Grupo Nº 23

TÉCNICO LISBOA

BASES DE DADOS

1.º Semestre 2015/2016

PROJECTO – 2 ª PARTE

Relatório de Projecto

Realizado por:

Ricardo Silva – 78414

Eduardo Rodrigues -78431

João Tiago Raposo - 79027

Índice

1 Introdução

1.1 Breve descrição do Problema

Este projecto consiste na implementação, estruturada sobre uma base de dados, de um **bloco de notas.**

Este bloco de notas é constituído por **páginas e registos**. Cada registo pode pertencer a um **tipo de registo** que por sua vez poderá ter vários campos que serão preenchidos consoante os registos.

Para aceder a um bloco de notas é necessário efectuar um **login**, e para tal é necessário um **email** e uma **password**; o utilizador poderá ter ou não sucesso consoante os dados inseridos e a sua correspondência à base de dados.

2 Consultas SQL

Independentemente de termos ou não uma base de dados bem estruturada, é essencial poder obter a informação que se encontra na base de dados e é relevante para o nosso problema; isto é, é essencial poder **filtrar a informação** que se encontra na base de dados consoante o problema em questão.

Nas secções abaixo é mostrado o raciocínio que nos leva da necessidade de responder a uma pergunta sobre o modelo de dados à query em SQL que o torna possível.

NOTA: Ao longo do desenvolvimento das Queries não consideramos o facto das paginas, registos, etc estarem ou não activos. No entanto, bastaria verificar se "activo=0" e se sim, não adicionar esse dado ao cálculo da Query.

2.1 Quais são os utilizadores que falharam o login mais vezes do que tiveram sucesso?

Explicação: Iremos verificar (contar) quantas vezes cada utilizador falhou o acesso (sucesso=0) e quantas vezes um utilizador conseguiu o acesso (sucesso=1). De seguida comparamos quais os utilizadores que falharam mais vezes o login do que tiveram sucesso. Devolvemos esses utilizadores.

Query:

```
SELECT nome
FROM utilizador
WHERE userid in (
      SELECT userid
      FROM (
        (SELECT userid, count(*) as s FROM login WHERE sucesso=1 GROUP BY userid ) as s1
                NATURAL JOIN
        (SELECT userid, count(*) as f FROM login WHERE sucesso=0 GROUP BY userid) as s2)
      WHERE s<f);
```

2.2 Quais são os registos que aparecem em todas as páginas de um utilizador?

Explicação: Neste caso temos uma **divisão,** isto é, bastaria dividir uma tabela com todos os registos que existem em associações registo-página de um utilizador, por todos os registos das respectivas páginas. Se existisse alguma página que tivesse todos os registos, então esta estaria no resultado da divisão**.** No entanto, o MySQL não suporta divisões de tabelas, pelo que vai ser necessária uma abordagem alternativa. Vamos verificar, para um dado utilizador, se existe algum registo que não apareça em alguma das suas páginas, e se existir, então esse registo não aparece, certamente, em todas as páginas.

Query:

```
SELECT R.regid
FROM reg_pag R
WHERE R.userid =”some_user_id” and not exists(
 SELECT RP.regid
 FROM reg_pag RP
 WHERE not exists(
      SELECT R2.regid
      FROM reg_pag R2
      WHERE
            R.pageid=R2.pageid and R2.regid = RP.regid and R.userid=*some_userid*));
```

2.3 Quais são os utilizadores que têm o maior número médio de registos por página?

Explicação: Verificamos, para cada utilizador, qual o seu número de registos distintos e páginas distintas em associações registo-página. Devolvemos uma tabela ordenada pela relação registos/páginas de cada utilizador.

Query:

```
SELECT nome
FROM utilizador
NATURAL JOIN
 (SELECT userid, MAX(t2.r/t1.p)
  FROM ((SELECT userid, count(distinct pageid) as p from reg_pag GROUP BY userid) as t1
   NATURAL JOIN
        (SELECT userid, count(distinct regid) as r FROM reg_pag GROUP BY userid) as t2)
 ) as t;
```

2.4 Quais são os utilizadores que, em todas as suas páginas, têm registos de todos os tipos de registos que criaram?

Explicação: Tal como em 2.2, temos novamente uma divisão. Uma vez que já sabemos que o MySQL não suporta divisões de tabelas vamos reformular o problema. A ideia é “Para cada tipo de registo de cada utilizador verificamos se existe alguma pagina em que esse tipo de registo não apareça, verificamos também se esse utilizador tem associações entre registos e páginas”.

Query:

```
SELECT U.nome FROM utilizador U
 WHERE NOT EXISTS (
      SELECT TR.typecnt
        FROM tipo_registo TR
      WHERE U.userid = TR.userid and NOT EXISTS(
                  SELECT RG.typeid
                  FROM reg_pag RG
                  WHERE
                              RG.typeid=TR.typecnt and RG.userid=TR.userid and RG.userid = U.userid))
  and U.userid in( select userid from reg_pag);
```

3 Restrições de Integridade

A necessidade de imposição de restrições de integridade surge por não haver possibilidade de uma dada restrição não poder ser definida pela estrutura do modelo de dados (nomeadamente, o modelo E-A ou o modelo Relacional). Para nos certificarmos que a base de dados mantém a integridade de acordo com estes parâmetros, podemos recorrer a diferentes procedimentos quer ao nível da aplicação quer ao nível da base de dados. Tendo em conta que, ao projectar uma base de dados, não há controlo sobre que tipo de aplicações vão trabalhar sobre esta, o mais seguro e correcto é assegurar o bom funcionamento da base dados a partir dela própria, e não expor fragilidades que teriam de ser compensadas ao nível aplicacional.

3.1 Explicação do Problema

É pedido que criemos uma forma de implementar a seguinte restrição de integridade:

"Todo o valor de contador_sequencia existente na relação *sequencia* existe numa e uma vez no universo das relações do tipo_registo, página, campo, registo e valor."

3.2 Explicação da ideia

Existem diversas formas de definir restrições de integridade. No entanto, para este caso, a forma indicada para representar esta restrição é o uso de um Trigger.

Um trigger requer esforço de forma a obter as condições que nos permitam considerar que uma determinada acção pode ou não ser executada, mas permite também implementar a maior parte das restrições sobre qualquer base de dados.

No nosso problema, após uma inserção ou um update a cada uma das tabelas sobre as quais incide a restrição de integridade, é despoletado um trigger. Este verifica se o valor que foi inserido existe em mais alguma tabela. Para tal, calcula a soma das vezes que esse valor aparece em todas as tabelas que contenham um *idseq* igual ao do registo que vai ser inserido. Caso o resultado desta soma seja seja diferente de 1, é lançada uma excepção.

É necessário que este trigger seja despoletado tanto na inserção como na inserção da qualquer uma das tabelas pagina, registo, tipo_registo, campo ou valor. Por isso, e como o código executado é idêntico para qualquer uma das situações descritas, são criados 10 triggers diferentes, um para cada situação, mas todos com a mesma função: chamar o procedimento que faz a verificação necessária (a stored procedure chkseq).

3.3 Código

Nota: O código não se encontra representado, na sua totalidade, nas linhas que se seguem. O código aqui apresentado foi o que considerámos essencial para perceber a ideia da implementação. A versão completa poderá ser consultada em formato digital nos ficheiros entregues no Fénix.

```
delimiter //

DROP PROCEDURE IF EXISTS chkseq;
CREATE PROCEDURE chkseq (seq_nr INT)
BEGIN
  DECLARE seq_count INT;

        SET seq_count = (SELECT count(*) FROM pagina WHERE idseq=seq_nr) +
                        (SELECT count(*) FROM registo WHERE idseq=seq_nr) +
                        (SELECT count(*) FROM tipo_registo WHERE idseq=seq_nr) +
                        (SELECT count(*) FROM campo WHERE idseq=seq_nr) +
                        (SELECT count(*) FROM valor WHERE idseq=seq_nr);
        IF (seq_count!=1)
        THEN
                SIGNAL sqlstate '45001' set message_text = "Something Went Wrong!";
        END IF;
END;

CREATE OR REPLACE TRIGGER before_pagina_insert
BEFORE INSERT ON pagina
FOR EACH ROW
CALL chkseq(NEW.idseq);
(…)

delimiter;
```

4 Desenvolvimento da Aplicação

De forma a interagir com a base de dados, foi criada uma aplicação em PHP que permite realizar diversas operações, o interface da aplicação foi baseado no exemplo do enunciado da parte 1 do projecto.

A aplicação encontra-se disponível em http://web.ist.utl.pt/ist178414/.

Antes de explicar cada alínea em baixo, vale a pena explicar o login.

Para que qualquer alteração seja feita ao bloco de notas é necessário efectuar um login sobre o mesmo. Para este caso, os ficheiros login.php e login_script.php são os responsáveis pela tarefa. Em login_script.php são realizadas as seguintes acções: verificação da existência do user, verificação do sucesso da tentativa de login e registo da informação gerada pela tentativa de login. Caso a password esteja incorrecta é apresentado a página wrong_password.php e caso o email não se encontre na base de dados é apresentada a página user_not_found.php.

Após o login é apresentado um ecrã definido pelo ficheiro main_page.php onde são listadas as páginas e os tipos de registos do utilizador.

4.1 Inserir uma nova página

Esta acção é efectuada a partir do ecrã principal, clicando no botão 'Criar página' e utiliza os ficheiros: new_page.php e new_page_script.php.

4.2 Inserir um novo tipo de registo

Esta acção é efectuada a partir do ecrã principal, clicando no botão 'Criar tipo registo' e utiliza os ficheiros: new_register_type.php e new_register_type_script.php.

4.3 Inserir novos campos para um tipo de registo

Esta acção é efectuada a partir do ecrã tipo de registo que é acedido clicando no nome de um tipo de registo na página principal, clicando no botão 'Criar campo' e utiliza os ficheiros: new_field.php e new_field_script.php.

4.4 Retirar uma página

Esta acção é efectuada a partir do ecrã principal, clicando no botão 'Remover página' e utiliza o ficheiro remove_page_script.php.

4.5 Retirar um tipo de registo

Esta acção é efectuada a partir do ecrã principal, clicando no botão 'Remover tipo de registo' e utiliza o ficheiro remove_register_type_script.php.

4.6 Retirar um campo de um tipo de registo

Esta acção é efectuada a partir do ecrã tipo de registo que é acedido clicando no nome de um tipo de registo na página principal, clicando no botão 'Remover campo' e utiliza o ficheiro: remove_field_script.php.

4.7 Inserir um registo e os respetivos valores dos campos necessários

Esta acção é efectuada a partir do ecrã página que é acedido clicando no nome de uma página na página principal, clicando no botão 'Criar registo' e utiliza os ficheiros: new_register.php, new_register2.php e new_register_script.php.

4.8 Ver uma páginas com os registos nela contidos

Esta acção é efectuada a partir do ecrã principal, clicando no nome de uma página e utiliza o ficheiro:

page.php.

5 Formas Normais

Dada uma relação, as formas normais permitem classificar o tipo de redundância que pode existir na relação.

5.1 Em que forma normal se encontra a relação utilizador?

Relação em causa:

utilizador(userid, email, nome, password, q1, r1, q2, r2, pais, categoria)

Dependências funcionais existentes:

userid → email, nome, password, q1, r1, q2, r2, pais, categoria

email → userid, nome, password, q1, r1, q2, r2, pais, categoria

Esta relação encontra-se na BCNF, e, por isso, também na 3NF, 2NF e 1NF. Explica-se de seguida como é que a relação atinge os critérios necessários para cada uma destas formas normais.

1NF:

Nesta relação, cada atributo contém apenas um valor atómico e não um conjunto de valores, pelo que se encontra na Primeira Forma Normal.

2NF:

Tendo em conta as chaves candidatas email e userid, todos os outros atributos da relação (ou seja, nenhum atributo que não faça parte de uma chave candidata) são determinados por cada uma destas chaves. Por este motivo, a relação encontra-se na Segunda Forma Normal.

3NF:

Todos os atributos da relação são determinados apenas pelas chaves candidatas, ou seja, não há nenhuma dependência funcional entre atributos que não pertençam a uma chave candidata. Isto é necessariamente verdadeiro, tendo em conta que as únicas dependências funcionais existentes têm como determinante uma chave candidata.

BCNF:

Para a relação *utilizador* estar na Forma Normal de Boyce Codd, é necessário que, para qualquer dependência funcional da relação:

- A dependência seja trivial;
- O determinante da dependência seja superchave.

Mais uma vez, como as únicas dependências existentes têm como determinante uma chave candidata (e, transitivamente, uma superchave), a relação respeita a BCNF.

5.2 Considerando uma nova dependência funcional

Acrescentando a dependência seguinte:

nome, password, q1, r1, q2, r2 → email

O conjunto {nome, password, q1, r1, q2, r2} passa a ser uma chave candidata, pois:

- É uma superchave: o seu fecho contém todos os atributos da relação;
- Nenhum seu subconjunto próprio é superchave da relação.

Por este motivo, a relação mantém-se na BCNF (e, por extensão, nas 1NF, 2NF e 3NF).

6 Índices

6.1 Breve explicação da importância dos índices

Em qualquer base de dados em que a quantidade de dados armazenados aumente é necessário aumentar a capacidade de acesso aos mesmos. Em certos casos, o excesso de tempo que pode ser gasto numa operação poderá originar uma grande penalidade na operação de consulta.
Por forma a reduzir esta situação, o uso de índices é uma solução, garantido o aumento da performance na consulta de dados, caso consideremos a consulta a uma grande quantidade de dados. Por outro lado, se analisarmos uma pequena quantidade de dados o uso de índices poderá ter o efeito contrário, ou seja, prejudicar a performance.

6.2 Devolver a média do número de registos por página de um utilizador

O primeiro passo a tomar é saber exactamente em que query é que se traduz esta pergunta. A query obtida foi:

```
SELECT AVG(count)
FROM (SELECT pageid, COUNT(*) as count
    FROM reg_pag
    WHERE userid=”some_user_id”
    GROUP BY pageid) as regcounts;
```

Para melhorar o desempenho da base de dados a responder a esta query, é necessário ter acesso a vários atributos da tabela reg_pag: o regid, o pageid e o userid. Com esta informação, temos tudo o que é necessário para responder à query. Assim, decidimos criar um índice nestes três atributos (apesar de serem mais atributos do que o normal num índice, são inteiros, pelo que não há grande impacto na memória gasta e desempenho do índice).

Interessa-nos ter o índice agrupado (clustered), mas isso apenas seria possível sobre o atributo primário da tabela, idregpag, que não traz informação relevante neste caso. É também necessário que o índice seja denso, caso contrário não evitaria a consulta da tabela em questão. Por estes motivos, o índice ideal a ser criado é um índice denso baseado numa Btree, não agrupado, sobre os atributos userid, pageid e regid da tabela reg_pag. Isto é conseguido através da seguinte query mySQL:

```
CREATE INDEX regpagidx USING BTREE ON reg_pag(userid, pageid, regid);
```

Tendo em conta que as tabelas contêm já diversos índices sobre estes e outros atributos, não foi possível verificar a melhoria de desempenho trazida por este índice, já que as queries eram resolvidas com base nos outros. O mesmo acontece para o índice criado na secção 6.3.

6.3 Ver o nome dos registos associados à página de um utilizador

Mais uma vez, começamos por ver que query corresponde à pergunta feita:

```
SELECT regcounter, nome
FROM registo
WHERE regcounter IN(   SELECT regid
                                    FROM reg_pag
                                    WHERE userid=”some_user_id” AND pageid=”some_page_id”);
```

Nesta query, verifica-se que o índice criado na secção anterior é também útil para a obtenção da resposta. Para além desse, é necessário ter conhecimento dos nomes dos registos. Isto pode ser possível a partir de um índice clustered por regcounter, o que mais uma vez não é possível devido à composição da chave primária desta tabela. Assim, pareceu-nos mais benéfico ter um índice composto com os atributos regcounter e nome; considerámos que o gasto de memória por ter os nomes dos registos na tabela do índice seria compensado por esta melhoria. Não havendo ordem relevante no índice, o tipo Hash possibilita a obtenção mais rápida dos valores pretendidos para um conjunto limitado de registos. Por esta razão, o índice criado foi:

```
CREATE INDEX regname USING HASH ON registo(regcounter, nome);
```

7 Transações

Para garantir a coerência da base de dados mesmo face a vários acessos concorrentes de aplicações diferentes, foi necessário incorporar no código PHP da aplicação do bloco de notas instruções SQL que garantissem esta coerência. Tendo em conta que o mySQL usa o *engine* InnoDB por defeito, que tem como nível de isolamento *repeatable read,* considerámos que não seria necessário acrescentar manualmente garantias de um maior nível de isolamento.

7.1 Alterações ao código PHP

Ao longo do código da nossa aplicação, surgem vários segmentos que precisam de realizar vários acessos sequenciais à base de dados, e que devem ser tratados como atómicos para que o resultado da operação seja coerente com o estado da base de dados. Assim, em partes como a criação de novos registos, tipos de registos ou páginas, ou na consulta de dados distintos interligados entre si, encapsulámos as queries SQL numa transação.

Para isso, no início do script correspondente à operação que vai ser atomizada, é executada a instrução $db->beginTransaction(); que inicia uma transação. Após a última query ser realizada, é executada a instrução $db->commit(); que tenta actualizar o estado da base de dados conforme necessário para que esta se mantenha coerente. Finalmente, caso deste processo resulte algum erro (sendo produzida e lançada uma excepção), a instrução $db->rollBack() faz com que as eventuais alterações realizadas não tenham efeito, mantendo a base de dados coerente. Neste ultimo cenário, é ainda mostrada uma mensagem de erro para que o utilizador tenha conhecimento do sucedido.

No código PHP enviado em anexo, é possível verificar a implementação das transações. Alguns dos scripts que as contêm são login_script.php, new_register_script.php, e remove_field_script.php.

8 Data Warehouse

8.1 Breve explicação do esquema de uma estrela na modelação da informação

A ideia de Data Warehouse consiste na consolidação de dados provenientes de muitas fontes num único repositório. Assenta num modelo em estrela. Um modelo em estrela tem: no centro uma Tabela de Factos, que armazena todos os factos ocorridos e as chaves para as dimensões correspondentes; em redor desta tabela existem tabelas de dimensões que guardam informação detalhada sobre os factos.

8.2 Criação, Estrutura e Carregamento do esquema em estrela

Para a criação do Data Warehouse foi em primeiro lugar necessário criar as tabelas envolvidas e depois populá-las. Foram criadas as tabelas d_tempo e d_utilizador, com os atributos pedidos conforme o enunciado, e foi ainda acrescentado um atributo chave a cada tabela, que pode facilmente ser obtido como função dos dados que representa. Desta forma, poupamos acessos às tabelas de dimensões quando estivermos a popular a tabela de factos.

Para popular as tabelas de dimensões foram seguidas duas abordagens diferentes, devido à natureza das tabelas. A dimensão tempo precisa de ter um conjunto de registos exaustivo que abranja todas as datas dos registos existentes na base de dados. Para gerar estas datas, foi usado um script que percorre todos os dias desde uma dada data inicial até à final e acrescenta, por cada dia, um registo com os atributos pretendidos à tabela d_tempo. A janela de datas pode ser facilmente ajustada face à criação de novos registos, alterando os parâmetros da chamada à stored procedure timedimbuild, que é o procedimento que popula esta tabela. Como identificador desta tabela foi usado um inteiro que representa simplesmente os seus três atributos concatenados (ano, mês, e dia), algo que pode ser facilmente criado sem aceder à tabela d_tempo.

Para popular a tabela da dimensão utilizador foi apenas necessário obter a informação de cada utilizador da tabela *utilizador,* recorrendo para isso à instrução SQL INSERT INTO SELECT. Foi usado como identificador da tabela o userid, por ser de fácil obtenção a partir de outras tabelas do modelo de dados, nomeadamente a tabela *login* que vai ser usada para preencher a tabela de factos.

Finalmente, é necessário popular a tabela de factos. Para isso, basta contar o número de acessos feitos por cada utilizador em cada intervalo de menor granularidade da dimensão tempo deste Data Warehouse, que neste caso é o dia. Mais uma vez, com a instrução INSERT INTO SELECT, foi possível fazer este processo com facilidade.

8.3 Obtenção da média de tentativas de login para todos os utilizadores de Portugal, em cada categoria, com um rollup por ano e mês

Esta query responde à pergunta fazendo uso do Data Warehouse criado:

```
SELECT categoria, ano, mes, AVG(attempts) as avg_attempts
FROM login_attempts NATURAL JOIN d_tempo NATURAL JOIN d_utilizador
WHERE pais='Portugal'
GROUP BY categoria, ano, mes WITH ROLLUP;
```

9 Conclusão

A segunda parte deste projecto permitiu consolidar diversas partes da matéria que seriam impossíveis quer com os exercícios regulares das aulas práticas, uma vez que não são tão complexos, quer com as aulas teóricas, uma vez que não é possível, dado o tempo, resolver exercícios tão variados.

Sendo assim, considerámos o projecto uma ferramenta essencial de estudo na perceção, compreensão e assimilação dos conhecimentos relativos à cadeira de Bases de Dados. Essencialmente, o desenvolvimento deste trabalho permitiu-nos, mais do que as outras componentes desta cadeira, perceber que ferramentas temos ao nosso dispor para a projecção, desenvolvimento e manutenção de uma base de dados, que engloba não só o modelo de dados que representa a aplicação em foco, mas também as estruturas que nos fornecem meta-dados de grande importância.

Ganhámos bastante experiência com a finalização de todo o projecto, o que nos faz sentir mais confiantes para trabalhar com bases de dados no futuro sempre que seja necessário.