Università degli Studi di Milano Bicocca

Dipartimento di Informatica

Progetto di Large Scale Data Management

Analysis On NYC Yellow Taxi Dataset

Nada Mohamed 857606
Alessia Rubini 851890

Indice

Elenco delle figure

Capitolo 1

Presentazione dei dati

Il dataset "NYC Taxi & Limousine Commission - yellow taxi trip records" costituisce una fonte di informazioni dettagliate riguardo i viaggi effettuati nei caratteristici taxi gialli tra le vie della Grande Mela.
Le informazioni riportate dal set di dati sono di ampia portata: si passa banalmente dal codice identificativo della compagnia di taxi all'ammontare della mancia rilasciata per ogni viaggio.
I dati sono reperibili sul sito ufficiale TLC Trip Record Data in formato Parquet, divisi mese per mese e coprono il periodo complessivo dal 2011 ad oggi.
Le informazioni di viaggio che costituiscono il dataset sono state raccolte e rese disponibili alla *NYC Taxi and Limousine Commission (TLC)* da fornitori di tecnologia autorizzati in ambito dei *Taxicab & Livery Passenger Enhancement Programs (TPEP/LPEP)*.
Si riporta una tabella riassuntiva dei campi riportati per ciascun record.

Nome colonna	Descrizione
VendorID	Codice che indica il fornitore TPEP
tpep pickup datetime	Data e ora in cui il tassametro è stato avviato
tpep dropoff datetime	Data e ora in cui il tassametro è stato disattivato
Passenger count	Numero di passeggeri nel veicolo
Trip distance	Distanza percorsa in miglia segnata dal tassametro
PULocationID	Zona taxi TLC in cui è stato avviato il tassametro
DOLocationID	Zona taxi TLC in cui è stato disattivato il tassametro
RateCodeID	Codice della tariffa finale in vigore alla fine del viaggio. 1 = Tariffa standard 2 = JFK 3 = Newark 4 = Nassau o Westchester 5 = Tariffa negoziata 6 = Viaggio di gruppo
Store and fwd flag	Questo flag indica se il record di viaggio è stato conservato nella memoria del veicolo prima dell'invio al fornitore, detto anche "store and forward", perché il veicolo non era collegato al server.
Payment type	Codice numerico che indica come il passeggero ha pagato il viaggio 1 = Carta di credito 2 = Contanti 3 = Nessun addebito 4 = Contestazione 5 = Sconosciuto 6 = Viaggio annullato
Fare amount	Tariffa per tempo e distanza calcolata dal tassametro
Extra	Extra e supplementi vari. 0,50$ = Ore di punta 1$ = Tariffe notturne

MTA tax	$0,50 Tassa MTA che viene attivata automaticamente in base alla tariffa di misurazione in uso
Improvement surcharge	Supplemento di miglioramento che viene addebitato sui viaggi in taxi quando la bandiera è alzata. La bandiera è solitamente alzata quando il taxi è occupato da un passeggero e il veicolo è in movimento
Tip amount	Questo campo viene popolato automaticamente per le mance con carta di credito
Tolls amount	Importo totale di tutti i pedaggi pagati durante il viaggio
Total amount	Importo totale addebitato ai passeggeri. Non include le mance in contanti.
Congestion Surcharge	Importo totale raccolto durante il viaggio per la sovrattassa di congestione del NYS. applicata sempre dopo il 2015
Airport fee	$1,25 per il prelievo solo negli aeroporti LaGuardia e John F. Kennedy.

Figura 1.1: Descrizione colonne dataset

Capitolo 2

Elaborazione dei dati: PySpark

Tra i principali obiettivi dell'elaborato vi è quello di esaminare e manipolare i dati presenti nel dataset originale andando a "costruire" nuovi DataFrame, ognuno dei quali mira ad analizzare aspetti specifici.
Per portare a termine tale obiettivo ci si è serviti di **PySpark**, un framework open-source per il calcolo distribuito progettato per l'elaborazione di grandi quantità di dati con Python. È costruito su Apache Spark, ossia un framework di elaborazione distribuita in memoria che fornisce un'interfaccia per la programmazione in batch e in streaming.
PySpark, quindi, consente di scrivere applicazioni di analisi dati utilizzando Python, sfruttando la potenza di Apache Spark; il ché significa che è possibile sfruttare tutte le funzionalità di Spark - tra cui l'elaborazione parallela, la gestione della memoria distribuita e la tolleranza ai guasti - utilizzando sintassi e librerie di Python.
In particolare, PySpark dispone di una libreria significativamente utile al caso studio, cioè **pyspark.sql**, che fornisce un'API per l'elaborazione e l'analisi dei dati strutturati. Utilizzando questa libreria, è possibile eseguire query *SQL-like* su DataFrame, manipolando i risultati per ottenere informazioni specifiche o costruire altri nuovi DataFrame, come in questo caso.
In totale, sono stati ottenuti tre nuovi DataFrame, rispettivamente focalizzati in:

- Distretti di partenza e di arrivo dei viaggi
- Tipologie di pagamento
- Numero di passeggeri

2.0.1 Dataframe sui Distretti

Il Dataset originale permette di conoscere la locazione geografica in cui è stato avviato *(PULocationID)* e arrestato *(DOLocationID)* il tassametro; tali dati rappresentano verosimilmente il punto di partenza e di arrivo del viaggio in Taxi.
Pertanto, si è ritenuto potesse essere interessante allocare questi ID di locazione all'interno dei vari Distretti di New York City. Il ché significa che se il viaggio è iniziato e terminato nello stesso distretto allora vi è coincidenza tra i due punti, quindi, durata e costo dovrebbero essere ridotti; altrimenti non vi è coincidenza.

Il sito Web Governativo che pone a disposizione il dataset sui viaggi, offre anche la possibilità di scaricare il file relativo ai Distretti. Questo associa ogni ID di locazione al rispettivo Distretto oltre ad entrare nello specifico di quale zona.
Dunque, andando ad effettuare una *JOIN* tra il Dataframe originale e quello dei distretti, filtrando poi mediante una query i dati di viaggio, si è ottenuto il nuovo DataFrame, composto dalle seguenti colonne:

- **PUBorough**: Distretto di partenza
- **DOBorough**: Distretto di arrivo
- **trips_number**: Numero di viaggi effettuati con i rispettivi punti di partenza e arrivo
- **avg_total_amount**: Importo totale medio per questo tragitto addebitato ai passeggeri
- **avg_tip_amount**: Ammontare medio della mancia per questo tragitto
- **avg_fare_amount**: Ammontare tariffa per tempo e distanza calcolata dal tassimetro
- **coincidence**: Si tratta di una colonna binaria; avrà valore pari a 1 se il Distretto di partenza coincide con quello di arrivo, 0 altrimenti

2.0.2 DataFrame sul numero passeggeri

Il numero di passeggeri per un singolo viaggio è ovviamente limitato. Dall'analisi del dataset originale si è notato che tale numero può variare da 1 a 9.
Essendo quindi un valore che assume una quantità di valori limitata, si è deciso di andare a filtrare i dati in modo da raggrupparli in base al numero di passeggeri del viaggio in questione.
In questo caso non è stato necessario ricorrere a file esterni, ma ci si è limitati semplicemente ad effettuare una QUERY sui dati originali, ottenendo un nuovo DataFrame composto dalle seguenti colonne:

- **passenger_count**: Numero di passeggeri all'interno del veicolo durante un determinato viaggio
- **avg_trip_distance**: Distanza media percorsa negli spostamenti con quel certo numero di passeggeri a bordo
- **avg_fare_amount**: Tariffa media per tempo e distanza calcolata dal tassametro nei viaggi con quel certo numero di passeggeri a bordo
- **avg_tip_amount**: Ammontare medio della mancia per gli spostamenti con quel certo numero di passeggeri a bordo
- **avg_total_amount**: Importo medio addebitato ai passeggeri per uno spostamento con a bord del veicolo un certo numero i passeggeri
- **num_trips**: Numero di viaggi totale effettuati con a bordo quel certo numero di passeggeri

2.0.3 DataFrame sul tipo pagamento

Le modalità di pagamento disponibili per saldare l'itinerario effettuato in taxi risultano essere 6. Quindi, il terzo ed ultimo DataFrame costruito è stato ottenuto mediante una query sui dati originali, raggruppando il tutto per la tipologia di pagamento registrata.
Tale DataFrame si rivela essere composto dalle colonne:

- **payment_type**: Codice numerico che indica la modalità di pagamento per il viaggio in taxi
- **avg_trip_distance**: Distanza media percorsa quando viene utilizzata una certa modalità di pagamento

- **avg_fare_amount**: Tariffa media per tempo e distanza calcolata dal tassametro quando viene utilizzata quella certa modalità di pagamento
- **avg_tip_amount**: Ammontare medio della mancia quando viene utilizzata quella certa modalità di pagamento
- **avg_total_amount**: Importo medio totale per il viaggio quando viene utilizzata quella certa modalità di pagamento
- **num_trips**: Numero di viaggi totale in cui è stata utilizzata quella specifica modalità di pagamento

Capitolo 3

Configurazione iniziale

Il fine ultimo del progetto è quello di comprendere come gestire un'ampia gamma di dati, manipolarla e analizzarla mediante query, calcolando velocità e tempi di calcolo.
Un primo esperimento di gestione è stato effettuato provando a lavorare su una delle macchine personali del team.
Per questa fase iniziale Spark è stato eseguito in modalità "**local mode**"; il ché significa che è stato eseguito sulla singola macchina senza l'ausilio di un cluster. In questo modo, tutti i processi Spark vengono eseguiti sulla medesima macchina.
Questa modalità *local* si è rivelata utile durante lo sviluppo e il debug delle applicazioni Spark, in quanto ha consentito di eseguire e testare il codice su una quantità di dati nettamente inferiore senza dover configurare un ambiente distribuito. Tuttavia, si limita alle risorse computazionali disponibili sulla macchina locale senza sfruttare le caratteristiche di scalabilità e parallelismo.
L'idea è stata quella di "saturare" le risorse computazionali disponibili aumentando gradualmente la quantità di dati.
Tale processo ha permesso, innanzi tutto, di avere un primo approccio alla gestione dei dati mediante PySpark, e, in secondo luogo, di stimare quanti dati la macchina fosse in grado di manipolare.
La macchina utilizzata in questa fase presenta le seguenti caratteristiche:

- Processore: *Intel(R) Core(TM) i7-8565U CPU @ 1.80GHz 1.99 GHz*
- RAM: *8,00 GB (7,85 GB utilizzabile)*
- Tipo di sistema: sistema operativo a 64bit

Si è giunti alla saturazione delle risorse utilizzando un file .parquet di dimensione 532 MB contenente 24.747.305 records.
La macchina non è riuscita a leggere, elaborare query e salvare allocando i nuovi dataframe in formato .parquet, per file di dimensioni superiori a queste.

Nella prossima sezione si entra nella fase viva del progetto, in cui ci si sposta su macchine virtuali con risorse computazionali maggiori rispetto a quelle personali del team.

Capitolo 4

Large scale data management

In questo capitolo si passa dalla fase di configurazione iniziale su macchine locali personali, alla configurazione di Spark su macchine virtuali con risorse computazionali maggiori. Questo dovrebbe garantire una capacità di elaborazione - in termini di quantità di dati - migliore.
Di fatti, è proprio questo l'obiettivo finale del progetto: imparare ad analizzare e manipolare una mole di dati importante lavorando parallelamente con più macchine. Tale principio incontra l'idea alla base del cloud computing: diverse macchine connesse tra loro lavorano in parallelo accedendo alla stessa sorgente dati per eseguire dei task.
L'Ateneo ha predisposto un totale di quattro macchine virtuali Azure per i fini dell'elaborato; in particolare, tre di queste dispongono delle stesse risorse computazionali e fungono quindi da nodi worker; mentre, uno di questi ha caratteristiche differenti e viene utilizzato come nodo Master.

Al fine di rispettare il concetto di separazione tra lo storage dei dati e le componenti esecutive, ci si è serviti di uno Storage Account Azure, sul quale sono stati caricati tutti i file .parquet necessari.
L'idea è quella di sfruttare una configurazione Spark cluster in modalità Standalone, utilizzando un nodo Master per "commissionare" e distribuire i task da far eseguire ai nodi Workers, che si limiteranno alla fase esecutiva.
Questa disposizione architetturale permette a tutte le macchine di accedere parallelamente a tutti i dati presenti nello Storage Azure.

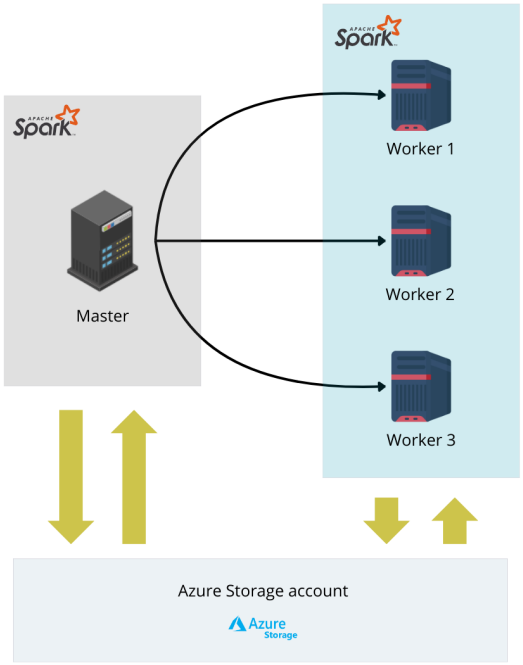

Figura 4.1: Architettura concettuale.

4.1 Spark standalone

Apache Spark è un framework open source per il calcolo distribuito. Presenta componenti come Spark Core, Spark Streaming, Spark SQL e Spark MLib dimostrandosi un tool ottimale per il mondo dei big data.
Il core di Spark contiene le funzionalità base di Spark, tra cui la gestione della memoria, della rete, dello scheduling, il recupero dal fallimento di un nodo del cluster(vengono ricalcolati solo i dati persi), ecc..
Il core di Spark fornisce un'API che permette di gestire i cosiddetti Resilient Distributed Datasets(RDDs), che rappresentano dati distribuiti sul cluster e su cui vengono svolte operazioni di trasformazione per poi recuperare i dati modificati.
Spark SQL permette di fare interrogazioni su dati strutturati e semi strutturati usando il linguaggio HiveQL, una variante del linguaggio SQL. Si possono interrogare file JSON, file di testo e qualunque formato di file supportato da Hive, tra cui Parquet, come nel caso in analisi.
SparkSQL si integra perfettamente con il resto del sistema Spark, infatti i risultati delle query possono essere trasformati e analizzati usando l'API di Spark che funziona sugli RDD, così come è possibile analizzare dati con HiveQL dopo avergli associato uno schema tabellare.

Spark può lavorare sia in un singolo nodo che in cluster. In altre parole, Spark supporta la modalità **cluster standalone**. Un "cluster standalone" si può vedere come un insieme autonomo di sistemi informatici, o nodi, interconnessi all'interno di una rete, in cui ciascun nodo funziona indipendentemente dagli altri per l'esecuzione di operazioni o compiti specifici. I nodi all'interno del cluster standalone sono solitamente gestiti in modo centralizzato da un sistema di gestione del cluster e possono operare insieme per fornire capacità computazionale condivisa, ridondanza o altri servizi.
Nel caso generale vi sono una serie di processi in esecuzione per ogni applicazione Spark: un

driver e molteplici executor.

Il driver è il processo principale, quello in cui è presente il metodo main contenente il codice utente. Il codice, che contiene operazioni di trasformazioni e azioni sugli DataFrames, dovrà essere eseguito in parallelo dai processi executor distribuiti nel cluster.
In questo progetto la macchina virtuale utilizzata come master funge anche da driver, andando a coordinare e gestire l'esecuzione dell'applicazione Spark.
Il driver si occupa principalmente di convertire il programma utente in un insieme di task, e di effetturarne lo scheduling sui nodi executor.
Tali nodi dispongono di una certa quantità di memoria, che gli permette - se richiesto dall'applicazione utente (tramite l'istruzione *cache* su un DataFrame) - di memorizzare i dataset in memoria.

Nel processo principale di un'applicazione Spark (processo driver) è presente un oggetto di tipo SparkContext, la cui istanza comunica con il gestore di risorse del cluster per richiedere un insieme di risorse (RAM, core, ecc.) per gli executors (4.2). Sono supportati diversi cluster manager tra cui YARN, Mesos, EC2 e lo Standalone cluster manager nativo di Spark, nonchè quello utilizzato in questo progetto.

In particolare, viene usata un'architettura di tipo master/worker; quindi, si possono

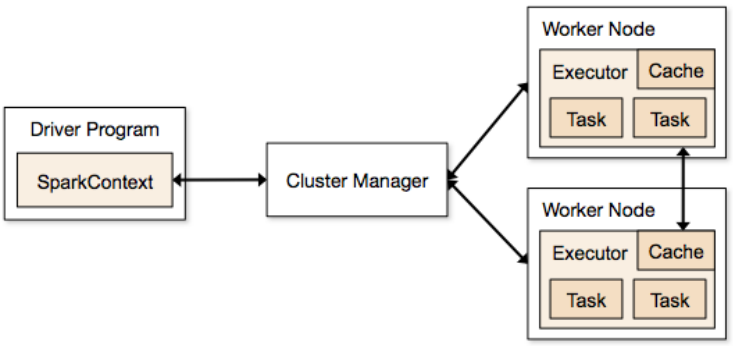

Figura 4.2: Cluster mode.

identificare:

- **Standalone Master**: è il nodo coordinatore delle risorse per il cluster Spark Standalone.
- **Standalone Worker**: è il nodo *lavoratore* di un cluster Spark Standalone. Vi possono essere uno o più workers.

Un'applicazione Spark è costituita da un insieme di job, ciascuno dei quali rappresenta una richiesta di esecuzione di una determinata operazione su un insieme di dati. Ogni volta che si vuole ottenere un risultato dalla computazione, si genera un job.
Ciascun job è composto da uno o più stages, che sono unità di lavoro atomiche che vengono eseguite sequenzialmente. Gli stages dipendono l'uno dall'altro e vengono eseguiti in ordine. Ognuno di questi può includere più task, che rappresentano le singole unità di lavoro eseguite parallelamente dagli executor del cluster.
In breve, un job rappresenta un'operazione da eseguire, gli stages rappresentano i passaggi necessari per completare l'operazione, e i task rappresentano i singoli compiti eseguiti parallelamente per completare ogni stage.

Apache Spark fornisce una suite di interfacce Web UI/User (Jobs, Stages, Tasks, Storage, Enviroment, Executors ed SQL) per monitorare lo stato dell'applicazione Spark/PySpark, il consumo di risorse del cluster Spark e le configurazioni Spark.

Prima di approfondire tali interfacce, è importante distinguere due concetti: azione e trasformazione.
Una trasformazione è una qualsiasi operazione Spark che restituisce un DataFrame, un set di dati o un RDD. Quando si crea una catena di trasformazioni, in realtà non si fa altro che aggiungere elementi costitutivi allo Spark job, ma non viene elaborato alcun dato. Questo è possibile perchè le trasformazioni vengono eseguite in modalità *lazy*. Spark andrà a calcolare il valore solo se necessario.
Le azioni, d'altra parte, non vengono eseguite in modalità lazy. Nel momento in cui si va ad inserire un'azione nel codice, e Spark la raggiunge durante l'esecuzione del Job, sarà necessario eseguire nuovamente tutte le trasformazioni fino a quell'azione, per poi produrre un valore. Di fatti, il concetto chiave di un'azione è la produzione di un valore: mentre le trasformazioni restituiscono uno dei tipi di dati Spark, le azioni possono restituire un conteggio di elementi, un elenco degli stessi o un'archiviazione dei dati in un archivio esterno.

In particolare, la modalità standalone di Spark offre un'interfaccia generale ed altre utili a monitorare quanto accade in modo più dettagliato. Di fatti, sia il master che ciascun worker dispongono di una propria interfaccia web U/I dedicata.
Per impostazione predefinita, l'interfaccia con le informazioni relative al Master è accessibile alla porta *localhost:8080*, mentre quella per le informazioni relative al Worker è accessibile alla porta *localhost:8081* (4.3). L'interfaccia Web generale, invece, da cui è possibile ottenere metriche e statistiche relative a tutto il processo in corso, si trova alla porta *localhost:4040*.

Figura 4.3: Worker UI.

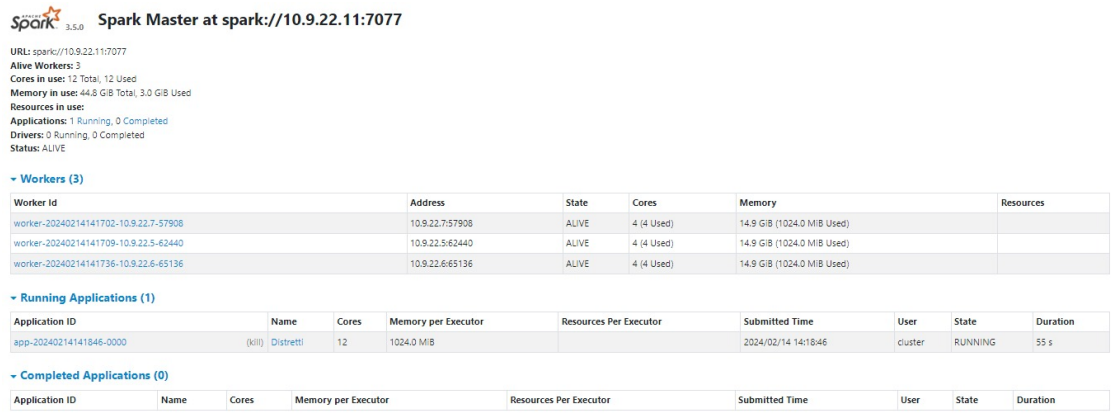

Figura 4.4: Master port 7070 UI.

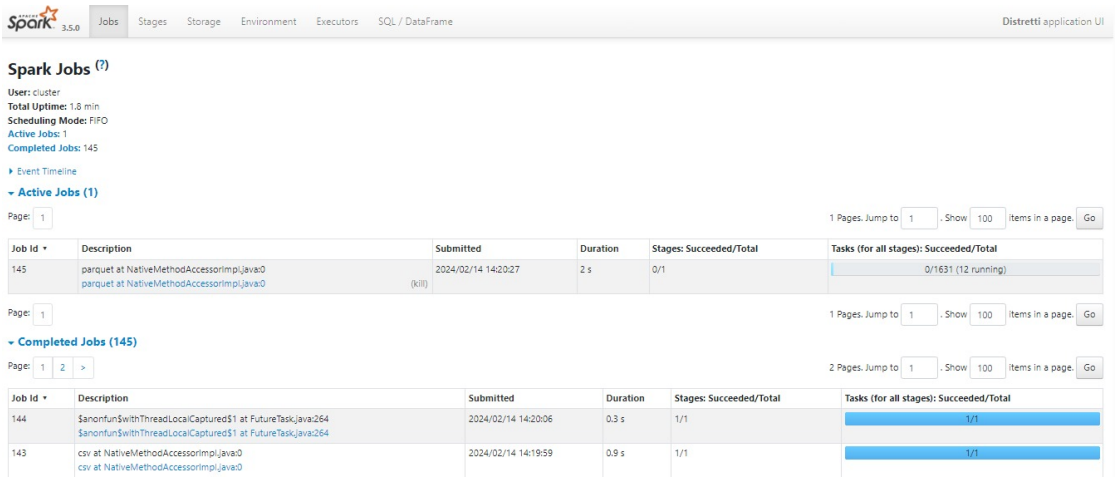

Figura 4.5: Master port 4040 UI.

4.1.1 Apache Spark nell'ecosistema Hadoop

Il software Apache Hadoop è un framework open source che permette l'archiviazione e l'elaborazione distribuita di grandi set di dati in cluster di computer utilizzando semplici modelli di programmazione. Hadoop è progettato per fare lo scale up da un singolo computer a migliaia di computer in cluster, dove ogni macchina fornisce il calcolo e l'archiviazione locali. Ciò consente ad Hadoop di archiviare ed elaborare in modo efficiente set di dati di grandi dimensioni che vanno da gigabyte a petabyte.

Il framework principale di Hadoop è costituito da quattro moduli che operano collettivamente per formare l'ecosistema Hadoop:

- **Hadoop Distributed File System (HDFS)**: componente principale dell'ecosistema Hadoop, HDFS è un file system distribuito che fornisce un accesso ad alta velocità effettiva ai dati dell'applicazione senza la necessità di definire schemi in anticipo.
- **Yet Another Resource Negotiator (YARN)**: una piattaforma di gestione delle risorse che si occupa di gestire le risorse di calcolo in cluster e di utilizzarle per programmare le applicazioni degli utenti. Esegue la programmazione e l'allocazione delle risorse in tutto il sistema Hadoop.

- **MapReduce**: è un modello di programmazione per l'elaborazione dei dati su vasta scala. Tramite algoritmi di calcolo distribuiti e paralleli, MapReduce rende possibile il trasferimento della logica di elaborazione e aiuta a scrivere applicazioni che trasformano grandi set di dati in un unico set gestibile.
- **Hadoop Common**: comprende le librerie utilizzate e condivise da altri moduli Hadoop.

Spark è integrato in un cluster Hadoop: può leggere i dati archiviati in HDFS, utilizzare YARN per la gestione delle risorse e sfruttare il ricco ecosistema Hadoop, fornendo al contempo un modello di elaborazione dei dati più potente e flessibile rispetto a MapReduce. Per quanto riguarda l'elaborazione dei dati, sebbene Hadoop sia dotato di un proprio framework di elaborazione dati - nonché MapReduce -, Spark può fungere da sostituto di questo componente, offrendo miglioramenti significativi delle prestazioni per determinati carichi di lavoro. Un processo Spark può caricare e memorizzare nella cache i dati in memoria ed eseguirne query ripetutamente. L'elaborazione in memoria è molto più veloce delle applicazioni basate su disco, come Hadoop, che condivide i dati tramite il file system distribuito Hadoop (HDFS).

Nel momento in cui uno Spark job viene inviato ad un cluster, vengono svolti i seguenti passaggi:

1. Job Submission: un utente commissiona uno Spark job specificando diversi parametri, come il path dell'applicazione, il Master - nonché dove viene eseguito il programma Driver -, eventuali file jar o Python aggiuntivi.
2. Application Master Initialisation: una volta commissionato il job, viene avviata un'Application Master per l'applicazione. Si tratta di un'istanza specifica di un programma driver Spark in esecuzione su un nodo nel cluster.
   L'Application Master comunica con il Resource Manager, il quale supervisiona e gestisce le risorse nel cluster. Il driver crea anche uno *SparkContext*.
   L'Application Master richiede, quindi, risorse per gli executors specificando quante ne sono necessarie in base ai requisiti dell'applicazione.
   Una volta che il Resource Manager approva queste richieste e alloca le risorse, passa all'avvio dei processi dell'executor sui nodi allocati.
3. Task execution: l'executor esegue queste attività in più thread che sono processi JVM separati. Gli executors deserializzano l'attività, la eseguono e, se necessario, serializzano il risultato prima di inviarlo al driver.
4. Data shuffling: se sono presenti trasformazioni nello Spark job che richiedono lo spostamento dei dati (come un *group by* o *distinct*), Spark lo gestirà. Questo può comportare la riorganizzazione dei dati nel cluster e solitamente corrisponde alle fasi dell'esecuzione del job.
5. Job Completion: una volta completata tutte le attività, il risultato viene inviato al programma del driver. Il job termina o quando la funzione principale del programma termina, oppure quando viene chiamata *SparkContext.stop()*. Infine, le risorse allocate sul cluster, come i core della CPU e la memoria dei nodi di lavoro, vengono liberate per essere utilizzate da altre applicazioni.

4.2 Azure storage account

Uno storage account è una risorsa di Azure Resource Manager, ossia il servizio di distribuzione e gestione per Azure.
Per il progetto in questione, ci si è serviti dell'archiviazione BLOB di Azure, utile nei contesti di archiviazione di oggetti (o più in generale dati) per il cloud. Di fatti, l'accesso ai dati non avviene mediante HDFS, bensì tramite ABFSS - acronimo di Azure Blob File System Storage.
L'archiviazione BLOB offre tre tipi di risorse:

- L'account di archiviazione, che in questo caso è *"clustersparkstoragea"*
- Un container all'interno dello storage account, che in questo caso si chiama *"dati"*
- Oggetti blob interni al container

Di seguito un diagramma puramente esplicativo:

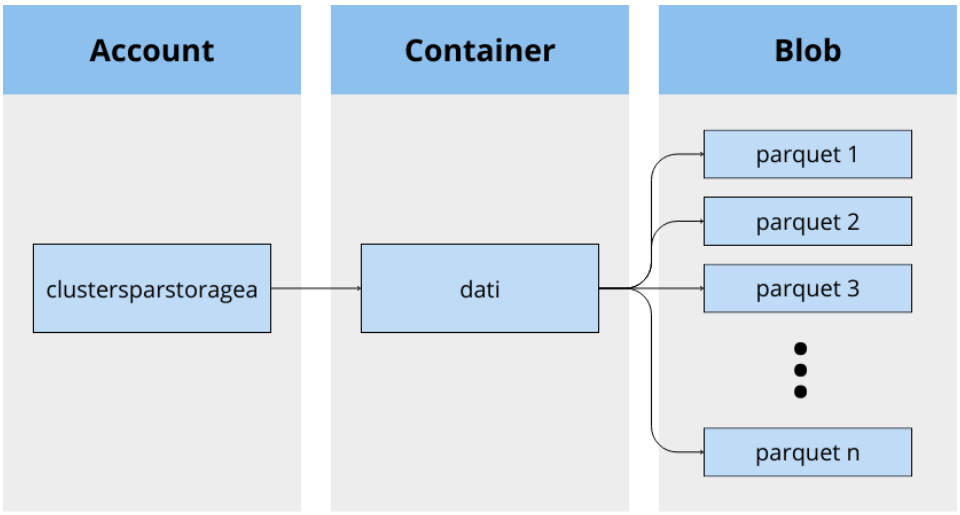

Figura 4.6: Esempio dell'architettura di storage account.

Un container consente di organizzare un set di BLOB in modo analogo all'organizzazione di una directory in un file system. Lo storage account può contenere un numero illimitato di container, ciascuno dei quali può, a sua volta, archiviare un numero illimitato di BLOB.
Il nome del container deve avere un nome DNS valido, poiché fa parte dell'URI (Uniform Resource Identifier) univoco utilizzato per accedere al container e ai relativi BLOB.
La struttura generale di uno URI è:

https://myaccount.blob.core.windows.net/mycontainer

In particolare, per questo progetto, si è ricorso ad Azure Data Lake Storage Gen2, ossia un un set di funzionalità dedicate all'analisi dei Big Data, basate su Archiviazione BLOB di Azure.
Un data lake è un singolo repository centralizzato in cui è possibile archiviare tutti i dati, sia strutturati che non strutturati.

Azure Data Lake Storage sfrutta ABFSS, che permette a molte applicazioni e framework di accedere direttamente ai dati di Archiviazione BLOB di Azure. ABFSS è ottimizzato in modo specifico per l'analisi di Big Data. Le API corrispondenti sono rilevate tramite

l'endpoint *dfs.core.windows.net*.

Spark effettua la connessione ad Azure Data Lake Gen2 seguendo gli step qui riportati

1. **Definizione delle credenziali**: Prima di tutto, è necessario essere in possesso delle credenziali di accesso allo storage account, principalmente alla chiave di accesso (access key) e connection string, che sono univoche per quell'account.

2. **Configurazione di Spark**: Una volta ottenute le credenziali, basta impostarle su Spark in modo da configurarlo per l'accesso allo storage

3. **Utilizzo del protocollo ABFSS**: Quando Spark accede ai dati in Azure Data Lake, lo fa utilizzando il protocollo ABFSS. Questo gli consente di comunicare con l'archiviazione Blob di Azure.

4. **Accesso ai dati**: Una volta stabilita la connessione, mediante il protocollo ABFSS e le credenziali appropriate, Spark è in grado di leggere e scrivere dati da e verso Azure Data Lake Gen2 come fosse un filesystem locale. Questo consente agli utenti di eseguire operazioni di trasformazione, analisi ed elaborazione dei dati, senza doversi occupare manualmente del trasferimento di questi utlimi tra lo storage e il cluster Spark.
   Un esempio di accesso ai dati è il seguente:

   abfss://containername@accountname.dfs.core.windows.net/file_path

Capitolo 5

Monitoraggio tempistiche

Nell'ambito dell'analisi e dell'ottimizzazione delle prestazioni dei sistemi di elaborazione distribuita, la valutazione dei tempi di esecuzione rappresenta un fattore cruciale per comprendere l'efficienza e la scalabilità di tali sistemi.

Uno degli obiettivi di questo elaborato è quello di esaminare due aspetti distinti ma complementari dell'efficienza computazionale: la **scalabilità orizzontale**, valutata mediante la variazione del numero di nodi Spark mantenendo costante il volume di dati, e la **scalabilità verticale**, esaminata attraverso l'incremento della quantità di dati elaborati conservando la stessa configurazione.

La prima analisi si concentrerà sulla misurazione dei tempi di esecuzione al variare del numero di nodi Spark, al fine di comprendere come l'aggiunta di risorse di calcolo influenzi le prestazioni complessive del sistema. Questo studio consentirà di valutare l'efficacia della distribuzione del carico di lavoro su più nodi e di identificare eventuali limitazioni o punti di saturazione nell'architettura distribuita.

La seconda analisi, invece, si concentrerà sulle prestazioni in relazione all'aumento della quantità di dati processati, esaminando come la dimensione del dataset influenzi i tempi di esecuzione su una singola configurazione. Questo approfondimento mira a fornire una comprensione chiara delle performance in scenari in cui la distribuzione su più nodi potrebbe non essere praticabile o vantaggiosa.

Attraverso queste due prospettive analitiche, ci si propone di fornire un quadro completo delle prestazioni del sistema in esame, consentendo di identificare le migliori strategie per ottimizzare le operazioni di elaborazione distribuita e massimizzare l'efficienza computazionale.

5.1 Scalabilità orizzontale

Per effettuare un'analisi di confronto dei tempi di calcolo, lavorando col dataset NYC Taxi e sfruttando diverse macchine, si è deciso di determinare una certa quantità di dati per poi osservare come variasse la complessità temporale al variare del numero di macchine e della modalità di Spark utilizzate.
Tale quantità di dati ricopre l'arco di tempo che va da Gennaio 2011 a Novembre 2023. Si tratta di un DataFrame Spark di dimensione 5.4 MiB - circa 5.7 MB - per un totale di 1,216,907,277 record (dati ottenuti mediante interfaccia web Spark).
Una volta stabilita la quantità di dati di lavoro, ci si è concentrati sull'effettuare un confronto

andando a variare il numero di macchine utilizzate. In particolare, tenendo fissa la quantità di dati sono stati svolti i seguenti esperimenti:

1. Spark in local mode
2. Spark in modalità cluster standalone con il master ed un worker
3. Spark in modalità cluster standalone con il master e due workers
4. Spark in modalità cluster standalone con il master e tre workers

Per quel che concerne l'applicazione Spark scritta per il presente elaborato, si possono distinguere le seguenti operazioni all'interno del processo:

- **Loading**: Fase in cui vengono caricati tutti i file parquet contenuti nello storage account e "letti" come un unico grande DataFrame Spark
- **Join**: Questa fase vale solo per la costruzione del DataFrame sui Dipartimenti, in cui è necessario effettuare un'operazione di Join tra il DataFrame complessivo dei dati e quello relativo ai dipartimenti
- **Query**: Si fa riferimento alla query SQL con la quale si filtrano i dati per poi costruire i nuovi DataFrame Spark
- **Writing**: Fase in cui i vari DataFrame Spark ottenuti dalla fase precedente vengono salvati e scritti nello storage account come file parquet

5.1.1 Analisi risultati

L'obiettivo finale di questo confronto è quello di valutare come, lavorando con un quantitativo di dati fisso, al variare dei nodi Spark utilizzati, i tempi di calcolo cambiano.

Risulta importante, in questo contesto, valutare la velocità di calcolo in quanto permette di comprendere quanto può o non può risultare opportuno utilizzare più o meno macchine, e per quale specifica operazione.

Si riportano i grafici relativi, prima, ai tempi ottenuti per il processo relativo al DataFrame sui distretti, in cui - come anticipato - vi è anche il plot relativo all'operazione di Join 5.1.

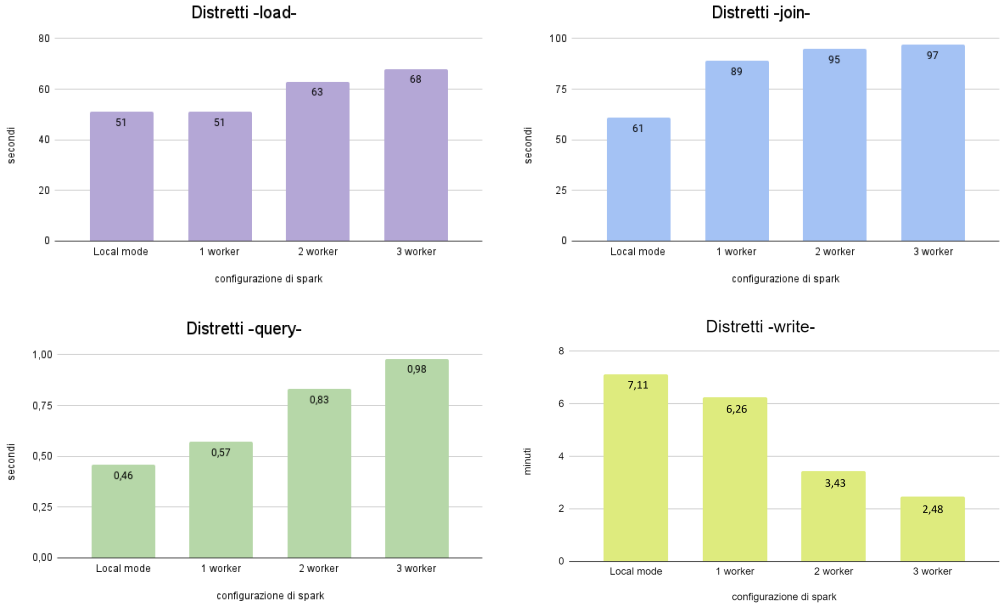

Figura 5.1: *Distretti:* variare del tempo di computazione in relazione alla tipologia di configurazione usata. I quattro grafici rappresentano le operazioni necessarie per la creazione e per il salvataggio del nuovo DataFrame.

Vengono, ora, allegati i plot relativi al processo legato alla creazione del DataFrame sulla tipologia dei pagamenti 5.2:

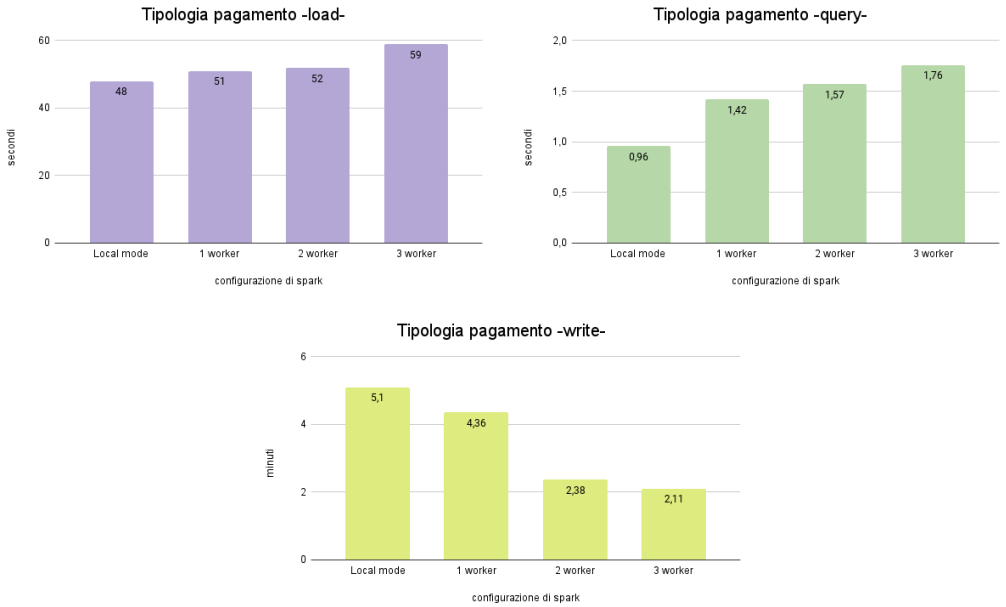

Figura 5.2: *Tipologia pagamento:* variare del tempo di computazione in relazione alla tipologia di configurazione usata. I tre grafici rappresentano le operazioni necessarie per la creazione e per il salvataggio del nuovo DataFrame.

In ultimo si riportano i grafici relativi ai tempi di calcolo del processo relativo al DataFrame sul numero dei passeggeri 5.3.

Come si può evincere dai plot riportati, i processi relativi ai tre diversi DataFrame Spark presentano comportamenti analoghi.
Per quanto riguarda l'operazione di loading dei dati, si osserva che all'aumentare dei nodi utilizzati aumenta gradualmente anche il tempo di computazione, incrementando di circa 10 secondi dalla prima rispetto all'ultima configurazione.
In particolare, l'aumento diventa significativamente visibile dall'utilizzo di Spark in modalità cluster standalone con un nodo worker ed uno master a quello in cui si utilizzano tre nodi worker ed uno master; mentre, la velocità di calcolo resta equiparabile quando si fa riferimento all'utilizzo di una sola macchina, che sia in modalità cluster standalone o local mode.

Durante il processo relativo al DataFrame sui dipartimenti, vi è anche un'operazione di join. In questo caso, invece, si nota un incremento dei tempi più rilevante nel passaggio dalla prima configurazione (local mode) alla seconda (cluster standalone con un worker ed un master). Dopo di ché, la velocità di calcolo si dilata progressivamente fino all'ultima configurazione (tre nodi worker ed uno master).

Sui grafici relativi all'operazione di query c'è da evidenziare, prima di tutto, che ha tempi di calcolo sempre inferiori ai due secondi; questa operazione, infatti, risulta essere la più rapida tra tutte quelle analizzate. Essendo la differenza di tempo tra una configurazione e l'altra riducibile a millisecondi, si tratta di una misurazione più variabile rispetto alle precedenti.
Analogamente all'operazione di join, si osserva che i tempi si dilatano all'aumentare di nodi interni alla configurazione.

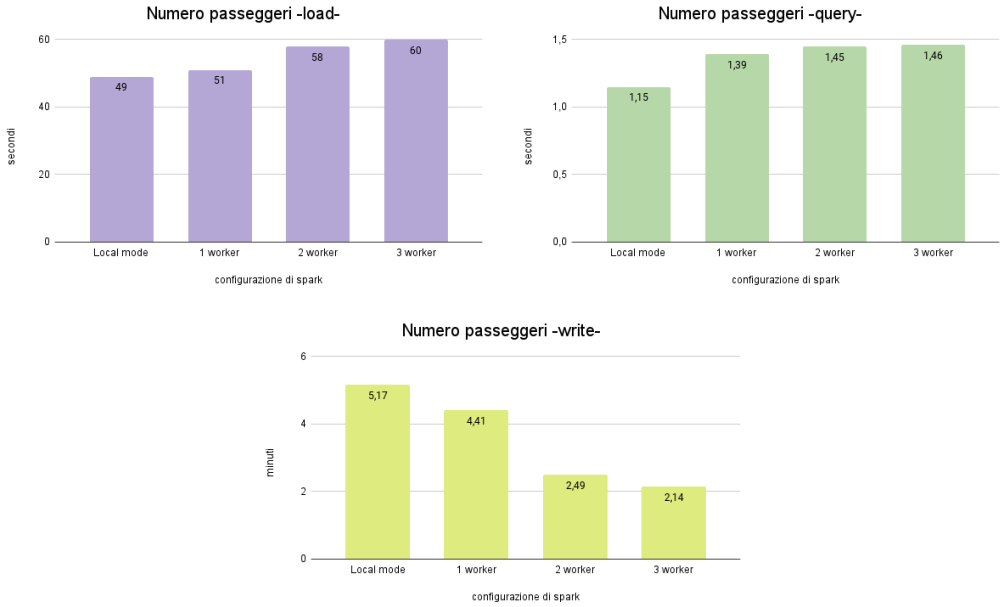

Figura 5.3: *Numero passeggeri:* variare del tempo di computazione in relazione alla tipologia di configurazione usata. I tre grafici rappresentano le operazioni necessarie per la creazione e per il salvataggio del nuovo DataFrame.

L'ultima operazione da analizzare resta quella della scrittura dei nuovi DataFrame Spark in file parquet all'intero dello storage account. In forte contrasto con quanto descritto per le operazioni antecedenti, nella fase di writing vi è un comportamento opposto: all'amentare del numero di nodi interni alla configurazione Spark, i tempi di calcolo diminuiscono.
L'operazione di scrittura risulta essere quella più onerosa, infatti il tempo di esecuzione richiede dai 7-5 minuti nel caso della configurazione local mode, a circa 2 minuti nel caso della modalità cluster standalone con tre nodi worker ed uno master.
In questa casistica si nota, per tutti e tre i processi dei DataFrame, un lieve miglioramento dei tempi tra l'esecuzione in local mode e quella in cluster standalone con un solo nodo worker ed uno master. Il vero margine di miglioramento lo si osserva all'aumentare dei nodi worker nella configurazione Spark standalone.

Per quanto riguarda il job di scrittura, essendo quello più costoso in termini di tempo, si è deciso di effettuare un'analisi di approfondimento riguardo la quantità di task generati da Spark per portare a termine il salvataggio del file (5.4).
Si osserva subito che la differenza tra quello che accade quando si utilizza un solo worker in local mode e quando si utilizzano un nodo master ed uno worker è quasi impercettibile a livello di quantità di task.
Questa analogia è dovuta al fatto che in entrambi i casi vi è un solo nodo esecutore disponibile, pertanto non vi può essere chissà quale ottimizzazione relativa alla distribuzione del lavoro; bisogna adattarsi ai limiti di capacità computazionale dell'unica macchina disponibile.
Dacché si utilizzano un nodo master e due workers in modalità Spark standalone si osserva un significativo aumento del numero di task: la quantità, infatti, raddoppia rispetto alle

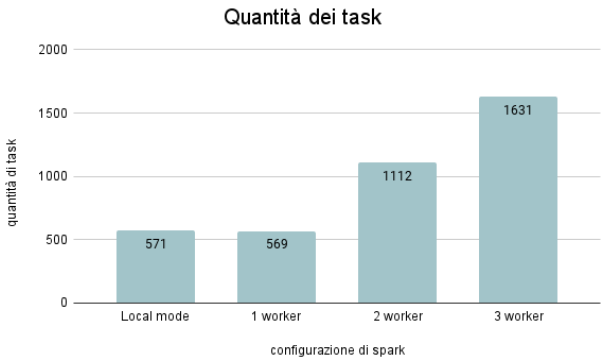

Figura 5.4: Quantità di task in relazione alla configurazione di spark usata.

configurazioni precedenti. Questo suggerisce che l'aggiunta di una seconda macchina ha comportato una distribuzione più efficiente del carico di lavoro.
Con l'inserimento del terzo Worker all'interno della configurazione, la quantità di task aumenta ulteriormente, indicando che si è apportato un ulteriore miglioramento della distribuzione del carico di lavoro.

Un numero maggiore di task consente una maggiore parallelizzazione delle operazioni. Ciò significa che più operazioni possono essere eseguite contemporaneamente, riducendo - generalmente- il tempo complessivo necessario per completare il lavoro, come avviene durante il job di scrittura.

5.2 Scalabilità verticale

Un secondo esperimento riguardo il confronto dei tempi di calcolo è stato effettuato variando la quantità di dati ma mantenendo sempre la stessa configurazione dell'ambiente Spark.
In particolare, si è partiti dalla quantità di dati impostata al punto precedente e, si è deciso di adoperare con due configurazioni Spark: in local mode su uno dei tre nodi worker, e in cluster standalone mode con tre nodi worker. Così facendo è risultato possibile osservare come variassero le tempistiche andando ad utilizzare prima il 25%, poi il 50%, poi il 75% ed infine il 100% dei file contenuti all'interno dello storage account.
Tenendo conto che i file parquet contenuti all'interno dello storage account hanno lunghezza differente in base al numero di record registrati, si è ritenuto importante evidenziare la quantità effettiva dei record per ciascuna frazione di dati considerata. Pertanto, si riporta un plot (vedere 5.5) chiarificatore da cui è possibile osservare che:

- Il DataFrame ottenuto usando il 25% dei file è composto da 454,458,694 record
- Il DataFrame ottenuto usando il 50% dei file è composto da 853,364,739 record
- Il DataFrame ottenuto usando il 75% dei file è composto da 1,103,612,467 record
- Il Dataframe totale è composto da 1,216,907,27 record

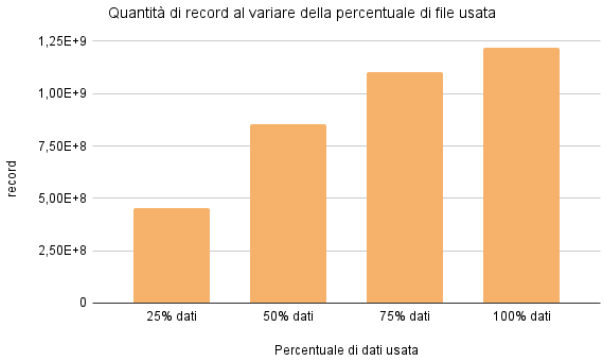

Figura 5.5: Quantità di dati utilizzati.

5.2.1 Analisi risultati

L'obiettivo finale di questo confronto è quello di valutare come, lavorando con un'unica configurazione Spark, al variare della quantità di dati, i tempi di calcolo cambiano.

Di seguito si riportano i plot 5.6 raffiguranti i tempi di computazione per le operazioni utili alla creazione di uno dei tre DataFrame Spark (Tipologia di pagamento), nonché: caricamento dei dati, query e salvataggio del nuovo DataFrame ottenuto con annessa scrittura in file parquet.
In particolare, ogni grafico rappresenta la variazione dei tempi di calcolo all'aumentare della quantità di dati utilizzata: prima il 25%, poi il 50%, poi il 75%, ed infine il 100%.

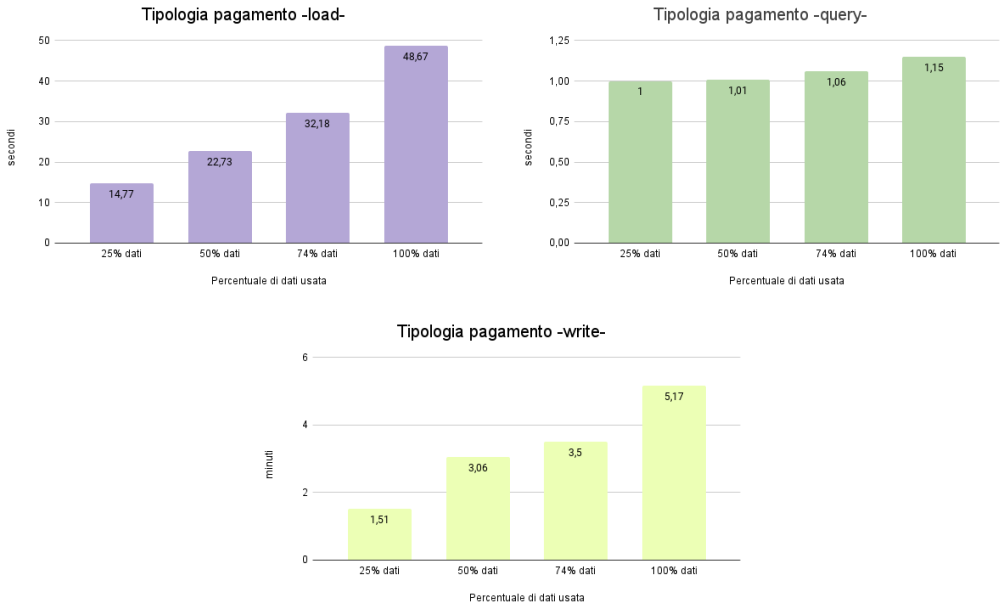

Figura 5.6: *Tipologia pagamento:* variare del tempo di computazione in relazione alla percentuale di dati usata. I tre grafici rappresentano le operazioni necessarie per la creazione e per il salvataggio del nuovo DataFrame in modalità local mode.

Osservando i plot 5.6 5.7 si può affermare che la variazione dei tempi di calcolo assume lo stesso comportamento per tutte e tre le operazioni analizzate.
Mentre le operazioni di caricamento e di scrittura riportano dei tempi computazionali che aumentano coerentemente con l'aumento della quantità di dati utilizzata 5.5- ossia di circa il 25% ogni volta -; per quanto riguarda l'operazione di query, che si ricorda essere la meno onerosa in termini di tempi di computazione, si osserva una dilatazione dei tempi quasi impercettibile: si passa, infatti, da 1 secondo nella prima configurazione, ad 1.15 secondi nell'ultima.

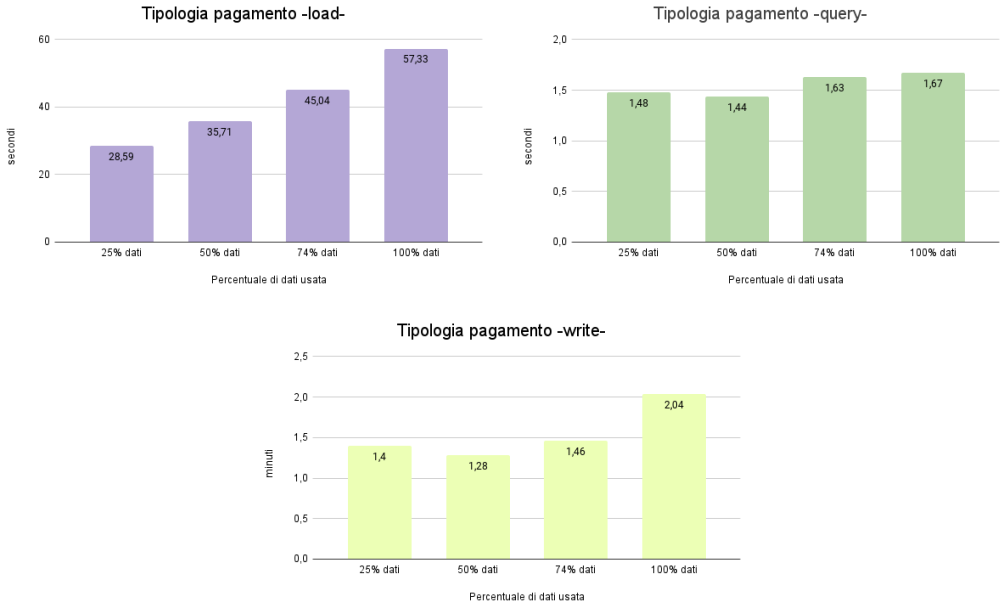

Figura 5.7: *Tipologia pagamento:* variare del tempo di computazione in relazione alla percentuale di dati usata. I tre grafici rappresentano le operazioni necessarie per la creazione e per il salvataggio del nuovo DataFrame in modalità Spark cluster standalone con 3 nodi worker.

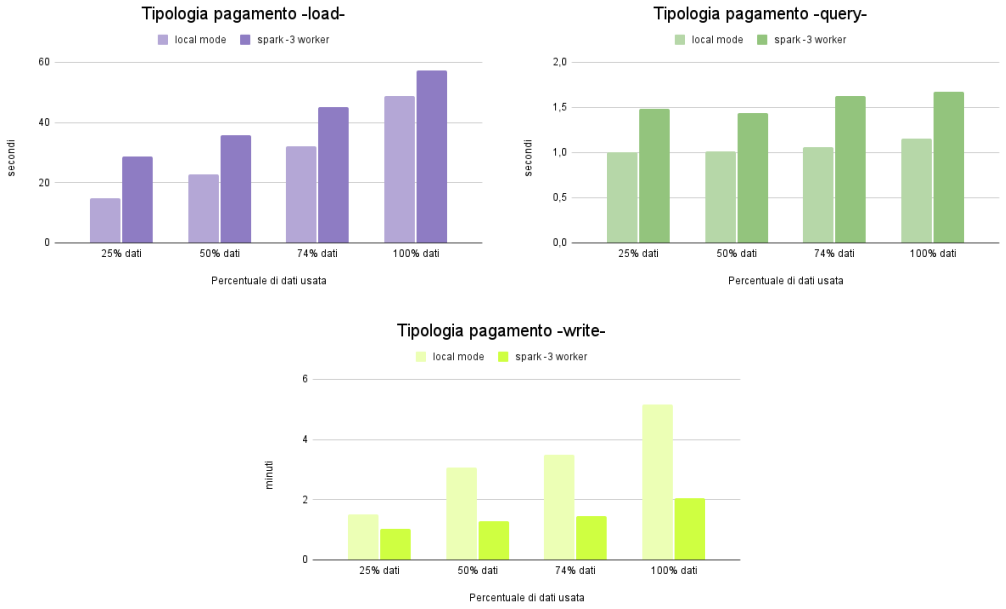

Figura 5.8: *Tipologia pagamento:* variare del tempo di computazione in relazione alla percentuale di dati usata. I tre grafici rappresentano le operazioni necessarie per la creazione e per il salvataggio del nuovo DataFrame. Si confrontano i tempi ottenuti in modalità local mode e in modalità Spark cluster standalone con 3 nodi worker-

Il plot 5.8 riporta un confronto incrociato tra le analisi di scalabilità orizzontale e verticale.
Come precedentemente riportato, in entrambi i casi all'aumentare della quantità di dati usata aumentano anche i tempi computazionali.
Osservando invece il confronto tra i tempi in local mode e in modalità Spark cluster standalone con tre nodi worker si osserva che, indipendentemente dalla percentuale di dati utilizzata, i tempi di computazione in local mode rimangono sempre inferiori quando si effettuano le operazioni di load e di query. Mentre, quando si effettua la scrittura del nuovo DataFrame creato risulta più efficiente utilizzare la modalita Spark cluster.

Tale analisi riguardante la linearità tra la quantità di dati e la modalità di Spark utilizzata confermano quanto ottenuti dalla sezione precedente (vedere 5.1).

Capitolo 6

Conclusioni

Come precedentemente illustrato, lo scopo ultimo del progetto in analisi è quello di comprendere come gestire un quantitativo di dati molto ampio lavorando su larga scala, ossia andando a disporre un ambiente in grado di elaborare tali dati parallelamente, e in tempi discreti.
Pertanto, per la realizzazione di questo elaborato ci si è occupati di configurare l'ambiente Spark cluster standalone utilizzando una macchina virtuale Azure come nodo Master, ed altre tre macchine virtuali Azure con risorse computazioni maggiori e tra loro equivalenti, come nodi Worker.
Inoltre, i file parquet relativi al dataset analizzato sono stati allocati all'interno di un Azure storage account, in modo da renderli accessibili parallelamente da ognuna delle macchine utilizzate.

Comprensibilmente, la fase più elaborata del progetto è proprio stata quella di configurare l'ambiente in modo che tutte le componenti comunicassero come previsto.

Volendo poi, testare tale configurazione, si è lavorato sull'implementazione di tre applicazioni Spark utili alla costruzione di nuovi DataFrame derivati dal set di dati originale.
Si rientra, così, in quella che è l'analisi dei risultati effettuata sui tempi di calcolo nei due seguenti esperimenti:

1. Scalabilità orizzontale: quanto variano i tempi al variare del numero di nodi utilizzati
2. Scalabilità verticale: quanto variano i tempi al variare della quantità di dati

Di seguito si riportano le due tabelle 6.1 6.2 sui risultati ottenuti, in termini di tempi computazionali, ripercorrendo entrambi gli esperimenti effettuati sui vari DataFrame costruiti.

		load	query	write	join
Dipartimenti	*Local mode*	00:51:00	00:00:46	07:11:00	00:00:61
	1 worker	00:51:05	00:00:57	06:26:00	00:00:89
	2 worker	01:03:00	00:00:83	03:43:00	00:00:95
	3 worker	01:08:00	00:00:98	02:48:00	00:00:97
Tipologia pagamento	*Local mode*	00:48:67	00:00:96	05:17:00	
	1 worker	00:51:37	00:01:42	04:36:00	
	2 worker	00:57:94	00:01:57	02:38:00	
	3 worker	00:59:69	00:01:76	02:11:00	
Numero passeggeri	*Local mode*	00:48:67	00:01:15	05:17:00	
	1 worker	00:51:22	00:01:39	04:41:00	
	2 worker	00:58:16	00:01:45	02:49:00	
	3 worker	00:59:94	00:01:46	02:14:00	

Figura 6.1: Tabella di confronto riportante i tempi computazionali (mm:ss:mmm) necessari per la creazione dei tre nuovi DataFrame. Le colonne corrispondono alle operazioni necessarie per la relativa creazione e salvataggio; le righe rappresentano, per ogni nuovo dataframe, le quattro configurazioni Spark analizzate.

Come è emerso nell'analisi dei risultati precedentemente argomentata (vedere Sez. 5.1.1), si può constatare che vi sono operazioni in cui risulta più efficiente l'utilizzo della modalità Spark cluster standalone con l'ausilio di più nodi, ed altre dove, invece, appare più vantaggioso evitare le connessioni all'interno del cluster e sfruttare il framework Spark in local mode su un'unica macchina esecutrice.

Infatti, per quanto riguarda le operazioni di caricamento, join e query, si è osservato che all'aumentare dei nodi, i tempi di esecuzione si dilatassero, seppur di poco, indicando che per tali operazioni - considerando gli specifici dati in analisi - si è rivelato più efficiente l'utilizzo di un'unica macchina con Spark in local mode.

L'operazione di scrittura, invece, ha presentato un andamento opposto, suggerendo che per un'esecuzione più rapida risulta più consono l'utilizzo di un numero maggiore di nodi in modalità Spark cluster standalone.
In un'operazione onerosa come questa, si può supporre che un maggior numero di nodi, si possa tradurre in una maggior parallelizzazione delle operazioni e quindi in un miglioramento della distribuzione del carico di lavoro; il ché implica un decremento significativo dei tempi computazionali.

Si riporta ora la tabella relativa ai risultati ottenuti dal secondo esperimento, nonché quello in cui si è osservato come variassero i tempi utilizzando una sola macchina in modalità Spark local mode e utilizzando Spark in modalita cluster standalone con 3 nodi worker, al viarare della quantità di dati.
I risultati esposti dalla tabella inducono a supporre che nelle operazioni di loading e writing i tempi di esecuzione aumentano quasi in modo direttamente proporzionalmente all'aumentare della quantità di dati.

		load	query	write
Local mode	***25%***	00:14:77	00:01:00	01:51:00
	50%	00:22:73	00:01:01	03:06:00
	75%	00:32:18	00:01:06	03:50:00
	100%	00:48:67	00:01:15	05:17:00
Spark cluster standalone 3 worker	***25%***	00:28:59	00:01:48	01:04:00
	50%	00:35:71	00:01:44	01:28:65
	75%	00:45:04	00:01:63	01:46:50
	100%	00:57:33	00:01:67	02:04:02

Figura 6.2: Tabella di confronto riportante i tempi computazionali (mm:ss:mmm) necessari per la creazione del DataFrame -Tipologia pagamento- in modalità local mode e in modalida cluster standalone con 3 nodi worker. Le colonne corrispondono alle operazioni necessarie per la relativa creazione e salvataggio; le righe rappresentano, le percentuali di dati usate.

Mentre, un andamento curioso si osserva guardando i risultati relativi ai tempi di esecuzione dell'operazione di query. Sebbene anche questi aumentano all'aumentare dei dati, l'incremento risulta decisamente poco significativo e quasi non evidente: basti pensare che utilizzando il 25% dei dati la query impiega 1 secondo, e che lavorando, invece, col 100% dei dati l'operazione richiede solo 15 millesimi di secondo in più.

Gli esiti appena analizzati, conducono ad ipotizzare che per le operazioni di caricamento e scrittura - almeno per quanto riguarda il dataset sotto analisi - la quantità di dati di lavoro influisce molto sui tempi di esecuzione; mentre, per l'operazione di query non vi è una differenza così significativa nel trattare più o meno dati - nonostante comunque i tempi si dilatano, ma in termini minimi.

Inoltre tale analisi conferma ulteriormente i risultati commentati nella tabella precedente 6.1, in quanto indipendentemente dalla percentuale di dati usata, nel caso delle operazioni load e query conviene sempre usare Spark in modalità locale, mentre quando si effettua l'operazione di scrittura risulta più efficiente utilizzare Spark in modalità cluster con un maggior numero di nodi.