UNIVERSITÀ DEGLI STUDI
DI NAPOLI FEDERICO II

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

Giuseppe Pollio, Mario Lombardo

Anno accademico 2023-2024

Indice

1 Modello Concettuale

1.1 Analisi dei requisiti

In questa sezione si analizza la specifica con lo scopo di definire le funzionalità che la base di dati deve soddisfare. Individueremo le entità e le associazioni del mini-word, e produrremo una 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

L'introduzione individua il mini-world da rappresentare e una prima entità: **Page** avente come attributi **title** e **creation**. Inoltre nel testo della **Page** possiamo individuare un'altra entità associata alla **Page**: **PageText** contente come attributo oltre che la frase individuata come **text** anche **link** (collegamento) e un collegamento all'entità **Page**.

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.

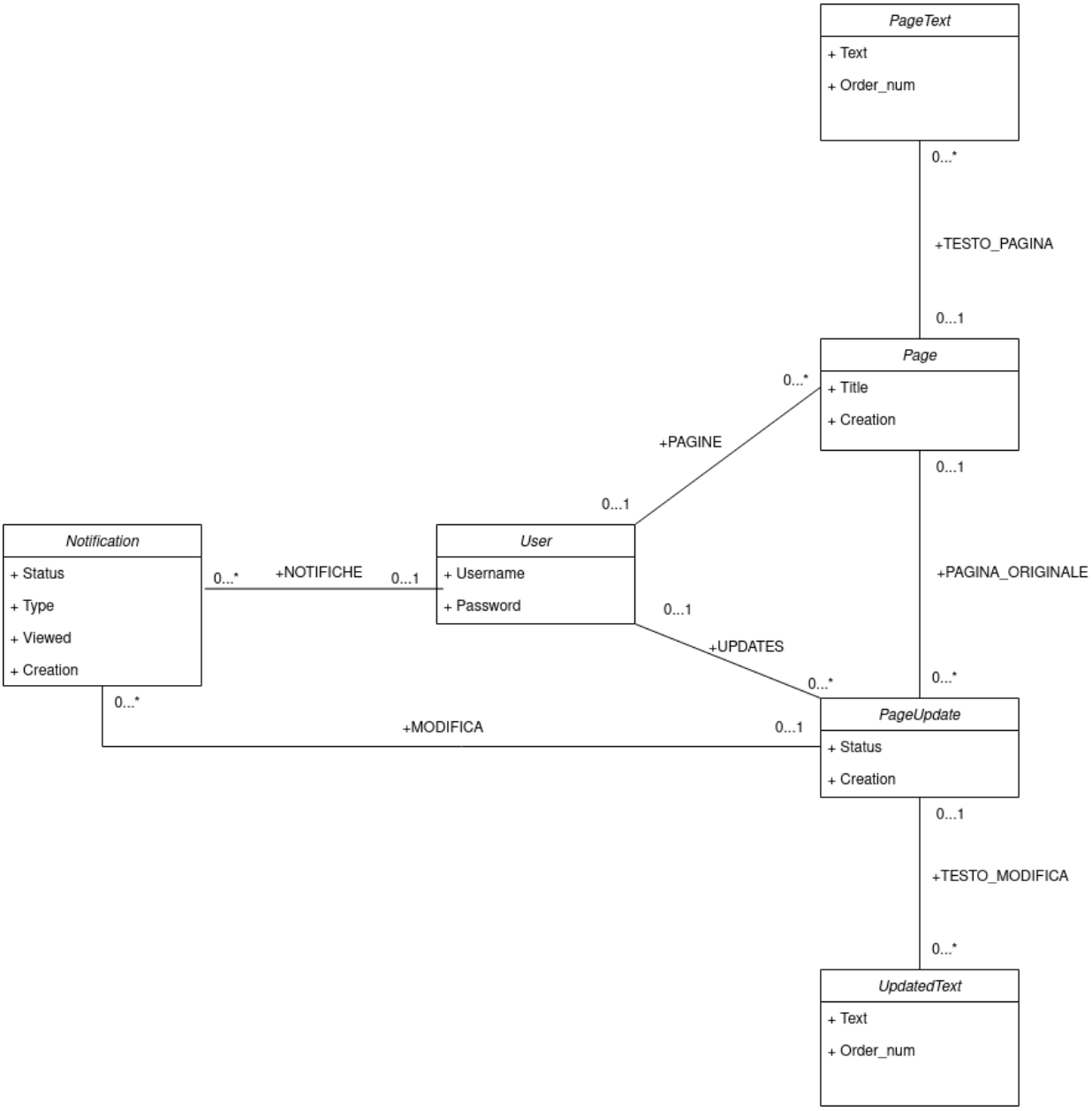
Dato che le pagine delle wiki saranno publiche oltre che modificabili, sarà necessario tenere traccia degli utenti tramita l'entità **User**, alla quale sarà possibile accederci tramite una **password** e un **username**. Di conseguenza alla **Page** viene individuata una nuova caratteristica **author**. Le modifiche del testo di una **Page** saranno salvate tramite le l'entità **PageUpdate** avente attributi **status** e un collegamento ad **User** che ne rappresenta l'autore (**author**). Il testo modificato viene individuato tramite l'entità: **UpdatedText** contenente il nuovo testo individutato dall'attributo **text**, un eventuale **link** (collegamento) e un collegamento all'entità **PageUpdate**, quando una modifica viene inoltrata all'autore esso invece deve essere avvisato tramite una notifica individuata dall'entità **Notification** avente come attributi **status** (stato di lettura), **type**, un collegamento a **User** per individuare il proprietario della notifica e un collegamento a **PageUpdate**.

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).

Viene in aggiunta inserito il nuovo attributo **order_num** alle entità **PageText** e **UpdatedText** che rappresenta l'ordinamento all'interno del testo della **Page**.

1.2 Diagramma (UML)

In seguito alle considerazioni espresse nella sezione precedente, 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. **Creation**(Data): Data e ora di creazione della pagina.
PageText	Frase di una Pagina (**Page**).	**Text**(Stringa): Contenuto della frase. **Link**(Stringa): Collegamento della frase (interno o esterno alla wiki). **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 frase. **Link**(Stringa): Collegamento della frase (interno o esterno alla wiki). **Order_num**(Intero): Ordinamento della frase all'interno del testo complessivo.
Notification	Notifiche di un **Utente (User)**.	**Status**(Booleano): Stato di lettura di una notifica. **Type**(Intero): Tipologia di notifica.

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 testo di una pagina, il titolo di una pagina e il testo di una modifica devono avere lunghezza non nulla.
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 Privacy	interrelazionale	Un utente deve avere un modo obbligatorio una password.
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

1.4 Ristrutturazione del Modello Concettuale

Prima di poter passare allo schema logico è necessario ristrutturare il diagramma delle classi per semplificare la traduzione in schema logico, ottimizzare il progetto, eliminare le generalizzazioni, eliminare gli attributi multivalore, eliminare gli attributi strutturati, accorpare o partizionare le entita figlie e scegliere gli identificatori delle nostre entità ove necessario

1.4.1 Analisi delle Ridondanze

Una ridondanza è un dato che è già presente nella base di dati o può essere derivato da altri dati. Nel modello concettuale originale non sono presenti ridondanze

1.4.2 Analisi delle generalizzazioni

La specializzazione è il processo di determinazione di sottoclassi per una data entità. La generalizzazione è il suo concetto complementare. Nel modello attuale è stato scelto di non ristrutturare le generalizzazioni presenti.

1.4.3 Eliminazione degli attributi Multivalore o Composti

Un attributo è multivalore se può essere associato ad un numero variabile di valori dello stesso dominio. Un attributo è composto se può essere suddiviso in sottoparti ognuna dotata di dominio. Nel modello concettuale non sono presenti attributi multivalore o attributi composti.

1.4.4 Analisi di Entità e Associazioni

Non si è ritenuto opportuno scomporre o accorpare entità. Tuttavia è stato deciso di rimuovere l'attributo **link** dalle entità **PageText** e **UpdatedText** in quanto i collegamenti verrano salvati dal programma seguendo la formattazione **href='link'**, facendo in questo modo il programma capirà in automatico quando un testo rappresenta un link e lo andrà a trattare come tale. Inoltre alla entità **PageUpdate** viene aggiunto l'attributo **old_text** che rappresenta il testo di una pagina prima che venga modificato inoltre viene aggiunto l'attributo **author** all'entità **PageText** che rappresenta chi ha scritto quel testo, queste modifiche vengono fatte in modo da poter tenere traccia della "storia" di una pagina.

Nel modello attuale si è evitato di creare relazioni di composizione.

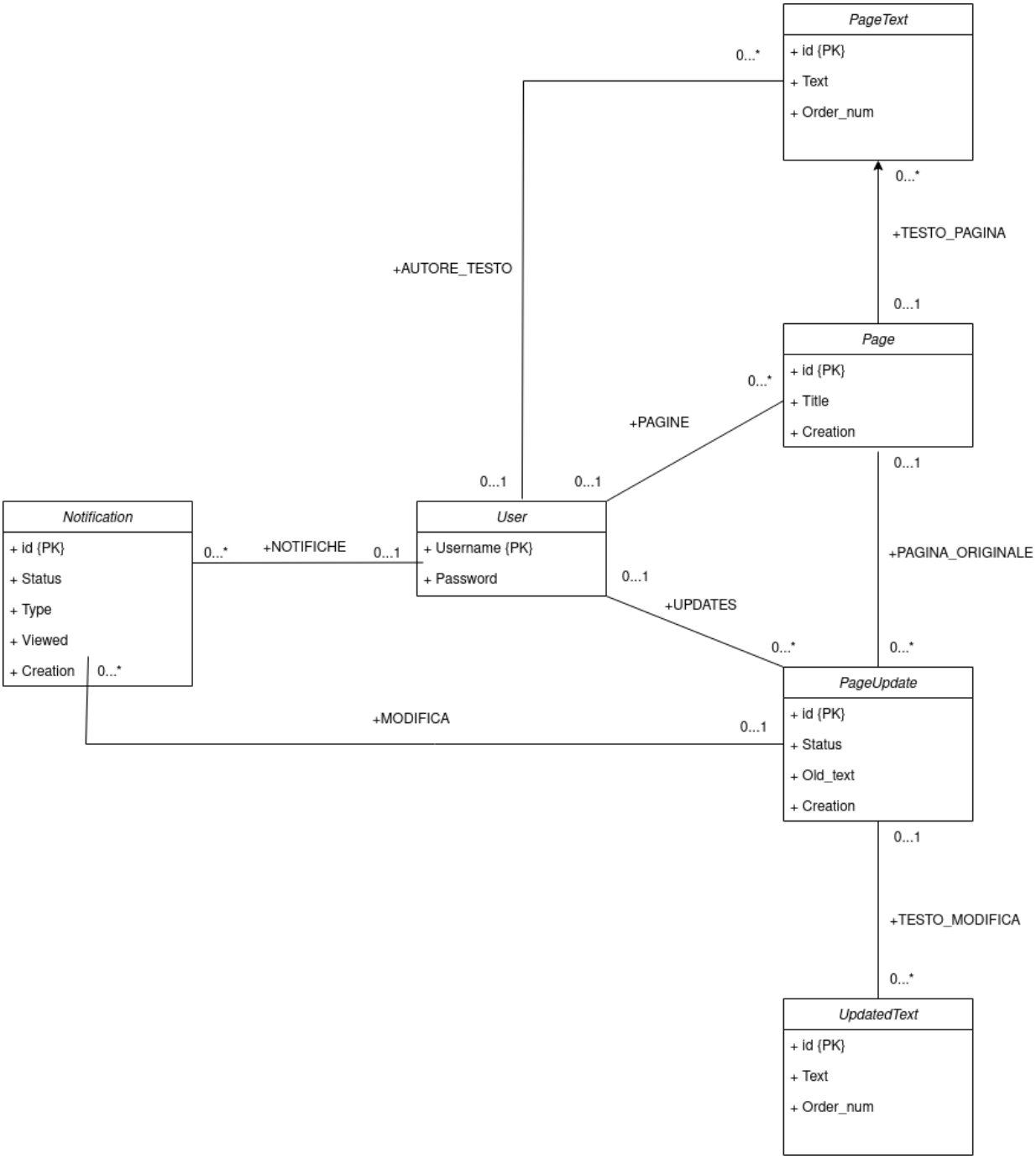
1.4.5 Scelta degli Identificatori Primari

Un identificatore o chiave minimale è un insieme di attributi che identificano univocamente un'entità. È possibile che un'entità sia dotata di molteplici identificatori, fra i quali ne sceglieremo uno principale che agirà da chiave primaria.

- **User**: Viene scelto **username** come identificatore primario **id**.
- **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**.

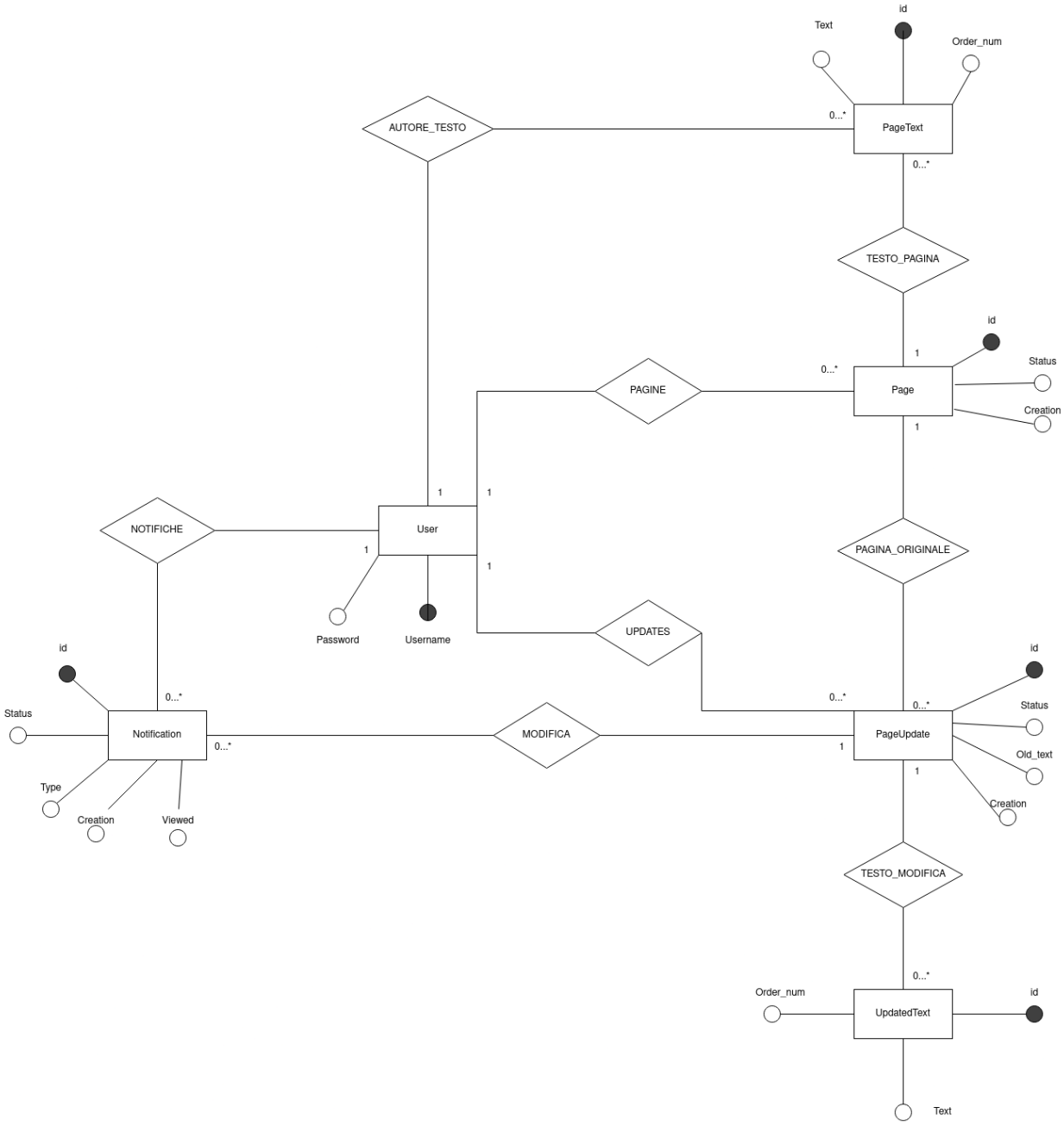
1.4.6 Diagramma UML Ristrutturato

Dopo aver ristrutturato il Class Diagram come descritto precedentemente, possiamo produrre il seguente schema concettuale ristrutturato espresso mediante diagramma UML:



1.4.7 Diagramma ER Ristrutturato

Ulteriore notazione per poter esprimere lo schema concettuale è l'ER. Il seguente è il diagramma ER:



2 Modello Logico

2.1 Traduzione di Entità e Associazioni

Avendo completato il processo di ristrutturazione, possiamo procedere col *mapping* di entità e associazioni. Per ogni entità del modello ristrutturato, definiremo una relazione equivalente con gli stessi attributi. 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 espriamiamo tramite chiave esterna, solo l'entità **Page** avrà come attributo **author**.
- **User, PageUpdate**: sono associazioni *uno-a-molti* che espriamiamo tramite chiave esterna, solo l'entità **PageUpdate** avrà come attributo **author**.
- **PageText, Page**: sono associazioni *uno-a-molti* che espriamiamo 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 espriamiamo tramite chiave esterna, solo l'entità **PageText** avrà come attributo **author**.
- **User, Notification**: sono associazioni *uno-a-molti* che espriamiamo tramite chiave esterna, solo l'entità **UpdatedText** avrà come attributo **user**.
- **PageUpdate, Notification**: sono associazioni *uno-a-molti* che espriamiamo tramite chiave esterna, solo l'entità **Notification** avrà come attributo **update_id**.

2.2 Relazioni

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

- **USER** (*username*, password)
- **PAGE** (*id*, title, creation, author↑) - author → USER.username
- **PAGETEXT** (*id*, order_num, text, page_id↑, author↑)
  - page_id → PAGE.id
  - author → USER.username
- **UPDATE** (*id*, status, creation, page_id↑, author↑)
  - page_id → PAGE.id
  - author → USER.username
- **NOTIFICATION** (*id*, status, type, user↑, update_id↑)
  - update_id → UPDATE.id
  - user → USER.username
- **UPDATEDTEXT** (*id*, text, order_num, type, update_id↑)
  - update_id → UPDATE.id

2.3 User View

Mentre le relazioni materiali sopra individuate definiscono la natura dei dati che andiamo a tracciare, in questa sezione definiamo un numero di viste che sono necessarie agli utenti per un comodo utilizzo del database. Esprimiamo queste viste come variabili e funzioni dell'algebra relazionale.

- **Pagine di un Utente:**

$$\text{PagineDiUnUtente} \leftarrow \pi_{\text{username, title}}(\sigma_{\text{author=username}}(\text{User} \bowtie_{\text{id=page\_id}} \text{Page}))$$

- **Modifiche di un Utente ad una Specifica Pagina con Testo Nuovo:**

$$\text{ModificheDiUnUtenteAdUnaPagina} \leftarrow \pi_{\text{username, title, text AS new\_text}} ((\text{Page} \bowtie_{\text{author=username}} \text{User}) \bowtie_{\text{id=page\_id}} \text{PageText})$$

- **Pagina con Testo:**

$$\text{PaginaConTesto} \leftarrow \pi_{\text{title, page\_id, text\_id, author}}(\text{Page} \bowtie_{\text{id=page\_id}} \text{PageText})$$

- **Notifiche di un Utente:**

$$\text{NotificheDiUnUtente} \leftarrow \pi_{\text{username, status, update\_id, type}}(\text{Notifications} \bowtie_{\text{user=username}} \text{User})$$

- **Quanti PageUpdate ha una Pagina in Ordine dal Più Recente:**

$$\text{UpdatePerPagina} \leftarrow \pi_{\text{page\_id, COUNT(id) AS num\_updates}}(\sigma_{\text{status IS NOT NULL}}(\text{PageUpdate}) \bowtie_{\text{id=page\_id}} \text{Page})$$

- **Utenti che Hanno Proposto una Modifica ad una Specifica Pagina:**

$$\text{UtentiConModificheProposte} \leftarrow \pi_{\text{username, page\_id}}(\text{PageUpdate} \bowtie_{\text{author=username}} \text{User})$$

- **Pagine con Più Modifiche Accettate:**

$$\begin{aligned}\text{PagineConPiuModificheAccettate} \leftarrow\ & \pi_{\text{title, num\_accepted}} (\\ & \pi_{\text{title, MAX(num\_accepted) AS num\_accepted}} (\\ & \pi_{\text{title, COUNT(id) AS num\_accepted}} (\\ & \text{Page} \bowtie_{\text{id=page\_id}} \text{PageUpdate} \bowtie_{\text{status=1}} \text{Notifications} \\ &) \\ &) \\ &)\end{aligned}$$

- **Pagine con Più Modifiche Proposte:**

$$\text{PagineConPiuModificheProposte} \leftarrow \pi_{\text{title, num\_proposed}} (\pi_{\text{title, MAX(num\_proposed) AS num\_proposed}} (\pi_{\text{title, COUNT(id) AS num\_proposed}} (\text{PageUpdate} \bowtie_{\text{status=0}} \text{Notifications})))$$

3 Modello Fisico

3.1 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. Applichiamo questo tipo per la maggior parte dei titoli e nomi nello schema, garantendo così il rispetto del *Vincolo di Leggibilità Testuale*.

3.2 Tabelle

Di seguito sono indicate le istruzioni DDL necessarie per definire le tabelle del database relazionale. Poiché queste sono una traduzione diretta delle relazioni definite nella sezione sul modello logico, si è preferito mantenere i commenti al minimo. Questi ultimi sono inclusi solo per chiarire concetti di SQL. Per informazioni sulla progettazione, fare riferimento alla sezione 3, Modello Logico.
Notazioni: Tutte le tabelle condividono lo stesso nome delle relazioni corrispondenti nel modello logico e utilizzano PascalCase. 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**.

```
CREATE TABLE IF NOT EXISTS User (
username SHORT_TEXT NOT NULL,
password SHORT_TEXT NOT NULL,

CONSTRAINT User_pk PRIMARY KEY (username)
);
```

```
CREATE TABLE IF NOT EXISTS Page (
id INT AUTO_INCREMENT,
title SHORT_TEXT NOT NULL,
creation DATE DEFAULT CURRENT_DATE NOT NULL,
author SHORT_TEXT NOT NULL,
CONSTRAINT Page_pk PRIMARY KEY (id),
CONSTRAINT Page_fk_User FOREIGN KEY (author) REFERENCES User(username)
ON DELETE CASCADE
);
```

```
CREATE TABLE IF NOT EXISTS PageText (
  id INT AUTO_INCREMENT,
  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),
  CONSTRAINT PageText_fk_Page FOREIGN KEY (page_id)
  REFERENCES Page(id) ON DELETE CASCADE,
  CONSTRAINT PageText_pk_User FOREIGN KEY (author)
  REFERENCES User(username)
);
```

```
CREATE TABLE IF NOT EXISTS `PageUpdate` (
id INT AUTO_INCREMENT,
```

```
    page_id INT DEFAULT NULL,
    author SHORT_TEXT DEFAULT NULL,
    status INT DEFAULT NULL,
    creation DATETIME DEFAULT CURRENT_TIMESTAMP() NOT NULL,
    old_text LONGTEXT DEFAULT NULL,
    CONSTRAINT Update_pk PRIMARY KEY (id),
    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
    );
```

```
    CREATE TABLE IF NOT EXISTS `UpdatedText` (
    id INT NOT NULL,
    text LONGTEXT DEFAULT NULL,
    order_num INT DEFAULT NULL,
    update_id INT DEFAULT NULL,
    type INT NOT NULL,
    CONSTRAINT UpdatedText_pk PRIMARY KEY (id),
    CONSTRAINT UpdatedText_fk_Update FOREIGN KEY (update_id)
    REFERENCES `PageUpdate`(id) ON DELETE CASCADE
    );
```

```
    CREATE TABLE IF NOT EXISTS Notification (
    id INT AUTO_INCREMENT,
    user SHORT_TEXT NOT NULL,
    status BOOLEAN,
    update_id INT NOT NULL,
    type INT NOT NULL,
    viewed BOOLEAN DEFAULT FALSE,
    creation DATETIME DEFAULT CURRENT_TIMESTAMP() NOT NULL,
    CONSTRAINT Notification_pk PRIMARY KEY (id),
    CONSTRAINT Notification_fk_User FOREIGN KEY (user)
    REFERENCES User(username)
    ON DELETE CASCADE,
    CONSTRAINT Notification_fk_User FOREIGN KEY (update_id)
    REFERENCES `Update`(id)
    ON DELETE CASCADE
    );
```

3.3 View

In questa sezione sono definite le istruzioni SQL corrispondenti alle Viste Utente dell'algebra relazionale delineate nella sezione "2.3 - User View". Le Viste Utente rappresentate come variabili sono tradotte in SQL mediante il comando **CREATE VIEW**.

Le Viste Utente rappresentate come funzioni dell'algebra relazionale sono tradotte come funzioni del plpgsql.

Notazioni: ogni vista condivide il nome con la corrispondente Vista Utente logica, in PascalCase. I parametri utilizzano camelCase.

C'è un commento quando la traduzione dall'algebra relazionale a SQL non è completamente diretta.

```
1     CREATE OR REPLACE FUNCTION PagineDiUnUtente(
2     utente_username user.username%TYPE
3     )
4     RETURNS TABLE(username user.username%TYPE, title Page.title%TYPE) AS
5     $$
6     BEGIN
7     RETURN QUERY (
8     SELECT User.username, Page.title
9     FROM User
10    JOIN Page ON User.username = Page.author
11    WHERE User.username = utente_username
12    );
13    END;
14    $$
15    LANGUAGE plpgsql;
```

```
1     CREATE OR REPLACE FUNCTION ModificheDiUnUtenteAdUnaPagina(
2     utente_username User.username%TYPE
3     )
4     RETURNS TABLE(
5     username User.username%TYPE,
6     title Page.title%TYPE,
7     new_text PageText.text%TYPE
8     ) AS
9     $$
10    BEGIN
11    RETURN QUERY (
12    SELECT User.username, Page.title, PageText.text AS new_text
13    FROM User
14    JOIN Page ON User.username = Page.author
15    JOIN PageText ON Page.id = PageText.page_id
16    WHERE User.username = utente_username
17    );
18    END;
19    $$
20    LANGUAGE plpgsql;
```

```
1     CREATE OR REPLACE FUNCTION PaginaConTesto(
2     pagina_id Page.id%TYPE
3     )
4     RETURNS TABLE(
5     title Page.title%TYPE,
```

```
    page_id Page.id%TYPE,
    text_id PageText.id%TYPE,
    author Page.author%TYPE
    ) AS
    $$
    BEGIN
    RETURN QUERY (
    SELECT Page.title, Page.id AS page_id, PageText.id AS text_id, Page.
       author
    FROM Page
    LEFT JOIN PageText ON Page.id = PageText.page_id
    WHERE Page.id = pagina_id
    );
    END;
    $$
    LANGUAGE plpgsql;
```

```
    CREATE OR REPLACE FUNCTION NotificheDiUnUtente(
    utente_username User.username%TYPE
    )
    RETURNS TABLE(
    username User.username%TYPE,
    status Notifications.status%TYPE,
    update_id Notifications.update_id%TYPE,
    type Notifications.type%TYPE
    ) AS
    $$
    BEGIN
    RETURN QUERY (
    SELECT User.username, Notifications.status, Notifications.update_id,
       Notifications.type
    FROM Notifications
    JOIN User ON Notifications.user = User.username
    WHERE User.username = utente_username
    );
    END;
    $$
    LANGUAGE plpgsql;
```

```
    CREATE OR REPLACE FUNCTION UpdatePerPagina(
    pagina_id Page.id%TYPE
    )
    RETURNS TABLE(
    page_id Page.id%TYPE,
    num_updates BIGINT
    ) AS
    $$
    BEGIN
    RETURN QUERY (
    SELECT Page.id AS page_id, COUNT(PageUpdate.id) AS num_updates
    FROM Page
    LEFT JOIN PageUpdate ON Page.id = PageUpdate.page_id
    WHERE Page.id = pagina_id AND PageUpdate.status IS NOT NULL
    GROUP BY Page.id
```

```
    );
    END;
    $$
    LANGUAGE plpgsql;

    CREATE OR REPLACE FUNCTION UtentiConModificheProposte(
    pagina_id Page.id%TYPE
    )
    RETURNS TABLE(
    username User.username%TYPE,
    page_id Page.id%TYPE
    ) AS
    $$
    BEGIN
    RETURN QUERY (
    SELECT User.username, PageUpdate.page_id
    FROM PageUpdate
    JOIN User ON PageUpdate.author = User.username
    WHERE PageUpdate.page_id = pagina_id
    );
    END;
    $$
    LANGUAGE plpgsql;

    CREATE OR REPLACE FUNCTION PagineConPiuModificheAccettate()
    RETURNS TABLE(
    title Page.title%TYPE,
    num_accepted BIGINT
    ) AS
    $$
    BEGIN
    RETURN QUERY (
    SELECT Page.title, MAX(num_accepted) AS num_accepted
    FROM (
    SELECT Page.id, Page.title, COUNT(PageUpdate.id) AS num_accepted
    FROM Page
    LEFT JOIN PageUpdate ON Page.id = PageUpdate.page_id
    LEFT JOIN Notifications ON PageUpdate.id = Notifications.update_id
    WHERE Notifications.status = 1
    GROUP BY Page.id, Page.title
    ) AS Subquery
    GROUP BY Subquery.id, Subquery.title
    );
    END;
    $$
    LANGUAGE plpgsql;

    CREATE OR REPLACE FUNCTION PagineConPiuModificheProposte()
    RETURNS TABLE(
    title Page.title%TYPE,
    num_proposed BIGINT
    ) AS
    $$
    BEGIN
```

```
    RETURN QUERY (
    SELECT Page.title, MAX(num_proposed) AS num_proposed
    FROM (
    SELECT Page.id, Page.title, COUNT(PageUpdate.id) AS num_proposed
    FROM Page
    LEFT JOIN PageUpdate ON Page.id = PageUpdate.page_id
    LEFT JOIN Notifications ON PageUpdate.id = Notifications.update_id
    WHERE Notifications.status = 0
    GROUP BY Page.id, Page.title
    ) AS Subquery
    GROUP BY Subquery.id, Subquery.title
    );
    END;
    $$
    LANGUAGE plpgsql;
```

3.4 Operazioni

In questa sezione sono presenti operazioni di tipo programmatico.

Notazioni: si utilizza PascalCase per i nomi delle funzioni/procedure e camelCase per i parametri

```
CREATE OR REPLACE FUNCTION search_pages(
search_text VARCHAR(255),
search_in_title BOOLEAN DEFAULT TRUE,
search_in_text BOOLEAN DEFAULT FALSE,
search_in_author BOOLEAN DEFAULT FALSE
) RETURNS TABLE (
page_id INT,
title VARCHAR(255),
creation DATE,
author VARCHAR(255)
) AS $$
BEGIN
RETURN QUERY
SELECT id, title, creation, author
FROM Page
WHERE
(search_in_title AND title ILIKE '%' || search_text || '%') OR
(search_in_text AND id IN (SELECT page_id FROM PageText WHERE text
   ILIKE '%' || search_text || '%')) OR
(search_in_author AND author ILIKE '%' || search_text || '%');
END;
$$ 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, data di creazione e autore delle pagine che corrispondono ai criteri di ricerca specificati.

Parametri:

- `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.

Utilizzo:

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

Note:

- La funzione esegue una ricerca case-insensitive utilizzando l'operatore ILIKE.
- Il risultato è una tabella con colonne: page_id, titolo, data di creazione e autore.

3.5 Trigger

In questa sezione sono definiti trigger necessari per l'implementazione di vincoli o altre caratteristiche della base di dati oppure necessari per l'automatizzazione di certe cose.

I trigger nel modello vengono utilizzati principalmente per gestire le notifiche degli updates di una pagina.

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 una caratteristica determinata da ciò che compie il trigger.

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

```
1  CREATE OR REPLACE FUNCTION notification_request_update()
2  RETURNS TRIGGER AS $$
3  DECLARE
4  page_creator VARCHAR(255);
5  TYPE_REQUEST_UPDATE INT := 0;
6  BEGIN
7  SELECT author INTO page_creator
8  FROM Page
9  WHERE id = NEW.page_id;
10
11 INSERT INTO notification ("user", update_id, type)
12 VALUES (page_creator, NEW.id, TYPE_REQUEST_UPDATE);
13
14 RETURN NEW;
15 END;
16 $$ LANGUAGE plpgsql;
17
18 CREATE TRIGGER Trigger_Update_Request_Notifica
19 AFTER INSERT ON PageUpdate
20 FOR EACH ROW
21 EXECUTE FUNCTION notification_request_update();
```

```
1  CREATE OR REPLACE FUNCTION notification_update_accepted()
2  RETURNS TRIGGER AS $$
3  DECLARE
4  page_creator VARCHAR(255);
5  TYPE_UPDATE_ACCEPTED INT := 1;
6  NOTIFICATION_READ INT := 1;
7  STATUS_PENDING INT := -1;
8  STATUS_ACCEPTED INT := 1;
9  BEGIN
10 SELECT author INTO page_creator
11 FROM Page
12 WHERE id = NEW.page_id;
13
14 IF OLD.status = STATUS_PENDING AND NEW.status = STATUS_ACCEPTED THEN
15 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();
```

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