2022/2023

Relazione progetto

CREATION OF A NOSQL DATABASE FOR CREDIT CARD FRAUD DETECTION

SAMUELE PALUMBO

1. Design

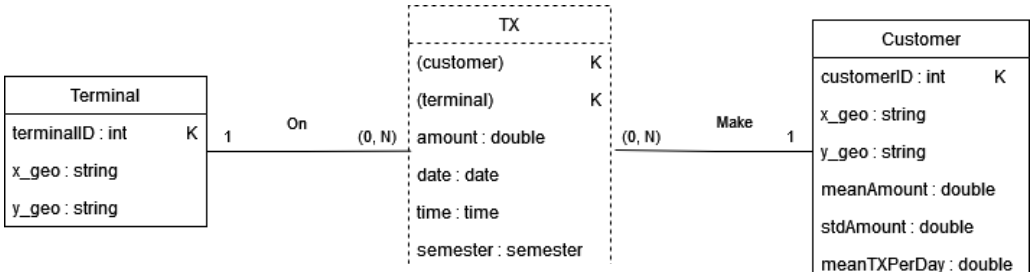

Figura1 - UML

Il diagramma UML della figura 1 prende completamente spunto dal testo d'esame. Il testo infatti richiede l'utilizzo di 3 tabelle:

- Cliente, le cui proprietà richieste sono:
  - Un identificatore
  - Una posizione geografica
  - Una media di spesa
  - Una frequenza di acquisti
- Terminale, le cui proprietà richieste sono:
  - Un identificatore
  - Una posizione geografica
- Transazione, le cui proprietà richieste sono:
  - L'identificatore del cliente
  - L'identificatore del terminale
  - L'ammontare della transazione
  - La data della transazione
  - Un eventuale flag se la stessa transazione è identificata come fraudolenta

La successiva valutazione del *workload* della figura 2 ha evidenziato come le operazioni richieste dal testo di esame hanno sempre inizio dai clienti o dai terminali.

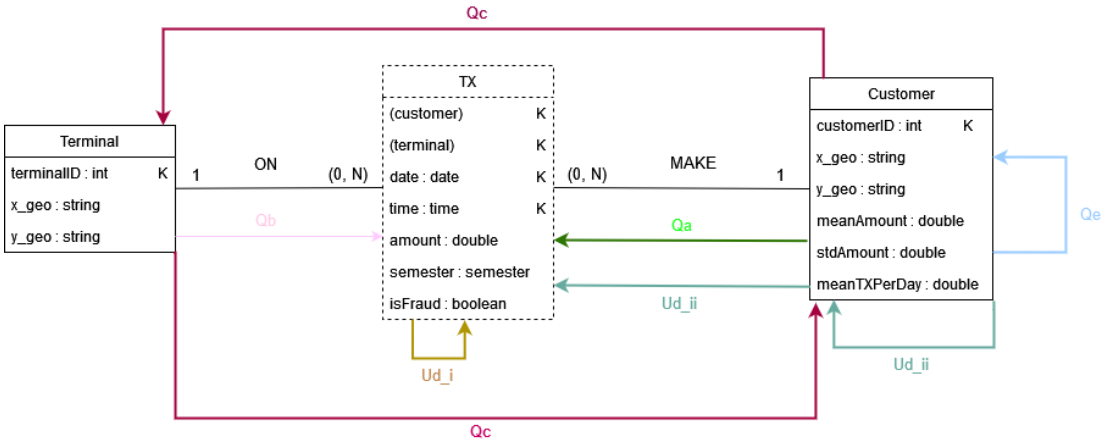

Figura 2 - Workload

Solo l'update *Ud_i* ha inizio dalle transazioni, ma non attraversa alcuna relazione. Inoltre è facile vedere come la query più costosa in termini computazionali sia la *Qc*. Questa richiede infatti di attraversare quattro relazioni per ogni coppia di clienti se il grado richiesto è di 2, otto relazioni invece se si richiede fino al grado 3. Ho quindi ritenuto necessario ridurre il numero di relazioni da attraversare in modo da ridurre i costi computazionali, soprattutto per quanto riguarda il terzo dataset. Le transazioni quindi da classi sono diventate delle relazioni tra il cliente e il terminale.

Il testo poi non specifica chiaramente come una transazione debba essere identificata, può essere utilizzata la combinazione degli attributi *<customer, terminal, data, ora>*. Per semplificare ho preferito creare un identificatore surrogato, *txID*.

A ogni transazione ho poi aggiunto l'attributo *semestre* che può assumere valore pari a *1* (da Gennaio a Giugno) o *2* (da Luglio a Dicembre). È un attributo che può essere calcolato run-time dalla data della transazione. Dato che però in due query, *Qa* e *Qb*, viene richiesto di utilizzarlo ho deciso di introdurlo come attributo a sé stante, calcolandolo in fase di creazione dei dati.

Il diagramma modificato è quindi quello mostrato nella figura 3.

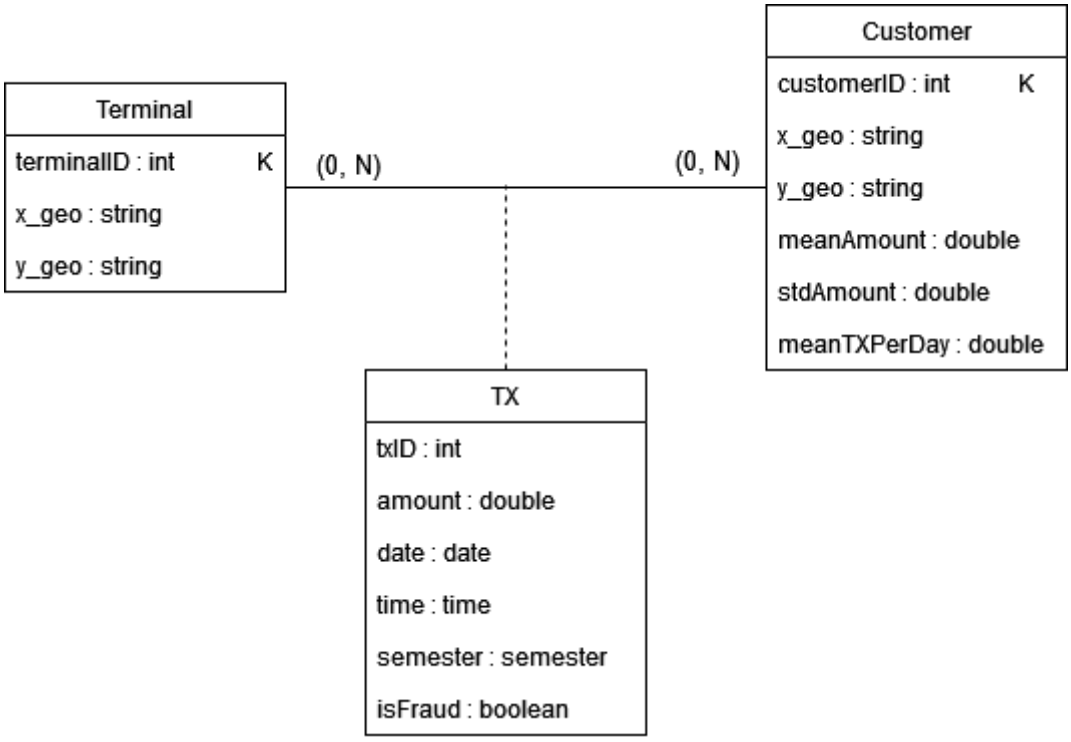

Figura 3 – UML finale

Il modello di database che ho deciso di adottare per sviluppare l'applicazione è Neo4j essendo il sistema che meglio si adatta a progetti il cui scopo è gestire transazioni ed individuare frodi. Una volta svolte tutte le operazioni richieste dal testo di esame il risultato sarà quello mostrato dalla figura 4.

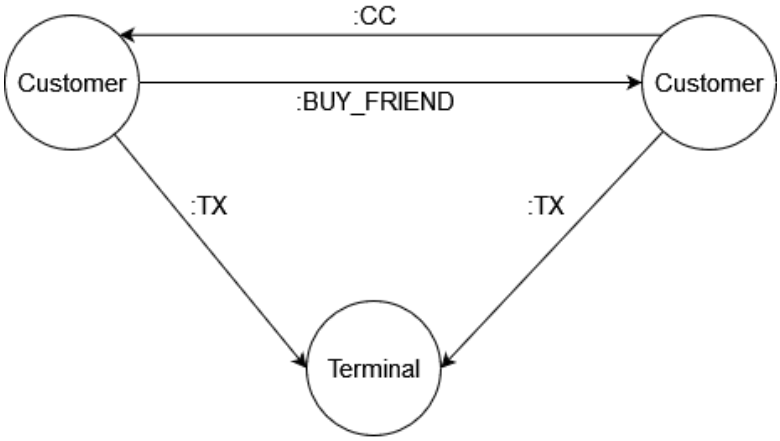

Figura 4

2. Loading scripts

2.1. Generazione dati

I dati sono generati a partire da codice scritto in *python* nella cartella */py/generator.py*, avviando il programma tramite terminale verrà richiesto quali fra i tre dataset si vogliono generare.

Una volta terminata la generazione dei dataset, nella cartella */csv/* vi saranno le tre cartelle corrispondenti ai tre dataset con diversi file csv contenenti i dati, per ogni dataset:

- Un file di clienti
- Un file di terminali
- *N* file di transazioni. Divise su più file in base alla grandezza del dataset per semplificare il successivo import (1 milione di entry per file)

2.2. Small dataset

```
// query 1
LOAD CSV WITH HEADERS FROM "file:///small_customers.csv" AS cust
MERGE(a:Customer {customerId: cust.customerId, x_geo: cust.x_geo,
y_geo: cust.y_geo, meanAmount: cust.meanAmount, std_amount:
cust.std_amount, mean_tx_per_day: cust.mean_tx_per_day})

// query 2
LOAD CSV WITH HEADERS FROM "file:///small_terminals.csv" AS terms
MERGE(b:Terminal {terminalId: terms.terminalId, x_geo:
terms.x_geo, y_geo: terms.y_geo})

// query 3
:auto LOAD CSV WITH HEADERS FROM "file:///small_txs1.csv" AS txs
CALL {
  WITH txs
  MATCH (c:Customer {customerId: txs.customerId})
  MATCH (te:Terminal {terminalId: txs.terminalId})
  MERGE ((c) -[:TX {txId: toInteger(txs.txId), amount:
toFloat(txs.amount), date: date(txs.date), time: time(txs.time),
semester: toInteger(txs.semester), isFraud:
toBoolean(txs.isFraud)}] -> (te))
} IN TRANSACTIONS OF 1500 ROWS
```

2.3.Medium dataset

```
// query 1
LOAD CSV WITH HEADERS FROM "file:///med_customers.csv" AS cust
MERGE(a:Customer {customerId: cust.customerId, x_geo: cust.x_geo,
y_geo: cust.y_geo, meanAmount: cust.meanAmount, std_amount:
cust.std_amount, mean_tx_per_day: cust.mean_tx_per_day})

// query 2
LOAD CSV WITH HEADERS FROM "file:///med_terminals.csv" AS terms
MERGE(b:Terminal {terminalId: terms.terminalId, x_geo:
terms.x_geo, y_geo: terms.y_geo})

// query 3
:auto LOAD CSV WITH HEADERS FROM "file:///med_txs1.csv" AS txs
CALL {
    WITH txs
    MATCH (c:Customer {customerId: txs.customerId})
    MATCH (te:Terminal {terminalId: txs.terminalId})
    MERGE ((c)-[:TX {txId: toInteger(txs.txId), amount:
toFloat(txs.amount), date: date(txs.date), time: time(txs.time),
semester: toInteger(txs.semester), isFraud:
toBoolean(txs.isFraud)}] -> (te))
} IN TRANSACTIONS OF 1500 ROWS
```

2.4. Large dataset

```
// query 1
LOAD CSV WITH HEADERS FROM "file:///big_customers.csv" AS cust
MERGE(:Customer {
  customerId: toInteger(cust.customerId),
  x_geo: toFloat(cust.x_geo),
  y_geo: toFloat(cust.y_geo),
  meanAmount: toFloat(cust.meanAmount),
  std_amount: toFloat(cust.std_amount),
  mean_tx_per_day: toFloat(cust.mean_tx_per_day)
})

// query 2
LOAD CSV WITH HEADERS FROM "file:///big_terminals.csv" AS terms
MERGE(b:Terminal {terminalId: toInteger(terms.terminalId),
x_geo: toFloat(terms.x_geo), y_geo: toFloat(terms.y_geo)})

// query 3
:auto LOAD CSV WITH HEADERS FROM "file:///med_txs1.csv" AS txs
CALL { WITH txs
    MATCH (c:Customer {customerId: txs.customerId})
    MATCH (te:Terminal {terminalId: txs.terminalId})
    MERGE ((c)-[:TX {txId: toInteger(txs.txId),
amount:toFloat(txs.amount), date: date(txs.date),
time:time(txs.time), semester:toInteger(txs.semester),
isFraud:toBoolean(txs.isFraud)}] ->(te))
} IN TRANSACTIONS OF 1500 ROWS
```

2.5. Indici

Avvenuta l'importazione dei dati tramite file *csv* si consiglia la creazione dei seguenti indici su tutti e tre i dataset

```
CREATE INDEX cust_ind
FOR (c:Customer) ON (c.customerId);

CREATE INDEX term_ind
FOR (t:Terminal) ON (t.terminalId);

CREATE INDEX txIndex
FOR ()-[r:TX]-()
ON (r.txId);
```

3. Query scripts

3.1. Query *a*

```
// For each customer identifies the amount that he/she has spent
for every week of the current semester
with date().month as currentMonth
with
    CASE
        // we are in July-December, it's second semester
        WHEN currentMonth >= 7  THEN 2
        // we are in January-June, it's first semester
        WHEN currentMonth <= 6 THEN 1
    END AS currentSemester
match(c:Customer)-[r:TX]->(n)
where r.semester = currentSemester
return c.customerId as customer, date.truncate("week", r.date) as
startingDayOfWeek, sum(r.amount) as totalOfTheWeek
ORDER BY startingDayOfWeek
```

3.2. Query *b*

```
// For each terminal identify the possible fraudulent
transactions. The fraudulent transactions are those whose import
is higher or lower than 10% of the average import of the
transactions executed on the same terminal in the previous
semester
with date().month as currentMonth
with
     CASE
          // we are in July-December, it's second semester
          WHEN currentMonth >= 7 THEN [1,2]
          // we are in January-June, it's first semester
          WHEN currentMonth <= 6 THEN [2,1]
     END AS semesters    // [previusSemester, currentSemester]
match (n)-[r:TX{semester:semesters[0]}]->(te:Terminal)
with te, avg(r.amount) as avg, semesters
with avg+avg/10 as upBound, avg-avg/10 as downBound, te, avg,
semesters
match (c:Customer)-[tr:TX]->(te2:Terminal)
where tr.semester = semesters[1] and te2.terminalId =
te.terminalId and (tr.amount > upBound or tr.amount < downBound)
set tr.isFraud = true
return te.terminalId as Terminal, collect(tr.txId) as
fraudolentTXS
```

3.3. Query *c*

La query *c* è stata costruita in due passaggi. Per ridurre la quantità di relazioni da attraversare, portando quindi a un minor costo computazionale, sono prima state introdotte le relazioni *CC* (*co-customer*) tra tutti quei clienti che hanno effettuato operazioni sullo stesso terminale.

```
// 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 “u1-t1-u2-t2-...tk-1-uk“ such that u1=u,
uk=u’, and for each 1<=i,j<=k, ui <> uj, and t1,..tk-1 are the
terminals on which a transaction has been executed. Therefore,
CCk(u)={u’| a chain exists between u and u’ of degree k}.
match
    (c1:Customer)-[r1:TX]-(t:Terminal),
    (c2:Customer)-[r2:TX]-(t)
where
    c1.customerId < c2.customerId
MERGE (c1)-[:CC]-(c2)
```

Per individuare quindi i *co-customer* di grado *k* si è sviluppata la seguente query, dove il parametro da impostare è *k-1* (nell’esempio k=3, impostato quindi 2 nella query)

```
match (c1:Customer)-[:CC*2]-(c2:Customer)
where c1.customerId < c2.customerId
return distinct c1.customerId as c1, c2.customerId as c2 order
by c1, c2
```

3.4. Query *d_i*

Le due query *d_i* non sono molto diverse, infatti si iterano tutte le transazioni e si aggiungono dei parametri randomizzati o basati su altri già esistenti. Ovviamente se effettuate insieme portano a un risparmio di tempo dovuta alla singola iterazione di tutte le transazioni, piuttosto che iterarle tutte due volte.

3.4.1. Query d_i1

```
// Each transaction should be extended with the period of
the day {morning (7-12), afternoon(13-18), evening(19-00),
night(1-6)} in which the transaction has been executed
:auto match (n)-[r:TX]-(m)
CALL {
    with r
    with r, r.time.hour as H
    with
        CASE
            WHEN H >= 7 and H <= 12  THEN "morning"
            WHEN H >= 13 and H <= 18  THEN "afternoon"
            WHEN H >= 19 or H = 0 THEN "evening"
            WHEN H >= 1 and H <= 6  THEN "night"
        END AS dayPeriod, r
    SET r += {period: dayPeriod}
} IN TRANSACTIONS OF 16000 ROWS
```

3.4.2. Query d_i2

```
// The kind of products that have been bought through the
transaction {high-tech, food, clothing, consumable, other}
:auto match (n)-[r:TX]-(m)
CALL {
    with r
    with toInteger(rand()*5) as X, r
    with
        CASE
            WHEN X = 0 THEN "high-tech"
            WHEN X = 1 THEN "food"
            WHEN X = 2 THEN "clothing"
            WHEN X = 3 THEN "consumable"
            WHEN X = 4 THEN "other"
        END AS type, r
    SET r += {kind: type}
} IN TRANSACTIONS OF 16000 ROWS
```

3.4.3. Query d_i1+d_i2

```
// d_i1 + d_i2
:auto match (n)-[r:TX]-(m)
CALL {
    with r
    with r.time.hour as H, r
    with
        CASE
            WHEN H >= 7 and H <= 12  THEN "morning"
            WHEN H >= 13 and H <= 18  THEN "afternoon"
            WHEN H >= 19 or H = 0 THEN "evening"
            WHEN H >= 1 and H <= 6  THEN "night"
        END AS dayPeriod, r
    with apoc.text.random(1, '01234') as X, dayPeriod, r
    with
        CASE
            WHEN X = '0' THEN "high-tech"
            WHEN X = '1' THEN "food"
            WHEN X = '2' THEN "clothing"
            WHEN X = '3' THEN "consumable"
            WHEN X = '4' THEN "other"
        END AS type, r, dayPeriod
    SET r += {kind: type, period: dayPeriod}
} IN TRANSACTIONS OF 16000 ROWS
```

3.5. Query *d_ii*

```
// Customers that make more than three transactions related to
the same types of products from the same terminal should be
connected as “buying_friends”.
:auto
UNWIND ["high-tech","food","clothing","consumable","other"]
as category
CALL{
    with category
    match (c:Customer)-[r:TX {kind:category}]-(t:Terminal)
    with c.customerId as customer, t.terminalId as terminal,
count(r) as numR
    where numR > 3
    with terminal, collect( distinct customer) as customers
    UNWIND apoc.coll.combinations(customers, 2) as couple
    CALL{
        with couple
        WITH couple[0] as aId, couple[1] as bId
        match (a:Customer {customerId:aId}), (b:Customer
{customerId:bId})
        MERGE (a)-[:BUY_FRIEND]-(b)
    }
} IN TRANSACTIONS OF 16000 ROWS
```

3.6. Query *e*

In maniera simile alla query *c* per trovare i *buying-friend* di grado *k* nella query va inserito *k-1* (nell’esempio si cercano i *buying-friend* di grado 4, settato quindi come parametro 3)

```
// Identify the buying-friend of degree K
match (c1:Customer)-[:BUY_FRIEND*3]-(c2:Customer)
where c1.customerId < c2.customerId
return distinct [c1.customerId, c2.customerId] as couple
order by couple[0], couple[1];
```

4. Performance

Come si evince dal grafico 1 i tempi richiesti per l'esecuzione delle query crescono al crescere dei dataset. Mantenendo le transazioni come archi tra i nodi del grafo si è riusciti a mantenere dei tempi bassi per quanto riguarda le query *Qc* e *Qe*. La query che ha sofferto di più per questa scelta è stata senza dubbio la *Qd_i1*+*Qd_i2*. Ad influenzare molto la prestazione è sicuramente anche il numero delle transazioni presenti nel dataset (si arriva a circa 7 milioni in quello Large), inevitabilmente sono tutte da iterare per assegnare i valori richiesti. La scelta di sacrificare le prestazioni durante l'esecuzione della *Qd_i* deriva dal fatto che in uno scenario applicativo reale una tale operazione venga svolta una singola volta per un possibile "upgrade" del modello. Avvenuta l'operazione si ipotizza che da quel momento in poi tutte le nuove transazioni inserite nel dataset siano già fornite degli attributi che la query *Qd_i* ha lo scopo di introdurre. Le query *Qc* e *Qe* invece andranno ricalcolate, ogni volta che si vogliono ottenere i *co-customer* o i *buying friend*. La tesi quindi a supporto di questa scelta consiste nel sostenere il costo di una *Qd_i* una volta per evitarsi i costi di *Qc* e *Qe* ogni volta che si desidera ottenere i loro risultati.

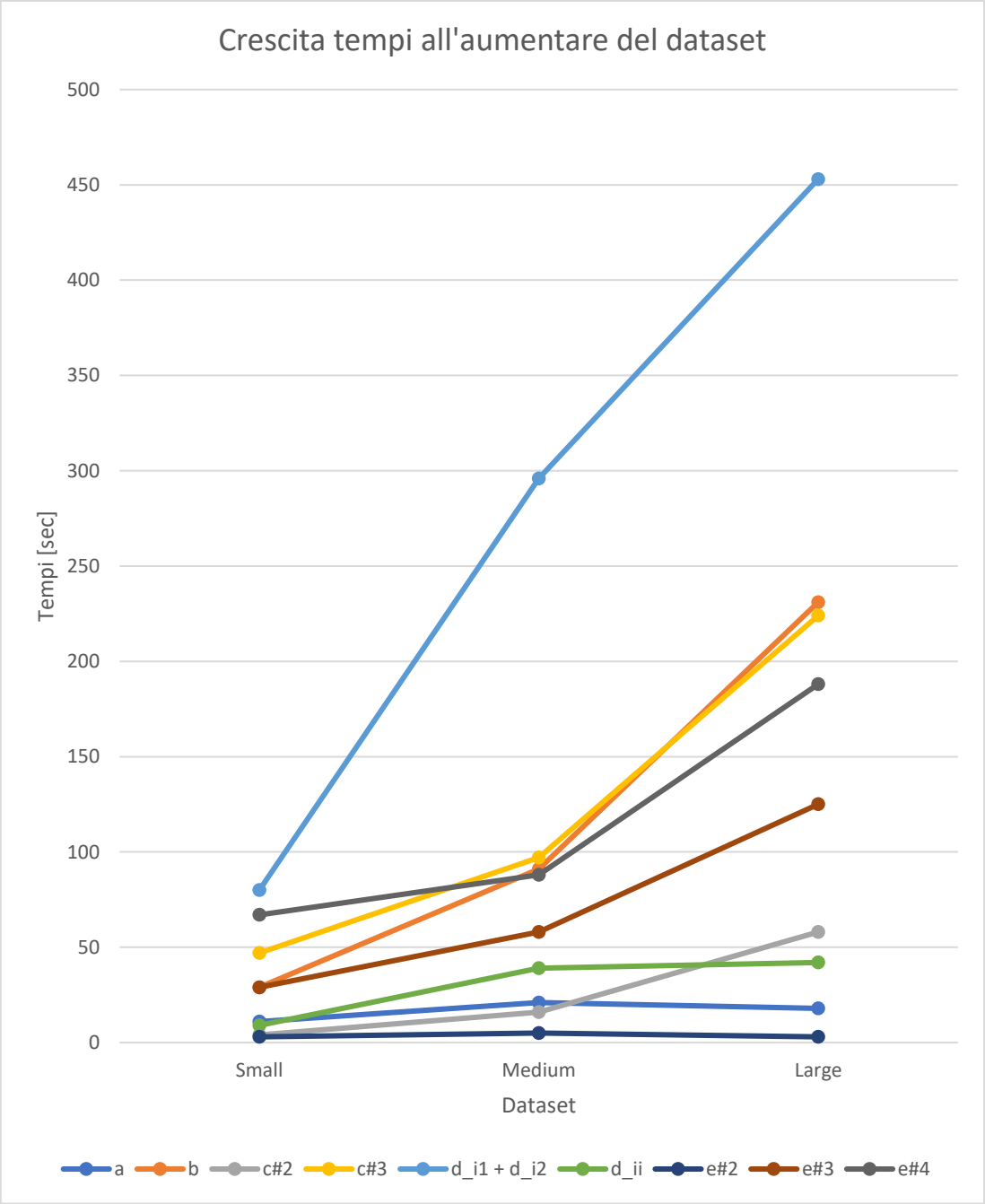
Crescita tempi all'aumentare del dataset
500
450
400
350
300
250
200
150
100
50
0
Tempi [sec]
Small
Medium
Large
Dataset
a
b
c#2
c#3
d_i1 + d_i2
d_ii
e#2
e#3
e#4

Grafico 1