UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

Progetto di Basi di Dati

CORSO DI LAUREA IN INFORMATICA
A.A. 2021/2022
PROFESSORE ADRIANO PERON
ID TRACCIA: TRACCIA 3

Lucia Brando N86003382
Enrico Amatore N86003572

Indice

1. Descrizione del problema

1.1. Traccia completa del problema

Si sviluppi un sistema informativo, composto da una base di dati relazionale e da un applicativo Java dotato di GUI (Swing o JavaFX), 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. Il sistema permette di effettuare interrogazioni avanzate sui corsi erogati, con possibilità di filtraggio per una o più categorie, per data, per parole chiave. Inoltre, il sistema permette di visualizzare, per ciascun corso, statistiche sul tasso di frequenza (e.g.: numero medio, minimo e massimo di studenti a lezione, percentuale di riempimento media) e un prospetto dove sono evidenziati gli studenti che hanno ottenuto il numero minimo di presenze e sono quindi idonei al superamento del corso.
Per i gruppi composti da 3 membri: Il sistema permette, per ciascun corso, l'inserimento di zero o più test di valutazione. Un test, come una lezione, è caratterizzato da un titolo, una descrizione, una durata (espressa in ore), e una data e orario di inizio. Un test di valutazione non può sovrapporsi a una lezione dello stesso corso. Inoltre, ogni test ha un punteggio minimo (espresso in

centesimi) per il superamento. Per ciascun test, un operatore può indicare, per ciascuno studente iscritto al corso, il punteggio ottenuto. Inoltre, il sistema include un prospetto finale dove sono evidenziati tutti gli studenti che hanno superato il corso, ovvero che hanno ottenuto il numero minimo di presenze e almeno il punteggio minimo a tutti i test.

1.2. Descrizione breve del problema ed analisi della soluzione

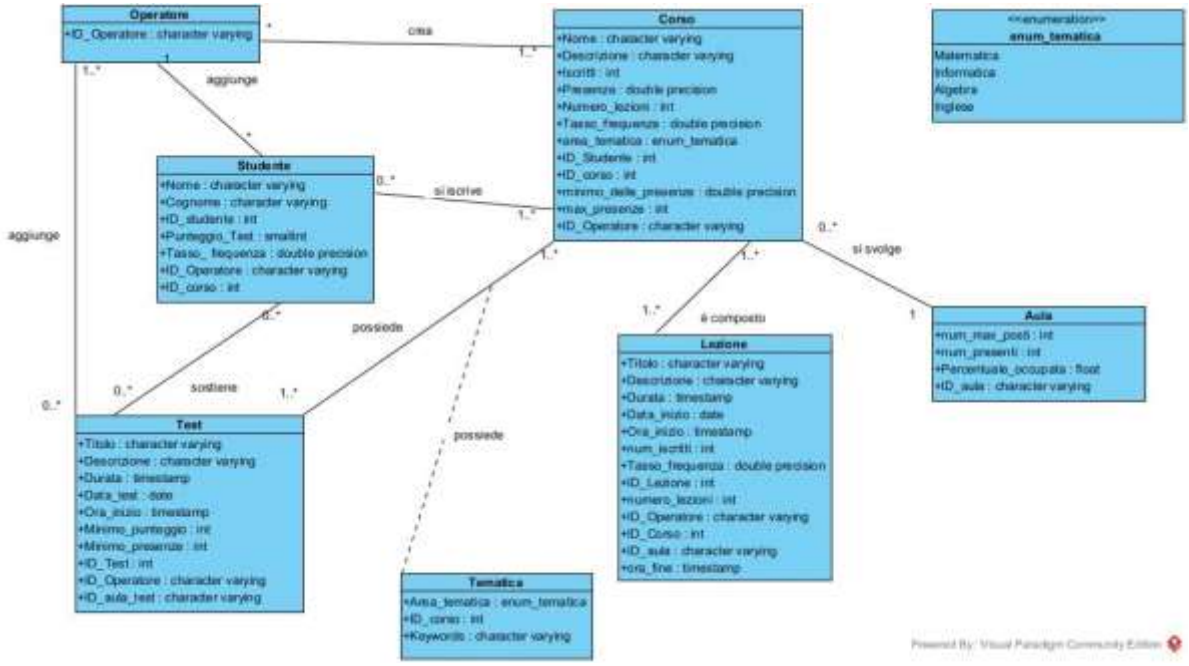
In questo capitolo inizia la progettazione della base di dati al livello di astrazione più alto. Dall'analisi dei requisiti che devono essere soddisfatti, si arriverà ad uno schema concettuale indipendente dalla struttura dei dati e dall'implementazione fisica: in tale schema concettuale verrà rappresentato un **Class Diagram** realizzato tramite il programma **Visual Paradigm** in cui si evidenzieranno le entità rilevanti ai fini della rappresentazione dei dati e le relazioni che intercorrono tra esse; si delineeranno anche eventuali vincoli da imporre. Il tutto verrà poi trasportato su **PgAdmin4 (PostgreSQL)** completandone anche la sua progettazione fisica.
Si implementerà una base di dati relazionale per un sistema informativo per la gestione di corsi di formazione. La base di dati permetterà ad un operatore di creare un nuovo corso e di aggiungere a quest'ultimo gli studenti che decideranno di iscrivervi. (Ricordando che un singolo studente può seguire un solo corso).
Ogni corso, in totale quattro, è svolto in un'aula ed è composto da lezioni che possono essere seguite dagli studenti.
Raggiunto un certo numero di presenze lo studente non solo può considerare il corso come "superato" ma, nonostante ciò, deve svolgere i test ad esso legato. Nel caso in cui il punteggio dei test siano superiori e/o uguali al minimo e il numero minimo di presenze sia stato soddisfatto, lo studente avrà concluso con successo il corso.
Le operazioni da svolgere sono:

1. Aggiungere gli studenti al corso scelto
2. Verificare che ogni aula non contiene più studenti del numero max consentito
3. Verificare che l'aula che si vuole prenotare sia ancora disponibile e nel caso, permettere l'inserimento e la conferma della prenotazione all'interno del sistema.
4. Verificare che lo studente raggiunga il numero minimo di presenze necessarie sia per il superamento del corso che per il sostenimento del test di valutazione
5. Stabilire un numero max e minimo di presenze che un corso, o una lezione, deve avere.

6. Aggiungere lezioni e test al corso
7. Verificare il punteggio del test sia maggiore-uguale al minimo necessario per il suo superamento.
8. Includere all'interno del sistema un prospetto finale dove sono evidenziati tutti gli studenti che hanno superato il corso, ovvero che hanno ottenuto il numero minimo di presenze e almeno il punteggio minimo a tutti i test.

2. Progettazione concettuale

2.1. Class Diagram[1]



2.2. Sintesi revisione del Class Diagram

Per poter rendere il Class Diagram idoneo alla traduzione in schemi relazionali si procede con la sua revisione.

Non sono state apportate grandi modifiche al Class Diagram di partenza.

Si è deciso di creare un'unica associazione che potesse andare a collegare la classe **Corso** con:

[1] Di seguito il link dell'immagine del Class Diagram caricata su GitHub per permettere una visione migliore: https://github.com/lbrando/Progetto-OO-BD-2021-2022/blob/main/BasiDiDati/class%20diagram%20finale.jpg

- **Operatore**, essendo colui che crea i corsi
- **Test**, in quanto sono compresi all'interno del corso
- **Studente**, in quanto sostiene i test, segue i corsi ma viene aggiunto dall'operatore nel sistema

Sono state eliminate tutte le cardinalità o..* dal diagramma e sono state create altre tre associazioni che riguardassero sempre la classe **Corso**:

- **Lezione**, in quanto il corso è composto da un determinato numero di lezioni
- **Tematica**, in quanto ogni corso affronta un'area tematica ben distinta: Algebra, Matematica, Inglese ed Informatica. (In un primo momento si era deciso dovesse essere una classe di associazione ma si è optato nel cambiarla nella successiva revisione del diagramma)
- **Aula**, luogo in cui si svolgono i corsi.

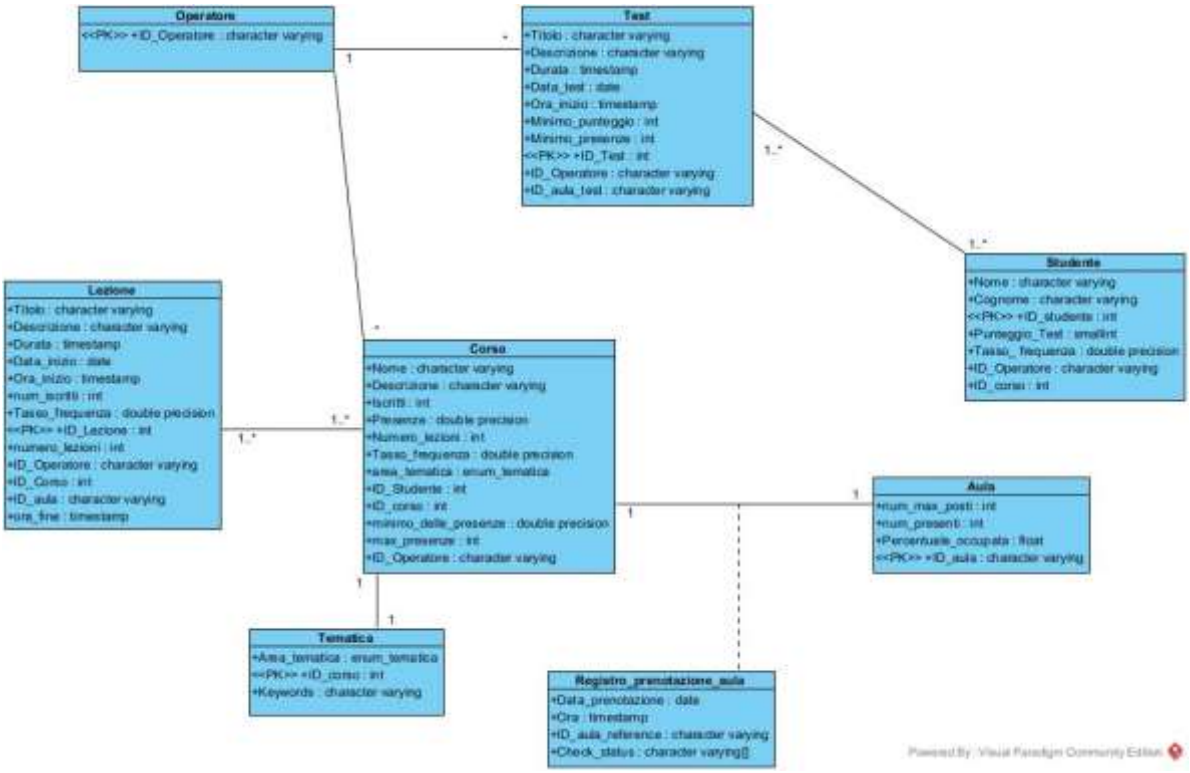
Per rendere ancora più pratica la progettazione del sistema si è optato nell'aggiungere una classe di associazione **Registro_prenotazione_aula** che ha il ruolo di un registro elettronico nel quale è possibile verificare la disponibilità (controllando data e ora) di un'aula cosi da poterla prenotare per sostenere una lezione di un corso o in vista di un test.

La ristrutturazione completa del diagramma ha portato, come c'era d'aspettarsi, alla necessità di aggiungere **chiavi primarie, esterne, vincoli di unicità e di check** (che verranno affrontati nel capitolo successivo).[2]

2.3. Class Diagram revisionato[3]

[2] Come verrà detto nel capitolo successivo, all'interno del Class Diagram ristrutturato sono state inserite solo le chiavi primarie in quanto Visual Paradigm non permetteva l'aggiunta di altro. In seguito verranno indicate chiavi esterne, vincoli di check e di unicità insieme alla dicitura che è stata scelta per loro all'interno della documentazione.

[3] Di seguito il link dell'immagine del Class Diagram ristrutturato caricata su GitHub per permettere una visione migliore: https://github.com/lbrando/Progetto-OO-BD-2021-2022/blob/main/BasiDiDati/class%20diagram%20ristrutturato%20finale.jpg



2.4. Dizionario dei dati

2.4.1. Dizionario delle classi

Classe	Descrizione	Attributi
Operatore	Entità che viene utilizzata per la creazione di un nuovo corso e/o per l'inserimento di un nuovo studente all'interno di quest'ultimo	• **ID (character varying):** identificativo alfanumerico di qualsiasi operatore che va a lavorare sul sistema
Studente	Entità che conserva tutte le informazioni che sono inserite nel sistema	• **Nome (character varying):** nome dello studente • **Cognome (character varying):** cognome dello studente • **ID (integer):**

	riguardo gli studenti che si iscrivono ai corsi, frequentano le lezioni ed eseguono i test	identificativo numerico di qualsiasi studente che decide di iscriversi ad un corso • **Punteggio_test (smallint):** punteggio dello studente ottenuto al test • **Tasso_frequenza (double precision):** frequenza con la quale lo studente partecipa alle lezioni • **ID_Operatore (character varying):** indica da quale operatore lo studente è stato aggiunto al sistema • **ID_Corso (integer):** indica a quale corso lo studente è iscritto
Corso	Entità che viene utilizzata per conservare tutte le informazioni relative al corso che sono state inserite dall'operatore	• **Nome (character varying):** nome del corso • **Descrizione (character varying):** descrizione del corso • **Iscritti (integer):** numero di studenti che si sono iscritti • **Presenze (double precision):** numero di presenze che gli studenti hanno effettuato • **Numero_lezioni (int):** numero di lezioni di cui è composto il corso • **Tasso_frequenza (double precision):** frequenza con la quale il corso viene seguito dagli studenti iscritti • **Area_tematica (enum_tematica):** indica la tematica principale del corso • **ID_Studente (int):**

		identificativo numerico di qualsiasi studente iscritto • **ID_Corso (int):** identificativo numerico del corso • **ID_Operatore (character varying):** indica l'operatore che ha aggiunto il corso al sistema • **Minimo_delle_presenze (double precision):** indica il numero minimo di persone che devono essere presenti affinché venga svolto il corso • **max_presenze (integer):** indica il massimo di presenze che ogni studente può raggiungere
Lezione	Entità che conserva le informazioni che vengono inserite nel sistema riguardo alle lezioni presenti nel corso mantenendo ne uno storico	• **Titolo (character varying):** nome della lezione • **Descrizione (character varying):** descrizione della lezione • **Durata (timestamp):** tempo effettivo della durata di ogni singola lezione • **Data_inizio (date):** data in cui inizia una determinata lezione • **Ora_inizio (timestamp):** ora in cui inizia una determinata lezione • **Num_iscritti (int):** quanti studenti sono iscritti ad una determinata lezione • **Tasso_frequenza (double precision):** frequenza con la quale una lezione viene seguita dagli studenti • **ID_Lezione (integer):**

		Identificativo numerico di qualsiasi lezione presente nel sistema • **Numero_lezioni (integer):** indica da quante lezioni è composto il corso • **ID_Operatore (character varying):** indica l'operatore che ha aggiunto la lezione al sistema • **ID_Corso (integer):** indica a quale corso appartiene la lezione • **ID_aula (character varying):** indica in quale aula si svolge la lezione • **Ora_fine (timestamp):** indica l'orario in cui termina una lezione
Test	Entità che conserva le informazioni relative ai test che vengono inseriti nel sistema e ne mantiene uno storico	• **Titolo (character varying):** nome del test • **Descrizione (character varying):** descrizione del test • **Durata (timestamp):** tempo complessivo della durata del test • **Data_test (date):** data in cui inizia un determinato test • **Ora_inizio (timestamp):** ora in cui inizia un determinato test • **Minimo_punteggio (integer):** punteggio minimo che si deve raggiungere per il superamento di un test (attualmente è stato scelto un valore di 50/100 come punteggio minimo) • **Minimo_presenze (integer):** numero minime di presenze

		necessarie per permettere ad uno studente di sostenere un test • **ID_Test (integer):** codice identificativo che rappresenta ogni test • **ID_Operatore (character varying):** codice che indica quale operatore ha aggiunto il test al sistema • **ID_aula_reference (character varying):** identificativo alfanumerico che indica in quale aula si svolge il test
Tematica	Entità che va ad indicare la presenza di interrogazioni relative ad un corso	• **Area_tematica (enum_tematica):** indica quale tematica viene affrontata nell'interrogazione avanzata • **ID_Corso (integer):** indica a quale corso si riferisce la tematica • **Keywords (character varying):** parole chiave che permettono la ricerca facilitata di un corso
Aula	Entità che va a conservare le informazioni inserite all'interno del sistema riguardo l'aula in cui si svolgeranno i corsi	• **Num_max_posti (int):** indica il numero massimo di posti che si trovano all'interno di un'aula • **Num_presenti (int):** quanti studenti sono presenti in un'aula • **Percentuale_occupata (float):** percentuale di posti occupati in aula • **ID_aula (character varying):** identificativo alfanumerico che indica una specifica aula
Registro_prenotazione_aula	Entità creata per andare a tenere traccia	• **Data_prenotazione (date):** indica la data in cui l'aula è stata prenotata

	delle prenotazioni delle aule in cui verranno sostenuti i test oppure in cui si seguiranno le lezioni	• **Ora (timestamp):** indica l'ora in cui l'aula è stata prenotata • **ID_aula_reference (character varying):** indica la particolare aula che è stata occupata • **Check_status (character varying):** indica la disponibilità o meno dell'aula

2.4.2. Dizionario delle associazioni

Associazione	Descrizione	Classi coinvolte
Crea	Mappa la creazione di un corso da parte dell'operatore	• **Operatore [1..*]:** un operatore può creare ed inserire molti corsi • **Corso [*]:** un corso può essere creato da molti operatori
Aggiunge	Mappa l'aggiunta di uno studente ad un corso da parte di un operatore	• **Operatore [*]:** un operatore può aggiungere molti studenti ad un corso • **Studente [1]:** uno studente può essere aggiunto ad un corso da un operatore
Si iscrive	Indica l'iscrizione da parte di uno studente ad un corso	• **Studente [1, *]:** uno studente può iscriversi ad uno oppure a molti corsi • **Corso [0, *]:** un corso può essere seguito da zero fino a molti studenti
Sostiene	Mappa quali test vengono sostenuti da uno studente	• **Studente [0, *]:** un test può essere svolto da zero fino a molti studenti • **Test [0, *]:** uno studente può sostenere da zero fino a molti test

Possiede	Mappa il numero di test che fanno parte di un corso e il numero di interrogazioni avanzate che si possono sostenere/ricercare sempre all'interno di un corso	• **Corso [1, *]:** un corso può essere caratterizzato da uno fino a molti test • **Test [1, *]:** i test possono appartenere ad uno oppure a molti corsi • **Interrogazione avanzata [0, *]:** un corso può possedere da zero fino a molte interrogazioni avanzate
È composto	Mappa il numero di lezioni che sono presenti all'interno di un corso	• **Corso [1, *]:** un corso può essere composta da una fino a molte lezioni • **Lezione [1, *]:** una lezione può appartenere ad uno fino a molti corsi
È svolto	Mappa le aule in cui vengono svolti i corsi	• **Corso [1]:** l'aula può ospitare un solo corso alla volta • **Aula [0, *]:** un corso si può svolgere in zero fino a molte aule

2.4.3. Dizionario dei vincoli

Nome	Descrizione
Tasso_frequenza	Valore minimo di cui necessita ogni corso per permettere ad uno studente di sostenere i test, seguire le lezioni e conseguire il superamento del corso. Si è deciso che tale valore sia maggiore-uguale al 75%.
Minimo_punteggio	Il punteggio del test viene calcolato su una scala di valori compresi tra 0 e 100; dunque per il superamento del test è previsto un punteggio che sia un valore maggiore-uguale a 50 (in quanto risulta essere l'esatta

	metà di 100).
Minimo_presenze	Indica il minimo di presenze utili per il superamento del corso e il sostenimento dei test è stato raggiunto.
Max_presenze	Indica il massimo delle presenze che ogni corso può raggiungere
Durata	Durata espressa in ore dei test e delle lezioni che deve essere strettamente maggiore di 0.
Ora_inizio	Indica l'ora in cui una lezione o un test iniziano. Viene utilizzata come uno dei parametri utili per la prenotazione di un'aula
Ora_fine	Indica l'ora in cui finisce una lezione. Viene utilizzata come uno dei parametri utili per la prenotazione di un'aula
Data_inizio e Data_test	Entrambi indicano la data in cui una lezione e un test iniziano. Anch'esse vengono utilizzate come parametri utili per la prenotazione di un'aula.
Num_max_posti	Ogni aula che ospita un corso può contenere un numero massimo di studenti che può contenere.
enum_tematica	Tipo di tematica su cui verte un corso che deve compreso tra questi quattro: inglese, algebra, matematica e informatica

3. Progettazione logica

In questo capitolo sarà trattata la fase successiva della progettazione della base di dati, andando in questo modo ad un livello di astrazione più basso rispetto alla precedente.

Si tradurrà lo schema concettuale (predisposto in seguito alla ristrutturazione) in uno schema logico, dipendente dal tipo di struttura dei dati prescelto cioè quello relazionale puto. Negli schemi relazionali che seguiranno le chiavi primarie sono indicate dall'utilizzo di una sottolineatura (Esempio: chiave primaria) mentre le chiavi esterne saranno indicate da una doppia sottolineatura (Esempio: chiave esterna). Qual ora si trattasse sia di chiave primaria che chiave esterna, la dicitura sarà una sottolineatura tratteggiata della parola (Esempio: chiave primaria ed esterna). Sono stati inseriti anche unique constraint e check constraint che verranno indicati come segue:

- Unique constraint
- Check constraint

Nota bene: *Nella foto del Class Diagram revisionato, già precedentemente caricata all'interno di questo stesso documento* ***(vedi punto 2.2.)*** *sono state indicate solamente le chiavi primarie con la dicitura <<PK>> in quanto Visual Paradigm non permetteva di indicare anche le chiavi esterne, l'unique constraint e il check constraint.*

3.1. Schema Logico

CORSO	(Tasso_frequenza, ID_Corso, Nome, Descrizione, Presenze, iscritti, ID_Operatore, Numero_lezioni, ID_Studente, Minimo_delle_presenze, max_presenze, Area_tematica)
LEZIONE	(ID_Lezione, ID_Corso, Data_inizio, Ora_inizio, Num_iscritti, Tasso_frequenza, ID_Operatore, Numero_lezioni, ID_aula, Titolo, Descrizione, Ora_fine, durata)
OPERATORE	(ID_Operatore)

TEST	(ID_Test, ID_Operatore, Titolo, Descrizione, Durata, Ora_inizio, Minimo_punteggio, Minimo_presenze, Data_test, ID_aula_test)
STUDENTE	(ID_Studente, Nome, Cognome, Tasso_frequenza, ID_Operatore, ID_Corso, Punteggio_test)
TEMATICA	(Area_tematica, ID_Corso, Keywords)
AULA	(Num_max_posti, Num_presenti, Percentuale_occupata, ID_aula)
REGISTRO_PRENOTAZIONE_AULA	(Data_prenotazione, ora, ID_aula_reference, Check_status)

ID_OPERATORE → OPERATORE

ID_CORSO → CORSO

ID_STUDENTE → STUDENTE

ID_AULA_REFERENCE → ID_AULA

ID_AULA → AULA

4. Progettazione fisica

4.1. Progettazione delle tabelle

Di seguito la definizione delle tabelle.

4.1.1. Definizione della tabella Corso

```sql
-- Table: public.corso

-- DROP TABLE IF EXISTS public.corso;

CREATE TABLE IF NOT EXISTS public.corso
(
    tasso_frequenza double precision,
    area_tematica character varying(100) COLLATE pg_catalog."default" NOT NULL DEFAULT ' '::character varying,
    id_corso integer NOT NULL,
    nome character varying(50) COLLATE pg_catalog."default" NOT NULL,
    descrizione character varying(500) COLLATE pg_catalog."default",
    presenze double precision,
    iscritti integer NOT NULL,
    "ID_operatore" character varying COLLATE pg_catalog."default",
    numero_lezioni integer,
    "ID_studente" integer,
    minimo_delle_presenze double precision GENERATED ALWAYS AS ((iscritti / 2)) STORED,
    max_presenze integer GENERATED ALWAYS AS (((iscritti * 2) + 10)) STORED,
    CONSTRAINT "Corso_pkey" PRIMARY KEY (id_corso),
    CONSTRAINT "Unique_Corso" UNIQUE (area_tematica),
    CONSTRAINT "Corso_ID_Operatore_fkey" FOREIGN KEY ("ID_operatore")
        REFERENCES public."Operatore" ("ID_Operatore") MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public.corso
    OWNER to postgres;
-- Index: fki_FK_Corso

-- DROP INDEX IF EXISTS public."fki_FK_Corso";

CREATE INDEX IF NOT EXISTS "fki_FK_Corso"
    ON public.corso USING btree
    ("ID_operatore" COLLATE pg_catalog."default" ASC NULLS LAST)
    TABLESPACE pg_default;
```

4.1.2. Definizione della tabella Operatore

```sql
-- Table: public.Operatore

-- DROP TABLE IF EXISTS public."Operatore";

CREATE TABLE IF NOT EXISTS public."Operatore"
(
    "ID_Operatore" character varying COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT "PK_Operatore" PRIMARY KEY ("ID_Operatore")
        INCLUDE("ID_Operatore")
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public."Operatore"
    OWNER to postgres;
-- Index: fki_ID_Operatore

-- DROP INDEX IF EXISTS public."fki_ID_Operatore";

CREATE INDEX IF NOT EXISTS "fki_ID_Operatore"
    ON public."Operatore" USING btree
    ("ID_Operatore" COLLATE pg_catalog."default" ASC NULLS LAST)
    TABLESPACE pg_default;
```

4.1.3. Definizione della tabella Test

```
-- Table: public.Test

-- DROP TABLE IF EXISTS public."Test";

CREATE TABLE IF NOT EXISTS public."Test"
(
    id_test integer NOT NULL,
    "ID_Operatore" character varying COLLATE pg_catalog."default" NOT NULL,
    "Titolo" character varying[] COLLATE pg_catalog."default" NOT NULL,
    "Descrizione" character varying[] COLLATE pg_catalog."default",
    "Durata" time without time zone[] NOT NULL,
    ora_inizio time with time zone NOT NULL,
    "Minimo_punteggio" integer[] NOT NULL,
    "Minimo_presenze" integer[] NOT NULL,
    data_test date[],
    id_aula_test character varying COLLATE pg_catalog."default",
    CONSTRAINT "Test_pkey" PRIMARY KEY (id_test),
    CONSTRAINT unicita_test UNIQUE (id_aula_test, data_test, id_test, ora_inizio),
    CONSTRAINT "FK_Test" FOREIGN KEY ("ID_Operatore")
        REFERENCES public."Operatore" ("ID_Operatore") MATCH FULL
        ON UPDATE NO ACTION
        ON DELETE NO ACTION
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public."Test"
    OWNER to postgres;
-- Index: fki_FK_Test
```

```
31 -- DROP INDEX IF EXISTS public."fki_FK_Test";
32
33 CREATE INDEX IF NOT EXISTS "fki_FK_Test"
34     ON public."Test" USING btree
35     ("ID_Operatore" COLLATE pg_catalog."default" ASC NULLS LAST)
36     TABLESPACE pg_default;
37 -- Index: fki_fk_aula
38
39 -- DROP INDEX IF EXISTS public.fki_fk_aula;
40
41 CREATE INDEX IF NOT EXISTS fki_fk_aula
42     ON public."Test" USING btree
43     (id_aula_test COLLATE pg_catalog."default" ASC NULLS LAST)
44     TABLESPACE pg_default;
```

4.1.4. Definizione della tabella Studente

```sql
-- Table: public.studente

-- DROP TABLE IF EXISTS public.studente;

CREATE TABLE IF NOT EXISTS public.studente
(
    id_studente integer NOT NULL,
    nome character varying COLLATE pg_catalog."default" NOT NULL,
    cognome character varying COLLATE pg_catalog."default" NOT NULL,
    tasso_frequenza double precision NOT NULL,
    id_operatore character varying COLLATE pg_catalog."default",
    id_corso integer NOT NULL,
    punteggio_test smallint,
    CONSTRAINT "Studente_pkey" PRIMARY KEY (id_studente),
    CONSTRAINT uno_studente_un_corso UNIQUE (id_studente, id_corso),
    CONSTRAINT "FK_Studente" FOREIGN KEY (id_operatore)
        REFERENCES public."Operatore" ("ID_Operatore") MATCH SIMPLE
        ON UPDATE NO ACTION
        ON DELETE NO ACTION,
    CONSTRAINT "Studente_Tasso_frequenza_check" CHECK (tasso_frequenza >= 0.00::double precision AND tasso_frequenza <= 100.00::double precision)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public.studente
    OWNER to postgres;

COMMENT ON COLUMN public.studente.id_corso
    IS 'Corso a cui sono iscritti';
-- Index: fki_FK_Studente

-- DROP INDEX IF EXISTS public."fki_FK_Studente";

CREATE INDEX IF NOT EXISTS "fki_FK_Studente"
    ON public.studente USING btree
    (id_operatore COLLATE pg_catalog."default" ASC NULLS LAST)
    TABLESPACE pg_default;
-- Index: fki_fk_votii

-- DROP INDEX IF EXISTS public.fki_fk_votii;

CREATE INDEX IF NOT EXISTS fki_fk_votii
    ON public.studente USING btree
    (punteggio_test ASC NULLS LAST)
    TABLESPACE pg_default;
```

4.1.5. Definizione della tabella Tematica

```
-- Table: public.Tematica

-- DROP TABLE IF EXISTS public."Tematica";

CREATE TABLE IF NOT EXISTS public."Tematica"
(
    area_tematica character varying COLLATE pg_catalog."default" NOT NULL,
    id_corso integer NOT NULL,
    keywords character varying COLLATE pg_catalog."default",
    CONSTRAINT "Tematica_pk" PRIMARY KEY (area_tematica)
        INCLUDE(area_tematica)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public."Tematica"
    OWNER to postgres;

GRANT ALL ON TABLE public."Tematica" TO postgres;

GRANT ALL ON TABLE public."Tematica" TO PUBLIC;
-- Index: fki_FK_Tematica

-- DROP INDEX IF EXISTS public."fki_FK_Tematica";

CREATE INDEX IF NOT EXISTS "fki_FK_Tematica"
    ON public."Tematica" USING btree
    (id_corso ASC NULLS LAST)
    TABLESPACE pg_default;
```

4.1.6. Definizione della tabella Aula

```
-- Table: public.aula

-- DROP TABLE IF EXISTS public.aula;

CREATE TABLE IF NOT EXISTS public.aula
(
    num_max_posti integer NOT NULL,
    num_presenti integer,
    percentuale_occupata double precision,
    id_aula character varying COLLATE pg_catalog."default" NOT NULL,
    CONSTRAINT "PK_Aula" PRIMARY KEY (id_aula)
        INCLUDE(id_aula),
    CONSTRAINT posti_max_check CHECK (num_max_posti >= num_presenti)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public.aula
    OWNER to postgres;

```

4.1.7. Definizione della tabella Lezione

```
-- Table: public.lezione

-- DROP TABLE IF EXISTS public.lezione;

CREATE TABLE IF NOT EXISTS public.lezione
(
    id_lezione integer NOT NULL,
    id_corso integer,
    data_inizio date[] NOT NULL,
    ora_inizio time without time zone NOT NULL,
    num_iscritti integer,
    tasso_frequenza double precision,
    id_operatore character varying COLLATE pg_catalog."default",
    numero_lezioni integer,
    id_aula character varying COLLATE pg_catalog."default",
    titolo character varying COLLATE pg_catalog."default",
    descrizione character varying COLLATE pg_catalog."default",
    ora_fine time without time zone,
    durata integer,
    CONSTRAINT "PK" PRIMARY KEY (id_lezione)
        INCLUDE(id_lezione),
    CONSTRAINT "Data_od_ora_in_cui_aula_già_prenotata" UNIQUE (id_aula, data_inizio, ora_inizio),
    CONSTRAINT "FK" FOREIGN KEY (id_corso)
        REFERENCES public.corso (id_corso) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE NO ACTION,
    CONSTRAINT "FK_Aula" FOREIGN KEY (id_aula)
        REFERENCES public.aula (id_aula) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE NO ACTION,
```

```
    CONSTRAINT "Lezione_Tasso_Frequenza_check" CHECK (tasso_frequenza >= 0.00::double precision AND tasso_frequenza <= 100.00::double precision)
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public.lezione
    OWNER to postgres;
-- Index: fki_FK

-- DROP INDEX IF EXISTS public."fki_FK";

CREATE INDEX IF NOT EXISTS "fki_FK"
    ON public.lezione USING btree
    (id_corso ASC NULLS LAST)
    TABLESPACE pg_default;
-- Index: fki_FK_Aula

-- DROP INDEX IF EXISTS public."fki_FK_Aula";

CREATE INDEX IF NOT EXISTS "fki_FK_Aula"
    ON public.lezione USING btree
    (id_aula COLLATE pg_catalog."default" ASC NULLS LAST)
    TABLESPACE pg_default;
```

4.1.8. Definizione della tabella Registro_prenotazione_aula

```sql
-- Table: public.registro_prenotazione_aula

-- DROP TABLE IF EXISTS public.registro_prenotazione_aula;

CREATE TABLE IF NOT EXISTS public.registro_prenotazione_aula
(
    data_prenotazione date[] NOT NULL,
    ora time with time zone NOT NULL,
    id_aula_reference character varying COLLATE pg_catalog."default",
    check_status character varying[] COLLATE pg_catalog."default",
    CONSTRAINT "Aula_non_occupata" UNIQUE (data_prenotazione, ora, id_aula_reference)
        INCLUDE(data_prenotazione, ora, id_aula_reference),
    CONSTRAINT "FK_prenotazione" FOREIGN KEY (id_aula_reference)
        REFERENCES public.aula (id_aula) MATCH SIMPLE
        ON UPDATE CASCADE
        ON DELETE NO ACTION
)

TABLESPACE pg_default;

ALTER TABLE IF EXISTS public.registro_prenotazione_aula
    OWNER to postgres;
-- Index: fki_FK_prenotazione

-- DROP INDEX IF EXISTS public."fki_FK_prenotazione";

CREATE INDEX IF NOT EXISTS "fki_FK_prenotazione"
    ON public.registro_prenotazione_aula USING btree
    (id_aula_reference COLLATE pg_catalog."default" ASC NULLS LAST)
    TABLESPACE pg_default;
```

4.2. Definizione dei vincoli di dominio

Di seguito la definizione dei vincoli di dominio

4.2.2. Dominio enum_tematica

```
-- CHECK: public.enum_tematica.enum_tematica_check

-- ALTER DOMAIN public.enum_tematica DROP CONSTRAINT enum_tematica_check;

ALTER DOMAIN public.enum_tematica
    ADD CONSTRAINT enum_tematica_check CHECK (VALUE::text = ANY (ARRAY['Inglese'::character varying, 'Algebra'::character varying,

'Informatica'::character varying, 'Matematica'::character varying]::text[]));
```

4.3. Definizione delle trigger functions e dei trigger

Di seguito la definizione dei trigger implementati con le relative funzioni

4.3.2. Definizione trigger function conferma_prenotazione ()

```
-- FUNCTION: public.conferma_prenotazione()

-- DROP FUNCTION IF EXISTS public.conferma_prenotazione();

CREATE OR REPLACE FUNCTION public.conferma_prenotazione()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$
BEGIN
--Prenotazioni si fanno entro un'ora prima della lezione e dieci minuti dopo la sua fine;--
IF((Current_Date+Current_Time>old.data_prenotazione+old.data_prenotazione-INTERVAL '1 ora')
OR (Current_Date+Current_Time>old.data_prenotazione+old.data_prenotazione+INTERVAL '10 minuti'))
AND new.check_status='Conferma prenotazione'
THEN
RAISE EXCEPTION 'Prenotazione non disponibile al momento';
ELSEIF new.check_status='Eliminata' AND Current_Date+Current_Time>old.data_prenotazione+old.ora
THEN
RAISE EXCEPTION 'Non puoi cancellare la prenotazione al momento';
ELSEIF new.check_status='Abbandonata' AND Current_Date+Current_Time<old.data_prenotazione+old.ora+INTERVAL '10 minuti'
THEN
RAISE EXCEPTION 'Non puoi abbandonare la prenotazione';
END IF;
RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.conferma_prenotazione()
    OWNER TO postgres;
```

```
-- Trigger: conferma_prenotazione

-- DROP TRIGGER IF EXISTS conferma_prenotazione ON public.registro_prenotazione_aula;

CREATE TRIGGER conferma_prenotazione
    BEFORE UPDATE
    ON public.registro_prenotazione_aula
    FOR EACH ROW
    EXECUTE FUNCTION public.conferma_prenotazione();
```

4.3.3. Definizione trigger function controllo_aula_non_occupata_lezione ()

```sql
-- FUNCTION: public.controllo_aula_non_occupata_lezione()

-- DROP FUNCTION IF EXISTS public.controllo_aula_non_occupata_lezione();

CREATE OR REPLACE FUNCTION public.controllo_aula_non_occupata_lezione()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$

DECLARE
contatore integer;

BEGIN

    SELECT COUNT(*) INTO contatore
    FROM Public.lezione as l
    WHERE NEW.data_inizio = l.data_inizio AND NEW.id_aula = l.id_aula AND((NEW.ora_inizio< l.ora_fine AND NEW.ora_inizio > l.ora_inizio)
                                                                          OR  (NEW.ora_fine>l.ora_inizio AND NEW.ora_fine<l.ora_fine));

    IF (contatore>0)
    THEN RAISE EXCEPTION 'AULA GIA OCCUPATA';
    END IF;

RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.controllo_aula_non_occupata_lezione()
    OWNER TO postgres;
```

```sql
-- Trigger: non_occupata

-- DROP TRIGGER IF EXISTS non_occupata ON public.lezione;

CREATE TRIGGER non_occupata
    BEFORE INSERT OR UPDATE
    ON public.lezione
    FOR EACH ROW
    EXECUTE FUNCTION public.controllo_aula_non_occupata_lezione();
```

4.3.4. Definizione trigger function controllo_aula_non_occupata_test ()

```sql
-- FUNCTION: public.controllo_aula_non_occupata_test()

-- DROP FUNCTION IF EXISTS public.controllo_aula_non_occupata_test();

CREATE OR REPLACE FUNCTION public.controllo_aula_non_occupata_test()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$
DECLARE
contatore integer;

BEGIN

    SELECT COUNT(*) INTO contatore
    FROM Public.Test as l
    WHERE NEW.data_test = l.data_test AND NEW.id_aula_test = l.id_aula_test AND((NEW.ora_inizio< l.durata AND NEW.ora_inizio > l.durata)
                                                                          OR  (NEW.durata>l.ora_inizio AND NEW.durata<l.ora_fine));

    IF (contatore>0)
    THEN RAISE EXCEPTION 'AULA GIA OCCUPATA';
    END IF;

RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.controllo_aula_non_occupata_test()
    OWNER TO postgres;
```

```
-- Trigger: non_occupa_test

-- DROP TRIGGER IF EXISTS non_occupa_test ON public."Test";

CREATE TRIGGER non_occupa_test
    BEFORE INSERT OR UPDATE
    ON public."Test"
    FOR EACH ROW
    EXECUTE FUNCTION public.controllo_aula_non_occupata_test();
```

4.3.5. Definizione trigger function controllo_tasso_frequenza ()

```
-- FUNCTION: public.controllo_tasso_frequenza()

-- DROP FUNCTION IF EXISTS public.controllo_tasso_frequenza();

CREATE OR REPLACE FUNCTION public.controllo_tasso_frequenza()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$

BEGIN

IF( NEW.tasso_frequenza<75.0 ) THEN

RAISE NOTICE 'VALORE SUFFICIENTE';

ELSE
RAISE NOTICE 'VALORE INSUFFICIENTE';

END IF;

RETURN NULL;

END;
$BODY$;

ALTER FUNCTION public.controllo_tasso_frequenza()
    OWNER TO postgres;
```

```
-- Trigger: controllo_tasso_frequenza

-- DROP TRIGGER IF EXISTS controllo_tasso_frequenza ON public.lezione;

CREATE TRIGGER controllo_tasso_frequenza
    AFTER INSERT OR UPDATE OF tasso_frequenza
    ON public.lezione
    FOR EACH STATEMENT
    EXECUTE FUNCTION public.controllo_tasso_frequenza();
```

4.3.6. Definizione trigger function inserimento_multitabella_lezione ()

```sql
-- FUNCTION: public.inserimento_mutltitabella_lezione()

-- DROP FUNCTION IF EXISTS public.inserimento_mutltitabella_lezione();

CREATE OR REPLACE FUNCTION public.inserimento_mutltitabella_lezione()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$


 BEGIN

INSERT INTO Public."Registro_prenotazione_aula"
    VALUES( NEW.data_inizio, NEW.ora_inizio,NEW.id_aula);

RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.inserimento_mutltitabella_lezione()
    OWNER TO postgres;
```

4.3.7.Definizione trigger function inserimento_multitabella_test ()

```sql
-- FUNCTION: public.inserimento_mutltitabella_test()

-- DROP FUNCTION IF EXISTS public.inserimento_mutltitabella_test();

CREATE OR REPLACE FUNCTION public.inserimento_mutltitabella_test()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$


 BEGIN

INSERT INTO Public.registro_prenotazione_aula (data_prenotazione,ora,id_aula_reference)
    VALUES( NEW.data_test, NEW.ora_inizio, NEW.id_aula_test);

RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.inserimento_mutltitabella_test()
    OWNER TO postgres;
```

```sql
-- Trigger: inserimento_tra_tabelle

-- DROP TRIGGER IF EXISTS inserimento_tra_tabelle ON public."Test";

CREATE TRIGGER inserimento_tra_tabelle
    AFTER INSERT OR UPDATE
    ON public."Test"
    FOR EACH ROW
    EXECUTE FUNCTION public.inserimento_mutltitabella_test();
```

4.3.8. Definizione trigger function inserisci_prenotazione ()

```
-- FUNCTION: public.inserisci_prenotazione()

-- DROP FUNCTION IF EXISTS public.inserisci_prenotazione();

CREATE OR REPLACE FUNCTION public.inserisci_prenotazione()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$
DECLARE Uguale INTEGER;
BEGIN
select count(*) INTO Uguale FROM registro_prenotazione_aula
WHERE new.data_prenotazione=data_prenotazione AND new.ora=ora AND new.id_aula_reference=id_aula_reference;
IF Uguale<>0
THEN
RAISE EXCEPTION 'Aula non disponibile al momento.Riprovare più tardi';
END IF;
RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.inserisci_prenotazione()
    OWNER TO postgres;
```

```
-- Trigger: inserisci_prenotazione

-- DROP TRIGGER IF EXISTS inserisci_prenotazione ON public.registro_prenotazione_aula;

CREATE TRIGGER inserisci_prenotazione
    BEFORE INSERT
    ON public.registro_prenotazione_aula
    FOR EACH ROW
    EXECUTE FUNCTION public.inserisci_prenotazione();
```

4.3.9. Definizione trigger function percentuale_frequenza ()

```
CREATE OR REPLACE FUNCTION public.percentuale_frequenza()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$

DECLARE

    var1 double precision;

BEGIN

var1 = NEW."presenze"*100;

var1 = var1/NEW."tscritti";

IF NEW.tasso_frequenza IS NULL
THEN
UPDATE Public.corso
SET tasso_frequenza = var1
WHERE tasso_frequenza IS NULL;
END IF;

RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.percentuale_frequenza()
    OWNER TO postgres;
```

```
-- Trigger: percentuale_frequenza

-- DROP TRIGGER IF EXISTS percentuale_frequenza ON public.corso;

CREATE TRIGGER percentuale_frequenza
    AFTER INSERT OR UPDATE
    ON public.corso
    FOR EACH ROW
    EXECUTE FUNCTION public.percentuale_frequenza();
```

4.3.10. Definizione trigger function percentuale_riempimento ()

```
CREATE OR REPLACE FUNCTION public.percentuale_riempimento()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$

DECLARE

    var1 double precision;

BEGIN

var1 = NEW."num_presenti"*100;

var1 = var1/NEW."num_max_posti";

IF NEW.percentuale_occupata IS NULL
THEN
UPDATE Public.aula
SET percentuale_occupata = var1
WHERE percentuale_occupata IS NULL;
END IF;

RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.percentuale_riempimento()
    OWNER TO postgres;
```

```
-- Trigger: percentuale_delle_presenze

-- DROP TRIGGER IF EXISTS percentuale_delle_presenze ON public.aula;

CREATE TRIGGER percentuale_delle_presenze
    AFTER INSERT OR UPDATE
    ON public.aula
    FOR EACH ROW
    EXECUTE FUNCTION public.percentuale_riempimento();
```

4.3.11. Definizione trigger function tasso_frequenza_calcolo ()

```
CREATE OR REPLACE FUNCTION public.tasso_frequenza_calcolo()
    RETURNS trigger
    LANGUAGE 'plpgsql'
    COST 100
    VOLATILE NOT LEAKPROOF
AS $BODY$

DECLARE

    var1 double precision;

BEGIN

var1 = NEW."num_iscritti";

var1 = var1/2;

IF NEW.tasso_frequenza IS NULL
THEN
UPDATE Public.lezione
SET tasso_frequenza = var1
WHERE tasso_frequenza IS NULL;
END IF;

RETURN NEW;
END;
$BODY$;

ALTER FUNCTION public.tasso_frequenza_calcolo()
    OWNER TO postgres;
```

```
-- Trigger: calcolo_tasso

-- DROP TRIGGER IF EXISTS calcolo_tasso ON public.lezione;

CREATE TRIGGER calcolo_tasso
    AFTER INSERT OR UPDATE
    ON public.lezione
    FOR EACH ROW
    EXECUTE FUNCTION public.tasso_frequenza_calcolo();
```

4.4. Definizione della Viste

Di seguito le Views che sono state implementate, il loro scopo e esempio

4.4.2. view_controlla_frequenza

La View in questione permette di visualizzare le lezioni di un corso alle quali è presente un tasso di frequenza maggiore e/o uguale al minimo previsto (si ricordi che è stato scelto un minimo del 75%)

```sql
-- View: public.view_controlla_frequenza

-- DROP VIEW public.view_controlla_frequenza;

CREATE OR REPLACE VIEW public.view_controlla_frequenza
 AS
 SELECT lezione.id_lezione AS "ID_Lezione",
    lezione.id_corso AS "ID_Corso",
    lezione.titolo AS "Titolo",
    lezione.data_inizio AS "Data_inizio",
    lezione.ora_inizio AS "Ora_inizio",
    lezione.tasso_frequenza,
    lezione.id_operatore AS "ID_Operatore"
   FROM lezione
  WHERE lezione.tasso_frequenza >= 75::double precision;

ALTER TABLE public.view_controlla_frequenza
    OWNER TO postgres;

```

	ID_Lezione integer	ID_Corso integer	Titolo character varying	Data_inizio date[]	Ora_inizio time without time zone	tasso_frequenza double precision	ID_Operatore character varying
1	555	3	Lezione 5: inglese	{2021-09-17}	08:00:00	98	DF0112345678912
2	666	3	Lezione 5:algebra	{2021-09-20}	08:00:00	90	DF0112345678912
3	2222	4	Lezione 5:matematica	{2021-09-07}	15:00:00	89	EA0112345678912
4	4	1	Lezione 1:matematica	{2021-09-07}	09:00:00	77	LB9912345678912
5	5	1	Lezione 2: inglese	{2021-09-08}	08:00:00	85	LB9912345678912
6	33	2	Lezione 3: inglese	{2021-09-08}	09:00:00	99	EA0112345678912
7	44	2	Lezione 3:algebra	{2021-09-09}	09:00:00	100	EA0112345678912
8	111	3	Lezione 4:inglese	{2021-09-03}	10:00:00	75	DF0112345678912
9	222	3	Lezione 4:algebra	{2021-09-06}	15:00:00	93	DF0112345678912

4.4.3. view_controlla_voti_frequenza

La View in questione permette di visualizzare tutti gli studenti che non solo hanno avuto un punteggio maggiore e/o uguale al minimo richiesto per i test (si ricordi che il valore minimo è stato indicato con 50/100) ma che riportano anche un tasso di frequenza maggiore e/o uguale al minimo richiesto.

```sql
-- View: public.view_controlla_voti_frequenza

-- DROP VIEW public.view_controlla_voti_frequenza;

CREATE OR REPLACE VIEW public.view_controlla_voti_frequenza
 AS
 SELECT studente.id_studente AS "ID_Studente",
    studente.nome AS "Nome",
    studente.cognome AS "Cognome",
    studente.tasso_frequenza,
    studente.punteggio_test
   FROM studente
  WHERE studente.punteggio_test >= 50 AND studente.tasso_frequenza >= 75::double precision;

ALTER TABLE public.view_controlla_voti_frequenza
    OWNER TO postgres;

```

	ID_Studente integer	Nome character varying	Cognome character varying	tasso_frequenza double precision	punteggio_test smallint
1	4321	Eva	Brighi	100	54
2	86004	Giovanni	Garau	98	53
3	1234	Martino	Rametta	85	80
4	86003	Niccolò	Fares	75	81
5	86005	Amelia Mia	Winter	100	100

4.4.4. view_visualizza_studenti

La View in questione permette di visualizzare tutti gli studenti che sono presenti nel sistema.

```
-- View: public.view_visualizza_studenti

-- DROP VIEW public.view_visualizza_studenti;

CREATE OR REPLACE VIEW public.view_visualizza_studenti
 AS
 SELECT studente.id_studente,
    studente.tasso_frequenza,
    studente.id_corso,
    studente.cognome
   FROM studente;

ALTER TABLE public.view_visualizza_studenti
    OWNER TO postgres;

```

	id_studente integer	tasso_frequenza double precision	id_corso integer	cognome character varying
1	4321	100	4	Brighi
2	86004	98	3	Garau
3	1234	85	1	Rametta
4	86003	75	2	Fares
5	86005	100	1	Winter
6	86006	60	3	Florenzi
7	860011	75	2	LONGOBARDO