New Generation Data Models and DBMSs

(a.a. 2024/25)

Creazione di un database NoSQL per il rilevamento delle frodi con carte di credito

A cura di

Nicolò Barbieri - 33244A
nicolo.barbieri2@studenti.unimi.it

Silvia Colombo - 45972A
silvia.colombo32@studenti.unimi.it

INDICE

1. Introduzione
2. Generazione dei dati
3. Modello concettuale
    - 3.1 UML Class Diagram
    - 3.2 Descrizione entità e classi
    - 3.3 Assunzioni e Constraints
4. Modello logico NoSQL
    - 4.1 Modello logico NoSQL e workload
    - 4.2 Schema MongoDB
5. Implementazione
    - 5.1 Creazione del dataset
        - *5.1.1 QUERY 1*
        - *5.1.2 QUERY 2*
        - *5.1.3 QUERY 3*
    - 5.2 Estensione del modello dati
        - *5.2.1 QUERY 4*
6. Valutazione delle prestazioni
    - 6.1 Tempo di esecuzione query
    - 6.2 Valutazione applicazione design pattern

1. Introduzione

Le frodi sulle carte di pagamento rappresentano una sfida importante per i titolari di aziende, gli emittenti di carte di pagamento e i fornitori di servizi di pagamento, e ogni anno causano perdite finanziarie ingenti e sempre più crescenti.

Ad oggi, esistono diversi approcci di Machine Learning (ML) che mirano ad automatizzare il processo di identificazione di pattern fraudolenti in una grande mole di dati.

L'obiettivo di questo progetto non è l'applicazione di un modello di ML ma progettare e implementare un database NoSQL in grado di gestire ed elaborare grandi volumi di dati transazionali, generati sfruttando il codice fornito nel Fraud Detection Handbook.

Il punto di partenza, fornito dal simulatore, sono 3 tabelle:

- Customer profile
- Terminal profile
- Transactions

Rappresentano le caratteristiche più essenziali per rappresentare una transazione con carta di pagamento e permetteranno di strutturare al meglio il database.

Al fine di raggiungere gli obiettivi richiesti, le attività principali del progetto sono:

- Generazione di tre dataset di dimensioni crescenti (50Mbyte, 100Mbyte, 200Mbyte).

- Definizione del modello concettuale tramite UML class diagram sulla base delle informazioni fornite dal simulatore.

- Progettazione del modello logico in un sistema NoSQL: fornendo una modellazione dei dati al fine di ottimizzare l'esecuzione delle query richieste.

- Estensione dello schema con nuove informazioni (periodo della giornata, categoria del prodotto, percezione di sicurezza, relazioni "buying_friends").

- Implementazione delle query richieste.

- Analisi delle prestazioni e tempi di esecuzione sui tre dataset.

- Valutazione dell'applicazione di design pattern.

2. Generazione dataset

Il dataset di partenza è stato realizzato tramite uno script Python a partire dal codice del *Fraud Detection Handbook*, utilizzando librerie come numpy, pandas e pandarallel per il calcolo parallelo.

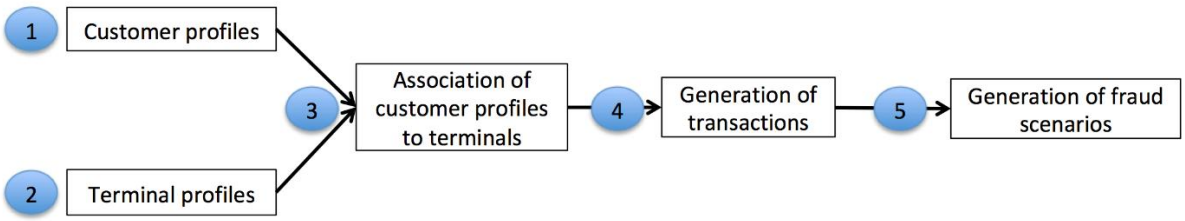
Il file *Script/Generate_dataset.py*, contenente lo script per la generazione del dataset, riceve da linea di comando la dimensione in MByte del dataset da generare.

2.1 Struttura generale dello script

Lo script è organizzato in più funzioni:

1. *generate_customer_profiles_table()* → genera la tabella dei clienti.
2. *generate_terminal_profiles_table()* → genera la tabella dei terminali.
3. *get_list_terminals_within_radius()* → associa a ogni cliente un insieme di terminali vicini in base alle coordinate.
4. *generate_transactions_table()* → genera la sequenza temporale delle transazioni di un cliente, in base a frequenze, importi di spesa e terminali disponibili.
5. *add_frauds()* → marca una parte delle transazioni come fraudolente.
6. *export_dataset()* → salva i dataset in formato .pkl e .json.

Di seguito è illustrato il processo di generazione delle transazioni:



2.2 Parametri di simulazione

I parametri principali impostati nello script sono:

- **Numero di clienti (*'CUSTOMERS_NUMBER'*)**: 2500
- **Numero di terminali (*'TERMINALS_NUMBER'*)**: 800
- **Raggio massimo cliente–terminale (*'r'*)**: 20 unità
- **Periodo di simulazione iniziale**: 90 giorni
- **Data di inizio**: 01/01/2025
- **Dimensione target**: passata da linea di comando (50MB, 100MB, 200MB)

2.3 Customer profile table (*'generate_customer_profiles_table()'*)

Ogni cliente è identificato da un id univoco e caratterizzato da:

- Coordinate geografiche (x, y).
- Importo medio delle transazioni (mean_amount).
- Deviazione standard dell'importo (std_amount).
- Numero medio di transazioni per giorno (mean_nb_tx_per_day).

La funzione richiede input il numero di clienti ('CUSTOMER_NUMBER') per cui generare un profilo.

Esempio di record Customer:

```
{
 "CUSTOMER_ID": 12,
 "x_customer_id": -0.421,
 "y_customer_id": 1.203,
 "mean_amount": 42.5,
 "std_amount": 10.3,
 "mean_nb_tx_per_day": 2.1,
}
```

Lo script ci permette di ottenere una tabella Pandas (Dataframe) contenente tutti i profili dei clienti, utilizzata poi nelle fasi successive di generazione del dataset: associazione ai terminali e generazione delle transazioni.

2.4 Terminal profile table (‘*generate_terminals_profiles_table()’*)

Ogni terminale è identificato da un id univoco e riporta una coppia di coordinate geografiche. La funzione richiede come input un numero di terminali per cui generare un profilo.

Esempio di record Terminal:


```
{
  "TERMINAL_ID": 48,
  "x_terminal_id": -0.317,
  "y_terminal_id": 1.146
}
```


Lo script ci permette di ottenere una tabella Pandas (Dataframe) contenente tutti i terminali, utilizzata poi nelle fasi successive di generazione del dataset. La tabella riporterà un id univoco del terminale e una posizione casuale nello spazio.

2.5 Associazioni Customer-Terminal (‘*get_list_terminals_within_radius()’*)

Lo script ci permette di associare a ciascun Customer la lista di Terminal che si trovano entro un raggio r dalla loro posizione geografica. All'aumentare del numero di terminali, il raggio deve essere adattato in modo da corrispondere al numero medio di terminali disponibili per cliente desiderato.

Per il calcolo dei terminali disponibili utilizza la funzione *apply* di Pandas, memorizzando i risultati in una nuova colonna *‘available_terminals’* nella tabella dei profili cliente.

I clienti potranno effettuare transazioni solo sui terminali a loro associati.

Esempio di tabella:

	CUSTOMER_ID	x_customer_id	y_customer_id	mean_amount	std_amount	mean_nb_tx_per_day	available_terminals
0	0	54.881350	71.518937	62.262521	31.131260	2.179533	[0, 1, 2, 3]
1	1	42.365480	64.589411	46.570785	23.285393	3.567092	[0, 1, 2, 3]
2	2	96.366276	38.344152	80.213879	40.106939	2.115580	[1, 4]
3	3	56.804456	92.559664	11.748426	5.874213	0.348517	[0, 1, 2, 3]
4	4	2.021840	83.261985	78.924891	39.462446	3.480049	[2, 3]

2.6 Transactions table (‘*generate_transactions_table()’*)

La tabella delle transazioni è la più estesa ed è generata a partire dalle abitudini di ogni cliente.

La funzione richiede in input un profilo cliente, una data di inizio e un numero di giorni per i quali generare le transazioni.

Ogni transazione contiene:

- Identificativo della transazione (TRANSACTION_ID).
- Cliente (CUSTOMER_ID).
- Terminale utilizzato (TERMINAL_ID).
- Importo (TX_AMOUNT).
- Data e ora (TX_DATETIME).
- Indicatore di frode (TX_FRAUD).

Le transazioni vengono ordinate cronologicamente e indicizzate progressivamente.

Lo script ci permette di ottenere una tabella Pandas (Dataframe) delle transazioni.

Esempio:


```
{
 "TRANSACTION_ID": 100321,
 "CUSTOMER_ID": 12,
 "TERMINAL_ID": 48,
 "TX_DATETIME": "2025-03-15T14:25:00",
 "TX_AMOUNT": 52.3,
 "TX_FRAUD": false
}
```

2.7 Introduzione delle frodi (‘*add_frauds()’)*

Per rendere il dataset più realistico e adatto ad attività di analisi e rilevamento, è stata implementata una fase aggiuntiva di iniezione di frodi simulate.
Tutte le transazioni inizialmente sono considerate genuine (TX_FRAUD = False).
Successivamente, tramite la funzione *add_frauds*, vengono aggiunti tre diversi scenari di frode, ciascuno con caratteristiche specifiche.

Scenario 1 – Importo elevato

Tutte le transazioni con importo superiore a 220 vengono automaticamente marcate come fraudolente.

Scenario 2 – Terminali compromessi

Ogni giorno vengono selezionati due terminali casuali. Tutte le transazioni effettuate su questi terminali nei 28 giorni successivi vengono considerate fraudolente.

Questo scenario simula il caso in cui un POS o terminale venga compromesso (ad esempio tramite phishing o malware), consentendo a un malintenzionato di catturare i dati delle carte utilizzate.

Scenario 3 – Clienti compromessi

Ogni giorno vengono selezionati tre clienti casuali. Per i 14 giorni successivi, circa un terzo delle loro transazioni viene marcato come fraudolento: in questi casi, l’importo originale viene moltiplicato per 5.

Questo scenario simula il furto di credenziali di pagamento (ad esempio nel caso di frodi online “card-not-present”), dove il cliente continua ad effettuare acquisti normali, ma allo stesso tempo un truffatore sfrutta i suoi dati per transazioni di importo anomalo.

Al termine di questa fase, il dataset contiene:

- transazioni genuine (TX_FRAUD= false)
- transazioni fraudolente (TX_FRAUD=true; TX_FRAUD_SCENARIO = 1, 2, 3)

In questo modo il dataset risulta più vario e realistico, permettendo di simulare sia frodi banali facilmente individuabili, sia frodi sofisticate e temporanee, più difficili da rilevare.

All’interno del progetto, la funzione è stata implementata ma non invocata, per non alterare i risultati delle operazioni successive.

2.8 Controllo della dimensione

Uno degli aspetti più importanti è il **controllo della dimensione finale del dataset**. Poiché la memoria occupata dipende dal numero di transazioni, lo script esegue un ciclo in cui aumenta progressivamente il numero di giorni simulati fino a raggiungere la dimensione desiderata (50MB, 100MB, 200MB).

Il calcolo è effettuato tramite sys.getsizeof() sui DataFrame Pandas.

2.9 Formati di output

I dataset vengono esportati in due formati:

- .pkl (Pickle), per uso interno in Python.
- .json, per importazione diretta in MongoDB tramite mongoimport.

La scelta del formato JSON è fondamentale perché consente di mantenere la compatibilità con il modello document-based ed una maggiore interoperabilità.

3. Modello Concettuale

3.1 UML Class Diagram

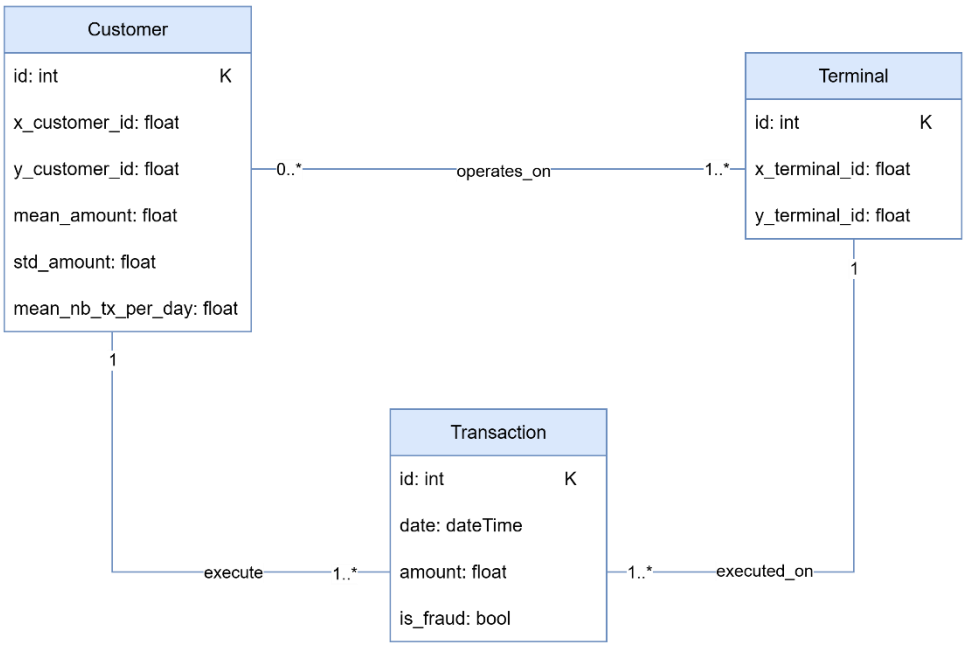
La Figura 1 rappresenta l'UML Class Diagram realizzato per il dominio proposto, partendo dalle informazioni fornite dal simulatore *Fraud Detection Handbook*.

La rappresentazione grafica riporta tre classi principali:

- **Customer**: rappresenta l'utente che effettua pagamenti elettronici/transazioni.
- **Terminal**: rappresenta il punto di pagamento (ATM, POS, ecc.).
- **Transaction**: rappresenta l'operazione eseguita da un Cliente sul Terminale.

Tra le classi sono state rilevati i seguenti vincoli di cardinalità:

- **Relazione Customer–Terminal (*'operates_on'*)**
  Un cliente (Customer) può operare su più terminali (Terminal) e un terminale può essere utilizzato da più clienti. La relazione tra Customer e Terminal è di tipo molti-a-molti (N:N).
- **Relazione Customer–Transaction (*'execute'*)**
  Un cliente può eseguire più transazioni mentre ogni transazione può essere associata a uno e un solo cliente. La relazione tra Customer e Transaction è di tipo uno-a-molti (1:N).
- **Relazione Transaction- Terminal (*'executed_on'*)**
  Una transazione viene eseguita su un solo terminale ma ciascun terminale può essere utilizzato per eseguire più transazioni. La relazione tra Transaction e Terminal è di tipo uno-a-molti (1:N).



3.2 Descrizione entità e classi

Più nello specifico, le tre classi individuate, riportano le seguenti informazioni di dettaglio.

Customer

- **id (int)**: identificatore univoco del cliente.
- **x_customer_id (float)**, **y_customer_id (float)**: coppia di coordinate che definisce la posizione geografica del cliente.
- **mean_amount (float)**: importo medio delle transazioni effettuate.
- **std_amount (float)**: deviazione standard degli importi delle transazioni.
- **mean_nb_tx_per_day (float)**: numero medio di transazioni giornaliere effettuate dal cliente.

Terminal

- **id (int)**: identificatore univoco del terminale.
- **x_terminal_id (float)**, **y_terminal_id (float)**: coppia di coordinate che definisce la posizione geografica del terminale.

Transaction

- **id (int)**: identificatore univoco della transazione.
- **date (dateTime)**: data e ora in cui la transazione è stata effettuata.
- **amount (float)**: importo della transazione.
- **is_fraud (bool)**: indicatore booleano che segnala se la transazione è fraudolenta (1) oppure no (0).

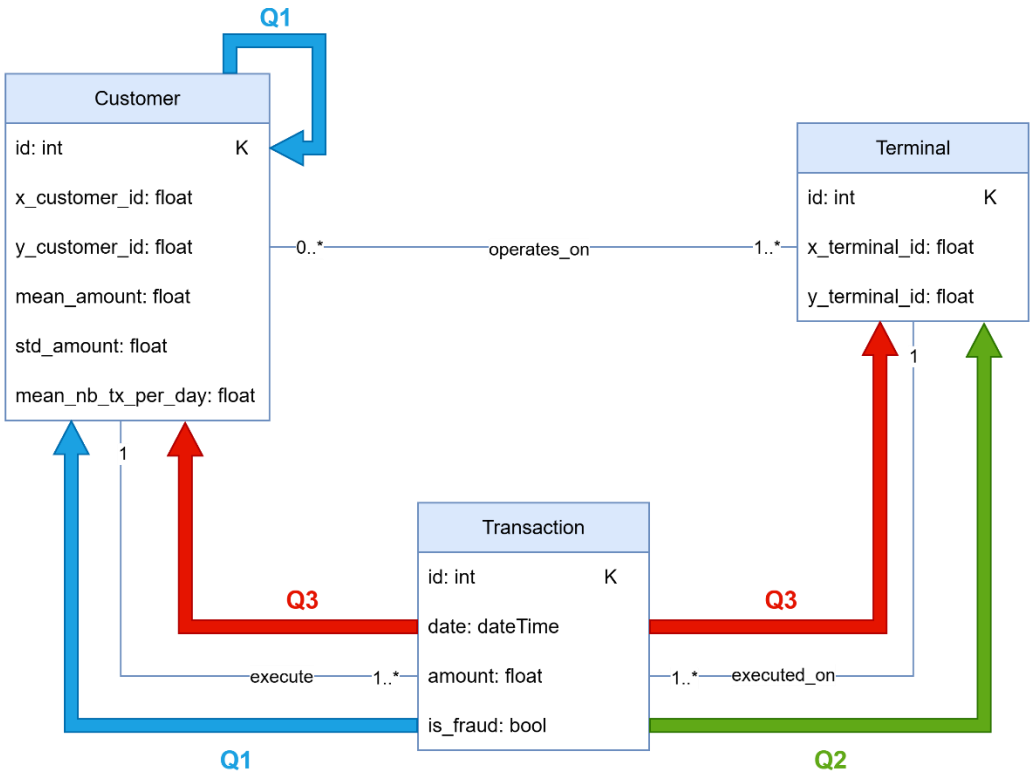
3.3 Assunzioni e Constraints

- Timestamp delle transazioni espresso con il formato 24 ore.
- Attributo *Fraud* di tipo booleano (true/false).
- Localizzazione delle transazioni tramite coordinate cartesiane.

4. Modello Logico NoSQL

4.1 Modello Logico NoSQL e workload

Tenendo conto del workload fornito dal testo del progetto, si identificano i modelli di navigazione riportati nella figura sottostante.



Q1: La query richiede di identificare, per ciascun cliente X, il cliente Y (o i clienti) che condivide almeno 3 terminali e ha l'importo di spesa inferiore del 10% rispetto a X. Per ottenere questo risultato, si parte dalla classe Transaction, raggruppando per *customer_id* e raccogliendo i terminali utilizzati e l'importo di spesa complessivo. Attraverso la relazione '*execute*', definita tra la classe Transaction e la classe Customer, si associa a ciascun cliente le informazioni aggregate. Infine, il confronto tra clienti avviene mettendo in relazione coppie di Customer sulla base dei terminali condivisi e della spesa media calcolata.

Q2: La query richiede di identificare, per ciascun terminale, le possibili transazioni fraudolente relative al mese corrente. Partendo dalla classe Transaction, si estraggono la data (*tx_data_time*) e l'importo della transazione (*tx_amount*). Attraverso la relazione '*executed_on*', definita tra la classe Transaction e la classe Terminal, ogni transazione viene associata al relativo Terminal, sul quale si calcola l'importo medio delle transazioni del mese precedente. Successivamente, per ogni transazione del mese corrente, si confronta l'importo con la soglia definita, individuando in questo modo le transazioni fraudolente.

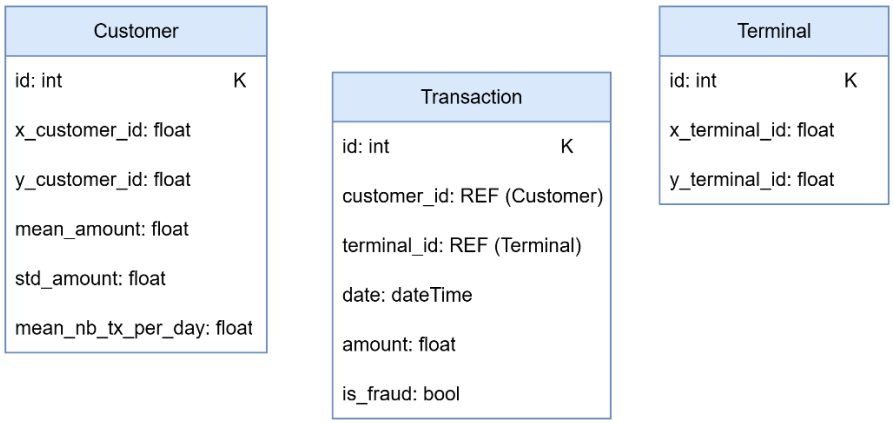
Q3: La query richiede di identificare i co-customers, a partire da un customer iniziale, fino al grado k=3 (la relazione di co-customing è modellata così: “u1-t1-u2", dove t1 è un terminale comune su cui entrambi gli utenti u1 e u2 hanno eseguito almeno una transazione). Partendo dalla classe Transaction, andiamo ad associare a ciascuna transazione il customer che l’ha svolta ed il terminale su cui è stata svolta mediante le relazioni *‘execute’* ed *‘executed_on’*. Una volta fatto ciò, procediamo a raggruppare, per ogni terminale, gli utenti che hanno svolto almeno una transazione su di esso, ed infine proiettiamo per costruire le coppie “ui-tj-uk" che definiranno una struttura a grafo. Infine, procediamo a navigare tale grafo per estrarre tutti gli utenti a distanza 3 dall’utente iniziale.

4.2 Schema MongoDB

Osservando il workload, si nota che non emergono conflitti tra le operazioni eseguite per la risoluzione delle query richieste.

Considerando inoltre le cardinalità delle relazioni ‘*execute*’ ed *‘executed_on’* rispettivamente tra Transaction-Customer e Transaction-Terminal e valutando l’elevato numero di elementi di tipo Transaction, risulta più efficiente implementare tali relazioni tramite riferimenti all’interno della collezione Transaction.

Attuando le osservazioni fatte durante la creazione del modello logico, la figura seguente mostra le collezioni generate nel database.



Si evidenzia che la relazione ‘*operates_on*’, definita nel modello concettuale tra la classe Customer e la classe Terminal, è stata modellata in maniera indiretta passando dalla collezione Transaction. Infatti, per risalire ai terminali su cui ha operato uno specifico utente, è necessario passare attraverso le transazioni da lui eseguite. Questa scelta è stata adottata poiché, analizzando il workload, non emerge la necessità di utilizzare direttamente tale relazione per la risoluzione delle query, rendendo quindi superflua una sua rappresentazione esplicita all’interno del modello logico.

5. Implementazione

5.1 Creazione del dataset

La fase di importazione e creazione dei dataset in MongoDB è gestita dallo script Script/Connector.py.

Più nello specifico, le funzionalità implementate riguardano:

- **Connessione al database**
  Utilizza pymongo per connettersi all'istanza locale di MongoDB (mongodb://localhost:27017/). Il database utilizzato è 'ProjectDB'.
- **Conversione JSON → Phyton dict :**
  La funzione *convert_from_json_to_dict(filepath)* permette di leggere un file JSON in un DataFrame Pandas e convertirlo in una lista di dizionari compatibile con l'inserimento in MongoDB.
  Questo approccio garantisce una maggiore efficienza e robustezza nel parsing dei dati.
- **Importazione del dataset:**
  La funzione *import_datasets(dim_mb)* carica i dati dalle tre tabelle generate (customers_table_XMB.json, terminals_table_XMB.json, transactions_table_XMB.json) e le importa in MongoDB come tre collezioni distinte.
  L'argomento dim_mb rappresenta la dimensione del dataset (50MB, 100MB, 200MB).
- **Reset del database:**
  La funzione *delete_datasets()* elimina, se già presenti, le tre collezioni (customers, terminals e transactions) così da consentire un nuovo caricamento privo di duplicati in caso di necessità.

5.1.1 QUERY 1

"For each customer X, identify the customer Y (or the costumers) that share at least 3 terminals in which Y executes transactions and the spending amount of Y differs less than the 10% with respect to that of X. Return the name of X, the spending amount of X, the spending amount of the related costumer Y and the spending amount of Y."

Per rispondere alla Query 1, richiesta dal progetto e sopra riportata, l'approccio utilizzato prevede i seguenti step:

1. Creazione del profilo cliente: crea una nuova collezione temporanea, definendo la spesa totale del cliente e l'elenco dei terminali utilizzati.

Più nello specifico, a partire dalle **transazioni**:

- Raggruppa per cliente (CUSTOMER_ID).
- Calcola la spesa totale del cliente ($sum di TX_AMOUNT).
- Costruisce la lista dei terminali utilizzati ($addToSet), eliminando i duplicati.
- Collega i dati ottenuti con la collezione ***customers*** ($lookup) e tramite una proiezione mantiene id, spesa totale e lista dei terminali.
- Salva il risultato in una nuova collezione temporanea '*customer_profiles'*.

2. Confronto tra clienti: definendo le correlazioni tra cliente X e clienti Y sulla base del numero di terminali condivisi e della spesa effettuata.

- Esegue un self-join ($lookup) sulla collezione 'customer_profiles'. In questo modo ogni cliente X viene confrontato con tutti gli altri clienti Y della collezione – escluso se stesso.
- Calcola:
  shared_terminals = elenco di terminali comuni tra X e Y ($setIntersection)
  shared_terminal_count = numero di terminali comuni ($size)
  spending_diff_ratio = differenza relativa tra spesa di X e spesa di Y
- Filtra i candidati Y eseguendo il $match sui vincoli richiesti:
  - Almeno 3 terminali in comune "shared_terminal_count": { "$gte": 3 }
  - Differenza di spesa tra X e Y è ≤ 10% "spending_diff_ratio": { "$lte": 0.10 }
- Ritorna le coppie X e Y, che rispecchiano i vincoli richiesti, e i rispettivi valori di spesa.

5.1.2 QUERY 2

"For each terminal identify the possible fraudulent transactions of the current month. The fraudulent transactions are those whose import is higher than 20% of the average import of the transactions executed on the same terminal in the previous month."

Per la risoluzione della Query 2, gli step necessari sono stati:

1. Definire il contesto temporale, individuando mese e anno correnti insieme al mese precedente. Tale passaggio si è reso necessario per poter filtrare correttamente le transazioni e calcolare le medie mensili sui periodi di interesse.

2. Calcolare la media per terminale: definisce una pipeline per calcolare la media per terminale del mese precedente.

Più nello specifico, a partire dalle **transazioni**:

- Estrae mese e anno dalla data della transazione
- Filtra solo le transazioni del mese precedente, come richiesto
- Raggruppa per terminale (TERMINAL_ID) e ne calcola la media degli importi ($avg di TX_AMOUNT).
- Salva il risultato in una nuova collezione temporanea '*avg_amount_prev_month*'.

3. Individuare le transazioni fraudolente

A partire dalle **transazioni** del mese corrente, esegue una lookup con le transazioni della collezione temporanea, ottenendo per ogni transazione le media del terminale relativo.

Considerata come fraudolenta una transazione con importo superiore del 20% rispetto alla media delle transazioni del terminale (nel mese precedente): definisce una **soglia di tolleranza** per individuare le transazioni sospette, che rappresenta il valore delle transazioni del mese precedente aumentato del 20%.

```
{
  "$addFields": {
    "threshold": { "$multiply": ["$avg_data.avg_prev_month", 1.2] }
}.
```

L'interrogazione, tramite un'operazione di $match, terrà conto solo delle transazioni che riportano una spesa superiore rispetto al threshold definito. Tali transazioni rappresentano le possibili operazioni fraudolente nel mese corrente.

5.1.3 QUERY 3

"Given a user u, determine the "co-customer-relationships CC of degree k". A user u' is a co-customer of u if you can determine a chain "u_1-t_1-u_2-t_2-...t_{k-1}-u_k " such that u_1=u, u_k=u', and for each 1<=l,j<=k, u_i <> u_j, and t_1,..t_{k-1} are the terminals on which a transaction has been executed. Therefore, CC_k (u)={u'| a chain exists between u and u' of degree k}."

Please, note that depending on the adopted model, the computation of CC_k(u) could be quite complicated. Consider therefore at least the computation of CC_3 (u) (i.e. the co-costumer relationships of degree 3).

Per la risoluzione della Query 3 è stata definita la funzione get_cck, che richiede in input l'identificativo di un cliente di partenza (u) e ne calcola le relazioni co-customers di grado k.

La funzione restituisce il set di utenti a distanza esatta k da user_id (CCk), dove esiste un arco u--v se hanno condiviso un terminale.

I risultati vengono restituiti come stringhe contenenti gli identificativi degli utenti.

Le fasi definite sono le seguenti:

1. Costruisce relazione tra terminale e lista utenti: a partire dalla collezione delle transazioni, definisce una pipeline che raggruppa i dati per TERMINAL_ID e associa la lista di utenti che hanno eseguito una transazione su quel terminale.

L'esecuzione dell'aggregazione restituisce un oggetto 'cursor', ossia un iteratore che permette di scorrere progressivamente i documenti risultanti.

```
pipeline = [
    {"$group": {"_id": "$TERMINAL_ID", "users": {"$addToSet": "$CUSTOMER_ID"}}}
]
cursor = db.transactions.aggregate(pipeline)
```

2. Costruisce un grafo utente → set(utenti):

Iterando sull'oggetto 'cursor' si costruisce un defaultDict, ovvero un dizionario di adiacenza che rappresenta il grafo delle relazioni tra utenti: a ciascun utente (chiave = identificativo utente) viene associato l'insieme degli utenti a lui connessi (value).

Più nello specifico, per ogni terminale considera la lista di utenti (users) e, per ogni utente u, aggiunge agli adiacenti tutti gli altri utenti v dello stesso terminale.

In questo modo si ottiene un grafo non orientato, in cui ogni utente è un nodo ed esiste un arco u — v se entrambi hanno effettuato transazioni sullo lo stesso terminale.

Si verifica poi la presenza dell'utente u di partenza all'interno del grafo: se non compare in alcuna relazione, significa che non condivide terminali con altri utenti e la funzione restituisce l'insieme vuoto, in quanto non esistono co-clienti da individuare.

```
adj = defaultdict(set)
for doc in cursor:
    users = doc.get("users", [])
    users_s = [str(u) for u in users]   # normalizziamo a stringhe per evitare mismatch
    for u in users_s:
        # aggiungi tutti gli altri utenti connessi tramite questo terminale
        adj[u].update(v for v in users_s if v != u)

start = str(user_id)
if start not in adj:
    print(adj)
    return set()  # utente senza connessioni
```

3. Definisce algoritmo di ricerca BFS fino a profondità k:

La funzione individua i co-clienti a distanza esatta **k** dall'utente di partenza tramite un algoritmo di ricerca in ampiezza **Breadth-First Search (BFS)** sul grafo costruito:

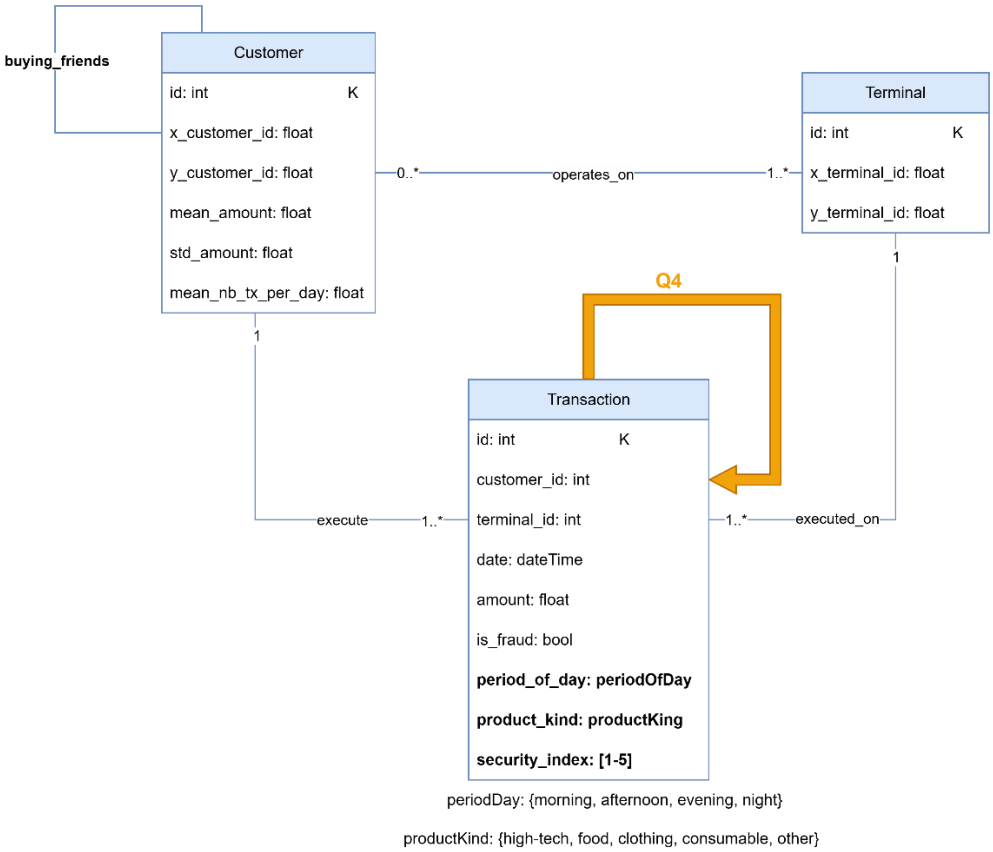
- Definisce un insieme di utenti già visitati inizialmente contenente solo l'utente u di partenza, per evitare duplicati, e un insieme di utenti da esplorare al livello corrente ('frontier').
- La BFS procede per livelli:
  - Al livello 1 si individuano tutti gli utenti direttamente collegati all'utente di partenza.
  - Al livello 2, gli utenti raggiungibili passando per due connessioni distinte.
  - Fino al livello k, con k connessioni distinte.
- Arrivato al livello k, l'insieme degli utenti presenti in 'frontier' rappresenta i co-clienti a distanza esatta k dall'utente di partenza.

```
visited = {start}
frontier = {start}
depth = 0

while depth < k:
    next_frontier = set()
    for u in frontier:
        for neigh in adj.get(u, ()):
            if neigh not in visited:
                next_frontier.add(neigh)
    visited.update(next_frontier)
    frontier = next_frontier
    depth += 1
    if not frontier:
        break
    return frontier
```

5.2 Estensione del modello dati

Come richiesto dal progetto il modello logico è stato esteso con ulteriori informazioni riguardanti la collezione delle transazioni e dei clienti e riportati nella figura sottostante.



Update - Transaction:

- **Periodo della giornata** (period_of_day) in cui è stata eseguita la transazione. Definito tramite la funzione *get_period_of_day()** a partire dal campo TX_DATETIME della transazione. I periodi definiti sono:

  Morning: 06:00–12:00 Afternoon: 12:00–18:00
  Evening: 18:00–22:00 Night: 22:00–06:00

```
*def get_period_of_day(t):
    t = datetime.strptime(t, "%Y-%m-%dT%H:%M:%S.%f")
    hour = t.hour

    if 6<= hour <12:
        return "morning"
    elif 12 <= hour < 18:
        return "afternoon"
    elif 18 <= hour < 22:
        return "evening"
    else:
        return "night"
```

- **Tipologia di prodotto acquistato** (product_kind)
  Definito in modo randomico tra le tipologie specificate:
  {high-tech, food, clothing, consumable, other}
- **Sensazione di sicurezza espressa dall'utente** (security_index) al momento della conclusione della transazione.
  Definito come valore intero randomico compreso tra 1 e 5.

Queste informazioni sono state aggiunte direttamente alla collezione Transactions tramite aggiornamenti (tramite la funzione *update_one()*) su ogni documento.

```
period = get_period_of_day(tnx["TX_DATETIME"])
index = randrange(0,len(products))
sec_val = randrange(1,6)

db.transactions.update_one(
    {"_id": tnx["_id"]},
    {"$set": {"security_index": sec_val,"product_kind": products[index],
                "period_of_day": period}}
)
```

Update - Customers:

- **Relazione 'buying_friends'**: riguarda coppie di clienti che effettuano più di tre transazioni dallo stesso terminale esprimendo una sensazione media di sicurezza simile. Due sensazioni sono considerate **simili** quando la loro differenza è inferiore a 1.

  1. A partire dalla collezione delle transazioni si raggruppano i dati per cliente (tramite CUSTOMER_ID) e terminale (TERMINAL_ID).
  2. Calcola la media della security_index e il numero di transazioni effettuate dall'utente sul terminale specificato.
  3. Considera solo i clienti con almeno 3 transazioni sullo stesso terminale.

```
pipeline = [
   {
     "$group": {
       "_id": {"user": "$CUSTOMER_ID", "terminal": "$TERMINAL_ID"},
       "avgFeeling": {"$avg": "$security_index"},
       "count": {"$sum": 1}
     }
   },
   {"$match": {"count": {"$gte": 3}}}
]

results = list(db.transactions.aggregate(pipeline))
```

Esempio di documento:

```
{"_id": { "user": "C123", "terminal": "T45" },
"avgFeeling": 4.2,
"count": 5}
```

4. I risultati vengono riorganizzati in un dizionario (by_terminal), che associa a ciascun terminale la lista di utenti con il relativo valore medio di sicurezza.
   chiave = terminale
   valore = lista di coppie (utente, avgFeeling) per tutti gli utenti che hanno usato quel terminale almeno 3 volte

```
by_terminal = {}
for r in results:
    term = r["_id"]["terminal"]
    by_terminal.setdefault(term, []).append((r["_id"]["user"], r["avgFeeling"]))
```

5. Per ogni terminale, sfruttando Pandas e Pandarallel, vengono individuate in parallelo le coppie di utenti (nella funzione *find_and_store_pairs_pandarallel()*)*.

   Per poter utilizzare pandarellel, il dizionario 'by_terminal' viene convertito in un DataFrame ('df') sul quale applica in parallelo la funzione *process_users()*** sulla lista di utenti associati a ciascun terminale.

   Tale funzione, prende in input la list di utenti e il relativo valore medio di security_index e:
   - genera tutte le possibili coppie utenti tramite **combinations**(users, 2) senza ripetizioni

   - verifica se la differenza tra valori medi di security_index è inferiore a 1 e in questo caso aggiunge la coppia al set 'pairs', ordinandola con *sorted* per impedire di avere documenti duplicati nella collezione come (u1, u2) e (u2, u1).

   - si ottiene un set di coppie per ogni terminale.

```
**def process_users(users):
    pairs = set()
    for (u1, f1), (u2, f2) in combinations(users, 2):
        if abs(f1 - f2) < 1:
            pairs.add(tuple(sorted((u1, u2)))) (u1 -> u2 e u2 -> u1)
    return pairs
```

6. Il risultato è l'insieme delle coppie di utenti che soddisfano la condizione per essere definiti **buying_friends**: hanno effettuato almeno 3 transazioni dallo stesso terminale e hanno un valore medio di security_index simile (con differenza < 1).

7. Per ogni coppia di utenti individuata viene generata un'operazione di aggiornamento da applicare al database.

   Se la coppia è già presente, non viene inserito nulla, per evitare i duplicati.

   Se invece la coppia non esiste, viene creato un nuovo documento che rappresenta la relazione di *buying_friends*.

8. Tutte le operazioni vengono eseguite in blocco tramite il metodo **bulk_write**, che permette di gestire in maniera molto più efficiente gli inserimenti ed evitare duplicazioni.

```
*def find_and_store_pairs_pandarallel(by_terminal, db):
    # trasforma in DataFrame per usare pandarallel
    df = pd.DataFrame(
        [(term, users) for term, users in by_terminal.items()],
        columns=["terminal", "users"]  )

    # calcolo parallelo dei pairs per ogni terminale
    df["pairs"] = df["users"].parallel_apply(process_users)

    all_pairs = set().union(*df["pairs"])

    # scrittura in bulk su MongoDB
    buying_friends = db["buying_friends"]
    ops = [pymongo.UpdateOne({"user_1": u1, "user_2": u2}, #filtro
            {"$setOnInsert": {"user_1": u1, "user_2": u2}}, #updateCommand
            upsert=True)
        for u1, u2 in all_pairs
    ]
    if ops:

buying_friends.bulk_write(ops,ordered=False,bypass_document_validation=True)
```

5.2.1 QUERY 4

"For each period of the day identifies the number of transactions that occurred in that period, and the average number of fraudulent transactions."

Nella risoluzione della Query 4, i dati vengono raggruppati per periodo della giornata (period_of_day, definito al punto 6.1) e per ciascun gruppo calcola il numero totale di transazioni e il numero di transazioni fraudolente, analizzando il campo TX_FRAUD.

Successivamente, tramite una proiezione, organizza i risultati per mostrare:

- periodo della giornata
- totale delle transazioni
- percentuale media di transazioni fraudolente (rapporto fra fraudolente e totali).

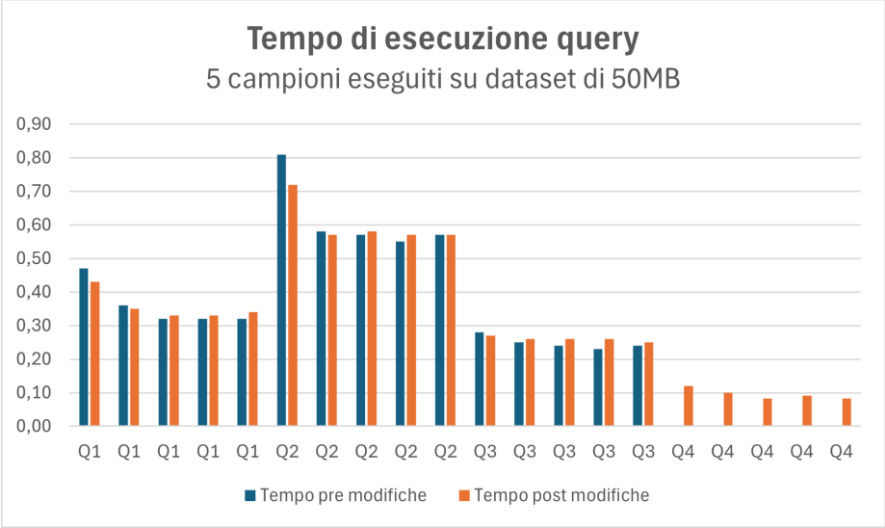
```
pipeline = [
    {
      "$group": {
        "_id": "$period_of_day",
        "totalTransactions": {"$sum": 1},
        "fraudulentTransactions": {
          "$sum": {"$cond": ["$TX_FRAUD", 1, 0]}
        }
      }
    },
    {
      "$project": {
        "_id": 0,
        "periodOfDay": "$_id",
        "totalTransactions": 1,
        "avgFraudulentTransactions": {
          "$divide": ["$fraudulentTransactions", "$totalTransactions"]
        }
      }
    }
  ]

  result = list(db.transactions.aggregate(pipeline))
```

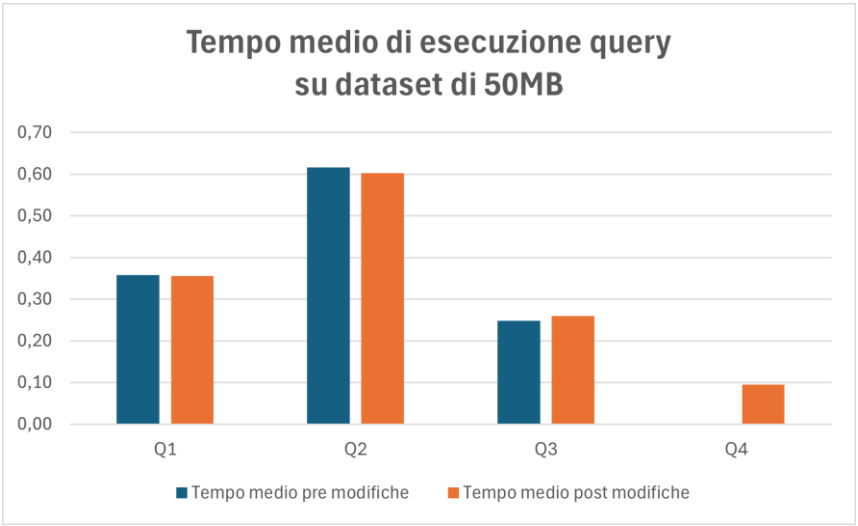
6. Valutazione delle prestazioni

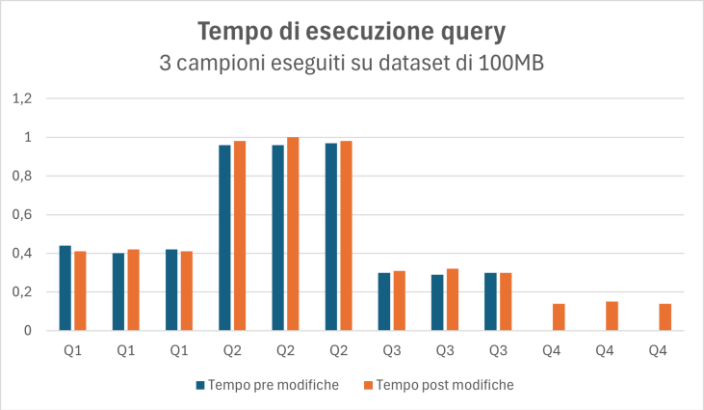
6.1 Tempo di esecuzione query

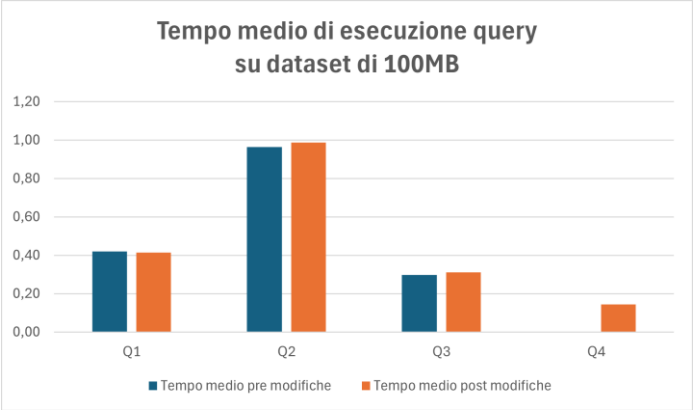
Il grafico sottostante mostra le statistiche di esecuzione delle query sul database ottenuto a partire dal dataset di 50 MB. Per ciascuna query sono state effettuate cinque esecuzioni, sia pre-modifiche al dataset, sia post estensione del dataset, in modo da disporre di un numero sufficiente di campioni utili alla definizione di statistiche affidabili.
Dall'analisi si osserva che in seguito all'aggiunta di campi extra nei documenti, eseguita dalle operazioni di modifica degli schemi, i tempi di esecuzione non subiscono variazioni significative. In media, i tempi delle query aumentano soltanto di pochi centesimi di secondo rispetto alla configurazione iniziale.



Questo secondo grafico, invece, mostra i tempi medi di esecuzione delle query eseguite sul medesimo dataset e rende ancora più evidente quanto osservato in precedenza.

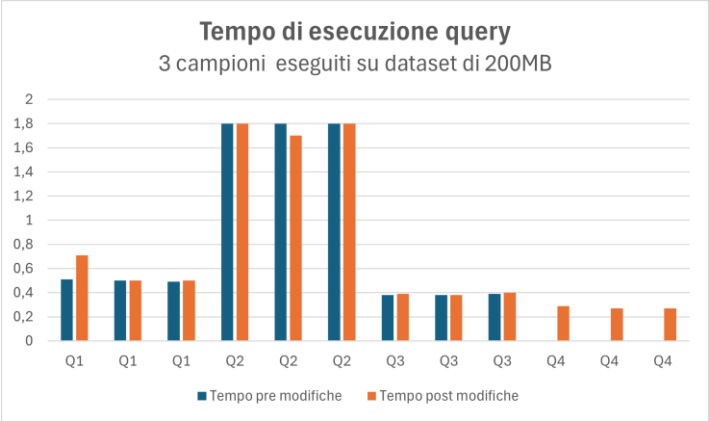


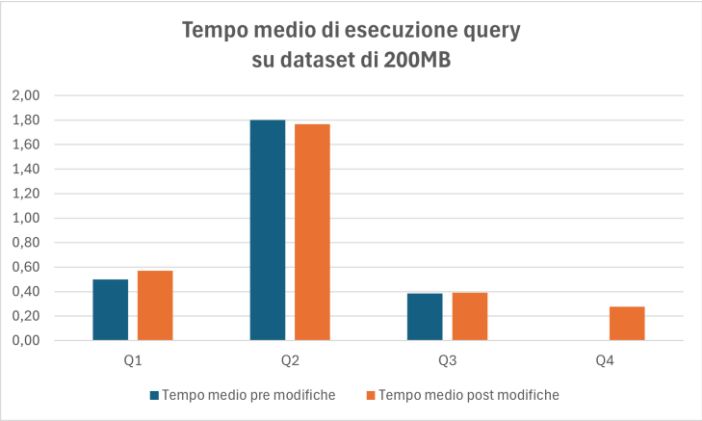




I grafici mostrati sopra riportano le statistiche di 3 campioni ed il corrispondente tempo medio di esecuzione delle varie query eseguite sul dataset di dimensione 100MB.

Confrontandolo con i risultati ottenuti col dataset da 50 MB, si può notare un aumento irrisorio nei tempi delle query 1, 3 e 4, mentre si ha avuto un incremento sostanziale del tempo di esecuzione della query 2.





Infine, vengono riportati i grafici con le stesse statistiche calcolate sul dataset da 200MB.

Anche in questo caso, gli aumenti nei tempi di esecuzione delle query 1, 3 e 4 risultano marginali, mentre si osserva un incremento significativo nel tempo di esecuzione della query 2 se confrontanto ai tempi, per la stessa query, sugli altri dataset generati.

Tale incremento è attribuibile al fatto che la collezione ‘Transaction’ è l’unica a crescere di dimensione nel passaggio da un dataset ad un altro, poiché le cardinalità delle altre collezioni vengono mantenute costanti durante il processo di generazione.

6.2 Valutazione applicazione design pattern

L'applicazione di design pattern alla struttura delle collezioni definite nel database non porterebbe a sostanziali miglioramenti nei tempi di esecuzione delle query in quanto, per come sono state strutturate le query, si è già cercato di limitare il più possibile l'accesso ai dati dei vari documenti, facendo in modo di utilizzare in ogni risoluzione solamente i campi strettamente necessari e richiesti.

Una possibile soluzione che potrebbe comportare un piccolo aumento delle performance è la definizione di nuovi indici, oltre a quelli standard definiti sulle chiavi primarie delle collezioni, per cercare di velocizzare l'accesso ai campi che più frequentemente vengono utilizzati per collegare tra loro i documenti delle varie collezioni.