Feita para Desenvolvedores de Software e DBAs

SQL magazine

Ano 7 :: Edição 88 :: R$ 13,90

DevMedia

ISSN 1677918-5

MySQL
Modelando um banco de dados para um sistema de gerência de campeonato de futebol

Dia a Dia
MySQL Benchmark: Linux x Windows

Oracle
Boas práticas com Oracle RAC

PostgreSQL na prática

Trabalhando com Full Text Search

Construindo stored procedures

Brinde :: Compre esta edição e ganhe 2 vídeo-aulas
- Desenvolvendo uma Aplicação Web Completa para Iniciantes - Partes 5 e 6

Banco de Dados
Questões de banco de dados do concurso do Ministério Público (2010) – Parte 1

Oracle
Desvendando o Automatic Storage Management - Parte 13

Introdução aos comandos vacuum, analyze, explain e count

Jim Nasby

Resumo DevMan

De que se trata o artigo:

Este artigo apresenta a importância e o uso dos comandos vacuum, analyze, explain e count. Ao final do artigo você entenderá a importância de cada um deles.

Para que serve:

Ao fazer a leitura deste artigo, você entenderá como estes comandos impactam ou facilitam a análise de questões associadas a desempenho no PostgreSQL.

Em que situação o tema é útil:

No dia a dia da administração de bancos de dados PostgreSQL.

Introdução aos comandos vacuum, analyze, explain e count:

Um componente chave de qualquer banco de dados é atender as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Em resumo, as regras ACID são o que protege os dados de um banco e dados, mantendo o controle de transações. Se um banco de dados não possui um mecanismo para atender as propriedades ACID, não existe nada para garantir que seus dados estejam seguros contra mudanças aleatórias que possam ocorrer. Neste sentido, este artigo apresenta descrições, explicações e exemplos para alguns dos principais comandos usados no PostgreSQL para gerenciamento de bancos de dados (VACUUM, ANALYZE,EXPLAIN e COUNT).

Um componente chave de qualquer banco de dados é atender as propriedades ACID (Atomicidade, Consistência, Isolamento e Durabilidade). Em resumo, as regras ACID são o que protege os dados de um banco e dados, mantendo o controle de transações. Se um banco de dados não possui um mecanismo para atender as propriedades ACID, não existe nada para garantir que seus dados estejam seguros contra mudanças aleatórias que possam ocorrer. Esse é o motive pelo qual todos os SGBDs atualmente atendem às propriedades ACID, ou seja, fazem controle de transações.

Focando um pouco mais na propriedade I (Isolamento), ela garante que múltiplos usuários acessando o mesmo dado obterão o mesmo resultado como se existisse apenas um usuário acessando o dado por vez.

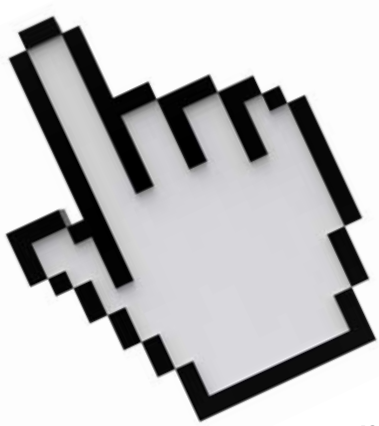
Uma forma simples de garantir isso é não permitindo que qualquer usuário modifique uma parte dos dados se outros usuários estão no momento lendo este dado. Isso garante que o dado não mudará até que todas as leituras em andamento sejam finalizadas. Isso é feito usando um "bloqueio de leitura", e é como a maioria dos bancos de dados trabalha. No entanto, bloqueio de leitura possui algumas sérias desvantagens.

Imagine um banco de dados que está sendo usado em um Web site. A maioria das páginas no site fará ao menos uma consulta ao banco de dados, e muitas páginas irão fazer várias consultas. É claro, existem outras páginas que também modificarão dados.

Agora lembre que para cada linha que é lida do banco de dados, um bloqueio de leitura deve ser adquirido. Então toda página irá adquirir muitos bloqueios, algumas vezes centenas deles. A todo momento que um bloqueio é adquirido ou liberado, o banco de dados não está processando seus dados; ele está "reclamando" da existência dos bloqueios.

E o que dizer das páginas que atualizam dados? Não podemos atualizar nada que está sendo lido, da mesma forma que qualquer coisa que está sendo atualizada não pode ser lida. Quando alguém quer atualizar dados, eles precisam aguardar até que todos aqueles que estão lendo o dado em questão finalizem suas leituras. Enquanto isso, para garantir que a pessoa que deseja atualizar esteja apta a eventualmente fazer essa operação, novas consultas que querem ler este dado são bloqueadas até que a atualização aconteça. Portanto, tendo apenas uma query que quer fazer uma atualização, temos uma grande quantidade de pessoas que estão aguardando a atualização concluir, e a atualização está aguardando a todas as leituras em andamento serem finalizadas.

Considere a mesma situação com um mecanismo de controle de concorrência com múltiplas versões (MVCC – *Multi Version Concurrency Control*). Nenhuma dessas consultas que estão lendo dados precisa adquirir qualquer bloqueio. Assim, o banco de dados não precisa mais



se preocupar, então ele pode gastar mais tempo cuidando dos dados (que é o que queremos de fato que um banco de dados faça por nós). Mais importante, a *query* de atualização não precisa aguardar por nenhuma *query* de leitura, ela pode executar imediatamente, e as *queries* de leitura não precisam aguardar pelas *queries* de atualização. Em vez de termos várias *queries* aguardando pelo fim de outras *queries*, seu web site estaria sempre disponível para executar uma operação.

É claro que o MVCC não vem sem uma desvantagem. O "MV" no MVCC é para Multi-versão. Isso significa que múltiplas versões do mesmo dado serão mantidas sempre que os dados mudarem. O Oracle faz isso colocando dados antigos em um "*undo log*" (log de operações desfeitas). PostgreSQL não usa um *undo log*, em vez disso ele mantém múltiplas versões do dado em tabelas bases. Isso significa que existe muito menos sobrecarga ao fazer atualizações, e devemos ocasionalmente remover as versões antigas dos dados. Isso é uma das tarefas que o comando VACUUM realiza.

O PostgreSQL gerencia essas múltiplas versões armazenando algumas informações extras em todas as linhas. Esta informação é usada para determinar qual transação deve estar apta para ver a linha. Se a linha é uma versão antiga, existe uma informação que diz ao PostgreSQL como encontrar essa nova versão da linha. Esta informação é precisa e está disponível para bloquear linhas durante uma atualização.

Considere este cenário: uma linha é inserida em uma tabela que possui vários índices, e esta transação é comitada. Várias atualizações acontecem nesta linha. Cada atualização irá criar uma nova linha em todos os índices, mesmo se a chave do índice não mudou. E cada atualização irá também deixar uma versão antiga da linha na tabela base, uma que foi atualizada para apontar para a localização da nova versão da linha que a substituiu. Todos os dados antigos serão mantidos até que o *vacuum* seja executado na tabela. Em um sistema complexo, não leva muito tempo para todos os dados antigos serem traduzidos em muitos espaços desperdiçados. E é muito difícil recuperar este espaço se ele cresce até um nível inaceitável.

Isso significa que para aqueles que desejam manter seus bancos de dados no PostgreSQL com bom desempenho, esta operação de *vacuum* é crítica. Isso é especialmente verdade naquelas tabelas que possuem alta carga de atualização/inserção/exclusão. Estas tabelas geralmente devem receber operações de *vacuum* frequentemente se elas são pequenas. Para cargas mais moderadas, o *autovacuum* irá normalmente fazer um bom trabalho de manter espaços "mortos" em um nível mínimo. Você pode e deve ajustar o *autovacuum* para manter apropriadamente tais tabelas, além de manualmente aplicar *vacuum* nelas.

O PostgreSQL lembra o que foi feito no vacuum?

Quando o banco de dados precisa adicionar novos dados a uma tabela como resultado de um INSERT ou UPDATE, ele precisa encontrar algum lugar para armazenar estes dados. Existem três formas de fazer isso:

1. Escanear toda a tabela em busca de espaço livre;
2. Apenas adicionar a informação no final da tabela;
3. Lembrar quais páginas na tabela possuem espaço livre disponível, e usar uma delas.

A opção 1 seria obviamente extremamente lenta. Imagine a leitura de toda a tabela toda vez que você precisar adicionar ou atualizar um dado! A opção 2 é rápida, mas ela resultaria no crescimento de tamanho da tabela cada vez que adicionarmos uma linha. Isso nos leva à opção 3, que é onde o FSM surge. O FSM é onde o PostgreSQL mantém rastros das páginas que possuem espaços livres disponíveis para uso. Toda vez que ele precisa de espaço em uma tabela ele procurará primeiramente no FSM; se ele não encontrar qualquer espaço livre para a tabela, ele desistirá e adicionará a informação no fim da tabela.

O que tudo isso significa na vida real? A única forma de páginas serem colocadas no FSM é através de um VACUUM. Mas o FSM é limitado em tamanho, então para cada tabela é permitida apenas certa capacidade para armazenar informações sobre páginas que possuem espaço livre. Se uma tabela possui mais páginas com espaço livre que a capacidade do FSM, as páginas com a menor quantidade de espeço livre não são armazenadas totalmente. Isso significa que o espaço nessas páginas não será usado até pelo menos a próxima vez que for feita uma operação de *vacuum* na tabela.

O resultado disso é que em um banco de dados com muitas páginas com espaços livres (tais como um banco de dados que está há muito tempo sem ser feito um *vacuum*) terão dificuldades em reusar espaços livres.

Felizmente, existe uma forma simples de estimar quanto espaço livre é necessário: VACUUM VERBOSE. Sempre que um VACUUM VERBOSE é executado em um banco de dados completo (vacuumdb –av), as duas últimas linhas retornadas contêm informações sobre a utilização do FSM, como exibido na **Listagem 1**.

A primeira linha indica que existem 81 relações no FSM e que essas 81 relações armazenam 235349 páginas com espaços livres nelas. O banco de dados estima que 220672 *slots* (vagas) são necessários no FSM.

Listagem 1. Exemplo de retorno do VACUUM VERBOSE

```
INFO: free space map: 81 relations, 235349 pages stored; 220672 total pages  needed
DETAIL: Allocated FSM size: 1000 relations + 2000000 pages = 11817 kB shared memory.
```

Listagem 2. Exemplo de retorno do comando EXPLAIN

```
decibel=# explain select * from customer;
QUERY PLAN
---------------------------------------------------------------
Seq Scan on customer (cost=0.00..12.50 rows=250 width=287)
(1 row)
```

A segunda linha mostra as configurações atuais do FSM. Esta instalação do PostgreSQL está configurada para rastrear 1000 relações (*max_fsm_relations*) com um total de 2000000 páginas livres (*max_fsm_pages*).

Note que esta informação não será precisa se existir um alto número de banco de dados na instalação do PostgreSQL e você apenas realizou *vacuum* em um deles. É melhor realizar *vacuum* na instalação completa.

Configurando o mapa de espaço livre (a partir da versão 8.3 do PostgreSQL)

A melhor forma de garantir que temos páginas FSM suficientes é periodicamente aplicar *vacuum* à instalação completa usando o comando *vacuum -av* e olhando em duas linhas no resultado gerado. Você quer garantir que *max_fsm_relations* é ao menos tão grande quanto o total de "páginas armazenadas" ou "total de páginas necessárias".

O que é normalmente mais crítico que *max_fsm_pages* é *max_fsm_relations*. Se a instalação possui mais relações que *max_fsm_relations* (e isso inclui tabelas temporárias), algumas relações não terão qualquer informação armazenada no FSM. Isso pode ainda incluir relações que possuem grande quantidade de espaço disponível. Então, é importante garantir que *max_fsm_relations* é sempre maior que VACUUM VERBOSE reporta e incluir alguma folga. Novamente, a melhor forma de garantir isso é monitorando os resultados das execuções periódicas do *vacuum verbose*.

Usando ANALYZE para otimizar queries no PostgreSQL

A operação de *vacuum* não a única manutenção periódica que seu banco de dados precisa. Você também precisa analisar o banco de dados, observando o que o *query planner* (planejador de consultas), que possui estatísticas sobre as tabelas que ele usa, está decidindo sobre como executar uma *query*. Simplesmente faça: garanta que você está executando o comando ANALYZE frequentemente de forma suficiente, de preferência através do *autovacuum*. E aumente o parâmetro *default_statistics_target* (no arquivo *postgresql.conf*) para 100.

O PostgreSQL possui um otimizador de *query* muito complexo. Dependendo de como você deseja contar, existem dezenas de diferentes possibilidades que podem ser seguidas ao executar uma *query*, e se a *query* faz junção com várias tabelas, podem existir centenas ou até milhares de formas diferentes de processar essas junções. Colocando tudo junto, não é difícil chegarmos a milhares de diferentes possibilidades de executar uma única *query*.

Então, como o *planner* (planejador) determina a melhor forma de executar uma *query*? Cada um desses diferentes "blocos de construções" (que tecnicamente são chamados de nós de *query*) possui uma função associada que gera um custo. Isso é o que você vê quando executa um comando EXPLAIN, como exibido na **Listagem 2**.

Sem detalhar muito sobre como ler a saída gerada pelo comando EXPLAIN, o PostgreSQL está estimando que esta consulta retornará 250 linhas, cada uma precisando de 287 bytes em média. O custo de obter a primeira linha é 0 (na verdade não é, é apenas um número pequeno que é arredondado para 0), e obter o conjunto de resultados completo possui um custo de 12.50. Tecnicamente, a unidade de custo é "o custo de ler uma única página do banco de dados do disco", mas na verdade a unidade é bem arbitrária. Ela na verdade não se relaciona a nada que você possa medir. Apenas pense no custo em termos de "unidades de trabalho"; então executar esta *query* teria o custo de "12.5 unidades de trabalho".

Como o banco de dados chegou ao custo de 12.5? O *planner* chamou a função de estimar custo para um *Seq Scan*. Esta função então olhou um conjunto de informações estatísticas sobre a tabela "cliente" e usou essa informação para produzir uma estimativa de quanto trabalho ele levaria para executar a *query*. Agora vamos ao que interessa: estatísticas de tabela!

O PostgreSQL mantém dois conjuntos diferentes de estatísticas sobre tabelas. O primeiro conjunto lida com quão grande a tabela é. Esta informação é armazenada na tabela de sistema *pg_class*. O campo "*relpages*" é o número de páginas do banco de dados que estão sendo usadas para armazenar a tabela, e o campo "*reltuples*" é o número de linhas na tabela. O valor de *reltuples/relpages* é o número médio de linhas em uma página, o que é um número importante para o *planner* conhecer. Tipicamente uma *query* estará lendo uma parte pequena da tabela, retornando um número limitado de linhas. Como todas as operações de IO são feitas no nível de páginas, quanto mais linhas estão em uma página, menos páginas o banco de dados terá de ler para pegar todas as linhas necessárias.

O outro conjunto de estatísticas que o PostgreSQL mantém lida diretamente com a questão de quantas linhas uma *query* irá retornar. Essas estatísticas são mantidas em uma base campo a campo. Para ver esta ideia em ação, vamos consultar um conjunto limitado de linhas, conforme **Listagem 3**.

Agora o *planner* pensa que iremos obter uma única linha. Ele estima isso olhando para *pg_stats.histogram_bounds*, que é um vetor de valores. Cada valor define o início de um novo "*bucket*," onde cada *bucket* possui aproximadamente o mesmo tamanho. Por exemplo, se temos uma tabela que contém os números 1 até 10 e temos um histograma que divide os dados em 2 buckets (duas partes), *pg_stats.histogram_bounds* seria {1,5,10}. Isso diz ao *planner* que existem tantas linhas na tabela onde o valor era entre 1 e 5 quanto linhas onde o valor é entre 5 e 10.

Se o *planner* usa esta informação em combinação com *pg_class.reltuples*, ele pode estimar quantas linhas serão retornadas. Neste caso, se fazemos *SELECT * FROM table WHERE value <= 5* o *planner* verá que existem tantas linhas com valores menores ou iguais a 5 como linhas com valores maiores que 5 o que significa que a *query* irá retornar a metade das linhas da tabela.

Existem 10 linhas na *pg_class.reltuples*, então fazendo uma matemática simples podemos estimar que 5 linhas serão retornadas. Ele pode então olhar para o número de linhas em cada página e decidir quantas páginas ele terá de ler. Finalmente, com todas essas informações, ele pode estimar

quantas unidades de trabalho serão necessárias para executar a *query*. É claro que é mais complicado que parece. Por exemplo, se temos que executar a query *SELECT * FROM table WHERE value <= 3*, o *planner* agora precisará estimar quantas linhas seriam retornadas por interpolação dos dados do histograma.

Um problema com esta estratégia é que se existem alguns valores que são extremamente comuns, eles podem complicar tudo. Por exemplo, considere este histograma: {1,100,101}. Existem tanto valores entre 100 e 101 como existem valores entre 1 e 100. Mas isso significa que temos todos os números entre 1 e 1000? Algum outro? Nós temos um único 1 e vários 50's?

Felizmente, o PostgreSQL possui dois campos de estatística adicionais para ajudar a eliminar este problema: *most_common_vals* e *most_common_freqs*. Como você pode imaginar, esses campos armazenam informações sobre os valores mais comuns encontrados na tabela. O campo *most_common_vals* armazena os valores atuais e *most_common_freqs* armazena quão frequente cada valor aparece, como uma fração do número total de linhas. Então, se *most_common_vals* é {1,2} e *most_common_freqs* é {0.2,0.11}, 20% dos valores na tabela são 1 e 11% são 2.

Mesmo com *most_common_vals*, você pode ainda ter problemas. O padrão é armazenar os 10 valores mais comuns, e 10 *buckets* no histograma. Mas se você tem vários valores diferentes e uma grande variação na distribuição desses valores, é fácil atingir uma sobrecarga nas estatísticas. Felizmente, é fácil aumentar os números de *buckets* do histograma e valores mais comuns armazenados. Existem duas formas de fazer isso. A primeira é através do parâmetro do *postgresql.conf* chamado *default_statistics_target*. Como o único ponto negativo em ter mais estatísticas é mais espaço usado na tabela de catálogos, para mais instalações é recomendado configurar este parâmetro com pelo menos o valor 100, e se você tem um número relativamente pequeno de tabelas, a configuração poderia ser 300 ou mais.

O segundo método é usar ALTER TABLE, ou seja: *ALTER TABLE table_name ALTER column_name SET STATISTICS 1000*. Esse comando sobrescreve o parâmetro *default_statistics_target* para a coluna *column_name* na tabela *table_name*. Se você tem um grande número de tabelas (por exemplo, acima de 100), ter um valor grande de *default_statistics_target* pode resultar no crescimento da tabela de estatísticas para um tamanho grande que pode prejudicar o desempenho do banco de dados. Nestes casos, é melhor manter *default_statistics_target* moderadamente baixo (provavelmente no intervalo entre 50-100), e manualmente aumentar o espaço de estatísticas nas tabelas maiores do banco de dados. Note que estatísticas em um campo são usadas apenas quando este campo é parte de uma cláusula WHERE, então não existe razão para aumentar as estatísticas nos campos que nunca são pesquisados.

Isso é suficiente sobre histogramas e valores comuns. Existe uma estatística final que lida com a probabilidade de encontrar um dado valor na tabela, é a *n_distinct*. Se este número é positivo, ele é uma estimativa de quantos valores distintos estão na tabela. Se ele é negativo, ele é a razão dos valores distintos pelo número total e linhas. A forma negativa é usada quando ANALYZE pensa que o número de valores distintos irá variar com o tamanho da tabela. Então, se cada valor em um campo é único, *n_distinct* será -1.

A correlação é uma medida de similaridade de ordem da linha na tabela com a ordem do campo. Se você escanear a tabela sequencialmente e o valor no campo aumenta a cada linha, a correlação é 1. Se pelo contrário, cada campo é menor que o anterior, a correlação é -1. Correlação é um fator chave na seleção de um escaneamento de índice, pois a correlação próximo de 1 ou -1 significa que um escaneamento de índice não terá muitos saltos ao longo da tabela.

Finalmente, *avg_width* é a largura média dos dados em um campo e *null_frac* é a fração de linhas na tabela onde o campo será NULL.

Como podemos ver, vários trabalhos têm sido feitos para manter informações suficientes para que o *planner* possa fazer boas escolhas ao executar *queries*. Mas todo este framework não funciona bem se as estatísticas não são atualizadas, ou no pior caso, não são coletadas totalmente. Lembra quando o *planner* decidiu que selecionar todos os clientes no Texas retornaria 1 linha (resultado na **Listagem 3**)? Isso foi feito antes da tabela ser analisada. Vamos ver o que realmente ocorre ao fazer esta consulta na **Listagem 4**.

Não apenas a estimativa do número de linhas foi imprecisa (2048 foram retornadas, e não 1), ela foi imprecisa o suficiente para mudar o plano de execução para esta *query*. Este é um exemplo do porquê é tão importante manter estatísticas atualizadas. Como foi dito no início do artigo, a melhor forma de fazer isso é usando o *autovacuum*.

Este é obviamente um tópico muito complexo. Mais informações sobre estatísticas podem ser encontradas em http://www.postgresql.org/docs/current/static/planner-stats-details.html.

O comando EXPLAIN

Agora que você sabe a importância de dar ao *query planner* estatísticas atualizadas, então ele poderá planejar a melhor forma de executar uma *query*. Mas como saber como o PostgreSQL está de fato executando uma *query*? É aí que surge o comando EXPLAIN.

Vamos olhar na **Listagem 5** um exemplo simples e então entender o significado das partes que o compõe.

Listagem 3. Consultando as estatísticas de linhas retornadas no PostgreSQL

```
decibel=# explain select * from customer where state='TX';
QUERY PLAN
------------------------------------------------------------------------Index Scan using customer__state on customer
(cost=0.00..5.96 rows=1 width=287)
Index Cond: (state = 'TX'::bpchar)
(2 rows)
```

Ele nos diz que o otimizador decidiu usar um scan sequencial para executar a *query*. Ele estima que isso custará 0.00 para retornar a primeira linha, e que custará 60.48 para retornar todas as linhas. Ele pensa que serão retornadas 2048 linhas, e que a média de tamanho de cada linha será 107 bytes.

Mas EXPLAIN de fato não executa a *query*. Se você quer ver quão próxima da realidade é a estimativa, você precisa usar o comando EXPLAIN ANALYZE, como na **Listagem 6**.

Note que nós agora temos um segundo conjunto de informações; o tempo real necessário para executar o scan sequencial, o número de linhas retornadas por este passo, e o número de vezes que essas linhas foram acessadas em *loop* (falaremos mais sobre isso a seguir). Também temos um tempo de execução total para a *query*.

Um leitor observador irá notar que os números do tempo real não batem exatamente com os custos estimados. Isso não ocorre de fato porque a estimativa falhou; isso é porque a estimativa não é medida em tempo, mas sim em uma unidade arbitrária, como já citado.

É claro que não existe exatamente muito a ser analisado em "*SELECT * FROM table*" da **Listagem 6**, então vamos tentar algo um pouco mais interessante, como apresentado na **Listagem 7**.

Agora vemos que o plano da consulta (*query plan*) inclui dois passos, uma ordenação e um scan sequencial. Apesar de parecer pouco intuitivo, os dados fluem dos passos mais baixos no plano para os mais altos, então a saída do scan sequencial é usado como entrada para o operador de ordenação.

Se olharmos para o passo de ordenação, notaremos que ele está nos dizendo o que ele está ordenando (a "Chave de Ordenação"). Muitos passos da *query* imprimirão informações adicionais como esta. Algo a mais a ser notado é que o custo para retornar a primeira linha a partir do operador de ordenação é muito alto, quase o mesmo custo de retornar todas as linhas. Isso ocorre porque uma ordenação não pode retornar nenhuma linha até que os dados estejam de fato ordenados, que é o que leva mais tempo.

Ainda podem existem múltiplos passos de *query*, o custo reportado em cada passo inclui não apenas o custo para realizar este passo, mas também o custo para realizar todos os passos abaixo dele. Então, neste exemplo, o custo real da operação de ordenação é 173.12 – 60.48 para a primeira linha, ou 178.24 – 60.48 para todas as linhas. Porque estamos subtraindo 60.48 a partir dos custos da primeira linha e de todas as linhas? Porque a operação de ordenação precisa obter todos os dados a partir do scan sequencial antes que ele possa retornar qualquer dado ordenado. Em geral, sempre que você vê um passo com custo de primeira e todas as linhas com valores muito próximos, esta operação requer todos os dados a partir de todos os passos precedentes. Vamos olhar algo mais interessante na **Listagem 8**.

Agora temos algo a ser destacado! Note como existem algumas endentações no resultado. Endentação é usada para mostrar quais passos de uma *query* estão dentro de outros passos. Aqui podemos ver que o HASH JOIN é formado por um scan sequencial e uma operação de hash. Esta operação de hash é, por sua vez, formada por outro scan sequencial. Note que a operação de hash possui o mesmo custo para retornar a primeira e todas as linhas; ela precisa de todas as linhas antes que ela possa retornar qualquer linha. Isso se torna interessante neste plano quando olhamos para o *hash join*: o custo da primeira linha reflete o custo total das linhas para o hash, mas ele reflete o custo da primeira linha de 0.00 para o scan sequencial na tabela *customer*. Isso ocorre porque um *hash join* pode iniciar retornando linhas assim que ele obtém a primeira linha a partir de ambas as suas entradas.

Tínhamos prometido voltar a falar sobre o que significa loops, então temos um exemplo na **Listagem 9**.

Listagem 4. Analisando as estatísticas de uma consulta no PostgreSQL

```
decibel=# analyze customer;
ANALYZE
decibel=# explain select * from customer where state='TX';
QUERY PLAN
------------------------------------------------------------------------------
Seq Scan on customer (cost=0.00..65.60 rows=2048 width=107)
Filter: (state = 'TX'::bpchar)
(2 rows)
```

Listagem 5. Exemplo do uso do EXPLAIN

```
EXPLAIN SELECT * FROM CUSTOMER;
QUERY PLAN
---------------------------------------------------------------
Seq Scan on customer  (cost=0.00..60.48 rows=2048 width=107)
(1 row)
```

Listagem 6. Exemplo do uso do EXPLAIN ANALYZE

```
EXPLAIN ANALYZE SELECT * FROM CUSTOMER;
QUERY PLAN
------------------------------------------------------------------- Seq Scan on customer
(cost=0.00..60.48 rows=2048 width=107) (actual time=0.033..6.577 rows=2048 loops=1)
 Total runtime: 7.924 ms
(2 rows)
```

Listagem 7. Exemplo mais complexo do uso do EXPLAIN ANALYZE

```
EXPLAIN SELECT * FROM customer ORDER BY city;
QUERY PLAN
------------------------------------------------------------------- Sort (cost=173.12..178.24 rows=2048 width=107)
 Sort Key: city
 -> Seq Scan on customer (cost=0.00..60.48 rows=2048 width=107)
```

Um loop aninhado é algo que deve ser familiar para codificadores de linguagem procedural; ele funciona como mostrado na **Listagem 10**.

Então, se existem 4 linhas em *input_a*, *input_b* será lida em sua totalidade 5 vezes. De outra forma, será feito um loop nele por 4 vezes. Isso é o que a seção do plano da consulta está exibindo na **Listagem 9**. Se fizermos a conta, veremos que 0.055 * 4 conta para a maioria da diferença entre o tempo total do *hash join* e o tempo total do loop aninhado (o que sobrou é provavelmente as sobras desta medição).

Então, o que é isso na "vida real"? Tipicamente, se você está executando EXPLAIN em uma *query* é porque você está tentando melhorar o desempenho dela. A chave para isso é identificar o passo que está tomando a maior quantidade de tempo e ver o que você pode fazer a respeito disso. Vamos olhar para um novo exemplo na **Listagem 11** e identificar qual é o "passo problema".

O loop aninhado possui a maioria do custo, com um tempo de execução de 20.035ms. Este loop aninhado está também obtendo dados a partir de um loop aninhado e um scan sequencial, e novamente o loop aninhado é onde a maioria do custo foi identificada (com 19.481 ms de tempo total).

Assim, nosso "caminho caro" é algo como o que está apresentado na **Listagem 12**.

Neste exemplo, todos esses passos aparecem juntos na saída, mas nem sempre isso ocorre.

O nó de loop aninhado de nível mais baixo obtém dados a partir do seguinte caminho, apresentado na **Listagem 13**.

Aqui podemos ver que o *hash join* possui boa parte do tempo de execução. Ele está sendo obtido a partir de um scan sequencial e um *hash*. Este *hash*, destacado na **Listagem 14**, possui a maioria do tempo de execução.

Finalmente, trabalharemos com a parte mais custosa da *query*: o *scan index* em *pg_class_relname_nsp_index*. Infelizmente para nós, será virtualmente impossível acelerar um scan index que apenas lê uma única linha. Mas também note que ele apenas leva 18.464 ms; ele dificilmente será o que estamos procurando para melhorar desempenho.

Um aspecto final a ser notado: a medição da sobra do EXPLAIN ANALYZE não é trivial. Em casos extremos ele pode tomar 30% ou mais do tempo de execução da *query*. Apenas lembre que EXPLAIN é uma ferramenta para medir desempenho relativo, e não desempenho absoluto.

Infelizmente, EXPLAIN é algo que é pouco documentado no manual do PostgreSQL. Este artigo serve como um ponto de apoio sobre esta ferramenta importante.

Listagem 8. Exemplo mais detalhado do uso do EXPLAIN ANALYZE

```
EXPLAIN ANALYZE SELECT * FROM customer JOIN contact USING (last_name);
QUERY PLAN
------------------------------------------------------------------------------ Hash Join (cost=1.02..92.23
rows=2048 width=351) (actual time=1.366..58.684 rows=4096 loops=1)
  Hash Cond: ((“outer”.last_name)::text = (“inner”.last_name)::text)
  -> Seq Scan on customer (cost=0.00..60.48 rows=2048 width=107)
     (actual time=0.079..21.658 rows=2048 loops=1)
  -> Hash (cost=1.02..1.02 rows=2 width=287) (actual time=0.146..0.146 rows=2 loops=1)
       ->  Seq Scan on contact  (cost=0.00..1.02 rows=2 width=287)
           (actual time=0.074..0.088 rows=2 loops=1)
 Total runtime: 62.233 ms
```

Listagem 9. Exemplo de loop no acesso às linhas

```
-> Nested Loop (cost=5.64..14.71 rows=1 width=140) (actual time=18.983..19.481
    rows=4 loops=1)
      -> Hash Join (cost=5.64..8.82 rows=1 width=72) (actual time=18.876..19.212
         rows=4 loops=1)
      -> Index Scan using pg_class_oid_index on pg_class i  (cost=0.00..5.88 rows=1 width=72)
      (actual time=0.051..0.055 rows=1 loops=4)
```

Listagem 10. Exemplo paralelo do que seria um loop

```
for each row in input_a
   for each row in input_b
      do something
   next
next
```

Listagem 11. Identificando o passo problema em um loop aninhado

```
EXPLAIN ANALYZE SELECT * FROM pg_index WHERE tablename='pg_constraint';
QUERY PLAN
------------------------------------------------------------------------------Nested Loop Left Join (cost=5.64..16.89
rows=1 width=260) (actual time=19.552..20.530 rows=4 loops=1)
  Join Filter: (“inner”.oid = “outer”.reltablespace)
  -> Nested Loop Left Join  (cost=5.64..15.84 rows=1 width=200) (actual time=19.313..20.035
     rows=4 loops=1)
    Join Filter: (“inner”.oid = “outer”.relnamespace)
    -> Nested Loop  (cost=5.64..14.71 rows=1 width=140) (actual time=18.983..19.481 rows=4
       loops=1)
      -> Hash Join  (cost=5.64..8.82 rows=1 width=72) (actual time=18.876..19.212 rows=4 loops=1)
         Hash Cond: (“outer”.indrelid = “inner”.oid)
         ->  Seq Scan on pg_index x  (cost=0.00..2.78 rows=78 width=8)
            (actual time=0.037..0.296 rows=80 loops=1)
         ->  Hash  (cost=5.63..5.63 rows=1 width=72)
            (actual time=18.577..18.577 rows=1 loops=1)
            -> Index Scan using pg_class_relname_nsp_index on pg_class c
               (cost=0.00..5.63 rows=1 width=72)
               (actual time=18.391..18.464 rows=1 loops=1)
               Index Cond: (relname = ‘pg_constraint’::name)
               Filter: (relkind = ‘r’::”char”)
      -> Index Scan using pg_class_oid_index on pg_class i  (cost=0.00..5.88 rows=1 width=72)
         (actual time=0.051..0.055 rows=1 loops=4)
         Index Cond: (i.oid = “outer”.indexrelid)
         Filter: (relkind = ‘i’::”char”)
    ->  Seq Scan on pg_namespace n  (cost=0.00..1.06 rows=6 width=68)
      (actual time=0.014..0.045 rows=6 loops=4)
  ->  Seq Scan on pg_tablespace t  (cost=0.00..1.02 rows=2 width=68)
     (actual time=0.010..0.018 rows=2 loops=4)
Total runtime: 65.294 ms
```

Agregações – porque min(), max() e count() são tão lentos?

Um ponto comum de reclamação do PostgreSQL é a velocidade de suas agregações. As pessoas normalmente se perguntam por que *count(*)* ou *min/max* são mais lentos que em alguns outros bancos de dados. Existem realmente dois problemas aqui, um simples de resolver e outro nem tanto.

O uso de ORDER BY / LIMIT

Antes da versão 8.1, o *query planner* não sabia que você podia usar um índice para executar um *min* ou *max*, então ele sempre faz um scan na tabela. Felizmente, você pode contornar isso fazendo o que está apresentado na **Listagem 15**.

É claro que isso é um pouco de forçar a barra, então na versão 8.1 o *planner* foi modificado para que ele faça essa substituição em tempo de execução. Infelizmente, ele não é perfeito; foi descoberto que *SELECT max()* em um campo com vários valores NULL levará muito tempo, mesmo se estivermos usando um índice neste campo. Se você tentar usar o *ORDER BY / LIMIT*, é igualmente lento. Suspeita-se que isso ocorre porque o banco de precisa fazer um scan no passado de todos os valores

Listagem 12. Identificando o passo problema no loop da listagem 11

```
Nested Loop Left Join (cost=5.64..16.89 rows=1 width=260) (actual time=19.552..20.530
 rows=4 loops=1)
      Join Filter: ("inner".oid = "outer".reltablespace)
     -> Nested Loop Left Join  (cost=5.64..15.84 rows=1 width=200) (actual time=19.313..20.035
     rows=4 loops=1)
     Join Filter: ("inner".oid = "outer".relnamespace)
     -> Nested Loop (cost=5.64..14.71 rows=1 width=140) (actual time=18.983..19.481
        rows=4 loops=1)
```

Listagem 13. Identificando a origem dos dados para o loop problema

```
-> Hash Join (cost=5.64..8.82 rows=1 width=72) (actual time=18.876..19.212 rows=4 loops=1)
    Hash Cond: ("outer".indrelid = "inner".oid)
     -> Seq Scan on pg_index x  (cost=0.00..2.78 rows=78 width=8)
      (actual time=0.037..0.296 rows=80 loops=1)
       -> Hash (cost=5.63..5.63 rows=1 width=72) (actual time=18.577..18.577 rows=1 loops=1)
        -> Index Scan using pg_class_relname_nsp_index on pg_class
          (cost=0.00..5.63 rows=1 width=72) (actual time=18.391..18.464 rows=1 loops=1)
          Index Cond: (relname = 'pg_constraint'::name)
          Filter: (relkind = 'r'::"char")
   -> Index Scan using pg_class_oid_index on pg_class i  (cost=0.00..5.88 rows=1 width=72)
     (actual time=0.051..0.055 rows=1 loops=4)
     Index Cond: (i.oid = "outer".indexrelid)
     Filter: (relkind = 'i'::"char")
```

Listagem 14. Identificando o caminho problema a partir do caminho da Listagem 13

```
-> Hash  (cost=5.63..5.63 rows=1 width=72) (actual time=18.577..18.577 rows=1 loops=1)
      ->  Index Scan using pg_class_relname_nsp_index on pg_class c
        (cost=0.00..5.63 rows=1 width=72) (actual time=18.391..18.464 rows=1 loops=1)
        Index Cond: (relname = 'pg_constraint'::name)
        Filter: (relkind = 'r'::"char")
```

Listagem 15. Contornando o problema com min/max

```
-- Encontrar o valor mínimo para o campo
SELECT campo FROM tabela WHERE campo IS NOT NULL ORDER BY campo ASC LIMIT 1;

-- Encontrar o valor máximo para o campo
SELECT campo FROM tabela WHERE campo IS NOT NULL ORDER BY campo DESC LIMIT 1;
```

Listagem 16. Exemplo de uso inapropriado do count(*)

```
SELECT count(*) INTO variavel FROM tabela WHERE condicao;
IF variavel > 0 THEN
...
END IF;
```

Listagem 17. Forma apropriada de verificar a existência de uma linha

```
IF EXISTS( SELECT * FROM tabela WHERE condicao ) THEN
```

Listagem 18. Segunda forma apropriada de verificar a existência de uma linha

```
SELECT 1 INTO dummy WHERE EXISTS( SELECT * FROM tabela WHERE condicao );
IF FOUND THEN
...
END IF;
```

Listagem 19. Estimando a quantidade de linhas em uma tabela

```
SELECT reltuples FROM pg_class WHERE oid = 'schemaname.tablename'::regclass::oid;
```

cobertura para fazer o scan (note que isso ocorre porque *count(*)* é muito melhor que *count(algum_campo)*, se você não se importa se valores NULL de *algum_campo* são contados). Como índices normalmente cabem na memória, isso significa que *count(*)* é normalmente muito rápido.

Mas como foi mencionado, o PostgreSQL deve ler a tabela base sempre que ele ler a partir de um índice. Isso significa que, não importa como, *SELECT count(*) FROM tabela*; deve ler a tabela inteira. Felizmente, isso tem sido resolvido nas novas versões do PostgreSQL.

Mesmo com as melhorias de desempenho nas novas versões do PostgreSQL, existem algumas formas que você pode melhorar o desempenho se estiver usando *count(*)*. A chave é considerar primeiramente o porquê você está usando *count(*)*. Você realmente precisa de uma conta exata? Existem vários casos onde ela não é necessária, mas tem sido usada. Talvez o pior uso é como uma forma de verificar se uma linha particular existe, como exemplificado na **Listagem 16**.

Não existe razão para precisarmos de uma conta exata aqui. Em vez disso, poderíamos tentar algo como a **Listagem 17**.

Ou, se você está usando uma linguagem externa, poderia ser algo como a **Listagem 18**.

Note que neste exemplo você obterá uma linha de retorno ou nenhuma.

Por outro lado, talvez você esteja trabalhando em algo onde de fato é preciso fazer uso de uma contagem. Neste caso, considere usar uma estimativa. Google é um exemplo perfeito disso.

Como obter estimativa para *count(*)*? Se você apenas deseja saber o número aproximado de linhas em uma tabela você pode simplesmente selecionar a partir de *pg_class*, conforme **Listagem 19**.

O número retornado é uma estimativa do número de linhas na tabela obtido no último ANALYZE.

Se você quer uma estimativa do número de linhas que serão retornadas a partir de uma consulta arbitrária, você infelizmente precisa traduzir a saída do *explain*.

Finalmente, se você precisa de uma conta exata e desempenho é um ponto importante, você pode construir uma tabela de resumo que contém o número de linhas de uma tabela. A forma mais simples é criar uma *trigger* que irá atualizar a tabela de resumo cada vez que linhas são inseridas ou excluídas. O ponto negativo desta abordagem é que ela força uma atualização na tabela resumo para todas as inserções e exclusões em uma tabela que está sendo mantida a contagem. Isso ocorre porque apenas uma transação pode atualizar a linha apropriada na tabela que conta as linhas por vez.

Conclusões

Neste artigo foram apresentados descrições, explicações e exemplos para alguns dos principais comandos usados no PostgreSQL para gerenciamento de bancos de dados (VACUUM, ANALYZE,EXPLAIN e COUNT). Uma série de evoluções nestes comandos tem sido providenciada nos últimos anos, mas entender seu funcionamento básico é sempre importante.

Jim Nasby