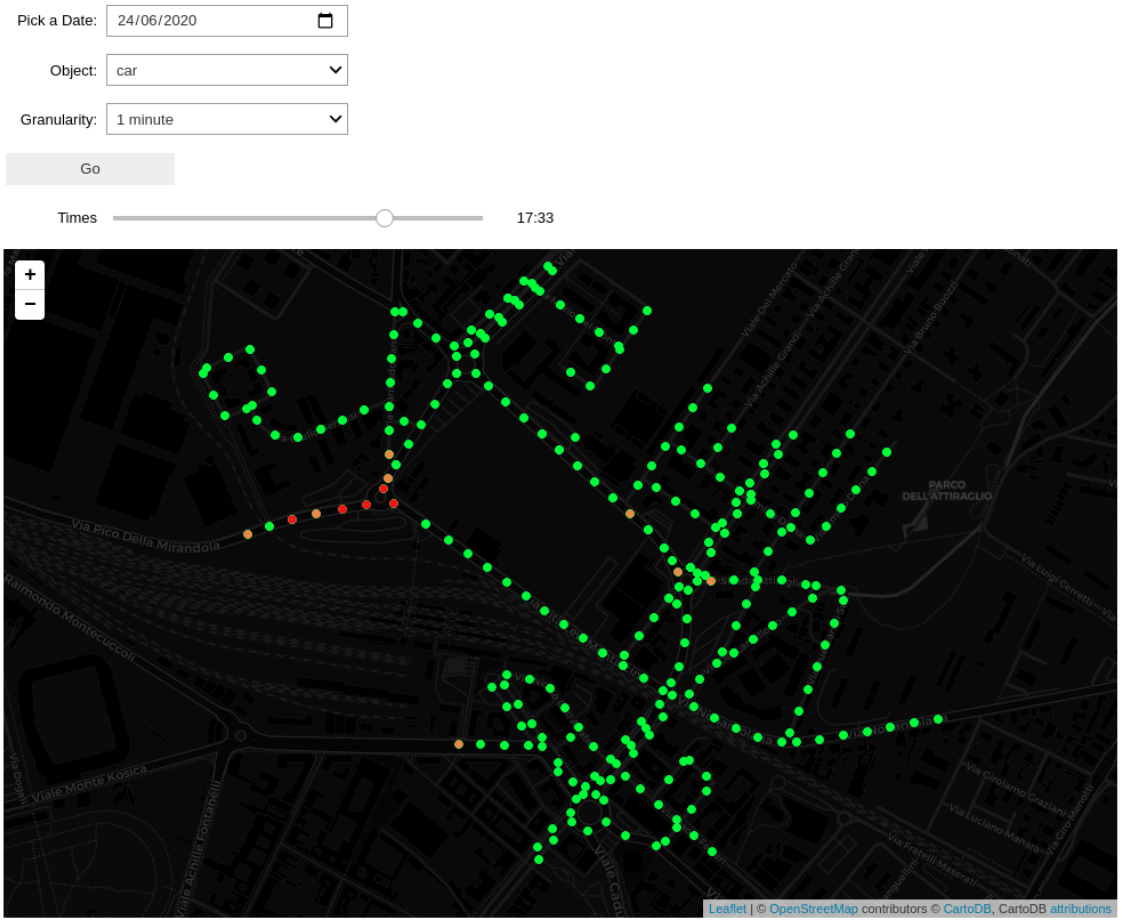

Figura 5.9: Traffico alle ore 17:33 con granularità 1 minuto

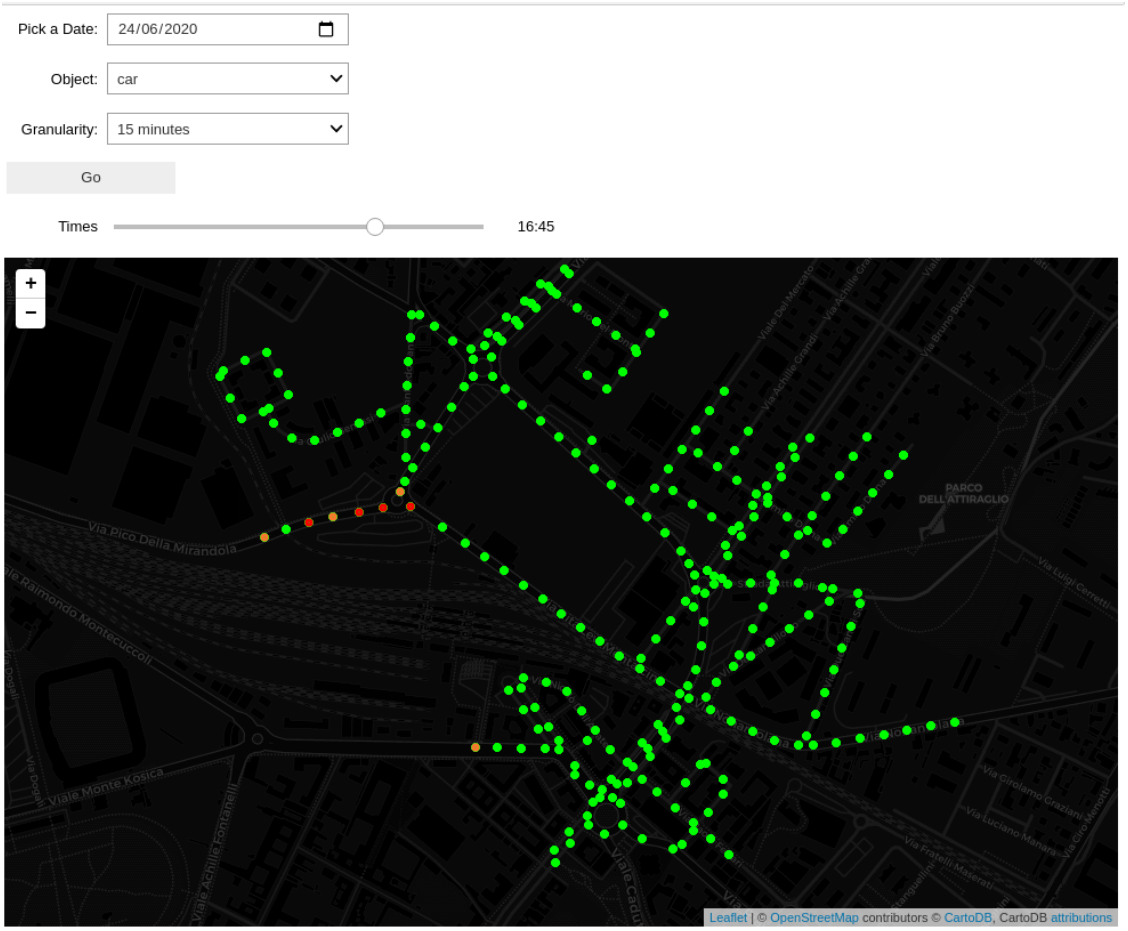

Figura 5.10: Traffico alle ore 16:45 con granularità 15 minuti

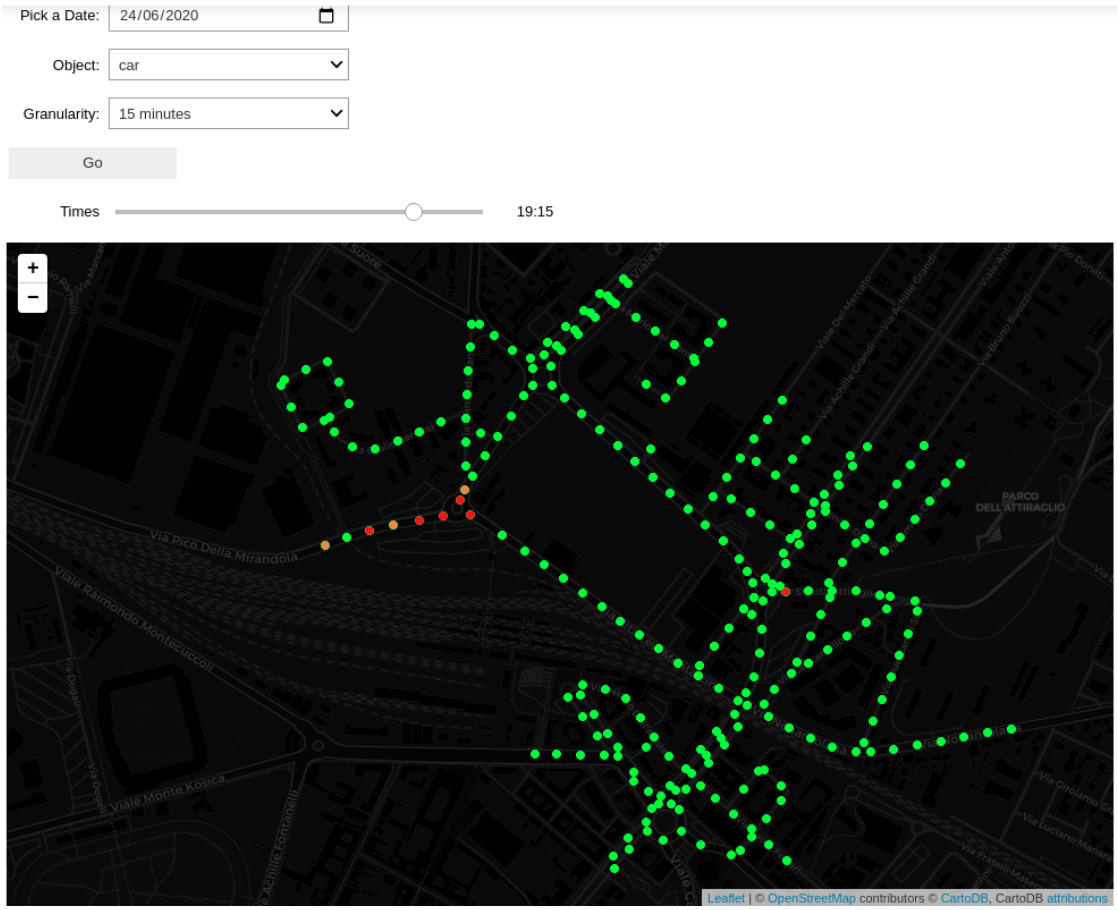

Figura 5.11: Traffico alle ore 19:15 con granularità 15 minuti

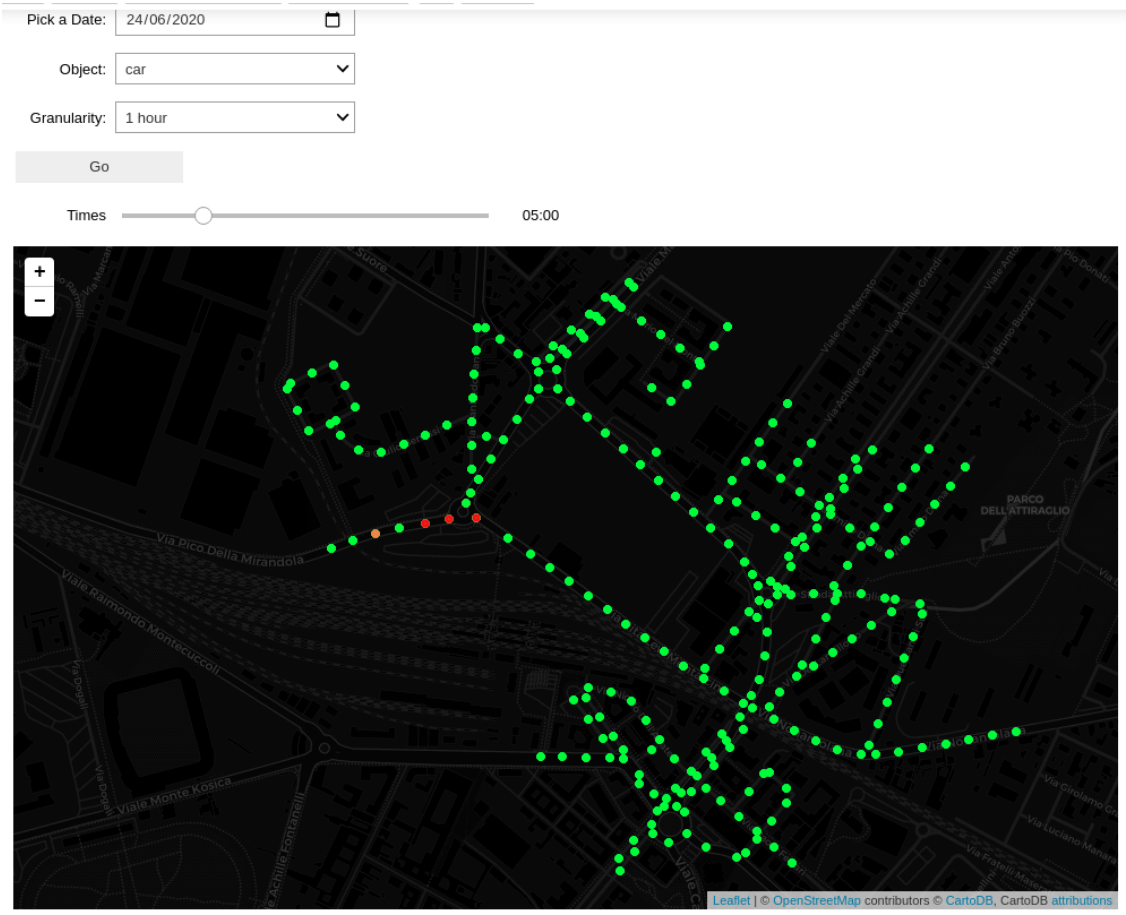

Figura 5.12: Traffico alle ore 05:00 con granularità 1 ora

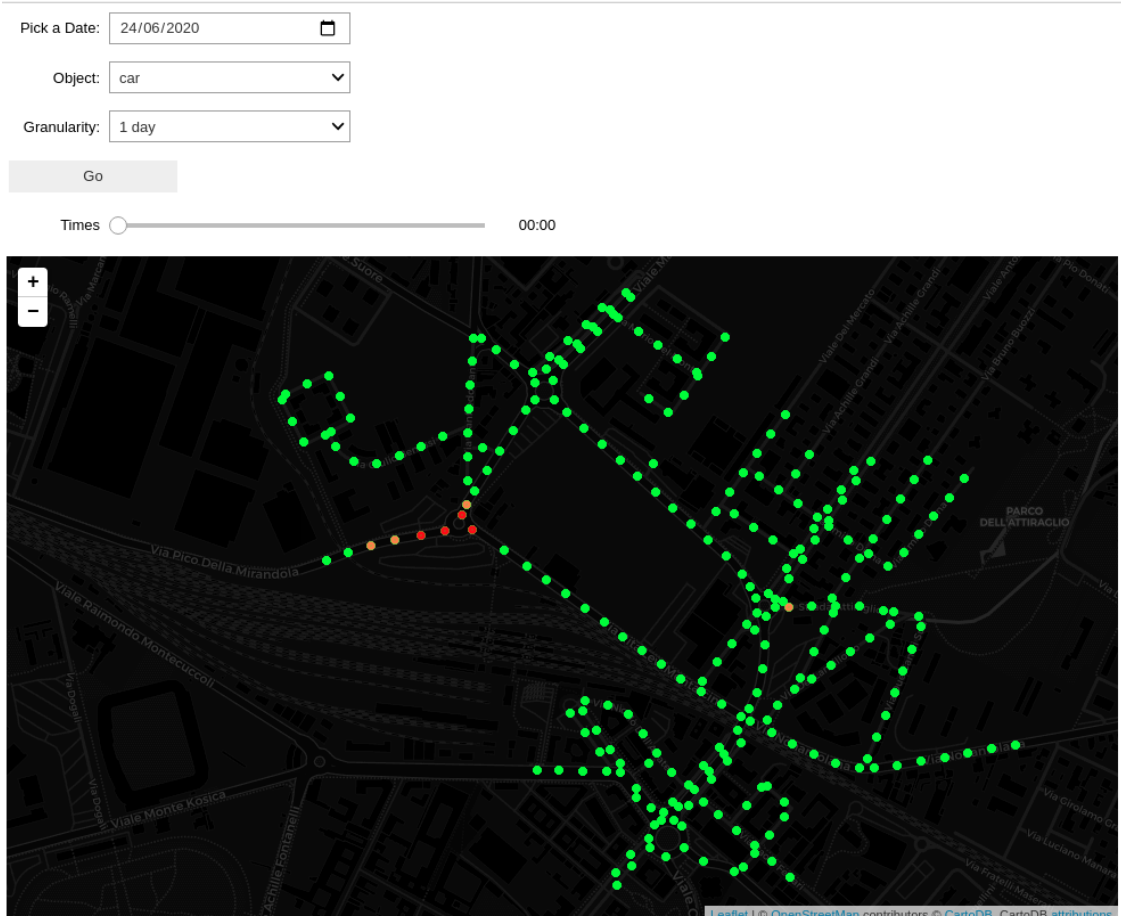

Figura 5.13: Traffico con granularità 1 giorno

Conclusioni

Si conclude il presente elaborato con alcune considerazioni derivate delle osservazioni maturate durante la fase di stesura di questa tesi.

In prima battuta, grazie allo studio di numerose fonti e articoli scientifici, si sono analizzati i moduli Spark SQL, Spark Streaming e Spark Structured Streaming propri del framework Apache Spark. In particolare, dopo un breve confronto tra Spark Streaming e Spark Structured Streaming (sezione 3.3), si è valutato quest'ultimo come più indicato all'ambito applicativo del modello da realizzare. Successivamente si è progettata un'aggregazione temporale gerarchica dei dati, la quale è stata influenzata fortemente dalla piattaforma di elaborazione streaming scelta.

In secondo luogo, grazie alla banca dati OpenStreetMap ed un software di elaborazione GIS chiamato QGIS, è stato possibile progettare ed implementare un reverse geocoder ad alte prestazioni. In questo modo, in accordo con il framework di elaborazione streaming, tale modulo ha permesso di trasformare le coordinate geografiche presenti in un flusso dati in coppie del tipo (strada, tratto di strada).

Infine, con l'obiettivo di mostrare su una mappa interattiva gli aggregati temporali elaborati nelle fasi precedenti, si è implementato un modulo per la visualizzazione dei dati. In questo modo, tramite una semplice interfaccia, l'utente è in grado di monitorare ed analizzare il traffico relativo alla Smart Model Area.

I dati raccolti ed elaborati rappresentano una vera e propria ricchezza, sulla quale è possibile intraprendere numerosi sviluppi futuri. Per esempio, a partire dai dati aggregati, si manifesta l'opportunità di implementare algoritmi di machine learning per la previsione del traffico.

Bibliografia

[1] IDC. «IDC's Global DataSphere Forecast Shows Continued Steady Growth in the Creation and Consumption of Data». In: (2020). URL: https://www.idc.com//getdoc.jsp?containerId=prUS46286020.

[2] URL: https://data-flair.training/blogs/dag-in-apache-spark.

[3] URL: http://spark.apache.org/docs/latest/streaming-programming-guide.html.

[4] Matei Zaharia, Mosharaf Chowdhury, Michael J. Franklin, Scott Shenker, Ion Stoica. «Spark: Cluster Computing with Working Sets». In: *University of California, Berkeley* (). URL: https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf.

[5] Michael Armbrust, Wenchen Fan, Reynold Xin, Matei Zaharia. *Introducing Apache Spark Datasets*. 2016. URL: https://databricks.com/blog/2016/01/04/introducing-apache-spark-datasets.html.

[6] URL: https://blog.twitter.com/engineering/en_us/a/2013/dremel-made-simple-with-parquet.html.

[7] URL: https://spark.apache.org/docs/latest/structured-streaming-programming-guide.html.

[8] Joseph Torres, Michael Armbrust, Tathagata Das, Shixiong Zhu. *Introducing Low-latency Continuous Processing Mode in Structured Streaming in Apache Spark 2.3*. 2018. URL: https://www.usenix.org/legacy/event/hotcloud10/tech/full_papers/Zaharia.pdf.

[9] URL: https://kafka.apache.org/documentation/#gettingStarted.

[10] URL: https://wiki.openstreetmap.org/wiki/Stats.

[11] URL: https://wiki.openstreetmap.org/wiki/Map_Features.

[12] URL: http://download.geofabrik.de/.

[13] URL: https://wiki.openstreetmap.org/wiki/Overpass_API/Overpass_QL.

[14] URL: https://wiki.openstreetmap.org/wiki/Overpass_turbo.

[15] URL: https://docs.qgis.org/3.10/it/docs/user_manual/preamble/features.html.

[16] URL: https://docs.qgis.org/3.10/it/docs/user_manual/plugins/core_plugins/plugins_geometry_checker.html#geometry-checker.

[17] URL: https://grass.osgeo.org/.

[18] URL: https://openlayers.org/.

[19] Sunil Arya, David M. Mount, Nathan S. Netanyahu, Ruth Silverman, Angela Y. Wu. «An Optimal Algorithm for Approximate Nearest Neighbor Searching in Fixed Dimensions». In: (). URL: https://graphics.stanford.edu/courses/cs468-06-fall/Papers/03%20AMNSW%20-%20JACM.pdf.

[20] URL: https://www.automotivesmartarea.it/.

[21] URL: https://storm.apache.org/about/multi-language.html.

[22] URL: https://databricks.com/spark/about.

[23] URL: https://www.scipy.org/.

[24] URL: https://github.com/jupyter/jupyter/wiki/Jupyter-kernels.

[25] URL: https://servizissiir.regione.emilia-romagna.it/FlussiMTS/.

[26] URL: https://wiki.openstreetmap.org/wiki/Key:highway.

[27] URL: https://wiki.openstreetmap.org/wiki/Key:bridge.

[28] URL: https://wiki.openstreetmap.org/wiki/Key:lit.

[29] URL: https://wiki.openstreetmap.org/wiki/Nominatim.