Esercitazione di preparazione all’esame per il corso di “Data Base & Data Analytics”

(SQL, SQLAlchemy Core e ORM)

Schema di riferimento

Tutti gli esercizi saranno relativi allo schema “**prodotti**”, le cui tabelle sono elencate sotto:

MAGAZZINO (codM, indirizzo, citta)
PRODOTTO (codP, nome, categoria)
INVENTARIO (magazzino, prodotto, quantita, prezzo)

Note

In MAGAZZINO non sono ammessi valori NULL sugli attributi indirizzo e citta

In PRODOTTO non sono ammessi valori NULL sugli attributi nome e categoria

In INVENTARIO:
- magazzino è una chiave esterna che fa riferimento alla tabella MAGAZZINO
- prodotto è una chiave esterna che fa riferimento alla tabella PRODOTTO
- quantita deve essere > 0 e non può assumere valori NULL
- prezzo deve essere > 0.00 e non può assumere valori NULL

Parte su SQL

Implementare in SQL le query relative ai seguenti quesiti:

1. Inserire nella base di dati i seguenti prodotti:

   (codP='P33',nome='TV Led 55',categoria='elettrodomestici')
   (codP='P15',nome='Laptop i9',categoria='informatica')
   (codP='P02',nome='Condizionatore G4',categoria='elettrodomestici')
   (codP='P54',nome='Docking station',categoria='informatica')

   Aggiungere inoltre nell'inventario la disponibilità dei prodotti come segue:

   (magazzino='M01', prodotto='P33', quantita=10, prezzo=499,00)
   (magazzino='M05', prodotto='P33', quantita=5, prezzo=499,00)
   (magazzino='M11', prodotto='P15', quantita=8, prezzo=649,00)
   (magazzino='M17', prodotto='P02', quantita=4, prezzo=287,00)

In entrambi i casi si usa l'istruzione INSERT di SQL.

NB Inserire anche il prodotto ('P27', 'burro', 'alimentari') per mettere in evidenza (nelle query successive) il fatto che occorre tener presente anche di prodotti che non sono ancora presenti in alcun magazzino.

2. Aumentare poi il prezzo degli elettrodomestici presenti in inventario del 2.5%.

Si usa l'istruzione UPDATE di SQL.

3. Selezionare tutti i prodotti presenti nel magazzino 'M05' in quantità maggiore di 30.

La tabella interessata è 'inventario'.

4. Selezionare la lista dei prodotti disponibili al magazzino 'M01', ordinandoli per categoria e per prodotto (in entrambi i casi con ordine alfabetico ascendente).

La tabella interessata è 'inventario'.

5. Selezionare tutti i prodotti che al momento non sono presenti in alcun magazzino.

Ci sono molti modi per risolvere. Per esempio con subquery o con differenza di insiemi.

6. Selezionare la lista dei prodotti di categoria 'elettrodomestici' che non sono presenti in qualche magazzino (per ognuno di tali prodotti vanno indicati i magazzini che ne sono sprovvisti).

Ci sono molti modi per risolvere. Per esempio con subquery o con differenza di insiemi.

7. Selezionare la lista dei prodotti che non sono disponibili in almeno 60 unità complessive (la verifica procede come segue: si sommano le quantità di ogni prodotto al variare dei magazzini e poi si selezionano quei prodotti che non raggiungono il valore imposto di 60 unità).

Va usato ‘group by’, unitamente alla funzione di aggregazione ‘sum’. Almeno in linea di principio occorre tener conto ANCHE di quei prodotti che non compaiono in alcun magazzino.

Parte SQLAlchemy Core

1. Illustrare con un semplice esempio come si possono sottoporre alla DBAPI di PostgreSQL, tramite SQLAlchemy Core, delle query SQL in formato testuale (a scelta dello studente: come stringa Python o come istanze della classe text).

Si crea la stringa (come testo ASCII o come istanza della classe ‘text’) e poi si effettua la query mediante ‘connection.execute.

2. Definire a livello di Core ‘magazzino’, ‘prodotto’ e ‘inventario’ (come istanze di Table, e riferite alle corrispondenti tabelle del database).

Vanno create istanze di Table, una per ogni tabella. Ogni tabella dovrà essere creata con riferimento alla corrispondente tabella del database e ai metadati.

3. Come esercizio 3 della parte SQL, però utilizzando SQLAlchemy Core.

Va usato select(…). La mappatura con l’istruzione SQL è immediata.

4. Come esercizio 4 della parte SQL, però utilizzando SQLAlchemy Core.

Va usato select(…). La mappatura con l’istruzione SQL è immediata.

5. Come esercizio 5 della parte SQL, però utilizzando SQLAlchemy Core.

Va usato select(…). La mappatura con l’istruzione SQL è immediata.

6. Come esercizio 6 della parte SQL, però utilizzando SQLAlchemy Core.

Va usato select(…). La mappatura con l’istruzione SQL è immediata.

7. Come esercizio 7 della parte SQL, però utilizzando SQLAlchemy Core.

Va usato select(…). Alcune operazioni (in particolare trattare anche i prodotti non presenti in alcun magazzino) possono essere eventualmente realizzate in Python a valle della query.

Parte SQLAlchemy ORM (senza relazioni)

1. Definire preliminarmente la classe BaseModel (da cui sarà derivata la classe Base) imponendo che esporti i seguenti servizi (come metodi invocabili):

   - def __init__(self):
     Contiene lo slot '__mapper__', che viene gestito da SQLAlchemy (impostare a None).

   - def __call__(self, *keys, dtype=tuple)
     Se dtype vale tuple restituisce i campi richiesti come tupla di valori. Se invece vale dict allora viene restituito un dizionario (cioé con i valori associati alle chiavi corrispondenti).

     NB Se non si specifica alcuna chiave allora s'intende che il metodo deve restituire TUTTI i valori (come tupla o come dict, secondo quanto specificato da dtype).

     Per esempio, immaginando di avere un'istanza della classe 'Prodotto' (con i campi 'codP', 'nome' e 'categoria'), referenziata dalla variabile *prodotto*:

     prodotto('codP', 'nome') --> ("P15", "Laptop i9") # dtype=tuple per default
     prodotto('codP', 'nome', dtype=dict) --> { 'codp': "P15", 'nome': "Laptop i9" }
     prodotto(dtype=dict) --> { 'codp': "P15", 'nome': "Laptop i9", 'categoria': "informatica" }

   - def describe(self)
     Stampa le info sulla tabella (nome ORM, nome tabella nel database e info sulle colonne).
     Bisogna premettere il decoratore @classmethod poiché si tratta di un metodo di classe.

   - def __str__(self)
     Restituisce una stringa che rappresenta una tupla di valori (tutte le chiavi).

     Suggerimento: Per implementare __str__ si può delegare il metodo __call__ passandogli tutti gli attributi (o nessuno) come argomento. Quanto restituito da __call__ va poi ovviamente convertito in stringa.

Non ci sono particolari problemi, eccetto per il metodo di classe 'describe', che va decorato con @classmethod. Mentre 'describe' lavora sulla classe, tutti gli altri sono metodi d'istanza. Per questo invocare Prodotto('P27', 'burro','alimentari') è ben diverso dall'invocare prodotto('codp','nome'). Infatti nel primo caso (almeno in linea di principio) si chiama la __init__ di Prodotto, mentre nel secondo caso si chiama la __call__ di Prodotto (assumendo ovviamente che 'prodotto' sia un'istanza di Prodotto). Naturalmente, in quest'ultimo caso, la __call__ viene ereditata dalla classe BaseModel.

2. Definire le classi ORM Magazzino, Prodotto, e Inventario. Tali classi sono da mettere (nell'ordine) in corrispondenza con le tabelle del database 'magazzino', 'prodotto' e 'inventario.

Come creare le classi ORM è già stato visto ripetutamente a lezione. Sono classi Python, con però slot e metodi principali "instrumented".

3. Come esercizio 3 della parte SQL, però utilizzando ORM.

E' il corrispondente ORM di una select su Inventario (si usa però 'session.query').

4. Come esercizio 4 della parte SQL, però utilizzando ORM.

E’ il corrispondente ORM di una select su Inventario (si usa però ‘session.query’).

5. Come esercizio 5 della parte SQL, però utilizzando ORM.

La soluzione può essere implementata in diversi modi. In particolare, evitando di utilizzare istruzioni del Core, mediante ORM si può lavorare di più a livello Python.

6. Come esercizio 6 della parte SQL, però utilizzando ORM.

La soluzione può essere implementata in diversi modi. In particolare, evitando di utilizzare istruzioni del Core, mediante ORM si può lavorare di più a livello Python.

7. Come esercizio 7 della parte SQL, però utilizzando ORM.

La soluzione può essere implementata in diversi modi. In particolare, evitando di utilizzare istruzioni del Core, mediante ORM si può lavorare di più a livello Python.

Parte SQLAlchemy ORM (addendum per relatioship)

2R. Con riferimento alle definizioni precedenti delle classi ORM Magazzino, Prodotto, e Inventario, aggiungere le seguenti relazioni (relationship):

class Magazzino:
lista_prodotti :: collegata a 'Prodotto.lista_magazzini'

class Prodotto:
lista_magazzini :: collegata a 'Magazzino.lista_prodotti'

NB Magazzino e Prodotto sono legati da una relazione molti-a-molti che si realizza tramite Inventario

Per implementare la soluzione occorre tener ben presente il fatto che la relazione molti-a-molti si realizza per tramite di Inventario (vedere il parametro **'secondary'** nella definizione di relationship).

5R. Come esercizio 5 della parte SQL, però utilizzando ORM e le relazioni.

Basta concentrarsi sullo slot Prodotto.lista_magazzini e selezionare i prodotti in cui questa lista è vuota.

6R. Come esercizio 6 della parte SQL, però utilizzando ORM e le relazioni.

Come partenza si usa Prodotto.lista_magazzini (ovviamente per i soli elettrodomestici). Poi occorre effettuare altre operazioni che possono essere svolte a livello di programma Python.

NB In questo caso è MOLTO più efficiente la soluzione SQL / SQL Core.

7R. Come esercizio 7 della parte SQL, però utilizzando ORM e le relazioni.

Si itera sui prodotti. Tramite l'inventario si recupera la quantità di ogni prodotto consultando soltanto i magazzini riportati nella lista_magazzini. Sommando le quantità a livello di programma Python si ottengono i totali di ogni prodotto. Ovviamente vengono selezionati soltanto i prodotti in cui il totale è inferiore alle 60 unità.

NB Anche in questo caso è MOLTO più efficiente la soluzione SQL / SQL Core.
