Tire sua dúvidaMarcar como concluídoAnotar

[Artigos](https://www.devmedia.com.br/artigos/)[Banco de Dados](https://www.devmedia.com.br/artigos/banco-de-dados)Uma Solução de Auditoria de Modificações no PostgreSQL

[**Esse artigo faz parte da revista SQL Magazine edição 56. Clique aqui para ler todos os artigos desta edição**](https://www.devmedia.com.br/articles/viewcomp.asp?comp=10372)

****

**ta style="MARGIN: 0cm 0cm 0pt">Parte 1 - Auditoria de Comandos DML**

**A segurança em uma base de dados é uma das coisas mais importantes que se deve ter em mente nos dias de hoje. Saber quem modificou o quê, quando, onde, são informações cada vez mais necessárias para gestores de negócios e DBAs.**

**Para contemplar essa necessidade, são utilizadas técnicas de auditoria nas bases de dados que registram todas as informações necessárias para esse objetivo.**

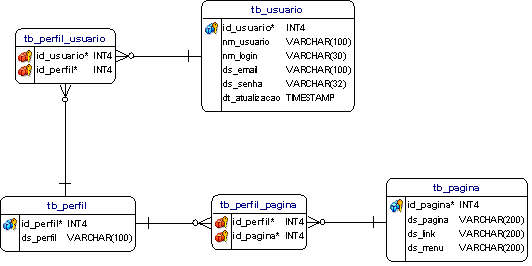
**Este artigo demonstrará uma solução para auditoria de modificação de dados no PostgreSQL. Para isso, construiremos um estudo de caso justificando a necessidade de auditoria e sua posterior implementação. Montaremos uma pequena base onde será implantada toda a estrutura de tabelas de auditoria, *triggers* e funções. Com a estrutura montada, realizaremos uma cópia dos dados que já estão em produção para as tabelas de auditoria, faremos alguns testes e por fim, mostraremos algumas dicas de como manter/consultar toda a estrutura/dados da auditoria. Em nosso próximo artigo mostraremos como fazer uma auditoria de estrutura nas bases de dados do PostgreSQL registrando comandos como CREATE, ALTER e DROP.**

**Os scripts e códigos aqui apresentados necessitam que a versão instalada do PostgreSQL seja a 8.2 ou superior.**

**Apresentação do Estudo de Caso**

**A base de dados que utilizaremos faz parte de um sistema de gerenciamento de acesso, onde utilizaremos apenas algumas tabelas que controlam qual página cada usuário pode acessar em um determinado sistema através dos perfis vinculados.**

**A Figura 1 mostra o diagrama da base de dados com cinco tabelas: tb\_usuario, onde ficarão os usuários do sistema; tb\_perfil, que armazenará os perfis (gerente, funcionário, diretor, etc.); tb\_pagina, onde serão registradas as páginas do sistema; tb\_perfil\_usuario, que associa os perfis aos usuários e tb\_perfil\_pagina, que associa as páginas que cada perfil pode acessar. O script SQL para criação da base está disponível para *download* no site da revista com o nome “script\_criacao\_base\_dados.sql”. Durante a criação da base, foram realizadas algumas inserções para popular a mesma.**

****

**Figura 1. Diagrama ER do banco de dados do gerenciador de acessos**

**Solução para Auditar as Modificações nos Dados**

**Para organizar a base de dados, será criado um novo esquema para conter as tabelas, funções e outros objetos necessários para o funcionamento da auditoria. O comando abaixo criará o esquema auditoria:**

**CREATE SCHEMA auditoria;**

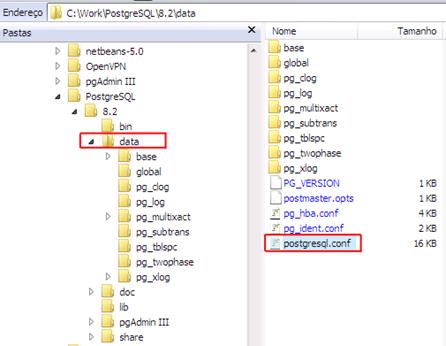
**A criação do esquema é uma boa prática para a organização lógica da base de dados. Se as tabelas de auditoria forem ficar no mesmo esquema das tabelas de produção, é opcional a criação do mesmo.**

**Após esse passo, é necessário incluir na variável *search\_path*, do arquivo postgresql.conf, o nome do novo esquema. O postgresql.conf é um arquivo de configuração do PostgreSQL e normalmente se encontra no diretório “data” do *cluster* de banco de dados utilizado, como pode ser visto na Figura 2**. A variável *search\_path* encontra-se na seção *CLIENT CONNECTION DEFAULTS* do postgresql.conf e pode estar com a configuração *default* mostrada na **Listagem 1, caso não existam outros esquemas criados nas bases de dados. Se existirem outros esquemas, esta variável não estará comentada (ou seja, sem o símbolo “#”) e com a lista dos esquemas separados por vírgula.**

**Caso a variável esteja comentada, devemos retirar o símbolo de comentário (“#”) e incluir o nome do novo esquema. Depois das alterações, a variável *search\_path* ficará como descrita abaixo. Feito isso, basta reiniciar o PostgreSQL.**

**search\_path = '"$user",public,auditoria'**

**Assim, o esquema “auditoria” poderá ser visto no “psql”, cliente *default* do PostgreSQL. Com essa alteração também poderemos ver a lista de tabelas e objetos criados no esquema de auditoria.**

****

**Figura 2. Localização do diretório “data” e do arquivo postgresql.conf no Windows**

**Listagem 1. Variável *search\_path* no postgresql.conf**

**#---------------------------------------------------------------------------**

**# CLIENT CONNECTION DEFAULTS**

**#---------------------------------------------------------------------------**

**# - Statement Behavior -**

**#search\_path = '"$user",public'                                     # schema names**

**Nesta solução de auditoria, cada alteração de dados realizada pelos comandos de INSERT, UPDATE e DELETE deve ser registrada, gerando-se uma coleção de alterações para cada registro modificado. Para isso acontecer é preciso ter a mesma estrutura das tabelas em produção dentro do esquema “auditoria”.**

**Criação da Estrutura de Auditoria**

**Para a criação da estrutura de auditoria devemos realizar uma seqüência de passos, onde o primeiro deles é criar tabelas com estrutura semelhante às de produção. Para isso é necessário realizar pequenas modificações nestas tabelas, como a retirada das chaves estrangeiras, primárias, índices, restrições (*check constraint*, NOT NULL, etc.), valores *default*, comentários, *triggers*, criações de seqüências, atribuições de incrementos das seqüências, etc.**

**Devemos deixar apenas os comandos CREATE TABLE das estruturas das tabelas, tomando cuidado para que a ordem das colunas nas tabelas de auditoria sejam a mesma das tabelas de produção. Ou seja, se na tabela tb\_usuario a ordem das colunas é id\_usuario, nm\_usuario e nm\_login, devemos ter a mesma ordem na tabela de auditoria que irá registrar os dados modificados desta tabela.**

**A explicação para a modificação da estrutura das tabelas é a seguinte: como a auditoria tem a utilidade de rastrear as modificações realizadas sobre os dados, não tem sentido termos as mesmas restrições e regras de uma tabela de produção, como verificação de valores (*check constraint*), integridade referencial, etc. Isso provoca uma carga a mais de verificação nas tabelas criadas para a auditoria durante a inserção/modificação/exclusão dos dados. Em resumo, as tabelas de auditoria devem ter uma estrutura simples para que as inserções sejam realizadas com eficiência e não prejudiquem o desempenho do banco.**

**Para realizar essa limpeza na estrutura das tabelas de auditoria será necessário gerar um *script* da estrutura da base de dados através do utilitário pg\_dump do PostgreSQL utilizando o comando abaixo no *prompt* do DOS. O *script* da estrutura, após a execução do “pg\_dump”,está disponível para *download* no site da revista com o nome “script\_estrutura\_antes.sql”.**

**pg\_dump -U postgres sql\_magazine\_auditoria -sOx -n public > C:\Temp\script\_estrutura\_antes.sql**

**Observe que o resultado desse comando é um script completo com toda a estrutura da base de dados. A partir dele é que iremos modificar a estrutura das tabelas para criar as tabelas de auditoria. Depois de realizar a limpeza do *script* é interessante incluir nos nomes das tabelas algum prefixo para diferenciar as tabelas da auditoria das tabelas em uso pelo sistema. Em nosso exemplo, usamos o prefixo “tb\_audit\_”. Também devemos incluir o nome do esquema nos nomes das tabelas, por exemplo, auditoria.tb\_audit\_tb\_pagina.**

**Para uma auditoria ser eficaz, não basta apenas sabermos o histórico de modificações dos dados. Devemos saber também algumas informações a mais, como quem realizou a mudança, o tipo de alteração realizada, de onde partiu o comando, data, etc. Para isso, incluímos algumas colunas nas tabelas de auditoria como pode ser visto na Tabela 1. O *script* com todas as alterações realizadas está disponível para *download* no site da revista com o nome “script\_estrutura\_depois.sql”. Execute-o na base “sql\_magazine\_auditoria”.**

|  |  |
| --- | --- |
| **Coluna** | **Descrição** |
| tp\_alteracao\_auditoria | Tipo da alteração realizada, podendo ser I (INSERT), U (DELETE) ou D (DELETE). |
| dt\_registro\_auditoria | Data de gravação do registro de auditoria, ou seja, quando a modificação foi realizada na tabela auditada. |
| ds\_cliente\_host\_auditoria | Host (IP:porta) de onde se originou o comando de alteração. |
| nm\_usuario\_banco\_auditoria | Nome do usuário que realizou a alteração. |

**Tabela 1. Especificação das colunas extras para auditoria**

**Observe que, para cada tabela em uso pelo sistema, existe uma tabela equivalente na auditoria. Em uma situação real, isto dependerá da necessidade de cada administrador, pois existem casos em que é necessário auditar apenas algumas tabelas.**

**Com as tabelas criadas no esquema “auditoria”, precisamos criar as *procedures* e *triggers* que farão com que as alterações nos dados sejam auditadas.**

**Conforme a Listagem 2**, a *procedure* deve ser do tipo *trigger*, onde ela é executada como uma função sem argumentos, retornando um tipo interno do PostgreSQL chamado “trigger”. A *procedure* identifica o tipo do comando que está sendo executado e grava os dados na tabela de auditoria (linhas 4 a 16).

**De acordo com as linhas 7, 11 e 15 da Listagem 2, são usados os registros internos do PostgreSQL NEW e OLD. O NEW é usado para INSERT e UPDATE, pois precisamos da nova linha que está sendo inserida na tabela de produção para registrá-la na tabela de auditoria. O OLD é apenas utilizado no comando de DELETE para pegarmos a linha antiga que acabou de ser apagada na tabela auditada.**

**Observe também que não estamos fazendo referência às colunas individualmente e sim utilizando a forma NEW.\* e OLD.\*. Isto foi desenvolvido propositalmente para facilitar a escrita da função uma vez que a ordem das colunas das tabelas de produção é igual às das tabelas de auditoria.**

**Caso aconteça algum erro durante a gravação dos dados nas tabelas de auditoria, será lançada uma mensagem de erro (linhas 17 a 23), mas sem interromper a operação/transação que está sendo executada. Ou seja, se existirem mais comandos depois do comando que disparou a *trigger*, estes seguirão sem problema sem abortar a transação. Se não for de interesse perder as informações de auditoria, deve-se abortar a transação quando ocorrerem erros na mesma, substituindo o RETURN da linha 22 por um RAISE EXCEPTION, gerando uma exceção e abortando a transação inteira. Pode-se também, caso não seja de interesse abortar a transação, configurar o PostgreSQL para armazenar em log o comando que gerou o erro permitindo uma análise posterior.**

**Listagem 2. *Procedure* criada para a tabela tb\_usuario**

**1.      CREATE OR REPLACE FUNCTION auditoria.doAudit\_tb\_usuario() RETURNS trigger AS $$**

**2.      BEGIN**

**3.      BEGIN**

**4.      IF (TG\_OP = 'UPDATE') THEN**

**5.      INSERT INTO auditoria.tb\_audit\_tb\_usuario select 'U', now(), case when**

**6.      inet\_client\_addr() is null then 'localhost' else inet\_client\_addr()::varchar || ':' ||**

**7.      inet\_client\_port() end, session\_user, NEW.\*;**

**8.      ELSIF (TG\_OP = 'INSERT') THEN**

**9.      INSERT INTO auditoria.tb\_audit\_tb\_usuario select 'I', now(), case when**

**10.  inet\_client\_addr() is null then 'localhost' else inet\_client\_addr()::varchar || ':' ||**

**11.  inet\_client\_port() end, session\_user, NEW.\*;**

**12.  ELSE**

**13.  INSERT INTO auditoria.tb\_audit\_tb\_usuario select 'D', now(), case when**

**14.  inet\_client\_addr() is null then 'localhost' else inet\_client\_addr()::varchar || ':' ||**

**15.  inet\_client\_port() end, session\_user, OLD.\*;**

**16.  END IF;**

**17.  EXCEPTION**

**18.  WHEN others THEN**

**19.  RAISE NOTICE 'Ocorreu um erro durante a auditoria de dados!!';**

**20.  RAISE NOTICE 'SQLSTATE: %', SQLSTATE;**

**21.  RAISE NOTICE 'ERROR: %', SQLERRM;**

**22.  RETURN null;**

**23.  END;**

**24.  RETURN null;**

**...**