A7: Restrições de Integridade, Triggers e Índices

1. Restrições

Restrições servem para garantir o comportamento normal da base de dados. Caso haja alguma anormalidade na informação recebida, as restrições cancelam a operação, garantindo a integridade da base de dados. No projeto *tidder* apenas necessitamos de *triggers* e das restrições base de SQL para garantir todas as regras de negócio, como *UNIQUE*, *NOT NULL*, ...

1.1. Foram alteradas as restrições do tipo *foreign keys*, que referenciam a tabela de *users*, para não permitir que a operação de *delete* não propaguem para as questões e as respostas (ON DELETE SET NULL).

1.2. Por questões de segurança, consideramos que o sistema deve banir temporariamente qualquer utilizador que atinja os 5 *warnings*. Para cumprir essa regra o seguinte *trigger* e *function* foram criados:

```
CREATE FUNCTION trigger_auto_ban_on_warning_limit()
  RETURNS "trigger" AS $func$
BEGIN
   IF (SELECT COUNT(*) FROM warnings WHERE warnings.user_id = NEW.user_id) =
5 THEN
      INSERT INTO bans (user_id, creator_id, notes) VALUES(NEW.user_id,
NULL, 'Automatic ban for warning limit 5 reached');
   END IF;
   RETURN NULL;
END;
$func$  LANGUAGE plpgsql;

CREATE TRIGGER auto_ban_on_warning_limit AFTER INSERT ON warnings
   FOR EACH ROW EXECUTE PROCEDURE trigger_auto_ban_on_warning_limit();
```

1.3. Definimos também que quando, em uma pergunta é criada uma resposta, um voto ou uma das suas respostas recebem votos, o seu campo *updated_at* (tabela *questions*) é atualizado para a data atual. Para cumprir essa regra os seguintes *triggers* e *functions* foram criados:

```
CREATE FUNCTION trigger_update_question_timestamp()
  RETURNS "trigger" AS $func$
BEGIN
    UPDATE questions SET updated_at = now() WHERE id = NEW.question_id;
END;
$func$  LANGUAGE plpgsql;

CREATE FUNCTION trigger_update_question_timestamp_from_vote()
  RETURNS "trigger" AS $func$
DECLARE vquestion_id INT;
BEGIN
```

```
    IF NEW.votable_type = 'q' THEN
        vquestion_id := NEW.votable_id;
    ELSE
        SELECT question_id INTO vquestion_id FROM answers WHERE id =
NEW.votable_id;
    END IF;

    UPDATE questions SET updated_at = now() WHERE id = NEW.votable_id;
    RETURN NULL;
END;
$func$  LANGUAGE plpgsql;

CREATE TRIGGER answer_update_question_timestamp AFTER INSERT ON answers
   FOR EACH ROW EXECUTE PROCEDURE trigger_update_question_timestamp();

CREATE TRIGGER votes_update_question_timestamp AFTER INSERT OR UPDATE ON
votes
   FOR EACH ROW EXECUTE PROCEDURE
trigger_update_question_timestamp_from_vote();
```

1.4. Com o objetivo de ordenar as repostas pelo seu *rating* de votos, o seguinte *trigger* foi criado para manter a coluna (também criada agora) atualizada. Para cumprir essa regra os seguintes triggers e functions foram criados (de realçar que o *trigger* utiliza uma user-defined function 4.1):

```
ALTER TABLE answers ADD COLUMN vote_rating INTEGER;
UPDATE answers SET vote_rating = votable_rating(answers.id, 'a');

CREATE FUNCTION trigger_update_answer_rating()
  RETURNS "trigger" AS $func$
BEGIN
    IF OLD.votable_type = 'a' THEN
        UPDATE answers SET vote_rating = votable_rating(OLD.votable_id, 'a')
WHERE id=OLD.votable_id;
    END IF;

    RETURN NULL;
END;
$func$  LANGUAGE plpgsql;

CREATE TRIGGER votes_update_answer_rating AFTER INSERT OR DELETE OR UPDATE
ON votes FOR EACH ROW EXECUTE PROCEDURE trigger_update_answer_rating();
```

2. Workload

Estimamos que o site tenha uma capacidade máxima de 1 milhão de utilizadores registados, uma vez que o site *Stackoverflow*, que é o principal competitor e o líder no mercado, atingiu os 850.000 utilizadores. A partir desta estimativa deduzimos que haverá 20 milhões de perguntas, 100 milhões de respostas, 10 milhões de *follows*, 2 milhões de *password_resets*, 50.000 *bans*, 150.000 *warnings*, 5.000 *tags*, 30 milhões de *question_tags* e 100 milhões de votos.

Considerando este caso extremo, é da maior importância garantir a eficiência das interrogações mais

frequentes e mais custosas. Abaixo segue-se uma breve explicação de todas as principais interrogações.

2.1. Sempre que um utilizador pesquisa por perguntas na barra de pesquisa, é necessário encontrar todas as perguntas cujo título, corpo, ou corpo das respostas contenham a *string* de pesquisa do *user*. Sendo esta a funcionalidade mais importante da aplicação, optou-se por usar o índice de *full text search* para obter os resultados pretendidos (3.1).

2.2. Sempre que é necessário obter dados de uma ou várias perguntas (versão curta ou detalhada), é necessário obter as suas *tags*, *rating* e *username* (também aplicavél a respostas).

2.2.1. Para as informações detalhadas de uma pergunta, também é necessário obter outras dados dispersos pelas tabelas answers e votes, por sua vez estas answers também tem os dados indicados em 2.2. para as perguntas (excepto as tags), por isso decidiu-se agrupar ambas estas *querys* no mesmo sub-tópico .

Para colmatar a complexidade destas *query* foi necessário a criação de índices (3.2.2.) e a criação de *user-defined functions* (4.1., 4.5., 4.6.).

2.3. Sempre que um voto de utilizador trocar de valor ou for removido, de uma pergunta ou resposta, é necessário atualizar ou remover esse voto da tabela *votes*. Com o objetivo de realizar essas acções as seguintes *query's* são usadas (para questões é usado votable_type='q' e para answers é usado 'a'):

```
    UPDATE votes SET VALUE = new_value WHERE vote.votable_id = pvotable_id
AND vote.votable_type = 'q';
```

```
    DELETE votes WHERE vote.votable_id = pvotable_id AND vote.votable_type =
'q';
```

Para otimizar estas *querys* são usados também os índices presentes em 3.2.2.

2.4. Sempre que uma página de perfil é acedida, é necessário obter as suas informações diretas(email, username), estatísticas (por exemplo *rating*) e as suas perguntas. Para otimizar e simplificar as *querys* necessárias, foram criadas *user-defined functions* (4.2., 4.3., 4.4., 4.7., 4.8.).

3. Índices

A escolha de indíces é uma das mais importantes no que toca à otimização de uma base de dados, especialmente quando se tenta resolver problemas de escalabilidade. A listagem dos índices (não inlui *foreign keys* e *primary keys*) encontra-se em seguida:

3.1. *Full-Text-Search*

Para a pesquisa mencionada em 2.1., foi decidido implementar um *index GIN*, pois as colunas não vão ser atualizadas frequentemente e a pesquisa é mais rápida que *GiST*. Para implementar este índice foi decidido realizar um conjunto de acções: criar uma coluna *full_text_index_col*(3.1.1.) na tabela *questions* para armazenar o conteúdo do *index* das duas colunas (*title* e *body*), um trigger(3.1.2.) para atualizar este último sempre que uma das colunas que o constituem também for atualizada, criação de dois *index GIN*(3.1.3.), um na coluna criada (full_text_index_col) e outro na coluna *body* da tabela *answers* e uma função para obter as questões filtradas(3.1.4.). A implementação encontra-se abaixo:

3.1.1.

```
ALTER TABLE questions ADD COLUMN full_text_index_col tsvector;
UPDATE questions SET full_text_index_col = to_tsvector('english',
COALESCE(title,'') || ' ' || COALESCE(body,''));
```

3.1.2.

```
CREATE TRIGGER questions_tsvector_update_trigger
BEFORE INSERT OR UPDATE ON questions FOR EACH ROW
EXECUTE PROCEDURE tsvector_update_trigger_column('full_text_index_col',
'pg_catalog.english', 'title', 'body');
```

3.1.3.

```
CREATE INDEX questions_question_search_idx ON questions USING
gin(full_text_index_col);
CREATE INDEX answers_question_search_idx ON answers USING gin(body);
```

3.1.4.

```
CREATE FUNCTION search_questions(psearch text)
RETURNS TABLE (question_id INTEGER) AS $func$
BEGIN
    RETURN QUERY
        SELECT DISTINCT(questions.id)
        FROM questions INNER JOIN answers ON questions.id =
answers.question_id
        WHERE questions.full_text_index_col @@ to_tsquery(psearch)
            OR to_tsvector(answers.body) @@ to_tsquery(psearch);
END
$func$  LANGUAGE plpgsql;
```

3.2. *Hash Index*

3.2.1. Identificamos que na tabela *users* duas colunas que beneficiem deste índice, nomeadamente a coluna *username* e a *email*, pois serão obtidos os seus dados através de ambas as colunas através de igualdades. A implementação encontra-se a seguir:

```
CREATE INDEX users_username ON users USING hash(username);
CREATE INDEX users_email ON users USING hash(email);
```

3.2.2. Identificamos também que na tabela *votes*, a coluna *votable_id* e *votable_type* beneficiam deste index pois muitas interrogações serão de joins e igualdades. A implementação encontra-se a seguir:

```
CREATE INDEX votes_votable_id ON votes USING hash(votable_id);
CREATE INDEX votes_votable_type ON votes USING hash(votes_votable_type);
```

3.3. Btree Index

3.3.1. Segundo o A4 submetido, as respostas de cada pergunta serão ordenadas pelo número de votos. Portanto decidimos aplicar um índice numa coluna da tabela *answers*, criada para aplicar o index (ordenando pelo rating dos votes):

```
CREATE INDEX answers_vote_rating ON answers USING btree(vote_rating);
```

3.3.2. As questões, por norma serão ordenadas pela coluna *updated_at*. Portanto decidimos aplicar um índice neste coluna da tabela *questions*:

```
CREATE INDEX questions_updated_at ON questions USING btree(updated_at);
```

4. User-defined Functions

A criação de user-defined functions, aumenta a performance dos pedidos realizados à base de dados, para computações complexas.

No projeto, foram criadas 8 destas funções, com o objetivo de facilitar, uniformizar e representar entidades, que necessitam de recolher e tratar dados de várias tabelas. As quais se encontram descritas em seguida:

4.1. Criada com o objetivo de obter o *rating* (soma de votos) de uma determinada pergunta ou resposta.

```
CREATE FUNCTION votable_rating(pvotable_id INT, pvotable_type CHARACTER)
RETURNS INTEGER AS $func$
DECLARE vrating INTEGER;
BEGIN
    SELECT COUNT(*) INTO vrating FROM votes WHERE votes.votable_id =
pvotable_id AND votes.votable_type = pvotable_type AND votes.value=TRUE;
    SELECT (vrating-COUNT(*)) INTO vrating FROM votes WHERE votes.votable_id
= pvotable_id AND votes.votable_type = pvotable_type AND votes.value=FALSE;

    RETURN vrating;
END
$func$  LANGUAGE plpgsql;
```

4.2. Criada com o objetivo de obter o *rating* (soma de votos recebidos nas perguntas e respostas) de um determinado *user* (usa o procedure votable_rating)

```
CREATE FUNCTION count_vote_rating_received_user(puser_id INT)
RETURNS INTEGER AS $func$
DECLARE vquestionsCount INTEGER;
DECLARE vanswersCount INTEGER;
BEGIN
    SELECT SUM(votable_rating(questions.id, 'q')) INTO vquestionsCount FROM
questions WHERE questions.user_id = puser_id;
    SELECT SUM(votable_rating(answers.id, 'a')) INTO vanswersCount FROM
answers WHERE answers.user_id = puser_id;

    RETURN vquestionsCount + vanswersCount;
END
```

```
$func$  LANGUAGE plpgsql;
```

4.3. Criada com o objetivo de obter o número de *users* que o *user* está a seguir.

```
CREATE FUNCTION number_users_followed_by(puser_id INT)
RETURNS INTEGER AS $func$
DECLARE vcount INTEGER;
BEGIN
    SELECT COUNT(*) INTO vcount FROM follows WHERE follows.user_id =
puser_id;

    RETURN vcount;
END
$func$  LANGUAGE plpgsql;
```

4.4. Criada com o objetivo de obter o número de *users* que se encontram a seguir o *user*.

```
CREATE FUNCTION number_users_following(puser_id INT)
RETURNS INTEGER AS $func$
DECLARE vcount INTEGER;
BEGIN
    SELECT COUNT(*) INTO vcount FROM follows WHERE follows.followed_id =
puser_id;

    RETURN vcount;
END
$func$  LANGUAGE plpgsql;
```

4.5. Criada com o objetivo de obter as respostas, o seu *rating* e informação do *user* de uma determinada pergunta.

```
CREATE FUNCTION question_answers(pquestion_id INT)
RETURNS TABLE (
    id INTEGER,
    user_id INTEGER,
    username CHARACTER VARYING(50),
    body TEXT,
    created_at TIMESTAMP,
    votes_rating INT
    ) AS $func$
BEGIN
    RETURN QUERY
      SELECT answers.id, answers.user_id, users.username, answers.body,
answers.created_at, votable_rating(answers.id, 'a')
      FROM answers RIGHT JOIN users ON answers.user_id = users.id
      WHERE answers.question_id = pquestion_id;
END
$func$  LANGUAGE plpgsql;
```

4.6. Criada com o objetivo de obter as tags de uma determinada pergunta.

```
CREATE FUNCTION question_tags(pquestion_id INT)
RETURNS TABLE (tag CHARACTER VARYING(10)) AS $func$
BEGIN
    RETURN QUERY
        SELECT tags.tag
        FROM tags INNER JOIN questions_tags ON tags.id =
questions_tags.tag_id
        WHERE questions_tags.question_id = pquestion_id;
END
$func$  LANGUAGE plpgsql;
```

4.7. Criada com o objetivo de obter as questões de um *user* e os seus *ratings*.

```
CREATE FUNCTION user_questions(puser_id INT)
RETURNS TABLE (
    id INTEGER,
    title CHARACTER VARYING(100),
    body TEXT,
    solved BOOLEAN,
    created_at TIMESTAMP,
    updated_at TIMESTAMP,
    votes_rating INT,
    count_answers BIGINT

    ) AS $func$
BEGIN
    RETURN QUERY
      SELECT questions.id, questions.title, questions.body,
questions.solved, questions.created_at, questions.updated_at,
votable_rating(questions.id, 'q'), (SELECT COUNT(*) FROM answers WHERE
question_id = questions.id)
      FROM questions
      WHERE questions.user_id = puser_id;
END
$func$  LANGUAGE plpgsql;
```

4.8. Criada com o objetivo de obter as informações de perfil de um *user*.

```
CREATE FUNCTION user_profile(puser_id INT)
RETURNS TABLE (
    id INTEGER,
    title CHARACTER VARYING(100),
    body TEXT,
    solved BOOLEAN,
    created_at TIMESTAMP,
    updated_at TIMESTAMP,
    votes_rating INT,
    count_answers BIGINT

    ) AS $func$
```

```
BEGIN
    RETURN QUERY
      SELECT questions.id, questions.title, questions.body,
questions.solved, questions.created_at, questions.updated_at,
votable_rating(questions.id, 'q'), (SELECT COUNT(*) FROM answers WHERE
question_id = questions.id)
      FROM questions
      WHERE questions.user_id = puser_id;
END
$func$  LANGUAGE plpgsql;
```

— *LBAW*

[MediaLibrary]