**Alexandre**

»» Apresentação

Boa tarde a todos. Hoje, eu (Alexandre Correia) juntamente com os meus colegas Jessica Jesus e Pedro Ferreira vamos fazer uma apresentação sobre o sistema de gestão de bases de dados que estudámos ao longo do semestre, PostgreSQL, de uma forma interativa com o sistema.

Para começar, achámos que seria interessante usar uma imagem docker do Postgres nesta apresentação hands-on em vez de o instalarmos nas nossas máquinas. Pelo que, se o entenderem, podem seguir os nossos passos mais facilmente.

Vou então começar pelo Armazenamento e Sistema de Ficheiros.

»» Cluster

Cada servidor do Postgres é designado de cluster e contém 3 bases de dados por omissão:

 - template1: template a ser usado para criar novas bases de dados, pelo que se for alterado, essas alterações aplicam-se a todas as novas bases de dados;

 - template0: semelhante ao template1, mas este não pode ser alterado após a criação do cluster. Permitindo assim a reconstrução de bases de dados sem que haja conflitos de objetos;

 - postgres: base de dados que pode ser usada pelos utilizadores e outras aplicações. É onde vamos trabalhar hoje.

(Mostrar) Se repararmos aqui na navegação lateral do DataGrip, só conseguimos ver a base de dados postgres. Isto é porque, quando nos ligamos a um cluster, ligamo-nos automaticamente a uma base de dados e, por razões de segurança, só conseguimos aceder ao conteúdo dessa base de dados.

»» Base de dados

Cada base de dados tem 4 schemas por omissão:

 - pg\_toast: é usado para guardar atributos com valores grandes;

 - pg\_catalog: é usado internamente pelo Postgres;

 - public: é onde, por omissão, os utilizadores e as aplicações guardam os dados;

 - information\_schema: é usado para mostrar as informações internas do Postgres via SQL.

(Mostrar) Podemos ver isto em ação ao correr esta query.

»» Schema

Mas porquê tantos schemas?

Os schemas de cada base de dados são bastante úteis pois permitem-nos:

 - organizar as tabelas e objetos logicamente;

 - ter várias tabelas com os mesmos nomes sem gerar conflitos, como iremos ver dentro de segundos;

 - entre outros.

(Mostrar) Vamos então ver a utilidade dos schemas. Vamos criar uma tabela em public e outra em fct. Agora inserimos dados em ambas as tabelas e podemos ver que não houve nenhum problema.

»» Particionamento

Uma outra funcionalidade interessante do Postgres é o particionamento de tabelas.

O particionamento permite-nos dividir uma tabela em várias sub-tabelas.

(Mostrar) Vamos então criar a tabela mãe que contém todos os alunos e agora 3 partições, uma com os alunos de Lisboa, outra com os do Porto e outra com os de Coimbra.

Inserimos agora alguns dados de alunos fictícios e conseguimos ver que já estão cá vários alunos espalhados pelas 3 cidades.

A grande vantagem de usar particionamento é, por exemplo, se quisermos eliminar os registos de todos os estudantes do Porto. Em vez de fazermos vários DELETE sobre a tabela dos estudantes, podemos simplesmente eliminar (com DROP) a partição do Porto, o que torna a operação bem mais rápida e eficiente.

-- NOP -- »» TOAST

Como sabemos, sempre que temos atributos com tamanho maior que 2kB, estes são armazenados num local especial do Postgres: pg\_toast.

(Mostrar) Se criarmos uma tabela para guardar o nosso artigo de SBD (com tipo bytea) e se tentarmos guardar (em binário) o conteúdo do artigo, reparamos que ele não guarda o conteúdo mas sim a localização do conteúdo, que neste caso está em pg\_toast, como podemos ver aqui.

Com esta breve demonstração de como o TOAST funciona, achamos que é também importante mostrar o particionamento e a sua utilidade.

**Jessica**

Passando agora para a parte dos Índices e Hashing, como foi dito, o PostgreSQL suporta os seguintes índices: B-trees, Hash, GiST, SP-GiST, GIN e BRIN.

Para a criação e demonstração destes indices, foi utilizado o script de uma das práticas, relativo aos census. Este script cria uma tabela e são inseridos cerca de 30 000 tuplos.

B-Trees

Começando pelos índices B-tree, estes são os índices default aquando a criação de um índice, através do seguinte comando: \*mostrar criação de indice com create index na base de dados\*. Este permite ser utilizado com vários operadores (neste caso temos o operador ‘=’, mas também é possível com combinações (mostrar a query com o between), condições ou operadores de pattern matching (mostrar com a query like).

Com estes indices, é possível ver que a performance da procura é muito melhor (geralmente metade).

Hash

Já para o hash ( e restantes indices) é necessário especificar que tipo vai ser usado. Este indice só começou a ser seguro a partir da versão 10 do Postgres, no entanto, ainda tem várias limitações, sendo só possível usá-lo com comparações de igualdade (mostrar ele a ser usado e usar o pattern matching operator para ver-se que o hash não é usado).

Este indice, principalmente com bastantes tuplos (na ordem de 500 000 ou 1M), tem uma melhor performance que o indice b-tree e ocupa menor espaço, como se pode ver com a execução destas queries \*mostrar queries\*. O postgresql geralmente escolhe o melhor indice a ser utilizado, podendo ver-se isso agora, pois, apesar de termos os dois indices ativos \*mostrar indices existentes\*, ele escolhe o de hash.

GIN

Este indice é um indice generalizado invertido e geralmente é usado para pesquisas de full-text. Neste caso, é utilizada a seguinte query \*mostrar\* que usa esta opção (gin\_trms\_ops) para dizer ao Postgres para usar trigramas nas colunas selecionadas. Trigrama é uma vertente do n-grama (uma sequência contígua de *n* itens de uma determinada amostra de texto).

GiST

Este indice é uma arvore de procura generalizada e funciona bem para pesquisas de full-text e dados geométricos. Como não temos dados geométrcos, vamos só demonstrar uma pesquisa de texto. Tal como o GIN, tambem usa a opção gist\_trms\_ops para dizer ao postgres para usar trigramas.

Como este indice pode ter perdas ou falsos matches, vamos só confirmar se dá o mesmo numero que a pesquisa sequencial.

SP-GiST

Este indice é a versão do GiST para particionamento de espaço, ou seja, é bom para espaços que possam ser divididos (ou particionados). Para este indice não vou mostrar nenhum exemplo, uma vez que seria melhor em outros tipos de dados além dos que temos nesta base de dados (e por causa do tempo).

BRIN

Por fim, temos o Block Range index que guarda um “resumo” da informação acerca dos blocos da página. Estes indices são mais pequenos e têm menos custo que as b-trees, permitindo assim usar em tabelas bastante grandes que se calhar com b-trees teria de se particionar.

Este indice é bom para ser usado quando existem timestamps na base de dados e quando estas são muito grandes, como por exemplo, terem cerca de 100 M filas, por isso não iremos mostrar.

Indices multicoluna

Por fim, fazer uma rápida demonstração de indices multicoluna para ter ideia da performance. Este foi feito com indices b-tree.

Respostas às perguntas das reviews:

**Ferreira**

Relativamente à parte de query processing, o processamento é feito através de um processo que corre em backend e que lida com todas as queries emitidas por clientes conectados. Estas queries passam por uma sequência de 5 passos:

* O parser - Responsável por analisar a query e criar uma parser tree da query
* O Analyzer - Responsável por analisar a semântica da query e construir a query tree usando a parse tree
* Rewriter
* Planner
* Executor

Este conjunto de passos é responsável por retornar um resultado da forma mais eficiente possível baseando a sua otimização no cálculo do caminho mais barato, ou seja, através de cost-based optimizations.

Para analisar o query plan das queries podemos usar o comando Explain e para avaliar os custos de execução podemos usar o comando Analyze.

Respostas às perguntas das reviews:

1 - A materialização é um mecanismo usado sempre que for conveniente para

armazenar o resultado da árvore de execução. Por exemplo, junção de nested loop join requer percorrer a tabela à direita várias vezes, portanto, se

não cabe na memória, é preferível materializá-lo temporariamente em

Disco. O PostgreSQL fornece internamente um módulo de armazenamento temporário para materializar tabelas, criar lotes em junção de hash híbrida e assim por diante.

2 - permite que os aplicativos enviem uma consulta sem ter que ler o resultado da consulta enviada anteriormente. Aproveitando o modo pipeline, um cliente irá esperar menos pelo servidor, uma vez que múltiplas consultas / resultados podem ser enviados / recebidos em uma única transação de rede.

3 - O genetic query optimizer só é aplicado se o número de joins for superior a 12, caso contrário é aplicado o algoritmo do caixeiro viajante em que um algoritmo de programação dinâmica vai tentar encontrar o caminho mais barato para a execução da query. Caso seja necessário user o genetic query optimizer, este começa por gerar randoms joins e faz esta geração random de acordo com um número pré definido nas definições e usa subjoins de trees para calcular o caminho mais barato.

Select \* from team;

Explain Select \* from player;

EXPLAIN SELECT \* FROM player AS a, team AS b WHERE a.id\_team = b.id;