PEGASO
Università Telematica

TEMPLATE
PROJECT WORK

Corso di Studio	INFORMATICA PER LE AZIENDE DIGITALI (L-31)
Dimensione dell'elaborato	Minimo 6.000 – Massimo 10.000 parole *(pari a circa Minimo 12 – Massimo 20 pagine)*
Formato del file da caricare in piattaforma	PDF
Nome e Cognome	Alfredo Verdosci
Numero di matricola	0312401147
Tema n. (Indicare il numero del tema scelto):	1
Titolo del tema (Indicare il titolo del tema scelto):	La digitalizzazione dell'impresa
Traccia del PW n. (Indicare il numero della traccia scelta):	4
Titolo della traccia (Indicare il titolo della traccia scelta):	Progettazione dello schema di persistenza dei dati a supporto dei servizi di un'azienda nel settore dei trasporti
Titolo dell'elaborato (Attribuire un titolo al proprio elaborato progettuale):	Database relazionale per TransferOffer – Welfare aziendale

PARTE PRIMA – DESCRIZIONE DEL PROCESSO

Utilizzo delle conoscenze e abilità derivate dal percorso di studio
(Descrivere quali conoscenze e abilità apprese durante il percorso di studio sono state utilizzate per la redazione dell'elaborato, facendo eventualmente riferimento agli insegnamenti che hanno contribuito a maturarle):

Grazie alle lezioni del docente mi è stato possibile rafforzare le conoscenze relative al linguaggio SQL, già acquisite in parte nella mia esperienza lavorativa.
Non avendo esperienza come Database Administrator (DBA), il corso mi ha consentito di migliorare non solo le basi teoriche, ma di approfondire concetti più avanzati, in particolare, ho avuto modo di comprendere con maggiore chiarezza l'importanza degli indici per l'ottimizzazione delle prestazioni e il miglioramento dell'efficienza delle query, valutando con precisione in quali contesti siano realmente necessari e come possano influire sul costo computazionale delle operazioni. Allo stesso modo, lo studio dei trigger e delle funzioni mi ha permesso di acquisire una visione più funzionale e teorica, comprendendo il loro ruolo nella gestione dell'integrità dei dati, nell'automazione dei processi e nell'implementazione di controlli maggiormente complessi.
Il contributo teorico fornito dal percorso di studi si è dimostrato particolarmente utile per ricollegare l'esperienza pratica già maturata con una solida base, facilitando così un approccio più strutturato alla progettazione e allo sviluppo del database.

Fasi di lavoro e relativi tempi di implementazione per la predisposizione dell'elaborato
(Descrivere le attività svolte in corrispondenza di ciascuna fase di redazione dell'elaborato. Indicare il tempo dedicato alla realizzazione di ciascuna fase, le difficoltà incontrate e come sono state superate):

Le fasi di lavoro si sono suddivise in:

- Studio dell'ambito
- Costruzione schema
- Sviluppo da schema logico a prodotto effettivo
- Ricerca e formalizzazione dati
- Inserimento dati
- Rollback e miglioramenti

Completate in trentadue giorni.
Per "giorno" si intende un tempo variabile dalle 5 alle 7 ore giornaliere, non consecutive.
Lo studio è avvenuto facendo una ricerca sui principali servizi italiani ferroviari, quali Trenitalia e Italo, cercando di formalizzare le operazioni di prenotazione, scontistica, le corse e tutte le gestioni che comportano. Lo studio è stato effettuato nell'arco di una giornata.
Per la costruzione dello schema, basandomi sullo studio effettuato il giorno precedente, è stato quello di schematizzare l'idea fatta precedentemente, cercando lo strumento adatto e perfezionando il più possibile il progetto. Questa fase è durata due giorni.
Lo sviluppo dello schema è stata la fase dove ha richiesto un maggior numero di iterazioni e controlli, poiché nel momento dello sviluppo della schema, le difficoltà maggiori riguardavano la coerenza tra le varie tabelle nel momento dell'eliminazione di un record o la modifica, sono stati per cui scritti appositamente dei trigger e funzioni per gestire possibili eccezioni e modifiche dati, inoltre sono stati implementati degli indici per velocizzare le esecuzioni delle query. La difficoltà maggiore è stata nell'aggiornare i record quando vengono aggiornate le corse dei treni, passando da stati Ritardo o Annullato e quello del calcolo della scontistica se un passeggero se ha un abbonamento sottoscritto. Per sviluppare tutta struttura e le logiche dietro, ci sono voluti quindici giorni.
La fase di ricerca dei dati è durata cinque giorni, ed è suddivisibile in due step, quella di ricerca effettiva di dati reali e quella di formalizzazione per la base dati. La ricerca è durata due giorni, per trovare le stazioni reali delle regioni, il file csv di riferimento ha portato poi alla fase successiva, cioè l'elaborazione e l'inserimento di questi ultimi, utilizzando Python per la fase di parsing del file, sfruttando le librerie "csv" e "psycopg2" per la connessione su PostgreSQL con cui ho già avuto modo di usare. La difficoltà maggiore è stata nel dover individuare le regioni, che sul file erano segnate con degli indici che andavano da 0 a 22, lo script è stato scritto quindi per leggere solo quelle da 1 a 20, escludendo le altre. Una volta estrapolati i dati in un

formato più leggibile, questi sono stati inseriti all'interno delle tabelle apposite "Regione" e "Stazione" automatizzando l'operazione. Per lo script è stato scritto utilizzando Visual Studio Code, data la piena compatibilità con Python attraverso i plugin ufficiali Python e Python Debugger di Microsoft.
Per la fase di inserimento dati, superata la fase di ricerca delle stazioni e regioni, sono stati scritte delle query per ogni tabella facendo riferimento a queste ultime reali. Per la tabella "Tratta", il campo "distanza_km" che riguarda il tragitto è stato scelto con l'ausilio di Google Maps. Questa fase è durata una giornata.
La fase "Rollback e miglioramenti" è l'ultima, dove sono stati reinseriti tutti i dati precedentemente presenti, sono stati commitatti altri indici per migliorare l'efficienza delle query e modifiche varie alle funzioni per l'avvio dei trigger. Inoltre alcune tabelle sono state ridefinite per migliorare l'atomicità e la coerenza dei dati. La fase è stata svolta minuziosamente effettuando numerosi test, per otto giorni.

Risorse e strumenti impiegati
(Descrivere quali risorse - bibliografia, banche dati, ecc. - e strumenti - software, modelli teorici, ecc. - sono stati individuati ed utilizzati per la redazione dell'elaborato. Descrivere, inoltre, i motivi che hanno orientato la scelta delle risorse e degli strumenti, la modalità di individuazione e reperimento delle risorse e degli strumenti, le eventuali difficoltà affrontate nell'individuazione e nell'utilizzo di risorse e strumenti ed il modo in cui sono state superate):

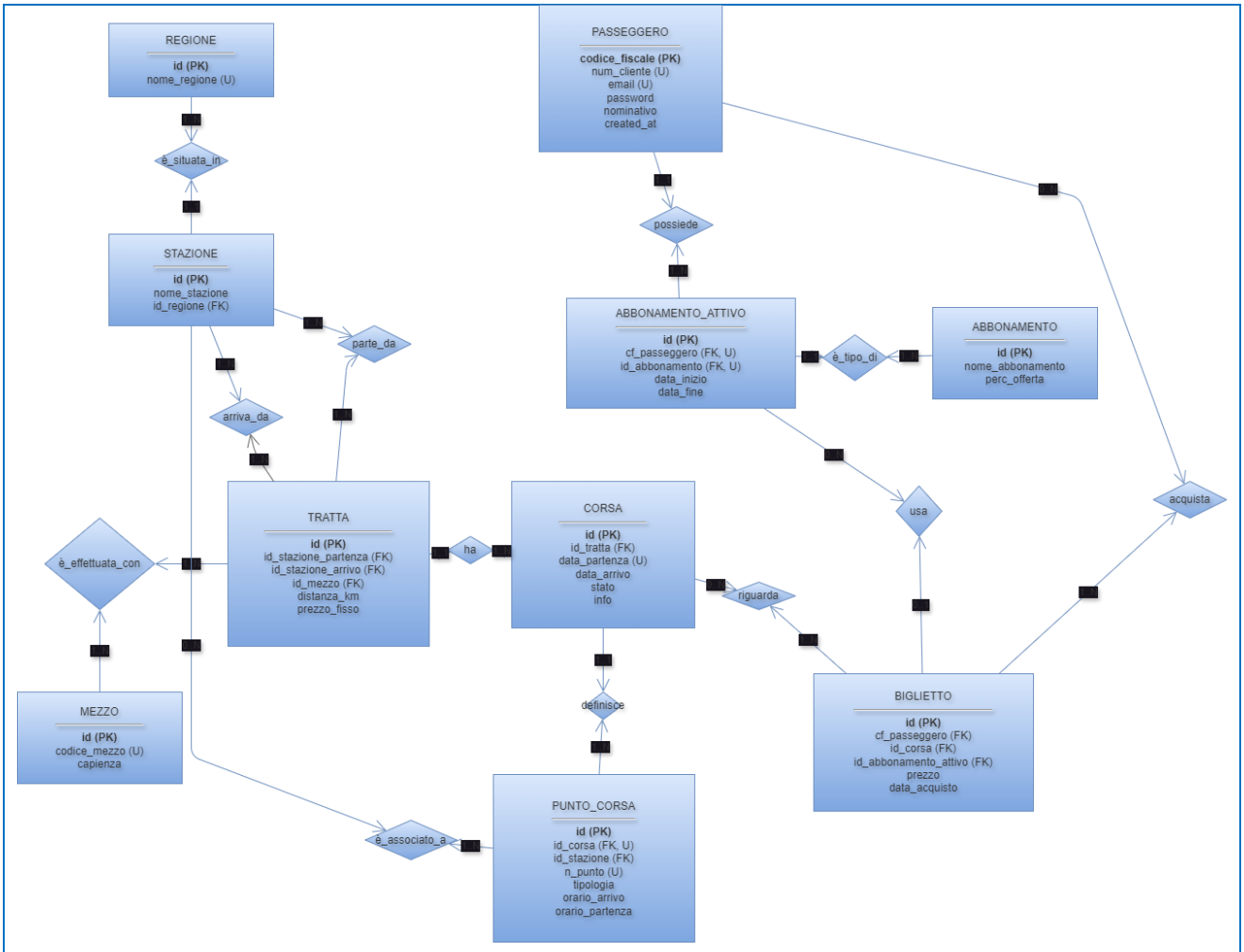
Per la fase di analisi e progettazione dello schema concettuale è stato adottato DBDiagram, un servizio online che consente la definizione dello schema tramite sintassi SQL e la generazione automatica del relativo modello grafico. Tale strumento ha permesso di validare rapidamente la struttura logica del database, evitando l'impiego di applicativi desktop più complessi, onerosi o dipendenti dall'ambiente di esecuzione.
Lo schema ER è stato costruito utilizzando lo strumento draw.io, data la semplicità d'uso dell'interfaccia.
Il DBMS selezionato per l'implementazione è PostgreSQL 17, scelto per la semplicità di distribuzione in ambiente locale mediante Docker Desktop, per le elevate prestazioni nella gestione dei dataset analizzati e per il supporto esteso e performante a funzionalità come procedure, trigger e tipi di dato più complessi, seppur questi ultimi non utilizzati. Inoltre PostgreSQL offre garanzie di integrità superiori, gestendo nativamente vincoli complessi e chiavi esterne senza le limitazioni di altri DBMS. La realizzazione delle componenti logiche lato database è stata condotta facendo riferimento alla documentazione ufficiale del sistema, oltre alle slide del docente presenti sulla piattaforma UniPegaso.
Il dataset in formato csv, contenente le informazioni relative alle stazioni e alle regioni, è stato reperito su GitHub, in quanto rappresentava l'unica fonte aggiornata al 2025 e caratterizzata da un numero di record superiore rispetto ad altre risorse disponibili online, seppur con alcuni dati non completi.
Per le attività di preprocessing e trasformazione del dataset è stato sviluppato uno script in Python 3.13. La gestione del formato csv è stata effettuata mediante la

libreria standard csv, mentre l'interazione con PostgreSQL è stata realizzata tramite la libreria psycopg2 versione 2.9.0.
Le operazioni di interrogazione, manipolazione e amministrazione del database, incluse la definizione delle operazioni CRUD e la verifica delle query, sono state svolte utilizzando DBeaver, selezionato per la sua interfaccia intuitiva, la completezza degli strumenti di gestione e l'elevata compatibilità con PostgreSQL.
La compilazione del Project Work è avvenuta attraverso il software del pacchetto Office Microsoft Word.
Infine, GitHub è stato adottato come piattaforma di versionamento e gestione del codice sorgente. Tale scelta è motivata dalla necessità di garantire tracciabilità delle modifiche, conservazione centralizzata degli artefatti, integrazione con workflow di sviluppo consolidati e una più agevole condivisione del materiale progettuale. Il repository è consultabile al seguente link.

PARTE SECONDA – PREDISPOSIZIONE DELL'ELABORATO

Obiettivi del progetto
(Descrivere gli obiettivi raggiunti dall'elaborato, indicando in che modo esso risponde a quanto richiesto dalla traccia):

La traccia parla di come bisogna progettare un sistema di persistenza dei dati a supporto del processo di vendita dei biglietti per un'azienda di trasporto, che offra servizi essenziali per la mobilità delle persone, quindi ho pensato potesse essere molto più interessante sviluppare la traccia come se fosse un servizio "esterno" da un provider ferroviario, quindi è stato costruito uno schema di persistenza dei dati ottimizzato per supportare efficacemente il processo di vendita dei biglietti di un sistema Corporate Welfare "TransferOffer". Il progetto nasce dall'analisi di un contesto reale, integrando le logiche di gestione dei benefit aziendali, nel mio caso ispirate al sistema in uso presso la mia azienda, con le dinamiche operative tipiche dei grandi fornitori ferroviari come Trenitalia e Italo.
L'attenzione è stata posta sulla modellazione fedele di elementi critici quali la gestione degli stati della corsa, la prenotazione ed emissione dei biglietti, il controllo in tempo reale della disponibilità e la gestione dinamica degli abbonamenti con le relative scontistiche. È stata data maggiore attenzione alla gestione di itinerari complessi che prevedono cambi o scali, garantendo una rappresentazione granulare del viaggio.
Lo schema è stato sviluppato seguendo un approccio rigorosamente relazionale, aderente ai principi della Terza Forma Normale (3NF). Le entità sono state collegate tramite un sistema pervasivo di chiavi esterne per mantenere l'integrità referenziale e permettere un'interazione fluida e sicura tra i dati.



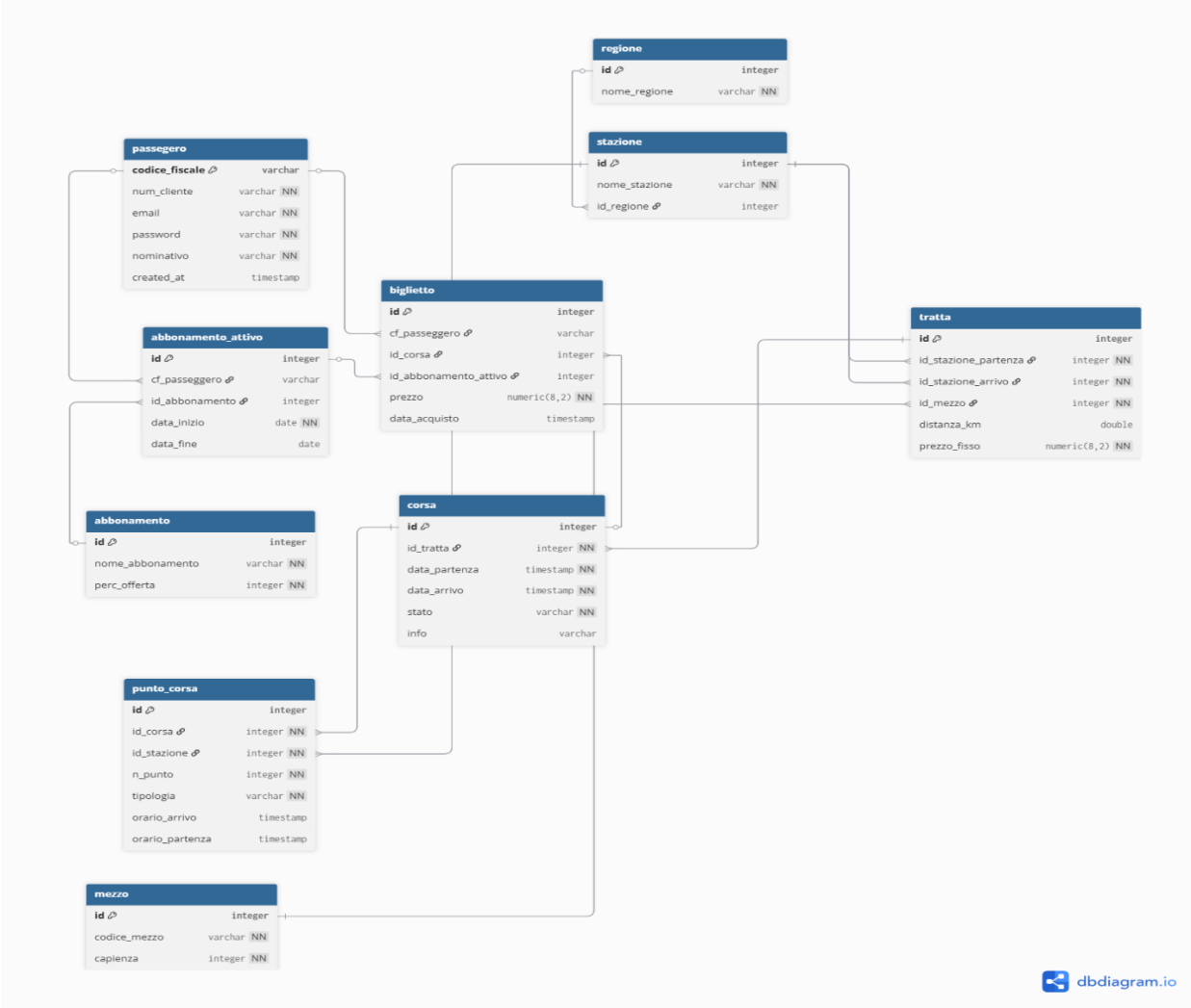
Dallo schema ER si nota una progettazione fortemente normalizzata. Non è stata introdotta alcuna forma di denormalizzazione strutturale, poiché l'obiettivo primario è garantire la massima coerenza, integrità e assenza di ridondanze. Un'eventuale ridondanza dei dati, infatti, introdurrebbe il rischio di disallineamenti nel tempo, richiedendo onerosi controlli applicativi per ogni operazione di scrittura.

L'uso della Terza Forma Normale (3NF) ha imposto che ogni relazione soddisfacesse tre requisiti fondamentali, l'atomicità dei valori (Prima forma normale), dipendenza completa dalla chiave primaria (Seconda forma normale) e, soprattutto, l'assenza di dipendenze transitive tra attributi non chiave. Nel contesto del sistema costruito, questo principio ha guidato decisioni strutturali complicate. Un esempio concreto di applicazione della 3NF si trova nella gestione delle tratte. Se avessimo inserito la distanza chilometrica direttamente nella tabella Corsa, e quindi una forma del genere "Corsa delle 8:00, Roma-Milano, 600km", avremmo creato una ridondanza: la distanza è una proprietà intrinseca della tratta fisica Roma-Milano, non del singolo treno che la percorre. Ripetere questo dato per ogni corsa avrebbe violato la 3NF, poiché l'attributo distanza dipenderebbe logicamente dalla tratta e non direttamente dalla chiave primaria della corsa (l'ID del viaggio specifico). Normalizzando lo schema e spostando la distanza nella tabella tratta, ho eliminato la duplicazione del dato, che resta comunque calcolabile, garantendo che una rettifica sulla lunghezza della linea ferroviaria si propaghi istantaneamente a tutte le corse associate, senza

necessità di aggiornamenti massivi.
L'ambito applicativo considerato è un sistema transazionale critico (OLTP), in cui ogni operazione deve essere atomica e affidabile. In questo contesto, la normalizzazione è preferibile poiché minimizza le anomalie di aggiornamento e semplifica la manutenzione evolutiva del modello. Oltre ai requisiti funzionali, il progetto risponde a stringenti requisiti non funzionali di integrità transazionale ACID, ovvero "atomicità", "coerenza", "isolamento" e "durabilità". In un sistema di welfare, dove il budget e la fidelizzazione dei dipendenti ha molta importanza, si è scelta di delegare i vincoli di validazione direttamente al motore database, tramite Constraints e Trigger, ciò assicura una robustezza superiore rispetto ai soli controlli software lato backend e frontend.

Contestualizzazione
(Descrivere il contesto teorico e quello applicativo dell'elaborato realizzato):

Per simulare un contesto reale, ho deciso di ideare TransferOffer – Welfare aziendale, un database per una piattaforma interna dedicata ai dipendenti di un'azienda, che offre soluzioni di viaggio a condizioni esclusive, sfruttando accordi stipulati con i principali fornitori nazionali, come Italo e Trenitalia.



TransferOffer permette una gestione riservata di un blocco di posti su tratte specifiche, offrendo una gestione attiva dell'inventario e delle regole di rimborso

interne.
Sebbene i treni siano operati da fornitori esterni, l'azienda gestisce un proprio inventario di posti riservati. La tabella “corsa” e il campo capienza non rappresentano l'intero treno, ma il "blocco posti” riservati, con la possibilità di osservare in tempo reale i posti disponibili, la tratta e le fermate, in modo completamente trasparente. Il servizio applica politiche di tutela del dipendente migliorative rispetto a quelle standard dei fornitori. Il sistema utilizza dei trigger sul database (come “trigger_corsa_biglietto”) per automatizzare queste regole interne, ad esempio, se una corsa viene segnata come 'In ritardo', il sistema può elaborare un rimborso automatico immediato, senza attendere le tempistiche burocratiche del provider esterno, che possono richiedere numerosi giorni.
Inoltre, è presente un sistema di "Abbonamento" (Bronze, Silver, Gold, Platinum) che modella lo status del dipendente basato sugli anni di servizio o su possibili premi produzione, con una scadenza nel caso in cui non si volesse rinnovare il servizio. Questo status garantisce sconti automatici sul prezzo di listino del viaggio riservato, il database riesce a calcolare il prezzo finale dinamicamente, incrociando il prezzo base della tratta, recuperato dal fornitore, con la percentuale di sconto spettante al dipendente al momento dell'acquisto. Informazioni come corse, mezzi con gli eventuali prezzi, sono offerti dai fornitori esterni, poiché TransferOffer non è in alcun modo incaricato alla gestione dell'erogazione fisica del servizio di trasporto. Il database, quindi, non si limita a visualizzare dati esterni, ma governa la logica di business proprietaria dell'azienda come la gestione dei posti, sconti, rimborsi speciali e statistiche, che costituisce il valore aggiunto della piattaforma a preferirla rispetto all’acquisto sui servizi principali.
Il sistema non opera in isolamento, ma è ideato per integrarsi con l'ecosistema IT esistente dell'azienda, agendo come collettore dati per il portale Intranet e interfacciandosi con il sistema HR per sincronizzare lo status degli abbonamenti. L’AS-IS si può vedere come questa architettura permette di superare le criticità dello scenario operativo tradizionale, caratterizzato da una forte frammentazione delle operazioni. Il dipendente doveva accedere autonomamente ai portali pubblici, anticipare il pagamento con carta personale, conservare i giustificativi e compilare manualmente note spese soggette a lunghi tempi di verifica e rimborso in busta paga. Con l'introduzione di TransferOffer, ovvero lo scenario TO-BE, questo flusso viene completamente reingegnerizzato e digitalizzato. L'accesso unico via Intranet elimina la necessità di utenze multiple, mentre l'integrazione con il budget welfare permette l'addebito diretto su un credito virtuale e un rimborso diretto su busta paga, rimuovendo l'onere dell'anticipo di cassa per il lavoratore. La gestione automatizzata delle anomalie trasforma infine un processo puramente amministrativo e burocratico in un servizio fluido, dove il rimborso per disservizi diventa un automatismo garantito dal codice e non più una pratica da istruire manualmente attraverso un

operatore.

Descrizione dei principali aspetti progettuali
(Sviluppare l'elaborato richiesto dalla traccia prescelta):

L'elaborato contiene sia script SQL, che uno script Python, la parte SQL contiene la parte di Data Defination Language, le varie operazioni CRUD, scrittura di indici, trigger, funzioni e View, mentre lo script Python, la parte di parsing e di validazione dei dati presi da un dataset in formato csv.
Il database è stato creato localmente utilizzando docker con i seguenti comandi:

```
docker pull postgres

docker run --name postgres \
 -e POSTGRES_PASSWORD=admin \
 -d postgres
```

Non è stata data molta importanza né al nome dell'istanza né alla complessità della password poiché non era richiesto.
L'esecuzione del container invece è stata data direttamente da UI di Docker Desktop.
Il dump del database è presente all'interno della repository git, ed è possibile importarlo solamente creando un database con lo stesso nome "postgres", altrimenti l'importazione darà errore.

SQL

Per garantire l'integrità dei dati e automatizzare le regole di business, il progetto sfrutta le funzionalità offerte da PostgreSQL. A differenza di un approccio tradizionale in cui la logica è interamente delegata unicamente dall'applicazione, qui si è scelto di implementare controlli direttamente nel livello di persistenza tramite l'uso di trigger.
Per definizione, un trigger è un oggetto del database associato a una tabella che viene attivato automaticamente al verificarsi di specifici eventi di manipolazione dei dati "BEFORE", cioè prima dell'inserimento del record, "AFTER", dopo l'inserimento del record, "INSTEAD OF", per le modifiche sulle viste, attraverso i comandi di "INSERT", "UPDATE", "DELETE". L'architettura del trigger segue il paradigma Event-Condition-Action, per Event intende l''operazione che scatena il trigger, la condition è un filtro opzionale che determina se il trigger deve essere eseguito, infine la action è la procedura che viene eseguita.
In PostgreSQL, la logica procedurale dell'azione è scritta in PL/pgSQL, Procedural Language/PostgreSQL, un linguaggio strutturato che estende l'SQL standard con costrutti di controllo come operatori condizionali e cicli, è presente una gestione delle eccezioni e la dichiarazione di variabili locali. L'uso di PL/pgSQL permette di eseguire calcoli complessi in modo atomico e performante, riducendo il traffico di

rete tra applicazione e database.

Le query di Data Definition Language sono visualizzabili sul Git, dando una visione alle seguenti tabelle in modo più approfondito:

```
CREATE TABLE corsa (
    id SERIAL PRIMARY KEY,
    id_tratta INTEGER NOT NULL REFERENCES tratta(id),
    data_partenza TIMESTAMP NOT NULL,
    data_arrivo TIMESTAMP NOT NULL,
    posti_disponibili INTEGER,
    stato VARCHAR NOT NULL DEFAULT 'Regolare',
    info VARCHAR DEFAULT NULL,
    CONSTRAINT check_corsa_date CHECK (data_arrivo >= data_partenza),
    CONSTRAINT check_posti_disponibili CHECK (posti_disponibili >= 0),
    CONSTRAINT stato_valido CHECK (stato IN ('Regolare', 'Completato', 'In ritardo', 'Annullato'))
)
```

La tabella corsa contiene in particolare dei controlli di validazione dei dati, ‘check_corsa_date’ non permette l’inserimento di una partenza inferiore a quella di arrivo, ‘check_posti_disponibili’ di non inserire nuovi posti se la colonna ‘posti_disponibili’ è uguale a 0, infine ‘stato_valido’ permette l’inserimento di dati solo seguento quattro possibilità, 'Regolare', 'Completato', 'In ritardo', 'Annullato', la modifica di una corsa negli ultimi due casi, scatena l’esecuzione di un trigger

```
CREATE OR REPLACE FUNCTION aggiorna_prezzo_biglietto_corsa()
RETURNS TRIGGER AS $$
DECLARE
    orario_previsto_arrivo TIMESTAMP;
    distanza_tratta DOUBLE PRECISION;
BEGIN
    IF NEW.stato = 'Annullato' THEN
        UPDATE biglietto
        SET prezzo = 0.00
        WHERE id_corsa = NEW.id;
        RETURN NEW;
    END IF;

    IF NEW.stato = 'In ritardo' THEN
        SELECT distanza_km INTO distanza_tratta
        FROM tratta
        WHERE id = NEW.id_tratta;

        IF distanza_tratta IS NOT NULL THEN
            orario_previsto_arrivo := NEW.data_partenza + (distanza_tratta / 100.0 * INTERVAL '1 hour');
            IF NEW.data_arrivo > (orario_previsto_arrivo + INTERVAL '2 hours') THEN
                UPDATE biglietto
                SET prezzo = 0.00
                WHERE id_corsa = NEW.id;
```

```
        END IF;
    END IF;
  END IF;

  RETURN NEW;
END;
$$ LANGUAGE plpgsql;


CREATE TRIGGER trigger_aggiorna_prezzo_biglietto_corsa
  AFTER UPDATE ON corsa
  FOR EACH ROW
  WHEN (NEW.stato = 'Annullato' OR NEW.stato = 'In ritardo')
  EXECUTE FUNCTION aggiorna_prezzo_biglietto_corsa();
```

Si nota che in caso di corsa annullata, il biglietto verrà completamente rimborsato, mentre il ritardo comporta ad una modifica del biglietto a seconda delle ore di ritardo, l'orario previsto per l'arrivo invece viene aggiornato applicando una formula di spazio/tempo, considerando una velocita di 100km/h. Se l'orario previsto supera di due ore, il biglietto sarà completamente rimborsato.
Mentre la gestione del prezzo e successivamente della scontistica del biglietto è data dalla seguente funzione e trigger

```
CREATE OR REPLACE FUNCTION calcola_prezzo_scontato()
RETURNS TRIGGER AS $$
DECLARE
  sconto_percentuale INTEGER DEFAULT 0;
  prezzo_base NUMERIC(8,2);
BEGIN
  SELECT t.prezzo_fisso INTO prezzo_base
  FROM corsa c
  JOIN tratta t ON c.id_tratta = t.id
  WHERE c.id = NEW.id_corsa;

  IF prezzo_base IS NULL THEN
    prezzo_base := COALESCE(NEW.prezzo, 0);
  END IF;

  IF NEW.id_abbonamento_attivo IS NOT NULL THEN
    SELECT a.perc_offerta INTO sconto_percentuale
    FROM abbonamento_attivo aa
    INNER JOIN abbonamento a ON aa.id_abbonamento = a.id
    WHERE aa.id = NEW.id_abbonamento_attivo
      AND (aa.data_fine IS NULL OR aa.data_fine >= CURRENT_DATE);

    IF sconto_percentuale IS NULL THEN
      sconto_percentuale := 0;
    END IF;
```

```
    END IF;

    NEW.prezzo := prezzo_base * (1 - sconto_percentuale::NUMERIC / 100);

    RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_calcola_prezzo_biglietto
    BEFORE INSERT ON biglietto
    FOR EACH ROW
    EXECUTE FUNCTION calcola_prezzo_scontato();
```

La funzione recupera il prezzo fisso per ogni tratta presente sulla corsa attiva, somma il valore e controlla se il nuovo biglietto ha associato un abbonamento attivo, in caso positivo, allora recupererà il campo perc_offerta e il totale del biglietto viene aggiornato a seconda della scontistica, applicando la formula sopra scritta, restituendo il nuovo prezzo, in caso negativo, il valore sarà uguale solamente alla somma.

```
CREATE TABLE punto_corsa (
    id serial PRIMARY KEY,
    id_corsa integer NOT NULL REFERENCES corsa(id) ON DELETE CASCADE,
    id_stazione integer NOT NULL REFERENCES stazione(id),
    n_punto integer NOT NULL DEFAULT 0,
    tipologia varchar NOT NULL DEFAULT 'Fermata',
    orario_arrivo timestamp,
    orario_partenza timestamp,
    CONSTRAINT punto_corsa_id_corsa_n_punto_key UNIQUE (id_corsa, n_punto),
    CONSTRAINT check_punto_tipologia CHECK (
        tipologia IN ('Partenza', 'Fermata', 'Cambio', 'Arrivo')
    )
);
```

La tabella ‘punto_corsa’ invece gestisce, data una corsa, in quali stazioni transita, dalla partenza, se queste sono un cambio di treno, una fermata oppure l’arrivo. È stato scritto il seguente trigger per evitare che una corsa abbia un numero maggiore di ‘Partenza’ o ‘Arrivo’. Questa struttura è stata progettata specificamente per soddisfare il requisito di gestione degli scali e dei cambi, permettendo di modellare tratte più complesse e avere una gestione più completa.

```
CREATE OR REPLACE FUNCTION check_partenza_arrivo()
RETURNS trigger AS $$
BEGIN
    IF NEW.tipologia = 'Partenza' THEN
        IF EXISTS (SELECT 1 FROM punto_corsa WHERE id_corsa = NEW.id_corsa AND tipologia =
'Partenza' AND id <> NEW.id) THEN
```

```
        RAISE EXCEPTION 'Esiste già un punto di partenza per questa corsa.';
      END IF;
   END IF;
   IF NEW.tipologia = 'Arrivo' THEN
       IF EXISTS (SELECT 1 FROM punto_corsa WHERE id_corsa = NEW.id_corsa AND tipologia =
'Arrivo' AND id <> NEW.id) THEN
         RAISE EXCEPTION 'Esiste già un punto di arrivo per questa corsa.';
      END IF;
   END IF;
   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_check_partenza_arrivo
BEFORE INSERT OR UPDATE ON punto_corsa
FOR EACH ROW EXECUTE PROCEDURE check_partenza_arrivo();
```

Infine, è stato scritto un trigger che impedisce che lo stesso mezzo sia in due corse diverse nello stesso momento.

```
CREATE OR REPLACE FUNCTION check_mezzo_unico_per_intervallo()
RETURNS TRIGGER AS $$
BEGIN
   IF NEW.id_mezzo IS NOT NULL THEN
      IF EXISTS (
         SELECT 1
         FROM corsa
         WHERE id_mezzo = NEW.id_mezzo
           AND id <> NEW.id
                  AND (NEW.data_partenza, NEW.data_arrivo) OVERLAPS (data_partenza,
data_arrivo)
      ) THEN
         RAISE EXCEPTION 'Il mezzo è già su un"altra tratta';
      END IF;
   END IF;

   RETURN NEW;
END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER trigger_check_mezzo_unico
   BEFORE INSERT OR UPDATE ON corsa
   FOR EACH ROW
   EXECUTE FUNCTION check_mezzo_unico_per_intervallo();
```

Nel caso in cui venisse inserito lo stesso mezzo su un’altra corsa, controllando se il nuovo record si sovrappone tra gli orari di data_partenza e data_arrivo, verrà lanciata l’eccezione “Il mezzo è già su un’altra tratta”, in modo tale che l’operazione non venga eseguita con successo.

Indici

Gli indici SQL sono stati creati in modo tale da permettere al sistema di recuperare righe specifiche da una tabella in modo efficiente, senza dover scansionare l'intera collezione e per ottimizzare le prestazioni delle query più frequenti, in particolare nelle operazioni di ricerca, di filtro e join, riducendo i tempi di accesso ai dati.
L'adozione degli indici non è priva di costi. Se da un lato migliorano drasticamente le prestazioni in lettura, dall'altro introducono un overhead nelle operazioni di scrittura. Ogni volta che si modifica un dato nella tabella principale, il DBMS deve aggiornare anche tutti gli indici associati per mantenerli sincronizzati. Per questo motivo, la progettazione degli indici richiede un'analisi accurata del carico di lavoro, in un sistema OLTP (Online Transaction Processing) come quello ferroviario, caratterizzato da frequenti scritture, è fondamentale evitare la sovra-indicizzazione, limitandosi a coprire solo le colonne utilizzate frequentemente nelle clausole condizionali e di join.

```
CREATE INDEX IF NOT EXISTS idx_biglietto_id_corsa ON biglietto(id_corsa);
CREATE INDEX IF NOT EXISTS idx_biglietto_cf ON biglietto(cf_passeggero);
CREATE INDEX IF NOT EXISTS idx_corsa_id_tratta ON corsa(id_tratta);
CREATE INDEX IF NOT EXISTS idx_stazione_id_regione ON stazione(id_regione);
CREATE INDEX IF NOT EXISTS idx_stazione_nome ON stazione(nome_stazione);
CREATE INDEX IF NOT EXISTS idx_passeggero_email ON passeggero(email);
CREATE INDEX IF NOT EXISTS idx_passeggero_num_cliente ON passeggero(num_cliente);
CREATE INDEX IF NOT EXISTS idx_stazione_regione_nome ON stazione(id_regione, nome_stazione);
```

View

Una Vista (View) è definita come una relazione virtuale derivata dal risultato di una query SQL predefinita. A differenza delle tabelle base, che memorizzano fisicamente i dati su disco, una vista standard non contiene dati propri, ma si comporta come come una "finestra" logica che presenta i dati in un formato aggregato o semplificato, calcolando il risultato dinamicamente al momento dell'accesso.
L'utilizzo delle viste nel progetto risponde a due esigenze architetturali fondamentali, l'incapsulamento della complessità, nascondono la complessità delle join sottostanti , offrendo all'applicazione un'interfaccia di lettura pulita e intuitiva e la presenza di valori calcolabili, permettendo così di esporre solo sottoinsiemi specifici di dati senza concedere accesso diretto alle tabelle.
Le due tipologie di viste sviluppate, presenti sotto, sono la vista standard per il monitoraggio in tempo reale e la vista materializzata per l'analisi statistica.
La vista "vista_posti_disponibili" è stata scritta per interrogare e sapere lo stato operativo del sistema in tempo reale. Essa aggrega i dati provenienti da tre entità differenti per fornire una metrica calcolata per la disponibilità rimanente dei posti a

sedere.
La query sfrutta una "Left Join" sulla tabella biglietto per includere nel conteggio anche le corse che non hanno ancora registrato alcuna vendita per avere una panoramica totale delle corse. Il calcolo presente restituisce la differenza tra la disponibilità dichiarata del mezzo e i titoli di viaggio emessi, offrendo al servizio di frontend o di backend un dato atomico e sempre aggiornato senza richiedere logiche di calcolo lato applicazione.

```
CREATE OR REPLACE VIEW vista_posti_disponibili AS
SELECT
    c.id AS id_corsa,
    m.codice_mezzo,
    m.capienza,
    COUNT(b.id) AS biglietti_venduti,
    (m.capienza - COUNT(b.id)) AS posti_disponibili
FROM corsa c
JOIN mezzo m ON m.id = c.id_mezzo
LEFT JOIN biglietto b ON b.id_corsa = c.id
GROUP BY c.id, m.codice_mezzo, m.capienza;
```

La Materialized view raccoglie varie statistiche sulle corse effettuate, aggregando dati sulle vendite dei biglietti, lo stato delle corse, le date di partenza/arrivo e le informazioni sulle stazioni di partenza e di arrivo. È stata preferita la costruzione di una view "materializzata" piuttosto che una standard poiché molto più veloce per la lettura di statistiche e informazioni su grandi moli di dati, dato che i dati vengono salvati su una tabella effettiva e non vengono ricalcolati ogni volta. Lo svantaggio è che per aggiornare la tabella, bisogna lanciare la funzione di refresh.

```
CREATE MATERIALIZED VIEW IF NOT EXISTS mv_statistiche_corse AS
SELECT
    c.id,
    c.stato,
    c.data_partenza,
    c.data_arrivo,
    COUNT(b.id) as num_biglietti,
    SUM(b.prezzo) as ricavo_totale,
    AVG(b.prezzo) as prezzo_medio,
    t.distanza_km,
    sp.nome_stazione as stazione_partenza,
    sa.nome_stazione as stazione_arrivo
FROM corsa c
JOIN tratta t ON c.id_tratta = t.id
JOIN stazione sp ON t.id_stazione_partenza = sp.id
JOIN stazione sa ON t.id_stazione_arrivo = sa.id
LEFT JOIN biglietto b ON c.id = b.id_corsa
GROUP BY c.id, c.stato, c.data_partenza, c.data_arrivo, t.distanza_km, sp.nome_stazione,
sa.nome_stazione;
```

```
CREATE UNIQUE INDEX IF NOT EXISTS idx_mv_statistiche_corse_id ON mv_statistiche_corse(id);

CREATE OR REPLACE FUNCTION refresh_statistiche_corse()
RETURNS void AS $$
BEGIN
   REFRESH MATERIALIZED VIEW CONCURRENTLY mv_statistiche_corse;
END;
$$ LANGUAGE plpgsql;
```

Sulla vista materializzata è stato inoltre creato un indice univoco e fondamentale, “idx_mv_statistiche_corse_id”, questo non serve solo a velocizzare le ricerche per l’identificativo della corsa, ma è un prerequisito tecnico per abilitare l'aggiornamento concorrente attraverso il comando non bloccante “REFRESH MATERIALIZED VIEW CONCURRENTLY”, senza questo indice, l'operazione di refresh richiederebbe un lock esclusivo sulla tabella, bloccando appunto tutte le letture fino al termine dell'aggiornamento. Grazie all'opzione “CONCURRENTLY”, invece, il database calcola la differenza tra i vecchi e i nuovi dati e applica solo le modifiche necessarie, permettendo agli utenti di continuare a consultare le statistiche anche mentre il sistema sta ricalcolando i report in background.

Select Query

Sono state scritte le seguenti sei query per mostrare delle operazioni tipiche sul database progettato.

1)

```
SELECT
   c.id AS id_corsa,
   s.nome_stazione,
   pc.n_punto ,
   pc.orario_arrivo,
   pc.orario_partenza
FROM corsa c
JOIN punto_corsa pc ON pc.id_corsa = c.id
JOIN stazione s ON s.id = pc.id_stazione
WHERE c.stato = 'Regolare' and id_corsa = 4
ORDER BY c.id, pc.n_punto;
```

La query ha l'obiettivo di restituire l'itinerario completo di una specifica corsa con stato “Regolare”, in questo caso dall'ID 4, ordinando sequenzialmente tutte le tappe previste.

ID Corsa	Stazione	N. Punto	Orario arrivo	Orario partenza
4	Napoli Centrale	0	2024-12-20 14:15:00	2024-12-20 14:15:00
4	Napoli P. Garibaldi	1	2024-12-20 16:00:00	2024-12-20 16:03:00
4	Bari Centrale	2	2024-12-20 19:05:00	2024-12-20 23:30:00.000

2)

```
SELECT DISTINCT
  p.codice_fiscale,
  p.nominativo
FROM passeggero p
JOIN biglietto b ON p.codice_fiscale = b.cf_passeggero
JOIN corsa c ON b.id_corsa = c.id
WHERE c.id_tratta = 41
```

Questa query ha l'obiettivo di profilare l'utenza di una specifica tratta ferroviaria.

Codice Fiscale	Nominativo
FRNGPP88E30H501W	Giuseppe Ferrari
NRIANR75D25M301V	Anna Neri
FRNMLN91L01H501C	Milena Ferrari
VRDLCA90C20L219T	Luca Verdi
BNCGNN85B15F205X	Giulia Bianchi
RSSMRA80A01H501U	Mario Rossi
BNCMRC87G05L219Z	Marco Bianchi
RSSPLA92F10I001Y	Paola Rossi

3)

```
SELECT
  p.codice_fiscale,
  p.nominativo,
  COUNT(DISTINCT b.id_corsa) AS numero_corse_effettuate,
  SUM(b.prezzo) AS totale_speso
FROM passeggero p
JOIN biglietto b ON p.codice_fiscale = b.cf_passeggero
GROUP BY p.codice_fiscale, p.nominativo;
```

La seguente query serve per mostrare quali utenze hanno utilizzato il servizio, qual è il numero di corse effettuate e il totale speso.

Codice Fiscale	Nominativo	Numero_corse_effettuate	Totale_speso
BNCGNN85B15F205X	Giulia Bianchi	3	56.95
BNCMRC87G05L219Z	Marco Bianchi	2	27.00
FRNGPP88E30H501W	Giuseppe Ferrari	2	22.95
FRNMLN91L01H501C	Milena Ferrari	2	27.00
NRIANR75D25M301V	Anna Neri	2	24.30
NRIFRC89I15M301B	Francesco Neri	2	0.00
RSSMRA80A01H501U	Mario Rossi	3	105.30
RSSPLA92F10I001Y	Paola Rossi	2	21.60
VRDLCA90C20L219T	Luca Verdi	1	43.20
VRDSRA83H20F205A	Sara Verdi	1	0.00

4)

```
SELECT
    s.nome_stazione,
    r.nome_regione,
    COUNT(c.id) AS numero_partenze_programmate
FROM stazione s
JOIN regione r ON s.id_regione = r.id
JOIN tratta t ON t.id_stazione_partenza = s.id
JOIN corsa c ON c.id_tratta = t.id
WHERE c.stato = 'Regolare'
GROUP BY s.nome_stazione, r.nome_regione;
```

Questa query genera un'ordinazione delle stazioni in base al volume di traffico in partenza.

5)

```
SELECT
  aa.id,
  aa.cf_passeggero,
  p.nominativo,
  aa.data_fine
FROM abbonamento_attivo aa inner join passeggero p on aa.cf_passeggero = p.codice_fiscale
WHERE aa.data_fine BETWEEN CURRENT_DATE AND (CURRENT_DATE + INTERVAL
'7 days')
```

Questa query serve a monitorare le scadenze imminenti degli abbonamenti nell'intervallo di sette giorni, utilizzando la funzione di PostgreSQL Interval, in modo da poter notificare possibilmente tramite un sistema di notifiche i dipendenti per scegliere di rinnovare o meno il piano welfare.

ID	CF_Passeggero	Nominativo	Data fine
13	VRDLRD96D24G964N	Alfredo Verdosci	2025-12-11

Nel caso in cui la query non restituisse risultati, eliminare l'abbonamento attivo con ID 13 ed eseguire la seguente insert

```
INSERT INTO abbonamento_attivo (cf_passeggero, id_abbonamento, data_inizio, data_fine)
VALUES (VRDLRD96D24G964N, 2, CURRENT_DATE - INTERVAL '1 month', CURRENT_DATE + INTERVAL '3 days');
```

6)

```
SELECT c.id, SUM(t.distanza_km) AS totale_km_percorsi
FROM corsa c
JOIN tratta t ON c.id_tratta = t.id
WHERE c.stato = 'Regolare'
GROUP BY c.id
```

La seguente query calcola il chilometraggio totale di una corsa con stato "Regolare", considerando tutte le tratte presenti.

ID	Totale_km_percorsi
4	374.0

Python

Lo script python 'insert_data.py' implementa l'intregazione dell'inserimento di dati attraverso un processo ETL (Extract, Transform, Load) per popolare le tabelle regione e stazione a partire da un dataset CSV esterno.

La connessione al database PostgreSQL, presente come container Docker, è gestita tramite la libreria psycopg2. I parametri di accesso sono passati in chiaro in un dizionario di configurazione.

Stazioni italane.csv è file di riferimento presente nella root del progetto nella cartella "utils".

```
DB_CONFIG = {
    'host': 'localhost',
    'port': 5432,
    'database': 'postgres',
    'user': 'postgres',
    'password': 'admin'
}

CSV_FILE = os.path.join(os.path.dirname(os.path.dirname(os.path.dirname(__file__))),
                        'utils', 'Stazioni italiane.csv')
```

La funzione read_csv() si occupa del parsing del file. Utilizzando la funzione DictReader della libreria csv, il metodo legge il dataset riga per riga, ignorando eventuali dati non validi, come i codici regione fuori dal mapping. È presente una mappatura semantica in modo tale che il dataset originale utilizza codici numerici per

le regioni che partono da 1 fino a 20, vengono decodificati in nomi leggibili con i nomi effettivi delle regioni tramite il dizionario REGIONI_MAPPING. Questa trasformazione arricchisce il dato indicizzato prima dell'inserimento.

```
REGIONI_MAPPING = {
    1: 'Lombardia',
    2: 'Liguria',
    3: 'Piemonte',
    4: 'Valle d\'Aosta',
    5: 'Lazio',
    6: 'Umbria',
    7: 'Molise',
    8: 'Emilia-Romagna',
    9: 'Trentino-Alto Adige',
    10: 'Friuli-Venezia Giulia',
    11: 'Marche',
    12: 'Veneto',
    13: 'Toscana',
    14: 'Sicilia',
    15: 'Basilicata',
    16: 'Puglia',
    17: 'Calabria',
    18: 'Campania',
    19: 'Abruzzo',
    20: 'Sardegna'
}

def read_csv():
    stazioni = []
    try:
        with open(CSV_FILE, 'r', encoding='utf-8') as file:
            reader = csv.DictReader(file)
            for row in reader:
                region_num = int(row['region'])
                if 1 <= region_num <= 20:
                    stazioni.append({
                        'code': row['code'],
                        'region': region_num,
                        'long_name': row['long_name'],
                        'short_name': row['short_name'],
                        'latitude': float(row['latitude']),
                        'longitude': float(row['longitude'])
                    })
        return stazioni
    except FileNotFoundError:
        print(f"File non trovato: {CSV_FILE}")
        raise
    except Exception as e:
        print(f"Errore nella lettura del CSV: {e}")
        raise
```

La fase di scrittura nel database è suddivisa in due step sequenziali per rispettare i vincoli di integrità referenziale.
Viene eseguito prima l'inserimento Regioni, la funzione "insert_regioni" popola la tabella regione e restituisce una mappa regione_to_id che associa il codice originale al nuovo ID generato dal database. Questo evita di dover eseguire query di lookup ripetute per ogni stazione.
Infine avviene l'inserimento delle stazioni come batch, la funzione "insert_stazioni" utilizza il metodo ottimizzato execute_values di psycopg2.extras, in modo tale, queesto approccio invia i dati in blocchi da 100 record invece di inserirne uno per volta. Questa tecnica riduce drasticamente la saturazione di rete e il carico transazionale, prevenendo possibili colli di bottiglia e time-out del database durante l'elaborazione di grandi volumi di dati.

```
def insert_regioni(conn, stazioni):
    cursor = conn.cursor()

    regioni = set(stazione['region'] for stazione in stazioni if 1 <= stazione['region'] <= 20)

    cursor.execute("SELECT id, nome_regione FROM regione")
    regioni_esistenti = {row[1]: row[0] for row in cursor.fetchall()}

    regione_to_id = {}

    for regione_num in sorted(regioni):
        nome_regione = REGIONI_MAPPING.get(regione_num, f"Regione {regione_num}")

        if nome_regione in regioni_esistenti:
            regione_to_id[regione_num] = regioni_esistenti[nome_regione]
        else:
            cursor.execute(
                "INSERT INTO regione (nome_regione) VALUES (%s) RETURNING id",
                (nome_regione,)
            )
            new_id = cursor.fetchone()[0]
            regione_to_id[regione_num] = new_id

    conn.commit()
    cursor.close()
    return regione_to_id

def insert_stazioni(conn, stazioni, regione_to_id):
    cursor = conn.cursor()
    dati_stazioni = []
    for stazione in stazioni:
        region_num = stazione['region']
        if 1 <= region_num <= 20 and region_num in regione_to_id:
            id_regione = regione_to_id[region_num]
            dati_stazioni.append((
                stazione['long_name'],
```

```
            id_regione
        ))

    execute_values(
        cursor,
        "INSERT INTO stazione (nome_stazione, id_regione) VALUES %s",
        dati_stazioni,
        template=None,
        page_size=100
    )
    conn.commit()
    print(f"Inserite {len(dati_stazioni)} stazioni nel database")
    cursor.close()
```

Nel metodo main, dove vengono eseguite le funzioni, la logica è incapsulata in un blocco try-except-finally che garantisce l'atomicità dell'operazione.

```
def main():
    stazioni = read_csv()
    conn = connect_db()

    try:
        regione_to_id = insert_regioni(conn, stazioni)
        insert_stazioni(conn, stazioni, regione_to_id)

    except Exception as e:
        print(f"Errore durante l'inserimento: {e}")
        conn.rollback()
        raise
    finally:
        conn.close()

if __name__ == "__main__":
    main()
```

È importante notare l'uso del blocco try-except combinato con il rollback.
Questa gestione garantisce l'atomicità del processo di importazione, ovvero se anche un solo record del CSV risulta corrotto o viola i vincoli del database oppure della formattazione del dizionario presente, l'intera transazione viene annullata, evitando di lasciare il database in uno stato parzialmente popolato o inconsistente.

Campi di applicazione
(Descrivere gli ambiti di applicazione dell'elaborato progettuale e i vantaggi derivanti della sua applicazione)**:**

Il sistema progettato, TransferOffer, pur nascendo come piattaforma di Corporate Welfare, presenta caratteristiche architetturali che lo rendono applicabile in diversi contesti operativi. La separazione tra la logica di persistenza del database e la

presenza di dati esterni forniti dai provider, permette di individuare diversi ambiti di impiego.
Uno di questi, è la possibilità di ricerca da parte delle Risorse Umane o da chi chi gestisce la parte del Welfare aziendale. Il sistema non si limita alla vendita di titoli di viaggio, ma funge da strumento di fidelizzazione per i dipendenti. L'implementazione della tabella abbonamento e delle logiche di sconto dinamico (Bronze, Silver, Gold, Platinum) permette all'azienda di automatizzare l'erogazione dei benefit. Invece di gestire rimborsi delle spese manuali o convenzioni statiche, il sistema applica le regole di business direttamente al momento della transazione. Questo modello è applicabile a qualsiasi grande azienda che necessiti di gestire la mobilità dei propri dipendenti con politiche di prezzo personalizzate basate sull'anzianità, sul ruolo o attraverso premi produzione.
Un altro ambito rilevante è quello dell'integrazione dati. Il sistema dimostra come un database relazionale possa fungere da "collettore" e normalizzatore di informazioni provenienti da fonti esterni, come ad esempio da API di Italo e Trenitalia. Attraverso le tabelle stazione, tratta e punto_corsa, il sistema astrae la complessità dei fornitori esterni, offrendo all'utente un'interfaccia unificata. Questo approccio è fondamentale in contesti come applicativi per la gestione viaggi, dove è necessario comparare offerte non omogenee e presentarle in un formato standardizzato, con gestione trasparente di cambi e scali indipendentemente dall'operatore, in questo modo è presentato come un pacchetto “unico” dove l’utente ha solo bisogno di effettuare il pagamento e non cercare la migliore offerta.
Un altro punto importante è quello della tutela del consumatore. Il sistema è in grado di reagire in tempo reale a eventi esterni, come ad esempio un ritardo comunicato dal provider, ed eseguire azioni finanziarie immediate, come il rimborso del credito parziale o totale. Questa logica di "Smart Contract" implementata su database relazionale è applicabile ovunque ci sia la necessità di garantire rimborsi certi e immediati al verificarsi di condizioni oggettive, eliminando la burocrazia tradizionale che porta spesso a tempistiche lunghe e tediose.

Valutazione dei risultati
(Descrivere le potenzialità e i limiti ai quali i risultati dell’elaborato sono potenzialmente esposti):

L’architettura sviluppata dimostra come un approccio rigoroso alla gestione della persistenza possa generare valore tangibile per un'organizzazione. Le principali potenzialità emerse sono diverse.
L'implementazione di logiche di business direttamente nel database, presenti tramite trigger e vincoli di integrità, rappresenta il punto di focus del sistema. La capacità di calcolare automaticamente i prezzi e i possibili sconti in base ad un servizio di abbonamento (Bronze, Silver, ecc.) e di processare rimborsi immediati in caso di ritardo (attraverso il trigger trigger_corsa_biglietto) riduce drasticamente il carico amministrativo e burocratico. Questo livello di automazione trasforma il

database non solo per la persistenza, ma anche come servizio attivo dei processi aziendali.
In un contesto transazionale come quello del Welfare Aziendale, la correttezza del dato è prioritaria. La scelta di un RDBMS relazionale, come in questo caso PostgreSQL, garantisce che non si verifichino anomalie contabili, come biglietti venduti senza copertura, prezzi negativi o corse senza posti disponibili. La gestione della concorrenza e dell'inventario locale attraverso la View "vista_posti_disponibili" elimina il rischio di prenotare più posti di quelli disponibili, assicurando che ogni posto venduto sia effettivamente acquistabile nella corsa considerata.
La separazione tra dati operativi e viste analitiche permette di estrarre valore informativo senza rallentare l'operatività. Il sistema è predisposto quindi per fornire reportistica in tempo reale su costi, risparmi e utilizzo dello sconto welfare, supportando la gestione nelle decisioni strategiche o semplicemente per avere delle statistiche di utilizzo del sistema.

Il sistema presenta in ogni caso dei limiti che ne penalizzano l'applicabilità immediata in un ambiente di produzione reale.
Poiché è stato sviluppato per un contesto accademico, c'è stata una semplificazione necessaria della realtà in mancanza delle conoscenze necessarie. In un sistema di produzione reale richiederebbe la gestione di una complessità molto superiore e molto più tecnico, includendo aspetti qui non trattati come la gestione avanzata della sicurezza e della privacy (come ad esempio il GDPR) per i dati sensibili dei dipendenti, manca un'integrazione con sistemi di pagamento bancari reali e casi molteplici come ad esempio scioperi, cambi treno improvvisi, servizi per portatori di handicap, ecc., che richiederebbero logiche di business ancora più articolate e rischiando di uscire fuori dall'obiettivo principale.
L'architettura, inoltre, si basa sull'assunzione di ricevere dati precisi e puntuali da fornitori esterni. Questo introduce una possibilità di errori dovuta dall'esterno. Se le API dei fornitori dovessero subire disservizi o cambiare formato, il sistema interno non potrebbe aggiornare le corse o rilevare i ritardi, bloccando di fatto tutte le funzionalità di vendita e rimborso. In uno scenario reale, sarebbe necessario implementare meccanismi di fallback o di possibili ridondanze che sono escluse dallo scopo di questa tesi.
Infine, lo script in Python non è generalizzato e non è pensato per essere eseguito più volte, una variazione nello schema del file sorgente, come un cambio nome delle colonne, o un formato totalmente differente, richiederebbe una riscrittura completa del parser. In un ambiente aziendale sarebbe necessario sviluppare procedure ETL più robuste, capaci di validare dinamicamente l'input e gestire errori di formato senza interrompere il flusso di importazione.