PROGETTAZIONE E SVILUPPO DI UNA BASE DI DATI RELAZIONALE PER LA GESTIONE DI CORSI DI FORMAZIONE

Roberto Ambrosino
roberto.ambrosino@studenti.unina.it
N86003873

Sadman Ahmed
s.ahmed@studenti.unina.it
N86003921

1 - Raccolta e analisi dei requisiti

1.1 Descrizione e analisi del problema

Si sviluppi un sistema informativo, composto da una base di dati relazionale, per la gestione di corsi di formazione.

Il sistema permette ad un operatore di gestire corsi di formazione.

Un corso è caratterizzato da un nome, una descrizione, un tasso di presenze minimo necessario (e.g.: 75%), un numero massimo di partecipanti, e da una o più lezioni.

Ciascuna lezione è caratterizzata da un titolo, una descrizione, una durata (espressa in ore), e una data e orario di inizio.

I corsi possono inoltre essere organizzati in aree tematiche definibili dagli operatori, e un corso può appartenere a più aree tematiche.

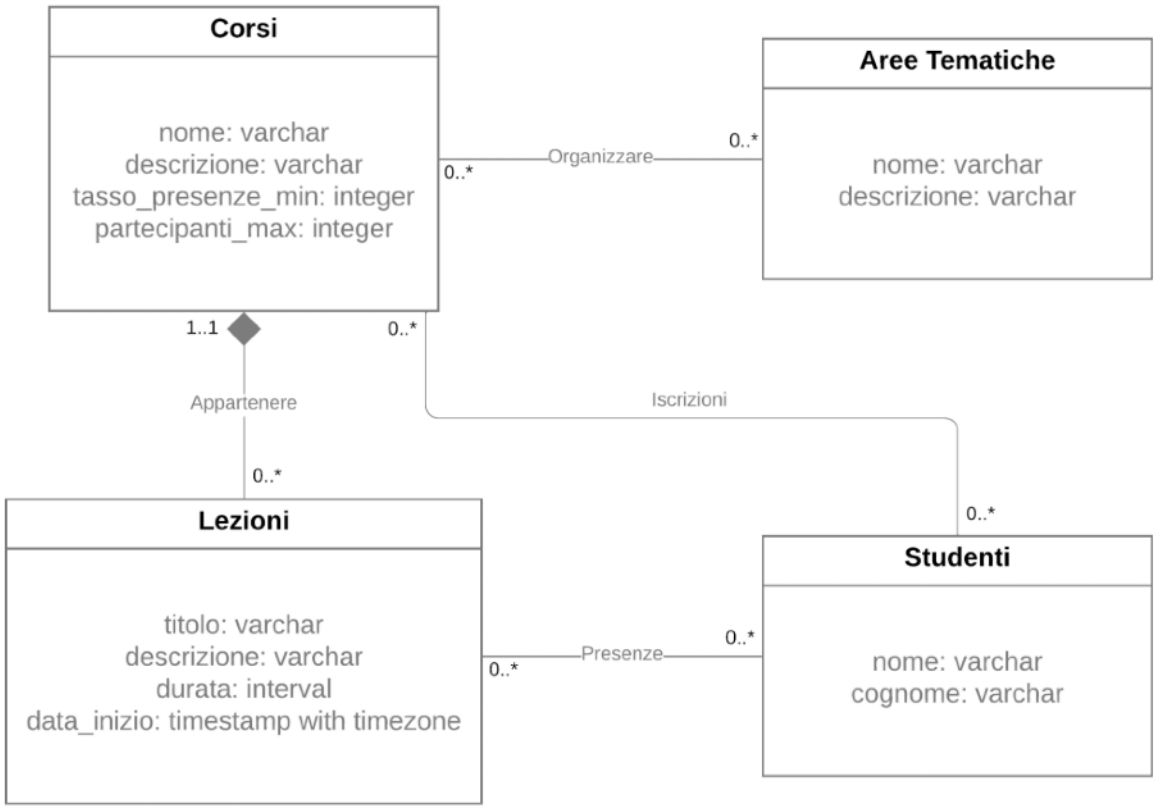
Gli operatori possono anche iscrivere studenti ai corsi e, per ogni lezione, tenere traccia delle presenze/assenze degli studenti iscritti

2 - PROGETTAZIONE CONCETTUALE

2.1 Introduzione

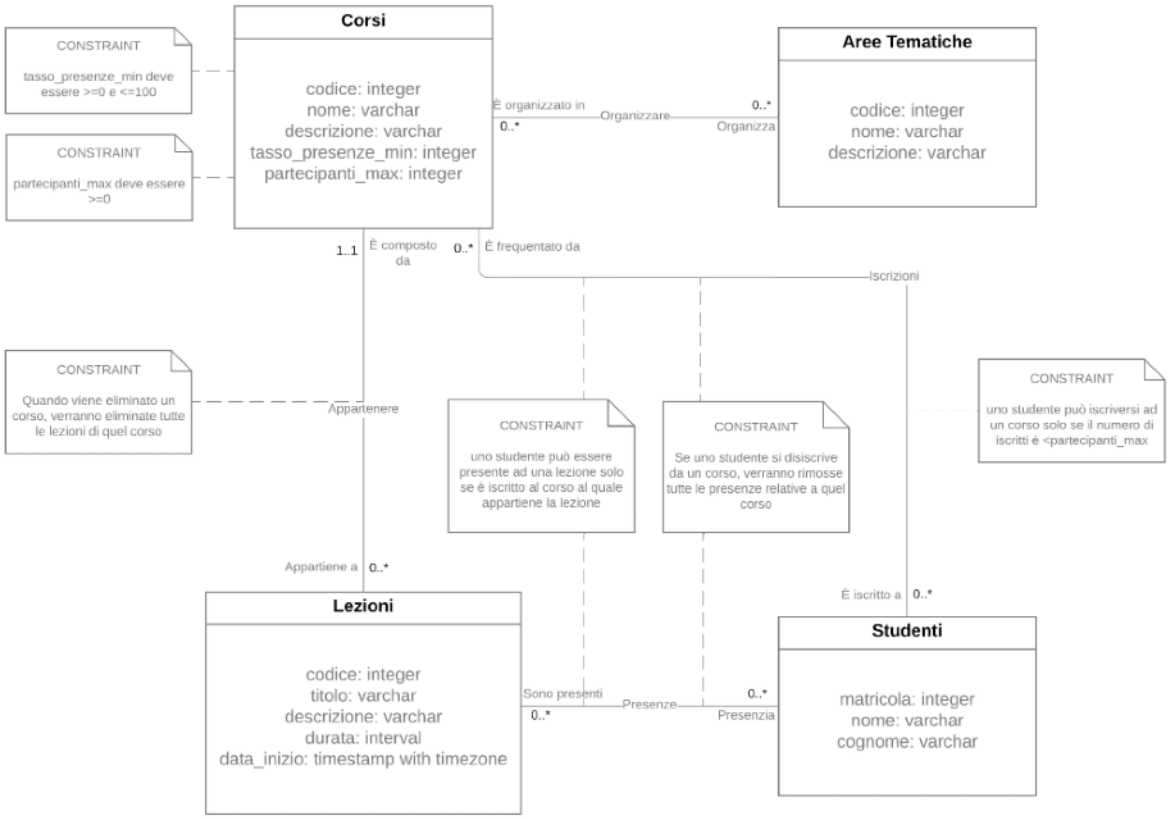
A partire dai requisiti forniti nella fase precedente, si modellano delle descrizioni formalizzate come Class Diagram UML.

2.2 Class Diagram



2.3 Class Diagram ristrutturato

Prima di procedere alla traduzione del Class Diagram in uno schema logico relazionale è necessario effettuare una sua ristrutturazione con lo scopo di semplificarne la traduzione. Un Class Diagram ristrutturato risulta "meno concettuale", perché tiene già conto del modello logico adottato (e quindi dei dettagli implementativi). Nell'attività di ristrutturazione è necessario quindi semplificare lo schema, eliminando le gerarchie e gli attributi multipli, ed effettuare un partizionamento o accorpamento di entità e relazioni.



Nella ristrutturazione del Class Diagram della fase precedente, non sono emerse particolari attività da svolgere.

Sono state definite le chiavi primarie di ogni tabella, che ci permetteranno di identificare univocamente una tupla nel nostro DBMS relazionale, e i vincoli, che verranno elencati nell'apposito dizionario, che garantiranno la consistenza dei dati memorizzati nel nostro database.

2.4 Dizionario delle classi

Nome classe	Descrizione	Attributi
Aree Tematiche	Classe atta a descrivere le aree tematiche definite per suddividere i corsi.	• Codice (integer): chiave primaria. Identifica univocamente ogni istanza. • Nome (varchar): nome dell'area tematica • Descrizione (varchar): descrizione dell'area tematica
Corsi	Classe atta a definire i corsi	• Codice (integer): chiave primaria. Identifica univocamente ogni istanza. • Nome (varchar): nome del corso • Descrizione (varchar): descrizione del corso • Tasso_presenze_min (integer): un tasso di presenze minimo necessario per poter superare il corso • partecipanti_max (integer): il numero massimo di studenti iscrivibili al corso

Nome classe	Descrizione	Attributi
Lezioni	Classe atta a definire le lezioni dei corsi	• Codice (integer): chiave primaria. Identifica univocamente ogni istanza. • titolo (varchar): titolo della lezione • Descrizione (varchar): descrizione della lezione • durata (interval) durata della lezione • data_inizio (timestamp with timezone): data e ora di inizio della lezione
Studenti	Classe atta a definire gli studenti, che potranno essere iscritti ai corsi e frequentarne le lezioni	• Matricola (integer): chiave primaria. Identifica univocamente ogni istanza. • Nome (varchar): nome dell'area dello studente • Cognome (varchar): cognome dello studente

2.5 Dizionario delle associazioni

Nome associazione	Descrizione	Classi coinvolte
Organizzare	Esprime la suddivisione dei corsi in aree tematiche	• Corsi [0..*], ruolo: “È organizzato in” • Aree Tematiche [0..*], ruolo: “Organizza”
Appartenere	Esprime l’appartenenza delle lezioni ai propri corsi	• Corsi [1..1], ruolo: “È composto da” • Lezioni [0..*], ruolo: “Appartiene a”
Iscrizioni	Esprime gli studenti iscritti ai corsi	• Corsi [0..*], ruolo: “È frequentato da” • Studenti [0..*], ruolo: “È iscritto a”
Presenze	Esprime le presenze degli studenti alle lezioni dei corsi che frequentano	• Lezioni [0..*], ruolo: “Sono presenti” • Studenti [0..*], ruolo “Presenzia”

2.6 Dizionario dei vincoli

Nome vincolo	Tipo	Descrizione
controlla_disponibilita	Interrelazionale	Consente l'iscrizione ad un corso, da parte di uno studente, solo se il numero di iscritti è < di corsi.partecipanti_max, attributo che indica il numero massimo di partecipanti di un corso
controlla_iscrizione	Interrelazionale	Consente di registrare la presenza di uno studente ad una lezione, solo se lo studente è iscritto al corso al quale appartiene la lezione
rimuovi_presenze	Interrelazionale	Se uno studente si disiscrive da un corso, verranno rimosse tutte le presenze registrate relative a quel corso
controlla_partecipanti_max	Interrelazionale	Se i dati di un corso vengono aggiornati, controlla che il nuovo valore di corsi.partecipanti_max non sia inferiore al numero attuale di studenti iscritti al corso
controllo_tasso_presenze	Intrarelazionale	L'attributo corsi.tasso_presenze_min deve essere >=0 e <=100
controllo_partecipanti_max	Intrarelazionale	L'attributo corsi.partecipanti_max deve essere >=0
vieta_aggiornamento_presenza	Intrarelazionale	Ogni aggiornamento sulla tabella delle presenze è vietato

vieta_aggiornamento_iscrizione	Intrarelazionale	Ogni aggiornamento sulla tabella delle iscrizioni è vietato
vieta_aggiornamento_corso_della_lezione	Intrarelazionale	Non è possibile cambiare corso ad una lezione

3 - Schema logico

Dal modello concettuale vengono derivate le tabelle che rappresentano le entità all'interno del nostro DBMS relazionale.

Le associazioni "uno a uno" si traducono in base alle interrogazioni che si eseguiranno sulle tabelle. L'associazione può essere tradotta scambiando le chiavi delle due tabelle, inserendo solo la chiave di una tabella nell'altra, oppure, se opportuno, si può pensare di unire le due entità in un'unica tabella.

Le associazioni "uno a molti" vengono tradotte inserendo la chiave dell'entità che ha il ruolo "a uno" nella tabella dell'entità che ha il ruolo "a molti".

Le associazioni "molti a molti" vengono tradotte introducendo una terza tabella contenente le chiavi delle due entità e gli eventuali attributi dell'associazione.

Aree Tematiche	(codice, nome, descrizione)
Corsi	(codice, nome, descrizione, tasso_presenze_min, partecipanti_max)
Lezioni	(codice, titolo, descrizione, durata, data_inizio, codice_corso) codice_corso → Corsi.codice
Studenti	(matricola, nome, cognome)
Aree_dei_corsi	(codice_area_tematica, codice_corso) codice_area_tematica → Aree Tematiche.codice codice_corso → Corsi.codice

Iscrizioni	(matricola, codice_corso) matricola → Studenti.matricola codice_corso → Corsi.codice
Presenze	(matricola, codice_lezione) matricola → Studenti.matricola codice_lezione → Lezioni.codice

4 - Implementazione

Di seguito saranno inseriti gli script SQL necessari ad implementare la nostra base di dati relazionale utilizzando il DBMS PostgreSQL

4.1 Tabelle

```
/*Creazione tabella corsi*/
CREATE TABLE corsi (
   /*definiziione attributi*/
   codice INTEGER GENERATED ALWAYS AS IDENTITY
                           (START WITH 0
                            INCREMENT BY 1
                            MINVALUE 0),
   nome VARCHAR (200) NOT NULL,
   descrizione VARCHAR (200) NOT NULL,
   tasso_presenze_min INTEGER NOT NULL,
   partecipanti_max INTEGER NOT NULL,

   /*Vincolo di chiave primaria*/
   CONSTRAINT pk_corso PRIMARY KEY (codice),

   /*L'attributo corsi.tasso_presenze_min deve essere >=0 e <=100*/
   CONSTRAINT controllo_tasso_presenze
   CHECK (tasso_presenze_min >= 0 AND tasso_presenze_min <=100),

   /*L'attributo corsi.partecipanti_max deve essere >=0*/
   CONSTRAINT controllo_partecipanti_max
   CHECK (partecipanti_max >= 0)
);

/*Creazione tabella aree tematiche*/
CREATE TABLE aree_tematiche (
   /*definizione attributi*/
   codice INTEGER GENERATED ALWAYS AS IDENTITY
                           (START WITH 0
                            INCREMENT BY 1
                            MINVALUE 0),
   nome VARCHAR (200) NOT NULL,
   descrizione VARCHAR (200) NOT NULL,

   /*vincolo di chiave primaria*/
   CONSTRAINT pk_area_tematica PRIMARY KEY (codice)
);
```

```sql
/*creazione tabella lezioni*/
CREATE TABLE lezioni (
   /*definizione attributi*/
   codice INTEGER GENERATED ALWAYS AS IDENTITY
                          (START WITH 0
                           INCREMENT BY 1
                           MINVALUE 0),
   titolo VARCHAR (200) NOT NULL,
   descrizione VARCHAR (300) NOT NULL,
   durata INTERVAL NOT NULL,
   data_inizio TIMESTAMP WITH TIME ZONE NOT NULL,
   codice_corso INTEGER NOT NULL,

   /*vincolo di chiave primaria*/
   CONSTRAINT pk_lezione PRIMARY KEY (codice),

   /*vincolo di chiave esterna necessario per effettuare
   l'associazione con la tabella corsi. Se il corso a cui
   si riferisce la chiave esterna venisse eliminato, allora
   anche la lezione che gli è associata verrebbe eliminata*/
   CONSTRAINT fk_corso_della_lezione FOREIGN KEY (codice_corso)
   REFERENCES corsi (codice)
   ON DELETE CASCADE
   ON UPDATE CASCADE
);

/*creazione tabella studenti*/
CREATE TABLE studenti (
   /*definizione attributi*/
   matricola INTEGER GENERATED ALWAYS AS IDENTITY
                     (START WITH 0
                      INCREMENT BY 1
                      MINVALUE 0),
   nome VARCHAR (200) NOT NULL,
   cognome VARCHAR (200) NOT  NULL,

   /*Vincolo di chiave primaria*/
   CONSTRAINT pk_studente PRIMARY KEY (matricola)
);
```

```sql
/*creazione tabella aree_dei_corsi, tabella nata dall'associazione
molti a molti che c'è tra corsi e aree tematiche*/
CREATE TABLE aree_dei_corsi (
   codice_area_tematica INTEGER NOT NULL,
   codice_corso INTEGER NOT NULL,

   /*Vincolo di chiave esterna verso la tabella aree_tematiche.
   Alla cancellazione della corrispondente area tematica, verrebbe
   cancellata anche la riga che ne contiene la chiave*/
   CONSTRAINT fk_adc_codice_area_tematica FOREIGN KEY (codice_area_tematica)
   REFERENCES aree_tematiche (codice)
   ON DELETE CASCADE
   ON UPDATE CASCADE,

   /*Vincolo di chiave esterna verso la tabella corsi.
   Alla cancellazione del corrispondente corso, verrebbe
   cancellata anche la riga che ne contiene la chiave*/
   CONSTRAINT fk_adc_codice_corso FOREIGN KEY (codice_corso)
   REFERENCES corsi (codice)
   ON DELETE CASCADE
   ON UPDATE CASCADE,

   /*vincolo di unicità per ogni riga della tabella, serve per
   evitare che non si ripeta la stessa associazione più volte*/
   CONSTRAINT unica_tupla_adc UNIQUE (codice_area_tematica, codice_corso)
);
```

```sql
/*creazione tabella iscrizioni, tabella nata dall'associazione
molti a molti che c'è tra studenti e corsi*/
CREATE TABLE iscrizioni (
   matricola INTEGER NOT NULL,
   codice_corso INTEGER NOT NULL,

   /*Vincolo di chiave esterna verso la tabella studenti.
   Alla cancellazione del corrispondente studente, verrebbe
   cancellata anche la riga che ne contiene la chiave*/
   CONSTRAINT fk_iscrizioni_matricola FOREIGN KEY (matricola)
   REFERENCES studenti (matricola)
   ON DELETE CASCADE
   ON UPDATE CASCADE,

   /*Vincolo di chiave esterna verso la tabella corsi.
   Alla cancellazione del corrispondente corso, verrebbe
   cancellata anche la riga che ne contiene la chiave*/
   CONSTRAINT fk_iscrizioni_codice_corso FOREIGN KEY (codice_corso)
   REFERENCES corsi (codice)
   ON DELETE CASCADE
   ON UPDATE CASCADE,

   /*vincolo di unicità per ogni riga della tabella, serve per
   evitare che non si ripeta la stessa associazione più volte*/
   CONSTRAINT unica_tupla_iscrizioni UNIQUE (matricola, codice_corso)
);
```

```sql
/*creazione tabella presenze, tabella nata dall'associazione
molti a molti che c'è tra studenti e lezioni*/
CREATE TABLE presenze (
   matricola INTEGER NOT NULL,
   codice_lezione INTEGER NOT NULL,

   /*Vincolo di chiave esterna verso la tabella studenti.
   Alla cancellazione del corrispondente studente, verrebbe
   cancellata anche la riga che ne contiene la chiave*/
   CONSTRAINT fk_presenze_matricola FOREIGN KEY (matricola)
   REFERENCES studenti (matricola)
   ON DELETE CASCADE
   ON UPDATE CASCADE,

   /*Vincolo di chiave esterna verso la tabella lezioni.
   Alla cancellazione della corrispondente lezione, verrebbe
   cancellata anche la riga che ne contiene la chiave*/
   CONSTRAINT fk_presenze_codice_lezione FOREIGN KEY (codice_lezione)
   REFERENCES lezioni (codice)
   ON DELETE CASCADE
   ON UPDATE CASCADE,

   /*vincolo di unicità per ogni riga della tabella, serve per
   evitare che non si ripeta la stessa associazione più volte*/
   CONSTRAINT unica_tupla_presenze UNIQUE (matricola, codice_lezione)
);
```

4.2 Vincoli

```
/*trigger che si attiva ad ogni insert
sulla tabella iscrizioni*/
CREATE OR REPLACE TRIGGER nuova_iscrizione
BEFORE INSERT
ON iscrizioni
FOR EACH ROW
EXECUTE FUNCTION controlla_disponibilita();

/*il trigger precedente chiama la function controlla_disponibilità,
che controlla, ad ogni chimata, se il numero di iscritti al corso al
quale si vuole iscrivere un nuovo studente è inferiore al numero
massimo di partecipanti di quel corso. L'inser viene autorizzato solo
se il numero di iscritti di un corso è inferiore al numero massimo
di iscritti di quel corso, altrimenti viene sollevata un'eccezione*/
CREATE OR REPLACE FUNCTION controlla_disponibilita() RETURNS TRIGGER AS
$$
BEGIN
   IF (SELECT count(*)
       FROM iscrizioni
       WHERE codice_corso = NEW.codice_corso) < (SELECT partecipanti_max
                                                 FROM corsi
                                                 WHERE codice =
NEW.codice_corso)
   THEN
       RETURN NEW;
   ELSE
       RAISE EXCEPTION 'Posti liberi insufficienti';
   END IF;
END;
$$ LANGUAGE plpgsql
```

```
/*trigger che si attiva ad ogni insert
sulla tabella presenze*/
CREATE OR REPLACE TRIGGER nuova_presenza
BEFORE INSERT
ON presenze
FOR EACH ROW
EXECUTE FUNCTION controlla_iscrizione();

/*il trigger precedente chiama la function controlla_iscrizione, che controlla
l'iscrizione dello studente del quale si vuole inserire una nuova presenza, al
corso al quale appartiene la lezione. Questa verifica viene fatta verificando
che
esista almeno una riga nella tabella iscrizioni, che abbia come matricola la
matricola
dello studente di cui vogliamo inserire la presenza e che abbia come
codice_corso
il codice_corso della lezione della quale stiamo inserendo una presenza*/
CREATE OR REPLACE FUNCTION controlla_iscrizione() RETURNS TRIGGER AS
$$
BEGIN
   IF (SELECT COUNT(*)
       FROM iscrizioni
       WHERE matricola = NEW.matricola AND codice_corso = (SELECT codice_corso
                                                           FROM lezioni
                                                           WHERE codice =
NEW.codice_lezione)) > 0
   THEN
       RETURN NEW;
   ELSE
       RAISE EXCEPTION 'Studente non iscritto al corso della lezione';
   END IF;
END;
$$ LANGUAGE plpgsql
```

```
/*trigger che si attiva ad ogni delete
sulla tabella iscrizioni*/
CREATE OR REPLACE TRIGGER disiscrizione
BEFORE DELETE
ON iscrizioni
FOR EACH ROW
EXECUTE FUNCTION rimuovi_presenze();

/*il trigger precedente chiama la function rimuovi_presenze, che
ha lo scopo di rimuovere tutte le presenze degli studenti presenti
alle lezioni del corso appena eliminato*/
CREATE OR REPLACE FUNCTION rimuovi_presenze() RETURNS TRIGGER AS
$$
BEGIN
   DELETE
   FROM presenze
   WHERE matricola = OLD.matricola AND codice_lezione IN (SELECT codice
                                                          FROM lezioni
                                                          WHERE codice_corso =
OLD.codice_corso);
   RETURN OLD;
END;
$$ LANGUAGE plpgsql


/*trigger che si attiva ad ogni update
sulla tabella corso*/
CREATE OR REPLACE TRIGGER update_corso
BEFORE UPDATE
ON corsi
FOR EACH ROW
EXECUTE FUNCTION controlla_partecipanti_max();
/*il trigger precedente chiama la function controlla_partecipanti_max, che
controlla che il nuovo valore del campo partecipanti_max sia uguale o superiore
del numero di studenti iscritti presenti nella tabella iscrizioni. Se il nuovo
valore fosse inferiore al numero di iscritti, verrebbe sollevata un eccezione
e verrebbe bloccato l'update*/
CREATE OR REPLACE FUNCTION controlla_partecipanti_max() RETURNS TRIGGER AS
$$
BEGIN
   IF (SELECT COUNT(*)
       FROM iscrizioni
       WHERE codice_corso = NEW.codice) > NEW.partecipanti_max
   THEN
       RAISE EXCEPTION 'Nuovo numero massimo di partecipanti inferiore al numero
di partecipanti attuali';
   ELSE
       RETURN NEW;
   END IF;
END;
$$ LANGUAGE plpgsql
```

```
/*trigger che si attiva ad ogni update
sulla tabella presenze*/
CREATE OR REPLACE TRIGGER update_presenza
BEFORE UPDATE
ON presenze
FOR EACH ROW
EXECUTE FUNCTION vieta_aggiornamento_presenza();

/*il trigger precedente chiama la function vieta_aggiornamento_presenza, che
impedirà di aggiornare le righe della tabella presenze, per evitare
che si possa marcare la presenza di uno studente ad una lezione di un corso
che non frequenta*/
CREATE OR REPLACE FUNCTION vieta_aggiornamento_presenza() RETURNS TRIGGER AS
$$
BEGIN
   RAISE EXCEPTION 'Impossibile aggiornare una riga della tabella presenze.';
END;
$$ LANGUAGE plpgsql



/*trigger che si attiva ad ogni update
sulla tabella iscrizioni*/
CREATE OR REPLACE TRIGGER update_iscrizione
BEFORE UPDATE
ON iscrizioni
FOR EACH ROW
EXECUTE FUNCTION vieta_aggiornamento_iscrizione();

/*il trigger precedente chiama la function vieta_aggiornamento_iscrizioni, che
impedirà di aggiornare le righe della tabella iscrizioni, per evitare
che si possa iscrivere uno studente ad un corso al completo*/
CREATE OR REPLACE FUNCTION vieta_aggiornamento_iscrizione() RETURNS TRIGGER AS
$$
BEGIN
   RAISE EXCEPTION 'Impossibile aggiornare una riga della tabella iscrizioni.';
END;
$$ LANGUAGE plpgsql



/*trigger che si attiva ad ogni update
sulla tabella lezioni*/
CREATE OR REPLACE TRIGGER update_lezione
BEFORE UPDATE
ON lezioni
FOR EACH ROW
EXECUTE FUNCTION vieta_aggiornamento_corso_della_lezione();
```

```
/*il trigger precedente chiama la function
vieta_aggiornamento_corso_della_lezione,
che impedirà di aggiornare il corso di una lezione per evitare che una lezione
possa essere frequentata da studenti non iscritti a quel corso*/
CREATE OR REPLACE FUNCTION vieta_aggiornamento_corso_della_lezione() RETURNS
TRIGGER AS
$$
BEGIN
   IF (NEW.codice_corso) <> (OLD.codice_corso)
   THEN
       RAISE EXCEPTION 'Impossibile cambiare il corso di una lezione.';
   ELSE
       RETURN NEW;
   END IF;
END;
$$ LANGUAGE plpgsql
```

4.3 Popolazione

```
INSERT INTO corsi (nome, descrizione, tasso_presenze_min,
partecipanti_max) VALUES
('Informatica', 'Si studia informatica', 75, 5),
('Matematica', 'Si studia matematica', 50, 10),
('Medicina', 'Si studia medicina', 100, 2),
('Giurisprudenza', 'Si studia giurisprudenza', 0, 5);

INSERT INTO lezioni (titolo, descrizione, durata, data_inizio,
codice_corso) VALUES
('Lezione 0', 'Lezione di presentazione del corso di informatica',
'02:00:00', '2021-09-17 09:00:00', 0),
('Lezione 1', 'Programmazione', '02:00:00', '2021-09-18 09:00:00', 0),
('Lezione 1', 'Introduzione al corso di medicina', '02:00:00',
'2021-09-17 11:00:00', 2),
('Lezione 1', 'Prima lezione matematica', '02:00:00', '2021-09-17
16:00:00', 1),
('Lezione 1', 'Inizio lezioni giurisprudenza', '02:30:00', '2021-09-17
18:00:00', 3);

INSERT INTO aree_tematiche (nome, descrizione) VALUES
('Scienze', 'Corsi scientifici'),
('Legge', 'Corsi di giurisprudenza'),
('Medicina', 'Corsi di medicina');

INSERT INTO studenti (nome, cognome) VALUES
('Roberto', 'Ambrosino'),
('Sadman', 'Ahmed'),
('Mario', 'Rossi'),
('Lucia', 'Gialli');

INSERT INTO aree_dei_corsi (codice_area_tematica, codice_corso ) VALUES
(0, 0),
(0, 1),
(2, 2),
(1, 3);

INSERT INTO iscrizioni (matricola, codice_corso) VALUES
(0, 0),
(1, 0),
(2, 2),
(3, 2);
```

```sql
INSERT INTO presenze (matricola, codice_lezione) VALUES
(0, 0),
(0, 1),
(1, 0),
(1, 1),
(2, 2),
(3, 2);
```