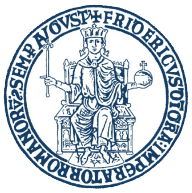


UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

Documentazione DB - Sistema Di Gestione Del Ciclo Di Vita Di Una Pagina Wiki

Giuseppe Pollio, Mario Lombardo

Anno accademico 2023-2024

Contents

1 Modello Concettuale

1.1 Analisi dei requisiti

In questa sezione si analizza la traccia in maniera specifica con lo scopo di definire le funzionalità che la base di dati deve soddisfare. Individueremo le entità e le associazioni del mini-word, producendo la prima versione del modello concettuale che sarà poi rielaborato nelle fasi successive della progettazione.

"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 del ciclo di vita di una pagina di una wiki"

"Una pagina di una wiki ha un titolo e un testo. Ogni pagina è creata da un determinato autore. Il testo è composto di una sequenza di frasi. Il sistema mantiene traccia anche del giorno e ora nel quale la pagina è stata creata. Una frase può contenere anche un collegamento. Ogni collegamento è caratterizzato da una frase da cui scaturisce il collegamento e da un'altra pagina destinazione del collegamento."

"Il testo può essere modificato da un altro utente del sistema, che seleziona una o più delle frasi, scrive la sua versione alternativa (modifica) e prova a proporla"

Dall'introduzione individuiamo il mini-world da rappresentare, ovvero un *"Sistema informativo per la gestione del ciclo di vita di una pagina di una wiki"*, e iniziamo a riconoscere la prima entità: **Page** avente come attributi `title` e `creation_date`. Inoltre una pagina della Wiki deve essere composta da un testo, a sua volta composto da una sequenza di frasi, e questo testo deve poter essere modificabile da un utente, individuiamo così un entità associata alla pagina: **PageText** contenente come attributi, la riga di testo `text` e un indice per l'ordinamento delle righe di testo `order_num`. Invece di utilizzare un singolo attributo per salvare tutto il contenuto di una pagina l'utilizzo dell'entità **PageText** ottimizza molto le operazioni di modifica del testo poichè lavora su singole righe di testo invece di lavorare su l'intero contenuto. Una frase del testo di una pagina puo' contenere un collegamento ad un altra pagina, otteniamo questo comportamento aggiungendo all'entità **PageText** l'attributo `link` (attributo parziale) tenendo traccia del `page_id` della Pagina alla quale ci riferiamo. Andiamo ad utilizzare il formato {`page_id:riga_di_testo`} quando una riga di testo è un collegamento, sarà l'applicativo a mostrare solo la riga_di_testo e gestire il collegamento alla pagina.

"La modifica proposta verrà notificata all'autore del testo originale la prossima volta che utilizzerà il sistema. L'autore potrà vedere la sua versione originale e la modifica proposta. Egli potrà accettare la modifica (in quel caso la pagina originale diventerà ora quella con la modifica apportata), rifiutare la modifica (la pagina originale rimarrà invariata). La modifica proposta dall'autore verrà memorizzata nel sistema e diventerà subito parte della versione corrente del testo. Il sistema mantiene memoria delle modifiche proposte e anche delle decisioni dell'autore (accettazione o rifiuto)."

"Nel caso in cui si fossero accumulate più modifiche da rivedere, l'autore dovrà accettarle o rifiutarle tutte nell'ordine in ordine dalla più antica alla più recente"

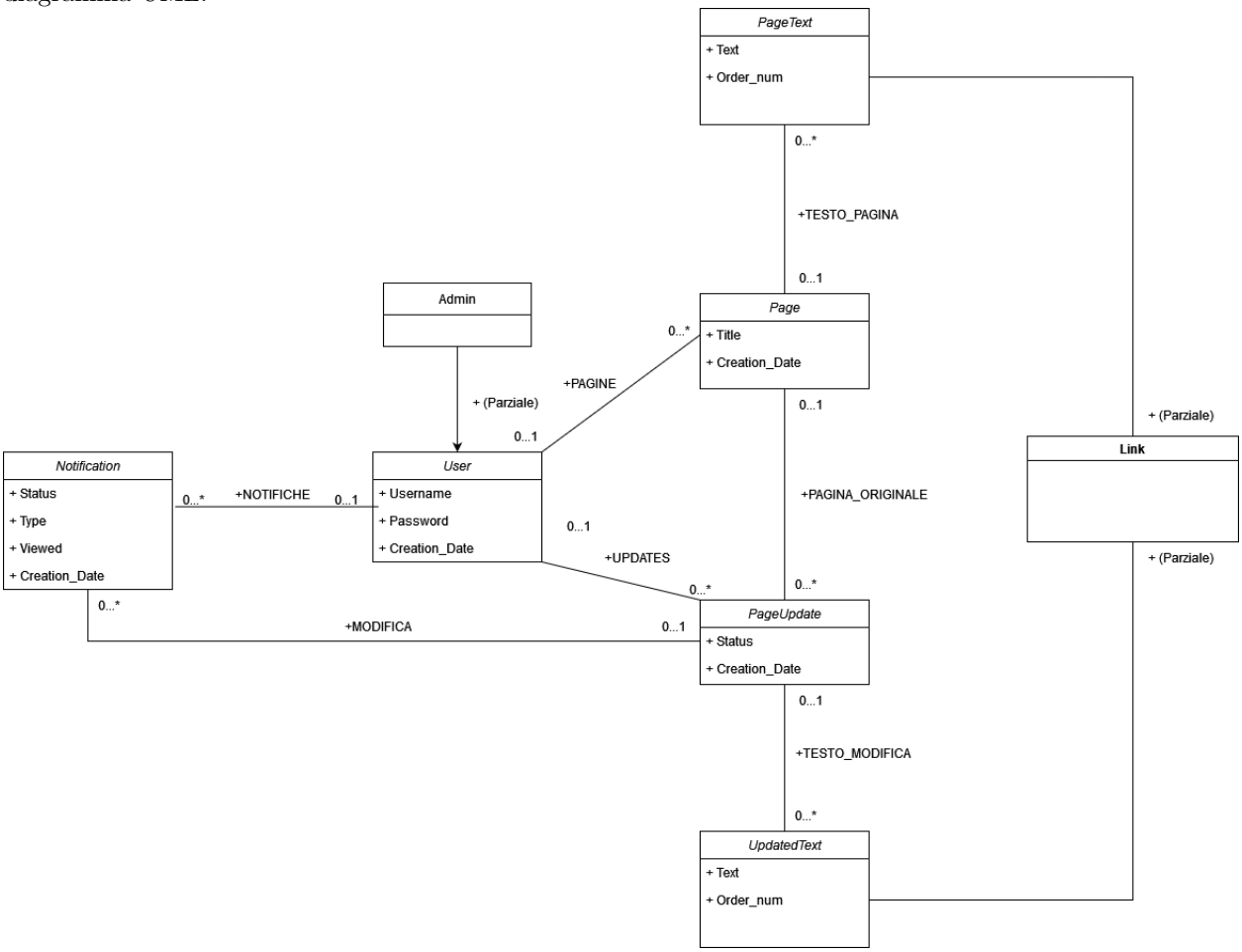
"Gli utenti generici del sistema potranno cercare una pagina e il sistema mostrerà la versione corrente del testo e i collegamenti. Gli autori dovranno prima autenticarsi fornendo la propria login e password. Tutti gli autori potranno vedere tutta la storia di tutti i testi dei quali sono autori e di tutti quelli nei quali hanno proposto una modifica"

Sarà necessario tenere traccia degli utenti della Wiki per poter permettere loro pubblicare nuove pagine e gestire le pagine esistenti (richiedendo modiche agli autori), da questo andiamo a creare l'entità **User**, alla quale sarà possibile accedere al sistema tramite una **password** e un **username**. Inoltre il sistema tiene conto del fatto che un utente possa essere un Gestore della Wiki tramite l'attributo `admin` (attributo parziale).

All'entità **Page** individuiamo la necessità di possedere un autore inserendo l'attributo **author**. Le modifiche del testo di una pagina (**Page**) saranno salvate tramite le l'entità **PageUpdate** avente attributi: **status**, che indica lo stato della modifica, e un collegamento all'utente **User** autore della modifica (`author`). Le righe di testo da modificare sono individuate tramite l'entità **UpdatedText** che contiene l'attributo `text` che indica la "nuova riga di testo", un collegamento `link` (attributo parziale) e un collegamento all'entità **PageUpdate**. Quando una modifica viene inoltrata all'autore esso invece deve essere avvisato tramite una notifica, questa verrà gestita dall'entità **Notification** avente come attributi `status` (stato della notifica, `OPEN, CLOSED`), il tipo di notifica `type` (`REQUEST_UPDATE, UPDATE_ACCEPTED, UPDATE_REJECTED`), `viewed` per indicare se la notifica è stata letta, e ovviamente, contiene le relazioni all'utente destinatario **User** e alla modifica del testo **PageUpdate**. In fine *Tutti gli autori potranno vedere tutta la storia di tutti i testi dei quali sono autori e di tutti quelli nei quali hanno proposto una modifica* sarà possibile per via delle relazioni **User** → **Page** usando l'attributo `author` contenuto in **Page**, e **PageUpdate** → **Page** usando l'attributo `page_id` in **PageUpdate**.

1.2 Diagramma (UML)

In seguito alle considerazioni espresse, si 'e prodotto il seguente schema concettuale espresso mediante diagramma UML:



1.3 Dizionari

1.3.1 Dizionario delle Entità

Entità	Descrizione	Attributi
User	Account Utente della wiki.	**Username**(Stringa): nome identificativo dell'account utente. **Password**(Stringa): password necessaria per accedere all'account utente.
Page	Pagina presente sulla wiki.	**Title**(Stringa): Titolo della pagina. **CreationDate**(Data/Timestamp): Data e ora di creazione della pagina.
PageText	Frase di una Pagina (**Page**).	**Text**(Stringa): Contenuto della frase. **Order_num**(Intero): Ordinamento della frase all'interno del testo complessivo.
PageUpdate	Modifica proposta ad pagina da parte di un altro utente.	**Status**(Intero): Stato della richiesta di modifica.
UpdatedText	Nuovo testo proposto durante la modifica (**PageUpdate**).	**Text**(Stringa): Contenuto della riga di testo. **Order_num**(Intero): Ordinamento della frase all'interno del testo complessivo.
Notification	Notifiche di un **Utente (User)**.	**Status**(Intero): Stato della notifica. (`OPEN, CLOSED`) **Viewed**(Booleano): Stato di lettura di una notifica. **Type**(Intero): Tipologia di notifica. (`REQUEST_UPDATE, UPDATE_ACCEPTED, UPDATE_REJECTED`)
Admin	Specializzazione parziale di un utente, un amministratore può modificare ed eliminare le pagine di altri utenti senza dover prima inviare una notifica di accettazione delle modifiche.	
Link	Specializzazione parziale di una riga di testo che rappresentà se una riga di testo è un collegamento o meno.	

1.3.2 Dizionario delle Associazioni

Associazione	Tipologia	Descrizione
Autore Pagina	uno-a-molti	Associa ad ogni pagina (Page) un utente (User) che ne rappresenta l'autore
Autore Modifica	uno-a-molti	Associa ad ogni modifica (PageUpdate) un utente (User) che ne rappresenta l'autore
Testo Pagina	uno-a-molti	Associa ad ogni pagina (Page) tutto il testo (PageText) appartenente a quella determinata pagina.
Testo Modifica	uno-a-molti	Associa ad ogni modifica (PageUpdate) tutto il testo (UpdatedText) appartenente a quella determinata modifica.
Proprietario Notifica	uno-a-molti	Associa ad ogni notifica (Notification) un utente (User) che ne rappresenta l'autore.
PageUpdate Notifica	uno-a-molti	Associa ad ogni notifica (Notification) una modifica (PageUpdate) che ne aiuta a rappresentare il contenuto.
Page PageUpdate	uno-a-molti	Associa ad ogni Modifica proposta (PageUpdate), La pagina (Page) per cui è stata proposta.

1.3.3 Dizionario dei Vincoli

Vincolo	**Tipologia**	**Descrizione**
Di Leggibilità Testuale	Intrarelazionale	Il titolo di una pagina, il nome utente e la password devono avere lunghezza non nulla, poichè fondamentali.
Di Autore	Interrelazionale	Una pagina e una modifica devono sempre avere un autore. Se un utente autore di una pagina viene eliminato, anche la pagina viene eliminata.
Di Sicurezza	Interrelazionale	Una pagina può essere modificata direttamente solo nel caso in cui, l'utente che effettua la modifica è l'autore della pagina oppure un gestore della wiki.
Di Gestione	Interrelazionale	Se esistono utenti almeno uno di essi dovrà essere il gestore della wiki (amministratore).

2 Ristrutturazione del Modello Concettuale

Prima di procedere con lo schema logico è fondamentale riorganizzare il **Diagramma delle Classi** al fine ottimizzare il progetto, rimuovere le generalizzazioni, eliminare gli attributi multivalore, eliminare attributi strutturati, riorganizzare le entità figlie e selezionare gli identificatori appropriati per le nostre entità quando necessario.

2.1 Analisi delle Ridondanze

Con Analisi delle Ridondanze intendiamo il processo di identificazione dei dati duplicati o derivabili da altre informazioni all'interno di una base di dati.
Nel modello concettuale originale non sono presenti ridondanze.

2.2 Analisi delle Generalizzazioni

Nell'Analisi delle Generalizzazioni, l'obiettivo è quello di generalizzare informazioni all'interno della base di dati per ottimizzare le relazioni tra entità.
Nel modello concettuale attuale ci sono due generalizzazioni:

1. La sottoclasse `Admin`, dove non sono presenti attributi e viene utilizzata per definire i permessi di un utente. Questa nel diagramma ristrutturato verrà accorpata all'entità genitore `User` come flag booleana `admin`.
2. La sottoclasse `Link`, usata per capire se una riga di testo è anche un link ad una pagina `Page`, viene eliminata del tutto in quanto un link ad una pagina doveva necessariamente essere una composta da un intera riga di testo, per risolvere questo problema e semplificare l'implementazione della base dati si è optato nell'utilizzo della formattazione `<href='link'>` contenuta nel testo della pagina dove si vuole assegnare un collegamento, così facendo la gestione e rappresentazione dei collegamenti è data dall'applicativo stesso.

2.3 Eliminazione degli attributi Multivalore o Composti

Durante l'Eliminazione degli attributi Multivalore o Composti, si procede a rimuovere dalla struttura del modello concettuale gli attributi che possono avere più di un valore o che sono composti da diverse parti, semplificando così la rappresentazione dei dati.
Nel modello concettuale non sono presenti attributi multivalore o attributi composti.

2.4 Analisi di Entità e Associazioni

Non si è ritenuto opportuno scomporre o accorpare le entità restanti, fanno accezione le sottoclassi `Admin` e `Link` che sono state eliminate e solo la prima è stata sostituita dal nuovo attributo `admin` nell'entità **User**.

2.4.1 Attributi aggiunti in fase di Ristrutturazione

Durante la fase di Ristrutturazione è stato necessario aggiungere dei nuovi attributi ad alcuni campi per andare a coprire tutti i casi d'uso. Le aggiunte sono:

1. L'attributo `admin` nella tabella **User**: Come detto in precedenza, viene introdotto per generalizzare la base dati.
2. L'attributo `creation_date` nella tabella **User**: Viene aggiunto per salvare la data di creazione dell'utente.
3. L'attributo `old_text` nella tabella **PageUpdate**: Viene aggiunto per rappresentare il testo di una pagina prima della modifica; impiegato per tenere traccia della "storia" delle modifiche in una pagina.
4. L'attributo `author` nella tabella **PageText**: Viene aggiunto per identificare l'utente che ha modificato/aggiunto una determinata riga alla pagina.

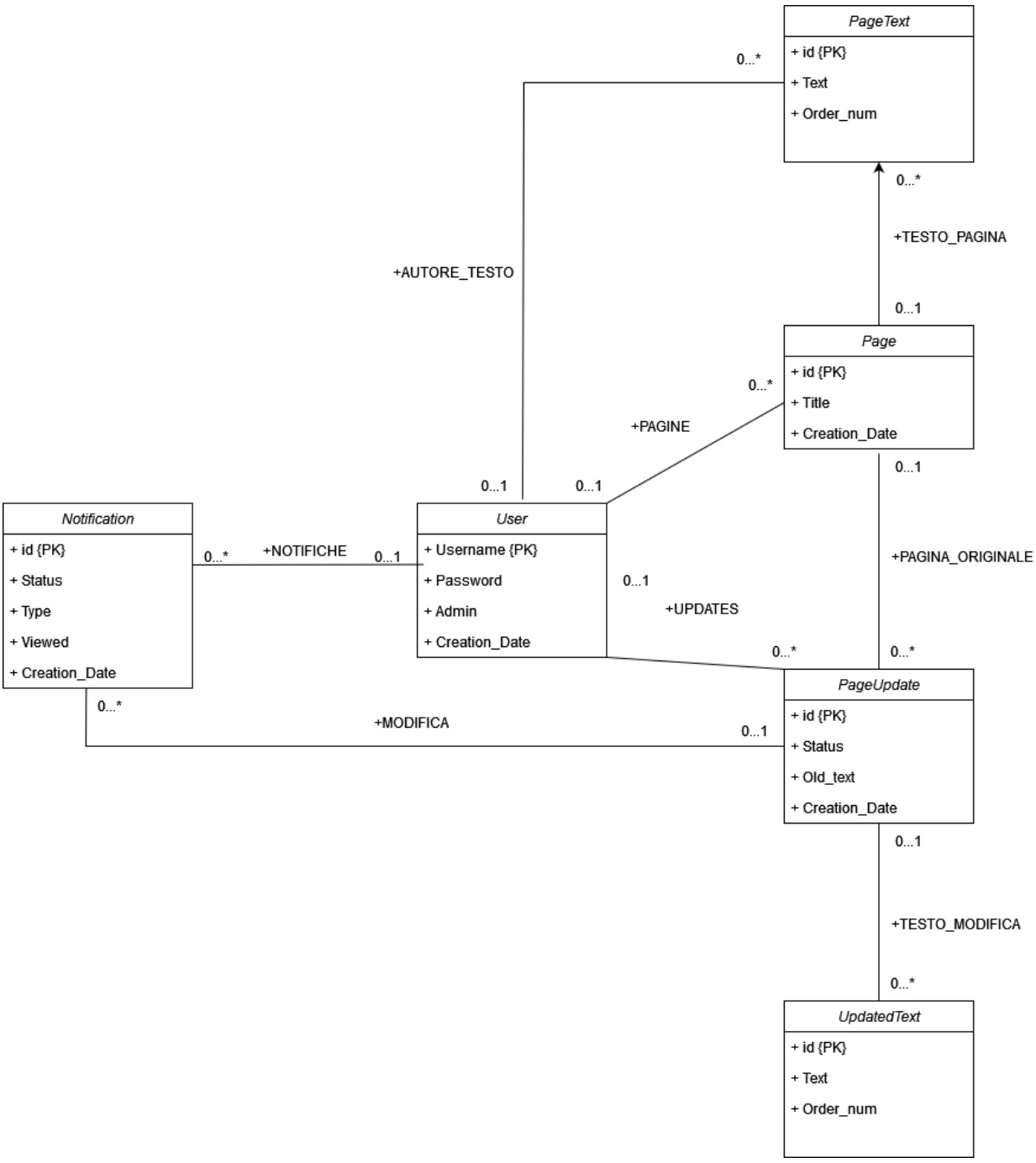
2.4.2 Scelta degli Identificatori Primari

La scelta degli identificatori primari è fondamentale poiché consente di identificare e distinguere univocamente ogni record delle varie tabelle.
Ecco i seguenti nomi delle `PRIMARY KEY` che sono state scelte:

- **User**: Viene scelto **username** come identificatore primario.
- **Notification**: Viene introdotto l'identificativo primario **id**.
- **Page**: Viene introdotto l'identificativo primario **id**.
- **PageText**: Viene introdotto l'identificativo primario **id**.
- **PageUpdate**: Viene introdotto l'identificativo primario **id**.
- **UpdatedText**: Viene introdotto l'identificativo primario **id**.

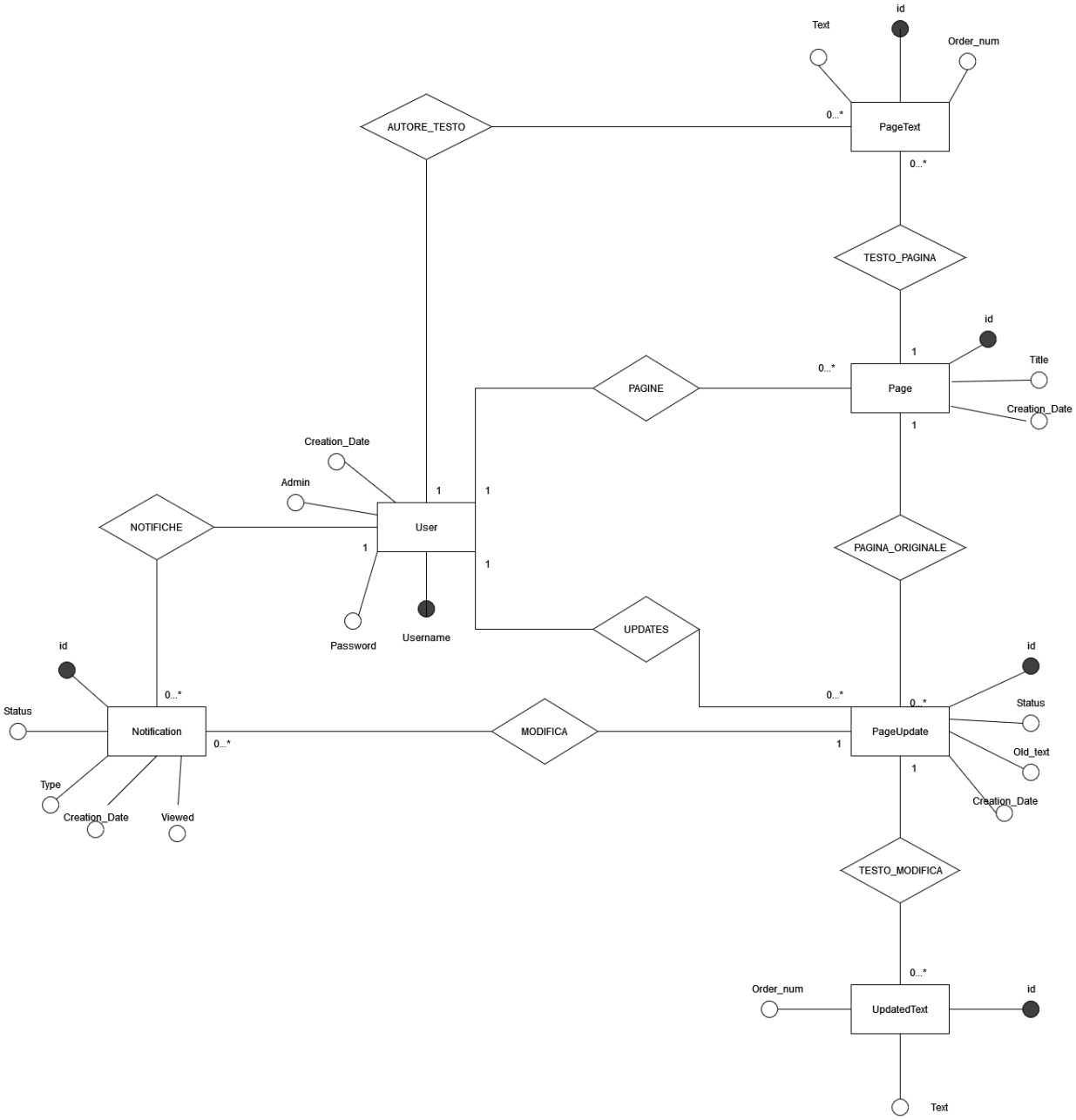
2.4.3 Diagramma UML Ristrutturato

Dopo aver completato la ristrutturazione del *Class Diagram* come descritto delle sezioni precedenti, possiamo mostrare il seguente schema concettuale espresso mediante diagramma UML:



2.4.4 Diagramma ER Ristrutturato

Ecco il diagramma ER della soluzione, prodotto aver completato la ristrutturazione del *Class Diagram* come descritto nelle sezioni precedenti.



3 Modello Logico

3.1 Traduzione di Entità e Associazioni

Avendo completato il modello concettuale possiamo procedere con il *mapping* delle entità e associazioni della soluzione. Per ogni entità del modello ristrutturato definiamo una *relazione equivalente*. Il processo di traduzione per le associazioni è invece più complesso e richiede un'analisi individuale di ogni associazione.

- **User, Page**: sono associazioni *uno-a-molti* che esprimiamo tramite chiave esterna, solo l'entità **Page** avrà come attributo **author**.
- **User, PageUpdate**: sono associazioni *uno-a-molti* che esprimiamo tramite chiave esterna, solo l'entità **PageUpdate** avrà come attributo **author**.
- **PageText, Page**: sono associazioni *uno-a-molti* che esprimiamo tramite chiave esterna, solo l'entità **PageText** avrà come attributo **page_id**.
- **UpdatedText, PageUpdate**: sono associazioni *uno-a-molti* che espriamiamo tramite chiave esterna, solo l'entità **UpdatedText** avrà come attributo **update_id**.
- **PageText, User**: sono associazioni *uno-a-molti* che esprimiamo tramite chiave esterna, solo l'entità **PageText** avrà come attributo **author**.
- **User, Notification**: sono associazioni *uno-a-molti* che esprimiamo tramite chiave esterna, solo l'entità **UpdatedText** avrà come attributo **user**.
- **PageUpdate, Notification**: sono associazioni *uno-a-molti* che esprimiamo tramite chiave esterna, solo l'entità **Notification** avrà come attributo **update_id**.

3.2 Relazioni

Legenda: *ChiavePrimaria*, Chiave Esterna↑, Attributo Nullabile?

- **USER** (*username*, `password`, `admin`, `creation_date`)
- **PAGE** (*id*, `title`, `author`↑, `creation_date`)
  - `author` → USER.username
- **PAGETEXT** (*id*, `order_num`, `text`, `page_id`↑, `author`↑)
  - `page_id` → PAGE.id
  - `author` → USER.username
- **PAGEUPDATE** (*id*, `status`, `old_text`?, `page_id`↑, `author`↑, `creation_date`)
  - `page_id` → PAGE.id
  - `author` → USER.username
- **NOTIFICATION** (*id*, `status`, `type`, `viewed`, `user`↑, `update_id`↑, `creation_date`)
  - `update_id` → UPDATE.id
  - `user` → USER.username
- **UPDATEDTEXT** (*id*, `order_num`, `text`, `update_id`↑)
  - `update_id` → UPDATE.id

4 Modello Fisico

4.1 Domini

Notazioni: I domini usano SNAKE_CASE maiuscolo.

```
CREATE DOMAIN SHORT_TEXT VARCHAR(128)
CHECK(LENGTH(VALUE) > 0)
```

Il dominio `SHORT_TEXT` rappresenta un tipo di testo i cui valori sono vincolati a essere non nulli in termini di lunghezza. Le applicazioni di questo dominio sono:

- Nomi utente.
- Password degli utenti.
- Titolo delle pagine.

Garantiamo in questo modo il rispetto del *Vincolo di Leggibilità Testuale*.

4.2 Tabelle

Di seguito sono indicate le istruzioni DDL necessarie per la creazione e definizione delle tabelle del database relazionale. Poiché queste sono una traduzione diretta delle relazioni definite in precedenza nella sezione sul modello logico, si è preferito utilizzare i commenti solo per chiarire le parti più complessi di SQL usate nelle query di creazione. Per le informazioni sulla progettazione, fare riferimento alla sezione "Modello Logico".

Notazioni: Tutte le tabelle condividono lo stesso nome in notazione PascalCase delle *relazioni corrispondenti* nel Modello Logico. Data una tabella Sorgente, tutti i vincoli di chiave primaria espliciti seguono il formato **Sorgente_pk**. Per ogni chiave esterna che fa riferimento a una tabella Destinazione, si avrà il corrispondente vincolo **Sorgente_fk_Destinazione**.

Creazione tabella User:

```
CREATE TABLE IF NOT EXISTS "User" (
        username SHORT_TEXT NOT NULL,
        password SHORT_TEXT NOT NULL,
        admin BOOLEAN DEFAULT FALSE,
        creation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        CONSTRAINT User_pk PRIMARY KEY (username)
);
```

Creazione tabella Page:

```
CREATE TABLE IF NOT EXISTS "Page" (
        id SERIAL NOT NULL,
        title SHORT_TEXT NOT NULL,
        creation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        author SHORT_TEXT NOT NULL,
        CONSTRAINT Page_pk PRIMARY KEY (id),
        -- Se un utente viene eliminato dalla tabella 'User'
        -- tutte le pagine correlate (con 'author' uguale
        -- al 'username') nella tabella 'Page' verranno eliminate automaticamente.
        CONSTRAINT Page_fk_User FOREIGN KEY (author) REFERENCES "User"(username)
        ON DELETE CASCADE
);
```

Creazione tabella PageText: (definisce il contenuto di una pagina)

```
CREATE TABLE IF NOT EXISTS "PageText" (
        id SERIAL NOT NULL,
        order_num INT NOT NULL,
        text TEXT NOT NULL,
        page_id INT NOT NULL,
        author SHORT_TEXT NOT NULL,
        CONSTRAINT PageText_pk PRIMARY KEY (id),
        -- Se una pagina viene eliminata dalla tabella 'Page'
        -- tutti i testi correlati a quella pagina
        -- verranno eliminate automaticamente.
        CONSTRAINT PageText_fk_Page FOREIGN KEY (page_id)
        REFERENCES "Page"(id) ON DELETE CASCADE,
        CONSTRAINT PageText_pk_User FOREIGN KEY (author)
        REFERENCES "User"(username)
);
```

Creazione tabella PageUpdate: (definisce una modifiche ad una pagina)

```
CREATE TABLE IF NOT EXISTS "PageUpdate" (
        id SERIAL NOT NULL,
        page_id INT DEFAULT NULL,
        author SHORT_TEXT DEFAULT NULL,
        status INT DEFAULT -1,
        creation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        old_text TEXT DEFAULT NULL,
        CONSTRAINT Update_pk PRIMARY KEY (id),
        -- Se una pagina o un utente che ha creato la modifica viene eliminato
        -- allora tutte le modifiche correlate
        -- verranno eliminate automaticamente.
        CONSTRAINT Update_fk_Page FOREIGN KEY (page_id)
        REFERENCES "Page"(id) ON DELETE CASCADE,
        CONSTRAINT Update_fk_User FOREIGN KEY (author)
        REFERENCES "User"(username) ON DELETE CASCADE
);
```

Creazione tabella PageUpdate: (definisce il contenuto della modifica)

```
CREATE TABLE IF NOT EXISTS "UpdatedText" (
        id SERIAL NOT NULL,
        text TEXT DEFAULT NULL,
        order_num INT DEFAULT NULL,
        update_id INT DEFAULT NULL,
        type INT NOT NULL,
        CONSTRAINT UpdatedText_pk PRIMARY KEY (id),
        -- Se una modifica viene eliminata dalla tabella 'Update'
        -- tutti i testi correlati a quella modifica
        -- verranno eliminate automaticamente.
        CONSTRAINT UpdatedText_fk_Update FOREIGN KEY (update_id)
        REFERENCES "PageUpdate"(id) ON DELETE CASCADE
);
```

Creazione tabella Notification:

```
CREATE TABLE IF NOT EXISTS "Notification" (
        id SERIAL NOT NULL,
        "user" SHORT_TEXT NOT NULL,
        status INT DEFAULT 0,
        update_id INT NOT NULL,
        "type" INT NOT NULL,
        viewed BOOLEAN DEFAULT FALSE,
        creation_date TIMESTAMP DEFAULT CURRENT_TIMESTAMP,
        CONSTRAINT Notification_pk PRIMARY KEY (id),
        -- Se viene eliminato l'utente a cui e' dedicata la notifica o
        -- Se viene eliminata la modifica su cui e' stata creata una notifica
        -- Di conseguenza anche le notifiche correlate saranno
        -- eliminate automaticamente dalla tabella 'Notification'.
        CONSTRAINT Notification_fk_User FOREIGN KEY ("user")
        REFERENCES "User"(username) ON DELETE CASCADE,
        CONSTRAINT Notification_fk_PageUpdate FOREIGN KEY (update_id)
        REFERENCES "PageUpdate"(id) ON DELETE CASCADE
);
```

4.3 View

Le seguenti viste sono state create per fornire una visione strutturata e semplificata di alcuni dei dati più significativi nel database. Ogni vista corrisponde a una specifica interrogazione o aggregazione di dati utili per analizzare e monitorare l'attività degli utenti e le modifiche alle pagine nel sistema.

Listing 1: Vista PagineDiUnUtente

```
-- Restituisce gli username degli utenti e i titoli delle pagine
-- associate a un utente specifico.
CREATE OR REPLACE VIEW PagineDiUnUtente AS
        SELECT "User".username, "Page".title
        FROM "User"
        JOIN "Page" ON "User".username = "Page".author;
```

Listing 2: Vista ModificheDiUnUtenteAdUnaPagina

```
-- Restituisce gli username degli utenti, i titoli delle pagine e
-- i nuovi testi delle modifiche effettuate da un utente su una pagina.
CREATE OR REPLACE VIEW ModificheDiUnUtenteAdUnaPagina AS
        SELECT "User".username, "Page".title, "PageText".text AS new_text
        FROM "User"
        JOIN "Page" ON "User".username = "Page".author
        JOIN "PageText" ON "Page".id = "PageText".page_id;
```

Listing 3: Vista PaginaConTesto

```
-- Restituisce il titolo, l'ID della pagina, l'ID del testo e
-- l'autore di una pagina insieme al testo associato.
CREATE OR REPLACE VIEW PaginaConTesto AS
        SELECT "Page".title, "Page".id AS page_id, "PageText".id AS text_id, "Page".au
        FROM "Page"
        LEFT JOIN "PageText" ON "Page".id = "PageText".page_id;
```

Listing 4: Vista NotificheDiUnUtente

```
-- Restituisce gli username degli utenti, lo stato,
-- l'ID dell'aggiornamento e il tipo di notifiche associate a un utente.
CREATE OR REPLACE VIEW NotificheDiUnUtente AS
        SELECT
                "User".username,
                "Notification".status,
                "Notification".update_id,
                "Notification"."type"
        FROM "Notification"
        JOIN "User" ON "Notification"."user" = "User".username;
```

Listing 5: Vista UpdatePerPagina

```sql
-- Restituisce l'ID della pagina e il numero di
-- aggiornamenti accettati per una pagina specifica.
CREATE OR REPLACE VIEW UpdatePerPagina AS
        SELECT "Page".id AS page_id, COUNT("PageUpdate".id) AS num_updates
        FROM "Page"
        LEFT JOIN "PageUpdate" ON "Page".id = "PageUpdate".page_id
        WHERE "PageUpdate".status IS NOT NULL
        GROUP BY "Page".id;
```

Listing 6: Vista UtentiConModificheProposte

```sql
-- Restituisce gli username degli utenti che hanno
-- proposto modifiche per una pagina specifica.
CREATE OR REPLACE VIEW UtentiConModificheProposte AS
        SELECT "User".username, "PageUpdate".page_id
        FROM "PageUpdate"
        JOIN "User" ON "PageUpdate".author = "User".username;
```

4.4 Operazioni

Notazioni: si utilizza snake_case minuscolo per i nomi delle funzioni/procedure e parametri

4.4.1 Funzione PL/pgSQL `search_pages`

Listing 7: Funzione SQL per la ricerca di pagine

```
CREATE OR REPLACE FUNCTION search_pages(
        search_text SHORT_TEXT,
        search_in_title BOOLEAN DEFAULT TRUE,
        search_in_text BOOLEAN DEFAULT FALSE,
        search_in_author BOOLEAN DEFAULT FALSE
) RETURNS TABLE (
        page_id INT,
        title SHORT_TEXT,
        creation_date TIMESTAMP,
        author SHORT_TEXT
)
AS $$
BEGIN
RETURN QUERY
        SELECT p.id, p."title", p.creation_date, p.author
        FROM "Page" p
        WHERE
        (search_in_title AND p."title" ILIKE '%' || search_text || '%')
        OR
        (search_in_text AND p.id IN (
        SELECT pt.page_id
        FROM "PageText" pt
        WHERE pt.text ILIKE '%' || search_text || '%'))
        OR
        (search_in_author AND p.author ILIKE '%' || search_text || '%');
END;
$$ LANGUAGE plpgsql;
```

Questa funzione PL/pgSQL, chiamata `search_pages`, accetta un testo di ricerca e tre parametri booleani come input. Restituisce una tabella contenente `page_id`, `titolo`, `creation_date` e `author` delle pagine che corrispondono ai criteri di ricerca specificati.

Note:

- La funzione esegue una ricerca case-insensitive utilizzando l'operatore ILIKE.
- Risultato: `page_id`, `titolo`, `creation_date` e `author`.

4.4.2 Esempio di utilizzo della funzione `search_pages`

```
SELECT * FROM search_pages('testo_di_ricerca', TRUE, TRUE, FALSE);
```

4.4.3 Parametri della funzione `search_pages`

- `search_text`: Il testo da cercare all'interno dei titoli, dei testi o degli autori delle pagine.
- `search_in_title` (Default: `TRUE`): Booleano che indica se cercare all'interno dei titoli delle pagine.
- `search_in_text` (Default: `FALSE`): Booleano che indica se cercare all'interno dei testi delle pagine.
- `search_in_author` (Default: `FALSE`): Booleano che indica se cercare all'interno degli autori delle pagine.

4.4.4 Funzione PL/pgSQL `search_notifications`

```
CREATE OR REPLACE FUNCTION search_notifications(
        titolo_pagina_text TEXT,
        tipo_notifica INT,
        stato_notifica INT,
        notifica_letta BOOLEAN
) RETURNS TABLE (
        id INT,
        user_name SHORT_TEXT,
        status INT,
        update_id INT,
        notification_type INT,
        viewed BOOLEAN,
        creation_date TIMESTAMP
) AS $$
BEGIN
RETURN QUERY
        SELECT
                n.id,
                n."user",
                n.status,
                n.update_id,
                n."type",
                n.viewed,
                n.creation_date
        FROM "Notification" n
        JOIN "PageUpdate" u ON n.update_id = u.id
        JOIN "Page" p ON u.page_id = p.id
        WHERE p.title ILIKE '%' || titolo_pagina_text || '%'
        AND n.type = tipo_notifica
        AND n.status = stato_notifica
        AND n.viewed = notifica_letta;
END;
$$
LANGUAGE plpgsql;
```

La funzione PL/pgSQL, denominata `search_notifications`, consente di effettuare una ricerca avanzata tra le notifiche del sistema. Restituisce una tabella contenente dettagli specifici delle notifiche che corrispondono ai criteri di ricerca specificati.

4.4.5 Esempio di utilizzo della funzione `search_notifications`

```
SELECT * FROM search_notifications('Pagina', 1, 0, FALSE);
```

Note:

- Questa query restituirà tutte le notifiche relative alle pagine che contengono il testo "Pagina", hanno tipo di notifica 1, stato 0 e non sono state ancora lette.

4.4.6 Parametri della funzione `search_notifications`

- `titolo_pagina_text`: Il testo da cercare all'interno dei titoli delle pagine associate alle notifiche.
- `tipo_notifica`: Il tipo di notifica da cercare.
- `stato_notifica`: Lo stato della notifica da cercare.
- `notifica_letta` (Default: `FALSE`): Booleano che indica se cercare notifiche lette o non lette.

4.4.7 Elementi di output della funzione `search_notifications`

La funzione restituisce una tabella con le seguenti colonne:

- `id`: L'identificativo univoco della notifica.
- `user_name`: Lo username dell'utente associato alla notifica.
- `status`: Lo stato della notifica.
- `update_id`: L'identificativo univoco dell'aggiornamento associato alla notifica.
- `notification_type`: Il tipo della notifica.
- `viewed`: Booleano che indica se la notifica è stata letta o meno.
- `creation_date`: La data di creazione della notifica.

4.5 Trigger

In questa sezione sono definiti Trigger necessari per l'implementazione di vincoli o altre caratteristiche della base di dati.
I Trigger di questa soluzione sono impiegati principalmente per gestire le notifiche degli utenti della Wiki.

Notazione: Ogni trigger viene definito dalla notazione **Trigger_NomeTabellaListen_Caratteristica_NomeTabellaEdit** dove:

- **NomeTabellaListen**: Rappresenta la tabella sulla quale il Trigger "guarda" le modifiche.
- **NomeTabellaEdit**: Rappresenta la tabella modificata dal Trigger.
- **Caratteristica**: Rappresenta ciò che scaturisce il Trigger.

Nota: Ogni trigger è composto da due parti, il trigger vero e proprio e la funzione che esso esegue.

4.5.1 Trigger_Update_Request_Notifica

```
CREATE OR REPLACE FUNCTION notification_request_update()
  RETURNS TRIGGER AS $$
  DECLARE
         page_creator VARCHAR(255);
        TYPE_REQUEST_UPDATE INT := 0;

  BEGIN
        SELECT author INTO page_creator
        FROM "Page"
        WHERE id = NEW.page_id;

        INSERT INTO "Notification" ("user", update_id, type)
        VALUES (page_creator, NEW.id, TYPE_REQUEST_UPDATE);

        RETURN NEW;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER Trigger_Update_Request_Notifica
        AFTER INSERT ON "PageUpdate"
        FOR EACH ROW
        EXECUTE FUNCTION notification_request_update();
```

4.5.2 Trigger_Update_Accettazione_Notifica

```
CREATE OR REPLACE FUNCTION notification_update_accepted()
  RETURNS TRIGGER AS $$
  DECLARE
        TYPE_UPDATE_ACCEPTED INT := 1;
        NOTIFICATION_READ INT := 1;
        STATUS_PENDING INT := -1;
        STATUS_ACCEPTED INT := 1;
        page_creator VARCHAR(255);

  BEGIN
        SELECT author INTO page_creator
        FROM "Page"
        WHERE id = NEW.page_id;

        IF OLD.status = STATUS_PENDING AND NEW.status = STATUS_ACCEPTED THEN
          INSERT INTO "Notification" ("user", update_id, type)
          VALUES (NEW.author, NEW.id, TYPE_UPDATE_ACCEPTED);

          UPDATE "Notification"
          SET "type" = TYPE_UPDATE_ACCEPTED, status = NOTIFICATION_READ
          WHERE "user" = page_creator AND update_id = NEW.id;
        END IF;

    RETURN NEW;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER Trigger_Update_Accettazione_Notifica
        AFTER UPDATE ON "PageUpdate"
        FOR EACH ROW
        EXECUTE FUNCTION notification_update_accepted();
```

4.5.3 Trigger_Update_Rifiuto_Notifica

```
CREATE OR REPLACE FUNCTION notification_update_rejected()
  RETURNS TRIGGER AS $$
  DECLARE
        TYPE_UPDATE_REJECTED INT := 2;
        NOTIFICATION_READ INT := 1;
        STATUS_PENDING INT := -1;
        STATUS_REJECTED INT := 0;
        page_creator VARCHAR(255);

  BEGIN
        SELECT author INTO page_creator
        FROM "Page"
        WHERE id = NEW.page_id;

        IF OLD.status = STATUS_PENDING AND NEW.status = STATUS_REJECTED THEN
          INSERT INTO "Notification" ("user", update_id, type)
          VALUES (NEW.author, NEW.id, TYPE_UPDATE_REJECTED);

          UPDATE "Notification"
          SET "type" = TYPE_UPDATE_REJECTED, status = NOTIFICATION_READ
          WHERE "user" = page_creator AND update_id = NEW.id;
        END IF;

  RETURN NEW;
  END;
$$ LANGUAGE plpgsql;

CREATE TRIGGER Trigger_Update_Rifiuto_Notifica
        AFTER UPDATE ON "PageUpdate"
        FOR EACH ROW
        EXECUTE FUNCTION notification_update_rejected();
```