My Last Project: Visões, Otimização, Stored Procedures e Controle de Concorrência

Marco A. Almeida[1]

[1]Instituto de Matemática – Universidade Federal da Bahia (UFBA)
Av. Adhemar de Barros S/N – 40.170-110 – Salvador – BA – Brazil

marco062@dcc.ufba.br

Resumo. *O presente artigo apresenta algumas funcionalidades da ferramenta My Last Project utilizando recursos avançados de banco de dados como o uso de visões (*views*), otimização através de índices, inserção de gatilhos (*triggers*),* stored procedures *e estratégias de controle de concorrência.*

1. Introdução

O objetivo da ferramenta My Last Project é promover a interação entre professores e estudantes, através de uma plataforma colaborativa para que os últimos atingam os seus interesses e aspirações na graduação, criando um projeto final de qualidade.

Na sua segunda versão, My Last Project oferece uma camada a mais de segurança e performance ao usar *views*, abordadas na seção 2; a seção 3 apresenta os atributos que foram indexados no banco e os efeitos dessa decisão são explicados na seção 4; ... por fim é apresentado planejamento.

2. Views

Uma visão (*view*) funciona como uma janela, dando uma determinada perspectiva do Banco de Dados. As *views* contudo, não existem fisicamente no banco, são apenas uma forma de obter os dados que existem em outras tabelas ou *views*.

Em My Last Project, estudantes recém cadastrados necessitam elegir um projeto para acessarem todas as funcionalidades presentes para o seu tipo de usuário. Eles irão interagir com uma lista onde estão os professores e seus respectivos projetos. O estudante não necessita ter acesso a todas as informações do professor, portanto como produto da junção das tabelas professor e projeto retornaríamos apenas as colunas: `professors.id`, `professors.name`, `professors.email`, `projects.id`, `projects.title`, `projects.summary` e `projects.due_at`. Para eliminar a necessidade de se escrever essa consulta com frequência foi criada a view `professor_projects` que realiza a listagem definida anteriormente.

A *query* resultante está descrita a seguir:

```
CREATE VIEW professor_projects AS
  SELECT professors.id, professors.name, professors.email,
  projects.id, projects.title, projects.summary, projects.due_at
  FROM professors
    INNER JOIN projects ON projects.professor_id = professors.id
```

O uso desse recurso diminue a *query* resultante utilizada para a ação de consulta dos projetos dos professores. Segue a consulta:

```
SELECT * FROM professor_projects ORDER BY professors.name
```

No *framework* de desenvolvimento da aplicação é necessário fazer algumas abstrações para poder trabalhar na *view*. A *view* será vista como uma tabela que só pode realizar ações de leitura. Dessa forma é criado um modelo chamado `ProfessorProject` que irá fazer a mesma listagem definida acima através do comando `ProfessorProject.order(:name)`. Comandos como `.save()` e `.destroy()` não funcionam para registros carregados desta *view*.

3. Indexes

Para evitar um custosa reordenação dos registros de uma tabela utilizamos o recurso de indexação para criar uma estrutura em paralelo para ordenar algumas colunas de uma determinada tabela. Em geral, utilizamos esse recurso quando queremos acelerar uma consulta por um determinado parâmetro. A decisão de uso de índices vai implicar também num período de alteração mais lento dos registros, já que eles deverão ser reordenados nos arquivos de indexação. Dessa forma, é importante ter consciência dos prós e contras no processo de indexação de uma tabela.

PostgreSQL cria automaticamente índices para as chaves primárias. Ela atribui automaticamente os índices de `NOT NULL` e `UNIQUE`. As chaves estrangeiras contudo, não tem índices automaticamente criados. Dessa forma, todas as chaves estrangeiras apresentadas na etapa de modelagem tiveram seus índices criados para garantir essa associação.

Além das colunas relacionadas aos relacionamentos entre tabelas terem sido indexadas, outras colunas também participaram do processo de indexação para facilitar as consultas frequentes que são feitas por esses parâmetros. A seguir está a lista completa das tabelas e suas respectivas colunas e motivos:

users

`email`: Quando um usuário executa a ação de log in, é através da consulta do seu e-mail que verificamos a sua existência no sistema.
`reset_password_token`: Quando o usuário perde informação da sua conta um *token* é criado para que ele possa recuperá-la. Esse token deve ser único de forma que só exista um token por usuário.

projects

`title`: Estudantes decididos em que projeto final deseja inscrever-se não necessita navegar por todos os projetos, pode diretamente buscar pelo seu título.

materials

`title`: Idem ao título de projeto.
`relevance`: Alguns materias podem ser mais relevantes que outros para um estudante. Estudantes podem consultar um material pelo nível de relevância com o seu projeto.

authors

`name`: Autores podem ser consultados pelo seu nome.

tasks

`title`: Idem ao título de projeto.

Graças à funcionalidade de migrations, o framework de desenvolvimento facilita a criação de índices no banco. Através do comando:

```
add_index :column, :field [, unique:  true]
```

índices podem ser criados e destruídos do banco.

4. Optimization

O tempo de resposta e os índices aplicados foram os parâmetros utilizados para analisar se uma tentativa de otimização de uma determinada consulta foi bem sucedida. Cada consulta foi executada dez vezes com índice e dez vezes sem índice e daí extraída a média para determinar o resultado.

A criação e remoção de índices é feita manualmente para realizar esses testes e com a opção `\timing` do PostgreSQL é possível calcular o tempo de resposta de determinadas consultas. A seguir estão listados os testes feitos e os seus respectivos resultados.

1. ***Log in* de usuário**
   O procedimento de *log in* é simples. Primeiro é encontrado o usuário através do seu email e depois um algoritmo é executado para avaliar se a senha é idêntica. A consulta resultante evidencia que podemos otimizá-la utilizando um índice no email. A base de usuários no momento atual conta com 1000 usuários.

   ```
   SELECT "users".* FROM "users"
     WHERE "users"."email" = 'marco062@dcc.ufba.br'}
   ```

   Resultado: Essa consulta sem índice tem uma média de resposta de 1,121 ms, enquanto com o índice tem uma resposta de 0,697 ms. A melhoria na performance da tabela com índice x sem índice foi de 160%.

2. **Busca por materiais**
   A página de materiais é uma página visitada constantemente tanto por professores como por estudantes. Os únicos materias que devem ser apresentados são aqueles que pertencem ao usuário, ou seja, aqueles com user_id iguais ao identificador do usuário. Nessa situação então dois índices devem ser criados: um para a chave estrangeira e outro para o título do material. A consulta para esse problema está listada a seguir. São 5.000 registros de referência e 1000 registros de usuário. Para testar a relevância do uso de um índice na chave estrangeira foi testado também o tempo de respota apenas com a sua exclusão.

   ```
   SELECT "materials".* FROM "materials"
     INNER JOIN "users" ON "users"."id" = "materials"."user_id"
     WHERE "users"."id" = 30
       AND "materials".title = 'Grass-roots tertiary utilisation'
   ```

   Resultado: Sem índice essa consulta tem uma média de tempo de resposta de 2,322 ms, para a remoção do índice da chave estrangeira tece um tempo médio de 1,012 ms e com o índice proposto de 0,944 ms. Nesse caso acrescentar o índice na chave estrangeira agregou pouco para a velocidade da consulta. A melhoria na performance da tabela com índice x sem índice foi de 246%.

3. **Busca de materiais por autor**
   Os autores tem uma relação de muitos para muitos com materiais. Para criar essa relação foi necessária uma tabela para registrar essa associação, dessa forma a consulta é um pouco mais extensa porque que é necessário agregar com várias tabelas e uma boa estratégia para a consulta é muito importante. Foi adicionado um índice no nome do autor

para acelerar essa consulta. A tabela de associação entre autores e materiais já está indexida, pois é uma chave primária composta, e como já foi dito anteriormente, o PostgreSQL já cria um índice único para as chaves primárias. A seguir está o SQL da consulta e os resultados da otimização. Estão registrados nos bancos de dados 20.000 autores.

```
SELECT "materials".* FROM "materials"
  INNER JOIN "users"
    ON "users"."id" = "materials"."user_id"
  INNER JOIN "authors_materials"
    ON "materials".id = "authors_materials".materials_id
  INNER JOIN "authors"
    ON "authors_materials".author_id = "authors".id
  WHERE "users"."id" = 30
    AND "authors"."name"='Makayla Hahn';
```

Resultado: Para a consulta sem índice o tempo médio de resposta foi de 1,761 ms. Removendo apenas o índice da chave estrangeira (user_id) a média sobe para 1,956 ms. Removendo também o índice do nome do autor o tempo médio é 6,481 ms. A melhoria na performance da tabela com índice x sem índice foi de 368%.

É visível que a boa aplicação de índices pode melhorar de maneira efetiva as consultas no banco. Deve-se apenas manter a atenção em não indexar toda a tabela e sim apenas o que é necessário, porque esse procedimento pode afetar as operações de escrita no banco.

5. Stored Procedures e Triggers

Stored Procedures são métodos que ficam armazenados no banco de dados e permitem serem chamadas por diversas aplicações sem essas terem que implementá-las. Aliada a triggers é uma estratégia útil para manter a integridade do banco de dados.

Em My Last Project quando novos estudantes se inscrevem em um projeto os professores necessitam ser notificados que existe um estudante se aplicando em um de seus projetos. Isso é feito através de uma trigger `student_subscription` que chama uma stored procedure `notify_user` para inserir um registro na tabela notifications com a mensagem predefinida depois de um estudante se increver em um projeto.

Abaixo segue a listagem com os códigos para a criação da stored procedure para armazenar a notificação no banco e a segunda para invocar a notificação quando o usuário seleciona o projeto no qual participará respectivamente.

```
CREATE FUNCTION notify_user(int, varchar)
  RETURNS void AS
    $$
      INSERT INTO notifications (user_id, message, created_at)
        VALUES ($1, $2, now());
    $$
  LANGUAGE SQL;

CREATE OR REPLACE FUNCTION student_subscription_trigger()
  RETURNS trigger AS
    $$
```

```
    DECLARE
      user_id integer;
      message varchar := 'projects.subscription.professor.notification';
    BEGIN
      SELECT INTO user_id \"projects\".\"user_id\" FROM \"projects\"
        WHERE \"projects\".\"id\" = NEW.\"project_id\";
      SELECT INTO user_id notify_user(user_id, message);
      RETURN NEW;
    END;
    $$
    LANGUAGE plpgsql;

CREATE TRIGGER student_subscription
  AFTER UPDATE OF project_id ON users
  FOR EACH ROW
    EXECUTE PROCEDURE student_subscription_trigger();
```

No código Ruby a chamada é feita através do comando de atualização do atributo project_id do estudante.

```
Student.update_attribute :project_id, project.id
```

6. Transactions

O controle de transações permite que a integridade dos dados se mantenham no banco. Uma transação consiste em uma unidade lógica de trabalho em que todas as ações serão realizadas ou nenhuma das operações é realizada.

No framework utilizado para o desenvolvimento da aplicação My Last Project todos os comandos que escrevem no banco de dados são involucrados em um transação. Isso garante por exemplo, que, se algum professor tentar alterar o material de um estudante ao mesmo tempo que este o edita, as transações serão realizadas na mesma ordem em que foram requeridas.