New generation data models and DBMSs

Progetto

Luca Milanesi 55053A
luca.milanesi1@studenti.unimi.it

2024-2025

Indice

1 Diagramma delle classi del dominio

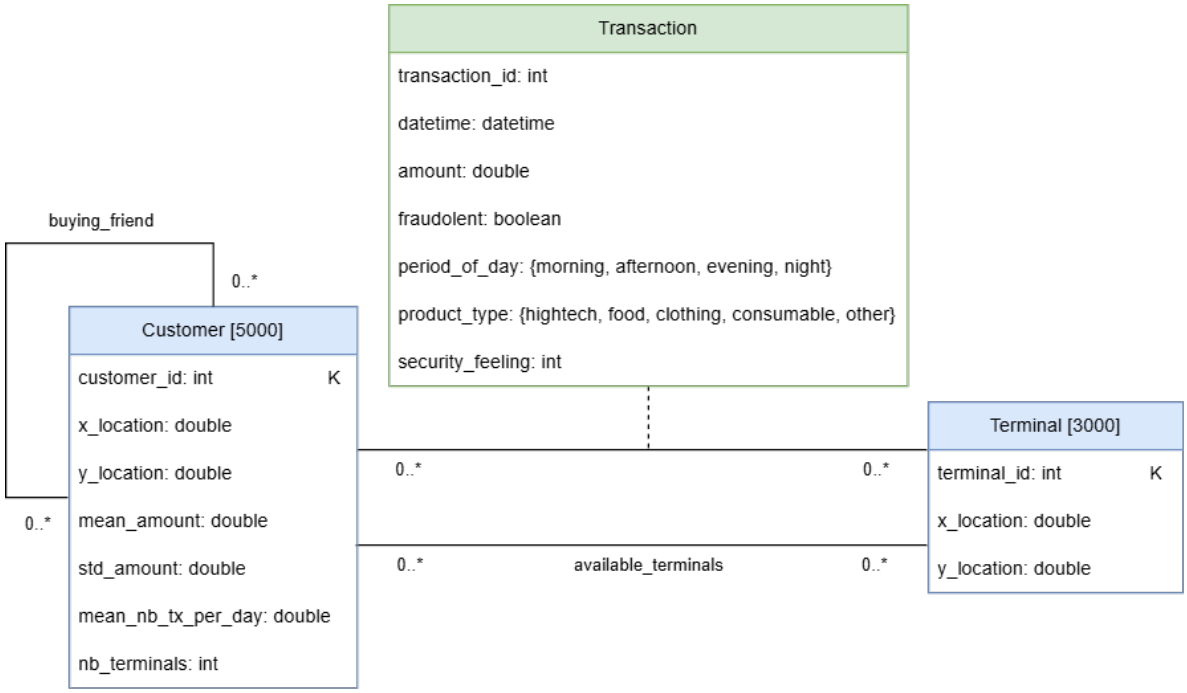

Figura 1: Diagramma delle classi del dominio.

Nella Figura 1 è descritto completamente il modello logico delle classi del dominio del progetto proposto. Per la progettazione di questo modello è stato fatto affidamento alla documentazione proposta per la generazione dei dati [1].

Inoltre, nel diagramma sono presenti, oltre che le proprietà delle richieste iniziali, anche le proprietà aggiuntive richieste nelle operazioni `d.i` e `d.ii`.

1.0.1 Descrizione

Il diagramma è composto da due classi principali:

- **Customer** che rappresenta un cliente composto da un identificatore univoco, una coppia di coordinate che indicano la sua posizione geografica, la media e la deviazione standard degli importi delle transazioni effettuate, il numero medio di transazioni al giorno e infine il numero di terminali accessibili dal cliente.
- **Terminal** che rappresenta un terminale composto da un identificatore univoco e da una coppia di coordinate che indicano la sua posizione geografica.

Inoltre, tra `Customer` e `Terminal` sono presenti due relazioni:

- `available_terminal` che indica che un cliente ha accesso a più terminali, mentre un terminale può essere acceduto da più clienti.
- `transaction` che indica che un cliente può effettuare una transazione su più terminali, mentre da un terminale possono essere fatte delle transazioni da più clienti. A questa relazione è associata la classe **Transaction**, che rappresenta la transazione effettuata da un cliente su un determinato terminale. Questa classe è composta da un identificatore univoco, data e ora di avvenimento, l'importo, una flag per indicare se è fraudolenta o meno, il periodo della giornata in cui è avvenuta, il tipo di prodotti che sono stati acquistati e infine il livello di sicurezza della transazione espresso dal cliente rispetto alle sue sensazioni.

Infine, `Customer` ha una relazione `buying_friend` ricorsiva verso se stessa, che indica che due clienti sono in relazione tra loro se eseguono più di tre transazioni dallo stesso terminale esprimendo una sensazione media di sicurezza simile.

1.0.2 Assunzioni

Di seguito sono riportate le assunzioni fatte sul modello proposto rispetto alle richieste del progetto:

- Dato che viene richiesto che il numero di clienti e terminali debba essere uguale per ogni dataset che verrà generato, ho assunto che il numero di clienti sia 5000 e quello dei terminali 3000.
- Per indicare che un terminale è accessibile ad un cliente, e che quindi appartiene alla sua area di accessibilità formata da una circonferenza, ho bisogno di un parametro r che andrà a formare la circonferenza rispetto alle coordinate del cliente. In questo caso ho assunto $r = 5$.
- Un cliente sarà identificato univocamente dal suo identificatore.
- Un terminale sarà identificato univocamente dal suo identificatore.
- Non è detto che se un cliente ha accesso ad un terminale implica esplicitamente che il cliente abbia effettuato almeno una transazione su di esso. Ci si potrebbe ritrovare nel caso in cui un cliente ha accesso ad un terminale ma non ha mai effettuato nessuna transazione grazie ad esso.

1.0.3 Vincoli

Di seguito sono riportati i vincoli del dominio del modello rispetto alle richieste del progetto:

- Un terminale t è accessibile da un cliente c se e solo se le coordinate del terminale t si trovano nella circonferenza che ha come centro le coordinate di c e come raggio il parametro r assunto.

- Un cliente c può effettuare una transazione tx su un terminale t se e solo se nella lista dei terminali $\{t_1, ..., t_n\}$ per la quale il cliente c ha accesso è presente t:

$$t \in \{t_1, ..., t_n\}$$

- Il parametro *nb_terminals* di un cliente c deve coincidere con la cardinalità dell'insieme dei terminali $\{t_1, ..., t_n\}$ per la quale il cliente c ha accesso:

$$c.nb\_terminals = n$$

- Le coordinate geografiche dei clienti e dei terminali sono definite in una griglia 100×100:

$$x\_location, y\_location \in [0, 100]$$

- Un cliente c non può essere `buying_friend` di se stesso, dunque data la lista di clienti $\{c_1, ..., c_n\}$ che sono `buying_friend` di c si deve avere:

$$c \notin \{c_1, ..., c_n\}$$

- La media degli importi delle transazioni effettuate *mean_amount* di un cliente c, data la lista delle transazioni $\{tx_1, ..., tx_n\}$ di c, deve essere:

$$c.mean\_amount = \frac{\sum_{i=1}^{n} tx_i.amount}{n}$$

- La deviazione standard degli importi delle transazioni effettuate *std_amount* di un cliente c, data la lista delle transazioni $\{tx_1, ..., tx_n\}$ di c, deve essere:

$$c.std\_amount = \sqrt{\frac{\sum_{i=1}^{n} (tx_i.amount - c.mean\_amount)^2}{n}}$$

- Dato un cliente c e la lista $\{x_1, ..., x_n\}$ del numero di transazioni effettuate giorno per giorno da c, si deve avere:

$$c.mean\_nb\_tx\_per\_day = \frac{\sum_{i=1}^{n} x_i}{n}$$

2 Modello logico a grafo per Neo4j

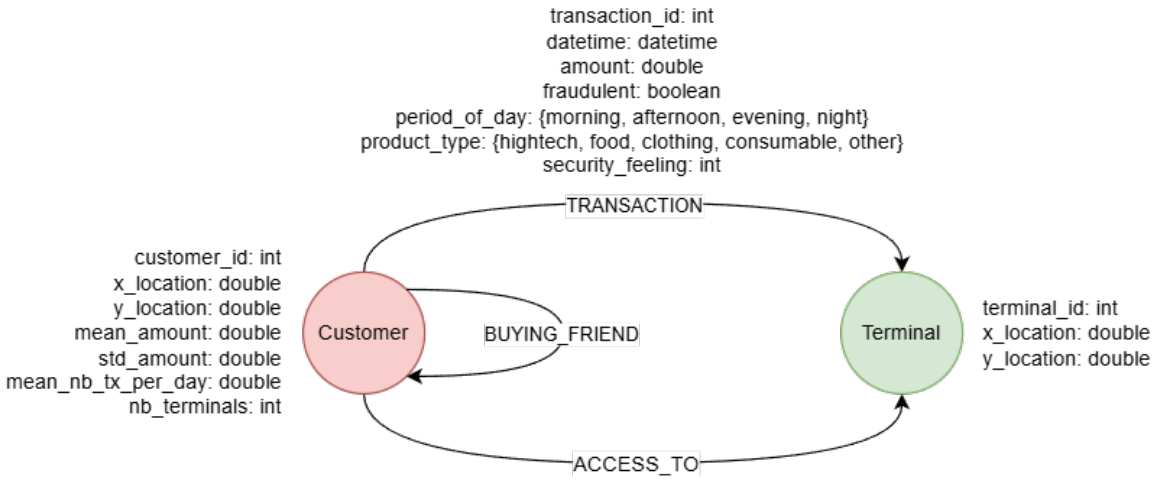

Figura 2: Modello logico a grafo.

La Figura 2 rappresenta il modello logico a grafo del progetto partendo dal diagramma delle classi progettato.

Come si può notare, il grafo è composto da due nodi, **Customer** e **Terminal** che rappresentano le rispettive entità con annesse proprietà descritte al capitolo precedente. Ho voluto evitare l'aggiunta di un terzo nodo per le transazioni, assumendola come se fosse una relazione, per poter semplificare il modello e, soprattutto, per ridurre il carico di lavoro dalle operazioni che si andranno ad eseguire. Infatti, seguendo questa strategia, si andranno a ridurre le operazioni richieste al DBMS per la navigazione del grafo di nodo in nodo.

Inoltre, le relazioni tra i nodi sono le rispettive relazioni elencate nel diagramma delle classi precedente, tra le quali:

- `BUYING_FRIEND` ricorsiva su **Customer**
- `ACCESS_TO` che parte da **Customer** e arriva a **Terminal**
- Infine, come già accennato, `TRANSACTION` da **Customer** a **Terminal** con relative proprietà annesse.

3 Descrizione dello script implementato

Lo script implementato è suddiviso in tre parti: uno per la generazione dei dati, uno per il caricamento dei suddetti dati nel database e infine lo script principale per l'esecuzione delle operazioni richieste.

Il processo generale del progetto è orchestrato dal file principale `main.py`, nella quale grazie agli script elencati precedentemente, viene eseguita la seguente sequenza di operazioni:

1. Inizializzo la connessione con il database da utilizzare grazie alle informazioni del DBMS impostate nel file `.env`.

2. Genero l'insieme di dataset richiesto da 50 Mbyte, 100 Mbyte e 200 Mbyte usando gli script di generazione dei dati. I dataset saranno nel formato `.csv`. Come detto in precedenza, per tutti i dataset è stato mantenuto fisso il numero di clienti e terminali, pari a 5000 e 3000, mentre il numero totale di transazioni è stato regolato in funzione della dimensione richiesta del dataset. La variazione nella quantità di transazioni è stata ottenuta modificando il parametro relativo al numero totale di giorni per cui generare le transazioni, partendo da una specifica data di riferimento.

3. Per ogni dataset generato al punto precedente vengono eseguite le seguenti operazioni:

   (a) Carico tutti i dati che compongono il dataset nel database usando lo script di caricamento dei dati nel database.

   (b) Eseguo tutte le operazioni richieste dalla consegna del progetto usando gli script per l'implementazione delle operazioni sul database in uso.

   (c) Salvo i tempi di esecuzione del caricamento e delle operazioni per il rispettivo dataset.

4. Infine, dai tempi di esecuzione ottenuti, genero un diagramma di analisi dei tempi osservati rispetto ad ogni dataset.

Nota Tutti gli script implementati per la soluzione finale sono disponibili nella repository di GitHub [2].

3.1 Script per la generazione dei dati

Gli script per la generazione dei dati hanno lo scopo di creare dataset sintetici che rappresentano il dominio del progetto, rispettando le assunzioni e i vincoli definiti nel modello dei dati descritto. I dati generati da questi script ovviamente non includeranno le proprietà aggiuntive richieste dalle operazioni `d.i` e `d.ii`.

Per la simulazione e generazione sintetica dei dati si sono adottate le operazioni descritte nel documento proposto per questo scopo [1], le quali sono state impiegate da `data_simulator.py` nei seguenti metodi:

- `generate_customer_profiles_table` per la generazione dei profili dei clienti utilizzando distribuzioni casuali per impostare proprietà come coordinate geografiche, importo medio delle transazioni e numero medio di transazioni giornaliere. Prende in input il numero di clienti da generare.

- `generate_terminal_profiles_table` per la generazione dei profili dei terminali assegnando coordinate casuali di una griglia geografica 100×100. Prende in input il numero di terminali da generare.

- `get_list_terminals_within_radius` per calcolare l'elenco dei terminali accessibili ad un cliente entro un determinato raggio d'azione. Prende in input il profilo di un cliente, la lista dei terminali e il raggio d'azione da prendere in considerazione.

- `generate_transactions_table` per la generazione delle transazioni giornaliere di un cliente determinando la data e l'ora, gli importi e i terminali coinvolti sulla base di determinate caratteristiche individuali del cliente [1]. Prende in input il profilo del cliente, la data di partenza per la generazione delle transazioni e il numero di giorni per cui generare le transazioni.

- `add_frauds` per aggiungere ad ogni transazione un parametro che indica se una transazione è fraudolenta o meno rispetto a determinati scenari di frode predefiniti [1]. Prende in input la lista dei profili dei clienti, dei profili dei terminali e delle transazioni generate ai punti precedenti.

Inoltre, la classe **Generator** progettata in `generator.py` gestisce la pipeline di generazione dei dati, facendo affidamento alle funzionalità che offre `data_simulator.py`. Questa classe ha come parametri di inizializzazione il numero di clienti e terminali da generare, la data di partenza e l'area di azione nella quale un terminale si deve trovare rispetto ad un cliente per essergli accessibile e offre il metodo `generate` che ha il compito di generare un dataset composto dai seguenti file:

- `customer_profiles.csv` contenente i profili cliente.
- `terminal_profiles.csv` contenente i profili dei terminali.
- `transactions.csv` contenente le transazioni dei clienti mediante i terminali.

Questo metodo, avendo a disposizione i parametri di inizializzazione della classe, prende in input il numero di giorni per la quale generare transazioni, il percorso locale nella quale salvare i file `.csv` e un nome facoltativo del dataset generato.

3.2 Script per il caricamento dei dati

Lo script per il caricamento dei dati gestisce l'inserimento dei dataset generati nel database, assicurandosi che siano rispettate le relazioni e i vincoli definiti nel modello logico. Questo processo è orchestrato dalla classe **Loader** presente in `loader.py`, che ha come parametro l'istanza del database da usare, offrendo il metodo `load_dataset` che ha il compito di caricare nel database il dataset identificato dai file `.csv` salvati nel percorso locale specificato passatogli in input. Per poter fare ciò, vengono eseguite in sequenza le seguenti operazioni:

1. Prima di iniziare il caricamento, elimino tutti i nodi e le relazioni esistenti nel database. Inoltre, vengono rimossi eventuali indici già definiti per assicurare un ambiente pulito per il caricamento dei nuovi dati.

```
MATCH (n) DETACH DELETE n;
DROP INDEX customer_index IF EXISTS;
DROP INDEX terminal_index IF EXISTS;
```

Listing 1: Query per la pulizia del database.

2. Carico i profili dei clienti presenti nel file `customer_profiles.csv` salvato nel percorso specificato, con le opportune funzionalità di Neo4j per poter importare l'apposito file e leggerne riga per riga, andando a creare un nodo per ogni cliente.

```
LOAD CSV WITH HEADERS FROM $file_path AS row
CALL (row) {
    MERGE (c:Customer {
        customer_id: toInteger(row.CUSTOMER_ID),
        x_location: toFloat(row.x_customer_id),
        y_location: toFloat(row.y_customer_id),
        mean_amount: toFloat(row.mean_amount),
        std_amount: toFloat(row.std_amount),
        mean_nb_tx_per_day: toFloat(row.mean_nb_tx_per_day),
        nb_terminals: toInteger(row.nb_terminals)
```

```
    })
} IN TRANSACTIONS OF 10000 ROWS
```

Listing 2: Query per il caricamento dei profili dei clienti.

3. In modo analogo, carico i profili dei terminali presenti nel file `terminal_profiles.csv` salvato nel percorso specificato, andando a creare un nodo per ogni terminale.

```
LOAD CSV WITH HEADERS FROM $file_path AS row
CALL (row) {
    MERGE (t:Terminal {
        terminal_id: toInteger(row.TERMINAL_ID),
        x_location: toFloat(row.x_terminal_id),
        y_location: toFloat(row.y_terminal_id)
    })
} IN TRANSACTIONS OF 10000 ROWS
```

Listing 3: Query per il caricamento dei profili dei terminali.

4. Per migliorare le prestazioni delle query successive, vengono creati degli indici sugli identificatori dei clienti e i terminali.

```
CREATE INDEX customer_index IF NOT EXISTS
    FOR (c:Customer) ON (c.customer_id);
CREATE INDEX terminal_index IF NOT EXISTS
    FOR (t:Terminal) ON (t.terminal_id);
```

Listing 4: Query per la creazione degli indici.

5. Carico le transazioni dal file `transactions.csv` salvato nel percorso specificato, creando relazioni `TRANSACTION` tra i nodi dei clienti e dei terminali.

```
LOAD CSV WITH HEADERS FROM $file_path AS row
CALL (row) {
    MATCH (c:Customer {customer_id: toInteger(row.CUSTOMER_ID)})
    MATCH (t:Terminal {terminal_id: toInteger(row.TERMINAL_ID)})
    MERGE (c)-[:TRANSACTION {
        transaction_id: toInteger(row.TRANSACTION_ID),
        datetime: datetime(replace(row.TX_DATETIME, ' ', 'T')),
        amount: toFloat(row.TX_AMOUNT),
        fraudulent: toInteger(row.TX_FRAUD) = 1
    }]->(t)
} IN TRANSACTIONS OF 10000 ROWS
```

Listing 5: Query per il caricamento delle transazioni.

6. Carico le associazioni dei terminali accessibili da ogni cliente dal file `customer_profiles.csv`, in cui viene letta la lista dei terminali accessibili per ciascun cliente andando a creare una relazione `ACCESS_TO` tra il cliente e i terminali inclusi nella lista.

```
LOAD CSV WITH HEADERS FROM $file_path AS row
CALL (row) {
    WITH toInteger(row.CUSTOMER_ID) AS customer_id,
        apoc.convert.fromJsonList(row.available_terminals)
        AS available_terminals
    UNWIND available_terminals AS terminal_id
    MATCH (c:Customer {customer_id: customer_id})
    MATCH (t:Terminal {terminal_id: terminal_id})
    CREATE (c)-[:ACCESS_TO]->(t)
} IN TRANSACTIONS OF 10000 ROWS
```

Listing 6: Query per il caricamento dei terminali accessibili da ogni cliente.

Tutte le query di caricamento sono state ottimizzate per migliorarne l'efficienza tramite gli indici e soprattutto anche grazie al caricamento in batch. Questo approccio permette di processare i dati in blocchi di 10000 elementi alla volta, riducendo il carico della memoria e migliorando l'efficienza complessiva del caricamento.

3.3 Script per l'implementazione delle operazioni richieste

Lo script per l'implementazione delle operazioni gestisce l'esecuzione delle query necessarie per soddisfare le richieste del progetto. Le operazioni sono orchestrate dalla classe **Operations** presente nel file `operations.py`, che ha come parametro l'istanza del database da utilizzare nella quale verranno analizzati e manipolati i dati. Questa classe offre un determinato metodo per ogni operazione richiesta dal progetto, per le quali di seguito è stata descritta la loro implementazione.

Operazione a

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.

1. Per ogni coppia di clienti X e Y che condivide un terminale, con $X \neq Y$, calcolo:
   (a) Il numero di terminali condivisi

(b) L'importo totale delle transazioni di X e Y su tali terminali condivisi, sapendo che X potrebbe anche non aver eseguito nessuna transazione sul terminale condiviso con Y.

2. Filtro i risultati per le coppie che hanno un numero di terminali condivisi maggiore o uguale a 3 nella quale Y ha un importo che differisce di meno del 10% rispetto a quello di X.

3. Restituisco l'identificativo del cliente X e Y, insieme agli importi spesi.

```
MATCH (x:Customer)-[tx_x:ACCESS_TO|TRANSACTION]->(t:Terminal),
    (t)<-[tx_y:TRANSACTION]-(y:Customer)
WHERE x <> y
WITH x.customer_id AS customer_x,
    y.customer_id AS customer_y,
    SUM(COALESCE(tx_x.amount, 0)) AS amount_x,
    SUM(tx_y.amount) AS amount_y,
    COUNT(DISTINCT t) AS shared_terminals
WHERE shared_terminals >= 3
    AND ABS(amount_x - amount_y) < 0.1 * amount_x
RETURN customer_x, amount_x, customer_y, amount_y
```

Listing 7: Operazione a.

Operazione b

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.

1. Calcolo la data corrente.

2. Per ogni terminale, calcolo la media degli importi delle transazioni avvenute nel lasso di tempo tra due mesi precedenti e un mese precedente rispetto alla data corrente.

3. Per ogni terminale, filtro le transazioni avvenute nel lasso di tempo tra l'inizio del mese precedente e la data corrente nella quale l'importo eccede del 20% rispetto a questa media, andando così a trovare le possibili transazioni fraudolente del mese corrente.

4. Restituisco, per ogni terminale, la lista delle transazioni potenzialmente fraudolente trovate al passo precedente.

```
WITH date() AS current_date

MATCH (t:Terminal)<-[tx:TRANSACTION]-(:Customer)
WITH t, tx, date(tx.datetime) AS tx_datetime, current_date
WHERE tx_datetime >= current_date - duration({months: 2})
    AND tx_datetime < current_date - duration({months: 1})
WITH t, AVG(tx.amount) AS avg_amount_last_month, current_date

MATCH (t)<-[tx:TRANSACTION]-(:Customer)
WITH t.terminal_id AS terminal_id, avg_amount_last_month,
    current_date, tx, date(tx.datetime) AS tx_datetime
WHERE tx_datetime >= current_date - duration({months: 1})
    AND tx_datetime < current_date
WITH terminal_id, avg_amount_last_month,
    tx.transaction_id AS transaction_id, tx.amount AS amount

WHERE amount - avg_amount_last_month > 0.2 * avg_amount_last_month
RETURN terminal_id,
    COLLECT(transaction_id) AS possible_fraudulent_transaction
```

Listing 8: Query b.

Operazione c

Given a user u, determine the "co-customer-relationships CC of degree k". A user u' is a cocustomer 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 $i, j \in [1, k]$, $u_i \neq u_j$, and $t_1, ..., t_{k-1}$ are the terminals on which a transaction has been executed. Therefore:

$$CC_k(u) = \{u' | \textit{a chain exists between } u \textit{ and } u' \textit{ of degree } k\}$$

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).

Questa è un operazione abbastanza elaborata da implementare. Prende in input l'identificativo dell'utente u da cui partire e il grado k.

Inanzitutto c'è da dire che se il grado k della relazione è inferiore a 2 non è possibile travare soluzioni, questo perché:

- nel caso $k < 0$ è impossibile.

- nel caso $k = 1$ la catena sarà composta solamente dal primo elemento $u_1 = u$, il che non soddisferebbe la condizione:

$$\forall i, j \in [1, k], u_i \neq u_j$$

Quindi il grado deve essere obbligatoriamente $k \geq 2$. Per costruire la query basata su k passi, viene adottato un approccio iterativo.

1. Ogni passo rappresenta un salto da un cliente a un altro attraverso un terminale.
2. Ad ogni iterazione i che va da 1 a $k - 1$:
   (a) Nel caso sia la prima iterazione la catena partirà dall'utente $u_1 = u$ passato in input.
   (b) Si aggiunge un nuovo nodo utente u_{i+1} connesso ad un terminale che condivide con l'ultimo utente u_i aggiunto alla catena all'iterazione precedente che non sia già presente in essa.
   (c) Si aggiorna la catena di utenti tracciata per evitare duplicati nelle iterazioni successive.

```
MATCH (u{i})-[:TRANSACTION]->(t:Terminal),
    (t)<-[:TRANSACTION]-(u{i+1}:Customer)
WHERE NOT u{i+1}.customer_id IN path_customers
WITH DISTINCT u{i+1},
    path_customers + u{i+1}.customer_id AS path_customers
```

Listing 9: Parte della query creata ad ogni iterazione.

3. All'ultimo passo $i = k - 1$ restituisco i co-customer finali di grado k trovati.

```
RETURN DISTINCT u{k} AS co_customer
```

Listing 10: Parte finale della query.

Ad esempio, la query generata con passati come input l'identificatore dell'utente 0 e il grado $k = 3$ sarà la seguente:

```
MATCH (u1:Customer {customer_id: 0}),
    (u1)-[:TRANSACTION]->(:Terminal)<-[:TRANSACTION]-(u2:Customer)
WHERE u1.customer_id <> u2.customer_id
WITH DISTINCT u2, [u1.customer_id, u2.customer_id] AS path_customers

MATCH (u2)-[:TRANSACTION]->(:Terminal)<-[:TRANSACTION]-(u3:Customer)
WHERE NOT u3.customer_id IN path_customers
```

```
WITH DISTINCT u3, path_customers + u3.customer_id AS path_customers

RETURN DISTINCT u3 AS co_customer
```

Listing 11: Query `c` partendo dall'utente 0 e di grado 3.

Operazione d.i

Each transaction should be extended with:

1. *The period of the day morning, afternoon, evening, night in which the transaction has been executed.*
2. *The kind of products that have been bought through the transaction hightech, food, clothing, consumable, other*
3. *The feeling of security expressed by the user. This is an integer value between 1 and 5 expressed by the user when conclude the transaction.*

The values can be chosen randomly.

Estendo ogni transazione con tre attributi:

1. Periodo del giorno basato sull'ora della transazione.
2. Tipo di prodotto scelto casualmente.
3. Livello di sicurezza percepito dal cliente (valore casuale tra 1 e 5).

```
MATCH ()-[tx:TRANSACTION]->()
WITH tx, time(tx.datetime).hour AS hour, rand() AS rand
SET tx.period_of_day = CASE
    WHEN hour >= 6 AND hour < 12 THEN 'morning'
    WHEN hour >= 12 AND hour < 18 THEN 'afternoon'
    WHEN hour >= 18 AND hour < 22 THEN 'evening'
    ELSE 'night'
END,
tx.product_type = CASE
    WHEN rand < 0.2 THEN 'hightech'
    WHEN rand < 0.4 THEN 'food'
    WHEN rand < 0.6 THEN 'clothing'
    WHEN rand < 0.8 THEN 'consumable'
    ELSE 'other'
END,
tx.security_feeling = toInteger(rand() * 5) + 1
```

Listing 12: Query `d.i`.

Operazione d.ii

Customers that make more than three transactions from the same terminal expressing a similar average feeling of security should be connected as ***buying_friends****. Therefore also this kind of relationship should be explicitly stored in the NOSQL database and can be queried. Note, two average feelings of security are considered similar when their difference is lower than 1.*

1. Per ogni cliente cerco i terminali sui quali sono state effettuate più di 3 transazioni, calcolando anche la media delle sensazioni di sicurezza espresse su suddette transazioni, andando così a creare coppie clienti e terminali.

2. Per ogni coppia cliente e terminale trovata, cerco i clienti (diversi rispetto al cliente iniziale) che hanno effettuato più di 3 transazioni sullo stesso terminale, calcolando la media delle sensazioni di sicurezza espresse su suddette transazioni, andando così a creare coppie di clienti che hanno effettuato almeno 3 transazioni sullo stesso terminale con relative medie delle sensazioni di sicurezza.

3. Filtro le coppie di clienti con una media della sensazione di sicurezza simile, ovvero che hanno una differenza minore di 1.

4. Creo una relazione `BUYING_FRIEND` per le coppie di clienti finali trovate.

```
MATCH (c1:Customer)-[tx1:TRANSACTION]->(t:Terminal)
WITH c1, t, COUNT(tx1) AS tx1_count,
    AVG(tx1.security_feeling) AS c1_avg_security
WHERE tx1_count > 3
WITH c1, t, c1_avg_security

MATCH (c2:Customer)-[tx2:TRANSACTION]->(t)
WHERE c1 <> c2
WITH c1, c2, c1_avg_security, COUNT(tx2) AS tx2_count,
    AVG(tx2.security_feeling) AS c2_avg_security
WHERE tx2_count > 3
WITH c1, c2, c1_avg_security, c2_avg_security

WHERE ABS(c1_avg_security - c2_avg_security) < 1
MERGE (c1)-[:BUYING_FRIEND]->(c2)
```

Listing 13: Query `d.ii`.

Operazione e

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

1. Raggruppa tutte le transazioni per periodo del giorno.
2. Conta il numero totale di transazioni per ogni periodo del giorno e calcola la media di quelle fraudolente.
3. Restituisco il periodo, il numero di transazioni e la media di frodi.

```
MATCH ()-[tx:TRANSACTION]->()
RETURN tx.period_of_day AS period_of_day,
    COUNT(tx) AS transactions,
    AVG(CASE WHEN tx.fraudulent
             THEN 1
             ELSE 0
        END) AS avg_fraudulent_transactions
```

Listing 14: Query e.

4 Discussione delle performance

Dataset	Caricamento	a	b	c	d.i	d.ii	e
Dataset 50MB	112	182	2	4	8	1160	0.5
Dataset 100MB	317	520	4	16	24	3494	1
Dataset 100MB	1082	2211	7	57	32	7550	2

Tabella 1: Tabella dei tempi di esecuzione in secondi.

La Tabella 1 riporta i tempi di esecuzione riscontrati per il caricamento e l'esecuzione delle operazioni sui vari dataset, di dimensione crescente, espressa in secondi. Per l'operazione `c` come parametri sono stati usati per tutti i dataset gli stessi valori (utente 0 e grado $k = 3$).

È importante sottolineare che tutte le operazioni di caricamento ed esecuzione sono state effettuate in assenza di concorrenza a livello di database, garantendo che i tempi rispecchino un'esecuzione sequenziale e non parallela.

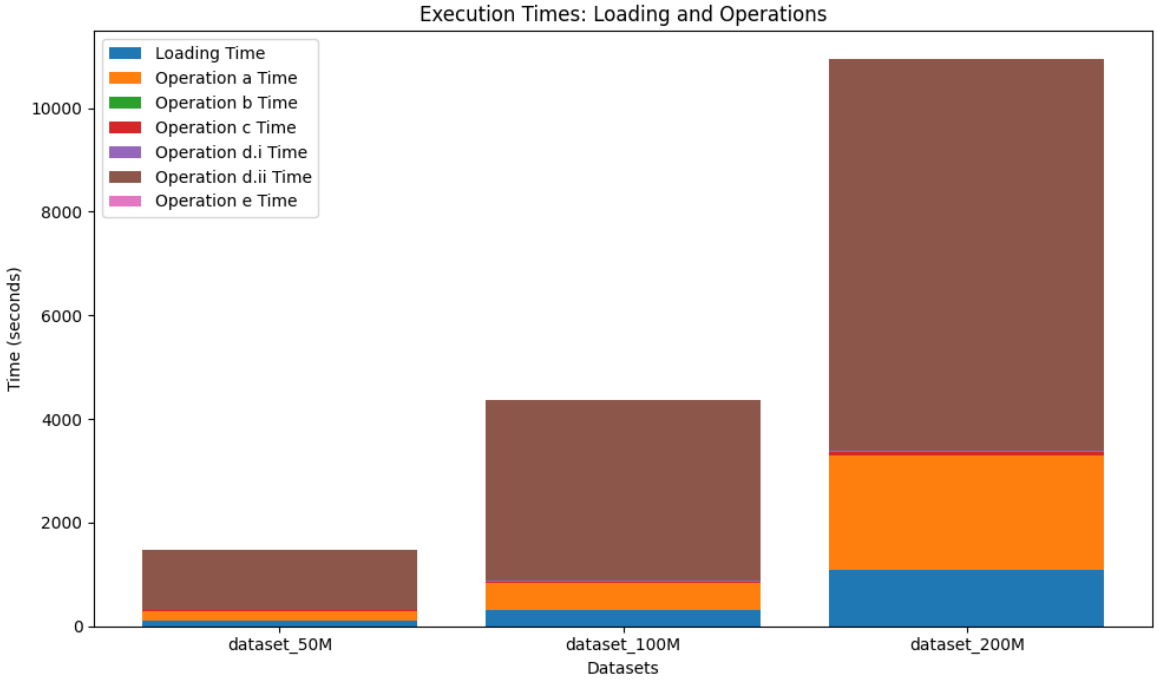

Figura 3: Diagramma di analisi dei tempi osservati.

Per rendere più chiara e immediata la comprensione dei tempi osservati, possiamo fare uso del plot grafico di analisi generato al termine dell'esecuzione delle operazioni richieste, mostrato nella Figura 3.

Come è evidente, i tempi di esecuzione fuori dalla media con l'aumentare dei dati sono il caricamento del dataset e le operazioni `a` e `d.ii`.

Performance nel caricamento dei dati

Per quanto riguarda i tempi registrati per le operazioni di caricamento, pur essendo superiori rispetto alla media delle altre operazioni, risultano comunque accettabili. Questo è giustificato dalla natura stessa dei dataset, caratterizzati da un elevato volume di dati (ad esempio, il dataset da 200 Mbyte contiene quasi 4 milioni di transazioni). Inoltre, questi tempi di caricamento sono influenzati significativamente dalla configurazione dell'ambiente in cui vengono eseguite le operazioni.

Performance e possibili migliorie nelle operazioni `a` e `d.ii`

Per quanto riguarda invece i tempi di esecuzione delle operazioni `a` e `d.ii` si vede chiaramente che sono sproporzionati rispetto alla media e che "esplodono" con l'aumentare dei dati, soprattutto l'operazione `d.ii`. Per poter capire dove peccano di efficienza bisogna porsi una domanda fondamentale:

Quali sono gli elementi più onerosi in termini computazionali nelle seguenti operazioni?

Rispetto alle altre query, queste due operazioni si distinguono per l'elevato numero di operazioni di aggregazione richieste.

Infatti, per l'operazione `a` viene richiesto di calcolare, per ogni coppia di clienti X e Y che condividono un terminale, la somma degli importi delle transazioni effettuate su suddetto terminale. Queste operazioni, infatti, richiedono una buona quantità tempo.

Per ovviare a questo problema, si potrebbe molto semplicemente salvare la somma totale degli importi delle transazioni che un cliente effettua rispetto un determinato terminale alla quale può accedere, attuando così una **precomputazione** dei dati necessari, riducendo il carico computazionale della query. Questa nuova priprietà potrebbe essere aggiunta nella relazione `ACCESS_TO` del nostro modello dei dati.

Invece, nell'operazione `d.ii` viene richiesto di calcolare, per ogni cliente rispetto ad un terminale a lui accessibile sulla quale ha effettuato delle transazioni, il numero totale di transazioni effettuate e la media delle sensazioni di sicurezza espresse dal cliente relative a queste transazioni sul terminale preso in considerazione. Anche in questo caso queste operazioni di aggregazione richiedono un discreto carico computazionale.

Anche in questo caso si potrebbe utilizzare la stessa strategia adottata precedentemente **precomputando** i dati necessari, salvando quindi per ogni cliente rispetto ad un terminale a lui accessibile, il numero di transazioni effettuate e la media delle sensazioni di sicurezza espresse dal cliente relative a queste transazioni. Come detto prima, queste nuove proprietà potrebbero essere salvate nella relazione `ACCESS_TO` del modello dei dati.

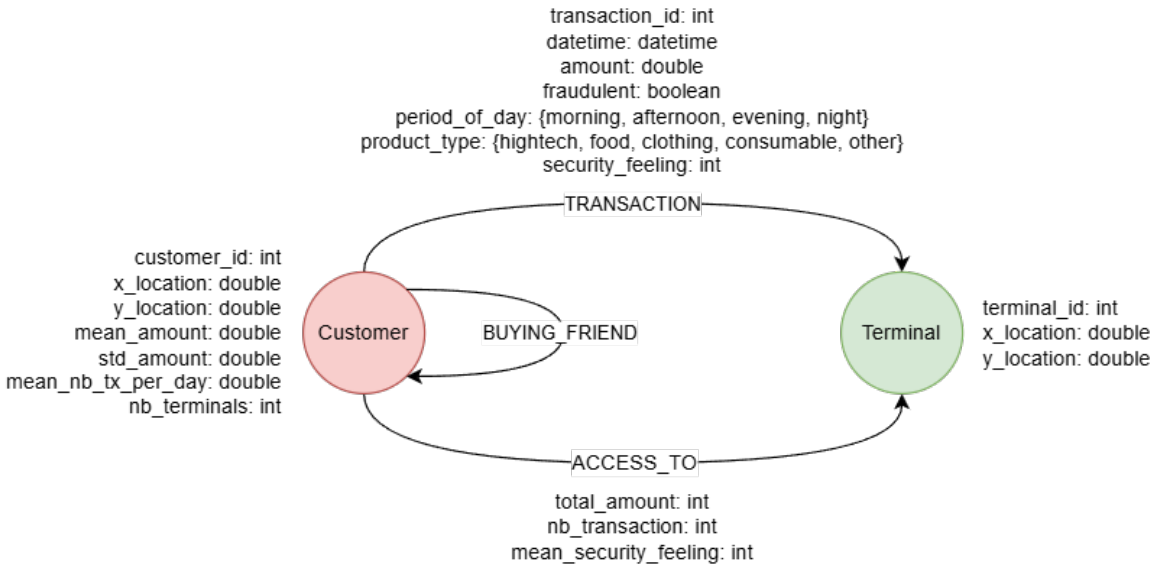

Figura 4: Modello logico a grafo con le nuove migliorie.

Nella Figura 4 è presente il modello dei dati a grafo modificato con le nuove migliorie per i tempi di esecuzione delle operazioni `a` e `d.ii` discusse precedentemente.

Riferimenti bibliografici

[1] Machine Learning Group (Université Libre de Bruxelles - ULB). *Fraud Detection Handbook, Simulated Dataset*. 2021. URL: https://fraud-detection-handbook.github.io/fraud-detection-handbook/Chapter_3_GettingStarted/SimulatedDataset.html.

[2] Luca Milanesi. *GitHub Repository del progetto*. 2025. URL: https://github.com/Luca-02/credit-card-fraud-detection.