Università degli Studi di Milano Bicocca

Dipartimento di Informatica

Corso di Laurea Magistrale

Corso di: ”Large Scale Data management”

Progetto data warehouse

Lorenzo Megna - 868929

Oscar Sacilotto - 866040

ANNO ACCADEMICO 2024/2025

Indice

Introduzione

Per il progetto si è deciso di scegliere come dominio applicativo due dataset relativi ai taxi di New York: i taxi verdi e i taxi gialli.

I taxi gialli (Yellow Taxis) costituiscono la forma tradizionale di trasporto a chiamata nella città di New York. Operano principalmente nei distretti centrali come Manhattan e possono essere fermati direttamente per strada (street-hailing). I taxi verdi (Green Taxis), invece, sono stati introdotti successivamente per migliorare la copertura del servizio nelle aree periferiche della città, come il Bronx, Brooklyn, Queens e Upper Manhattan. A differenza dei taxi gialli, non possono raccogliere passeggeri in strada nella zona centrale di Manhattan, ma sono autorizzati a operare principalmente fuori da essa e su prenotazione.

Il progetto si articolerà in due fasi principali. La prima consisterà nella creazione di un flusso automatico che permetta l'integrazione delle due fonti dati. Successivamente, si procederà alla progettazione dello schema concettuale con la definizione delle tabelle dei fatti e delle varie dimensioni. In conclusione verranno effettuate alcune *query* per testarne la velocità.

1. Introduzione al dataset

1.1 Taxi Gialli

Il dataset relativo ai *taxi gialli* (Yellow Taxis) costituisce una delle principali fonti pubbliche utilizzate per l'analisi della mobilità urbana a New York. Il dataset raccoglie milioni di corse effettuate. Quello utilizzato è inerente agli anni 2017, 2018 e 2019.

Descrizione delle variabili principali

Di seguito si riportano le principali variabili contenute nel dataset, suddivise per categorie logiche:

- **Informazioni di identificazione e configurazione corsa**:
  - `VendorID`: Identificativo del fornitore del servizio taxi (di solito rappresenta la compagnia).
  - `store_and_fwd_flag`: Flag che indica se i dati della corsa sono stati memorizzati nel veicolo prima di essere inviati al server centrale.
  - `RatecodeID`: Codice della tariffa applicata (es. standard, JFK, ecc.).
  - `trip_type`: Tipo di corsa (1 = standard, 2 = chiamata via app o centralino).
- **Informazioni temporali e geografiche**:
  - `lpep_pickup_datetime`, `lpep_dropoff_datetime`: Timestamp di inizio e fine della corsa.
  - `PULocationID`, `DOLocationID`: ID delle zone di partenza e arrivo, secondo la codifica del *Taxi Zone Lookup*.
- **Dettagli sulla corsa**:
  - `passenger_count`: Numero di passeggeri trasportati.
  - `trip_distance`: Distanza percorsa (in miglia).

- **Informazioni economiche e di pagamento**:
  - `fare_amount`: Tariffa base calcolata per la corsa.
  - `extra`: Supplementi (es. notturni o traffico).
  - `mta_tax`: Tassa della Metropolitan Transportation Authority.
  - `tip_amount`: Mancia versata dal passeggero.
  - `tolls_amount`: Totale dei pedaggi pagati.
  - `improvement_surcharge`: Supplemento per il miglioramento del servizio taxi.
  - `total_amount`: Totale corrisposto, comprensivo di tutti i componenti tariffari.
  - `payment_type`: Metodo di pagamento (1 = contanti, 2 = carta, ecc.).
- **Variabili temporali derivate (per analisi e join)**:
  - `Date`, `FiscalYear`, `FiscalQuarter`, `FiscalMonthNumber`, `FiscalMonthOfQuarter`, `FiscalWeekOfYear`, `DayOfWeek`, `FiscalMonthName`, `FiscalMonthYear`, `FiscalQuarterYear`, `DayOfMonthNumber`, `DayName`: Variabili temporali derivate dai timestamp per facilitare le operazioni di aggregazione, filtro e join con la dimensione temporale.
- **Informazioni sulle zone geografiche**:
  - `LocationID`: Identificativo univoco della zona (può riferirsi a `PULocationID` o `DOLocationID`).
  - `Borough`: Quartiere della città (es. Manhattan, Brooklyn).
  - `Zone`: Nome specifico della zona (es. JFK Airport, Upper East Side).
  - `service_zone`: Classificazione della zona secondo le categorie del servizio taxi (es. Yellow Zone, Airport).

1.2 Taxi Verdi

Il dataset dei *taxi verdi* (Green Taxis) nasce con l'obiettivo di estendere il servizio taxi nelle aree periferiche della città di New York, in particolare nei distretti meno centrali come il Bronx, Brooklyn, Queens e Upper Manhattan. A differenza dei taxi gialli, i taxi verdi non possono effettuare

il servizio di *street-hailing* nel centro di Manhattan, ma possono raccogliere passeggeri nelle zone esterne e tramite prenotazione.

Purtroppo di questo dataset abbiamo le informazioni inerenti all'anno 2019.

Descrizione delle variabili principali

Di seguito vengono descritte le principali variabili contenute nel dataset dei taxi verdi, organizzate per categoria:

- **Informazioni di identificazione e configurazione corsa**:
  - `vendorid`: Identificativo del fornitore del servizio taxi.
  - `store_and_fwd_flag`: Flag che indica se i dati sono stati salvati temporaneamente nel veicolo prima della trasmissione.
  - `rate_code`: Codice della tariffa applicata.
  - `trip_type`: Tipo di corsa (1 = standard, 2 = prenotazione via app o centralino).
- **Informazioni temporali e geografiche**:
  - `pickup_datetime`, `dropoff_datetime`: Timestamp di inizio e fine corsa.
  - `pickup_longitude`, `pickup_latitude`: Coordinate geografiche del punto di partenza.
  - `dropoff_longitude`, `dropoff_latitude`: Coordinate geografiche del punto di arrivo.
- **Dettagli sulla corsa**:
  - `passenger_count`: Numero di passeggeri trasportati.
  - `trip_distance`: Distanza percorsa (in miglia).
- **Informazioni economiche e di pagamento**:
  - `fare_amount`: Tariffa base della corsa.
  - `extra`: Supplementi (es. traffico, orari notturni).
  - `mta_tax`: Tassa applicata dalla Metropolitan Transportation Authority.

  - `tip_amount`: Mancia pagata dal cliente.
  - `tolls_amount`: Pedaggi sostenuti.
  - `ehail_fee`: Tariffa legata a prenotazioni elettroniche (frequentemente mancante).
  - `improvement_surcharge`: Sovrapprezzo per il miglioramento del servizio.
  - `total_amount`: Totale pagato dal passeggero.
  - `payment_type`: Metodo di pagamento utilizzato.

- **Variabili temporali derivate**: Variabili calcolate per abilitare analisi temporali e join coerenti con la dimensione tempo.
  - `FiscalYear`, `FiscalQuarter`, `FiscalMonthNumber`, `FiscalMonthOfQuarter`, `DayOfWeek`, `FiscalWeekOfYear`, `FiscalMonthName`, `FiscalMonthYear`, `FiscalQuarterYear`, `DayOfMonthNumber`, `DayName`

- **Informazioni geografiche derivate**:
  - `LocationID`: Identificativo della zona di pickup/dropoff.
  - `Borough`: Quartiere della città.
  - `Zone`: Zona specifica all'interno del borough.
  - `service_zone`: Classificazione della zona secondo la TLC.

2. ETL e Star Schema

In questa sezione ci concentreremo sul creare il *"percorso"* che i dati, provenienti dalle due sorgenti, effettuano per fondersi in un unico grande schema generale.

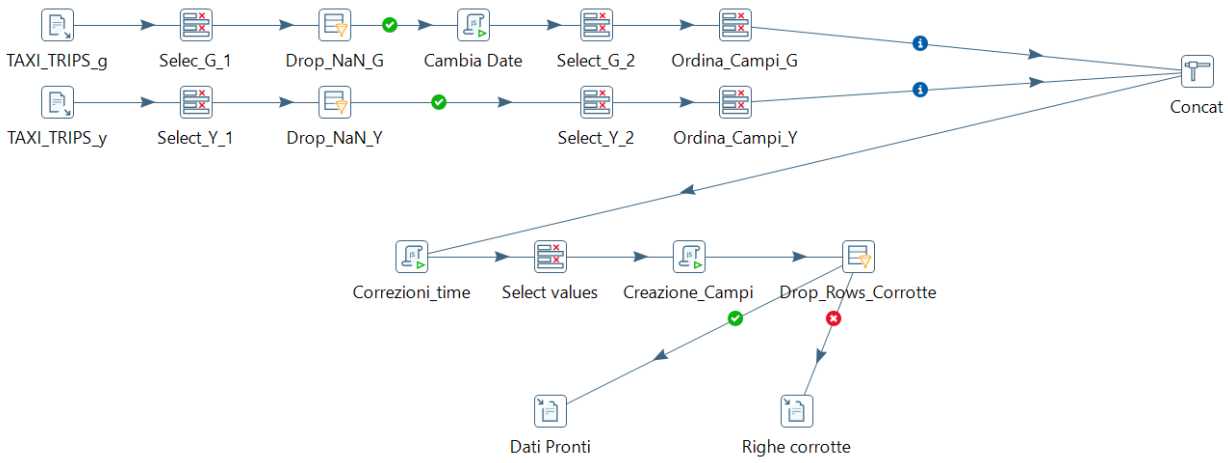
Successivamente, andremo a definire lo schema che adotteremo per salvare i dati.

2.1 ETL

La parte di integrazione dei dati è stata effettuata utilizzando il software (in versione gratuita) Penthao[1]. Avendo a disposizione solo 1 anno di *taci verdi* si è deciso di creare 2 dataset. Il primo con le istanze, delle due tipologie di taxi, inerenti allo stesso anno. Il secondo è stato popolato con tutte le informazioni a disposizione; sono stati ottenuti un dataset da circa **1.4GB** con al suo interno *circa 4 milioni di righe* e uno da **3.7GB** con *circa 12 milioni di righe*.

In questa sezione andremo a vedere le variabili che sono state create e selezionate.

2.1.1 Penthao



[1] https://pentaho.com/products/pentaho-data-integration/

Questo è il processo di ETL. Di seguito sono descritte le principali fasi del processo:

- **Import dei dati**: I primi due *job* servono per importare nel software: `TAXI_TRIPS_g` e `TAXI_TRIPS_y`.
- **Selezione iniziale dei campi**: I due dataset vengono letti in parallelo e vengono selezionati soltanto i campi rilevanti tramite i blocchi `Select_G_1` e `Select_Y_1`, riducendo così la dimensionalità iniziale.
- **Rimozione dei valori nulli**: Tramite i blocchi `Drop_NaN_G` e `Drop_NaN_Y`, vengono eliminate le righe contenenti valori mancanti nei campi chiave.
- **Uniformazione temporale**: Il blocco `Cambia Date` normalizza il formato delle date nei due dataset, rendendo compatibili i timestamp di pickup e dropoff, così da facilitare le operazioni di unione e aggregazione.
- **Rinomina e riordinamento**: Le trasformazioni `Select_G_2`, `Select_Y_2` e i successivi `Ordina_Campi_G` / `Ordina_Campi_Y` assicurano che le colonne dei due dataset siano ordinate allo stesso modo.
- **Concatenazione**: Una volta rese compatibili, le due sorgenti vengono unite verticalmente tramite il blocco `Concat`, ottenendo così un unico dataset integrato con le corse sia dei taxi verdi sia dei taxi gialli.
- **Correzioni temporali**: Il blocco `Correzioni_time` effettua ulteriori verifiche e modifiche sui campi temporali derivati (come anno fiscale, mese, giorno, ecc.).
- **Creazione di ulteriori campi**: Vengono create le variabili inerenti alla durata dei viaggi (`Creazione_Campi`) utilizzando gli orari di inizio e fine corsa.
- **Pulizia finale e selezione**: Alla fine di questo processo si passa ad una fase in cui vengono cancellate le istanze (`Drop_Rows_Corrotte`) che sono state elaborate in maniera corrotta (probabilmente dovuto ad una cattiva codifica iniziale).
- **Output finale**: Il dataset pronto e pulito viene infine esportato tramite il blocco `Dati Pronti`, pronto per essere caricato all'interno del *Data Warehouse*.

Variabili finali

Al termine del processo ETL, il dataset risultante presenta le seguenti colonne, organizzate per area tematica:

- **Informazioni sul fornitore e sul tipo di corsa**:
  - `VendorID`
  - `Trip_type`
  - `TaxiColor`
  - `Store_and_fwd_flag`
- **Dettagli passeggero e corsa**:
  - `Passenger_count`
  - `Trip_distance`
  - `durata_viaggio_secondi`
  - `durata_viaggio_minuti`
- **Informazioni economiche e di pagamento**:
  - `Fare_amount`
  - `Extra`
  - `MTA_tax`
  - `Tip_amount`
  - `Tolls_amount`
  - `Improvement_surcharge`
  - `Total_amount`
  - `Payment_type`
- **Informazioni temporali**:

  - Date
  - time_PU
  - time_DO
  - FiscalYear
  - FiscalQuarter
  - FiscalMonthNumber
  - DayOfWeek
  - DayName
  - FiscalMonthName
  - DayOfMonthNumber

- **Informazioni geografiche - zona di partenza (pickup)**:
  - PU_location_ID
  - PU_borough
  - PU_zone

- **Informazioni geografiche - zona di arrivo (dropoff)**:
  - DO_location_ID
  - DO_borough
  - DO_zone

2.2 Star Schema

È proprio a partire dalla struttura finale del dataset, ottenuta al termine del processo ETL, che è stato costruito lo *star schema* (schema a stella), un modello concettuale e logico particolarmente adatto ad ambienti di *Data Warehousing* e analisi OLAP. L'obiettivo è quello di centralizzare le

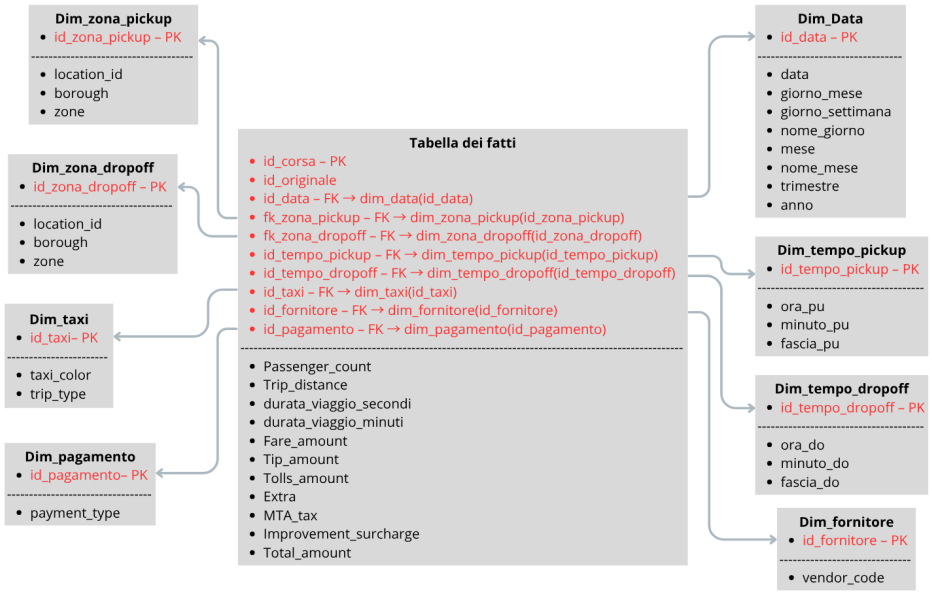
informazioni quantitative delle corse all'interno di una *fact table*, collegata a più tabelle dimensionali, ognuna delle quali descrive un aspetto specifico della corsa (tempo, spazio, pagamento, veicolo, ecc.).

La costruzione dello schema è avvenuta utilizzando PySpark, secondo i seguenti passaggi principali:

- **Lettura e preparazione iniziale**: è stato caricato il dataset integrato `dati_pronti_rip2.csv`, aggiungendo un identificativo univoco `id_originale` a ogni riga e correggendo eventuali formati errati (es. virgole decimali nella durata del viaggio).
- **Estrazione di informazioni temporali**: dalle colonne `time_PU` e `time_DO` sono state derivate ore e minuti, con relative fasce orarie (*Notte*, *Mattina*, *Pomeriggio*, *Sera*) per facilitare l'analisi temporale.
- **Costruzione delle dimensioni**:
  - **dim_data**: contiene informazioni temporali derivate dal campo `Date`, come giorno del mese, giorno della settimana, mese, trimestre e anno fiscale.
  - **dim_zona_pickup** e **dim_zona_dropoff**: rappresentano rispettivamente la zona di partenza e arrivo della corsa, costruite a partire da `PU_location_ID`, `DO_location_ID` e dai relativi borough e nomi di zona.
  - **dim_tempo_pickup** e **dim_tempo_dropoff**: includono ora, minuto e fascia oraria sia per il prelievo che per la discesa del passeggero.
  - **dim_taxi**: codifica la combinazione `TaxiColor` e `Trip_type`, identificando il tipo di servizio e il colore del veicolo.
  - **dim_pagamento**: mappa il metodo di pagamento utilizzato (es. contanti, carta, ecc.).
  - **dim_fornitore**: registra l'identificativo del fornitore di servizio taxi (`VendorID`).
- **Costruzione della *fact table***: È stata definita la tabella centrale `fatt_corsa_taxi`, contenente:
  - le **foreign key** verso tutte le dimensioni sopra elencate;

- le **misure quantitative** della corsa: `Passenger_count`, `Trip_distance`, `durata_viaggio_secondi`, `durata_viaggio_minuti`;
- le **informazioni economiche**: `Fare_amount`, `Tip_amount`, `Tolls_amount`, `Extra`, `MTA_tax`, `Improvement_surcharge`, `Total_amount`.

- **Esportazione dei dati**: tutte le tabelle (dimensioni e fatti) sono state salvate in formato `Parquet`, così da garantirne una lettura efficiente e un'ottima integrazione con strumenti di analisi distribuita.



3. Query e Analisi

In questo capitolo vengono descritte le query OLAP fatte per ricavare informazioni di sintesi. Le tabelle coinvolte fanno parte dello schema a stella e includono la fact table `fatt_corsa_taxi` e le dimensioni `dim_data`, `dim_zona_pickup`, `dim_zona_dropoff`, `dim_taxi`, `dim_pagamento`.

A fine didattico e di test, sono stati utilizzati 2 framework: **Apache Spark** e **Polars**.

3.1 Apache Spark con dati Parquet

Il codice è stato scritto con pyspark su un cluster di 5 macchine virtuali con le seguenti specifiche:

- Sistema poerativo: **Linux**
- RAM: **16 Gb per macchina**
- vCPU: **4 virtual CPU per macchina**
- accesso: **via SSH**

Una delle 5 macchine è stata inizializzata come nodo master, mentre le altre sono state configurate come nodi worker. Fatto ciò, si è proceduto all'esecuzione delle query incrementando progressivamente il numero di worker con l'obiettivo di valutare l'impatto della parallelizzazione sull'efficienza dell'elaborazione distribuita.

Query 1: Numero di corse per mese

```
SELECT d.nome_mese, COUNT(*) AS numero_corse
FROM fatt f
JOIN dim_data d ON f.id_data = d.id_data
GROUP BY d.nome_mese
```

Questa query calcola il numero totale di corse effettuate per ciascun mese. L'obiettivo è analizzare la stagionalità del servizio taxi, evidenziando eventuali picchi nei volumi di corsa.

Query 2: Importo medio per zona di partenza

```
SELECT z.zone, AVG(f.total_amount) AS media_importo
FROM fatt f
JOIN dim_zona_pickup z ON f.fk_zona_pickup = z.id_zona_pickup
GROUP BY z.zone
```

Restituisce l'importo medio delle corse in base alla zona di pick-up. Permette di individuare aree geografiche caratterizzate da corse mediamente più costose, utile per decisioni su pricing dinamico o allocazione dei mezzi.

Query 3: Durata media per tipo di taxi

```
SELECT t.taxi_color, AVG(f.durata_viaggio_minuti) AS media_durata
FROM fatt f
JOIN dim_taxi t ON f.id_taxi = t.id_taxi
GROUP BY t.taxi_color
```

Confronta la durata media dei viaggi in funzione del tipo di taxi (giallo o verde), evidenziando eventuali differenze operative o territoriali tra i due servizi.

Query 4: Mancia media per metodo di pagamento

```
SELECT p.payment_type, AVG(f.tip_amount) AS media_mancia
FROM fatt f
JOIN dim_pagamento p ON f.id_pagamento = p.id_pagamento
GROUP BY p.payment_type
```

Analizza la mancia media ricevuta per ciascun metodo di pagamento, utile per individuare tendenze comportamentali dei clienti e la loro propensione alla generosità in base alla modalità di pagamento.

Query 5: Numero di corse per borough di destinazione

```
SELECT z.borough, COUNT(*) AS numero_corse
FROM fatt f
JOIN dim_zona_dropoff z ON f.fk_zona_dropoff = z.id_zona_dropoff
GROUP BY z.borough
```

Conta il numero di corse terminate in ciascun borough di New York. Questa analisi consente di identificare le zone a maggiore domanda di drop-off.

Risultati

Workers	Q1	Q2	Q3	Q4	Q5	Totale
1	5.02	2.16	1.38	1.62	1.18	11.36
2	4.87	1.96	1.37	1.23	1.28	10.71
3	4.31	2.18	1.40	1.22	1.07	10.18
4	4.58	2.02	1.71	1.64	1.12	11.07

Tabella 3.1: Tempi di esecuzione per ciascuna query (Q1–Q5) su Dataset 1

Workers	Q1	Q2	Q3	Q4	Q5	Totale
1	5.93	3.49	2.85	2.18	2.30	16.75
2	8.62	3.27	2.59	2.26	1.77	18.51
3	5.88	3.12	2.11	1.96	1.68	14.75
4	5.84	3.36	2.17	1.85	1.72	14.94

Tabella 3.2: Tempi di esecuzione per ciascuna query (Q1–Q5) su Dataset 2

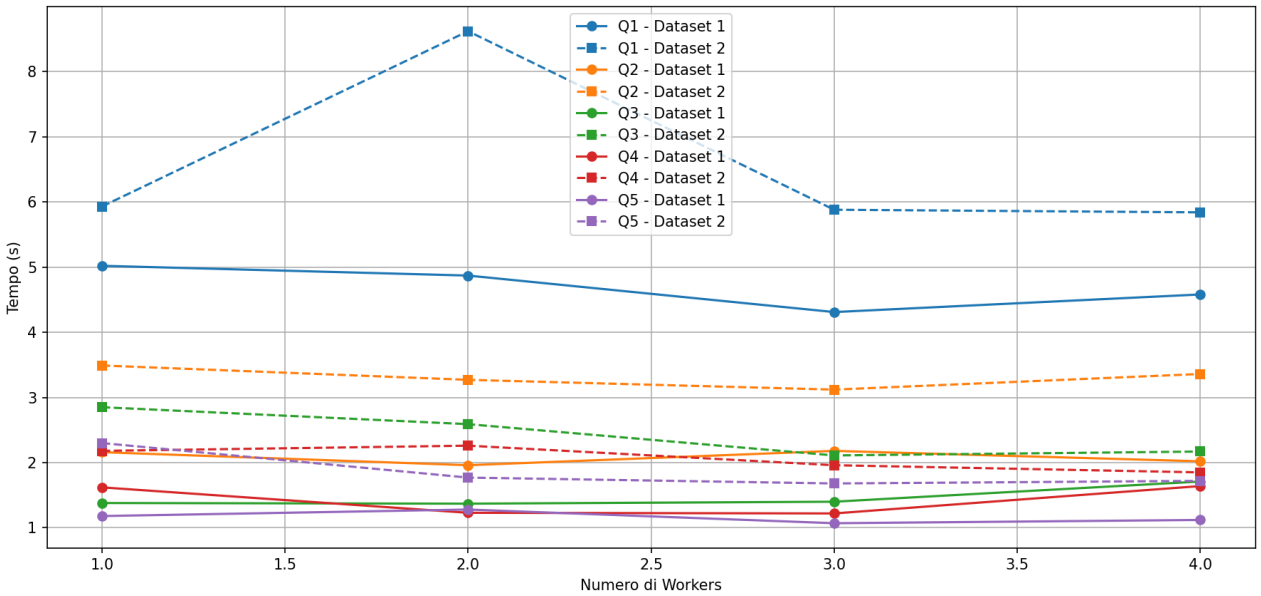


Figura 3.1: Benchmark Spark – tempi per dataset 1 e 2

Come si può notare, la configurazione con 3 worker è quasi sempre la più veloce. Lo dimostra anche la somma dei tempi per eseguire tutte le 5 query.

3.2 Polars con dati Feather

Si è deciso di testare anche un formato alternativo, ovvero i file **Feather**, elaborati tramite la libreria `Polars`, che sfrutta il formato Apache Arrow per garantire lettura e scrittura efficienti in RAM.

Risultati

Query	Dataset Piccolo	Dataset Grande
Q1	0.3232	0.8289
Q2	0.2462	0.6384
Q3	0.1935	0.5098
Q4	0.1961	0.4978
Q5	0.2043	0.5795
Totale	**1.1633**	**3.0544**

Tabella 3.3: Tempi di esecuzione su dataset in formato Feather

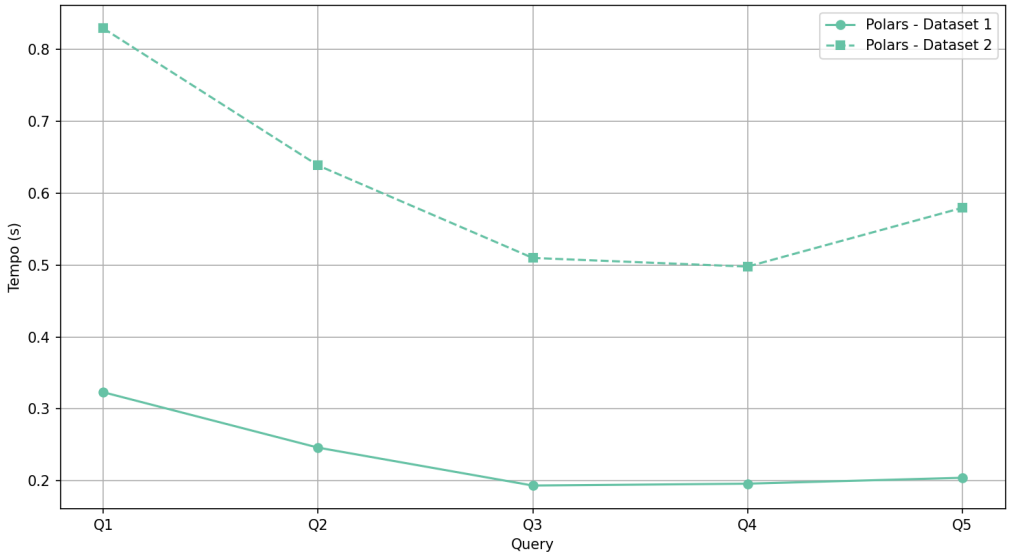


Figura 3.2: Benchmark Polars – prestazioni con Feather

Come evidenziato dai risultati, Polars mostra performance notevolmente superiori rispetto a Spark su dataset di dimensioni ridotte. Tuttavia, presenta un limite strutturale: l'impossibilità di eseguire query in ambiente distribuito. All'aumentare del volume dei dati, si nota un incremento significativo dei tempi (da circa 0.2 a 0.5 secondi per la query 3), sintomo di un peggioramento dello *scalability factor*.

3.3 Query 6: Caso complesso

Per concludere, è stata eseguita una query di tipo complesso con molteplici JOIN tra dimensioni, un filtro, un'aggregazione e un ordinamento, al fine di stressare entrambi gli strumenti.

```
SELECT
    d.nome_mese,
    t.taxi_color,
    p.payment_type,
    z.zone,
    COUNT(*) AS numero_corse,
    AVG(f.Total_amount) AS avg_importo
FROM fatt f
JOIN dim_data d ON f.id_data = d.id_data
JOIN dim_taxi t ON f.id_taxi = t.id_taxi
JOIN dim_pagamento p ON f.id_pagamento = p.id_pagamento
JOIN dim_zona_pickup z ON f.fk_zona_pickup = z.id_zona_pickup
WHERE f.durata_viaggio_minuti > 0
GROUP BY d.nome_mese, t.taxi_color, p.payment_type, z.zone
ORDER BY d.nome_mese
```

Lo scopo di questa query è proprio quello di andare a testare i due framework per vedere quando effettivamente (oppure se prima o poi) spark *"vince"* su polars in termini di performance e tempistiche (considerando che con spark si potrebbe distribuire all'infinito).

Risultati

Workers	Dataset 1	Dataset 2
1	8.94	11.84
2	6.58	9.79
3	6.45	8.84
4	7.02	11.01

Tabella 3.4: Tempi della query complessa con Apache Spark

Dataset	Tempo (s)
Dataset 1	0.7553
Dataset 2	2.0468

Tabella 3.5: Tempi della query complessa con Polars

Anche in questo caso Polars si dimostra più rapido su volumi contenuti, mentre Spark mantiene prestazioni più stabili all'aumentare dei dati, grazie alla sua natura distribuita.

Conclusioni

In questo paragrafo vengono riepilogati i principali risultati, le osservazioni sperimentali e le considerazioni conclusive.

Confronto tra le configurazioni dei worker Spark

Dai risultati ottenuti nei benchmark effettuati con Apache Spark, emerge che l'utilizzo di tre worker rappresenta il miglior compromesso in termini di performance medie. Infatti, nel caso del Dataset 1 (più piccolo), la configurazione con 3 worker ha fatto registrare il tempo complessivo più basso (10.18 secondi), mentre per il Dataset 2 (più grande), il miglior risultato è stato ottenuto sempre con 3 worker (14.75 secondi).

È interessante notare che l'aumento del numero di worker non comporta sempre un miglioramento lineare delle prestazioni: con 4 worker, le performance tendono in alcuni casi a peggiorare leggermente, probabilmente a causa dell'overhead di comunicazione e gestione dei task distribuiti.

Perché Polars può battere Spark: vantaggi e limiti

Polars, grazie alla sua architettura basata su *lazy evaluation* e all'utilizzo del formato Arrow in memoria (Feather), ha dimostrato prestazioni nettamente superiori su dataset di dimensioni contenute. I vantaggi principali di Polars sono:

- **Velocità di esecuzione** su operazioni di aggregazione e join non distribuite.
- **Efficienza in memoria** e gestione ottimale delle strutture dati columnar.
- **Facilità d'uso** in ambienti singola macchina o prototipali.

Tuttavia, questi vantaggi vanno bilanciati con alcune limitazioni strutturali:

- **Assenza di parallelismo distribuito**: Polars non può sfruttare cluster multi-nodo.

- **Scalabilità limitata**: all'aumentare dei dati in memoria, le performance si degradano rapidamente.
- **Minor ecosistema**: rispetto a Spark, Polars è meno integrato in pipeline enterprise su larga scala.

Andamento delle performance al crescere dei dati

Come mostrato nei grafici seguenti, Polars mostra un incremento significativo dei tempi di esecuzione quando si passa dal dataset più piccolo (4 milioni di righe) a quello più grande (12 milioni di righe). Questo comportamento evidenzia come l'efficienza di Polars sia strettamente legata alla dimensione del dataset trattabile in memoria singola.

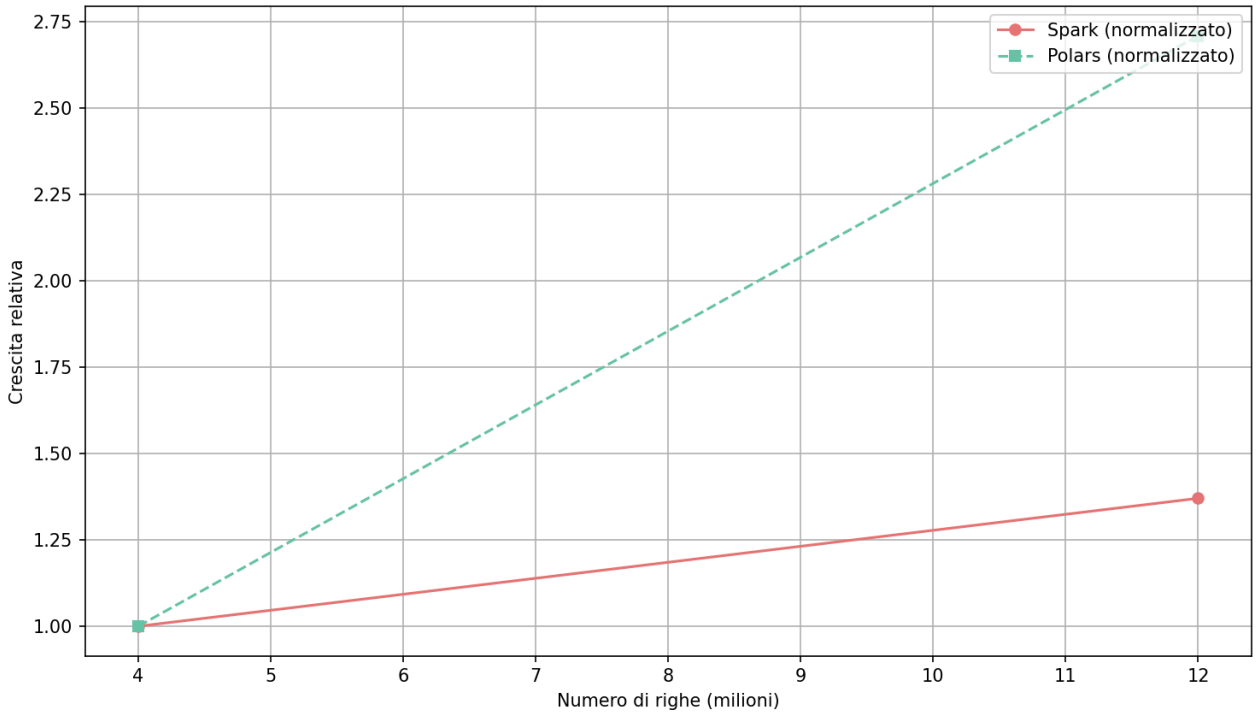


Figura 3.3: Crescita relativa dei tempi – Polars vs Spark

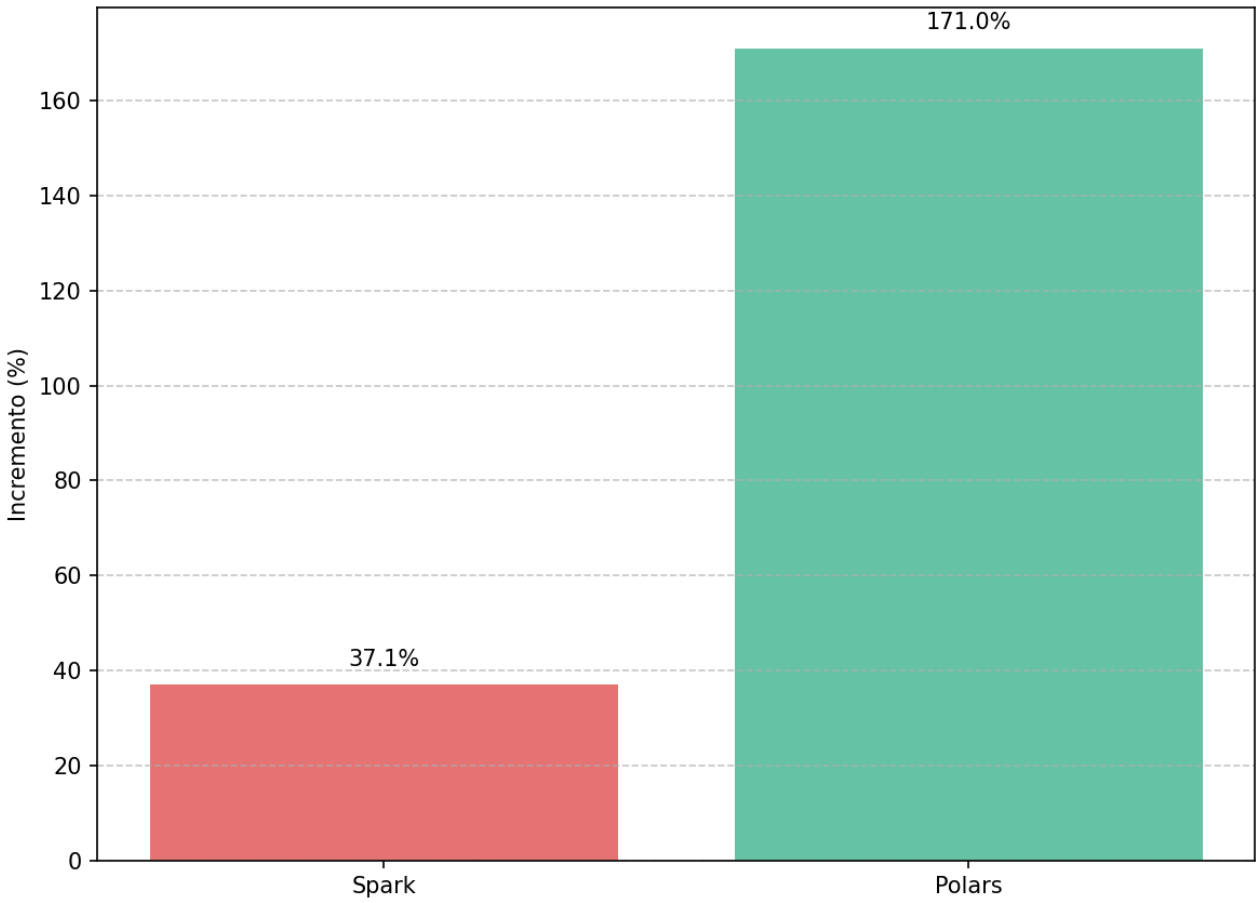


Figura 3.4: Crescita percentuale dei tempi – Polars vs Spark

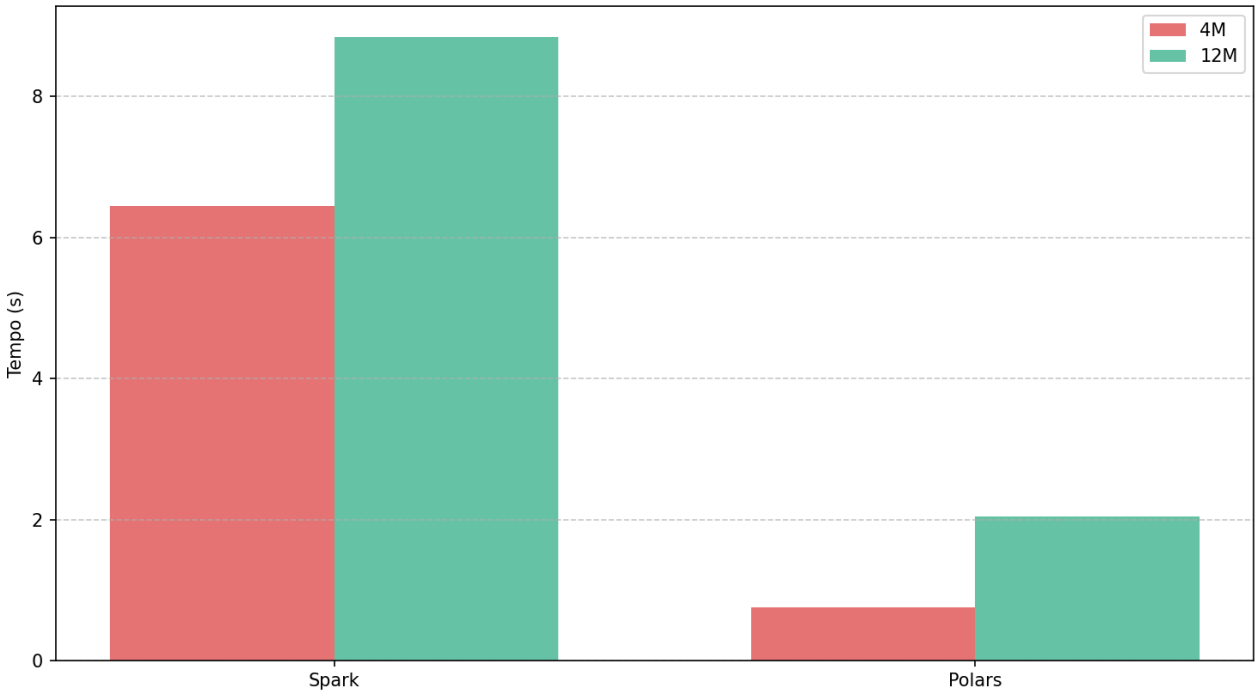


Figura 3.5: Confronto diretto dei tempi – Polars e Spark

Nei grafici si nota chiaramente come Spark mantenga tempi di esecuzione più stabili e gestisca meglio la crescita dei dati, grazie alla distribuzione del carico di lavoro su più nodi.

Conclusione generale

In sintesi, Polars rappresenta un'ottima scelta per analisi veloci su dataset medio-piccoli, sfruttando al massimo le prestazioni offerte da elaborazioni in memoria. Tuttavia, per carichi di lavoro su larga scala e necessità di scalabilità orizzontale, Apache Spark rimane la soluzione più robusta ed efficiente.