Estratégia Concursos

Aula 10 - Profs Thiago Cavalcanti e Erick Muzart

Banco do Brasil (Escriturário - Agente de Tecnologia) Banco de Dados - 2023 (Pós-Edital)

Autor:
Thiago Rodrigues Cavalcanti,
Erick Muzart Fonseca dos Santos,
Diego Carvalho

10 de Fevereiro de 2023

Índice

NOÇÕES SOBRE SGBDS POSTGRESQL

INTRODUÇÃO AO POSTGRESQL

Olá, pessoal. Começamos aqui mais uma aula de banco de dados cujo foco é entender as funcionalidades de um SGBD específico. Vamos contextualizar o ***PostgreSQL*** dentro do arcabouço de Banco de Dados para que você possa se situar no assunto.

A nossa divisão do conteúdo foi pensada da seguinte forma: primeiro veremos o processo de **administração de banco de dados** utilizando o SGBD PostgreSQL; em seguida, analisaremos as **peculiaridades dos comandos** de manipulação e criação de bases de dados. Neste segundo momento, trataremos também dos tipos de dados e das extensões do PostgreSQL à linguagem SQL/ANSI conhecida como pg/PLSQL.

Começaremos respondendo a uma pergunta básica!

O QUE É POSTGRESQL?

O banco de dados ***open source* mais avançado do mundo**! Isso mesmo! Do mundo! Alguém duvida disso? É desta forma que o site oficial anuncia! De forma mais racional, podemos dizer que o PostgreSQL é um sistema de gerenciamento de banco de dados **objeto-relacional** (ORDBMS) baseado no POSTGRES versão 4.2, desenvolvido na Universidade de Berkeley na Califórnia (**UCB**), mais especificamente no departamento de ciência da computação.

O PostgreSQL foi pioneiro em muitos conceitos que só se tornaram disponíveis em alguns sistemas de banco de dados comerciais posteriormente. Perceba que a comunidade de software livre em parceria com um conjunto de empresas e universidades trabalharam juntos para construir um sistema estável com diversas funcionalidades relevantes.

As principais qualidades que atraem massas de novos usuários a cada ano e mantêm os usuários atuais entusiasmados com seus projetos são sua **estabilidade**, **escalabilidade** e **segurança sólidas**, bem como os recursos que um sistema de gerenciamento de banco de dados de nível empresarial oferece.

O PostgreSQL é um descendente de código aberto do programa original desenvolvido em Berkeley. Ele **suporta** grande parte dos comandos da **linguagem SQL/ANSI** padrão e oferece muitas características modernas, entre elas: consultas complexas, chaves estrangeiras, *triggers*, visões atualizáveis, integridade transacional e controle de concorrência multiversão.

Além disso, o PostgreSQL **pode ser estendido** pelo usuário de várias maneiras. Por exemplo, adicionando novos tipos de dados, por meio dos *user defined types* (UDT), ou funções, utilizando as *user defined functions* (UDF). É possível ainda a criação de novos operadores, funções agregadas, métodos para criação de índice e linguagens procedurais.

Atribui-se boa parte do sucesso do SGBD à licença liberal ou *open source*. Por conta dela que o PostgreSQL pode ser usado, modificado e distribuído por qualquer pessoa gratuitamente para qualquer fim, seja ele privado, comercial ou acadêmico.

Após entendermos como o Postgres está situado dentro de mercados de SGBDs, vamos seguir nosso estudo analisando, com um pouco mais de detalhes, a trajetória que trouxe o PostgreSQL à versão atual. No dia 17 de fevereiro de 2021, a versão mais recente do Postgres é a 13, e a plataforma ainda dá suporte as versões 12, 11, 10 e 9.6, versões anteriores não possuem mais suporte (Unsupported versions: 9.5 / 9.4 / 9.3 / 9.2 / 9.1 / 9.0 / 8.4 / 8.3 / 8.2).

Uma rápida história do PostgreSQL

Primeiramente deixem-me justificar as próximas linhas. Você deve estar se perguntando: por que eu quero conhecer a história do PostgreSQL? Calma! Nosso intuito é apresentar como **as principais funcionalidades** evoluíram ao longo do tempo. Aproveitando essa oportunidade, para que você possa ir se familiarizando com **os termos associados** aos **serviços** providos por um SGBD. Vamos nessa?

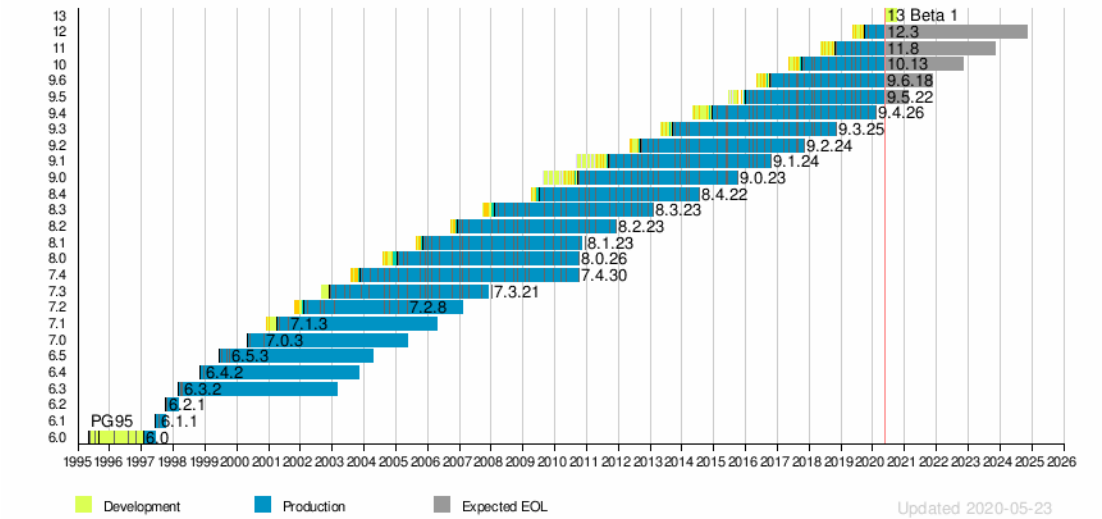


Figura 1 - Linha do tempo das releases do PostgreSQL

Originalmente chamado de **Postgres**, foi criado na UCB por um professor de ciência da computação chamado **Michael Stonebraker**, que se tornou o CTO da Informix Corporation, empresa que viria a ser comprada pela IBM. Em 1995, dois alunos de Ph.D. do laboratório de Stonebraker, **Andrew Yu e Jolly Chen**, substituíram a linguagem de consulta PostQUEL do Postgres por um subconjunto estendido de funções do SQL. Eles renomearam o sistema para Postgres95.

Percebam que neste momento o Postgres95 deu um grande passo para se tornar **um SGBD relacional**. Ao incorporar a linguagem SQL, facilitou a vida dos DBAs e outros profissionais que precisam fazer uso do banco de dados. Mas a evolução não parou por ai!

Em 1996, o Postgres95 saiu de dentro da academia e começou uma nova vida no universo ***open source*** fora do campus. Um grupo de desenvolvedores dedicados, fora de Berkeley, viu o sistema como uma promessa e se dedicou ao seu desenvolvimento contínuo. Se valendo de contribuições enormes de tempo, habilidade, trabalho e conhecimento técnico, esse **grupo global** de desenvolvimento **transformou radicalmente** o Postgres.

Durante os anos subsequentes, trouxeram **consistência e uniformidade** para o código fonte, criando **testes de regressão** detalhados para a garantia de qualidade e **listas de discussão** para relatórios de bugs, o que fez com que inúmeros bugs fossem corrigidos. E, ainda, foram adicionadas **novas funcionalidades** incríveis. Eles ajustaram todo o sistema, preenchendo várias lacunas, tais como **documentação** para desenvolvedores e usuários.

O fruto deste trabalho foi um novo banco de dados, que ganhou uma **reputação sólida e estabilidade**. O início de sua nova vida no mundo *open source* trouxe novos recursos adicionados e vários aprimoramentos que fizeram o sistema de banco de dados receber seu nome atual: **PostgreSQL**. O nome "Postgres" ainda é usado como **um apelido** mais fácil de pronunciar.

Agora com um novo nome ☺, o PostgreSQL começou sua contagem de versionamentos pela **versão 6.0**, dando crédito a seus muitos anos anteriores de desenvolvimento. Com a ajuda de centenas de desenvolvedores de todo o mundo, o sistema foi alterado e melhorado em quase todas as áreas. Ao longo de quatro anos (versões 6.0-7.0), grandes melhorias e novas funcionalidades foram implementadas:

Multiversion Concurrency Control (MVCC). O nível de bloqueio de tabela foi substituído por um sistema de **controle de concorrência multiversão** sofisticado, que permite aos usuários continuar lendo dados consistentes durante a atividade de escrita, possibilitando backups on-line, ou seja, enquanto o banco de dados está em execução.

Recursos importantes de SQL. Várias melhorias foram feitas na sintaxe do SQL, incluindo subconsultas, padronização, restrições, chaves primárias, chaves estrangeiras, identificadores entre aspas, tipo de coerção de string literal (*literal string type coercion*), tipo *casting*, entrada dos tipos **inteiro binário e hexadecimal**, entre outros.

Tipos internos (built-in) melhorados. Foram adicionados novos tipos nativos, incluindo uma ampla gama de tipos de **data/hora** e tipos **geométricos** adicionais.

Speed. Velocidade e desempenho tiveram grandes aumentos, na ordem de 20 a 40%, e o tempo de inicialização foi reduzido em 80%.

HORA DE PRATICAR!

Antes de continuar tratando das evoluções feitas no Postgres, vamos nos concentrar para fazermos a questão abaixo:

1. BANCA: CESPE ANO: 2014 ÓRGÃO: ANATEL PROVA: ANALISTA ADMINISTRATIVO - SUPORTE E INFRAESTRUTURA DE TI

A respeito de banco de dados, julgue os itens que se seguem.

O PostgreSQL 9.3, ao gerenciar o controle de concorrência, permite o acesso simultâneo aos dados. Internamente, a consistência dos dados é mantida por meio do MVCC (*multiversion concurrency control*), que impede que as transações visualizem dados inconsistentes.

Comentário. O PostgreSQL fornece um rico conjunto de ferramentas para os desenvolvedores gerenciarem o acesso simultâneo aos dados. Internamente, a consistência dos dados é mantida por meio de um **modelo de concorrência multiversão** (MVCC). Isto significa que, ao consultar um banco de dados, **cada transação enxerga uma fotografia dos dados** (ou uma versão do banco de dados) em algum momento passado, independentemente do estado atual dos dados subjacentes ou relacionados.

Isso protege a transação de enxergar dados inconsistentes. Transações simultâneas sobre as mesmas linhas de dados poderiam levar o banco de dados a um estado inválido. O MVCC acaba fornecendo um isolamento entre as transações para cada sessão de banco de dados. Assim, por **não** utilizar as metodologias de **bloqueio** de sistemas de banco de dados tradicionais, **minimiza a disputa** pelos objetos de dados, a fim de permitir um bom desempenho em ambientes multiusuários.

A principal vantagem de usar o modelo MVCC no controle de concorrência, em vez de bloqueios de itens de dados é que, no MVCC, bloqueios adquiridos para consulta de dados (leitura) não conflitam com os bloqueios adquiridos para a gravação de dados. Desta forma, **a leitura nunca bloqueia a escrita**, e a escrita nunca bloqueia a leitura. O PostgreSQL mantém essa garantia mesmo quando fornece um nível mais rigoroso de isolamento de transações.

Gabarito: C.

Os quatro anos seguintes (versões 7.0 a 7.4) trouxeram o ***Write-Ahead Log*** (WAL), esquemas SQL, *prepared queries, outer joins,* consultas complexas, sintaxe do *join* do SQL92, TOAST[1] (The Oversized-Attribute Storage Technique), suporte a IPv6, ***information schema* do SQL-padrão**, ***full-text index***, *auto-vacuum*, linguagens procedurais **Perl/Python/TCL**, melhor suporte a SSL, otimizador de consulta revisado, informações estatísticas sobre o banco de dados, maior segurança, *table functions* e, ainda, melhorias no log e avanços significativos no desempenho, entre outras coisas.

Podemos observar que uma pequena amostra do desenvolvimento intenso do PostgreSQL, ao longo do tempo, é refletida em suas notas de lançamento (release notes). Vejam um exemplo das notas de lançamento da versão 9.6.1. Chegaremos nela mais à frente! Antes precisamos dar continuidade ao contexto histórico que descreve a evolução das funcionalidades.

As versões 8.0 a 8.4 foram lançadas entre os anos de 2005 a 2009. Nestes quatro anos de desenvolvimento, tivemos o incremento das seguintes funcionalidades: servidor nativo Microsoft Windows, *savepoints, tablespaces*, manipulação de exceção em funções, recuperação *point-in-time,* otimização de desempenho, commit em duas fases,

[1] O PostgreSQL utiliza a técnica TOAST (The Oversized-Attribute Storage Technique), que permite que campos grandes sejam armazenados mesmo se for necessário que parte do dado seja armazenada em um bloco diferente da memória.

particionamento de tabelas, verificação de índice bitmap, bloqueio de linha compartilhada, papéis (roles), *online index builds, advisory locks, warm standby*, tuplas do tipo *heap-only, full-text search*, SQL/XML, tipo ENUM, tipo UUID (universally unique identifiers), funções de janelas, parâmetros default para funções, permissões em nível de coluna, restauração de banco de dados paralelos, agrupamento por banco de dados, expressões de tabela comuns (CTE) e consultas recursivas.

Em seguida vieram as versões 9.0 a 9.6. Por meio delas, vários incrementos de funcionalidades foram feitos ao sistema. Vejam quais foram as melhorias inclusas, de acordo com as versões na tabela abaixo:

9	Replicação de streaming binário interna (built-in), *hot standby*, suporte ao Windows 64-bits, gatilhos-por-coluna e execução trigger condicional, restrições de exclusão, blocos de código anônimos, parâmetros nomeados, regras de senha.
9.1	Replicação síncrona, agrupamentos por colunas, tabelas *unlogged*, indexação dos vizinhos mais próximos (k-NN), isolamento de instantâneo serializado (SSI), expressões de tabela comuns armazenáveis, integração com Linux-SE, extensões, tabelas anexadas SQL/Med (Wrappers para dados externos), gatilhos em views.
9.2	*Cascading replication streaming*, varreduras index-only, suporte a JSON nativo, melhoria da gestão de bloqueio, range de tipos, ferramenta pg_receivexlog, índices GiST para espaço-particionado.
9.3	Tarefas de background customizáveis, checksums de dados, operadores JSON dedicados, LATERAL JOIN, pg_dump mais rápido, nova ferramenta de monitoramento de servidor **pg_isready**, características de triggers, funcionalidades para visões, tabelas estrangeiras graváveis, visões materializadas e melhorias de replicação.
9.4	Tipo de dados JSONB, instrução ALTER SYSTEM para alterar valores de configuração, atualização de visões materializadas sem bloqueio de leitura, registro dinâmico para registro/start/ stop de processos em segundo plano, Logical API Decoding, melhorias de

	índice GiN, suporte a Linux huge page e cache de recarga de banco de dados via pg_prewarm.
9.5	IMPORT FOREIGN SCHEMA, row-level security policies, BRIN Index (Block Rage), Foreign table inheritance, GROUPING SETS, CUBE e ROLLUP, JSONB – operadores de modificações e funções, INSERT … ON CONFLICT DO NOTHING/UPDATE(UPSERT), pg_rewind.
9.6	Esta nova versão permitirá aos usuários escalar com flexibilidade cargas de dados em alta performance. As novas funcionalidades incluem a realização de consultas de forma paralela, melhorias nas técnicas de replicação síncrona, busca textual por frases, além de melhorias em performance e usabilidade.

As versões continuam, chegamos às versões 10 e 11, abaixo temos mais algumas funcionalidades que forma adicionadas:

10.6	Replicação Lógica (uma estrutura de publicação/assinatura para distribuição de dados), Particionamento Declarativo de Tabelas, Melhor paralelismo de consulta, Commit por Quórum para Replicação Síncrona, Autenticação SCRAM-SHA-256
11.1	Maior Robustez e Desempenho para Particionamento, Transações Suportadas em Procedimentos Armazenados, Recursos Aprimorados para Paralelismo de Consultas, Compilação Just-in-Time (JIT) para Expressões, Melhorias Gerais para Experiência do Usuário, como inclusão das palavras-chave "quit" e "exit" na interface de linha de comando

Já estamos acabando nossas apresentações das versões, falta apenas comentarmos sobre às versões 12 e 13. E ... **O que há de novo no PostgreSQL 12?** O PostgreSQL 12 foi lançado em 3 de outubro de 2019. Inclui um amplo conjunto de novos recursos em relação às versões anteriores, incluindo o seguinte:

- Várias otimizações de desempenho, variando de expressões de tabela comuns (CTE) embutidas ao enorme gerenciamento de partições de tabelas
- Algumas otimizações administrativas, incluindo a reconstrução simultânea de índices, soma de verificação off-line e, mais notavelmente, relatórios sobre o progresso dos processos de manutenção

- Recursos de segurança, incluindo autenticação multifator e criptografia TCP/IP via GSSAPI[2] (Generic Security Service Application Program Interface)
- Suporte para expressões de caminho SQL/JSON que especificam os itens a serem recuperados dos dados JSON, semelhantes às expressões XPath usadas para acesso SQL a XML.
- Colunas geradas armazenadas são colunas especiais que sempre são calculadas a partir de outras colunas.

O PostgreSQL 12 também contém um conjunto de mudanças destinadas a facilitar a vida do **administrador de banco de dados** (**DBA**), por exemplo, removendo opções conflitantes, e termos e tipos obsoletos de SQL. Isso enfatiza o fato de que os desenvolvedores PostgreSQL sempre cuidam do banco de dados e de sua aderência ao padrão SQL atual.

Enfim chegamos à versão atual!! Aleluia!! E ... o que há de novo no PostgreSQL 13?

O PostgreSQL 13 contém um conjunto muito rico de otimizações internas, com atenção especial ao seguinte:

- **Particionamento**, que agora inclui a capacidade de executar *antes dos* gatilhos em tabelas particionadas, a capacidade de remover partições em casos extremos específicos para acelerar a execução de consultas e uma maneira melhor de unir partições em consultas (chamadas de **junções** por **partição**).
- **Replicação**, que agora pode funcionar no nível lógico mesmo em tabelas particionadas, publicando automaticamente todas as partições. Além disso, agora não há promoção automática de um servidor se ele não atingir o alvo específico para recuperação e um servidor escravo pode ser promovido sem cancelar nenhuma solicitação de pausa pendente. É importante notar que é possível alterar as configurações de uma replicação de streaming sem ter que reiniciar o cluster, portanto, sem impacto de tempo de inatividade.

[2] A GSSAPI é uma interface que permite desenvolvedores escreverem aplicações que aproveitam mecanismos de segurança tais como Kerberos, sem ter de programar explicitamente para qualquer mecanismo, ou seja, aplicações genéricas do ponto de vista de segurança.

- **Índices**, que agora são mais eficientes em geral para armazenar dados e aceitar operadores com parâmetros.
- **Estatísticas**, com particular atenção às melhorias nas estatísticas estendidas, aos dados coletados e usados pelo otimizador e a algumas mudanças nos catálogos de monitoramento.

Você pode ter visto uma quantidade enorme de palavras acima cujo significado você ainda não tem conhecimento. Afirmo de antemão que não vamos detalhar a definição de todas elas, mas posso garantir que passaremos pelas **principais funcionalidades**. Essa lista vai funcionar como uma memória auxiliar. Você verá como seu cérebro funciona bem, quando solicitarem informações a respeito das funcionalidades existentes ou não no PostgreSQL.

Por ser um banco de dados relacional, o PostgreSQL oferece muitos recursos e é muito difícil "assustar" uma instância do PostgreSQL. Na verdade, uma única instância pode conter mais de 4 bilhões de bancos de dados individuais, cada um com tamanho total ilimitado e capacidade para mais de 1 bilhão de tabelas, cada uma contendo 32 TB de dados.

Além disso, se houver alguma preocupação de que esses limites superiores não sejam suficientes, considere que uma única tabela pode ter 1.600 colunas, cada uma com 1 GB de tamanho, com um número ilimitado de índices de várias colunas (até 32 colunas). Resumindo, o PostgreSQL pode armazenar muito mais dados do que você pode imaginar! Agora, para usar essa ferramenta tão poderosa, precisamos instalar o SGBD na nossa máquina.

INSTALANDO O POSTGRESQL

Vamos então dar os primeiros passos para o entendimento. O PostgreSQL é um servidor de banco de dados SQL avançado, disponível em uma ampla gama de plataformas. Está disponível gratuitamente para usar, alterar ou redistribuir da forma como você quiser. Sua licença é de código aberto, muito parecida com a licença BSD (Berkeley Software Distribution), embora diferente o suficiente para ser conhecida como TPL (**The PostgreSQL License**).

O PostgreSQL já é usado por muitos pacotes de aplicativos diferentes e, por isso, é possível que você já o encontre instalado em seus servidores. Muitas distribuições Linux incluem PostgreSQL como parte da instalação básica ou fornecem a opção de incluí-lo durante a instalação.

PostgreSQL é executado em praticamente todos os sistemas operacionais existentes, incluindo Linux, Unix, Mac OS X e Microsoft Windows, e pode até mesmo ser executado em

hardware comum, como placas Raspberry Pi. Existem também vários provedores de computação em nuvem que listam PostgreSQL em seu catálogo de software.

Se você ainda não tem uma cópia ou não tem a versão mais recente, pode baixar o código-fonte ou baixar um dos pacotes de binários para uma ampla variedade de sistemas operacionais a partir da seguinte URL:

http://www.postgresql.org/download/

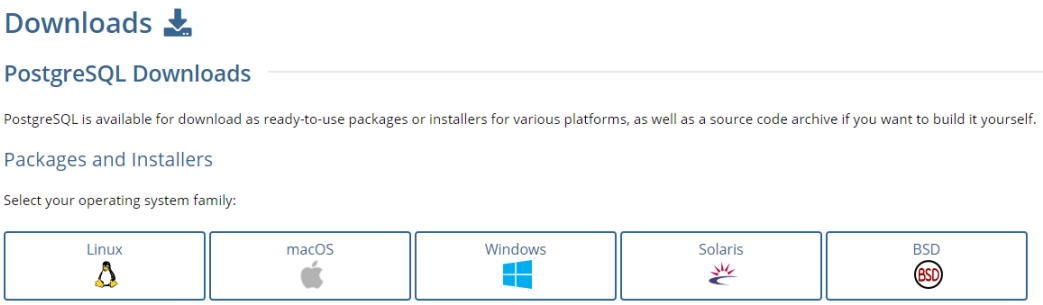


Figura 2 - Tela para baixar a versão do PostgreSQL. Veja que você vai escolher de acordo com o sistema operacional.

Detalhes da instalação variam significativamente de uma plataforma para outra e não existem truques especiais ou receitas para mencionar. Basta seguir o guia de instalação e você vai longe! Apenas a título de exemplo, vou mostrar um pouca da instalação que fiz na minha máquina da versão que usaremos como exemplo ao longo do nosso curso. Primeiramente, nesta máquina uso o sistema Windows, logo, após clicar no ícone do Windows na tela acima, cheguei neste ponto:

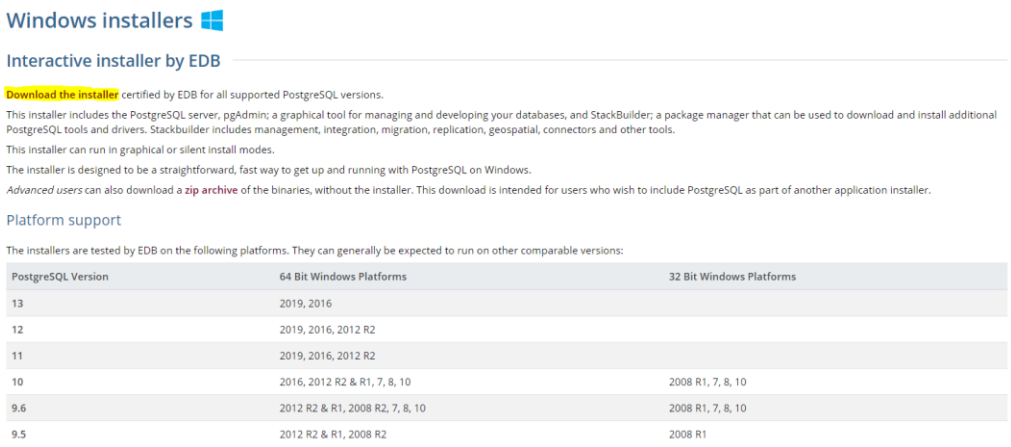


Figura 3 - Segunda tela para download do programa.

Perceba que deixei salientando o link que você deve clicar para baixar o instalador. Passamos então para esta tela:

PostgreSQL Database Download

Version	Linux x86-64	Linux x86-32	Mac OS X	Windows x86-64	Windows x86-32
13.2	N/A	N/A	Download	Download	N/A
12.6	N/A	N/A	Download	Download	N/A
11.11	N/A	N/A	Download	Download	N/A
10.16	Download	Download	Download	Download	Download
9.6.21	Download	Download	Download	Download	Download
9.5.25	Download	Download	Download	Download	Download
9.4.26 (Not Supported)	Download	Download	Download	Download	Download
9.3.25 (Not Supported)	Download	Download	Download	Download	Download

Figura 4 - Selecionei a opção de Windows 64 bits.

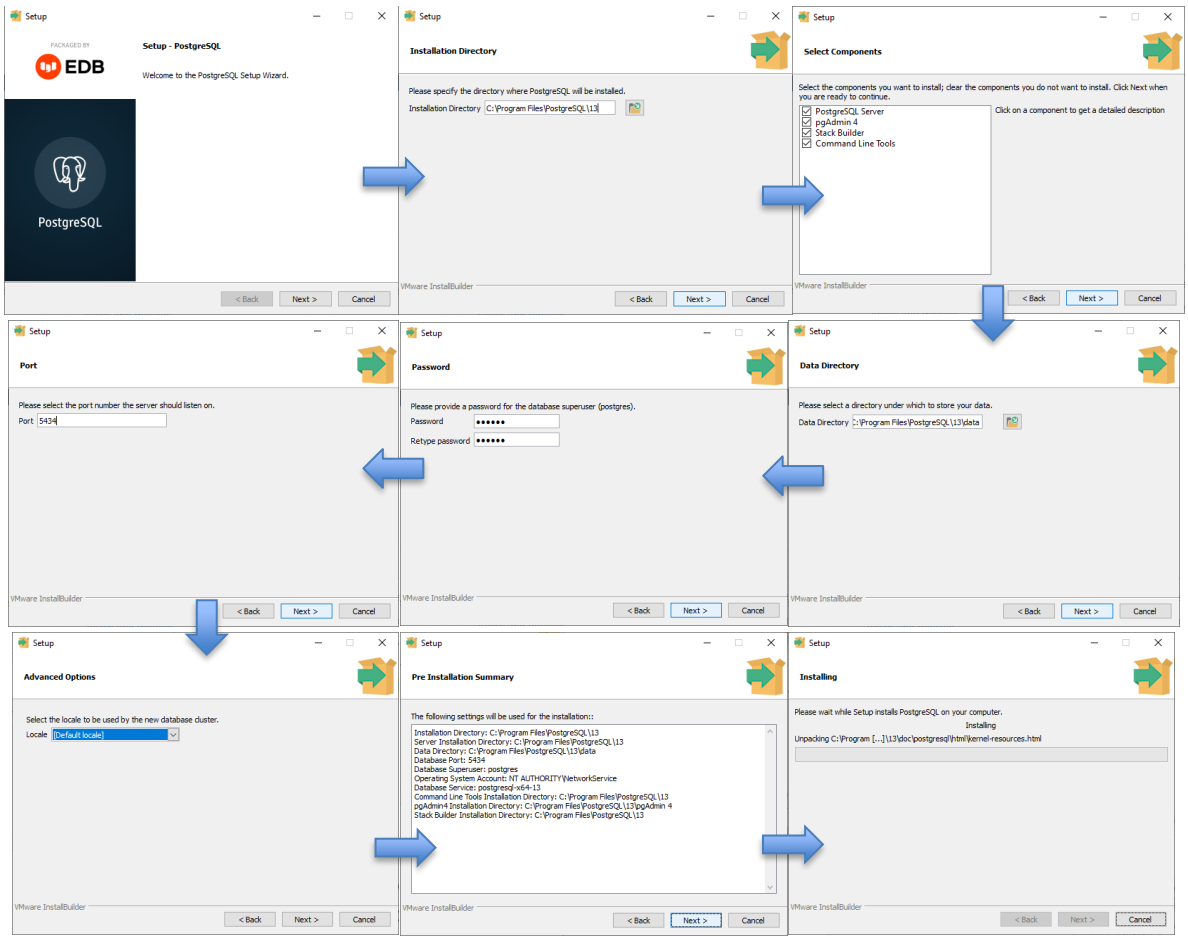


Figura 5 - Passo a passo das telas de instalação no Windows

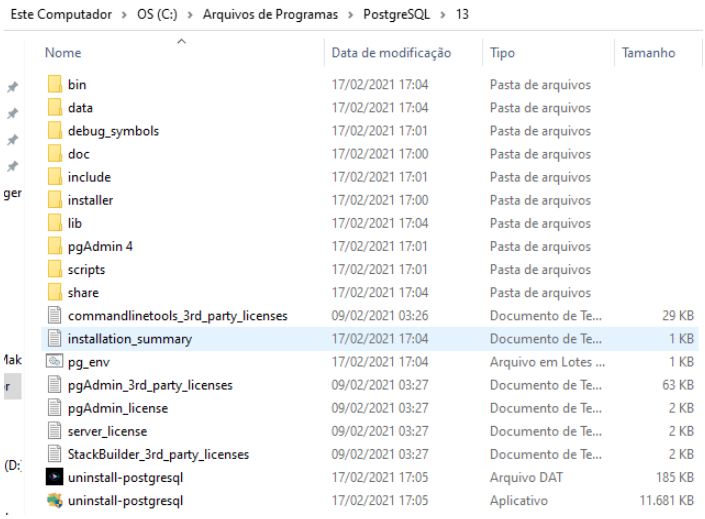


Figura 6 - Pasta com diretórios e arquivos do PostgresSQL

O processo de instalação envolve a cópia de todos os programas compilados em um diretório que irá servir como a base para todas as atividades do PostgreSQL. Ele também conterá todos os programas, bancos de dados PostgreSQL e arquivos de log. O diretório é normalmente chamado /usr/local/pgsql ou c:/Arquivos de Programa/PostgreSQL.

Outro ponto importante é que existem duas maneiras principais de colocar o PostgreSQL em execução, da seguinte maneira: compilando o código fonte e usando um pacote binário

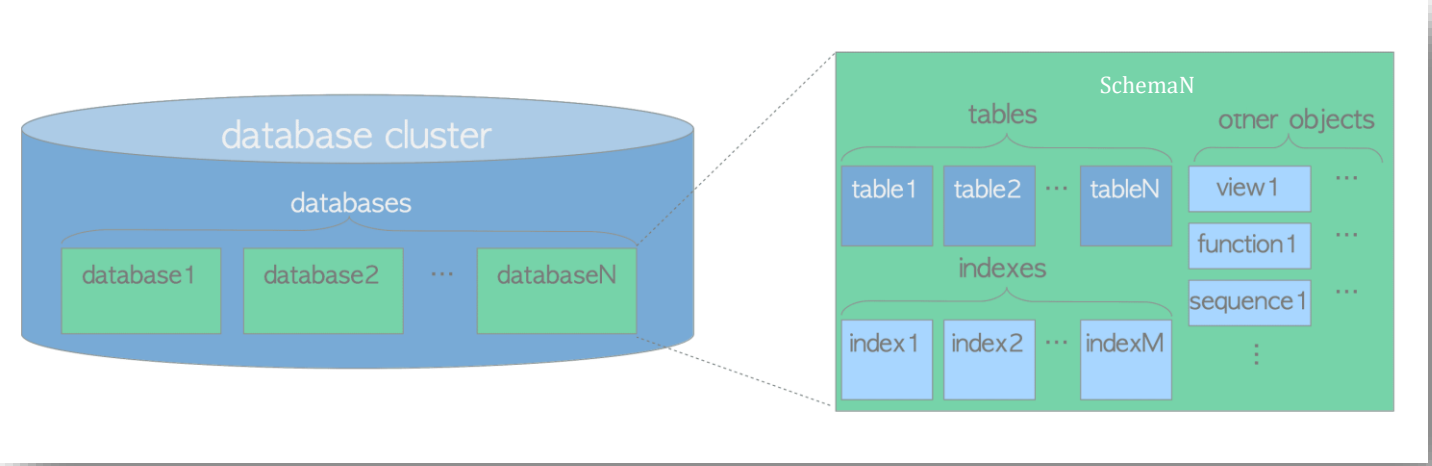
Os pacotes binários são fornecidos pela comunidade PostgreSQL ou pelo sistema operacional, e usá-los tem a vantagem de fornecer uma instalação do PostgreSQL muito rapidamente. A instalação acima foi feita a partir de um pacote binário para Windows. Além disso, os pacotes binários não requerem uma cadeia de ferramentas de compilação e, portanto, são mais fáceis de adotar. Por último, um pacote binário segue as convenções do sistema operacional para o qual foi criado (por exemplo, onde colocar os arquivos de configuração).

Veja que a versão instalada foi a 13.2, os dois números representam o código da versão principal (major) e da versão menor (minor). Uma versão principal é uma versão estável que apresenta novos recursos e possíveis incompatibilidades com versões anteriores. Durante seu ciclo de vida, uma versão principal é constantemente aprimorada por meio de versões menores, que geralmente são versões de manutenção e correção de bugs.

O PostgreSQL fornece suporte e atualizações por **5 anos após uma nova versão** ser lançada. Após esse período, uma versão principal atingirá seu fim de vida (EOL - End of Life) e os desenvolvedores do PostgreSQL não farão mais manutenção. Isso não significa que você não pode executar uma versão antiga do PostgreSQL, significa simplesmente que esta versão não receberá nenhuma atualização do projeto oficial.

EXPLORANDO A TERMINOLOGIA DO POSTGRESQL

Uma instância do PostgreSQL é chamada de **cluster** porque uma única instância pode **servir e lidar com vários bancos de dados**. Cada banco de dados é um espaço isolado onde usuários e aplicativos podem armazenar dados. Um banco de dados é acessado por usuários autorizados, mas os usuários conectados a um banco de dados não podem cruzar os limites do banco de dados e interagir com os dados contidos em outro banco de dados, a menos que se conectem explicitamente ao último banco de dados também.



Um banco de dados pode ser organizado em ***namespaces***, chamados de *esquemas (schema)*. Um esquema pode ser descrito por um nome mnemônico que o usuário atribui para organizar objetos de banco de dados, como tabelas, em uma coleção mais estruturada. Os esquemas **não podem ser aninhados**, portanto, representam um namespace simples.

Os objetos de banco de dados são representados por tudo que o usuário pode criar e gerenciar no banco de dados - por exemplo, **tabelas, funções, gatilhos e tipos de dados**. Cada objeto pertence a um e apenas um esquema que, se não for especificado, é o esquema *público* padrão.

Os usuários são definidos em todo o cluster, o que significa que não estão vinculados a um banco de dados específico no cluster. Um usuário pode se conectar e gerenciar qualquer banco de dados no cluster para o qual tenha permissão.

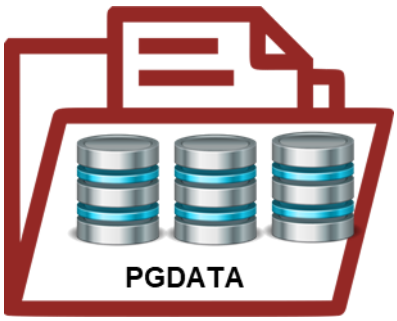
O PostgreSQL divide os usuários em duas categorias principais:

- **Usuários normais**: esses usuários são aqueles que podem se conectar e manipular bancos de dados e objetos, **dependendo de seu conjunto de privilégios**.
- **Superusuários**: Esses usuários podem fazer qualquer coisa com qualquer objeto de banco de dados.

O PostgreSQL permite a configuração de quantos superusuários você precisar, e cada superusuário tem as mesmas permissões: eles podem fazer tudo com todos os bancos de dados e objetos e, mais notavelmente, também podem controlar o ciclo de vida do cluster (por exemplo, eles podem terminar as conexões normais do usuário, recarregar a configuração, parar todo o cluster e assim por diante).

Os dados internos do PostgreSQL, como usuários, bancos de dados, namespaces, configuração e status de tempo de execução do banco de dados, são fornecidos por meio de **catálogos**: **tabelas especiais que apresentam informações de forma interativa com SQL**. Muitos catálogos são ocultados dependendo do usuário que os inspeciona, com a exceção de que os superusuários geralmente veem todo o conjunto de informações disponíveis.

O PostgreSQL armazena os dados do usuário (por exemplo, tabelas) e seu status interno no sistema de arquivos local. Este é um ponto importante a se ter em mente: o PostgreSQL **depende do sistema de arquivos subjacente para implementar a persistência** e, portanto, ajustar o sistema de arquivos é uma tarefa importante para fazer o PostgreSQL funcionar bem. Em particular, o PostgreSQL armazena todo o seu conteúdo (dados do usuário e status interno) em um único diretório do sistema de arquivos conhecido como **PGDATA**.



O diretório PGDATA representa o que o cluster está servindo como banco de dados, portanto, é possível ter uma única instalação do PostgreSQL e fazer com que ele alterne para diretórios diferentes para fornecer conteúdo diferente. Na verdade, essa é uma forma possível de implementar atualizações rápidas entre as versões principais. O diretório PGDATA precisa ser inicializado antes de ser usado pelo PostgreSQL; a inicialização é a criação da estrutura de diretório com o PGDATA.

O conteúdo detalhado de PGDATA será detalhado em um momento oportuno, mas por agora, será suficiente para você lembrar que o diretório é onde o PostgreSQL espera encontrar os dados e os arquivos de configuração. Em particular, **o diretório possui os write-ahead logs (WALs) e um espaço para armazenamento dos dados**. Sem qualquer

uma dessas duas partes, o cluster é incapaz de garantir a consistência dos dados e, em algumas circunstâncias críticas, até mesmo iniciar.

Os WALs são uma tecnologia que muitos sistemas de banco de dados usam e até mesmo alguns sistemas de arquivos de transação (como ZFS e ReiserFS) fornecem. A ideia é que, antes de aplicar qualquer alteração a um bloco de dados, uma intenção é registrada no log. Nesse caso, se o cluster travar, ele sempre pode contar com o log já gravado para entender quais operações foram concluídas e o que deve ser recuperado.

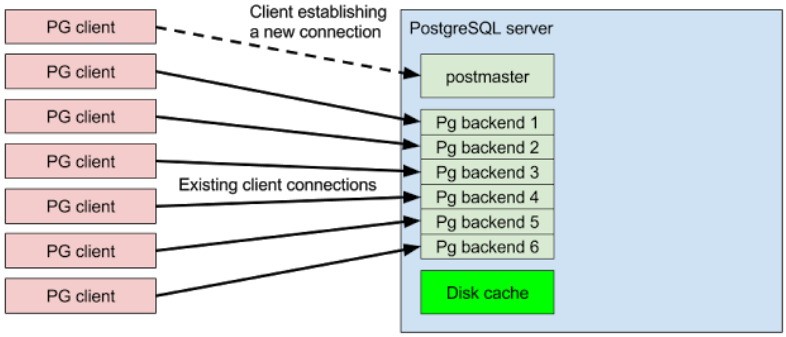
Observe que, com o termo "travamento", nos referimos a qualquer possível desastre que pode atingir seu cluster, incluindo um bug de software, mas mais provavelmente a falta de energia elétrica, falhas no disco rígido e assim por diante. Com isso você percebe que o PostgreSQL se compromete a fornecer a você a melhor consistência de dados possível.

Internamente, o PostgreSQL mantém o controle das estruturas das tabelas, índices, funções e todas as coisas necessárias para gerenciar o cluster em armazenamento dedicado denominado **catálogo**. O catálogo PostgreSQL é fundamental para o ciclo de vida do cluster e reflete praticamente todas as ações do banco de dados nas estruturas e dados do usuário. O PostgreSQL fornece acesso ao catálogo de superusuários do banco de dados por meio de uma interface SQL, o que significa que o catálogo é totalmente explorável e, em certa medida, manipulável, por meio de instruções SQL.

O padrão SQL define um chamado o information schema, uma coleção de tabelas comuns a todas as implementações de banco de dados padrão, incluindo PostgreSQL, que o DBA pode usar para inspecionar o estado interno do próprio banco de dados. Por exemplo, o information schema define uma tabela que coleta informações sobre todas as tabelas definidas pelo usuário para que seja possível consultá-lo para ver se uma tabela específica existe ou não.

O catálogo do PostgreSQL é o que alguns chamam de "information schema com esteroides": o catálogo é muito mais preciso e específico que o information schema geral, assim o DBA pode extrair mais informações sobre o status do PostgreSQL do catálogo.

Quando o cluster é iniciado, o PostgreSQL inicia um único processo chamado ***postmaster***. O objetivo do ***postmaster*** é aguardar as conexões de entrada do cliente, geralmente feitas através de uma conexão TCP/ IP, e dão origem a outro processo denominado *processo de* ***back-end***, que por sua vez é responsável por servir uma e apenas uma conexão.



Isso significa que toda vez que uma nova conexão com o cluster é aberta, o cluster reage lançando ***um novo processo de back-end*** para atendê-lo até que a conexão termine e o processo seja, consequentemente, destruído. O *postmaster* geralmente também inicia alguns processos utilitários que são responsáveis por manter o PostgreSQL funcionando corretamente; esses processos serão discutidos posteriormente.

Para resumir, o PostgreSQL fornece executáveis que podem ser instalados onde você quiser em seu sistema e podem servir a um único cluster. O cluster, por sua vez, fornece dados de um único diretório (PGDATA) que contém, entre outras coisas, os dados do usuário, o status interno do cluster e os WALs. Cada vez que um cliente se conecta ao servidor, o processo ***postmaster*** bifurca um novo processo de ***back-end*** que é o lacaio encarregado de servir a conexão.

Esta é uma rápida recapitulação dos principais termos usados no PostgreSQL:

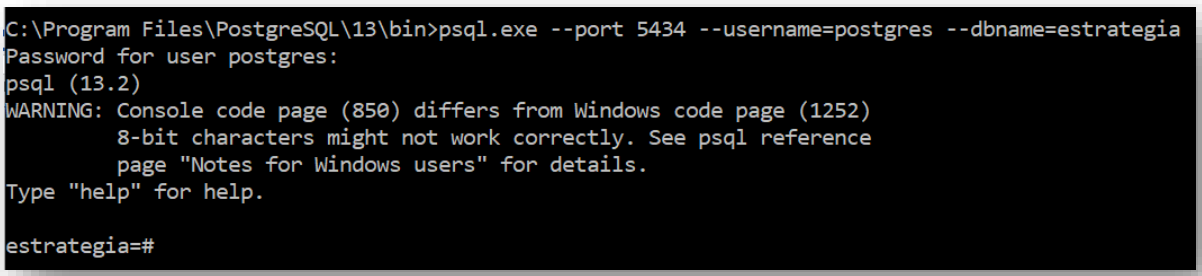
- **Cluster**: refere-se a todo o serviço PostgreSQL.
- **Postmaster**: este é o primeiro processo que o cluster executa, e este processo é responsável por manter o controle das atividades de todo o cluster. O postmaster bifurca-se em um processo de back-end toda vez que uma nova conexão é estabelecida.
- **Banco de dados**: o banco de dados é um contêiner de dados isolado ao qual os usuários (ou aplicativos) podem se conectar. Um cluster pode lidar com vários bancos de dados. Um banco de dados pode ser feito por diferentes objetos, incluindo esquemas (*namespaces*), tabelas, gatilhos e outros objetos.
- PGDATA: é o nome do diretório que, no armazenamento persistente, é totalmente dedicado ao PostgreSQL e seus dados. O PostgreSQL armazena os dados nesse diretório.
- **WALs**: contém o log de intenção de alterações do banco de dados, usado para recuperar dados de um travamento crítico.

CONECTANDO A UM SERVIDOR POSTGRES

Ok! Instalamos nosso banco de dados, conhecemos seus conceitos básicos e agora? Como faremos para utilizá-lo? Como devemos nos comunicar com ele? Conectar-se ao banco de dados é a primeira experiência da maioria das pessoas. Então, vamos tê-la e corrigir quaisquer problemas que tivermos ao longo do caminho. Lembre-se que a conexão deve ser feita de forma segura, portanto, pode haver alguns obstáculos para garantir que os dados que desejamos acessar estejam seguros.

Para se conectar, precisamos ter um servidor PostgreSQL rodando no host, escutando uma porta. Nesse servidor, devem existir um banco de dados e um usuário devidamente cadastrado. Uma dica é rodar o pgAdmin4, que vem instalado com a versão 13, e observar que o servidor ser inicializado antes da ferramenta de administração abrir. O ícone do elefante vai aparecer na barra de ferramentas do Windows. O host deve permitir explicitamente conexões dos clientes que fornecem os dados para autenticação, usando o método especificado pelo servidor. Por exemplo, especificar uma senha não irá funcionar, se o servidor solicitou uma forma diferente de autenticação.

Agora vamos usar o psql, esse programa está disponível na pasta bin da sua instalação. Você pode abrir o comand (cmd) do Windows e fazer uma conexão com o banco de dados. Vejam abaixo um exemplo de uma conexão usando o SQL Shell (psql). Nele utilizamos os seguintes dados: localhost para o servidor, estrategia para o banco de dados, 5434 para a porta e postgreSQL para o usuário.



Dica: O **psql** é um front-end baseado em terminal para PostgreSQL. Ele permite que você digite os comandos interativamente, envie para o PostgreSQL e veja os resultados da consulta. Alternativamente, sua entrada pode ser de um arquivo. Além disso, ele fornece um número de meta-comandos e diversas funcionalidades *shell-like* para facilitar a escrita de scripts e automatizar uma grande variedade de tarefas. É importante conhecê-lo!

A primeira coisa a observar é que, uma vez que a conexão foi estabelecida, o prompt de comando muda: o psql relata o banco de dados ao qual o usuário foi conectado (**estrategia**) e, se for o caso, apresenta um sinal para indicar que ele é um **superusuário (#).** Caso o

usuário não seja um administrador de banco de dados, um sinal de (>) é colocado no final do prompt.

O servidor de banco de dados PostgreSQL é classificado como cliente-servidor. O local onde o sistema roda é conhecido como host. Podemos acessar o servidor PostgreSQL remotamente através da rede. No entanto, devemos especificar o host, que é um nome de host, ou um hostaddr, que é um endereço IP. Podemos especificar o host como "localhost", se quisermos fazer uma conexão **TCP/IP** para o mesmo servidor

Em qualquer sistema, pode haver mais de um servidor de banco de dados. Cada servidor de banco de escuta em exatamente uma porta de rede, que não pode ser compartilhada entre os servidores no mesmo host. O número da porta padrão para PostgreSQL é **5432**, que foi registrado na Internet Assigned Numbers Authority (**IANA)**, e é atribuída exclusivamente ao PostgreSQL. O número da porta pode ser usado para identificar exclusivamente um servidor de BD específico, se existirem muitos.

Um servidor de banco de dados também é conhecido como um "cluster de banco de dados", porque o servidor PostgreSQL permite definir um ou mais bancos de dados em cada servidor. Cada solicitação de conexão deve identificar exatamente um banco de dados identificado por seu DBNAME. Quando você se conectar, você só será capaz de ver objetos do banco de dados criados dentro dessa base especificada na conexão.

Dica do professor 01:

Se você quiser confirmar que se conectou ao servidor certo e da maneira certa, você pode executar alguns ou todos os seguintes comandos:

SELECT inet_server_port(); - Exibe a porta na qual o servidor está escutando.

SELECT current_database(); - Mostra o banco de dados atual.

SELECT current_user; - Mostra o ID do usuário atual.

SELECT inet_server_addr(); - Mostra o endereço IP do servidor que aceitou a conexão.

A senha do usuário **não** é acessível usando SQL geral, por razões óbvias.

```
estrategia=# select current_user;
 current_user
--------------
 postgres
(1 row)


estrategia=# select current_database();
 current_database
------------------
 estrategia
(1 row)


estrategia=# select inet_server_port();
 inet_server_port
------------------
             5434
(1 row)


estrategia=#  select inet_server_addr();
 inet_server_addr
------------------
 ::1
(1 row)
```

Vejamos então como esse assunto já foi cobrado em provas anteriores:

HORA DE PRATICAR!

1. BANCA: FCC ANO: 2012 ÓRGÃO: TCE-AM PROVA: ANALISTA TÉCNICO DE CONTROLE EXTERNO - TECNOLOGIA DA INFORMAÇÃO

Sobre os fundamentos arquiteturais do banco de dados PostgreSQL, considere:

I. Utiliza um modelo cliente/servidor, consistindo de um processo servidor que gerencia os arquivos do banco de dados, controla as conexões dos clientes ao banco dados e efetua ações no banco de dados em favor dos clientes.

II. A aplicação cliente, que irá efetuar as operações no banco de dados, poderá ser de diversas naturezas, como uma ferramenta em modo texto, uma aplicação gráfica, um servidor web que acessa o banco de dados para exibir as páginas ou uma ferramenta de manutenção especializada.

III. A aplicação cliente pode estar localizada em uma máquina diferente da máquina em que o servidor está instalado. Neste caso, a comunicação entre ambos é efetuada por uma conexão TCP/IP. O servidor pode aceitar diferentes conexões dos clientes ao mesmo tempo.

Está correto o que se afirma em

A I, II e III.

B I e II, apenas.

C I e III, apenas.

D II e III, apenas.

E III, apenas.

Comentário. Observem que, segundo o conteúdo apresentado acima, podemos afirmar que as alternativas I, II e III estão corretas. A alternativa II foi dada como incorreta pelo examinador. Mas, se olharmos o texto abaixo, retirado do manual de referência do PostgreSQL, veremos que ela nada mais é do que uma tradução do texto:

“The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution; most are developed by users.”

Gabarito: A

Dica do professor 02:

Ao entrar no **psql**, o que você pode fazer? Executar comandos SQL para criação de objetos na base ou para manipulação de tabelas e outros objetos da base dados. Algumas dicas podem ajudar, digite:

\ h - para ajuda com comandos SQL

\? - para obter ajuda sobre comandos internos

\g ou ponto e vírgula (;) - para executar consulta

\copyright - para termos de distribuição

\q - para sair

Existe uma variedade de distribuições possíveis para o PostgreSQL. Em muitas delas, você vai verificar que o acesso remoto é desabilitado por padrão, como medida de segurança. Vamos, a seguir, aprender a fazer ajustes nos arquivos de configuração para aceitação de conexões remotas.

ARQUIVOS DE CONFIGURAÇÃO PARA ACESSO REMOTO

Vamos agora conhecer alguns arquivos nos quais precisamos fazer alterações para admitirmos o acesso remoto. Primeiramente mostraremos quais são as alterações em cada arquivo. Em seguida, teceremos os comentários teóricos explicativos a respeito de cada alteração.

- Adicionar/editar a seguinte linha no seu arquivo postgresql.conf:
    - listen_addresses = '*'
    - O parâmetro listen_addresses especifica qual o range de endereços IP é possível escutar. Isso permite que você tenha mais de uma placa de rede (NICs) por sistema. Na maioria dos casos, nós queremos aceitar conexões

em todas as NICs. Por isso, usamos "*", que significa "todos os endereços IP".

- Adicionar a seguinte linha como a primeira linha do arquivo pg_hba.conf, para permitir o acesso a todos os bancos de dados para todos os usuários, utilizando uma senha criptografada:

```
# TYPE    DATABASE    USER          CIDR-ADDRESS    METHOD
host          all     all           0.0.0.0/0       md5
```

A autenticação do cliente é controlada por um arquivo de configuração, que tradicionalmente é chamado **pg_hba.conf** e é armazenado no diretório de dados raiz do banco de dados (HBA significa **autenticação baseada em host**). Um arquivo pg_hba.conf padrão é criado quando o diretório de dados é inicializado pelo initdb.

O formato geral do arquivo pg_hba.conf é um conjunto de registos, um por linha. As linhas em branco são ignoradas, assim como qualquer texto após o **caractere de comentário #**. Os registros não podem ter mais de uma linha. Um registro é constituído por um número de campos que são separados por espaços e/ou tabulação. Os campos podem conter espaços em branco, se o valor do campo for colocado entre aspas.

Cada registro especifica um **tipo de conexão**, uma faixa de endereço IP de cliente (se for relevante para o tipo de conexão), um nome de banco de dados, um nome de usuário, bem como **o método de autenticação** a ser usado para conexões que utilizam estes parâmetros. O primeiro registro com os seguintes dados: tipo de correspondência de conexão, endereço do cliente, banco de dados solicitado e nome de usuário é usado para executar a autenticação. Não há nenhuma possibilidade de múltiplas tentativas, ou seja, se um registro é escolhido e a autenticação falhar, os registros subsequentes não são considerados. Se nenhum registro coincide, o acesso é negado. Abaixo temos um exemplo do arquivo:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.54.1/32         reject
host    all             all             0.0.0.0/0               gss
```

Perceba que a ordem pela qual as regras são listadas no pg_hba.conf arquivo é importante. A primeira regra que satisfaz a lógica é aplicada e as outras são ignoradas. Cada regra é considerada em sequência até que uma regra possa ser disparada ou a tentativa de acesso seja rejeitada, especificamente, pelo uso do método ***reject***. (falaremos dos métodos de acesso mais adiante). Vamos retomara a linha que foi adicionada ao arquivo pg_hba.conf:

```
# TYPE    DATABASE    USER          CIDR-ADDRESS    METHOD
host          all     all           0.0.0.0/0       md5
```

A regra anterior significa que uma conexão remota especifica é admitida para qualquer (all) usuário ou qualquer (all) banco de dados, em qualquer endereço de IP. É solicitada a autenticar, usando uma senha **md5** criptografada. Vejamos cada um dos parâmetros indicados:

TYPE = host significa uma conexão remota. Os valores podem ser é local (ou seja, aceita conexções através de soquetes do sistema operacional), host (conexão através de TCP/IP) ou hostssl (conexão criptografada TCP/IP).

DATABASE = all significa "para todos os bancos de dados". Outros nomes devem corresponder exatamente a um nome de banco, exceto quando tiver um sinal de mais (+) como prefixo. Neste caso, queremos indicar um "*group role*" ao invés de um único usuário. Você também pode especificar uma lista de usuários, separados por vírgulas, ou usar o símbolo @ para incluir um arquivo com uma lista de usuários.

- **USER = all** "para todos os usuários." Outros nomes devem corresponder exatamente, exceto quando prefixado com um sinal de mais (+), caso em que queremos dizer que estamos usando um "*group role*" ao invés de um único usuário. Da mesma forma que o DATABASE, pode-se usar uma lista de usuários, separados por vírgulas, ou usar o símbolo @ para incluir um arquivo com uma lista de usuários.

CIDR-ADDRESS consiste em duas partes: endereço IP/máscara de subrede. A máscara de subrede é especificada como os primeiros números (bits) do endereço IP que compõem a máscara. Assim /0 significa zero bits do endereço IP, de modo que todos os endereços IP serão comparados. Por exemplo, 192.168.0.0/24 significaria igualar os primeiros 24 bits, portanto, qualquer endereço IP do formulário 192.168.0.x poderia corresponder.

Method = trust efetivamente significa "sem autenticação". De maneira mais geral, trata de como as credenciais de login devem ser verificadas. Outros métodos de autenticação incluem GSSAPI, SSPI, LDAP, RADIUS e PAM. Conexões com o PostgreSQL também podem ser feitas usando SSL, no qual os certificados SSL do cliente em questão são utilizados para fornecer autenticação. Os principais métodos são **scram-sha-256** (o método mais robusto, disponível desde PostgreSQL 10), **md5** (o método usado em versões anteriores), **reject** sempre recusar a conexão e **trust** que sempre aceitar a conexão sem qualquer consideração às credenciais fornecidas.

CRIANDO UM BANCO DE DADOS

Agora que já aprendemos a ajustar o servidor para fazermos conexões remotas, vamos seguir em frente para entender os conceitos sobre alguns comandos ou aplicações oferecidas pelo PostgreSQL para criação das estruturas de objetos. Começaremos pelo **initdb**.

Usando a seguinte sintaxe: **initdb [option...] [--pgdata | -D] directory**, é possível criar um cluster de banco de dados. Este é uma coleção de bancos de dados que é gerenciada por uma instância do servidor. Utilizando o parâmetro PGDATA podemos especificar o diretório onde o cluster de banco de dados deve ser armazenado. Outra opção com o mesmo efeito é o uso do -D.

Veremos agora o comando CREATE DATABASE. É possível, utilizando a interface Shell (psql), rodar o comando \help e ver as características deste comando que serve basicamente para criação de uma nova base de dados. Observe a figura a seguir:

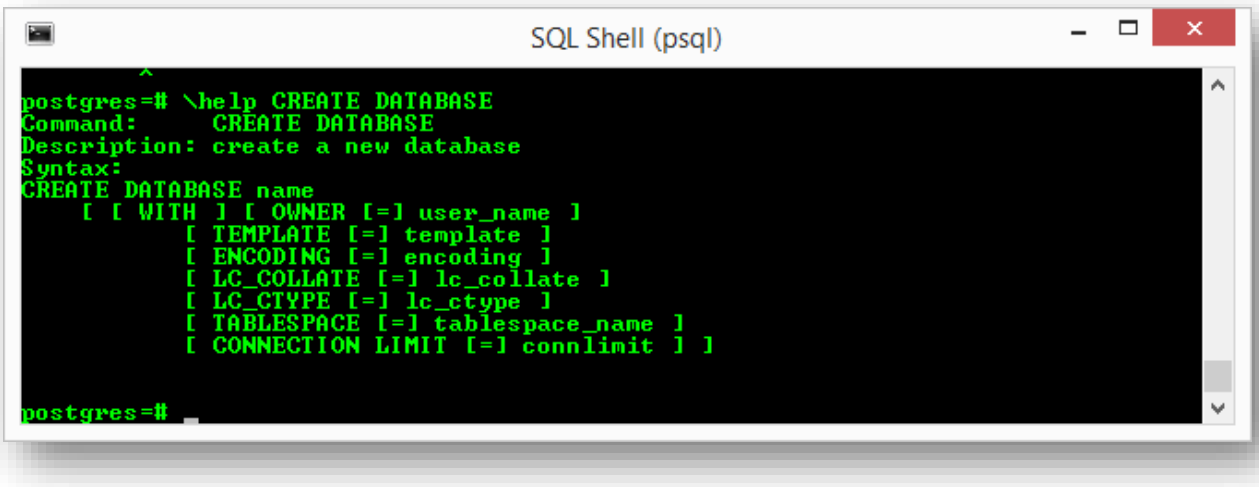


Figura 1 - Criando um banco de dados

Lembrando que, para poder executar o comando, é preciso ter o privilégio de CREATEDB. Por default, o novo banco de dados criado será basicamente um clone do banco de dados padrão do sistema denominado ***template1***. Perceba que o parâmetro utilizado para definir o padrão é **TEMPLATE** e não **clone**. É possível executar o comando usando o ***wrapper createdb***. No Windows, ele é um executável (.exe) disponível na pasta bin da instalação do PostgreSQL.

Abaixo apresentamos uma questão de prova que cobra este assunto. Usaremos a questão para detalharmos alguns aspectos do comando CLUSTER e das definições de LOCALE.

HORA DE PRATICAR!

2. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Localização refere-se ao fato de uma aplicação respeitar as preferências culturais sobre alfabetos, classificação, formatação de números etc. PostgreSQL usa o padrão ISO C e POSIX fornecidos pelo sistema operacional do servidor para aplicar as regras de localização. O suporte à localização é automaticamente inicializado quando um cluster de banco de dados é criado usando o comando

A create cluster.

B create database.

C initdb.

D ccluster.

E locale init.

Comentário. Vamos fazer alguns comentários sobre as alternativas. Na alternativa A, temos o comando CREATE CLUSTER. Ele não existe dentro do rol de comandos do SGBD. O comando que existe é o **CLUSTER**. Ele é utilizado basicamente para reordenar uma tabela fisicamente, de acordo com as informações de um dos índices. Vejamos a sintaxe do comando:

CLUSTER [VERBOSE] table_name [USING index_name]

Sobre as alternativas B e C, acabamos de definí-las acima. A alternativa D trata de um comando que não existe no Postgres. O utilitário (utility) existente é o **clusterdb**, usado como **wrapper** para o comando CLUSTER que acabamos de falar.

Na letra E, temos mais uma expressão que não existe na documentação do Postgres: ”locale init”. Mas sabemos que o suporte a *locale* é algo imprescindível dentro de qualquer SGBD. Vejamos então como o suporte a locale é tratado.

O suporte a *locale* refere-se ao fato de as aplicações respeitarem características culturais, como alfabeto, ordenação, formato de numeração ou moedas etc. O PostgreSQL usa o padrão ISO C e POSIX fornecidos pelo sistema operacional do servidor para aplicar as regras de localização.

O suporte a *Locale* é automaticamente inicializado quando um cluster de banco de dados é criado usando o initdb. Ele inicializará o cluster de banco de dados com a definição do seu ambiente de execução por default. Por isso, se o seu sistema já está definido para utilizar o idioma que você quer usar em seu cluster de banco de dados, então não há mais nada que você precise fazer. Se você quiser usar um Locale diferente (ou você não tem certeza de qual Locale está definido para sistema), você pode instruir initdb qual *locale* usar, especificando a opção --locale.

Por exemplo:

initdb --locale = sv_SE

Neste exemplo, para sistemas Unix definimos o idioma para Sueco (sv) que é falado na Suécia (SE).

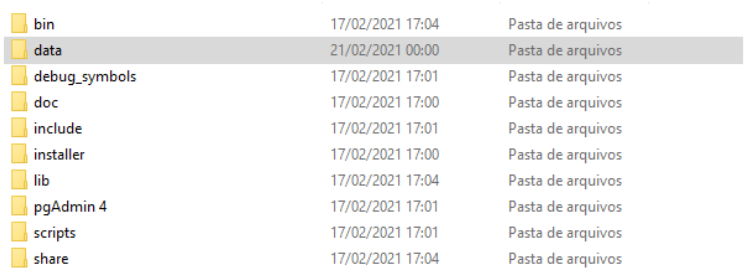
Gabarito: C

ORGANIZAÇÃO DO SERVIDOR

Esta parte do curso abrange os temas que são interessantes para o administrador de banco de dados PostgreSQL. Inclui detalhes da instalação do software, configuração do servidor, gerenciamento de usuários e bancos de dados, e tarefas de manutenção. Qualquer pessoa que executa um servidor PostgreSQL, mesmo para uso pessoal, mas especialmente em produção, deve estar familiarizado com os temas aqui abordados.

DISTRIBUIÇÃO DOS ARQUIVOS EM DIRETÓRIOS

Quando o PostgreSQL é instalado, ele cria em seu diretório raiz, normalmente, o diretório **/usr/local/pgsql (no meu caso: C:\Program Files\PostgreSQL\13)**. Este diretório mantém todos os arquivos necessários utilizados pelo PostgreSQL e é dividido em vários subdiretórios:



/bin - programas de linha de comando do PostgreSQL, como **psql**.

/data – arquivos de configuração e tabelas compartilhadas por todos os bancos de dados. Por exemplo, uma tabela **pg_shadow** compartilhada por todos os bancos de dados.

/data/base - um subdiretório é criado para cada banco de dados. Usando os comandos **du** e **ls**, os administradores podem exibir a quantidade de espaço em disco usado por cada banco de dados, tabela ou índice.

/doc - documentação do PostgreSQL.

/include – inclui os arquivos usados por várias linguagens de programação.

/lib - bibliotecas usadas por várias linguagens de programação. Este subdiretório também contém os arquivos usados durante a inicialização, e exemplos e *templates* de arquivos de configuração, que podem ser copiados para /data e modificados.

É importante ter em mente a estrutura dos arquivos dentro do diretório data, também conhecido como PGDATA. Lembrando que este diretório atua como o contêiner de disco que armazena todos os dados do cluster, incluindo os dados dos usuários e a configuração

do cluster. O diretório PGDATA está estruturado em vários arquivos e subdiretórios. Os arquivos principais são os seguintes:

- **postgresql.conf** é o arquivo de configuração principal, usado como padrão quando o serviço é iniciado.
- **postgresql.auto.conf** é o arquivo de configuração incluído automaticamente usado para armazenar configurações alteradas dinamicamente por meio de instruções SQL.
- **pg_hba.conf** é o arquivo que fornece a configuração referente às conexões de banco de dados disponíveis.
- **PG_VERSION** é um arquivo de texto que contém o número da versão principal (útil ao inspecionar o diretório para entender qual versão do cluster gerenciou o diretório PGDATA).
- **postmaster.pid** é o PID (número do processo) do cluster em execução.

Já os principais diretórios disponíveis PGDATA são os seguintes:

- **base** é um diretório que contém todos os dados dos usuários, incluindo bancos de dados, tabelas e outros objetos.
- **global** é um diretório que contém objetos de todo o cluster.
- **pg_wal** é o diretório que contém os arquivos WAL.
- **pg_stat** e **pg_stat_tmp** são, respectivamente, o armazenamento das informações estatísticas permanentes e temporárias sobre o status e a saúde do cluster.

Observação importante!! O PostgreSQL não nomeia objetos no disco, como tabelas, de forma mnemônica ou legível por humanos; em vez disso, cada arquivo é nomeado por um identificador numérico. Tente abrir as pasta acima, verifique que cada pasta contém arquivos cujos nomes são números. Como você pode ver, cada arquivo é nomeado com um identificador numérico, também conhecido como OID (Object Identifier).

Controlando status do Servidor (pg_ctl)

O pg_ctl é um utilitário para controlar o status de um cluster de banco de dados PostgreSQL, ou seja, iniciar, parar ou reiniciar o servidor back-end do PostgreSQL, ou ainda, exibir o status de um servidor em execução. Embora o servidor possa ser iniciado manualmente, o

pg_ctl encapsula tarefas como redirecionar a saída do log. Ele também fornece opções convenientes para uma parada controlada.

O pg_ctl aceita o comando a ser executado como o primeiro argumento, seguido por outros parâmentros específicos - os comandos principais são os seguintes:

- **start**, **stop** e **restart** que executam as ações correspondentes no cluster.

Observação importante

Interromper um cluster pode ser muito mais problemático do que iniciá-lo e, por esse motivo, é possível passar argumentos extras para o comando stop. Em particular, existem três maneiras de interromper um cluster:

O modo ***smart*** significa que o cluster PostgreSQL aguardará suavemente que todos os clientes conectados se desconectem e só então desligará o cluster.

O modo ***fast*** que desconectará imediatamente todos os clientes e desligará o servidor sem ter que esperar.

O modo ***immediate*** que abortará todos os processos PostgreSQL, incluindo conexões de clientes, e desligará o cluster de forma suja, o que significa que a integridade dos dados não é garantida e o servidor precisa de uma recuperação de falha no momento da inicialização.

- **status** que relata o status atual (em execução ou não) do cluster.
- **initdb** (ou ***init*** abreviadamente) que executa a inicialização do cluster, possivelmente removendo quaisquer dados existentes anteriormente.
- **reload** que faz com que o servidor PostgreSQL recarregue a configuração, o que é útil quando você deseja aplicar alterações de configuração.
- **promote** que é usado quando o cluster está rodando como um servidor subordinado (denominado ***standby***) em uma configuração de replicação e, a partir de agora, deve ser desanexado do mestre original e tornar-se independente.

Após iniciar o servidor usando comando pg_ctl, podemos visualizar alguns arquivos que compõem a configuração do servidor:

- **postmaster.pid** - a existência deste arquivo no diretório de dados que é usado para ajudar **pg_ctl** a determinar se o servidor está sendo executado ou não.
- **postmaster.opts.default** - se esse arquivo existir no diretório de dados, o pg_ctl (no modo **start**) passará o conteúdo do arquivo como opções para o comando.
- **postmaster.opts** - se esse arquivo existir no diretório de dados, o pg_ctl (no modo **restart**) passará o conteúdo do arquivo como opções para o servidor PostgreSQL. O conteúdo deste arquivo também é exibido no modo de status.
- **postgresql.conf** - este arquivo, localizado no diretório de dados, é analisado para descobrir a porta apropriada para usar com psql, quando o parâmetro -w é informado.

PROCESSOS DO POSGRESQL

O PostgreSQL iniciará **vários processos diferentes na inicialização**. Esses processos são responsáveis por manter o cluster em bom estado de funcionamento, bem como por observar e instruir o cluster nas suas ações. Esta seção fornece uma visão geral dos principais processos que você pode encontrar em um cluster em execução, permitindo que você reconheça cada um deles e seus respectivos propósitos.

Se você executar a seguinte consulta sql num terminal Shell com um usuário administrador, você verá os processos em execução para o banco de dados conectado.

select datname,pid, wait_event, backend_type from pg_stat_activity;

```
estrategia=# Select datname,pid, wait_event, backend_type from pg_stat_activity order by pid;
  datname   |  pid  |     wait_event      |         backend_type
------------+-------+---------------------+------------------------------
            |  2344 | WalWriterMain       | walwriter
            |  6924 | AutoVacuumMain      | autovacuum launcher
 estrategia | 11420 |                     | client backend
            | 13096 | LogicalLauncherMain | logical replication launcher
            | 17488 | CheckpointerMain    | checkpointer
            | 18220 | BgWriterMain        | background writer
(6 rows)
```

Interessante perceber que esses números de processos são os mesmos que aparecem no sistema operacional. Vejam a lista de processos do SO associadas as Postgres.

```
postgres                2256   8   3  401 4527356  20000   2776
  postgres              2344   8   2  346 4479296  11660   2588
  postgres              6924   8   2  346 4480320   8248   3180
  postgres             11420   8   3  401 4489160  14532   4956
  postgres             13096   8   2  346 4480320   7960   3184
  postgres             15088   8   3  343 4483016   7256   2304
  postgres             15972   8   2  345 4479296   7676   2300
  postgres             17488   8   2  346 4479296   8888   2668
  postgres             18220   8   2  345 4479296   8724   2632
```

O processo postmaster cujo PID é igual a 2256 é aquele que possui todos os outros subprocessos. Os principais processos de manutenção são os seguintes:

- ***checkpointer*** é um processo responsável por executar os pontos de verificação, que são pontos no tempo em que o banco de dados garante que todos os dados sejam realmente armazenados de forma persistente no disco.

- ***background writer*** é responsável por ajudar a enviar os dados da memória para o armazenamento permanente.

- ***walwriter*** é responsável por gravar os Logs Write-Ahead (WALs), os logs que são necessários para garantir a confiabilidade dos dados, mesmo no caso de uma falha do banco de dados.
- ***stats collector*** é um processo que monitora a quantidade de dados que o PostgreSQL está manipulando, armazenando-os para uma análise e decisões posteriores, como decidir quais índices usar para satisfazer uma consulta.
- ***logical replication launcher*** é um processo responsável por lidar com a replicação lógica.

CATÁLOGO DO SISTEMA

Os catálogos do sistema são o lugar onde um SGBD armazena os metadados de um esquema relacional, tais como informações sobre tabelas e colunas, e as informações sobre a estruturação interna. Os catálogos do sistema do PostgreSQL são tabelas regulares. Você pode eliminar e recriar as tabelas, as colunas, adicionar valores de inserção e atualização nestas tabelas, contudo, é preciso ter cuidado para não danificar seu sistema.

Normalmente, não se deve alterar os catálogos do sistema à mão, há sempre comandos SQL para fazer isso. (Por exemplo, CREATE DATABASE insere uma linha para o **catálogo pg_database** e realmente cria o banco de dados no disco). Há algumas exceções para operações particularmente esotéricas, como a adição de métodos de acesso a índice. Outros exemplos de visões presentes no catálogo de sistemas são:

Nome	Propósito
pg_attrdef	Armazena os valores default das colunas. Apenas as colunas que explicitamente especificam o valor default (quando a tabela é criada ou a coluna é adicionada) terão uma entrada aqui. A principal informação sobre colunas é armazenada em **pg_attribute** (ou tabela do catálogo).
pg_attribute	Descrição das colunas das tabelas

pg_cast	Grava o caminho de conversão entre os dados, tanto as possibilidades disponíveis nativamente no sistema, quanto aquelas que são definidas pelo usuário.
pg_constraint	Armazena informações sobre as restrições de integridade: check constraints, unique constraints, primary key constraints, foreign key constraints.
pg_depend	Armazena os relacionamentos de dependência entre os objetos do banco de dados. Esta informação permite ao comando DROP descobrir quais outros objetos devem ser removidos pelo DROP CASCADE. Também permite impedir a remoção deles no caso de DROP RESTRICT.
pg_namespace	Lista os schemas de um determinado banco de dados.
pg_proc	Registra as informações sobre as funções e as procedures. A tabela contém dados para funções agregadas, bem como funções simples.
pg_type	Grava informações sobre os tipos de dados. Os tipos básicos e tipos enum (escalares) são criados com CREATE TYPE. Já os domínios são criados com CREATE DOMAIN. Um tipo composto é criado automaticamente para cada tabela no banco de dados, para representar a estrutura de linha da tabela. Também é possível criar tipos compostos com CREATE TYPE AS.

Para finalizar gostaria de falar sobre a visão pg_settings que fornece acesso aos parâmetros de run-time do servidor. É essencialmente uma interface alternativa para os comandos SHOW e SET usados para manipular os parâmetros de sistema. Ela também fornece acesso a algumas informações sobre cada parâmetro que não estão diretamente disponíveis no comando SHOW, como valores mínimos e máximos.

A visão pg_settings não pode ter linhas inseridas ou apagadas, mas pode ser atualizada. Uma atualização aplicada a uma linha da visão é equivalente a executar o comando SET no parâmetro. A alteração afeta apenas o valor utilizado pela **sessão atual**. Se uma atualização for emitida dentro de uma transação que é abortada mais tarde, os efeitos do comando UPDATE desaparecem quando a transação é revertida. Uma vez que a transação sofre COMMIT, os efeitos vão persistir até o final da sessão, a menos que substituída por outro comando UPDATE ou SET.

Vamos a seguir fazer uma questão a respeito do assunto.

HORA DE PRATICAR!

1. BANCA: FCC ANO: 2014 ÓRGÃO: TJ-AP PROVA: ANALISTA JUDICIÁRIO - BANCO DE DADOS - DBA

Um dos itens da administração do sistema gerenciador de bancos de dados PostgreSQL (V.9.3.4) refere-se a gerenciar informações sobre os bancos de dados por ele controlados. O PostgreSQL contém algumas visões que auxiliam nessa tarefa, dentre elas, a visão pg_settings que contém dados sobre

A os parâmetros run-time do servidor.

B estatísticas das tabelas do servidor.

C os usuários dos bancos de dados.

D a lista de bloqueios impostos.

E a lista das funções presentes no banco de dados.

Comentário. A questão pergunta sobre os dados que são armazenados na visão pq_setting. Sabemos o que são os dados de run-time, mas quais exatamente? Segue uma tabela com os parâmetros, os tipos e a descrição das informações contidas na visão.

Nome	Tipo	Descrição
name	text	Nome do parâmetro de configuração
setting	text	O valor atual do parâmetro
unit	text	Unidade implícita do parâmetro
category	text	Grupo lógico do parâmetro
short_desc	text	Uma breve descrição do parâmetro
extra_desc	text	Descrição adicional, mais detalhada do parâmetro
context	text	Contexto necessário para definir o valor do parâmetro.
vartype	text	Tipo de parâmetro (bool, enum, integer, real, ou string)
source	text	Fonte do valor do parâmetro atual
min_val	text	Valor mínimo permitido do parâmetro (null para valores não-numéricos)
max_val	text	Valor máximo permitido do parâmetro (null para valores não-numéricos)
enumvals	text[]	Os valores permitidos de um parâmetro de enum (NULL para valores não-ENUM)
boot_val	text	O valor do parâmetro assumido na inicialização do servidor, se o parâmetro não for definido de outro modo
reset_val	text	Valor que o RESET iria redefinir para o parâmetro na sessão atual

Gabarito: A

Para saber mais sobre outros elementos contidos nas quase 100 tabelas presentes no catálogo de dados, acesse: https://www.postgresql.org/docs/current/catalogs.html.

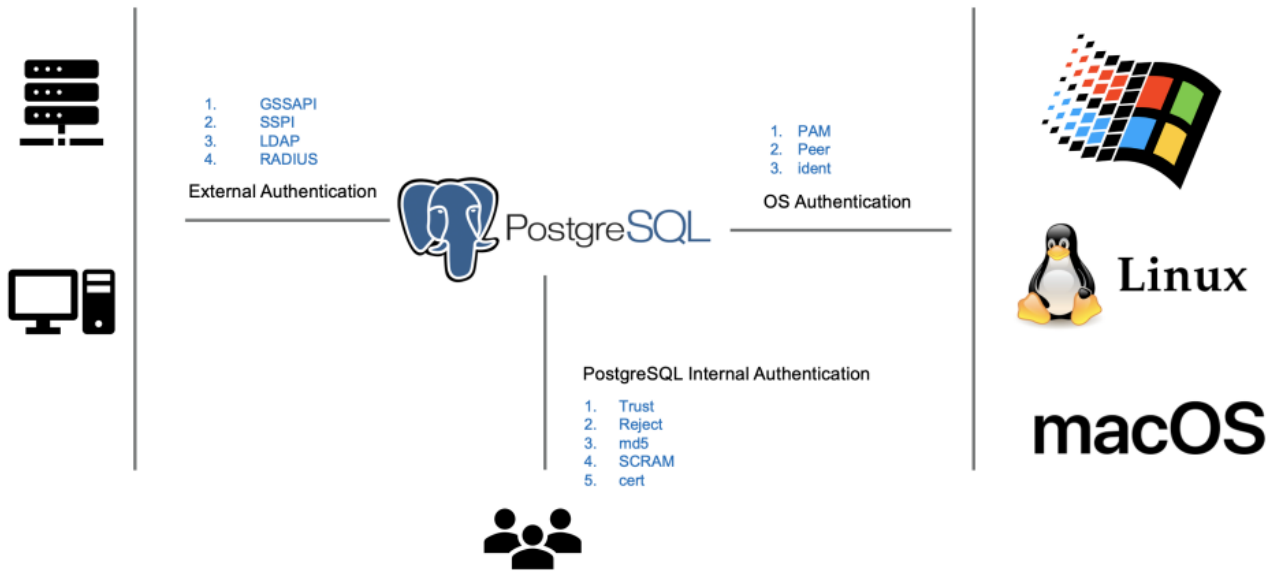
AUTENTICAÇÃO DO CLIENTE

A autenticação é o processo pelo qual o servidor de banco de dados estabelece a identidade do cliente e, por extensão, determina se o aplicativo cliente (ou o usuário que executa o aplicativo cliente) tem permissão para se conectar com o banco de dados que foi solicitado.

O PostgreSQL oferece diferentes métodos de autenticação para o cliente. O método utilizado para autenticar uma conexão cliente em particular pode ser selecionado com base no endereço de host do cliente, no banco de dados e nos usuários.

MÉTODOS DE AUTENTICAÇÃO

Apresentamos abaixo a lista de possibilidades de métodos de autenticação disponíveis no PostgreSQL. Lembra que falamos que a autenticação acontece com a verificação das linhas do arquivo pg_hba.conf? Já vimos um exemplo de linha, agora vamos detalhar os métodos de autenticação (METHOD).



Autenticação	Descrição
trust	Permite a conexão incondicionalmente. Este método permite a qualquer pessoa se conectar ao servidor de banco de dados

	PostgreSQL e se autenticar com o usuário que desejarem, sem a necessidade de senha ou qualquer outra autenticação.
reject	Rejeita qualquer conexão incondicionalmente. Isso é útil para "filtragem" de certos hospedeiros de um grupo, por exemplo, uma linha de reject poderia bloquear um host específico para conexão, enquanto uma linha mais tarde permite que os hosts restantes possam se conectar.
scram-sha-256	Executa a autenticação SCRAM-SHA-256 para verificar a senha do usuário.
md5	Exige que o cliente forneça uma senha MD5 para autenticação.
password	Exige que o cliente forneça uma senha não criptografada para autenticação. Uma vez que a senha é enviada em texto simples através da rede, não deve ser usado em redes não confiáveis.
gss	Usa GSSAPI para autenticar o usuário. Este método só está disponível para conexões TCP/IP.
sspi	Usa SSPI para autenticar o usuário. O método só está disponível no Windows.
ident	Obtém o nome do usuário do sistema operacional do cliente entrando em contato com o servidor e verifica se ele corresponde ao nome do usuário do banco de dados. A autenticação ident só pode ser utilizada em conexões TCP/IP. Quando especificado para conexões locais, peer authentication será utilizado em seu lugar.
peer	Obtém o nome de usuário do sistema operacional do cliente e verifica se ele corresponde ao nome do usuário solicitado do banco de dados. Isto só está disponível para conexões locais.
ldap	Autentica o cliente usando um servidor LDAP.
radius	Autentica usando um servidor RADIUS.
cert	Autentica usando certificados de cliente SSL.

PAM	Autentica utilizando o serviço Pluggable Authentication Modules (PAM) fornecido pelo sistema operacional.

Vejamos uma questão de uma prova anterior sobre o assunto ...

HORA DE PRATICAR!

(Ministério da Economia – Infraestrutura - 2020) Acerca de PostgreSQL, julgue o item a seguir.

99 O arquivo pg_hba.conf controla a autenticação de usuários e hosts, de acordo com a sua origem; quando o parâmetro trust é utilizado na opção auth-metod, são aceitas conexões desde que o usuário se autentique usando senha.

Comentários:

.

O parâmetro ***trust*** quer dizer que não será solicitada qualquer senha para conectar ao banco de dados. Então, qualquer pessoa com acesso à sua rede poderá se conectar ao servidor PostgreSQL sem nenhuma restrição. Se a sua rede for restrita como uma turma de um curso ou um ambiente de desenvolvimento interno de empresa não há riscos, porém, é necessário possuir um ambiente de rede interno isolado.

Gabarito Errado.

PERMISSÕES E PAPÉIS (ROLES)

O PostgreSQL administra as permissões de acesso ao banco de dados utilizando o conceito de papéis. Podemos pensar em um papel como uma base de dados de usuário ou um grupo de usuários da base de dados, dependendo de como o papel é configurado. Roles (papéis) podem possuir permissões sobre objetos de banco de dados (por exemplo, tabelas) e podem atribuir privilégios sobre os objetos a outros papéis, visando controlar quem tem acesso a quais objetos.

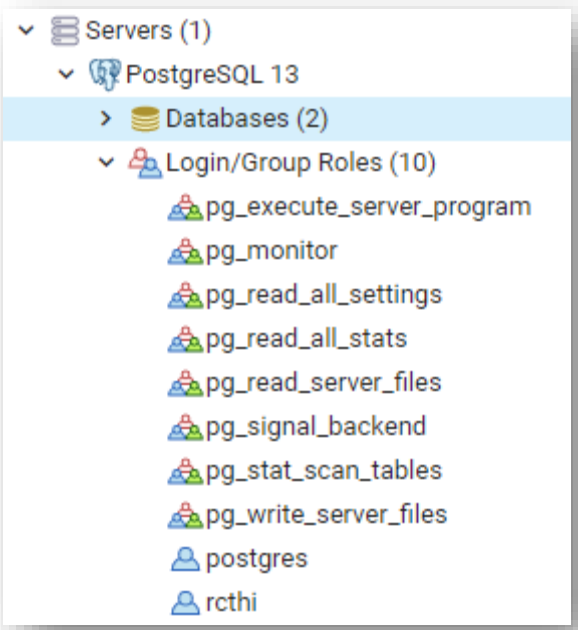


Figura 1 - Grupos de Usuários no Postgres

O conceito de papéis une os conceitos de "usuários" e "grupos". Em versões do PostgreSQL anteriores a 8.1, os usuários e os grupos eram tipos de entidades distintas, mas agora há apenas papéis. Qualquer papel pode atuar como um usuário, um grupo ou ambos. A seguir,

descreveremos como criar e gerenciar papéis, e introduziremos o sistema de privilégios do SGBD.

É importante entender que um papel (ou função) é definido no nível do cluster. Isso significa que o mesmo papel pode ter privilégios e propriedades diferentes dependendo do banco de dados que está usando (por exemplo, ter permissão para se conectar a um banco de dados e não a outro).

PAPÉIS NO BANCO DE DADOS

Papéis de bancos de dados são conceitos completamente separados dos usuários do sistema operacional. Na prática, pode ser conveniente manter uma correspondência, mas isso não é necessário. Papéis de bancos de dados são globais através de uma instalação de um cluster de banco de dados (e não por base de dados individual). Para criar um papel, use o comando CREATE ROLE SQL:

```
-- Role: userpadrao
-- DROP ROLE userpadrao;

CREATE ROLE userpadrao WITH
  LOGIN
  NOSUPERUSER
  INHERIT
  NOCREATEDB
  NOCREATEROLE
  NOREPLICATION
  CONNECTION LIMIT 10
  ENCRYPTED PASSWORD 'SCRAM-SHA-256$

COMMENT ON ROLE userpadrao IS 'Esse
```

CREATE ROLE nome;

O **nome** segue as regras usadas para identificadores SQL: sem caracteres especiais ou aspas duplas. (Na prática, normalmente você vai querer adicionar opções, como LOGIN, para o comando). Para remover uma função existente, use o comando análogo DROP ROLE:

DROP ROLE nome;

Por conveniência, os executáveis **createuser** e **dropuser** são fornecidos como *wrappers* em torno esses comandos SQL, que podem ser chamados a partir da linha de comando shell.

Para determinar o conjunto de papéis existentes, você pode examinar o catálogo do sistema pg_roles, por exemplo:

SELECT rolname FROM pg_roles;

O programa psql \du é um meta-comando útil para listar os papéis existentes. Abaixo temos um exemplo da criação de um ROLE e da execução do comando \du no psql. Logo em seguida, faremos uma questão para fixação do conteúdo.

```
estrategia=> \du
                                   List of roles
  Role name  |                         Attributes                         | Member of
-------------+------------------------------------------------------------+-----------
 postgres    | Superuser, Create role, Create DB, Replication, Bypass RLS | {}
 rcthi       | Superuser, Create role, Create DB                          | {}
 userpadrao  | 10 connections                                             | {}
 userteste01 | Create DB                                                 +| {}
             | Password valid until 2021-03-01 11:50:51-03                |
 userteste02 |                                                            | {}
 userteste03 | Cannot login                                               | {}
```

Figura 2 - Uma lista de usuários

Dica: É uma boa prática criar uma ROLE que tem os privilégios CREATEDB e CREATEROLE, mas não é um SUPERUSER, e depois usar essa ROLE para todo o gerenciamento de rotinas dos bancos de dados e papéis. Essa abordagem evita os perigos de operar como um superusuário em tarefas que realmente não necessitam dele.

Quando um usuário tenta se conectar a um banco de dados, o PostgreSQL verifica as credenciais de login e algumas outras propriedades do usuário para garantir que ele tenha permissão para efetuar login e tenha credenciais válidas. As principais opções que permitem manipular e gerenciar as tentativas de login são as seguintes:

- **PASSWORD** ou **ENCRYPTED PASSWORD** são opções equivalentes e permitem definir a senha de login para a função. Ambas as opções existem para compatibilidade com versões anteriores do PostgreSQL, mas hoje em dia, o cluster sempre armazena as senhas de funções de forma criptografada, portanto, o uso de ENCRYPTED PASSWORD não adiciona nenhum valor à PASSWORD opção.
- **PASSWORD NULL** força explicitamente uma senha nula (não vazia), evitando que o usuário efetue login com qualquer senha. Esta opção pode ser usada para negar autenticação baseada em senha.
- **CONNECTION LIMIT <n>** permite que o usuário abra não mais do que <n> conexões simultâneas ao cluster, sem qualquer consideração a um banco de dados específico. Isso geralmente é útil para evitar que um usuário desperdice recursos no cluster.
- **VALID UNTIL** permite que você especifique um instante (no futuro) quando a função expirará.
- **IN ROLE** serve para criar um papel como membro de um outro especificado.

Apenas para complementar nosso conhecimento a respeito de ROLES, apresentamos abaixo a sintaxe completa do comando. Observe que a maioria dos parâmetros é de entendimento intuitivo.

```
CREATE ROLE name [ [ WITH ] option [ ... ] ]
As opções podem ser as seguintes:
    SUPERUSER | NOSUPERUSER
  | CREATEDB | NOCREATEDB
  | CREATEROLE | NOCREATEROLE
  | CREATEUSER | NOCREATEUSER
  | INHERIT | NOINHERIT
  | LOGIN | NOLOGIN
  | REPLICATION | NOREPLICATION
  | CONNECTION LIMIT <número de conexões>
  | [ ENCRYPTED | UNENCRYPTED ] PASSWORD 'password'
  | VALID UNTIL 'timestamp'
  | IN ROLE role_name [, ...]
  | IN GROUP role_name [, ...]
  | ROLE role_name [, ...]
  | ADMIN role_name [, ...]
  | USER role_name [, ...]
  | SYSID uid
```

ORGANIZAÇÃO DO BANCO DE DADOS E TABELAS

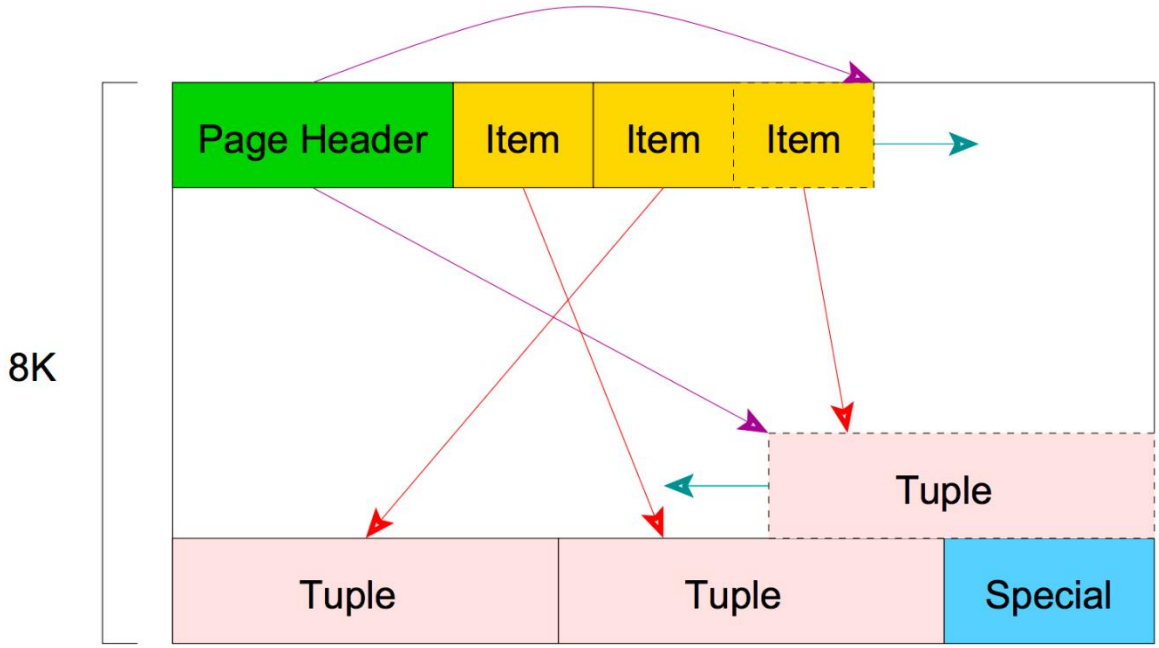
Quando você cria um banco de dados, além de esquemas public criados pelo usuário, cada banco de dados contém um esquema pg_catalog, que contém as tabelas do sistema e todos os tipos de dados, funções e operadores integrados. O pg_catalog é efetivamente parte do caminho de pesquisa, portanto, você não precisa usar o prefixo ao consultar as tabelas do sistema.

Uma vez conectado ao seu banco de dados via psql, você pode listar a lista das tabelas do sistema via:

```
\dt pg_catalog.*
```

Cada tabela é armazenada como uma matriz de páginas de tamanho fixo (geralmente 8 KB). Em uma tabela, todas as páginas são logicamente equivalentes, portanto, um determinado item (linha) pode ser armazenado em qualquer página.

A estrutura usada para armazenar a tabela é um **arquivo heap**. Os arquivos heap são listas de registros não ordenados de tamanho variável. O arquivo heap é estruturado como uma coleção de páginas (ou bloco), cada uma contendo uma coleção de itens. O termo item se refere a uma linha armazenada em uma página. Uma estrutura de página se parece com o seguinte:



Ela contém alguns cabeçalhos que não iremos cobrir, mas fornecem informações sobre a checksum, início do espaço livre, fim do espaço livre, ... Itens após os cabeçalhos são um array identificado composto de pares (deslocamento, comprimento) apontando para os itens reais.

Como um identificador de item nunca é movido até que seja liberado, seu índice pode ser usado a longo prazo para fazer referência a um item, mesmo quando o próprio item é movido pela página para compactar o espaço livre. Um ponteiro para um item é denominado CTID (ItemPointer), criado pelo PostgreSQL, consiste em um número de página e no índice de um identificador de item.

Os próprios itens são armazenados no espaço alocado ao contrário (de trás para frente) do final do espaço não alocado para o início. Para resumir, dentro de uma página, os ponteiros para a linha são armazenados no início e as tuplas (linhas) são armazenadas no final da página.

Você pode acessar o CTID de uma linha adicionando ctid nas colunas selecionadas:

```
SELECT ctid, * from bar;
```

HORA DE PRATICAR!

(Ministério da Economia – Desenvolvimento de Sistemas - 2020) Acerca de sistemas gerenciadores de banco de dados, julgue o item subsequente.

No PostgreSQL, a principal unidade de armazenamento é uma tabela, sendo as tabelas armazenadas em arquivos de heap.

Comentários: No postgreSQL, cada tabela é armazenada como uma matriz de páginas de tamanho fixo (geralmente 8 KB). Em uma tabela, todas as páginas são logicamente equivalentes, portanto, um determinado item (linha) pode ser armazenado em qualquer página.

A estrutura usada para armazenar a tabela é um **arquivo heap**. Os arquivos heap são listas de registros **não ordenados de tamanho variável**. O arquivo heap é estruturado como uma coleção de páginas (ou bloco), cada uma contendo uma coleção de itens. O termo item se refere a uma linha armazenada em uma página.

Gabarito Certo.

Agora que já conhecemos os perfis de usuários e a forma como o servidor está organizado vamos brincar com os comandos de criação e manipulação dos dados.

POSTGRESQL INTERACTIVE TERMINAL: PSQL

Como a maioria dos sistemas de banco de dados com capacidade de operar em rede, o PostgreSQL se encaixa no paradigma cliente-servidor. O coração do PostgreSQL é a infraestrutura do servidor ou o processo ***postmaster***. Ele é chamado de "back-end", porque não deve interagir diretamente com o usuário. Em vez disso, ele pode ser conectado a diferentes tipos de clientes.

Quando você iniciar o serviço do SGBD PostgreSQL, o processo *postmaster* começa a ser executado em segundo plano, escutando uma porta específica TCP/IP que é utilizada para conexões de clientes. A menos que explicitamente configurado, ***postmaster*** irá escutar a porta **5432**.

Existem várias interfaces disponíveis por meio da quais os clientes podem se conectar ao processo ***postmaster***. Os exemplos a seguir utilizam o ***psql***, um aplicativo cliente portátil, de fácil acesso e distribuído do PostgreSQL.

Esta parte da aula detalha conceitos do psql, como criar e usar tabelas, e como recuperar e gerenciar dados dentro dessas tabelas. Ela também aborda subconsultas SQL e visões.

O cliente ***psql*** é um aplicativo de linha de comando distribuído integrado ao PostgreSQL. É muitas vezes chamado de monitor interativo ou terminal interativo. Com ele, você tem uma ferramenta simples, mas poderosa, com a qual é possível interagir diretamente com o servidor PostgreSQL.

Numa parte anterior da nossa aula, mostramos alguns comandos que podem ser utilizados para gerenciar o SGBD pela interface do psql. Primeiramente apresentamos como criar uma conexão com um banco de dados utilizando esta interface. Vimos que precisamos definir o servidor, o banco de dados, a porta, e um usuário e uma senha. Depois apresentamos como usar o comando **\help** para visualizar as opções de outro comando disponível no SGBD. Por fim, apresentamos detalhes para criação de objetos DATABASE e do ROLES.

Neste momento, manteremos nosso foco nos comandos relativos à **criação e à manipulação das bases de dados PostgreSQL, conhecidos como DDL e DML**. Antes de iniciarmos com o psql, vamos fazer uma breve sinopse de quatro comandos essenciais psql (slash commands): \h para ajuda, \? para obter ajuda sobre comandos específicos do psql, \g para executar consultas e \q para realmente sair do psql, uma vez você tenha terminado de executar seus comandos.

Cada comando específico do psql é prefixado por uma barra invertida; daí o termo "slash commands" utilizado anteriormente. Para obter uma lista completa de comandos de barra e uma breve descrição das suas funções, digite \? na linha de comando psql e pressione enter.

Alguns desses comandos já foram cobrados em provas de concurso. Vejam a questão abaixo:

HORA DE PRATICAR!

1. BANCA: FCC ANO: 2013 ORGÃO: TRT - 12ª Região (SC) PROVA: Analista Judiciário - Tecnologia da Informação

O comando em SQL capaz de serializar dados de uma tabela para um arquivo em disco, ou efetuar a operação contrária, transferindo dados de um arquivo em disco para uma tabela de um banco de dados, é o comando:

(a) COPY.

(b) TRANSFER.

(c) SERIALIZE.

(d) FILE TRANSFER.

(e) EXPORT.

Comentário. Analisando cada uma das alternativas. O comando **copy** permite escrever o conteúdo de uma tabela em um arquivo ASCII ou carregar uma tabela a partir de um arquivo. Esses arquivos podem ser usados para backup ou para transferência de dados entre o PostgreSQL e outras aplicações. É possível usar as várias STDIN e STDOUT para especificar que os dados a serem transferidos sejam inseridos ou exibidos na linha de comando. Podemos ainda usar o modificar DELIMITERS para especificar o caractere que separa as colunas dentro de cada linha do arquivo. O padrão é um caractere de tabulação para arquivos em formato de texto ou uma vírgula no formato CSV. Este delimitador deve ser um único caractere de um byte. Esta opção não é permitida quando usando o formato binário. Vejam que já achamos nossa resposta.

As demais alternativas apresentam termos que não estão no rol de palavras reservadas ou comandos do postgreSQL. Dentre os comandos utilizados para exportar arquivo, existe uma lista, além do COPY, responsável pela manipulação de arquivos LOB (large objects). Vejam a lista na figura abaixo:

```
COPY, LARGE OBJECT
  \copy ...                          perform SQL COPY with data stream to the client host
  \lo_export LOBOID FILE             LOBOID FILE
  \lo_import FILE [COMMENT]          FILE [COMMENT]
  \lo_list
  \lo_unlink LOBOID                  large object operations
```

Gabarito: A.

Vejam que vale a pena conhecer os principais comandos slash. Optamos por colocar abaixo uma lista destes comandos com suas respectivas descrições.

```
GENERAL:
  \c[onnect] [DBNAME|- USER|- HOST|- PORT|-]   connect to new database
  \cd [DIR]                                    change the current working directory
  \encoding [ENCODING]                         show or set client encoding
  \h [NAME]                                    help on syntax of SQL commands, * for all commands
  \set [NAME [VALUE]]                          set internal variable, or list all if no parameters
  \timing                                      toggle timing of commands (currently off)
  \unset NAME                                  unset (delete) internal variable
  \prompt [TEXT] NAME                          prompt user to set internal variable
  \! [COMMAND]                                 execute command in shell or start interactive shell

QUERY BUFFER:
  \e [FILE]                                    edit the query buffer (or file) with external editor
  \g [FILE]                                    send query buffer to server (and results to file or |pipe)
  \p                                           show the contents of the query buffer
  \r                                           reset (clear) the query buffer
  \w FILE                                      write query buffer to file

INPUT/OUTPUT:
  \echo [STRING]                               write string to standard output
  \i FILE                                      execute commands from file
  \o [FILE]                                    send all query results to file or |pipe
  \qecho [STRING]                              write string to query output stream (see \o)
```

2. BANCA: CESPE - Oficial Técnico de Inteligência/Área 9/2018

Julgue o próximo item, a respeito de conceitos e comandos PostgreSQL e MySQL.

No programa psql do PostgreSQL, a instrução \h permite mostrar o histórico de comandos SQL na sessão atual.

Certo

Errado

Comentário: O psql é um cliente no modo terminal do PostgreSQL. Permite digitar comandos interativamente, submetê-los para o PostgreSQL e ver os resultados. Como alternativa, a entrada pode vir de um arquivo. Além disso, disponibiliza vários meta-comandos e diversas funcionalidades semelhantes às do interpretador de comandos (shell), para facilitar a criação de scripts e automatizar muitas tarefas.

Ao digitar \help (ou \h) [comando], o sistema fornece ajuda de sintaxe para o comando SQL especificado. Se não for especificado o comando, então o psql listará todos os comandos para os quais existe ajuda de sintaxe disponível. Se o comando for um asterisco ("*"), então será mostrada a ajuda de sintaxe para todos os comandos SQL. Desta forma, a alternativa está incorreta.

Gabarito: E

COMANDOS DDL

Vamos agora passar a analisar as peculiaridades dos comandos DDL do PostgreSQL, começando pelo principal comando desta categoria: o CREATE TABLE.

CREATE DATABASE

O comando CREATE DATABASE cria um banco de dados PostgreSQL. Para criar um banco de dados, você deve ser um superusuário ou ter o privilégio especial CREATEDB. O comando possui a seguinte sintaxe:

```
CREATE DATABASE nome                          # define o nome do banco de dados
  [ [ WITH ] [ OWNER [=] user_name ]          # define o nome da role que vai possuir o banco de dados.
       [ TEMPLATE [=] template ]              # define o nome do modelo a partir do qual será criado o novo BD
       [ ENCODING [=] encoding ]              # define a codificação do conjunto de caracteres
       [ LC_COLLATE [=] lc_collate ]
       [ LC_CTYPE [=] lc_ctype ]
       [ TABLESPACE [=] tablespace ]
       [ CONNECTION LIMIT [=] connlimit ] ]
```

Vejamos um questão sobre o assunto ...

HORA DE PRATICAR!

(Ministério da Economia – Infraestrutura - 2020) Acerca de PostgreSQL, julgue o item a seguir.

97 Com PostgreSQL é possível criar um banco de dados utilizando-se outro como template: se o nome do banco de dados que já exista for database01 e o nome do banco a ser criado for database02, então o comando a ser usado será o seguinte.

CREATE DATABASE database02 CLONE database01;

Comentários: CREATE DATABASE cria um banco de dados PostgreSQL. Para criar um banco de dados, você deve ser um superusuário ou ter o privilégio especial CREATEDB.

Por padrão, o novo banco de dados será criado clonando o modelo de banco de dados do sistema template01. Um modelo diferente pode ser especificado escrevendo o nome do TEMPLATE. Em particular, ao escrever TEMPLATE template0, você pode criar um banco de dados virgem contendo apenas os objetos padrão predefinidos por sua versão do PostgreSQL. Assim, o comando da questão deveria ser:

CREATE DATABASE database02 TEMPLATE database01;

Não existe a opção de clone no comando.

Gabarito Errado.

CREATE TABLE

O comando CREATE TABLE irá criar uma tabela nova, inicialmente vazia, no banco de dados. A tabela será de propriedade do usuário que emite o comando.

Se o nome do esquema for fornecido (por exemplo, CREATE TABLE esquema.tabela ...), então a tabela será criada no esquema especificado. Caso contrário, ela é criada no esquema corrente. As tabelas temporárias são criadas em um esquema especial, portanto não é possível definir o nome do esquema. O nome da tabela deve ser distinto do nome de qualquer outra **tabela**, **sequência**, **índice**, **visão** ou **tabela estrangeira** no mesmo esquema.

O CREATE TABLE também cria automaticamente um tipo de dados que representa o tipo de composto correspondente a uma linha da tabela. Portanto, uma tabela não pode ter o mesmo nome de um tipo de dado existente no mesmo esquema.

As cláusulas opcionais especificam as restrições (testes) que as linhas novas ou modificadas devem satisfazer durante a inserção ou uma operação de atualização. Uma restrição é um objeto SQL que ajuda a definir o conjunto de valores válidos na tabela de várias maneiras.

Existem duas formas para definir restrições: restrições **de tabela** e restrições **de coluna**. A restrição de coluna é definida como parte da definição da coluna. A restrição de tabela não está vinculada a uma determinada coluna, e pode abranger mais de uma coluna. Toda restrição de coluna também pode ser escrita como uma restrição de tabela; a restrição de coluna é somente uma notação conveniente para uso quando a restrição afeta apenas uma coluna.

Veja abaixo a sintaxe do comando:

```
CREATE [ [ GLOBAL | LOCAL ] { TEMPORARY | TEMP } | UNLOGGED ] TABLE [ IF NOT EXISTS ] table_name ( [
  { column_name data_type [ COLLATE collation ] [ column_constraint [ ... ] ]
    | table_constraint
    | LIKE source_table [ like_option ... ] }
    [, ... ]
] )
[ INHERITS ( parent_table [, ... ] ) ]
[ WITH ( storage_parameter [= value] [, ... ] ) | WITH OIDS | WITHOUT OIDS ]
[ ON COMMIT { PRESERVE ROWS | DELETE ROWS | DROP } ]
[ TABLESPACE tablespace_name ]
```

Primeiramente, as opções Global e Local existem para manter a compatibilidade com versões anteriores. A definição de uma tabela como temporária faz com que ela seja removida ao final da sessão. Uma tabela pode não usar o log (write-ahead log) para garantir a integridade, basta utilizar o UNLOGGED na criação dela. Observe que as restrições definidas acima para tabelas podem ser detalhadas abaixo:

```
[ CONSTRAINT constraint_name ]
{ CHECK ( expression ) [ NO INHERIT ] |
  UNIQUE ( column_name [, ... ] ) index_parameters |
  PRIMARY KEY ( column_name [, ... ] ) index_parameters |
  EXCLUDE [ USING index_method ] ( exclude_element WITH operator [, ... ] ) index_parameters [ WHERE ( predicate ) ] |
  FOREIGN KEY ( column_name [, ... ] ) REFERENCES reftable [ ( refcolumn [, ... ] ) ]
    [ MATCH FULL | MATCH PARTIAL | MATCH SIMPLE ] [ ON DELETE action ] [ ON UPDATE action ] }
[ DEFERRABLE | NOT DEFERRABLE ] [ INITIALLY DEFERRED | INITIALLY IMMEDIATE ]
```

CREATE SEQUENCE

CREATE SEQUENCE cria um gerador de números sequenciais. Isso envolve a criação e a inicialização de uma nova tabela especial de uma única linha. O gerador será de propriedade do usuário que emite o comando.

Se um nome de esquema for fornecido, então a sequência é criada no esquema especificado. Caso contrário, ela é criada no esquema corrente. Sequências temporárias são criadas em um esquema especial, portanto o nome do esquema não pode ser fornecido quando se cria uma sequência temporária. O nome da sequência deve ser distinto do nome de qualquer outra sequência, tabela, índice, visão ou tabela estrangeira no mesmo esquema.

Depois de uma sequência ser criada, você usa as funções ***nextval***, ***currval*** e ***setval*** para operar a sequência. Veja a sintaxe do comando abaixo:

```
CREATE [ TEMPORARY | TEMP ] SEQUENCE name [ INCREMENT [ BY ] increment ]
    [ MINVALUE minvalue | NO MINVALUE ] [ MAXVALUE maxvalue | NO MAXVALUE ]
    [ START [ WITH ] start ] [ CACHE cache ] [ [ NO ] CYCLE ]
    [ OWNED BY { table_name.column_name | NONE } ]
```

CREATE INDEX

O comando CREATE INDEX constrói um índice na(s) coluna(s) especificada(s) da relação, que pode ser uma tabela ou uma visão materializada. Os índices são utilizados principalmente para melhorar o desempenho do banco de dados (embora o uso inadequado possa resultar em um desempenho mais lento).

O(s) campo(s) chave(s) para o índice são especificados como nomes de coluna, ou, alternativamente, como expressões escritas entre parênteses. Vários campos podem ser especificados, se o método de índice suportar índices com múltiplas colunas.

Um campo de índice pode ser uma expressão calculada a partir dos valores de uma ou mais colunas da linha da tabela. Este recurso pode ser usado para obter acesso rápido aos dados baseado em alguma transformação dos dados básicos. Por exemplo, um índice computado como upper (col) permite a cláusula WHERE upper (col) = 'JIM' utilizar um índice.

PostgreSQL fornece os métodos de índice B-tree, hash, GiST, SP-GiST e GIN. Os usuários também podem definir seus próprios métodos de índice, mas isso é bastante complicado.

Quando a cláusula WHERE está presente, um índice parcial é criado. Um índice parcial é um índice que contém entradas para apenas uma parte de uma tabela, geralmente uma porção mais útil para a indexação do que o resto do quadro. Por exemplo, se você tiver uma tabela que contém ambas as ordens faturadas e não faturadas, em que os pedidos não faturados ocupam uma pequena fração da tabela. Se esta é uma seção usada com frequência, pode melhorar o desempenho criando um índice apenas para esta parte. Outra

aplicação possível é a utilização da cláusula WHERE junto com UNIQUE para exigir a unicidade de um subconjunto de uma tabela.

A expressão utilizada na cláusula WHERE pode referenciar apenas as colunas da tabela subjacente, mas ele pode usar todas as colunas, e não apenas as que estão sendo indexadas. Atualmente, subconsultas e expressões de agregação não são permitidas na cláusula WHERE. As mesmas restrições se aplicam a campos de índice que são expressões. Veja a sintaxe do comando abaixo:

```
CREATE [ UNIQUE ] INDEX [ CONCURRENTLY ] [ name ] ON table_name [ USING method ]
    ( { column_name | ( expression ) } [ COLLATE collation ] [ opclass ] [ ASC | DESC ] [ NULLS { FIRST | LAST } ] [, ...] )
    [ WITH ( storage_parameter = value [, ... ] ) ]
    [ TABLESPACE tablespace_name ]
    [ WHERE predicate ]
```

Todas as funções e operadores utilizados em uma definição de índice deve ser "imutável", isto é, os seus resultados devem depender somente de seus argumentos e nunca de uma influência externa (como o conteúdo de outra tabela ou a hora atual). Esta restrição garante que o comportamento do índice seja bem definido. Para utilizar uma função definida pelo usuário em uma expressão de índice ou na cláusula WHERE, lembre-se de marcar a função imutável ao criá-lo.

CREATE VIEW

O CREATE VIEW define uma visão sobre uma tabela. A visão não é fisicamente materializada. Em vez disso, a consulta é executada toda vez que a visão é referenciada em uma consulta.

Se utilizarmos a sintaxe CREATE OR REPLACE VIEW é semelhante ao comando sem a opção de REPLACE, mas se uma visão com o mesmo nome já existir, ela será substituída. A nova consulta deve gerar as mesmas colunas que foram geradas na visão existente (isto é, os mesmos nomes de colunas e na mesma ordem e com os mesmos tipos de dados), mas pode adicionar colunas adicionais para o fim da lista. Os cálculos que deram origem às colunas de saída podem ser completamente diferentes.

Se um nome de esquema for fornecido (por exemplo, CREATE VIEW esquema_visao ...), então a visão será criada no esquema especificado. Caso contrário, ele é criado no esquema corrente. Visões temporárias são criadas em um esquema especial, portanto o nome do esquema não pode ser dado quando se cria uma visão temporária. O nome da visão deve ser distinto do nome de qualquer outra visão, tabela, sequência, índice ou tabela estrangeira no mesmo esquema. Veja abaixo a sintaxe do comando:

```
CREATE [ OR REPLACE ] [ TEMP | TEMPORARY ] [ RECURSIVE ] VIEW name [ ( column_name [, ...] ) ]
    [ WITH ( view_option_name [= view_option_value] [, ... ] ) ]
    AS query
    [ WITH [ CASCADED | LOCAL ] CHECK OPTION ]
```

HORA DE PRATICAR!

3. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

A instrução SQL em PostgreSQL abaixo está mal formulada.

```
CREATE VIEW vista AS SELECT 'Hello World';
```

Isto aconteceu, porque

A a criação de uma visualização requer a utilização da cláusula WHERE para a restrição dos dados.

B não é possível criar uma VIEW sem a identificação do tipo de dado e sem a determinação da quantidade de registros selecionados.

C o comando CREATE VIEW deve utilizar a cláusula FROM para o nome da tabela.

D a criação de uma visualização (VIEW) requer a definição de um gatilho (trigger) correspondente ao nome da coluna.

E por padrão, o tipo de dado será considerado indefinido (unknown) e a coluna irá utilizar o nome padrão ?column?.

Comentário. Vamos analisar as alternativas. De cara podemos dizer que a resposta a essa questão está nas notas da documentação do comando CREATE VIEW. A alternativa A aponta uma restrição que não faz sentido. Basta pensarmos em uma visão criada sobre duas colunas de uma determinada tabela que retorna todas as linhas dessa tabela. Neste caso, não precisamos ter a cláusula WHERE.

A alternativa B também está incorrera, pois não precisamos criar uma VIEW determinando a quantidade de tuplas retornadas na seleção. Contudo, a primeira parte da alternativa está correta. Para que o SGBD consiga criar a VIEW, ele precisa que cada coluna ou atributo tenha um nome e um tipo de dado correspondente. Veja que isso acontece naturalmente quanto selecionamos os dados de uma tabela já existente. Porém, não é o que acontece quando optamos por utilizar uma constante, como acontece no enunciado da questão.

Na alternativa C, observamos que o erro está em dizer que é necessária a utilização da cláusula FROM. Não é verdade, pois é possível selecionar uma tabela apenas de valores constantes. Veremos um exemplo logo em seguida.

A criação de gatilhos ou triggers para que a VIEW seja atualizada depende da complexidade da VIEW. Algumas restrições são impostas pelo Postgres para que a VIEW seja considerada UPDATABLE. Desta forma, não podemos dizer que uma VIEW requer, necessariamente, um gatilho.

Por fim, a nossa resposta, a alternativa E. Veja que o comando da questão não faz referência a nenhuma tabela para que sejam extraídos o tipo e o nome dos atributos, sendo

necessário dar-lhe um nome e um tipo. Veja abaixo como ficaria a sintaxe correta do comando:

CREATE VIEW vista AS SELECT text 'Hello World' AS hello;

Inclusive, o exemplo do problema apresentado no enunciado é o mesmo que temos na documentação. A resolução do problema é a mesma apresentada acima. Observe que agora temos o tipo text e o nome hello.

Gabarito: E.

CREATE RULE E CREATE TRIGGER

No PostgreSQL, existem duas maneiras de lidar com eventos: Regras (RULES) e Gatilhos (Triggers). Como ponto de partida, podemos geralmente dizer que as regras são geralmente manipuladores de eventos simples, enquanto os gatilhos são manipuladores de eventos mais complexos.

Gatilhos e regras são frequentemente usados para atualizar acumuladores e modificar ou deletar registros que pertencem a tabelas diferentes daquela em que estamos modificando registros. São ferramentas muito poderosas que nos permitem realizar operações em tabelas diferentes daquela em que estamos modificando os dados.

As regras (RULES) são manipuladores de eventos simples. No nível do usuário, é possível gerenciar todos os eventos que realizam operações de gravação, que são as seguintes: INSERT, DELETE e UPDATE.

O conceito fundamental por trás das regras é modificar o fluxo de um evento. Se nos é dado um evento, o que podemos fazer quando certas condições ocorrem é o seguinte:

- Não fazer nada e desfazer a ação daquele evento.
- Acionar outro evento em vez do padrão.
- Acionar outro evento em conjunto com o padrão.

Portanto, dada uma operação de gravação, por exemplo, uma operação de INSERT, podemos realizar uma destas três ações:

- Cancelar a operação.
- Executar outra operação em vez de INSERT.
- Executar o INSERT e simultaneamente execute outra operação.

Para tal, escrevemos um comando usando a seguinte sintaxe?

```
CREATE [ OR REPLACE ] RULE nome AS ON evento
   TO tabela [ WHERE condição ]
   DO [ ALSO | INSTEAD ] { NOTHING | command | ( command ; command ... ) }
```

O comando CREATE TRIGGER cria um gatilho. O gatilho fica associado a uma tabela, visão ou tabela estrangeira especificada e executa a função especificada quando ocorrem certos eventos. Conhecido como evento, condição, ação.

Um gatilho pode ser especificado para disparar antes de uma operação ser realizada em uma linha (antes das restrições serem verificadas e os comandos de INSERT, UPDATE ou DELETE serem executados); ou após a conclusão da operação (após as restrições serem verificadas e o INSERT, UPDATE ou DELETE serem completados). Outra opção é executar um TRIGGER em vez da operação (INSTEAD OF), no caso de **inserções, atualizações ou exclusões em uma visão**.

Se o gatilho for disparado antes ou no lugar do evento, o gatilho pode evitar que a operação para a linha corrente seja feita, ou ainda, mudar os valores da linha (somente para operações de INSERT e UPDATE). Se o gatilho disparar após o evento, todas as alterações, incluindo os efeitos de outros gatilhos, são "visíveis" para o gatilho.

Um gatilho definido com os termos FOR EACH ROW é chamado uma vez para cada linha que a operação modifica. Por exemplo, uma exclusão que afeta 10 linhas irá executar os gatilhos ON DELETE 10 vezes separadamente, uma para cada linha excluída. Em contraste, um gatilho que está definido com FOR EACH STATEMENT somente executa uma vez para uma determinada operação, independentemente de quantas linhas ele modifica (em particular, uma operação que não modifica nenhuma linha, ainda assim, resulta na execução do gatilho).

Gatilhos podem ser especificados para disparar em vez de (INSTEAD OF). Neste caso, um evento de disparo deve ser construído com FOR EACH ROW e só pode ser definido sobre visões. Gatilhos BEFORE e AFTER sobre visões devem ser marcados com FOR EACH STATEMENT.

Além disso, os gatilhos podem ser definidos para disparar quando ocorre a execução do comando TRUNCATE da tabela. Neste caso, apenas o FOR EACH STATEMENT pode ser usado.

A tabela a seguir resume os tipos de gatilhos que podem ser utilizados em tabelas, views e tabelas estrangeiras:

Quando	Evento	Nível de linha	Nível de comando
BEFORE	INSERT/UPDATE/DELETE	TABELAS E TABELAS ESTRANGEIRAAS	TABELAS, VISÕES E TABELAS ESTRANGEIRAS
	TRUNCATE	-	TABELAS

AFTER	INSERT/UPDATE/DELETE	TABELAS E TABELAS ESTRANGEIRAAS	TABELAS, VISÕES E TABELAS ESTRANGEIRAS
	TRUNCATE	-	TABELAS
INSTEAD OF	INSERT/UPDATE/DELETE	VISÕES	-
	TRUNCATE	-	-

Perceba que:

1. Não faz sentido definir o um trigger para um evento de TRUNCATE em nível de linha, nem faz sentido definir eventos de TRUNCATE para visões.
2. Também não faz sentido definir instruções de INSTEAD OF associada ao evento de TRUCANTE.
3. Outro ponto é que não existe gatilhos de nível de linhas (FOR EACH ROW) para visões.

   Observem agora a sintaxe do comando abaixo:

```
CREATE [ CONSTRAINT ] TRIGGER name { BEFORE | AFTER | INSTEAD OF } { event [ OR ... ] }
    ON table_name
    [ FROM referenced_table_name ]
    [ NOT DEFERRABLE | [ DEFERRABLE ] { INITIALLY IMMEDIATE | INITIALLY DEFERRED } ]
    [ FOR [ EACH ] { ROW | STATEMENT } ]
    [ WHEN ( condition ) ]
    EXECUTE PROCEDURE function_name ( arguments )

where event can be one of:

    INSERT
    UPDATE [ OF column_name [, ... ] ]
    DELETE
    TRUNCATE
```

OUTROS COMANDOS DDL

Além dos comandos de CREATE, usados para criação/modificação de objetos dentro dos bancos de dados Postgres, temos outros comandos que são importantes dentro do escopo da linguagem DDL. São eles o DROP e o ALTER.

O DROP TABLE, por exemplo, remove tabelas do banco de dados. Somente o proprietário da tabela, o proprietário do esquema, e superusuário podem descartar uma tabela. Para esvaziar uma tabela de linhas sem destruir a tabela, utilize DELETE ou TRUNCATE.

O comando ALTER TABLE altera a definição de uma tabela existente. Existem várias maneiras para fazer isso. Note-se que o nível de bloqueio necessário pode ser diferente para cada uma das formas possíveis. Um bloqueio ACCESS EXCLUSIVE é realizado. Quando vários subcomandos são listados, o bloqueio mantido será o mais rigoroso exigido de qualquer subcomando.

COMANDOS DML

Nas próximas sessões, apresentaremos detalhes sobre os comandos DML: SELECT, INSERT, UPDATE e DELETE.

SELECT

Visando analisar o passo a passo da execução do comando SELECT, apresentaremos a seguir a sintaxe do comando:

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
SELECT [ ALL | DISTINCT [ ON ( expression [, ...] ) ] ]
    [ * | expression [ [ AS ] output_name ] [, ...] ]
    [ FROM from_item [, ...] ]
    [ WHERE condition ]
    [ GROUP BY expression [, ...] ]
    [ HAVING condition [, ...] ]
    [ WINDOW window_name AS ( window_definition ) [, ...] ]
    [ { UNION | INTERSECT | EXCEPT } [ ALL | DISTINCT ] select ]
    [ ORDER BY expression [ ASC | DESC | USING operator ] [ NULLS { FIRST | LAST } ] [, ...] ]
    [ LIMIT { count | ALL } ]
    [ OFFSET start [ ROW | ROWS ] ]
    [ FETCH { FIRST | NEXT } [ count ] { ROW | ROWS } ONLY ]
    [ FOR { UPDATE | NO KEY UPDATE | SHARE | KEY SHARE } [ OF table_name [, ...] ] [ NOWAIT ] [...] ]

where from_item can be one of:

    [ ONLY ] table_name [ * ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] ( select ) [ AS ] alias [ ( column_alias [, ...] ) ]
    with_query_name [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    [ LATERAL ] function_name ( [ argument [, ...] ] ) [ AS ] alias ( column_definition [, ...] )
    [ LATERAL ] function_name ( [ argument [, ...] ] ) AS ( column_definition [, ...] )
    [ LATERAL ] ROWS FROM( function_name ( [ argument [, ...] ] ) [ AS ( column_definition [, ...] ) ] [, ...] )
                [ WITH ORDINALITY ] [ [ AS ] alias [ ( column_alias [, ...] ) ] ]
    from_item [ NATURAL ] join_type from_item [ ON join_condition | USING ( join_column [, ...] ) ]

and with_query is:

    with_query_name [ ( column_name [, ...] ) ] AS ( select | values | insert | update | delete )

TABLE [ ONLY ] table_name [ * ]
```

SELECT recupera linhas de zero ou mais tabelas. O processamento geral do **SELECT** é o seguinte:

Todas as consultas listas na cláusula **WITH** são computadas. Estas servem efetivamente como tabelas temporárias que podem ser referenciadas na cláusula FROM. Uma consulta WITH referenciada mais de uma vez no **FROM** é calculada apenas uma vez.

Todos os elementos na lista do FROM são computados. (Cada elemento na lista FROM é uma tabela real ou virtual). Se mais de um elemento ou uma tabela estão especificados na lista, eles são cruzados por meio da operação de CROSS JOIN.

Se a cláusula **WHERE** é especificada, todas as linhas que não satisfazem a condição são eliminadas da saída.

Se a cláusula **GROUP BY** for especificada ou se há chamadas a uma função de agregação, a saída é combinada em grupos de linhas que correspondem a um ou mais valores, e os resultados das funções agregadas são computados. Se a cláusula **HAVING** estiver presente, ela elimina os grupos que não satisfaçam a condição dada.

As linhas de saída reais são calculadas utilizando as expressões de saída do SELECT para cada grupo de linha ou linha selecionada.

SELECT **DISTINCT** elimina as linhas duplicadas a partir do resultado. SELECT DISTINCT ON elimina linhas que correspondem a todas as expressões especificadas. SELECT **ALL** (o padrão) retorna todas as linhas candidatas, incluindo as repetidas.

Usando a operadores **UNION, INTERSECT e EXCEPT**, a saída de mais de uma instrução SELECT pode ser combinada para formar um único conjunto de resultado. O operador UNION retorna todas as linhas que estão em um ou ambos os conjuntos de resultados. O operador INTERSECT retorna todas as linhas presentes em ambos os conjuntos de resultados. O operador EXCEPT retorna as linhas que estão no primeiro conjunto de resultados, mas não no segundo.

Em todos os três casos, as linhas duplicadas são eliminadas, a menos que a cláusula ALL seja especificada. A palavra DISTINCT pode ser adicionada para especificar explicitamente a eliminação de linhas duplicadas. Observe que DISTINCT é o comportamento padrão aqui, embora ALL seja o padrão para o SELECT.

Se a cláusula **ORDER BY** é especificada, as linhas retornadas são classificadas na ordem especificada. Se ORDER BY não é dado, as linhas são retornadas na ordem em que o sistema encontra mais rapidamente o resultado.

Se o limite (LIMIT ou **FETCH FIRST**) ou cláusula **OFFSET** for especificado, a instrução de SELECT retorna apenas um subconjunto das linhas do resultado. O LIMIT vai estabelecer a quantidade de linhas que será retornada no resultada. Já o OFFSET desloca o ponteiro no resultado retornado, por exemplo, LIMIT 5 OFFSET 10 vai retornar 5 elementos a partir do décimo primeiro elemento.

Se FOR UPDATE, FOR NO KEY UPDATE, FOR SHARE ou FOR KEY SHARE for especificado, a instrução SELECT bloqueia as linhas selecionadas contra atualizações simultâneas.

Você deve ter o privilégio SELECT em cada coluna usada em um comando SELECT. A utilização de FOR NO KEY UPDATE, FOR UPDATE, FOR SHARE ou FOR KEY SHARE requer também o privilégio UPDATE.

INSERT

O comando **INSERT** adiciona novas linhas em uma tabela. Pode-se inserir uma ou mais linhas especificando seus valores, ou zero ou mais linhas resultantes de uma consulta.

Os nomes das colunas de destino podem ser listados em qualquer ordem. Se nenhuma lista de nomes de coluna é dada, o padrão é usar todas as colunas da tabela na ordem declarada. Também é possível usar os primeiros N nomes de colunas, se existirem apenas N colunas fornecidas pela cláusula VALUES ou no resultado da consulta. Os valores fornecidos pela cláusula VALUES ou consulta são associados com a lista de colunas explícita ou implícita da esquerda para a direita.

Cada coluna que não estiver presente na lista de colunas explícita ou implícita será preenchida com um valor padrão. O valor padrão será o declarado ou nulo, se não houver

nenhum. Se a expressão de qualquer coluna não é do tipo de dados correto, a conversão automática de tipo será tentada.

Você deve ter o privilégio INSERT na tabela para inserir nela. Se uma lista de colunas é especificada, você só precisa do privilégio de inserir nas colunas listadas. A utilização da cláusula **RETURNING** requer o privilégio SELECT em todas as colunas mencionadas. Se você usar a cláusula consulta (query) para inserir linhas de uma consulta, você, naturalmente, precisa ter o privilégio SELECT em qualquer tabela ou coluna usada na consulta. Veja abaixo a sintaxe do comando INSERT:

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
INSERT INTO table_name [ ( column_name [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expression | DEFAULT } [, ...] ) [, ...] | query }
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

DELETE

O **DELETE** apaga as linhas que satisfazem a cláusula WHERE da tabela especificada. Se a cláusula WHERE estiver ausente, o efeito é excluir todas as linhas na tabela. O resultado é uma tabela válida, porém vazia.

Dica: TRUNCATE é uma extensão do PostgreSQL que fornece um mecanismo mais rápido para remover todas as linhas de uma tabela.

Há duas maneiras de excluir linhas de uma tabela utilizando informações contidas em outras tabelas no banco de dados: usando subseleções ou especificando tabelas adicionais na cláusula **USING**. A técnica mais apropriada depende das circunstâncias específicas.

A cláusula opcional **RETURNING**, facultativa ao DELETE, calcula e retorna valor(es) com base em cada linha realmente excluída. Qualquer expressão pode ser computada, utilizando as colunas da tabela e/ou colunas de outras tabelas mencionadas no USING.

Você deve ter o privilégio DELETE na tabela para excluir suas linhas. Também deve ter o privilégio SELECT para qualquer tabela que apareça na cláusula USING ou cujos valores são lidos na condição. Veja abaixo a sintaxe do DELETE:

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
DELETE FROM [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    [ USING using_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...]
```

UPDATE

O comando **UPDATE** muda os valores das colunas especificadas em todas as linhas que satisfazem a condição. Somente as colunas a serem modificadas devem ser mencionadas na cláusula SET. Colunas que não serão modificadas explicitamente manterão seus valores anteriores.

Há duas maneiras de modificar uma tabela utilizando informações contidas em outras tabelas no banco de dados: usando subseleções ou especificando tabelas adicionais na cláusula FROM. A técnica mais apropriada depende das circunstâncias específicas.

A cláusula opcional **RETURNING** faz com que o UPDATE possa calcular e retornar valor(es) com base em cada linha atualizada. Você deve ter o privilégio UPDATE na tabela ou pelo menos na(s) coluna(s) que estão listadas para serem atualizadas. Você também deve ter o privilégio SELECT em qualquer coluna cujos valores são lidos pela expressão ou pela condição. Veja a sintaxe do comando abaixo:

```
[ WITH [ RECURSIVE ] with_query [, ...] ]
UPDATE [ ONLY ] table_name [ * ] [ [ AS ] alias ]
    SET { column_name = { expression | DEFAULT } |
          ( column_name [, ...] ) = ( { expression | DEFAULT } [, ...] ) } [, ...]
    [ FROM from_list ]
    [ WHERE condition | WHERE CURRENT OF cursor_name ]
    [ RETURNING * | output_expression [ [ AS ] output_name ] [, ...] ]
```

1. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Considere o trecho em PostgreSQL abaixo.

INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99), (2,'Bread',1.99), (3,'Milk', 2.99);

Considerando a existência prévia da tabela products contendo as colunas product_no, name e price, e desconsiderando os tipos de dados, esse trecho irá resultar:

A na adição de 3 novas colunas na tabela products.

B na adição de 3 novas linhas na tabela products.

C em erro, pois não é possível múltiplas inserções em um único comando SQL.

D em erro, pois para se realizar múltiplas inserções é necessário a utilização da cláusula SELECT.

E em erro, pois múltiplas inserções são possíveis somente com a utilização de colchetes para a limitação dos registros.

Comentário. Para respondermos a essa questão, precisaríamos de mais detalhes sobre a tabela. Mas, considerando que ela só possui essas três colunas e que os tipos de dados não são relevantes, podemos avaliar que o comando INSERT criará novas linhas na tabela *products*. Desta forma, podemos marcar nossa resposta na alternativa B.

Gabarito: B.

PECULIARIDADES DOS TIPOS DE DADOS DO POSTGRESQL

O PostgreSQL suporta os tipos de dados tradicionais de qualquer banco de dados: numérico, string de caracteres, datas e horários, booleanos, e assim por diante. Ele ainda partiu à frente, adicionando suporte para datas e horas com fusos horários, intervalos de tempo, matrizes e XML.

Se isso não for suficiente, você pode até mesmo adicionar seus tipos personalizados. Nesta aula, nós vamos apresentar os principais tipos de dados do PostgreSQL agrupados de acordo com suas características. Antes apresentaremos alguns fatores que tornam importante a presença de diferentes tipos de dados nos SGBDs.

Resultados consistentes - Colunas de um tipo uniforme produz resultados consistentes. A exibição, a classificação, a agregação e a junção também entregam resultados consistentes.

Validação de dados - Colunas de um tipo uniforme aceitam apenas dados no formato definido pelo tipo, dados inválidos são rejeitados. Por exemplo, uma coluna do tipo INTEGER rejeitará um valor DATE.

Armazenamento compacto - Colunas de um tipo uniforme são armazenadas de forma mais compacta.

Desempenho - Colunas de um tipo uniforme são processadas mais rapidamente.

TIPOS DE STRINGS DE CARACTERES

Tipos de cadeias de caracteres são os tipos de dados mais comumente usados. Eles podem conter qualquer sequência de letras, números, pontuação e outros caracteres válidos. Cadeias de caracteres típicas são nomes, descrições e endereços para correspondência. Você pode armazenar qualquer valor em uma cadeia de caracteres. No entanto, este tipo deve ser usado somente quando outros tipos de dados são inadequados. Os outros tipos fornecem uma validação melhor para os dados, um armazenamento mais compacto e um melhor desempenho.

São três os tipos de dados sequência de caracteres: TEXT, VARCHAR (length) e CHAR(length). TEXT não limita o número de caracteres armazenados. VARCHAR(length) limita o comprimento do campo de caracteres ao comprimento passado como parâmetro. TEXT e VARCHAR() armazenam apenas o número de caracteres da string. CHAR (length) é semelhante ao VARCHAR (), porém ele sempre armazena exatamente os caracteres definidos no comprimento. Este tipo completa o valor com espaços à direita para alcançar o comprimento específico. Ele também fornece acesso ligeiramente mais rápido do que TEXT ou VARCHAR().

Entender por que os tipos de cadeia de caracteres diferem de outros tipos de dados pode ser difícil. Por exemplo, você pode armazenar 763 como uma cadeia de caracteres. Nesse

caso, você irá armazenar os símbolos 7, 6 e 3, não o valor numérico 763. Por conseguinte, não é possível adicionar um número à cadeia de caracteres 763, porque não faz sentido adicionar um número a três símbolos.

Da mesma forma, a cadeia de caracteres 3/8/1992 consiste em oito símbolos, começando com 3 e terminando com 2. Se você armazenar esse valor em um tipo de dados sequência de caracteres, ele não é uma data. Você não pode ordenar essa string com outros valores e esperar que elas sejam classificadas em ordem cronológica. A sequência de 1/4/1994 é inferior a 3/8/1992, quando ambos são cadeias de caracteres, porque 1 é menor do que 3.

Esses exemplos ilustram porque os outros tipos de dados são valiosos. Os outros tipos usam formatos predefinidos para os seus dados. Assim, eles apoiam as operações sobre as informações armazenadas de forma mais adequada.

TIPOS NUMÉRICOS

Tipos de número permitem a armazenagem de números. Os tipos numéricos consistem de inteiros de dois, quatro e oito bytes. Consistem também de números de ponto flutuante de quatro e oito bytes, e decimais de precisão selecionável. Os tipos de números são INTEGER, INT2 ou SMALLINT, INT8, OID, NUMERIC(), DECIMAL, FLOAT e FLOAT4.

INTEGER, INT2 e INT8 armazenam números inteiros de diferentes ranges. Intervalos maiores requerem mais espaço de armazenamento. Por exemplo, INT8 requer o dobro do armazenamento de INTEGER e é mais lento.

OID é usada para armazenar o PostgreSQL identificadores de objetos. Embora você possa usar INTEGER para este fim, OID documenta melhor o significado do valor armazenado na coluna. O valor do OID é único para cada objeto dentro de um cluster.

NUMERIC (precisão, escala) permite que o usuário defina os dígitos de precisão e o arredondamento em casas decimais. Este tipo é mais lento do que os outros tipos de números. A escala de um NUMERIC é o número de dígitos decimais na parte fracionária, à direita do ponto decimal. A precisão é o número total de dígitos significativos em todo o número, ou seja, o número de dígitos de ambos os lados do ponto decimal. Os tipos DECIMAL e NUMERIC são equivalentes. Ambos fazem parte do padrão SQL.

FLOAT e FLOAT4 permitem o armazenamento de valores de ponto flutuante. Os números são armazenados usando 15 (FLOAT) ou 6 (FLOAT4) dígitos de precisão. A localização do ponto decimal é armazenada separadamente. Então, valores grandes, tais como 4.78145e+32, podem ser representados.

FLOAT e FLOAT4 são rápidos e têm um armazenamento compacto, mas podem produzir arredondamento impreciso durante os cálculos. Quando você precisar de uma precisão absoluta para valores de ponto flutuante, use NUMERIC() em seu lugar. Por exemplo, sugerimos que você armazene quantias monetárias como NUMERIC().

Observem abaixo uma tabela com os tipos numéricos presentes no PostgreSQL:

Name	Storage Size	Range
smallint	2 bytes	-32768 to +32767
integer	4 bytes	-2147483648 to +2147483647
bigint	8 bytes	-9223372036854775808 to +9223372036854775807
decimal	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	6 decimal digits precision
double precision	8 bytes	15 decimal digits precision
smallserial	2 bytes	1 to 32767
serial	4 bytes	1 to 2147483647
bigserial	8 bytes	1 to 9223372036854775807

Vejam que os tipos ***decimal*** e ***numeric*** têm o tamanho de armazenamento variável. Outra novidade apresentada acima são os tipos ***smallserial***, ***serial*** e ***bigserial***, utilizados para valores auto incrementados, cujos domínios só permitem **valores positivos**. Vamos agora resolver duas questões sobre o assunto.

1. BANCA: FCC - Analista Judiciário (TRT 23ª Região)/Apoio Especializado/Tecnologia da Informação/2016

São vários os tipos de dados numéricos no PostgreSQL. O tipo

a) smallint tem tamanho de armazenamento de 1 byte, que permite armazenar a faixa de valores inteiros de −128 a 127.

b) bigint é a escolha usual para números inteiros, pois oferece o melhor equilíbrio entre faixa de valores, tamanho de armazenamento e desempenho.

c) integer tem tamanho de armazenamento de 4 bytes e pode armazenar valores na faixa de −32768 a 32767.

d) numeric pode armazenar números com precisão variável de, no máximo, 100 dígitos.

e) serial é um tipo conveniente para definir colunas identificadoras únicas, semelhante à propriedade auto incremento.

Comentário: A questão trata dos tipos de dados numéricos do PostgreSQL. Observem abaixo uma tabela que vimos com os tipos numéricos presentes no PostgreSQL:

Name	Storage Size	Range
smallint	2 bytes	-32768 to +32767
integer	4 bytes	-2147483648 to +2147483647
bigint	8 bytes	-9223372036854775808 to +9223372036854775807
decimal	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	6 decimal digits precision
double precision	8 bytes	15 decimal digits precision
smallserial	2 bytes	1 to 32767
serial	4 bytes	1 to 2147483647
bigserial	8 bytes	1 to 9223372036854775807

Vejam que os tipos ***decimal*** e ***numeric*** têm o tamanho de armazenamento variável. Já os tipos smallserial, serial e bigserial são utilizados para valores auto incrementados, cujos domínios só permitem valores positivos.

Agora vamos aos erros das alternativas. Na letra A, diz que smallint tem 1 byte, ao invés de 2. Na alternativa B, fala que o bigint é uma alternativa intermediária, quando na realidade ele é o valor numérico que possui o maior range. A assertiva C afirma erroneamente que integer possui apenas 2 bytes, quando na realidade apresenta 4. Por fim, a letra E diz que decimal tem precisão de 100 dígitos, quando na realidade podemos ter até 16383 dígitos depois da casa decimal.

E a alternativa E? Essa é a nossa resposta, apresenta o tipo serial, que é utilizado para definir colunas identificadoras únicas, semelhante à propriedade auto incremento.

Gabarito: E

HORA DE PRATICAR!

2. BANCA: FCC - Técnico Judiciário (TRE-PB)/Apoio Especializado/Programação de Sistemas/2015

No PostgreSQL, o tipo de dados numérico considerado meramente uma notação conveniente para definir colunas identificadoras únicas, semelhante à propriedade auto incremento em alguns Sistemas Gerenciadores de Banco de Dados, é o tipo

a) serial.

b) smallint.

c) byte.

d) bit.

e) blob.

Comentário: O gabarito da questão é a letra a). Novamente a banca FCC cobrou uma questão sobre o tipo numérico serial. Conforme vimos na questão anterior, o tipo serial é utilizado para definir colunas identificadoras únicas, semelhante à propriedade auto incremento.

Gabarito: A

TIPOS TEMPORAIS E LÓGICO

Tipos temporais permitem o armazenamento de data, hora e informação do intervalo de tempo. Embora esses dados possam ser armazenados em cadeias de caracteres, é melhor usar tipos temporais, pelas razões descritas anteriormente.

Os quatro tipos temporais são DATE, TIME, TIMESTAMP e INTERVAL. DATE permite o armazenamento de uma data única que consiste em ano, mês e dia. O formato usado para datas de entrada e de exibição é controlado pela configuração DATESTYLE. TIME permite o armazenamento de uma hora, minuto e segundo, separados por dois pontos. TIMESTAMP armazena tanto a data e a hora, por exemplo, 2015/30/10 10:38:29.

INTERVAL representa um intervalo de tempo, por exemplo, 5 horas ou 7 dias. Um INTERVAL é muitas vezes gerado subtraindo dois valores TIMESTAMP para calcular o tempo decorrido. Por exemplo, 1996/12/15 19:00:40 menos 1996/12 /8 14:00:10 resulta em um valor de intervalo de 7 5:00:30, que é de 7 dias, 5 horas e 30 segundos. Tipos temporais também podem lidar com designações de fuso horário.

O único tipo de dados lógico é BOOLEAN. Um campo BOOLEAN pode armazenar apenas verdadeiro ou falso, e, claro, NULL. Você pode dar entrada a verdadeiro como true, t, yes, y ou 1. False pode ser introduzido como false, f, no, n ou 0. Embora o verdadeiro e o falso possam ser inseridos de uma variedade de maneiras, na saída, os valores são sempre t para verdadeiro e f para falso.

TIPOS GEOMÉTRICOS E DE REDE

Os tipos geométricos suportam armazenamento de primitivas geométricas. Eles incluem POINT, LSEG, PATH, BOX, CIRCLE e POLYGON. A tabela abaixo mostra os tipos geométricos e os valores típicos para cada:

Tipo	Exemplo	Comentário
POINT	(2,7)	(x,y) são as coordenadas
LSEG	[(0,0),(1,3)]	Os pontos de início e fim de um segmento de reta
PATH	((0,0),(3,0),(4,5),(1,6))	() é um caminho fechado, [] é um caminho aberto
BOX	(1,1),(3,3)	Pontos opostos dos vértices de um retângulo
CIRCLE	<(1,2), 60>	Ponto Central e raio de um círculo
POLYGON	((3,1),(3,3),(1,0))	Pontos de um polígono fechado

Os tipos de rede são INET, CIDR e MACADDR. INET permite o armazenamento de um endereço IP, com ou sem uma máscara de rede. Um valor típico INET com uma máscara de rede é 172.20.90.150 255.255.255.0. O CIDR armazena endereços IP de rede. Ele permite que uma máscara especifique o tamanho do segmento de rede. Um valor típico para o CIDR é 172.20.90.150/24.

Já MACADDR armazena o endereço MAC (Media Access Control), que são atribuídas às placas de rede Ethernet no momento da sua fabricação. Um valor típico MACADDR é 0:50:4:1d:F6:db.

ARRAYS

PostgreSQL permite que colunas de uma tabela sejam definidas como arrays ou matrizes multidimensionais de comprimento variável. Um array pode ser definido de qualquer tipo nativo, de um tipo definido pelo usuário, do tipo de enumeração ou, ainda, de um tipo composto a ser criado. Arrays de domínios ainda não são suportados.

Para ilustrar o uso de tipos de arrays, vamos criada uma tabela:

```
CREATE TABLE sal_emp (
    name            text,
    pay_by_quarter  integer[],
    schedule        text[][]
);
```

Como mostrado, um atributo do tipo de dados array é criado anexando colchetes ([]) após o nome do tipo de dados dos elementos do array. O comando acima criará uma tabela chamada sal_emp com uma coluna do tipo text (nome), um array unidimensional do tipo inteiro (coluna pay_by_quarter), que representa o salário do empregado por trimestre, e um array bidimensional de texto (Schedule), que representa a agenda semanal do empregado.

A sintaxe para CREATE TABLE permite que o tamanho exato de matrizes possa ser especificado, por exemplo:

```
CREATE TABLE tictactoe (
    squares   integer[3][3]
);
```

No entanto, a implementação atual ignora quaisquer limites de tamanho de array fornecidos, ou seja, o comportamento é o mesmo dos arrays com comprimento não especificado. Agora que aprendemos um pouco sobre arrays, vamos responder à questão abaixo:

1. BANCA: FCC ANO: 2014 ÓRGÃO: TRT - 1ª REGIÃO (RJ) PROVA: TÉCNICO DO JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

No sistema gerenciador de Banco de Dados PostgreSQL (v. 9.1), a forma para se declarar um atributo com o tipo de dados Array, com duas dimensões, tendo o nome teste é

A . . . teste [[ ]] . . .

B . . . teste 2 [ ] . . .

C . . . teste [2] . . .

D . . . teste [ ] [ ] . . .

E . . . teste [ ] x [ ] . . .

Comentário. Pelo que conhecemos da sintaxe do comando utilizado para criação de atributos de tabela como arrays, podemos marcar como resposta a alternativa D. Vamos agora conhecer um pouco sobre os BLOBs ou Binary Large Objects.

Gabarito: D.

LARGE OBJECTS (BLOBS)

PostgreSQL não pode armazenar valores maiores que alguns milhares de bytes usando os tipos de dados discutidos até agora. Nem os dados binários podem ser facilmente inseridos dentro de aspas simples. Em vez disso, grandes objetos também chamados de Binary Large Objects ou BLOBS são usados para armazenar grandes valores e dados binários.

PostgreSQL tem uma facilidade para grandes objetos. Ele fornece acesso por meio de fluxo de dados (streaming) para o usuário aos dados que são armazenados em uma estrutura especial para objetos grandes. Streaming é útil quando se trabalha com valores de dados que são grandes demais para serem manipulados convenientemente de uma só vez.

Objetos grandes permitem o armazenamento de qualquer arquivo do sistema operacional, incluindo imagens ou grandes arquivos de texto, diretamente no banco de dados. Você carrega o arquivo no banco de dados usando o comando lo_import() e pode recuperá-lo a partir do banco de dados, usando o lo_export().

A figura abaixo mostra um exemplo que armazena uma imagem com o nome fruit. O comando lo_import() armazena a figura /usr/images/peach.jpg no banco de dados. A chamada de função retorna um OID, que é usado para se referir ao objeto importado. Este valor é armazenado no atributo fruit.image.

A função lo_export() usa o valor de OID para retornar o objeto armazenado no banco de dados e, em seguida, coloca a imagem na nova pasta /tmp/outimage.jpg. O 1 retornado por lo_export() indica uma exportação bem sucedida. A função lo_unlink() remove objetos grandes.

```
test=> CREATE TABLE fruit (name CHAR(30), image OID);
CREATE
test=> INSERT INTO fruit
test-> VALUES ('peach', lo_import('/usr/images/peach.jpg'));
INSERT 27111 1
test=> SELECT lo_export(fruit.image, '/tmp/outimage.jpg')
test-> FROM   fruit
test-> WHERE  name = 'peach';
 lo_export
-----------
         1
(1 row)

test=> SELECT lo_unlink(fruit.image) FROM fruit;
 lo_unlink
-----------
         1
```

Nomes de caminho completos devem ser usados com objetos grandes, pois o servidor de banco de dados é executado em um diretório diferente do que o cliente psql, por exemplo.

Os arquivos são importados e exportados pelo usuário *postgres*. Então, o *Postgres* deve ter permissão para ler, usando o lo_import(), e para gravação no diretório, usando o lo_export().

Devido ao fato de grandes objetos usarem o sistema de arquivos local, usuários que se conectam através de uma rede não podem usar lo_import ou lo_export(). Podem, no entanto, usar as funcionalidades \lo_import e \lo_export do psql.

TIPO DE DADOS NOSQL

Nesta seção, abordaremos os tipos de dados NoSQL que estão presentes no PostgreSQL. O PostgreSQL lida com os seguintes tipos de dados NoSQL: hstore, xml, json.

O **hstore** foi o primeiro tipo de dados NoSQL implementado no PostgreSQL. Este tipo de dados é usado para armazenar pares de valores-chave em um único valor.

O tipo de dados xml pode ser usado para armazenar dados XML. Sua vantagem sobre o armazenamento de dados XML em um campo de texto é que ele verifica os valores de entrada quanto à boa formação e há funções de suporte para executar operações de tipo seguro (type-safe). O uso deste tipo de dados requer que a instalação tenha sido construída com configure --with-libxml.

JSON significa **JavaScript Object Notation**. JSON é um formato de padrão aberto e é formado por pares de valores-chave. PostgreSQL suporta o tipo de dados JSON nativamente. Ele fornece muitas funções e operadores usados para manipular dados JSON. O PostgreSQL, além do tipo de dados **json**, também suporta o tipo **jsonb**. A diferença entre esses dois tipos de dados é que o primeiro é **representado internamente como texto**, enquanto o segundo é representado internamente de **maneira binária** e **indexável**.

JSON é amplamente utilizado ao trabalhar com tabelas grandes e quando uma estrutura de dados é necessária que minimiza o número de junções a serem feitas durante a fase de pesquisa.

FUNCIONALIDADES DO PGADMIN 4

SETUP E OPERAÇÃO DO SERVIDOR

Ferramentas de administração com interface gráfica são frequentemente solicitadas pelos administradores do sistema. PostgreSQL tem uma gama de opções de ferramentas. As duas opções mais populares são: pgAdmin IV e phpPgAdmin. Apresentamos abaixo alguns aspectos da ferramenta pgAdmin.

O pgAdmin é uma plataforma para administração e desenvolvimento de banco de dados PostgreSQL. O aplicativo pode ser usado em Linux, FreeBSD, Solaris, Mac OSX e plataformas Windows para gerenciar versões PostgreSQL 7.3 ou superior. Ele é executado em qualquer plataforma, bem como em versões comerciais e derivadas de PostgreSQL, como Postgres Plus Advanced Server and Greenplum.

O pgAdmin é projetado para atender às necessidades de todos os usuários. Entre suas funcionalidades, inclui escrever consultas SQL para desenvolver bases de dados complexas. A interface gráfica suporta todas as funcionalidades do PostgreSQL, tornando fácil a administração. O aplicativo também inclui um editor de **destaque de sintaxe SQL**, um editor de código para o server-side, um agente de agendamento de tarefas *SQL/batch/shell*, um suporte para o mecanismo de replicação *Slony-I* e muito mais.

Uma conexão com o servidor pode ser feita usando TCP/IP ou socket no domínio Unix (nas plataformas * nix), e pode ser criptografado usando SSL. Drivers adicionais não são necessários para se comunicar com o servidor de banco de dados.

O pgAdmin é desenvolvido por uma comunidade de especialistas em PostgreSQL de todo o mundo e está disponível em mais de uma dúzia de idiomas. Ele é um Software Livre liberado sob a licença do PostgreSQL. É possível visualizar a interface da plataforma na figura abaixo:

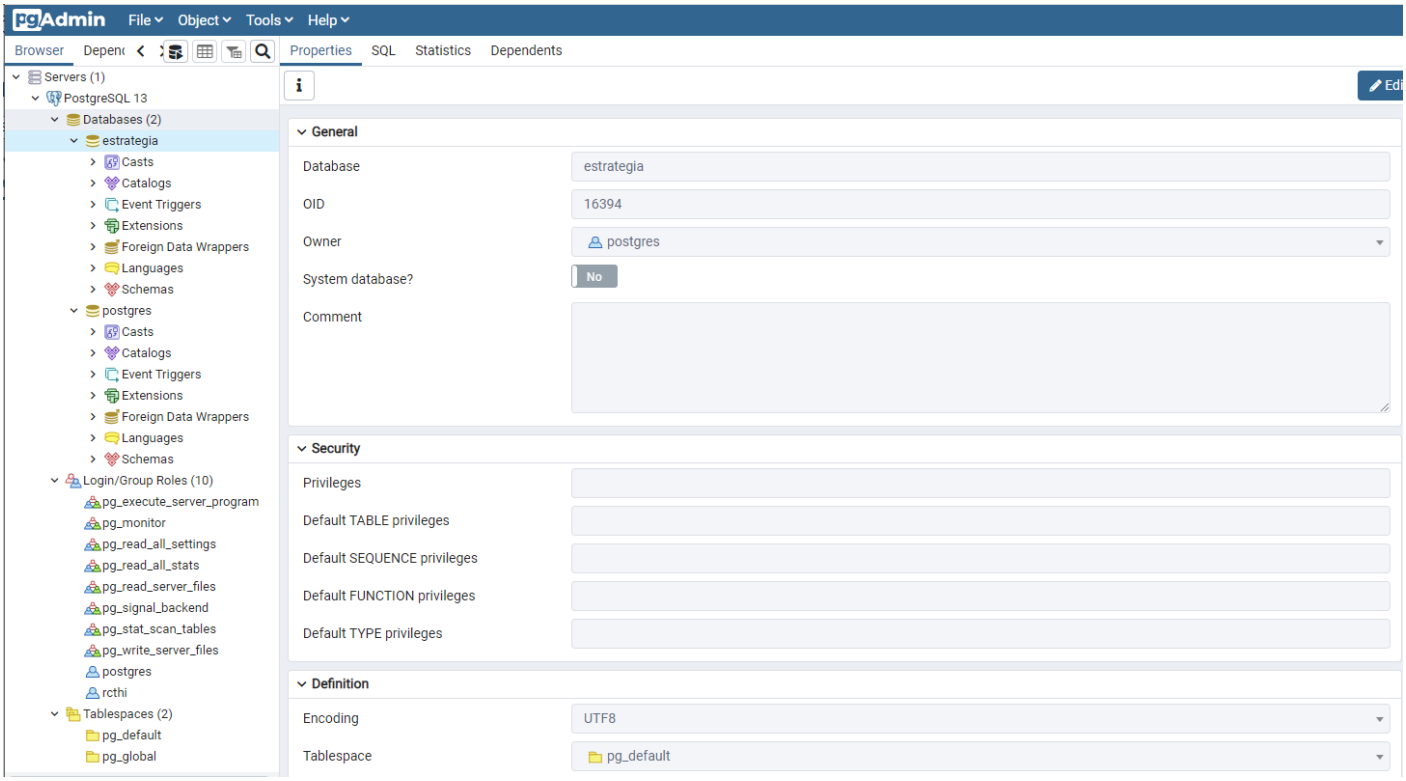


Figura 1 -Interface do pgAdmin 4

Observamos que, se fôssemos detalhar de forma prática o uso do pgAdmin, poderíamos perder muito tempo em assuntos que efetivamente não aparecem em provas de concursos. Desta forma, resolvemos listar as principais funcionalidades do pgAdmin:

• Gráfico para planos de consultas usando o **EXPLAIN**. Esta característica é impressionante, pois oferece uma visão pictórica sobre como o planejador de consultas está pensando. Acabaram os dias tentando percorrer a verbosidade das saídas do planejador baseados em texto.

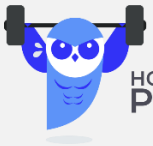
• **Painel SQL**. pgAdmin consegue interagir com PostgreSQL via SQL, permitindo inclusive que você veja o SQL gerado. Quando você usa a interface gráfica para fazer alterações em seu banco de dados, o SQL subjacente criado para executar as tarefas é exibido automaticamente no painel SQL. Para os novatos de SQL, estudar o SQL gerado é uma grande oportunidade de aprendizagem. Para profissionais, aproveitar o SQL gerado é uma grande economia de tempo.

• A edição direta de arquivos de configuração, tais como **postgresql.conf** e **pg_hba.conf**. Você já não precisa ficar procurando os arquivos ou utilizar outro editor.

Uma outra forma de mudar os parâmetros de configuração é usando o comando ***postgres***. Por exemplo, é possível definir as configurações de log do seu servidor de banco de dados usando o seguinte comando:

```
postgres -c log_connections=yes -c log_destination='syslog'
```

No comando acima estamos definindo que todas as tentativas de conexão com o servidor serão adicionadas ao log (log_connectins=yes) e que o destino do log é definido pelo sistema operacional em uso. Vejamos uma questão sobre o assunto:



(Ministério da Economia – Infraestrutura - 2020) Acerca de PostgreSQL, julgue o item a seguir.

98 No PostgreSQL, os parâmetros podem ser passados via shell; como exemplo, o comando abaixo enviará os logs para o sistema de logs syslog, que é mantido pelo sistema operacional Linux.

postgres -c log_connections=yes -c log_destination='syslog'

Comentários:

O comando acima começa com o log_connections .... esse parâmetro faz com que cada tentativa de conexão com o servidor seja registrada, bem como a conclusão bem-sucedida da autenticação do cliente. Este parâmetro não pode ser alterado após o início da sessão. O padrão é desligado.

O PostgreSQL suporta vários métodos para registrar mensagens do servidor, incluindo stderr, csvlog e syslog. É possível definir o parâmetro log_destination, especificando uma lista de destinos de log desejados separados por vírgulas. O padrão é registrar apenas no stderr. Este parâmetro só pode ser definido no arquivo postgresql.conf ou na linha de comando do servidor.

O uso do syslog ou do eventlog faz com que os logs sejam direcionados para o destino padrão dos logs no seu sistema operacional.

Assim, uma maneira de definir os parâmetros de configuração é fornecê-los como uma opção de linha de comando para o comando **postgres**. As opções da linha de comando substituem quaisquer configurações conflitantes no postgresql.conf.

Gabarito Certo.

• **Exportação de dados**. pgAdmin pode facilmente exportar os resultados da consulta para CSV ou outro formato delimitado. Ele pode até mesmo exportar como HTML, proporcionando-lhe um mecanismo de relatório facilmente ajustável.

• **Assistente de backup e restauração**. Não é preciso mais memorizar os comandos e interruptores para realizar um backup ou restauração, usando o pg_restore e pg_dump. O pgAdmin tem uma boa interface para fazer backup e restaurar banco de dados, esquemas, uma única tabela e informações globais. Na guia mensagem é possível ver o comando pg_dump ou pg_restore.

• **Assistente de Grant**. O assistente permitirá que você altere as permissões em muitos objetos de banco de dados de uma só vez, gerando uma enorme economia de tempo.

• Motor **pgScript**. Esta é uma maneira rápida e suja para executar scripts que não precisam completar suas operações como uma transação. Com isso, você pode executar loops que façam ***commit*** de cada atualização SQL, ao contrário de funções armazenadas, as quais

requerem que todas as etapas sejam concluídas antes do trabalho ser efetivado de fato. Infelizmente, você não pode usá-lo fora da interface do pgAdmin.

• **Arquitetura de Plugin**. Recentemente desenvolvidos, os add-ons são rapidamente acessíveis com um único clique do mouse. Você pode até mesmo instalar o seu próprio Plugin.

• **pgAgent plugin**. Este agente de agendamento de trabalho multiplataforma é semelhante ao agendador de tarefas do SQL Server (SQLAgent). O pgAdmin fornece uma interface legal para isso.

MANIPULANDO OS TIPOS: FUNÇÕES E OPERADORES DO POSTGRES

O PostgreSQL é capaz de executar código do lado do servidor. Existem muitas maneiras de fornecer ao PostgreSQL o código a ser executado. Por exemplo, o usuário pode criar funções em diferentes linguagens de programação. As principais linguagens suportadas pelo PostgreSQL são as seguintes: SQL, PL/pgSQL e C.

Esses idiomas listados são os idiomas integrados; também existem outras linguagens que o PostgreSQL pode gerenciar, mas antes de usá-las, precisamos instalá-las em nosso sistema. Alguns desses outros idiomas suportados são os seguintes: PL/Python, PL/Perl, PL/tcl, PL/Java. Nesta seção, falaremos sobre as funções SQL e PL / pgSQL.

No PostgreSQL, cada função pode ser definida como VOLATILE, STABLE ou IMMUTABLE. Se não especificarmos nada, o valor padrão será VOLATILE. Uma função **VOLATILE** pode fazer qualquer coisa, incluindo modificar o banco de dados. Ele pode retornar resultados diferentes em chamadas sucessivas com os mesmos argumentos.

Uma função **STABLE** não pode modificar o banco de dados e tem a garantia de retornar os mesmos resultados, dados os mesmos argumentos para todas as linhas em uma única instrução. Uma função **IMMUTABLE** não pode modificar o banco de dados e tem a garantia de retornar os mesmos resultados dados os mesmos argumentos para sempre.

Para manipular os tipos de dados, podemos utilizar funções e alguns operadores específicos. Vamos conhecê-los agora.

FUNÇÕES

Funções permitem que você acesse rotinas especializadas de SQL. Eles tomam um ou mais argumentos e retornam um resultado.

Suponha que você queira transformar todas as letras de uma coluna em caracteres maiúsculos. Nenhum comando pode executar esta operação, mas uma função pode lidar com isso facilmente. POSTGRESQL tem uma função chamada UPPER(col), que recebe como argumento uma string e retorna o argumento em letras maiúsculas.

POSTGRESQL oferece muitas funções. As tabelas a seguir mostram as funções mais comuns, organizadas pelos tipos de dados suportados. O comando psql \df mostra todas as funções definidas e seus argumentos.

Type	Function	Example	Returns
Character String	length()	length(*col*)	length of *col*
	character_length()	character_length(*col*)	length of *col*, same as length()
	octet_length()	octet_length(*col*)	length of *col*, including multibyte overhead
	trim()	trim(*col*)	*col* with leading and trailing spaces removed
	trim(BOTH…)	trim(BOTH, *col*)	same as trim()
	trim(LEADING…)	trim(LEADING *col*)	*col* with leading spaces removed
	trim(TRAILING…)	trim(TRAILING *col*)	*col* with trailing spaces removed
	trim(…FROM…)	trim(*str* FROM *col*)	*col* with leading and trailing *str* removed
	rpad()	rpad(*col*, *len*)	*col* padded on the right to *len* characters
	rpad()	rpad(*col*, *len*, *str*)	*col* padded on the right using *str*
	lpad()	lpad(*col*, *len*)	*col* padded on the left to *len* characters
	lpad()	lpad(*col*, *len*, *str*)	*col* padded on the left using *str*
	upper()	upper(*col*)	*col* uppercased
	lower()	lower(*col*)	*col* lowercased
	initcap()	initcap(*col*)	*col* with the first letter capitalized
	strpos()	strpos(*col*, *str*)	position of *str* in *col*
	position()	position(*str* IN *col*)	same as strpos()
	substr()	substr(*col*, *pos*)	*col* starting at position *pos*
	substring(…FROM…)	substring(*col* FROM *pos*)	same as substr()
	substr()	substr(*col*, *pos*, *len*)	*col* starting at position *pos* for length *len*
	substring(…FROM… FOR…)	substring(*col* FROM pos FOR *len*)	same as substr()
	translate()	translate(*col*, *from*, *to*)	*col* with *from* changed to *to*
	to_number()	to_number(*col*, *mask*)	convert *col* to NUMERIC() based on *mask*
	to_date()	to_date(*col*, *mask*)	convert *col* to DATE based on *mask*
	to_timestamp()	to_timestamp(*col*, *mask*)	convert *col* to TIMESTAMP based on *mask*

Type	Function	Example	Returns
Number	round()	round(*col*)	round to an integer
	round()	round(*col*, *len*)	NUMERIC() *col* rounded to *len* decimal places
	trunc()	trunc(*col*)	truncate to an integer
	trunc()	trunc(*col*, *len*)	NUMERIC() *col* truncated to *len* decimal places
	abs()	abs(*col*)	absolute value
	factorial()	factorial(*col*)	factorial
	sqrt()	sqrt(*col*)	square root
	cbrt()	cbrt(*col*)	cube root
	exp()	exp(*col*)	exponential
	ln()	ln(*col*)	natural logarithm
	log()	log(*log*)	base-10 logarithm
	to_char()	to_char(*col*, *mask*)	convert *col* to a string based on *mask*

Type	Function	Example	Returns
Temporal	date_part()	date_part(*units, col*)	*units* part of *col*
	extract(...FROM...)	extract(*units* FROM *col*)	same as date_part()
	date_trunc()	date_trunc(*units, col*)	*col* rounded to *units*
	isfinite()	isfinite(*col*)	BOOLEAN indicating whether *col* is a valid date
	now()	now()	TIMESTAMP representing current date and time
	timeofday()	timeofday()	string showing date/time in Unix format
	overlaps()	overlaps(*c1, c2, c3, c4*)	BOOLEAN indicating whether *col's* overlap in time
	to_char()	to_char(*col, mask*)	convert *col* to string based on *mask*
Geometric			see psql's *\df* for a list of geometric functions
Network	broadcast()	broadcast(*col*)	broadcast address of *col*
	host()	host(*col*)	host address of *col*
	netmask()	netmask(*col*)	netmask of *col*
	masklen()	masklen(*col*)	mask length of *col*
	network()	network(*col*)	network address of *col*
NULL	nullif()	nullif(*col1, col2*)	return NULL if *col1* equals *col2*, else return *col1*
	coalesce()	coalesce(*col1, col2,...*)	return first non-NULL argument

HORA DE PRATICAR!

(Ministério da Economia – Desenvolvimento de Sistemas - 2020)

Tendo como referência o diagrama de entidade relacionamento precedente, julgue próximo item, a respeito de linguagem de definição de dados e SQL.

Considerando-se o diagrama apresentado, é correto afirmar que a execução das expressões SQL a seguir, em um SGBD PostgreSQL 9.4 ou superior, permite gerar o resultado apresentado na tabela mostrada imediatamente após a expressão.

```
insert into aluno values (1,'Fulano');
insert into aluno values (2,'Cicrano');
insert into aluno values (3,'Beltrano');

insert into disciplina (id, descricao) values (1,'Matemática');
insert into disciplina (id, descricao) values (3,'História');
insert into disciplina (id, descricao) values (5,'Geografia');

insert into matricula (aluno, disciplina,ano, nota) values (1, 1, 2020, 6.5);
insert into matricula (aluno, disciplina, ano, nota) values (1, 3, 2020, 9.5);
```

```
insert into matricula (aluno, disciplina, ano, nota) values (1, 5, 2020, 10.0);
insert into matricula (aluno, disciplina, ano, nota) values (3, 3, 2020, 8.5);
insert into matricula (aluno, disciplina, ano, nota) values (3, 1, 2020, 5.6);
insert into matricula (aluno, disciplina, ano, nota) values (3, 5, 2020, 7.7);

SELECT * FROM crosstab($$
select a.nome, d.descricao, m.nota
from matricula m left join aluno a on a.id=m.aluno
left join disciplina d on d.id=m.disciplina
order by 1,2
$$)
as final_result(
nome varchar,
geografia numeric,
historia numeric,
matematica numeric
);
```

nome	geografia	história	matemática
Beltrano	7.7	8.5	5.6
Fulano	10.0	9.5	6.5

Comentários: Alguns anos atrás, quando o PostgreSQL versão 8.3 foi lançado, uma nova extensão chamada **tablefunc** foi introduzida. Esta extensão fornece um conjunto de funções realmente interessante. Um deles é a função **crosstab**, que é usada para a criação da tabela dinâmica. Para chamar a função de crosstab , você deve primeiro habilitar a extensão tablefunc executando o seguinte comando SQL:

```
CREATE extension tablefunc;
```

A função de crosstab recebe um comando SQL SELECT como parâmetro, que deve ser compatível com as seguintes restrições:

- O SELECT deve retornar (no mínimo) 3 colunas.
- A **primeira** coluna no SELECT será o identificador de cada linha na tabela dinâmica ou resultado da consulta. Na questão, o nome do aluno faz esse papel. Observe como os nomes dos alunos (Beltrano e Fulano) aparecem na primeira coluna.
- A **segunda** coluna do SELECT representa as categorias da tabela dinâmica. No exemplo, essas categorias são as disciplinas escolares. É importante observar que os valores desta coluna **se expandirão em muitas colunas na tabela dinâmica**. Se a segunda coluna retornar cinco valores diferentes (geografia, história e assim por diante), a tabela dinâmica terá cinco colunas.
- A **terceira** coluna no SELECT representa o valor a ser atribuído a cada célula da tabela dinâmica.

O $$ é um delimitador que você usa para indicar onde a definição da função começa e termina.

O AS é um alias usado para definir o nome dada colunas e da relação de saída.

Gabarito Certo.

OPERADORES

Os operadores são semelhantes às funções. A tabela abaixo apresenta os operadores mais comuns. No psql, o comando \do mostra todos os operadores definidos e seus argumentos.

Type	Function	Example	Returns
Character String	||	*col1* || *col2*	append *col2* on to the end of *col1*
	~	*col* ~ *pattern*	BOOLEAN, *col* matches regular expression *pattern*
	!~	*col* !~ *pattern*	BOOLEAN, *col* does not match regular expression *pattern*
	~*	*col* ~* *pattern*	same as ~, but case-insensitive
	!~*	*col* !~* *pattern*	same as !~, but case-insensitive
	~~	*col* ~~ *pattern*	BOOLEAN, *col* matches LIKE pattern
	LIKE	*col* LIKE *pattern*	same as ~~
	!~~	*col* !~~ *pattern*	BOOLEAN, *col* does not match LIKE pattern
	NOT LIKE	*col* NOT LIKE *pattern*	same as !~~
Number	!	!*col*	factorial
	+	*col1* + *col2*	addition
	–	*col1* – *col2*	subtraction
	*	*col1* * *col2*	multiplication
	/	*col1* / *col2*	division
	%	*col1* % *col2*	remainder/modulo
	^	*col1* ^ *col2*	*col1* raised to the power of *col2*
Temporal	+	*col1* + *col2*	addition of temporal values
	–	*col1* – *col2*	subtraction of temporal values
	(…) OVERLAPS (…)	(*c1*, *c2*) OVERLAPS (*c3*, *c4*)	BOOLEAN indicating *cols* overlap in time
Geometric			see psql's *\do* for a list of geometric operators
Network	<<	*col1* << *col2*	BOOLEAN indicating if *col1* is a subnet of *col2*
	<<=	*col1* <<= *col2*	BOOLEAN indicating if *col1* is equal or a subnet of *col2*
	>>	*col1* >> *col2*	BOOLEAN indicating if *col1* is a supernet of *col2*
	>>=	*col1* >>= *col2*	BOOLEAN indicating if *col1* is equal or a supernet of *col2*

Vamos agora responder duas questões de prova sobre esses assuntos:

1. BANCA: FCC ANO: 2012 ÓRGÃO: TCE-AM PROVA: ANALISTA TÉCNICO DE CONTROLE EXTERNO - TECNOLOGIA DA INFORMAÇÃO

Em PostgreSQL, a função que converte a primeira letra da string informada em letra maiúscula, alterando todas as letras subsequentes dessa string para minúsculas se chama

A chgstr.

B altertext.

C initcap.

D upper.

E toupper.

Comentário. As alternativas A, B e E não correspondem a funções incluídas na lista de funções do Postgres para operações com *strings*. As alternativas C e D são definidas da seguinte forma:

initcap (string) - Converte a primeira letra de cada palavra para maiúscula e o restante para letras minúsculas. As palavras são sequências de caracteres alfanuméricos separados por caracteres não alfanuméricos.

upper (string) – Converte toda a string para letras maiúsculas.

Analisando a descrição de cada uma das funções, podemos confirmar nossa resposta na alternativa C. Vamos para a próxima questão.

Gabarito: C.

2. BANCA: FCC ANO: 2012 ORGÃO: MPE-PE - Analista Ministerial PROVA: Informática

No banco de dados *PostgreSQL*, a função COALESCE

A retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é muito semelhante ao LIKE, exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL.

B é uma expressão condicional genérica, semelhante às declarações *if/else* de outras linguagens.

C é uma declaração SELECT arbitrária, ou uma subconsulta. A subconsulta é processada para determinar se retorna alguma linha.

D retorna o primeiro de seus argumentos que não for nulo. Só retorna nulo quando todos os seus argumentos são nulos.

E permite a conversão do carimbo do tempo (*time stamp*) para uma zona horária diferente.

Comentário. Sobre a função solicitada na questão, ela retorna o primeiro argumento que não for nulo. Null é retornado somente se todos os argumentos forem nulos. Ela é frequentemente usada para substituir um valor padrão por valores nulos quando os dados são recuperados para exibição. Ela está incluída no rol de funções ou expressões condicionais presentes no Postgres. Essa lista é completada pelas expressões CASE, NULLIF, GREATEST e LEAST. Vejamos abaixo a definição de cada uma delas.

A expressão CASE do SQL é uma expressão condicional genérica, semelhante às declarações if/else em outras linguagens de programação. Vejam a sintaxe do comando abaixo:

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

A função NULLIF retorna um valor nulo se value1 é igual a value2; caso contrário, retorna o valor1. Para entender melhor, observe a sintaxe e um exemplo a seguir:

```
NULLIF(value1, value2)
```

```
SELECT NULLIF(value, '(none)') ...
```

Por fim, tempos as funções GREATEST e LEAST, que selecionam o valor maior ou menor de uma lista de expressões. Temos a resposta para a questão na alternativa D.

Gabarito: D

USANDO PROCEDURE LANGUAGE: PL/PGSQL

Nesta parte da aula, vamos falar das principais características do PL/pgSQL, uma linguagem procedural carregável para o sistema de banco de dados PostgreSQL. Os objetivos de projeto da PL/pgSQL foram para criar uma linguagem procedural que (1) possa ser usada para criar funções e procedimentos de gatilhos, (2) acrescente estruturas de controle à linguagem SQL, (3) possa realizar cálculos complexos, (4) herde todos os tipos, as funções e os operadores definidos pelo usuário, (5) possa ser definida para ser confiável pelo servidor, e (6) seja fácil de usar.

Funções criadas com PL/pgSQL podem ser usadas em qualquer lugar que uma das funções internas possam ser. Por exemplo, é possível criar funções de cálculo condicional complexo e, depois, usá-las para definir operadores ou utilizá-las em expressões de índice.

No PostgreSQL 9.0 e versões posteriores, a PL/pgSQL é instalada por padrão. No entanto, ainda é um módulo carregável, em especial para que os administradores de segurança possam optar por removê-lo conscientemente.

PL/pgSQL é uma linguagem estruturada em blocos. O texto completo da definição da função deve ser um bloco. Um bloco é definido como:

```
CREATE FUNCTION nome_da_função (p1 tipo, p2 tipo,p3 tipo, ....., pn tipo)
 RETURNS tipo AS
BEGIN
 -- lógica da função ou corpo
END;
LANGUAGE nome_da linguagem
```

Cada declaração (*declaration)* e cada instrução (*statement)* dentro de um bloco são encerradas por um ponto e vírgula. Um bloco que aparece dentro de outro bloco deve ter um ponto e vírgula depois de END, como mostrado acima. No entanto, o END final que conclui um corpo de função não necessita de um ponto e vírgula.

A label só é necessária se você deseja identificar o bloco para uso em uma declaração de EXIT, ou para qualificar os nomes das variáveis declaradas no bloco. Se um rótulo ou uma label é dado após o END, deve corresponder ao rótulo/label declarado no início do bloco.

Todas as palavras-chave são **case-insensitive**. Os identificadores são convertidos implicitamente em minúsculas, ao menos que sejam definidos entre aspas. Esse comportamento também acontece em comandos SQL comuns.

Comentários funcionam da mesma maneira no código PL/pgSQL e no SQL comum. Um traço duplo (--) inicia um comentário, que se estende para a extremidade da linha. A /* começa um bloco de comentário, que se estende até encontrar a ocorrência de */.

Qualquer declaração na seção *declarations* de um bloco pode ser um sub-bloco. Sub-blocos podem ser usados para agrupamento lógico ou para localizar variáveis associadas a um grupo menor de declarações. Variáveis declaradas em um sub-bloco vão mascarar quaisquer variáveis com mesmo nome de blocos externos durante a execução do sub-bloco. Você pode acessar as variáveis externas se você qualifica seus nomes com a etiqueta do seu bloco. Vejam o exemplo de comando abaixo para entender melhor a sintaxe na prática:

```
CREATE FUNCTION somefunc() RETURNS integer AS $$
<< outerblock >>
DECLARE
    quantity integer := 30;
BEGIN
    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 30
    quantity := 50;
    --
    -- Create a subblock
    --
    DECLARE
        quantity integer := 80;
    BEGIN
        RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 80
        RAISE NOTICE 'Outer quantity here is %', outerblock.quantity;  -- Prints 50
    END;

    RAISE NOTICE 'Quantity here is %', quantity;  -- Prints 50

    RETURN quantity;
END;
$$ LANGUAGE plpgsql;
```

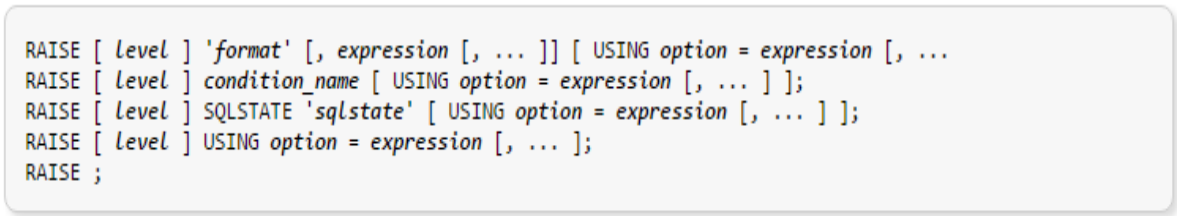
Observem a presença de um sub-bloco e a utilização da variável *quantity*. Seu valor vai depender da posição em que é chamada dentro da função. Vejam a sintaxe para declaração de variável e atribuição de valor:

name [CONSTANT] ***type*** [COLLATE ***collation_name***] [NOT NULL] [{DEFAULT|:=|=}***expression***];

A cláusula DEFAULT, caso exista, especifica o valor inicial atribuído à variável, quando o bloco é inserido. Se a cláusula DEFAULT não for fornecida, então a variável é inicializada com o valor NULL. A opção CONSTANT evita que a seja atribuído à variável outro valor após a inicialização, a fim de que o seu valor se mantenha constante durante a execução do bloco.

A opção COLLATE especifica um agrupamento de caractere relacionado à linguagem utilizada para a variável. Se NOT NULL for especificado, uma atribuição de valor nulo resulta em um erro em tempo de execução. Todas as variáveis declaradas como NOT NULL devem ter um valor padrão não nulo especificado. Existe a opção de usar o sinal de igual (=) para atribuição de valor à variável. Este sinal é usado em vez do padrão PL/SQL, que é o := (dois pontos igual).

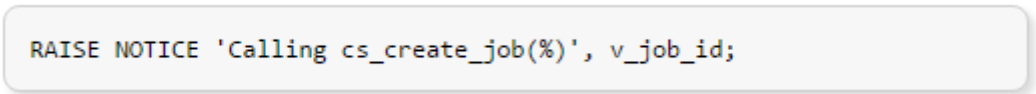
Outra característica é podermos utilizar a instrução **RAISE** para relatar mensagens e exibir erros. Veja as possibilidades de uso do comando na figura abaixo:



```
RAISE [ level ] 'format' [, expression [, ... ]] [ USING option = expression [, ...
RAISE [ level ] condition_name [ USING option = expression [, ... ] ];
RAISE [ level ] SQLSTATE 'sqlstate' [ USING option = expression [, ... ] ];
RAISE [ level ] USING option = expression [, ... ];
RAISE ;
```

A opção **level** especifica a gravidade do erro. Os níveis permitidos são DEBUG, LOG, INFO, NOTICE, WARNING e EXCEPTION, com EXCEPTION sendo o padrão. EXCEPTION gera um erro que normalmente aborta a transação corrente. Os outros níveis apenas geram mensagens de diferentes níveis de prioridade. As mensagens de um nível de prioridade podem ser informadas ao cliente, escritas no log do servidor ou ambos. Esse controle é efetuado pelas variáveis de configuração log_min_messages e client_min_messages.

Após definir o **level**, você pode escrever um *format* (que deve ser uma string literal simples, não uma expressão). A *string format* especifica o texto da mensagem de erro a ser relatada. O formato da cadeia pode ser seguido por expressões com argumentos opcionais a serem inseridos na mensagem. Dentro da string format, % passa a ter a representação de cadeia de valor do próximo argumento opcional. Veja no exemplo abaixo a substituição na prática:



```
RAISE NOTICE 'Calling cs_create_job(%)', v_job_id;
```

Você pode anexar informações adicionais para o relatório de erro por escrito, utilizando a sintaxe option = expressão. Cada expressão pode ser qualquer expressão de valor de string. As palavras-chave permitidas para opção são:

MESSAGE - Define o texto da mensagem de erro. Esta opção não pode ser usada quando o RAISE inclui uma string formatada antes do USING. Observe as descrições do comando RAISE acima.

DETAIL- Fornece uma mensagem de detalhe de erro.

HINT - Fornece uma mensagem dica para a resolução do problema.

ERRCODE - Especifica o código de erro (SQLSTATE) reportado. Vejam mais um exemplo abaixo:

```
RAISE EXCEPTION 'Nonexistent ID --> %', user_id
      USING HINT = 'Please check your user ID';
```

Vejamos agora algumas informações a respeito de estruturas de controle e definições de procedimentos de TRIGGERS com PL/pgSQL. Observem que PL/pgSQL parte da estrutura DECLARE, BEGIN e END definida anteriormente.

ESTRUTURAS DE CONTROLE

As estruturas de controle são, provavelmente, a parte mais útil (e importante) da PL/pgSQL. Com estruturas de controle, você pode manipular os dados de forma muito flexível e poderosa.

Existem dois comandos disponíveis que permitem retornar dados de uma função: RETURN e RETURN NEXT. O RETURN, seguido de uma expressão, termina a função e retorna o valor da expressão. Este é utilizado para funções PL/pgSQL que não retornam um conjunto de linhas. RETURN NEXT e RETURN QUERY não são exatamente os retornos de uma função. Eles simplesmente acrescentam zero ou mais linhas no conjunto de resultados da função.

Aqui está um exemplo de uma função usando RETURN NEXT:

```
CREATE TABLE foo (fooid INT, foosubid INT, fooname TEXT);
INSERT INTO foo VALUES (1, 2, 'three');
INSERT INTO foo VALUES (4, 5, 'six');

CREATE OR REPLACE FUNCTION get_all_foo() RETURNS SETOF foo AS
$BODY$
DECLARE
    r foo%rowtype;
BEGIN
    FOR r IN
        SELECT * FROM foo WHERE fooid > 0
    LOOP
        -- can do some processing here
        RETURN NEXT r; -- return current row of SELECT
    END LOOP;
    RETURN;
END
$BODY$
LANGUAGE plpgsql;

SELECT * FROM get_all_foo();
```

As declarações IF e CASE permitem que você execute comandos alternativamente com base em certas condições. PL/pgSQL possui três formas de IF e duas formas para o CASE:

- IF ... THEN
- IF ... THEN ... ELSE
- IF ... THEN ... ELSIF ... THEN ... ELSE
- CASE ... WHEN ... THEN ... ELSE ... END CASE
- CASE WHEN ... THEN ... ELSE ... END CASE

Com as declarações LOOP, EXIT, CONTINUE, WHILE, FOR e FOREACH, você pode mandar sua função PL/pgSQL repetir uma série de comandos. Observem o exemplo abaixo da sintaxe dos comandos LOOP, IF e EXIT utilizados em conjunto:

```
LOOP
    -- some computations
    IF count > 0 THEN
        EXIT;  -- exit loop
    END IF;
END LOOP;

LOOP
    -- some computations
    EXIT WHEN count > 0;  -- same result as previous example
END LOOP;

<<ablock>>
BEGIN
    -- some computations
    IF stocks > 100000 THEN
        EXIT ablock;  -- causes exit from the BEGIN block
    END IF;
    -- computations here will be skipped when stocks > 100000
END;
```

TRIGGRES

O PL/pgSQL pode ser usado para definir os procedimentos dos gatilhos. Um procedimento de gatilho é criado com o comando CREATE FUNCTION, declarando-o como uma função sem argumentos e um tipo de retorno de TRIGGER. Note que a função deve ser declarada sem argumentos, mesmo que você espere receber argumentos especificados no CREATE TRIGGER - argumentos do gatilho são passados pelo parâmetro TG_ARGV.

Quando uma função PL/pgSQL é chamada como um gatilho, diversas variáveis especiais são criadas automaticamente no bloco de nível superior, TG_AGRV é uma delas. Uma função de gatilho deve retornar NULL, ou um valor de registro ou linha com exatamente a estrutura da tabela que o gatilho foi disparado.

O exemplo de gatilho a seguir garante que qualquer inserção, atualização ou exclusão de uma linha na tabela emp é gravado na tabela de emp_audit. A hora atual e o nome de usuário são carimbados na linha, juntamente ao tipo de operação executada nele.

```
CREATE TABLE emp (
    empname           text NOT NULL,
    salary            integer
);

CREATE TABLE emp_audit(
    operation         char(1)   NOT NULL,
    stamp             timestamp NOT NULL,
    userid            text      NOT NULL,
    empname           text      NOT NULL,
    salary integer
);

CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$
    BEGIN
        --
        -- Create a row in emp_audit to reflect the operation performed on emp,
        -- make use of the special variable TG_OP to work out the operation.
        --
        IF (TG_OP = 'DELETE') THEN
            INSERT INTO emp_audit SELECT 'D', now(), user, OLD.*;
            RETURN OLD;
        ELSIF (TG_OP = 'UPDATE') THEN
            INSERT INTO emp_audit SELECT 'U', now(), user, NEW.*;
            RETURN NEW;
        ELSIF (TG_OP = 'INSERT') THEN
            INSERT INTO emp_audit SELECT 'I', now(), user, NEW.*;
            RETURN NEW;
        END IF;
        RETURN NULL; -- result is ignored since this is an AFTER trigger
    END;
$emp_audit$ LANGUAGE plpgsql;

CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

3. BANCA: CESPE - Analista Judiciário (STM)/Apoio Especializado/Análise de Sistemas/2018

Julgue o item subsequente, a respeito do Postgres 9.6.

Ao se criar uma trigger, a variável especial TG_OP permite identificar que operação está sendo executada, por exemplo, DELETE, UPDATE, INSERT ou TRUNCATE.

Certo

Errado

Comentário: A questão está correta. De fato, TG_OP é uma variável do tipo texto que pode ser preenchida com os valores INSERT, UPDATE, DELETE ou TRUNCATE. Eles identificam quais operações irão disparar o TRIGGER. Vejamos alguns exemplos:

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
CREATE TRIGGER emp_audit
```

```
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

Gabarito: C

ADMINISTRAÇÃO DE SERVIDORES

PARTICIONAMENTO

Em primeiro lugar, vamos tentar entender por que temos que particionar os dados. Devemos começar dizendo que uma constante comum a todos os bancos de dados é que seu tamanho sempre cresce. É, portanto, possível que um banco de dados, após alguns meses de crescimento, possa atingir o tamanho de gigabytes, terabytes ou petabytes.

Outra coisa que devemos ter sempre em mente é que nem todas as tabelas crescem na mesma taxa ou no mesmo nível; existem tabelas que serão maiores do que outras tabelas e haverá índices que serão maiores do que outros índices.

Também precisamos saber que existe uma parte da memória RAM do nosso servidor compartilhada entre todos os processos Postgres que é usada para gerenciar os dados que estão presentes nas tabelas. Esta parte da memória RAM do servidor é chamada de shared_buffers. A forma como o PostgreSQL funciona é a seguinte:

- Os dados são retirados dos discos.
- Os dados são colocados em buffers compartilhados.
- Os dados são processados em buffers compartilhados.
- Os dados são baixados para discos.

Normalmente, em um servidor dedicado apenas para PostgreSQL, o tamanho shared_buffers é cerca de um terço ou um quarto da memória RAM total do servidor. Quando uma tabela cresce excessivamente em relação ao tamanho do shared_buffers, existe a possibilidade do desempenho cair. Nesse caso, o particionamento de dados pode nos ajudar. Particionar dados significa dividir uma tabela muito grande em tabelas menores de uma forma transparente para o programa cliente. O programa cliente pensará que o servidor ainda possui apenas uma mesa. O particionamento de dados pode ser feito de duas maneiras:

- Usando herança de tabela
- Usando particionamento declarativo

O PostgreSQL 12 gerencia os seguintes tipos de particionamento de tabela:

- **Range partitioning** - quando a tabela é dividida em "intervalos". Os intervalos não devem se sobrepor e a faixa é definida por meio do uso de um campo ou conjunto de campos. Aqui estamos falando de valores numéricos.
- **List partitioning** - a tabela será particionada usando uma lista de valores. Essa lista tende a ser um conjunto discreto, por exemplo, cidades ou time de futebol.
- **Hash partitioning** - a tabela será particionada usando um valor de hash que será usado como o valor para dividir os dados em tabelas diferentes.

Para não deixar o conceitos de herança de tabelas em aberto. Vejamos a sua definição. O PostgreSQL adota o conceito de herança de bancos de dados para objetos. O conceito é muito simples e pode

ser resumido da seguinte forma: suponha que temos duas tabelas, a tabela A e a tabela B. Se definirmos a tabela A como uma tabela-mãe e a tabela B como uma tabela-filha, isso significa que todos os registros da tabela B serão ser acessível a partir da tabela A.

SEGURANÇA DA INFORMAÇÃO NO POSTGRES

Nesta parte da aula, apresentamos funcionalidades ligadas à segurança de banco de dados Postgres. Podemos listar como formas de garantir a segurança dos dados: revogar o acesso do usuário a uma tabela, concessão do acesso de usuário a uma tabela, criação de um novo usuário, impedir temporariamente um usuário se conectar, remover um usuário sem deixar apagar os seus dados, verificar se todos os usuários têm uma senha segura, limitar os poderes de superusuário a usuários específicos, criar auditoria para comandos DDL, integração com LDAP, conectar-se usando SSL e criptografia de dados confidenciais.

O PostgreSQL lida com permissões em diferentes objetos de banco de dados por meio de **listas de controle de acesso** (**ACLs**), e cada ACL contém informações sobre o conjunto de permissões, os usuários aos quais as permissões são concedidas e o usuário que as concedeu. Em termos de dados tabulares, é até possível definir permissões baseadas em coluna e permissões em nível de linha para impedir que os usuários tenham acesso a um subconjunto específico de dados.

O **Role Level Security** (RLS) é um mecanismo de aplicação de política que impede que certos papéis obtenham acesso a tuplas específicas dentro de tabelas específicas. Em outras palavras, ele aplica restrições de segurança no nível das linhas da tabela, portanto, também podemos usar o nome segurança no nível da linha.

As permissões são concedidas por funções aninhadas de maneira herdada dinamicamente ou sob demanda, deixando a opção de ajustar como um papel deve explorar os privilégios.

No que diz respeito à segurança, o PostgreSQL permite dois algoritmos diferentes para criptografia de senhas, MD5 e SCRAM-SHA-256, sendo o segundo o mais moderno e robusto. Quando configurado oportunamente, o servidor pode lidar com conexões de rede via SSL, criptografando assim todo o tráfego de rede e dados.

Antes de vermos algumas dessas questões, vamos verificar como segurança já foi cobrada em prova pelo CESPE.

1. BANCA: CESPE ANO: 2014 ÓRGÃO: ANATEL PROVA: ANALISTA ADMINISTRATIVO - SUPORTE E INFRAESTRUTURA DE TI

Julgue o item abaixo:

A conexão com o PostgreSQL 9.3 é realizada, por padrão, na porta TCP 5432. Uma das configurações de segurança permitida é o acesso por meio de SSL que é true, por padrão, e é aceito, neste caso, com o uso dos protocolos TCP, IP ou NTP.

Comentário. Vimos, no início da aula, que a porta padrão do PostgreSQL é a 5432. Até aqui a alternativa estaria correta.

O PostgresSQL possui a habilidade de usar SSL para proteger as conexões de banco de dados, criptografando todos os dados passados por essa conexão. Usando SSL, fazemos com que seja muito mais difícil de capturar o tráfego de banco de dados, incluindo nomes de usuários, senhas e dados confidenciais que são trafegados entre o cliente e o servidor. Uma alternativa ao uso de SSL está em executar a conexão através de uma VPN (Virtual Private Network).

Este método de autenticação utiliza certificados de cliente SSL para realizar a autenticação. E, portanto, só está disponível para conexões SSL. Ao usar esse método de autenticação, o servidor irá requerer que o cliente forneça um certificado válido. Nenhum aviso ou requisição de senha será enviado para o cliente. O atributo CN (COMMON NAME) do certificado será comparado com o nome do usuário do banco de dados solicitado e, se eles combinam, o login será permitido. O mapeamento de nome de usuário pode ser usado para permitir que o CN seja diferente do nome do usuário do banco de dados.

Para obter ou gerar uma chave de servidor SSL, o par de certificados para o servidor, e armazená-los para o diretório de dados do banco de dados atual, usamos os arquivos **server.key** e **server.crt**. Ele já pode ser criado em algumas plataformas. Por exemplo, no Ubuntu, Postgres está configurado para suportar conexões SSL por padrão.

Vejam que a instalação do PostgreSQL não está configurada por default para conexões SSL, apesar de possuir suporte nativo. Outro ponto é que, embora seja possível usar SSL tanto para o protocolo IP quanto sobre NTP, isso não é utilizado pelo PostgreSQL, que concentra seu uso apenas sobre o TCP.

Para utilizar SSL, o libpq lê o arquivo de configuração do OpenSSL para todo o sistema. Por padrão, esse arquivo é nomeado openssl.cnf e está localizado no diretório relatado pelo comando openssl version -d. Este padrão pode ser substituído, definindo a variável de ambiente OPENSSL_CONF para o nome do arquivo de configuração desejado.

Apenas para informação, libpq é a interface para o programador de aplicação escrita em C para PostgreSQL. libpq possui um conjunto de funções de biblioteca, que permitem os programas clientes passarem comandos para o servidor PostgreSQL e receberem os resultados dessas consultas.

Gabarito: E

TRANSAÇÕES, MVCC, WALS E PONTOS DE VERIFICAÇÃO

O PostgreSQL possui um mecanismo de transação muito rico e compatível com os padrões que permite aos usuários definir exatamente as propriedades da transação, incluindo transações aninhadas.

O PostgreSQL depende muito de transações para manter os dados consistentes em conexões simultâneas e atividades paralelas e, graças aos **registros Write-Ahead** (**WALs**), o PostgreSQL faz o possível para manter os dados seguros e confiáveis. Além disso, o PostgreSQL implementa o **Multi-Version Concurrency Control** (**MVCC**), uma forma de manter a alta simultaneidade entre as transações.

Uma transação é uma unidade atômica de trabalho que tem êxito ou falha. As transações são um recurso fundamental de qualquer sistema de banco de dados e permitem que um banco de dados implemente as propriedades do ACID: atomicidade, consistência, isolamento e durabilidade. Em conjunto, as propriedades do ACID significam que o banco de dados deve ser capaz de lidar com unidades de trabalho em sua totalidade (atomicidade), armazenar dados de forma permanente (durabilidade), sem alterações misturadas aos dados (consistência), e de uma forma que as ações simultâneas são executadas como se estivessem sozinhas (isolamento).

Chamamos de "transações implícitas" transações que o banco de dados inicia para você sem que você precise perguntar, e "transações explícitas" aquelas que você pede para o banco de dados iniciar.

Qualquer transação recebe um número exclusivo, denominado identificador de transação, ou **xid**. O sistema atribui automaticamente um **xid** a transações recém-criadas - implícitas ou explícitas - e garante que não existam duas transações com o mesmo xid no banco de dados. O PostgreSQL armazena o xid que gera e/ou modifica uma determinada tupla dentro da própria tupla. Você pode inspecionar qual é a transação atual por meio da função especial **txid_current()**. Vejamos um exemplo:

```
estrategia=> SELECT current_time, txid_current ();
    current_time     | txid_current
--------------------+--------------
 11:14:54.418179-03 |          707
(1 row)


estrategia=> SELECT current_time, txid_current ();
    current_time     | txid_current
--------------------+--------------
 11:15:38.073236-03 |          708
(1 row)
```

Como você pode ver no exemplo anterior, o sistema atribuiu dois identificadores de transação diferentes, respectivamente 707 e 708, a cada instrução, confirmando que essas instruções foram

executadas em diferentes **transações implícitas**. Você provavelmente obterá números diferentes em seu sistema.

O PostgreSQL gerencia algumas colunas ocultas diferentes que você precisa pedir explicitamente ao consultar uma tabela para poder vê-las. Em particular, cada tabela tem a xmin, xmax, cmin, e cmax colunas ocultas. Veja um exemplo de uma consulta que retorna as respectivas colunas:

```
16  SELECT xmin, xmax, * from city2 where  city_id in (6,7,8);
```

Data Output | Explain | Messages | Notifications

	xmin xid	xmax xid	city_id integer	city character varying (50)	country_id smallint	last_update timestamp without time zone
1	749	0	6	Addis Abeba	31	2006-02-15 09:45:25
2	749	0	7	Aden	107	2006-02-15 09:45:25
3	749	0	8	Adoni	44	2006-02-15 09:45:25

Veja que a transação que criou todas as linhas acima foi a mesma, de número 749. Agora, vamos executar um comando para apagar a linha 7, para isso vou iniciar uma transação explicitamente por meio do comando BEGIN em um terminal psql. Depois vou executar o comando de deleção da cidade cujo city_id é igual a 6.

```
BEGIN;
DELETE FROM city2 WHERE city_id=1000;
```

Neste momento, eu volto na tela do pgAdmin e peço para rodar novamente a consulta acima. Vejamos o que acontece ...

Data Output | Explain | Messages | Notifications

	xmin xid	xmax xid	city_id integer	city character varying (50)	country_id smallint	last_update timestamp without time zone
1	749	751	6	Addis Abeba	31	2006-02-15 09:45:25
2	749	0	7	Aden	107	2006-02-15 09:45:25
3	749	0	8	Adoni	44	2006-02-15 09:45:25

Perceba que o xmax agora possui um número de xid, que é exatamente o número da transação que vai remover o item após o commit. Assim, as transações cujos valores estão entre o xmin e xmax vão enxergar os dados desta linha. Vou aproveitar para fazer uma atualização na linha na qual o city_id = 7, e, em seguida, rodar o COMMIT

```
UPDATE city SET city='novacidade' WHERE city_id = 7;
COMMIT;
```

Vejamos como ficou a tabela depois dessas ações ...

```
3  SELECT xmin, xmax, * FROM city2 where city_id in (6,7,8);
```

Data Output | Explain | Messages | Notifications

	xmin xid	xmax xid	city_id integer	city character varying (50)	country_id smallint	last_update timestamp without time zone
1	749	0	8	Adoni	44	2006-02-15 09:45:25
2	751	0	7	novacidade	107	2006-02-15 09:45:25

Primeiramente, a linha cujo city_id era igual a 6 foi de fato removida pela transação 751 e não aparece mais no resultado. Temos ainda o valor atualizado para a linha na qual o city_id é igual a 7.

Como não temos muitas transações simultâneas de conexões diferentes não temos como visualizar o que acontece com essa variáveis em um grande banco de dados corporativo. Mas vou apresentar abaixo uma tabela que descreve as colunas do sistemas que influenciam nas transações.

Coluna	Descrição
xmin	A identidade (ID da transação) da transação de inserção para esta versão de linha. Aqui você começa a entender o MVCC. Toda vez que uma modificação é feita, uma nova linha é inserida. A versão anterior da linha vai ver excluída do banco de dados em um momento posterior por um processo conhecido como VACCUM.
xmax	Armazena o ID da transação (“xid”) da transação que excluiu a tupla. Lembre-se de que UPDATE também exclui uma tupla no PostgreSQL. Então você passa a ter um intervalo entre o xmin e xmax no qual a tupla é válida. O xmax também pode armazenar a transação que está bloqueando a tupla.
cmin	O identificador de comando (começando em zero) dentro da transação de inserção.
cmax	O identificador de comando dentro da transação de exclusão ou zero.

ÍNDICES E OTIMIZAÇÃO DE DESEMPENHO

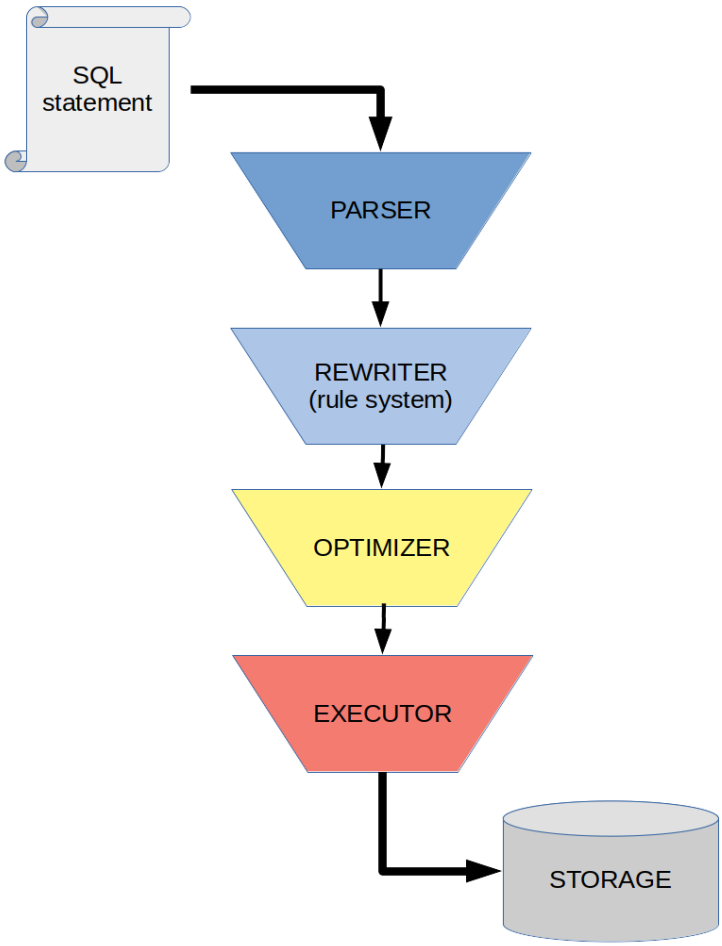
O ajuste de desempenho é uma das tarefas mais complexas do trabalho diário de um administrador de banco de dados. SQL é uma linguagem declarativa e, portanto, não define como acessar os dados subjacentes - essa responsabilidade é deixada para o mecanismo de banco de dados. O PostgreSQL, portanto, deve selecionar, para cada instrução, o melhor acesso disponível aos dados.

Um componente específico, o planejador, é responsável por decidir sobre o melhor entre todos os caminhos disponíveis para os dados subjacentes e outro componente, o otimizador, é responsável por executar a instrução com esse plano de acesso específico.

SQL é uma linguagem declarativa: você pede ao banco de dados para executar algo nos dados que ele contém, mas não especifica como o banco de dados deve completar a instrução SQL. Por exemplo, quando você pede para obter alguns dados, você executa uma instrução SELECT, mas especifica apenas as cláusulas que especificam qual subconjunto de dados você precisa, não como o banco de dados deve extrair os dados de seu armazenamento persistente.

Você tem que confiar no banco de dados - em particular, PostgreSQL - para poder fazer seu trabalho e obter o caminho mais rápido para os dados, sempre, em qualquer circunstância de carga de trabalho. A boa notícia é que o PostgreSQL é realmente bom nisso e é capaz de entender (e até certo ponto, interpretar) suas instruções SQL e sua carga de trabalho atual para fornecer a você acesso aos dados da maneira mais rápida.

No entanto, encontrar o caminho mais rápido para os dados geralmente requer um equilíbrio entre a busca pelo caminho mais rápido absoluto e o tempo gasto no raciocínio sobre esse caminho; em outras palavras, o PostgreSQL às vezes escolhe um meio-termo para obter os dados de maneira rápida o suficiente, mesmo que não seja absolutamente o mais rápido.



Para ajudar seu cluster a otimizar suas instruções, você precisa entender como o PostgreSQL lida com uma instrução SQL primeiro. Para resumir, uma única instrução SQL passa por quatro estágios, todos mostrados no diagrama a seguir: uma fase de análise que verifica a sintaxe da instrução, uma fase de reescrita que transforma a consulta em algo mais específico, a fase de otimização que decide como acessar os dados solicitados pela consulta e, por último, a fase de execução, que obtém acesso físico aos dados. Isso pode ser visualizado no seguinte diagrama:

É possível que o DBA interaja como o otimizador para encontrar a melhor forma de executar uma determinada consulta. Às vezes, é necessário a criação de índices para agilizar o acesso aos dados. O PostgreSQL fornece recursos muito ricos para a criação e gerenciamento de índices, tanto com base em coluna única quanto com base em várias colunas, bem como vários tipos de índices que podem ser construídos.

Graças ao comando EXPLAIN, um administrador de banco de dados pode inspecionar uma consulta lenta e ver como o otimizador pensou sobre qual é o melhor acesso aos dados subjacentes, e graças a uma compreensão de como o PostgreSQL funciona, o administrador pode decidir quais índices criar a fim de afinar o desempenho do banco de dados.

O PostgreSQL também fornece um **rico conjunto de estatísticas** que é usado para verificar a qualidade e a quantidade dos dados dentro de cada tabela, portanto, ser capaz de gerar um plano de execução, e monitorar quais índices são usados. A explicação automática (auto-explain) é outro módulo útil que pode ser usado para monitorar silenciosamente consultas lentas e planos de execução e ver como o cluster está funcionando sem a necessidade de executar manualmente todas as instruções suspeitas.

É importante enfatizar que o ajuste de desempenho é uma das tarefas mais complexas na administração de banco de dados e que não existe uma solução mágica ou solução única, portanto, é necessário ter experiência e muita prática.

BACKUP E RESTORE

O postgreSQL possui 2 tipos básicos de backups: lógico e físico. Os backups lógicos são feitos por meio da leitura dos dados do próprio banco de dados, por meio de interações SQL comuns. Os backups físicos são feitos por meio da clonagem do diretório PGDATA, usando ferramentas do sistema operacional ou soluções ad hoc PostgreSQL. A restauração é realizada por ferramentas específicas no caso de backups lógicos e pelo mecanismo de autocorreção do banco de dados no caso de backups físicos.

Apresentamos a seguir três utilitários do PostgreSQL para backup lógicos e restauração de banco de dados. O **pg_dump** extrai um banco de dados PostgreSQL para um arquivo script ou outro arquivo. O **pg_dumpall** tem a função de extrair um cluster de banco de dados PostgreSQL para um arquivo script. E o **pg_restore** restaura um banco de dados PostgreSQL a partir de um arquivo gerado pelo pg_dump.

O **pg_dump** é um utilitário para fazer backup de um banco de dados PostgreSQL. Faz backups consistentes, mesmo se o banco de dados está sendo usado. O pg_dump não bloqueia os outros usuários que acessam o banco de dados (leitura ou gravação).

Ele pode ter sua saída em formato de scripts ou arquivos customizados (por exemplo -Fc faz com que o arquivo seja salvo em um formato compactado). Dumps de script são arquivos de texto simples que contêm os comandos SQL necessários para reconstruir o banco de dados para o estado em que estava quando foi salvo. Para restaurar a partir de um script, carregue o mesmo no prompt do psql. Os arquivos de script podem ser usados para reconstruir o banco de dados até mesmo em outras máquinas com outras arquiteturas; com algumas modificações, até em outros bancos de dados SQL. Vejamos alguns exemplos do comandos pg_dump:

Backup	Restore
pg_dump dbname>dumpfile	psql dbname < dumpfile
pg_dump -U postgres -Fc myDB > myDB.dump	pg_restore -j 8 -U postgres -d myDB myDB.dump

A primeira linha mostra a forma mais simples de backup usando pg_dump, o arquivos salvo é em formato de script SQL. Ao lado, vemos que é possível fazer restauração usando a linha de comando do utilitário psql. Na segunda linha, temos uma forma de backup usando um tipo de arquivo customizados (Fc) que gera um arquivo com os dados compactados, o que economiza espaço. Ao lado, temos um exemplo do pg_restore, nele o parâmetro -j executa o restore em paralelo carregando múltiplas tabelas (8) ao mesmo tempo.

Os formatos de arquivo de saída mais flexíveis são o formato de "custom" (-Fc) e o formato de "diretório" (-Fd). Eles permitem a seleção e a reordenação de todos os itens arquivados, suportam a restauração paralela, e são compactados por padrão. O formato de "diretório" é o único formato que suporta descargas paralelas.

Outros dois formatos podem ser usados. O primeiro é o plano (plain -Fp), cuja saída do arquivo é um script SQL em texto plano. O outro seria o tar (-Ft), que exporta um arquivo em formato ".tar" adequado para entrada do pg_restore. O formato tar é compatível com o formato de diretório. Extrair um arquivo tar formatado produz um arquivo no formato de diretório válido. No entanto, o tar-format **não suporta a compressão** e tem um limite de 8 GB para o tamanho das tabelas individuais. Além disso, **a ordem relativa de itens de dados da tabela não pode ser mudada** durante a restauração.

Agora vamos falar do **pg_dumpall**, que é outro utilitário para fazer "dumping" de todos os bancos de dados do PostgreSQL em um arquivo script. Este contém comandos SQL que podem ser usados como entrada do psql para restaurar os bancos de dados. Ele faz isso chamando pg_dump para cada banco de dados em um cluster. O pg_dumpall também despeja os objetos globais que são comuns a todos os bancos de dados (**Obs: O pg_dump não salva estes objetos**). Isto inclui atualmente informações sobre usuários do banco de dados e grupos, *tablespaces* e propriedades, tais como permissões de acesso que se aplicam ao banco de dados como um todo.

Como o pg_dumpall lê as tabelas de todos os bancos de dados, você provavelmente vai ter de se conectar como um superusuário do banco de dados, a fim de produzir uma "descarga" completa dos dados. Também será necessário o privilégio de superusuário para executar o script salvo, a fim de possuir autorização para adicionar usuários e grupos, e para criar bases de dados.

Por fim, vamos falar do **pg_restore**, o último utilitário deste nosso bloco. Ele serve para restaurar um banco de dados PostgreSQL a partir de um arquivo gerado pelo pg_dump em qualquer um dos formatos. São executados os comandos necessários para reconstruir o banco de dados para o estado em que estava quando foi salvo. Os arquivos de backup permitem que o pg_restore seja seletivo sobre o que é restaurado, ou mesmo reordene os itens antes de restaurar. Esses arquivos de exportação são projetados para serem portáveis entre arquiteturas.

O pg_restore pode operar de dois modos. Se um nome de banco de dados for especificado, pg_restore se conecta ao banco de dados e restaura o conteúdo do arquivo diretamente no banco de dados. Caso contrário, um script contendo os comandos SQL necessários para reconstruir o banco de dados é criado e gravado em um arquivo ou na saída padrão. Este script de saída é equivalente ao formato de saída de texto sem formatação do pg_dump.

Os backups são importantes porque, mesmo em um produto testado em batalha e de alta qualidade como o PostgreSQL, as coisas podem dar errado: muitas vezes, os usuários podem danificar acidentalmente seus dados, mas outras vezes, o hardware ou o software podem falhar terrivelmente. Ser capaz de restaurar dados, parcial ou totalmente, é, portanto, muito importante e todo administrador de banco de dados deve planejar cuidadosamente as estratégias de backup.

É importante enfatizar o conceito de que um backup por si só não é válido até que seja restaurado com sucesso, portanto, para garantir que você será capaz de recuperar seu cluster, você precisa testar seus backups também.

2. BANCA: FCC - Analista em Gestão (DPE-AM)/Especializado em Tecnologia da Informação de Defensoria/Analista de Sistema/2018

No PostgreSQL 9.0, para efetuar o backup e a restauração de um banco de dados utilizam-se, respectivamente, os comandos

a) bman e rman.

b) backup e restore.

c) sqlbac e sqlrest.

d) pg_dump e psql.

e) sql_backup e sql_restore.

Comentário: Para executarmos o backup de um banco de dados PostgreSQL, devemos usar o comando pg_dump ou pg_dumpall. Já para restauração podemos fazer uso dos comandos psql ou pg_restore. Logo, nossa resposta pode ser encontrada na alternativa D. Se você quiser conhecer alguns exemplo de utilização destes comandos, sugiro que acesse este site[1].

Gabarito: D

3. BANCA: FCC ANO: 2014 ÓRGÃO: TRT - 13ª REGIÃO (PB) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

1 https://pt.wikibooks.org/wiki/PostgreSQL_Pr%C3%A1tico/Administra%C3%A7%C3%A3o/Backup_e_Restore – Obs.: Apesar de usar a versão 8.1, a maioria dos exemplos são válidos e consistentes com a versão mais recente do SGBD.

Paulo utiliza o pg_dump do PostgreSQL para fazer cópia de segurança de um banco de dados. Normalmente faz cópias de segurança no formato tar e utiliza o pg_restore para reconstruir o banco de dados, quando necessário. O pg_restore pode selecionar o que será restaurado, ou mesmo reordenar os itens antes de restaurá-los, além de permitir salvar e restaurar objetos grandes. Certo dia Paulo fez uma cópia de segurança do banco de dados chamado trt13 para o arquivo tribunal.tar, incluindo os objetos grandes. Paulo utilizou uma instrução que permitiu a seleção manual e reordenação de itens arquivados durante a restauração, porém, a ordem relativa de itens de dados das tabelas não pôde ser alterada durante o processo de restauração. Paulo utilizou, em linha de comando, a instrução

A pg_dump -Ec -h trt13 > tribunal.tar

B pg_dump -Ft -b trt13 > tribunal.tar

C pg_dump -tar -a trt13 > tribunal.tar

D pg_dump -tar -c trt13 > tribunal.tar

E pg_dump -Fp -b trt13 > tribunal.tar

Comentário. Para fazermos a questão, é preciso entender alguns detalhes do comando pg_dump. Vejam que ele deu a dica na questão: (1) ”permitir salvar e restaurar objetos grandes” e durante a restauração (2) “permitiu a seleção manual e reordenação de itens arquivados durante a restauração, porém, (3) a ordem relativa de itens de dados das tabelas não pôde ser alterada durante o processo de restauração”. Vamos então entender o que significa.

Primeiramente, para restaurar grandes objetos ou large objects, é necessário usar o parâmetro –b. Este é o comportamento padrão, exceto quando --schema, --table, ou –schema-only é especificado. Nestes casos, o -b só é útil para adicionar objetos grandes para dumps seletivos.

Quando usado com um dos formatos de ficheiros de arquivo e combinado com o pg_restore, o pg_dump fornece um mecanismo de arquivamento e transferência flexível. O pg_dump pode ser usado para fazer backup de um banco de dados inteiro. Em seguida, o pg_restore pode ser usado para examinar o arquivo e/ou selecionar as partes do banco de dados que devem ser restauradas.

Os formatos de arquivo de saída mais flexíveis são o formato de "custom" (-Fc) e o formato de "diretório" (-Fd). Eles permitem a seleção e a reordenação de todos os itens arquivados, suportam a restauração paralela, e são compactados por padrão. O formato de "diretório" é o único formato que suporta descargas paralelas.

Outros dois formatos podem ser usados. O primeiro é o plano (plain -Fp), cuja saída do arquivo é um script SQL em texto plano. O outro seria o tar (-Ft), que exporta um arquivo em formato “.tar” adequado para entrada do pg_restore. O formato tar é compatível com o formato de diretório. Extrair um arquivo tar formatado produz um arquivo no formato de diretório válido. No entanto, o tar-format **não suporta a compressão** e tem um limite de 8 GB para o tamanho das tabelas individuais. Além disso, **a ordem relativa de itens de dados da tabela não pode ser mudada** durante a restauração.

Veja que, com essas informações, podemos nos aventurar a montar nosso comando pg_dump:

pg_dump –Ft –b trt13 > tribunal.tar

Veja que, depois do –b, definimos a base de dados sobre a qual vamos fazer o dump. Em seguida, definimos após o operador “>” o arquivo no qual os dados serão gravados. Gabarito confirmado na alternativa B.

Gabarito: B.

QUESTÕES COMENTADAS – MULTIBANCAS

1. DIRENS Aeronáutica - Estágio de Adaptação à Graduação de Sargento (EEAR)/Informática/2019/EAGS 2020

Em se tratando do utilitário psql, para a criação de um banco de dados no PostgreSQL, devemos recorrer a esse utilitário para saber quais são os bancos de dados que já foram criados e quais são as tabelas e os índices existentes em cada um.

Considerando essa proposição, relacione as colunas de acordo com cada definição e assinale a alternativa correta.

1 – \?

2 – \h

3 – \di

4 – \dt

( ) Para visualizar os índices.

( ) Para obter ajuda na sintaxe de comandos SQL.

( ) Para visualizar o nome das tabelas existentes.

( ) Para obter acesso a todas as opções oferecidas pelo gerenciador.

a) 3 – 1 – 4 – 2

b) 2 – 1 – 4 – 3

c) 3 – 2 – 4 – 1

d) 4 – 1 – 2 – 3

Comentário: Tudo o que você digita no psql que começa com uma barra invertida sem citação é um meta-comando psql que é processado pelo próprio psql. Esses comandos tornam o psql mais útil para administração ou script. Meta-comandos são frequentemente chamados de comandos barra ou barra invertida. Vamos analisar cada um dos meta comandos:

O comando **\?** lista os comandos disponíveis no PostgreSQL. Não são os comandos SQL, mas sim os comandos do próprio SGBD.

O comando **\h** é utilizado para ajudar com a sintaxe de um comando SQL. Funciona assim:

\h [comando]

Se não for especificado um comando, \h irá mostrar todos os comandos SQL que possuem ajuda com a sintaxe.

O comando **\di** lista todos os índices do banco. O "i" é para indicar que queremos informações sobre os indexes.

O comando **\dt** lista o nome das tabelas. O "t" é para indicar que queremos informações sobre as tables.

Assim, temos a seguinte sequência: 3 – 2 – 4 – 1. Podemos encontrar o gabarito na alternativa C.

Gabarito: C

---

2. CEBRASPE (CESPE) - Auxiliar Judiciário (TJ PA)/Programador de Computador/2020

No sistema de gerenciamento de banco de dados PostgreSQL, para criar uma tabela contendo uma coluna com um tipo de dados inteiro e com a propriedade de autoincremento, é correto o uso de dados dos tipos

a) boolean e bigint.

b) character varying e cidr.

c) inet e integer.

d) smallint e real.

e) serial e bigserial.

Comentário: O PostgreSQL conta com três tipos de dados seriais que possibilitam a propriedade de autoincremento: ***smallserial*** (inteiro de 2 bytes), ***serial*** (inteiro de 4 bytes) e ***bigserial*** (inteiro de 8 bytes). Logo, temos a respos ta na alternativa E.

Gabarito: E

---

3. IBFC - Técnico Judiciário (TRE PA)/Apoio Especializado/Operação de Computadores/2020

Quanto às principais características do PostgreSQL, analise as afirmativas abaixo e dê valores Verdadeiro (V) ou Falso (F).

( ) não impõe limites no tamanho de armazenamento dos tipos de dados.

( ) suporta um único tipo de índice, denominado índice em cluster (clustered index).

( ) o PostgreSQL possui o maior TCO (Total Cost of Ownership) dos SGBD's.

Assinale a alternativa que apresenta a sequência correta de cima para baixo.

a) V, F, F

b) V, V, F

c) F, V, V

d) F, F, V

Comentário: Vamos comentar cada um dos itens acima.

I. VERDADEIRO. O PostgreSQL utiliza a técnica TOAST (The Oversized-Attribute Storage Technique), que permite que campos grandes sejam armazenados mesmo se for necessário que parte do dado seja armazenada em um bloco diferente da memória. Isso faz com que não exista uma limitação de tamanho por parte do SGBD, mas sim uma limitação do hardware existente na máquina na qual o banco de dados esteja executando.

II. FALSO. O PostgreSQL oferece vários tipos de índices, são eles: **B-tree, Hash, GiST and GIN**.

III. FALSO. O TCO (Total Cost of Ownership) é uma estimativa financeira que avalia os custos relacionados à compra de um investimento. Existem vários SGBDs pagos no mercado, como o Oracle e o Microsoft SQL Server. O PostgreSQL, no entanto, é um banco de dados gratuito e justamente por isso possui um TCO mais baixo.

Assim, temos nosso gabarito na alternativa A.

Gabarito: A.

---

4. NC-UFPR - Profissional Nível Universitário Jr (ITAIPU)/Gestão da Informação/2019

Considere a seguinte instrução SQL:

```
WITH RECURSIVE cte(n) AS (
  SELECT 1
  UNION ALL
  SELECT n+1 FROM cte WHERE n<5
)
SELECT * FROM cte;
```

Ao ser executada no PostgreSQL, ela produz como resultado:

a) 0,1,2,3,4

b) 1,1,2,3,4

c) 1,2,3,4,5

d) 1,2,3,4

e) 1

Comentário: Primeiramente vamos observar a instrução da questão.

```
WITH RECURSIVE cte(n) AS (
  SELECT 1
  UNION ALL
  SELECT n+1 FROM cte WHERE n<5
)
```

SELECT * FROM cte;

CTE (Common table expression) é um conjunto de dados **temporário**, que é nomeado (recebe um nome), deriva de consultas e pode ser usado em declarações de **leitura, escrita, atualização ou exclusão de dado**s (SELECT/UPDATE/INSERT/DELETE). A CTE existe apenas até a execução da próxima declaração SQL. No caso da questão, temos um CTE do tipo **recursivo**, que é uma cte que refere a si mesma e, na prática, é muito útil na consulta de dados hierárquicos (ex: organogramas).

Geralmente, uma **CTE recursiva** possui **três partes**. (1) Uma **consulta inicial** que retorna o conjunto de resultados base do CTE. A consulta inicial também é chamada de âncora. (2) Uma **consulta recursiva** que referencia a expressão de tabela comum, portanto, é chamada de membro recursivo. **O membro recursivo é unido ao membro âncora usando o operador UNION ALL.** (3) Uma **condição de finalização** especificada no membro recursivo que finaliza a execução.

Deste modo, **a ordem de execução**, na nossa questão, será a seguinte:

A consulta inicial (SELECT 1) retorna **1**

A consulta recursiva usará o conjunto de resultados de entrada da iteração e retornará um conjunto de resultados até que a condição seja atendida (**n<5**). Mas **como a consulta recursiva faz a soma de n+1, o último resultado será 5**.

Todos os **resultados serão combinados e exibidos**.

Por fim, o resultado exibido será: **1,2,3,4,5. O que nos leva ao gabarito na alternativa C**.

Gabarito: C

---

5. NC-UFPR - Profissional Nível Universitário Jr (ITAIPU)/Gestão da Informação/2019

Em relação à busca por texto utilizando os operadores LIKE, SIMILAR TO, expressão regular, Full Text Search (FTS), funções e operadores relacionados à busca textual no PostgreSQL, é correto afirmar:

a) O operador LIKE realiza busca por semelhança de palavras, resolvendo o problema de ortografia incorreta.

b) O operador SIMILAR TO realiza busca baseada em expressões regulares, realizando o ranqueamento de semelhança entre as palavras do resultado em relação às palavras da busca.

c) Erros de ortografia podem ser tratados pelo mecanismo de busca FTS, por semelhança entre os termos.

d) O operador @@ do PostgreSQL é equivalente ao operador RLIKE do MySQL.

e) Os tipos tsquery e tsvector, criados respectivamente pelas funções to_tsquery e to_tsvector, são conjuntos de trigrans das strings informadas, que são comparadas em ordem alfabética durante a busca.

Comentário: Vamos comentar cada uma das alternativas:

a) **ERRADA**. O operador **LIKE** realiza busca por semelhança de palavras. Essa é sua principal finalidade, achar registros que possuem um padrão na sua cadeia de caracteres. Porém não resolve o problema de ortografia incorreta, pois é só um mecanismo de busca.

b) **ERRADA.** O operador **SIMILAR TO** realiza busca baseada em expressões regulares, realizando o ranqueamento de semelhança entre as palavras do resultado em relação às palavras da busca. O operador SIMILAR TO no postgres é usado com a mesma finalidade que o operador LIKE, realizar busca por semelhança de palavra através de um padrão na sua cadeia de caracteres.

A **diferença** entre o SIMILAR TO e o LIKE é que enquanto o LIKE usa um padrão de busca através de um SQL, o **SIMILAR TO é baseado em expressões regulares** (ReGeX). Porém não há "ranqueamento de semelhança entre as palavras do resultado em relação às palavras da busca" como diz na afirmativa.

c) **CERTA**. Erros de ortografia podem ser tratados pelo mecanismo de busca FTS, por semelhança entre os termos. O FTS (full text search) é um motor de busca que usa técnica de pesquisa e recuperação de informações de texto armazenada em banco de dados. Ela usa uma linguagem natural como critério para busca em banco (consultas) e pode ordená-la por relevância da consulta. Ele usa o conceito de motor de busca de site de busca, como google e bing. O FTS as seguintes características:

Ranqueamento / Atribuição de peso

Suporte a vários idiomas

Pesquisa Fuzzy para erros de ortografia

Apoio a acentuação

d) **ERRADA.** O operador @@ é usado na FTS. @@ é um operador de correspondência que retorna TRUE (verdadeiro) se um **tsvector** (documento) corresponder a um **tsquery** (consulta).

Um valor tsvector é uma lista ordenada de lexemas distintos o qual são palavras que foram normalizadas para fazer variações diferentes de uma mesma palavra e que são parecidas. Por exemplo, açoitar, açoite, abdome, abdômen o tsvector organiza estes resultados para que uma palavra que possua variações não seja contada/exibida mais de uma vez no resultado.

Para a correção ortográfica é usado o módulo **PG_TRGM.**

O RLIKE no mysql é um operador que tem a mesma função do SIMILAR TO no postgres, que usa o padrão de expressões regulares (RLIKE = Regex LIKE). Então, para tornar a alternativa correta, teríamos que ter: "O operador SIMILAR TO do PostgreSQL é equivalente ao operador RLIKE do MySQL."

e) ERRADA. TSQUERY é um formato do tipo consulta para o FTS e é criado pela função to_tsquery. TSVECTOR é um formato do tipo documento para o FTS e é criado pela função to_tsvector. Não são conjuntos de "TRIGAMS", mas usam o conceito dela

no motor de busca. Para uso nas buscas, são comparadas com o operador @@ e não necessariamente há a comparação em ordem alfabética

Gabarito: C

---

6. VUNESP - Técnico de Tecnologia da Informação (UFABC)/2019

No Sistema Gerenciador de Bancos de Dados PostgreSQL (versão 9), há um comando que exibe o plano que o gerenciador irá utilizar para realizar uma determinada consulta.

O comando descrito é o

a) EXPLAIN.

b) OFFSET.

c) NOTIFY.

d) LOCK.

e) FETCH.

Comentário: Vamos analisar cada uma das alternativas:

a) CORRETA. O comando EXPLAIN é utilizado para exibir o plano de execução de uma determinada consulta.

b) ERRADA. A cláusula OFFSET serve para retornar apenas uma parte do resultado de uma consulta, indicando quantas linhas a partir do início do resultado serão ignoradas.

c) ERRADA. O comando NOTIFY envia uma notificação aos clientes que se inscreveram para receber notificações através do comando LISTEN.

d) ERRADA. LOCK é um comando utilizado para travar uma tabela.

e) ERRADA. O comando FETCH é utilizado para retornar linhas de uma consulta utilizando um cursor criado previamente.

Gabarito: A.

---

7. CEBRASPE (CESPE) - Analista de Gestão de Resíduos Sólidos (SLU DF)/Informática/2019

No que diz respeito a ferramentas de desenvolvimento, julgue o item a seguir.

O PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional (ORDBMS) que oferece suporte a tipos de dados especializados como o JSon e o JSonb.

Comentário: O PostgreSQL é um sistema de banco de dados objeto-relacional, de código aberto, que usa a linguagem SQL, sendo altamente extensível, com suporte ao tipo de dados JSON desde a versão 9.2, permitindo a manipulação de dados no

formato JSON de maneira eficiente, podendo armazenar documentos JSON no dois tipos de dados disponíveis: JSON e JSONB.

A diferença entre eles é que os dados do tipo JSON são armazenados da mesma forma como são inseridos, em texto, o que causa lentidão nas consultas. Mas, com JSONB, o texto é processado no momento da inserção e armazenado em formato binário, com isso as consultas ficam mais rápidas.

Gabarito: C.

8. VUNESP - Analista de Tecnologia da Informação (Pref Olímpia)/2019

O sistema gerenciador de bancos de dados PostgreSQL (versão 9.5) possui os seguintes modos de desligamento (shutdown):

a) Hard, Premium e Single.

b) Hibernate, Full e Soft.

c) Initial, Intermediate e Final.

d) Partial, Permanent e Semi-permanent.

e) Smart, Fast e Immediate.

Comentário: Existem três modos de desligamento de um servidor PostgreSQL:

Smart: o modo smart desabilita novas conexões, mas deixa as sessões existentes terminarem seu trabalho normalmente. O servidor é desligado após todas as sessões finalizarem seus trabalhos.

Fast: o modo fast desabilita novas conexões e envia um sinal para que todos os processos abortem suas transações atuais e finalizem seus trabalhos.

Immediate: o modo immediate é o menos "amigável". Ele envia um sinal para que todos os processos terminem. Se isso não ocorrer em até 5 segundos, é enviado um sinal de kill para finalizar o processo imediatamente.

A única alternativa que traz os três modos é a letra e) Smart, Fast e Immediate.

Gabarito: E.

9. CEBRASPE (CESPE) - Analista de Tecnologia da Informação (TCE-RO)/Desenvolvimento de Sistemas/2019

Em geral, a sintaxe para a criação de índice em banco de dados relacional segue uma estrutura- padrão, como demonstra, por exemplo, a seguinte estrutura no banco relacional PostgreSQL, em versão 9 ou superior.

CREATE [UNIQUE] INDEX [CONCURRENTLY] [name] ON table [USING method]

Tendo como referência essas informações, assinale a opção correta.

a) CREATE INDEX constrói uma linha de índice de acordo com uma coluna específica da tabela.

b) O parâmetro method depende do tamanho da tabela e não deve ser utilizado se o tamanho da tabela for menor que 1 MB.

c) Um campo de índice não pode ser uma expressão calculada a partir dos valores de uma ou mais colunas da tabela.

d) O método de indexação btree armazena dados de forma que cada nó contenha chaves em ordem crescente.

e) Quando a cláusula WHERE está presente, um índice total é criado, porque a cláusula já é restritiva na operação de selecionar dados ou de inserir dados.

Comentário: Vamos comentar cada uma das alternativas:

a) ERRADA. O CREATE INDEX constrói **um índice** nas colunas especificadas da tabela especificada. Os índices são usados principalmente para aprimorar o desempenho do banco de dados (embora o uso inadequado possa resultar em desempenho mais lento).

b) ERRADA. O parâmetro method especifica qual algoritmo é utilizado pelo índice e não depende do tamanho da tabela. As opções são B-tree, hash, GiST e GIN. Quando não é especificado, o tipo B-tree é criado.

c) ERRADA. Os campos-chave do índice são especificados como nomes de colunas ou, alternativamente, como expressões escritas entre parênteses. Vários campos podem ser especificados se o método index suportar índices de várias colunas. Um campo de índice pode ser **uma expressão calculada a partir dos valores de uma ou mais colunas da linha da tabela.** Esse recurso pode ser usado para obter acesso rápido aos dados com base em alguma transformação dos dados básicos.

d) CERTO. B-tree (ou árvore B) é uma estrutura de dados em árvore auto balanceada. Uma das propriedades da árvore B é que todas as chaves são armazenadas em ordem crescente, de forma que esse tipo de índice é recomendado para dados que podem ser ordenados de alguma forma.

e) ERRADA. Quando a cláusula WHERE está presente, um índice parcial é criado. Ou seja, é criado um índice em apenas uma parte da tabela (a parte que satisfaz a condição da cláusula WHERE).

Gabarito: D

10. CEBRASPE (CESPE) - Analista Judiciário (TJ AM)/Analista de Sistemas/2019

A respeito de bancos de dados relacionais, julgue o item a seguir.

Em um banco de dados PostgreSQL, a manipulação de ROLES é feita exclusivamente por comandos CREATE e DROP fornecidos com o banco de dados.

Comentário: Um ROLE (papel) é um conjunto de permissões (ou privilégios) que pode ser atribuído aos usuários do banco. No PostgreSQL, a manipulação dos ROLES é feita através dos comandos CREATE ROLE, ALTER ROLE, SET ROLE e DROP ROLE. Note que a assertiva erra ao afirmar que a manipulação é feita **exclusivamente** pelos comandos CREATE e DROP.

Gabarito: E.

11. IDECAN - Técnico (IF Baiano)/Tecnologia da Informação /2019

Sobre o PostgreSQL, assinale a alternativa correta.

a) Só é possível instalar o PostgreSQL com privilégios de super usuário, ou seja, é necessário o acesso de usuário root do sistema.

b) Existem três abordagens fundamentais de backup no PostgreSQL: SQL dump, backup a nível de arquivos e arquivamento contínuo.

c) O comando para se efetuar um dump de um banco de dados chamado dbname gerando um arquivo dumpfile é o seguinte: psql dbname < dumpfile.

d) O comando createdb mydb, considerando que o PostgreSQL está instalado corretamente, cria uma tabela de nome mydb.

e) O comando dropdb mydb remove todos os arquivos associados a um banco de dados. No entanto, o PostgreSQL cria um backup antes.

Comentário: Vamos comentar cada uma das alternativas:

a) ERRADA. Para simplesmente instalar o postgreSQL em um sistema operacional, não é necessário acesso de super usuário.

b) CERTA. No PostgreSQL, há três diferentes métodos para realização de backups: SQL Dump, Filesystem level Backup (backup a nível de arquivos) e Continuous Archive (arquivamento contínuo). O SQL Dump é realizado através da ferramenta pg_dump, sendo considerada a mais simples de ser implementada. O backup a nível de arquivos, por sua vez, consiste em realizar um backup do diretório PostgreSQL, através de comandos como tar, cp, rsync. Por último, o arquivamento contínuo se refere às cópias de logs arquivados no diretório $PGDATA/pg_xlog, ou seja, é realização de backup a nível de arquivos do diretório de logs de transação.

c) ERRADA. É utilizado o comando **pg_dump**, da seguinte forma:

pg_dump [opção...] [nome_do_banco_de_dados]

d) ERRADA. Para criação de tabelas, é usado o comando CREATE TABLE.

e) ERRADA. O utilitário dropdb de fato remove um banco de dados, mas não realiza backup, conforme dito na questão.

Gabarito: B

12. BANCA: CESPE ANO: 2014 ÓRGÃO: ANATEL PROVA: ANALISTA ADMINISTRATIVO - SUPORTE E INFRAESTRUTURA DE TI

A respeito de banco de dados, julgue os itens que se seguem.

O PostgreSQL 9.3, ao gerenciar o controle de concorrência, permite o acesso simultâneo aos dados. Internamente, a consistência dos dados é mantida por meio do MVCC (*multiversion concurrency control*), que impede que as transações visualizem dados inconsistentes.

Comentário. O PostgreSQL fornece um rico conjunto de ferramentas para os desenvolvedores gerenciarem o acesso simultâneo aos dados. Internamente, a consistência dos dados é mantida por meio de um **modelo de concorrência multiversão** (MVCC). Isto significa que, ao consultar um banco de dados, **cada transação enxerga uma fotografia dos dados** (ou uma versão do banco de dados) em algum momento passado, independentemente do estado atual dos dados subjacentes ou relacionados.

Isso protege a transação de enxergar dados inconsistentes. Transações simultâneas sobre as mesmas linhas de dados poderiam levar o banco de dados a um estado inválido. O MVCC acaba fornecendo um isolamento entre as transações para cada sessão de banco de dados. Assim, por **não** utilizar as metodologias de **bloqueio** de sistemas de banco de dados tradicionais, **minimiza a disputa** pelos objetos de dados, a fim de permitir um bom desempenho em ambientes multiusuários.

A principal vantagem de usar o modelo MVCC no controle de concorrência, em vez de bloqueios de itens de dados é que, no MVCC, bloqueios adquiridos para consulta de dados (leitura) não conflitam com os bloqueios adquiridos para a gravação de dados. Desta forma, **a leitura nunca bloqueia a escrita**, e a escrita nunca bloqueia a leitura. O PostgreSQL mantém essa garantia mesmo quando fornece um nível mais rigoroso de isolamento de transações.

Gabarito: C.

13. BANCA: FCC ANO: 2012 ÓRGÃO: TCE-AM PROVA: ANALISTA TÉCNICO DE CONTROLE EXTERNO - TECNOLOGIA DA INFORMAÇÃO

Sobre os fundamentos arquiteturais do banco de dados PostgreSQL, considere:

I. Utiliza um modelo cliente/servidor, consistindo de um processo servidor que gerencia os arquivos do banco de dados, controla as conexões dos clientes ao banco dados e efetua ações no banco de dados em favor dos clientes.

II. A aplicação cliente, que irá efetuar as operações no banco de dados, poderá ser de diversas naturezas, como uma ferramenta em modo texto, uma aplicação gráfica, um servidor web que acessa o banco de dados para exibir as páginas ou uma ferramenta de manutenção especializada.

III. A aplicação cliente pode estar localizada em uma máquina diferente da máquina em que o servidor está instalado. Neste caso, a comunicação entre ambos é efetuada por uma conexão TCP/IP. O servidor pode aceitar diferentes conexões dos clientes ao mesmo tempo.

Está correto o que se afirma em

A I, II e III.

B I e II, apenas.

C I e III, apenas.

D II e III, apenas.

E III, apenas.

Comentário. Observem que, segundo o conteúdo apresentado acima, podemos afirmar que as alternativas I, II e III estão corretas. A alternativa II foi dada como incorreta pelo examinador. Mas, se olharmos o texto abaixo, retirado do manual de referência do PostgreSQL, veremos que ela nada mais é do que uma tradução do texto:

"The user's client (frontend) application that wants to perform database operations. Client applications can be very diverse in nature: a client could be a text-oriented tool, a graphical application, a web server that accesses the database to display web pages, or a specialized database maintenance tool. Some client applications are supplied with the PostgreSQL distribution; most are developed by users."

Gabarito: A

14. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Localização refere-se ao fato de uma aplicação respeitar as preferências culturais sobre alfabetos, classificação, formatação de números etc. PostgreSQL usa o padrão ISO C e POSIX fornecidos pelo sistema operacional do servidor para aplicar as regras de localização. O suporte à localização é automaticamente inicializado quando um cluster de banco de dados é criado usando o comando

A create cluster.

B create database.

C initdb.

D ccluster.

E locale init.

Comentário. Vamos fazer alguns comentários sobre as alternativas. Na alternativa A, temos o comando CREATE CLUSTER. Ele não existe dentro do rol de comandos do SGBD. O comando que existe é o **CLUSTER**. Ele é utilizado basicamente para reordenar uma tabela fisicamente, de acordo com as informações de um dos índices. Vejamos a sintaxe do comando:

CLUSTER [VERBOSE] table_name [USING index_name]

Sobre as alternativas B e C, acabamos de definí-las acima. A alternativa D trata de um comando que não existe no Postgres. O utilitário (utility) existente é o **clusterdb**, usado como **wrapper** para o comando CLUSTER que acabamos de falar.

Na letra E, temos mais uma expressão que não existe na documentação do Postgres: ”locale init”. Mas sabemos que o suporte a *locale* é algo imprescindível dentro de qualquer SGBD. Vejamos então como o suporte a locale é tratado.

O suporte a *locale* refere-se ao fato de as aplicações respeitarem características culturais, como alfabeto, ordenação, formato de numeração ou moedas etc. O PostgreSQL usa o padrão ISO C e POSIX fornecidos pelo sistema operacional do servidor para aplicar as regras de localização.

O suporte a *Locale* é automaticamente inicializado quando um cluster de banco de dados é criado usando o initdb. Ele inicializará o cluster de banco de dados com a definição do seu ambiente de execução por default. Por isso, se o seu sistema já está definido para utilizar o idioma que você quer usar em seu cluster de banco de dados, então não há mais nada que você precise fazer. Se você quiser usar um Locale diferente (ou você não tem certeza de qual Locale está definido para sistema), você pode instruir initdb qual *locale* usar, especificando a opção --locale.

Por exemplo:

initdb --locale = sv_SE

Neste exemplo, para sistemas Unix definimos o idioma para Sueco (sv) que é falado na Suécia (SE).

Gabarito: C

15. BANCA: FCC ANO: 2014 ÓRGÃO: TJ-AP PROVA: ANALISTA JUDICIÁRIO - BANCO DE DADOS - DBA

Um dos itens da administração do sistema gerenciador de bancos de dados PostgreSQL (V.9.3.4) refere-se a gerenciar informações sobre os bancos de dados por ele controlados. O PostgreSQL contém algumas visões que auxiliam nessa tarefa, dentre elas, a visão pg_settings que contém dados sobre

A os parâmetros run-time do servidor.

B estatísticas das tabelas do servidor.

C os usuários dos bancos de dados.

D a lista de bloqueios impostos.

E a lista das funções presentes no banco de dados.

Comentário. A questão pergunta sobre os dados que são armazenados na visão pq_setting. Sabemos o que são os dados de run-time, mas quais exatamente? Segue uma tabela com os parâmetros, os tipos e a descrição das informações contidas na visão.

Nome	Tipo	Descrição
name	text	Nome do parâmetro de configuração
setting	text	O valor atual do parâmetro
unit	text	Unidade implícita do parâmetro
category	text	Grupo lógico do parâmetro
short_desc	text	Uma breve descrição do parâmetro
extra_desc	text	Descrição adicional, mais detalhada do parâmetro
context	text	Contexto necessário para definir o valor do parâmetro.
vartype	text	Tipo de parâmetro (bool, enum, integer, real, ou string)
source	text	Fonte do valor do parâmetro atual
min_val	text	Valor mínimo permitido do parâmetro (null para valores não-numéricos)
max_val	text	Valor máximo permitido do parâmetro (null para valores não-numéricos)
enumvals	text[]	Os valores permitidos de um parâmetro de enum (NULL para valores não-ENUM)
boot_val	text	O valor do parâmetro assumido na inicialização do servidor, se o parâmetro não for definido de outro modo
reset_val	text	Valor que o RESET iria redefinir para o parâmetro na sessão atual

Gabarito: A

16. BANCA: CESPE ANO: 2014 ÓRGÃO: ANATEL PROVA: ANALISTA ADMINISTRATIVO - SUPORTE E INFRAESTRUTURA DE TI

Julgue o item abaixo:

A conexão com o PostgreSQL 9.3 é realizada, por padrão, na porta TCP 5432. Uma das configurações de segurança permitida é o acesso por meio de SSL que é true, por padrão, e é aceito, neste caso, com o uso dos protocolos TCP, IP ou NTP.

Comentário. Vimos, no início da aula, que a porta padrão do PostgreSQL é a 5432. Até aqui a alternativa estaria correta.

O PostgresSQL possui a habilidade de usar SSL para proteger as conexões de banco de dados, criptografando todos os dados passados por essa conexão. Usando SSL, fazemos com que seja muito mais difícil de capturar o tráfego de banco de dados, incluindo nomes de usuários, senhas e dados confidenciais que são trafegados entre o cliente e o servidor. Uma alternativa ao uso de SSL está em executar a conexão através de uma VPN (Virtual Private Network).

Este método de autenticação utiliza certificados de cliente SSL para realizar a autenticação. E, portanto, só está disponível para conexões SSL. Ao usar esse método de autenticação, o servidor irá requerer que o cliente forneça um certificado válido. Nenhum aviso ou requisição de senha será enviado para o cliente. O atributo CN (COMMON NAME) do

certificado será comparado com o nome do usuário do banco de dados solicitado e, se eles combinam, o login será permitido. O mapeamento de nome de usuário pode ser usado para permitir que o CN seja diferente do nome do usuário do banco de dados.

Para obter ou gerar uma chave de servidor SSL, o par de certificados para o servidor, e armazená-los para o diretório de dados do banco de dados atual, usamos os arquivos **server.key** e **server.crt**. Ele já pode ser criado em algumas plataformas. Por exemplo, no Ubuntu, Postgres está configurado para suportar conexões SSL por padrão.

Vejam que a instalação do PostgreSQL não está configurada por default para conexões SSL, apesar de possuir suporte nativo. Outro ponto é que, embora seja possível usar SSL tanto para o protocolo IP quanto sobre NTP, isso não é utilizado pelo PostgreSQL, que concentra seu uso apenas sobre o TCP.

Para utilizar SSL, o libpq lê o arquivo de configuração do OpenSSL para todo o sistema. Por padrão, esse arquivo é nomeado openssl.cnf e está localizado no diretório relatado pelo comando openssl version -d. Este padrão pode ser substituído, definindo a variável de ambiente OPENSSL_CONF para o nome do arquivo de configuração desejado.

Apenas para informação, libpq é a interface para o programador de aplicação escrita em C para PostgreSQL. libpq possui um conjunto de funções de biblioteca, que permitem os programas clientes passarem comandos para o servidor PostgreSQL e receberem os resultados dessas consultas.

Gabarito: E

17. BANCA: FCC - Analista em Gestão (DPE AM)/Especializado em Tecnologia da Informação de Defensoria/Analista de Sistema/2018

No PostgreSQL 9.0, para efetuar o backup e a restauração de um banco de dados utilizam-se, respectivamente, os comandos

a) bman e rman.

b) backup e restore.

c) sqlbac e sqlrest.

d) pg_dump e psql.

e) sql_backup e sql_restore.

Comentário: Para executarmos o backup de um banco de dados PostgreSQL, devemos usar o comando pg_dump ou pg_dumpall. Já para restauração podemos fazer uso dos comandos psql ou pg_restore. Logo, nossa resposta pode ser encontrada na alternativa D.

Se você quiser conhecer alguns exemplo de utilização destes comandos, sugiro que acesse este site[1].

Gabarito: D

18. BANCA: FCC ANO: 2014 ÓRGÃO: TRT - 13ª REGIÃO (PB) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Paulo utiliza o pg_dump do PostgreSQL para fazer cópia de segurança de um banco de dados. Normalmente faz cópias de segurança no formato tar e utiliza o pg_restore para reconstruir o banco de dados, quando necessário. O pg_restore pode selecionar o que será restaurado, ou mesmo reordenar os itens antes de restaurá-los, além de permitir salvar e restaurar objetos grandes. Certo dia Paulo fez uma cópia de segurança do banco de dados chamado trt13 para o arquivo tribunal.tar, incluindo os objetos grandes. Paulo utilizou uma instrução que permitiu a seleção manual e reordenação de itens arquivados durante a restauração, porém, a ordem relativa de itens de dados das tabelas não pôde ser alterada durante o processo de restauração. Paulo utilizou, em linha de comando, a instrução

A pg_dump -Ec -h trt13 > tribunal.tar

B pg_dump -Ft -b trt13 > tribunal.tar

C pg_dump -tar -a trt13 > tribunal.tar

D pg_dump -tar -c trt13 > tribunal.tar

E pg_dump -Fp -b trt13 > tribunal.tar

Comentário. Para fazermos a questão, é preciso entender alguns detalhes do comando pg_dump. Vejam que ele deu a dica na questão: (1) ”permitir salvar e restaurar objetos grandes” e durante a restauração (2) “permitiu a seleção manual e reordenação de itens arquivados durante a restauração, porém, (3) a ordem relativa de itens de dados das tabelas não pôde ser alterada durante o processo de restauração”. Vamos então entender o que significa.

Primeiramente, para restaurar grandes objetos ou large objects, é necessário usar o parâmetro –b. Este é o comportamento padrão, exceto quando --schema, --table, ou –schema-only é especificado. Nestes casos, o -b só é útil para adicionar objetos grandes para dumps seletivos.

Quando usado com um dos formatos de ficheiros de arquivo e combinado com o pg_restore, o pg_dump fornece um mecanismo de arquivamento e transferência flexível. O pg_dump pode ser usado para fazer backup de um banco de dados inteiro. Em seguida, o pg_restore

[1] https://pt.wikibooks.org/wiki/PostgreSQL_Pr%C3%A1tico/Administra%C3%A7%C3%A3o/Backup_e_Restore – Obs.: Apesar de usar a versão 8.1, a maioria dos exemplos são válidos e consistentes com a versão mais recente do SGBD.

pode ser usado para examinar o arquivo e/ou selecionar as partes do banco de dados que devem ser restauradas.

Os formatos de arquivo de saída mais flexíveis são o formato de "custom" (-Fc) e o formato de "diretório" (-Fd). Eles permitem a seleção e a reordenação de todos os itens arquivados, suportam a restauração paralela, e são compactados por padrão. O formato de "diretório" é o único formato que suporta descargas paralelas.

Outros dois formatos podem ser usados. O primeiro é o plano (plain -Fp), cuja saída do arquivo é um script SQL em texto plano. O outro seria o tar (-Ft), que exporta um arquivo em formato ".tar" adequado para entrada do pg_restore. O formato tar é compatível com o formato de diretório. Extrair um arquivo tar formatado produz um arquivo no formato de diretório válido. No entanto, o tar-format **não suporta a compressão** e tem um limite de 8 GB para o tamanho das tabelas individuais. Além disso, **a ordem relativa de itens de dados da tabela não pode ser mudada** durante a restauração.

Veja que, com essas informações, podemos nos aventurar a montar nosso comando pg_dump:

pg_dump –Ft –b trt13 > tribunal.tar

Veja que, depois do –b, definimos a base de dados sobre a qual vamos fazer o dump. Em seguida, definimos após o operador ">" o arquivo no qual os dados serão gravados. Gabarito confirmado na alternativa B.

Gabarito: B.

19. BANCA: FCC ANO: 2013 ORGÃO: TRT - 12ª Região (SC) PROVA: Analista Judiciário - Tecnologia da Informação

O comando em SQL capaz de serializar dados de uma tabela para um arquivo em disco, ou efetuar a operação contrária, transferindo dados de um arquivo em disco para uma tabela de um banco de dados, é o comando:

(a) COPY.

(b) TRANSFER.

(c) SERIALIZE.

(d) FILE TRANSFER.

(e) EXPORT.

Comentário. Analisando cada uma das alternativas. O comando **copy** permite escrever o conteúdo de uma tabela em um arquivo ASCII ou carregar uma tabela a partir de um arquivo. Esses arquivos podem ser usados para backup ou para transferência de dados entre o PostgreSQL e outras aplicações. É possível usar as várias STDIN e STDOUT para especificar que os dados a serem transferidos sejam inseridos ou exibidos na linha de comando. Podemos ainda usar o modificar DELIMITERS para especificar o caractere que separa as colunas dentro de cada linha do arquivo. O padrão é um caractere de tabulação

para arquivos em formato de texto ou uma vírgula no formato CSV. Este delimitador deve ser um único caractere de um byte. Esta opção não é permitida quando usando o formato binário. Vejam que já achamos nossa resposta.

As demais alternativas apresentam termos que não estão no rol de palavras reservadas ou comandos do postgreSQL. Dentre os comandos utilizados para exportar arquivo, existe uma lista, além do COPY, responsável pela manipulação de arquivos LOB (large objects). Vejam a lista na figura abaixo:

```
COPY, LARGE OBJECT
  \copy ...                              perform SQL COPY with data stream to the client host
  \lo_export LOBOID FILE                 LOBOID FILE
  \lo_import FILE [COMMENT]              FILE [COMMENT]
  \lo_list
  \lo_unlink LOBOID                      large object operations
```

Gabarito: A.

20. BANCA: CESPE - Oficial Técnico de Inteligência/Área 9/2018

Julgue o próximo item, a respeito de conceitos e comandos PostgreSQL e MySQL.

No programa psql do PostgreSQL, a instrução \h permite mostrar o histórico de comandos SQL na sessão atual.

Certo

Errado

Comentário: O psql é um cliente no modo terminal do PostgreSQL. Permite digitar comandos interativamente, submetê-los para o PostgreSQL e ver os resultados. Como alternativa, a entrada pode vir de um arquivo. Além disso, disponibiliza vários meta-comandos e diversas funcionalidades semelhantes às do interpretador de comandos (shell), para facilitar a criação de scripts e automatizar muitas tarefas.

Ao digitar \help (ou \h) [comando], o sistema fornece ajuda de sintaxe para o comando SQL especificado. Se não for especificado o comando, então o psql listará todos os comandos para os quais existe ajuda de sintaxe disponível. Se o comando for um asterisco ("*"), então será mostrada a ajuda de sintaxe para todos os comandos SQL. Desta forma, a alternativa está incorreta.

Gabarito: E

21. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

A instrução SQL em PostgreSQL abaixo está mal formulada.

```
CREATE VIEW vista AS SELECT 'Hello World';
```

Isto aconteceu, porque

A a criação de uma visualização requer a utilização da cláusula WHERE para a restrição dos dados.

B não é possível criar uma VIEW sem a identificação do tipo de dado e sem a determinação da quantidade de registros selecionados.

C o comando CREATE VIEW deve utilizar a cláusula FROM para o nome da tabela.

D a criação de uma visualização (VIEW) requer a definição de um gatilho (trigger) correspondente ao nome da coluna.

E por padrão, o tipo de dado será considerado indefinido (unknown) e a coluna irá utilizar o nome padrão ?column?.

Comentário. Vamos analisar as alternativas. De cara podemos dizer que a resposta a essa questão está nas notas da documentação do comando CREATE VIEW. A alternativa A aponta uma restrição que não faz sentido. Basta pensarmos em uma visão criada sobre duas colunas de uma determinada tabela que retorna todas as linhas dessa tabela. Neste caso, não precisamos ter a cláusula WHERE.

A alternativa B também está incorrera, pois não precisamos criar uma VIEW determinando a quantidade de tuplas retornadas na seleção. Contudo, a primeira parte da alternativa está correta. Para que o SGBD consiga criar a VIEW, ele precisa que cada coluna ou atributo tenha um nome e um tipo de dado correspondente. Veja que isso acontece naturalmente quanto selecionamos os dados de uma tabela já existente. Porém, não é o que acontece quando optamos por utilizar uma constante, como acontece no enunciado da questão.

Na alternativa C, observamos que o erro está em dizer que é necessária a utilização da cláusula FROM. Não é verdade, pois é possível selecionar uma tabela apenas de valores constantes. Veremos um exemplo logo em seguida.

A criação de gatilhos ou triggers para que a VIEW seja atualizada depende da complexidade da VIEW. Algumas restrições são impostas pelo Postgres para que a VIEW seja considerada UPDATABLE. Desta forma, não podemos dizer que uma VIEW requer, necessariamente, um gatilho.

Por fim, a nossa resposta, a alternativa E. Veja que o comando da questão não faz referência a nenhuma tabela para que sejam extraídos o tipo e o nome dos atributos, sendo necessário dar-lhe um nome e um tipo. Veja abaixo como ficaria a sintaxe correta do comando:

CREATE VIEW vista AS SELECT text 'Hello World' AS hello;

Inclusive, o exemplo do problema apresentado no enunciado é o mesmo que temos na documentação. A resolução do problema é a mesma apresentada acima. Observe que agora temos o tipo text e o nome hello.

Gabarito: E.

22. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Considere o trecho em PostgreSQL abaixo.

INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99), (2,'Bread',1.99), (3,'Milk', 2.99);

Considerando a existência prévia da tabela products contendo as colunas product_no, name e price, e desconsiderando os tipos de dados, esse trecho irá resultar:

A na adição de 3 novas colunas na tabela products.

B na adição de 3 novas linhas na tabela products.

C em erro, pois não é possível múltiplas inserções em um único comando SQL.

D em erro, pois para se realizar múltiplas inserções é necessário a utilização da cláusula SELECT.

E em erro, pois múltiplas inserções são possíveis somente com a utilização de colchetes para a limitação dos registros.

Comentário. Para respondermos a essa questão, precisaríamos de mais detalhes sobre a tabela. Mas, considerando que ela só possui essas três colunas e que os tipos de dados não são relevantes, podemos avaliar que o comando INSERT criará linhas na tabela *products*. Desta forma, podemos marcar nossa resposta na alternativa B.

Gabarito: B.

---

23. BANCA: FCC - Analista Judiciário (TRT 23ª Região)/Apoio Especializado/Tecnologia da Informação/2016

São vários os tipos de dados numéricos no PostgreSQL. O tipo

a) smallint tem tamanho de armazenamento de 1 byte, que permite armazenar a faixa de valores inteiros de −128 a 127.

b) bigint é a escolha usual para números inteiros, pois oferece o melhor equilíbrio entre faixa de valores, tamanho de armazenamento e desempenho.

c) integer tem tamanho de armazenamento de 4 bytes e pode armazenar valores na faixa de −32768 a 32767.

d) numeric pode armazenar números com precisão variável de, no máximo, 100 dígitos.

e) serial é um tipo conveniente para definir colunas identificadoras únicas, semelhante à propriedade auto incremento.

Comentário: A questão trata dos tipos de dados numéricos do PostgreSQL. Observem abaixo uma tabela que vimos com os tipos numéricos presentes no PostgreSQL:

Name	Storage Size	Range
smallint	2 bytes	-32768 to +32767
integer	4 bytes	-2147483648 to +2147483647
bigint	8 bytes	-9223372036854775808 to +9223372036854775807
decimal	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	6 decimal digits precision
double precision	8 bytes	15 decimal digits precision
smallserial	2 bytes	1 to 32767
serial	4 bytes	1 to 2147483647
bigserial	8 bytes	1 to 9223372036854775807

Vejam que os tipos ***decimal*** e ***numeric*** têm o tamanho de armazenamento variável. Já os tipos smallserial, serial e bigserial são utilizados para valores auto incrementados, cujos domínios só permitem valores positivos.

Agora vamos aos erros das alternativas. Na letra A, diz que smallint tem 1 byte, ao invés de 2. Na alternativa B, fala que o bigint é uma alternativa intermediária, quando na realidade ele é o valor numérico que possui o maior range. A assertiva C afirma erroneamente que integer possui apenas 2 bytes, quando na realidade apresenta 4. Por fim, a letra E diz que decimal tem precisão de 100 dígitos, quando na realidade podemos ter até 16383 dígitos depois da casa decimal.

E a alternativa E? Essa é a nossa resposta, apresenta o tipo serial, que é utilizado para definir colunas identificadoras únicas, semelhante à propriedade auto incremento.

Gabarito: E

24. BANCA: FCC - Técnico Judiciário (TRE PB)/Apoio Especializado/Programação de Sistemas/2015

No PostgreSQL, o tipo de dados numérico considerado meramente uma notação conveniente para definir colunas identificadoras únicas, semelhante à propriedade auto incremento em alguns Sistemas Gerenciadores de Banco de Dados, é o tipo

a) serial.

b) smallint.

c) byte.

d) bit.

e) blob.

Comentário: O gabarito da questão é a letra a). Novamente a banca FCC cobrou uma questão sobre o tipo numérico serial. Conforme vimos na questão anterior, o tipo serial é utilizado para definir colunas identificadoras únicas, semelhante à propriedade auto incremento.

Gabarito: A

25. BANCA: FCC ANO: 2014 ÓRGÃO: TRT - 1ª REGIÃO (RJ) PROVA: TÉCNICO DO JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

No sistema gerenciador de Banco de Dados PostgreSQL (v. 9.1), a forma para se declarar um atributo com o tipo de dados Array, com duas dimensões, tendo o nome teste é

A . . . teste [[ ]] . . .

B . . . teste 2 [ ] . . .

C . . . teste [2] . . .

D . . . teste [ ] [ ] . . .

E . . . teste [ ] x [ ] . . .

Comentário. Pelo que conhecemos da sintaxe do comando utilizado para criação de atributos de tabela como arrays, podemos marcar como resposta a alternativa D. Vamos agora conhecer um pouco sobre os BLOBs ou Binary Large Objects.

Gabarito: D.

26. BANCA: FCC ANO: 2012 ÓRGÃO: TCE-AM PROVA: ANALISTA TÉCNICO DE CONTROLE EXTERNO - TECNOLOGIA DA INFORMAÇÃO

Em PostgreSQL, a função que converte a primeira letra da string informada em letra maiúscula, alterando todas as letras subsequentes dessa string para minúsculas se chama

A chgstr.

B altertext.

C initcap.

D upper.

E toupper.

Comentário. As alternativas A, B e E não correspondem a funções incluídas na lista de funções do Postgres para operações com *strings*. As alternativas C e D são definidas da seguinte forma:

initcap (string) - Converte a primeira letra de cada palavra para maiúscula e o restante para letras minúsculas. As palavras são sequências de caracteres alfanuméricos separados por caracteres não alfanuméricos.

upper (string) – Converte toda a string para letras maiúsculas.

Analisando a descrição de cada uma das funções, podemos confirmar nossa resposta na alternativa C. Vamos para a próxima questão.

Gabarito: C.

27. BANCA: FCC ANO: 2012 ORGÃO: MPE-PE - Analista Ministerial PROVA: Informática

No banco de dados *PostgreSQL*, a função COALESCE

A retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é muito semelhante ao LIKE, exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL.

B é uma expressão condicional genérica, semelhante às declarações *if/else* de outras linguagens.

C é uma declaração SELECT arbitrária, ou uma subconsulta. A subconsulta é processada para determinar se retorna alguma linha.

D retorna o primeiro de seus argumentos que não for nulo. Só retorna nulo quando todos os seus argumentos são nulos.

E permite a conversão do carimbo do tempo (*time stamp*) para uma zona horária diferente.

Comentário. Sobre a função solicitada na questão, ela retorna o primeiro argumento que não for nulo. Null é retornado somente se todos os argumentos forem nulos. Ela é frequentemente usada para substituir um valor padrão por valores nulos quando os dados são recuperados para exibição. Ela está incluída no rol de funções ou expressões condicionais presentes no Postgres. Essa lista é completada pelas expressões CASE, NULLIF, GREATEST e LEAST. Vejamos abaixo a definição de cada uma delas.

A expressão CASE do SQL é uma expressão condicional genérica, semelhante às declarações if/else em outras linguagens de programação. Vejam a sintaxe do comando abaixo:

```
CASE WHEN condition THEN result
     [WHEN ...]
     [ELSE result]
END
```

A função NULLIF retorna um valor nulo se value1 é igual a value2; caso contrário, retorna o valor1. Para entender melhor, observe a sintaxe e um exemplo a seguir:

```
NULLIF(value1, value2)
```

```
SELECT NULLIF(value, '(none)') ...
```

Por fim, tempos as funções GREATEST e LEAST, que selecionam o valor maior ou menor de uma lista de expressões. Temos a resposta para a questão na alternativa D.

Gabarito: D

28. BANCA: CESPE - Analista Judiciário (STM)/Apoio Especializado/Análise de Sistemas/2018

Julgue o item subsequente, a respeito do Postgres 9.6.

Ao se criar uma trigger, a variável especial TG_OP permite identificar que operação está sendo executada, por exemplo, DELETE, UPDATE, INSERT ou TRUNCATE.

Certo

Errado

Comentário: A questão está correta. De fato, TG_OP é uma variável do tipo texto que pode ser preenchida com os valores INSERT, UPDATE, DELETE ou TRUNCATE. Eles identificam quais operações irão disparar o TRIGGER. Vejamos alguns exemplos:

```
CREATE TRIGGER emp_stamp BEFORE INSERT OR UPDATE ON emp
    FOR EACH ROW EXECUTE PROCEDURE emp_stamp();
CREATE TRIGGER emp_audit
AFTER INSERT OR UPDATE OR DELETE ON emp
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```

Gabarito: C

29. Ano: 2018 Banca: CESPE Órgão: STM Cargo: Analista de Sistemas Questão: 71 e 72

Um sistema gerenciador de banco de dados (SGBD) instalado no Linux deve ser configurado de modo a permitir os seguintes requisitos:

I no máximo, 1000 conexões simultâneas;

II somente conexões originadas a partir do servidor de

aplicação com IP 10.10.10.2.

Tendo como referência essas informações, julgue os seguintes itens.

71 Caso o SGBD instalado seja o Postgres 9.6, para atendimento do requisito I, deve-se modificar o arquivo postgres.conf para o referido cluster; alterar o parâmetro max_connections para 1000; e reiniciar o serviço do SGBD.

72 Caso o SGBD instalado seja o Postgres 9.6, para atendimento do requisito II, deve-se modificar o arquivo pg_hba.conf para o referido cluster; alterar o parâmetro listen_addresses para o IP fornecido; e reiniciar o serviço do SGBD.

Comentário: Vamos comentar cada uma das alternativas acima:

71. O parâmetro *max_connections* determina o número máximo de conexões simultâneas para o servidor de banco de dados. O padrão geralmente é 100 conexões, mas pode ser menor se as configurações do kernel não o suportarem. Este parâmetro só pode ser

configurado no início do servidor. Ou seja, para alterar esse valor, precisamos reiniciar o SGBD. Contudo, a afirmação apresenta **o nome do arquivo de configuração incorreto**. O certo seria ***postgresql.conf***. Sendo assim, a alternativa está incorreta! (A banca inicialmente tinha dado o gabarito como correto e, em seguida anulou a questão por conta deste equívoco.)

72. O arquivo pg_hba.conf indicará ao PostgreSQL como autenticar usuários que fazem acesso ao banco de dados. Em geral, as entradas do arquivo pg_hba.conf têm o seguinte layout:

```
# local      DATABASE  USER  METHOD  [OPTIONS]
# host       DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostssl    DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
# hostnossl  DATABASE  USER  ADDRESS  METHOD  [OPTIONS]
```

Vejamos alguns exemplos de regras que podem estar registradas em um arquivo pg_hba.conf:

```
# TYPE   DATABASE     USER        ADDRESS            METHOD
 local        all          all                          trust
 host         all          all        127.0.0.1/32          trust
 host         all          all          ::1/128             trust
```

O termo local diz que todos os usuários usando sockets Unix locais devem ser confiáveis para todos os bancos de dados. O método *trust* significa que nenhuma senha deve ser enviada para o servidor e as pessoas podem fazer *login* diretamente. As outras duas regras dizem que o mesmo se aplica às conexões do localhost 127.0.0.1 e ::1/128, que é um endereço IPv6.

Perceba que até agora **não** falamos do parâmetro *listen_addresses*. É porque ele não está no arquivo ***pg_hba.conf***. Ele é configurado no arquivo **postgresql.conf**. Desta forma, a alternativa está **incorreta**.

Gabarito: E E

30. Ano: 2018 Banca: CESPE Órgão: STM Cargo: Analista de Sistemas Questão: 82

Julgue os itens subsequentes, a respeito do Postgres 9.6.

82 Nas instruções seguintes, a palavra-chave IMMUTABLE indica que a função criada não pode modificar o banco de dados. CREATE FUNCTION add(integer, integer) RETURNS integer

```
AS 'select $1 + $2;'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;
```

Comentário: A questão trata dos parâmetros IMMUTABLE, STABLE, VOLATILE. Esses atributos informam ao otimizador de consulta sobre o comportamento da função. No máximo, uma opção pode ser especificada. Se nenhum deles aparecer, VOLATILE é definido por padrão.

Ser o valor passado for **IMMUTABLE**, vai indicar que **a função não pode modificar o banco de dados e sempre retorna o mesmo resultado quando dados os mesmos valores de argumento**; ou seja, não faz pesquisas de banco de dados ou, de outra forma, usa informações que não estão diretamente presentes na lista de argumentos. Se esta opção for dada, qualquer chamada da função com argumentos constantes pode ser imediatamente substituída pelo valor da função. Desta forma, temos mais uma alternativa **correta**.

STABLE indica que a função não pode modificar o banco de dados, e que, dentro de uma única tabela, retornará consistentemente o mesmo resultado para os mesmos valores de argumento. Mas seu resultado pode mudar em declarações SQL. Esta é a seleção apropriada para funções cujos resultados dependem de pesquisas de banco de dados, variáveis de parâmetros (como o fuso horário atual) etc.

VOLATILE indica que o valor da função pode mudar mesmo dentro de uma varredura de tabela única. Portanto, nenhuma otimização pode ser feita. Relativamente poucas funções de banco de dados são voláteis nesse sentido; alguns exemplos são aleatórios (), currval (), timeofday (). Mas note que qualquer função que tenha efeitos colaterais deve ser classificada como volátil, mesmo que seu resultado seja bastante previsível, para evitar que as chamadas sejam otimizadas; um exemplo é setval ().

Gabarito: C

31. Ano: 2018 Banca: CESPE Órgão: STM Cargo: Programação de Sistemas Questão: 66 a 70

Julgue os próximos itens, que dizem respeito aos SGBDs Oracle, MySQL e PostgreSQL.

69 Uma desvantagem do PostgreSQL em relação aos demais SGBDs é que ele não oferece recursos necessários para se realizar a replicação de dados.

Comentário: A partir da versão 9.0 do PostgreSQL, foi adicionado um recurso de Streaming Replication (SR) nativamente ao PostgreSQL, permitindo a replicação de transações, assim que concluídas sem necessidade de aguardar que um segmento seja completado. A replicação por meio de streaming funciona através de uma conexão comum. Sendo assim, observamos que o SGBD em questão de fato implementa mecanismos de replicação de dados. Logo, a alternativa está **errada**.

Gabarito: E

32. Ano: 2018 Banca: CESPE Órgão: STM Cargo: Analista de Sistemas Questões: 86 e 88

Julgue os itens que se seguem, a respeito do processamento de transações e otimização de desempenho do SGBD e de consultas SQL.

86 O controle de nível de isolamento de transações é importante para gerenciar a forma como as transações concorrentes se comportarão no SGBD. No Postgres 9.6, o nível de isolamento padrão é READ COMMITTED, mas pode ser alterado para SERIAZABLE por meio do comando SET TRANSACTION ISOLATION LEVEL SERIAZABLE.

88 No Oracle 12C, a Automatic Workload Repository (AWR) é uma funcionalidade similar ao autovacuum no Postgres 9.6, haja vista que ambos processam e mantêm estatísticas de desempenho para detecção de problemas e manutenção automática do banco de dados, por exemplo, reusando, ajustando e excluindo dados temporários e reusando espaço em blocos por linhas excluídas.

Comentário: Vamos comentar cada uma das alternativas acima:

86. Por padrão, o PostgreSQL é executado no modo de isolamento de transação READ COMMITTED. Isso significa que cada declaração dentro de uma transação obterá um novo instantâneo dos dados, que será constante ao longo da consulta. Para mudar de nível de isolamento, usamos o comando SET TRANSACTION, que possui a seguinte sintaxe.

SET TRANSACTION ***transaction_mode*** [, ...]

SET TRANSACTION SNAPSHOT ***snapshot_id***

SET SESSION CHARACTERISTICS AS TRANSACTION ***transaction_mode*** [, ...]

onde ***transaction_mode*** pode ser definido da seguinte forma:

ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }

READ WRITE | READ ONLY

[NOT] DEFERRABLE

Logo, a afirmação **estaria correta**. Contudo, observem que o enunciado apresenta o termo SERIAZABLE, quando o correto seria SERIALIZABLE. Sendo assim a banca optou por anular a questão!

88. O PostgreSQL e outros bancos de dados relacionais usam uma técnica chamada **Multi-Version Concurrency Control** (MVCC) para manter o controle das transações. Uma penalidade de espaço surge quando usamos o MVCC, ela é conhecida como inchaço. O PostgreSQL precisa de ajuda de uma ferramenta externa chamada **VACUUM** para poder limpar essa “sujeira”.

As tabelas e os índices inchados não somente desperdiçam espaço, como também deixam as consultas mais lentas. Então, isso não é só uma questão de conseguir mais espaço no disco rígido. Antigamente, os DBAs precisavam executar o VACUUM manualmente. Hoje, é possível configurar um deamon chamado Autovacuum para executar essas limpezas em momentos oportunos.

Veja que Autovacuum não guarda nenhuma relação de similaridade funcional com o AWR. Podemos afirmar, portanto, que a alternativa está **incorreta**. Sabemos que o AWR significa

Automatic Workload Repository, ou seja, é um repositório de informações a respeito da carga de trabalho do banco de dados. O framework do AWR coleta, processa e mantém estatísticas de desempenho para possibilitar detecção de problemas e é a base para as tarefas de tuning automáticas do Oracle. Estas estatísticas são coletadas através de snapshots regulares e armazenadas no AWR por um período definido. Elas são baseadas no momento do snapshot e podem ser utilizadas para elaborar um relatório. Os valores capturados pelo snapshot representam as mudanças em cada estatística coletada no período.

Gabarito: Anulada E

33. BANCA: CESPE - Oficial Técnico de Inteligência/Área 8/2018

A respeito de sistemas gerenciadores de banco de dados, julgue o próximo item.

No arquivo pg_hba.conf de configuração do PostgreSQL, as diretivas são avaliadas a partir da linha superior, para a linha inferior.

Comentário: O *pg_hba.conf* controla quais máquinas terão acesso ao PostgreSQL e a autenticação dessas máquinas clientes (sem autenticação ou através de outras formas: trust, md5, crypt, ...). Com o pg_hba.conf, podemos controlar o acesso pelo IP, pela máscara, pelo banco, pelo usuário ou pelo método (trust, md5, password, etc).

No arquivo, temos uma lista organizada de registros que vão fazer a configuração da autenticação dos clientes do PostgreSQL. A avaliação destes registros é feita em ordem. Contudo, se houver algum conflito, o último registro avaliado é descartado, ficando apenas o primeiro. Vejamos dois exemplos:

Exemplo 01:

TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
local	all	all		md5
local	all	all		trust

Acima todas as conexões locais exigirão senha **md5**.

Exemplo 02:

TYPE	DATABASE	USER	CIDR-ADDRESS	METHOD
local	all	all		trust
local	all	all		md5

Já este acima aceitará todas as conexões locais incondicionalmente (sem senha, trust).

Logo, temos uma alternativa **correta**.

Gabarito: C

34. BANCA: FCC - Técnico Judiciário (TRT 2ª Região)/Apoio Especializado/Tecnologia da Informação/2018

Considere que um Técnico de TI deseja criar as tabelas abaixo em um banco de dados PostgreSQL 8 aberto e em condições ideais.

```
CREATE TABLE departamento (
        codDep     varchar(10) primary key,
        local point
);

CREATE TABLE funcionario (
        codDep      varchar(10) .....I.....,
        salario        real,
        dataAdm    date
);
```

Para que codDep na tabela funcionario seja definido como chave estrangeira com relação à tabela departamento, a lacuna I deve ser preenchida com

a) foreign key departamento(codDep)

b) constraint foreign key REF departamento(codDep)

c) foreign key IN departamento(codDep)

d) constraint departamento(codDep)

e) references departamento(codDep)

Comentário: Veja a sintaxe do PostgreSQL para criação de chave estrangeira:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products (product_no),
    quantity integer
);
```

Você também pode encurtar o comando acima para:

```
CREATE TABLE orders (
    order_id integer PRIMARY KEY,
    product_no integer REFERENCES products,
    quantity integer
);
```

Temos, então, a resposta na letra e).

Gabarito: E

---

35. BANCA: CESPE - Analista Judiciário (TRT 7ª Região)/Apoio Especializado/Tecnologia da Informação/2017

No sistema gerenciador de banco de dados PostgreSQL, a restrição de acesso pelo endereço IP do cliente é feita mediante alteração do arquivo de configuração

a) pg_subtrans.

b) pg_hba.conf.

c) postmaster.opts.

d) pg_ctl.

Comentário: Como vimos na aula, o formato geral do arquivo pg_hba.conf é um conjunto de registros, um por linha. Cada registro especifica um tipo de conexão, **uma faixa de endereço IP de cliente** (se for relevante para o tipo de conexão), um nome de banco de dados, um nome de usuário, bem como o método de autenticação a ser usado para conexões que utilizam estes parâmetros. O primeiro registro com os seguintes dados: tipo de correspondência de conexão, endereço do cliente, banco de dados solicitado e nome de usuário é usado para executar a autenticação. Abaixo temos um exemplo do arquivo:

```
# TYPE  DATABASE        USER            ADDRESS                 METHOD
host    all             all             192.168.54.1/32         reject
host    all             all             0.0.0.0/0               gss
```

Gabarito: B

---

36. BANCA: FCC - Técnico de Nível Superior (PGM Teresina)/Analista de Sistemas/2016

Uma das recomendações na prática do tuning no PostgreSQL como forma de melhorar o desempenho das tabelas com grandes quantidades de registros e especialmente com muitos acessos é a inserção de

a) índices em todos os campos das tabelas.

b) valores nulos em campos que compõem a cláusula WHERE ou que fazem parte de cláusulas ORDER BY, GROUP BY.

c) valores nulos em campos que compõem o statement SELECT sem cláusula WHERE.

d) índices em campos que compõem a cláusula WHERE ou que fazem parte de cláusulas ORDER BY, GROUP BY.

e) índices em todos os campos que compõem um statement SELECT DISTINCT e/ou cláusula CONSTRAINT.

Comentário: Como vimos na aula, um campo de índice pode ser uma expressão calculada a partir dos valores de uma ou mais colunas da linha da tabela. Este recurso pode ser usado para obter **acesso rápido aos dados** baseado em alguma transformação dos dados básicos.

Quando a cláusula WHERE está presente, um índice parcial é criado. Um índice parcial é um índice que contém entradas para apenas uma parte de uma tabela, geralmente

uma porção mais útil para a indexação do que o resto do quadro. Da mesma forma, os índices também são recomendados para utilização nas cláusulas ORDER BY e GROUP BY.

Temos o gabarito na letra d).

Gabarito: D

---

37. BANCA: FCC - Analista de Sistemas (DPE RR)/2015

Um Analista de Sistemas deseja fazer uma cópia de segurança consistente de um banco de dados PostgreSQL, mesmo que ele esteja sendo utilizado por outros usuários, gerando um arquivo texto contendo comandos SQL. Estes comandos, ao serem processados pelo servidor, recriam o banco de dados no mesmo estado em que este se encontrava quando o arquivo foi gerado. O Analista deve usar o utilitário

a) pg_dump.

b) psql.

c) pgbackup.

d) gunzip.

e) sql_backup.

Comentário: Temos a resposta na letra a). Vimos na aula que o pg_dump é um utilitário para fazer backup de um banco de dados PostgreSQL. Ele faz backups consistentes, mesmo se o banco de dados está sendo usado. O pg_dump não bloqueia os outros usuários que acessam o banco de dados (leitura ou gravação).

Ele pode ter sua saída em formato de scripts ou arquivos. Dumps de script são arquivos de texto simples que contêm os comandos SQL necessários para reconstruir o banco de dados para o estado em que estava no momento em que foi salvo. Para restaurar a partir de um script, carregue o mesmo no psql. Os arquivos de script podem ser usados para reconstruir o banco de dados até mesmo em outras máquinas com outras arquiteturas; com algumas modificações, até em outros bancos de dados SQL.

Gabarito: A

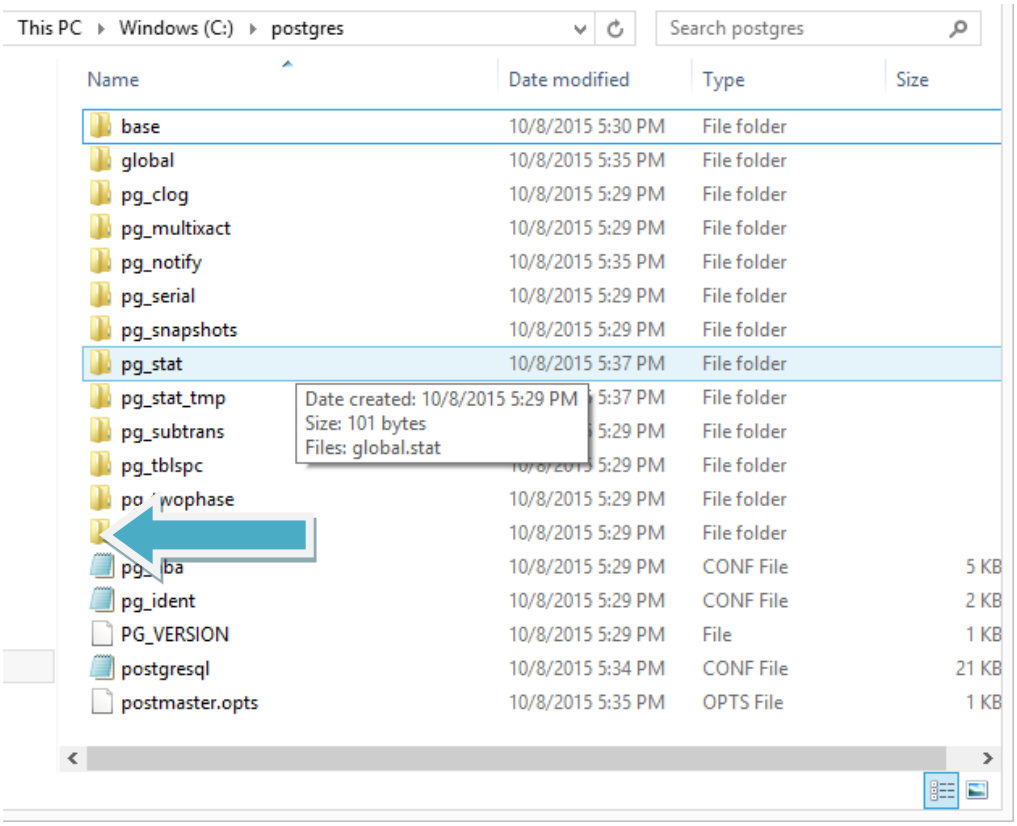
---

38. BANCA: CESPE ANO: 2015 ÓRGÃO: CGE-PI PROVA: AUDITOR GOVERNAMENTAL - TECNOLOGIA DA INFORMAÇÃO

Acerca de bancos de dados, julgue os itens a seguir.

No PostgreSQL 9.3, os arquivos de WAL (write-ahead logging), que armazenam as transações do SGBD na forma de segmentos de log, são gravados por padrão no diretório pg_wal abaixo do diretório data.

Comentário. Essa questão é um pouco maldosa, força o candidato a conhecer detalhes da estrutura de pastas de um cluster de banco de dados do PostgreSQL. Veja a figura abaixo com uma lista de pastas criadas pela execução do comando initdb.



Observem que existe uma seta que aponta para a pasta pg_xlog. Essa é a pasta responsável por armazenar os logs do WAL. Mas o que é WAL?

Write-ahead logging (WAL) é um método padrão para garantir a integridade dos dados. Resumidamente, o conceito central do WAL é que as alterações nos arquivos de dados (onde tabelas e índices residem) devem ser escritas somente após essas alterações serem registradas, ou seja, após os registros que descrevem as alterações serem liberados para armazenamento permanente.

Se seguirmos esse procedimento, nós não precisamos dar um flush das páginas de dados no disco a cada commit de transações, porque sabemos que, no caso de um acidente, seremos capazes de recuperar o banco de dados usando o log. Quaisquer alterações que não foram aplicadas nas páginas de dados podem ser refeitas a partir dos registros de log. (Esta recuperação denomina-se roll-forward, também conhecido como REDO.)

Gabarito: E.

---

39. Ano: 2018 Banca: FCC Órgão: DPE-AM Cargo: Analista Área: Banco de Sistemas Questão: 46

Considere as instruções SQL a seguir, digitadas no PostgreSQL 9.0, em condições ideais.

CREATE TABLE processo (proc_num character(24));

INSERT INTO processo SELECT '0000125–40.' || '1981.403.6100';

SELECT proc_num, octet_length(proc_num) FROM processo;

Será exibido na tela

(A) 0000125–40.1981.403.6100 e 24.

(B) uma mensagem de erro, pois para concatenar valores, no lugar de || deve ser usado &&.

(C) 1981.403.6100 e 24.

(D) uma mensagem de erro, já que o comando octet_length não existe.

(E) uma mensagem de buffer overflow.

Comentário: Essa questão me pareceu interessante, mas **acho** que não está 100%. Vamos olhar para os 3 comandos SQL descritos no enunciado. O primeiro comando é utilizado para a criação de uma tabela com apenas uma coluna do tipo *character* com tamanho 24. O segundo comando insere dentro da tabela uma tupla, cujo valor é a concatenação (||) das strings '0000125–40.' e '1981.403.6100'. Desta forma, acho que é facilmente percebido que o primeiro valor retornado pelo comando SELECT será exatamente 0000125–40.1981.403.6100.

Agora vamos olhar para a próxima coluna que será retornada. Neste caso, temos como valor o resultado da função *octet_length(string)*, que retorna a quantidade de bytes existentes na *string*. Observe que o tipo associado ao atributo é *character*. Neste caso, mesmo que o parâmetro inserido tivesse menos que 24 caracteres, o SGBD completaria com espaços em branco. Logo, a resposta para a segunda coluna do SELECT é 24.

Ficamos, portanto, com os valores 0000125–40.1981.403.6100 e 24, que estão presentes na alternativa A.

Gabarito: A

---

40. Ano: 2018 Banca: FCC Órgão: TRT-06 Cargo: Analista Judiciário Área: Tecnologia da Informação Questão: 34

Em um banco de dados PostgreSQL aberto e em condições ideais, um Analista especializado em Tecnologia da Informação executou as instruções abaixo em uma tabela chamada funcionario.

BEGIN;

UPDATE funcionario SET salario = salario - 1000.00

WHERE nome = 'João';

SAVEPOINT ps1;

UPDATE funcionario SET salario = salario + 1000.00

WHERE nome = 'Paulo';

I

UPDATE salario SET salario = salario + 1000.00

WHERE nome = 'Marcos';

COMMIT;

Na segunda instrução UPDATE, o Analista aumentou o salário do funcionário Paulo em 1000.00, quando deveria aumentar o salário do funcionário Marcos nesse valor. Para cancelar a operação realizada, a lacuna I deve ser preenchida pela instrução

(A) ROLLBACK -1;

(B) ROLLBACK TO ps1;

(C) CANCEL OPERATION;

(D) RESTORE TO ps1;

(E) CANCEL UPDATE;

Comentário: O comando ROLLBACK TO SAVEPOINT reverte todos os comandos que foram executados depois que o ponto de salvamento especificado no parâmetro foi estabelecido. O ponto de salvamento permanece válido e pode ser revertido novamente mais tarde, se necessário.

Devemos perceber ainda que o ROLLBACK TO SAVEPOINT destrói, implicitamente, todos os savepoints que foram estabelecidos após o savepoint nomeado.

Desta forma, podemos marcar nossa resposta na alternativa B.

Gabarito: B

41. Ano: 2017 Banca: FCC Órgão: TRT-11 Cargo: Técnico Judiciário de TI – Q. 46

Um comando SQL, cuja sintaxe é válida no PostgreSQL 9.3, está apresentado em:

(A) SELECT nome1, nome2, nome 3 FROM tab2TRT ORDER FOR nome1 AFTER nome2;

(B) DELETE FROM tabTRT WHERE data processo= “13-AGO-2013”;

(C) UPDATE trabTRT SETTING mes = 10 INSTEAD mes = 5;

(D) CREATE TABLE tab2 TRT { id integer, nome texts, salario numerical};

(E) INSERT INTO tabTRT DEFAULT VALUES;

Comentário: Vamos analisar a sintaxe do comando INSERT definida pelo PostgreSQL.

```
INSERT INTO tabela [ ( coluna [, ...] ) ]
    { DEFAULT VALUES | VALUES ( { expressão | DEFAULT } [, ...] ) [, ...] | consulta }
    [ RETURNING * | expressão_de_saída [ AS nome_de_saída ] [, ...] ]
```

O comando INSERT insere novas linhas na tabela. Podem ser inseridas uma ou mais linhas especificadas por expressões de valor, ou zero ou mais linhas resultantes de uma consulta.

Os nomes das colunas de destino podem ser listados em qualquer ordem. Se não for fornecida nenhuma lista de nomes de colunas, o padrão é usar todas as colunas da tabela na ordem em que foram declaradas; ou os primeiros N nomes de colunas, se existirem apenas N colunas fornecidas na cláusula VALUES ou na consulta. Os valores fornecidos

pela cláusula VALUES ou pela consulta são associados à lista de colunas explícita ou implícita da esquerda para a direita.

As colunas que **não estão presentes na lista de colunas explícita ou implícita são preenchidas com o valor padrão**, seja o valor padrão declarado ou nulo, se não houver nenhum.

A cláusula opcional **RETURNING** faz com que o comando INSERT compute e retorne valores baseados em cada linha realmente inserida. Sua utilidade principal é obter valores fornecidos por padrão, como o número sequencial de uma coluna serial.

Ao utilizar o **DEFAULT VALUES** todas as colunas são preenchidas com seu valor padrão. Percebam que foi exatamente isso que a questão apresentou na alternativa E.

Gabarito: E

42. BANCA: FCC ANO: 2017 ÓRGÃO: TST PROVA: ANALISTA JUDICIÁRIO – SUPORTE EM TECNOLOGIA DA INFORMAÇÃO

[70] Um Analista de Suporte que utiliza o PostgreSQL possui uma tabela chamada employee, com os campos id, name e salary. Deseja executar uma consulta que exiba todos os nomes e salários dos funcionários, de forma que, se o salário for nulo, exiba o valor 0 (zero). Para realizar a consulta terá que utilizar a instrução SELECT name,

a) NVDL(salary, 0) FROM employee;

b) IFNL(salary, 0) FROM employee;

c) COALESCE(salary, 0) FROM employee;

d) IFNULL(salary; '0') FROM employee;

e) NVL(salary; 0) FROM employee;

Comentário: A função COALESCE retorna o primeiro de seus argumentos que não é nulo. Null é retornado somente se todos os argumentos forem nulos. É frequentemente usado para substituir um valor padrão por valores nulos quando os dados são recuperados para exibição, por exemplo:

SELECT COALESCE(description, short_description, '(none)') ...

Outro comando que temos no PostgreSQL é o NULLIF(value1, value2). A função NULLIF retorna um valor nulo, se value1 for igual a value2; caso contrário, ele retorna value1. Isso pode ser usado para executar a operação inversa do exemplo COALESCE dado acima:

SELECIONE NULLIF (valor, '(nenhum)') ...

Por fim, a outra expressão condicional que temos no postgreSQL é o CASE. A expressão SQL CASE é uma expressão condicional genérica, semelhante a instruções if/else em outras linguagens de programação:

CASE WHEN condição THEN resultado

[WHEN ...]

[Resultado ELSE]

END.

Desta forma, podemos marcar nossa resposta na alternativa C.

Gabarito: C

43. BANCA: FCC ANO: 2017 ÓRGÃO: DPE-RS PROVA: ANALISTA – BANCO DE DADOS

[49] No sistema gerenciador de banco de dados PostgreSQL 8 (versão 8.3) há duas bases de dados padrão utilizadas quando se cria uma nova base de dados. Se o objetivo for utilizar a base de dados padrão denominada template0, o comando a ser empregado na criação da base de dados teste, é CREATE DATABASE

a) USEDB template0

b) STANDARDDB template0

c) WITH template0

d) HAVING template0

e) teste TEMPLATE template0

Comentário: Vejamos a sintaxe de criação do banco de dados no PostgreSQL:

CREATE DATABASE nome

[[WITH]

[OWNER [=] dono_do_banco_de_dados]

[TEMPLATE [=] modelo]

[ENCODING [=] codificação]

[TABLESPACE [=] espaço_de_tabelas]

[CONNECTION LIMIT [=] limite_de_conexões]]

Em particular, se o template não for especificado, será usado o template1, que é o padrão. Desta forma, podemos encontrar nossa resposta na alternativa E. Veja que o sinal de igual entre TEMPLATE e template0 é opcional.

Gabarito: E

44. BANCA: FCC ANO: 2017 ÓRGÃO: DPE-RS PROVA: ANALISTA – BANCO DE DADOS

[53] Alguns sistemas gerenciadores de bancos de dados, como o PostgreSQL versão 8 permitem a definição de parâmetros de armazenamento de tabelas e de índices. Um desses parâmetros é o FILLFACTOR, sendo que no caso de

a) índices B-Tree, seu valor padrão é 50.

b) tabelas, pode variar entre os valores 50 e 100.

c) índices pouco atualizados, o melhor valor é 10.

d) tabelas muito atualizadas seu valor mais indicado é 100.

e) tabelas, seu valor padrão é 70.

Comentário: O FILLFACTOR de uma tabela é uma porcentagem entre 10 e 100. 100 é o padrão. Quando um fator de preenchimento menor é especificado, as páginas de tabela somente são preenchidas até a porcentagem indicada. O espaço restante em cada página é reservado para atualização das linhas nessa página. Isso dá ao UPDATE a chance de colocar a cópia atualizada de uma linha na mesma página que a original, o que é mais eficiente do que colocá-la em uma página diferente. Para uma tabela cujas entradas nunca são atualizadas, definir o valor 100 (completa) é a melhor opção. Mas em tabelas fortemente atualizadas, FILLFACTORS menores são apropriados. Veja que a alternativa B fala que os valores podem variar entre 50 e 100, que é um intervalo contido entre 10 e 100. Logo essa é a nossa resposta.

Gabarito: B

---

45. BANCA: FCC ANO: 2016 ÓRGÃO: TRT - 14ª REGIÃO (RO E AC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

[34] Para conceder, a todos os usuários de um banco de dados PostgreSQL, o privilégio de inserção na tabela Clientes utiliza-se a instrução

a) SET GRANT INSERT ON Clientes ALL;

b) SET ROLE ALL PRIVILEGES TO INSERT ON Clientes;

c) SET GRANT ALL TO Clientes ON INSERT;

d) GRANT INSERT ON Clientes TO ALL GROUP;

e) GRANT INSERT ON Clientes TO PUBLIC;

Comentário: O comando **GRANT** tem duas variantes básicas: uma que concede **privilégios a um objeto de banco de dados** (tabela, coluna, visão, sequence, banco de dados, *wrapper* de dados estrangeiros, servidor estrangeiro, função, linguagem procedural, esquema ou espaço de tabela) e aquele que concede **participação em um papel**. Vejamos alguns privilégios que podem ser concedidos pelo PostgreSQL.

Primeiramente, as permissões sobre tabelas. Perceba que o primeiro comando concede privilégios sobre a tabela inteira. Já o segundo possibilita especificar colunas.

GRANT { { SELECT | INSERT | UPDATE | DELETE | TRUNCATE | REFERENCES | TRIGGER }

[, ...] | ALL [PRIVILEGES] }

ON { [TABLE] ***table_name*** [, ...]

| ALL TABLES IN SCHEMA ***schema_name*** [, ...] }

TO { [GROUP] ***role_name*** | PUBLIC } [, ...] [WITH GRANT OPTION]

GRANT { { SELECT | INSERT | UPDATE | REFERENCES } (***column_name*** [, ...])
[, ...] | ALL [PRIVILEGES] (***column_name*** [, ...]) }
ON [TABLE] ***table_name*** [, ...]
TO { [GROUP] ***role_name*** | PUBLIC } [, ...] [WITH GRANT OPTION]

Agora vejamos as permissões sobre sequências:

GRANT { { USAGE | SELECT | UPDATE }
[, ...] | ALL [PRIVILEGES] }
ON { SEQUENCE ***sequence_name*** [, ...]
| ALL SEQUENCES IN SCHEMA ***schema_name*** [, ...] }
TO { [GROUP] ***role_name*** | PUBLIC } [, ...] [WITH GRANT OPTION]

Agora vejamos as permissões sobre os bancos de dados:

GRANT { { CREATE | CONNECT | TEMPORARY | TEMP } [, ...] | ALL [PRIVILEGES] }
ON DATABASE ***database_name*** [, ...]
TO { [GROUP] ***role_name*** | PUBLIC } [, ...] [WITH GRANT OPTION]

Finalmente, podemos definir nossa resposta na alternativa E.

Gabarito: E

46. BANCA: FCC ANO: 2016 ÓRGÃO: TRT - 23ª REGIÃO (MT) PROVA: TÉCNICO JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

[33] Um Técnico digitou as instruções abaixo.

No PostgreSQL:

SELECT TRIM (0 FROM 0004872348400) AS "Exemplo";

No Oracle:

SELECT TRIM (0 FROM 0004872348400) "Exemplo" FROM sys.dual;

O valor exibido foi

a) 4872348400

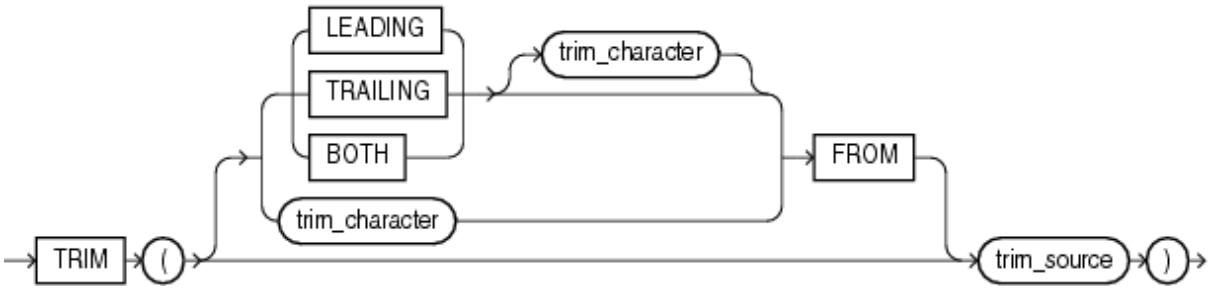
b) 0048432784000

c) 48432784

d) 48723484

e) 00048723484

Comentário: A função **TRIM** é utilizada para remover caracteres especificados. No caso da questão acima, o técnico especificou o numeral '0', e esse número foi removido de ambos os lados da variável. Vejam a definição da função, de acordo com a Oracle:



O TRIM permite que você recorte caracteres à esquerda (*leading*) ou à direita (*trailing*), ou ambos (*both*) de uma sequência de caracteres. Se você não especificar um caractere, o default é um espaço em branco. No caso da questão, foi especificado o “O”.

Já no Postgres, a função faz a mesma coisa, remove a cadeia mais longa contendo apenas os caracteres (um espaço por padrão) do início/fim/ambas as extremidades da string. Assim, podemos marcar nossa resposta na alternativa D.

Gabarito: D.

47. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 5ª REGIÃO (BA) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

No PostgreSQL, a função utilizada para obter informações sobre arquivos é chamada

A pg_header_info

B pg_file_info

C pg_stat_file

D pg_read_file

E pg_file_access.

Comentário. Vamos entender cada das alternativas acima. Quais informações cada uma das funções fornece e sobre exatamente o que? As alternativas A, B e E não fazem parte do conjunto de funções utilizadas para acesso aos arquivos. As alternativas pg_stat_file e pg_read_file fazem parte das funções genérica de acesso a dados. Vamos então conhecê-las.

As funções mostradas fornecem acesso nativo a arquivos na máquina que hospeda o servidor. Somente os arquivos dentro do diretório de cluster de banco de dados e do log_directory podem ser acessados. O uso dessas funções é restrito aos superusuários. São quatro as funções:

pg_ls_dir retorna todos os nomes do diretório especificado, exceto as entradas especiais "." e "..".

pg_read_file retorna parte de um arquivo de texto, iniciando no offset determinado, retornando na maioria os bytes definidos pelo length (ao menos que o fim do arquivo seja atingido primeiro). Se o offset for negativo, é relativo ao final do arquivo. Se offset e length são omitidos, o arquivo inteiro é retornado. Os bytes lidos do arquivo são interpretados como uma string na codificação (encoding) do servidor; um erro é lançado, se eles não são válidos para esta codificação.

pg_read_binary_file é semelhante ao *pg_read_file*, exceto pelo resultado que é um valor bytea; por conseguinte, o controle de codificação não é executado. Em combinação com a função *convert_from*, esta função pode ser usada para ler um arquivo em uma codificação especificada:

SELECT convert_from(pg_read_binary_file('arquio_em_utf8.txt'), 'UTF8');

pg_stat_file retorna um registro que contém o tamanho do arquivo, último timestamp de acesso, último timestamp de modificação, timestamp do último status do arquivo de mudança (apenas plataformas Unix), timestamp de criação do arquivo (somente para Windows) e um boolean, indicando se é um diretório. Usos típicos incluem:

SELECT * FROM pg_stat_file('filename');

SELECT (pg_stat_file('filename')).modification;

Vejam que esta última função é a que mais se adequa ao enunciado e, portanto, torna-se nossa resposta.

Gabarito: C.

48. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Em PostgreSQL, se uma função de um gatilho (trigger) executar comandos SQL, existe a possibilidade destes comandos também executarem outros gatilhos. Este processo é conhecido como

A cascading triggers.

B recursive triggers.

C sub-triggers.

D indented triggers.

E multi-trigger.

Comentário. Sabemos que uma função de gatilho executa comandos SQL, então estes comandos podem disparar outros gatilhos. Isto é conhecido como gatilhos em cascata ou ***cascading triggers***. Não há nenhuma limitação para o número de níveis desta cascata. É possível que as cascatas provoquem uma invocação recursiva do mesmo gatilho; por

exemplo, um INSERT pode executar um comando que insere uma linha adicional na mesma tabela, fazendo com que o gatilho INSERT possa ser disparado mais uma vez. É responsabilidade do desenvolvedor evitar recursão infinita em tais cenários.

Dica: Também é possível escrever uma função de gatilho em C, embora a maioria das pessoas ache que é mais fácil usar uma das linguagens procedurais.

Gabarito: A.

49. BANCA: FCC ANO: 2007 ORGÃO: MPU PROVA: Analista de Informática - Banco de Dados

NÃO é um tipo de junção suportado pelos gerenciadores de banco de dados PostgreSQL:

A INNER OUTER JOIN ON.

B LEFT OUTER JOIN ON.

C RIGHT OUTER JOIN ON.

D NATURAL JOIN.

E CROSS JOIN.

Comentário. A questão testa nossos conhecimentos a respeito de junção. Se você observar a sintaxe do comando SELECT, temos a definição das operações de junções possíveis permitidas pelo PostgreSQL. Analisando cada uma das alternativas, encontramos nossa resposta logo na alternativa A. Observem que, segundo a teoria relacionada a junções, não podemos ter uma junção como sendo INNER e OUTER ao mesmo tempo.

Apenas para relembrar, no INNER JOIN, consideramos as linhas cujos atributos participantes da junção possuem valores correspondentes nas duas tabelas que participam da junção. Caso contrário, a linha não aparece no resultado final.

Quando analisamos o comando de OUTER JOIN, temos uma situação diferente. Os valores considerados na junção serão considerados de acordo com um dos modificadores utilizados: LEFT, RIGHT ou FULL. No primeiro, consideramos todas as linhas da tabela à esquerda, preenchendo com NULL quando não existe correspondente na tabela da direita.

Situação inversa ocorre no RIGHT OUTER JOIN. Neste caso, consideramos todas as linhas da tabela da direita, completando com NULL quando não tivermos correspondente na relação da esquerda. Por fim, o FULL OUTER JOIN considera todas as linhas das duas tabelas, preenchendo com NULL quando não existir correspondência.

O NATURAL JOIN é um tipo de INNER JOIN que considera como atributos de junção as colunas das tabelas que possuem os mesmos nomes nas duas relações participantes da junção.

O CROSS JOIN é basicamente um produto cartesiano entre as duas relações participantes da junção.

Vejam que confirmamos nossa resposta na alternativa A.

Gabarito: A.

50. BANCA: FCC ANO: 2013 ORGÃO TRT - 12ª Região (SC) PROVA: Técnico Judiciário - Tecnologia da Informação

No banco de dados PostgreSQL, após uma operação de CROSS JOIN entre uma determinada tabela 1 e uma determinada tabela 2, a tabela resultante irá conter

A ambas as linhas das tabelas 1 e 2, porém somente as colunas que possuam correspondência em ambas as tabelas.

B as colunas da tabela 1 que possuam uma correspondência na tabela 2. O número de linhas será determinado pelas colunas que tenham ocorrência em uma das duas tabelas.

C apenas as linhas da tabela 1 e ambas as colunas da tabela 1 e da tabela 2.

D apenas as colunas da tabela 1 e ambas as linhas da tabela 1 e da tabela 2.

E todas as colunas da tabela 1, seguidas por todas as colunas da tabela 2. Caso as tabelas possuam N e M linhas, respectivamente, a tabela resultante irá conter o produto de N e M (N × M) linhas.

Comentário. Na questão anterior, tercemos comentários teóricos sobre as operações de junção. Aqui vamos apenas relembrar que o CROSS JOIN aplica um produto cartesiano sobre as relações participantes da operação. Assim, cada linha da tabela 1 irá se relacionar com todas as linhas da tabela 2. O resultado desta operação terá M x N linhas, que é o produto da quantidade de linhas de cada relação. Assim, a resposta que se alinha com essa definição está presente na alternativa E.

Gabarito: E.

51. BANCA: FCC ANO: 2013 ORGÃO: TRT - 12ª Região (SC) PROVA: Técnico Judiciário - Tecnologia da Informação

Em PostgreSQL, uma função permite que o processamento da query seja interrompido por um determinado número de segundos. Este comando é chamado de

A until

B delay

C wait

D pg_sleep

E stop

Comentário. Observando as alternativas da questão, temos que as letras A, B, C e E não existem na documentação do PostgreSQL. Resta, portanto, analisarmos a alternativa D.

A função pg_sleep aparece dentro das funções utilizadas para gerar atrasos nas execuções de um processo no servidor. São basicamente três comandos: pg_sleep(N), pg_sleep_for(T) e pg_sleep_until(H). O pg_sleep é para o processo da sessão atual até por N segundos. N é um valor do tipo de precisão dupla, de modo que atrasos fracionários de segundos podem ser especificados. O pg_sleep_for é uma função de conveniência para os tempos especificados como um intervalo. pg_sleep_until é uma função utilizada quando queremos definir uma hora específica para o despertar do processo. Vejamos alguns exemplos:

```
SELECT pg_sleep(1.5);
SELECT pg_sleep_for('5 minutes');
SELECT pg_sleep_until('tomorrow 03:00');
```

Gabarito: D.

52. BANCA: FCC ANO: 2013 ORGÃO: TRT - 12ª Região (SC) PROVA: Analista Judiciário - Tecnologia da Informação

Considere o trecho do comando em SQL abaixo.

CREATE USER MAPPING FOR

Este comando é utilizado para a criação de um mapeamento do usuário para

A um trecho de pesquisa específico.

B uma tabela remota.

C um banco de dados do tipo *cluster*.

D uma visualização (VIEW).

E um servidor estrangeiro (*foreign server*).

Comentário. O comando CREATE USER MAPPING FOR, cuja sintaxe completa pode ser vista abaixo, é utilizado para definir um novo mapeamento entre um usuário e um servidor externo ou estrangeiro.

```
CREATE USER MAPPING FOR { user_name | USER | CURRENT_USER | PUBLIC }
    SERVER server_name
    [ OPTIONS ( option 'value' [ , ... ] ) ]
```

Gabarito: E.

53. BANCA: FCC ANO: 2012 ORGÃO: MPE-AP – Prova: Analista Ministerial - Tecnologia da Informação

No banco de dados *PostgreSQL*, o comando MOVE é utilizado para reposicionar

A uma tabela em um banco de dados.

B uma linha em uma tabela.

C uma coluna em uma linha.

D o cursor sem trazer dados.

E uma view em uma tabela

Comentário. Vamos aproveitar essa questão para falar um pouco sobre cursores. Em vez de executar uma consulta inteira de uma só vez, é possível criar um **cursor** que encapsula a consulta e, em seguida, ler algumas linhas de cada vez. Uma razão para fazer isso é para evitar o estouro de memória quando o resultado contém um grande número de linhas. (No entanto, os usuários PL/pgSQL normalmente não precisam se preocupar com isso, uma vez que, quando utilizamos loops, automaticamente um cursor é criado internamente para evitar problemas de memória.) Um uso mais interessante é retornar uma referência a um cursor criado por uma função, permitindo ler as linhas. Isso fornece uma maneira eficiente para retornar grandes conjuntos de linhas em funções.

Todo o acesso aos cursores em PL/pgSQL passa por variáveis de cursor, que são sempre do tipo de dados especial refcursor. Uma maneira de criar uma variável de cursor é apenas para declará-la como uma variável do tipo refcursor. Outra forma é usar a sintaxe declaração do cursor, o que em geral é:

name [[NO] SCROLL] CURSOR [(arguments)] FOR query;

Antes de um cursor ser utilizado para recuperar linhas, ele deve ser aberto. (Esta é a ação equivalente ao comando SQL DECLARE CURSOR.) PL/pgSQL possui três formas de instrução OPEN. A primeira possibilidade seria o OPEN FOR query (sintaxe: OPEN unbound_cursorvar [[NO] SCROLL] FOR query). Neste caso, a variável cursor é aberta e a consulta especificada é entregue para execução. O cursor não pode estar aberto e deve ter sido declarado como uma variável de cursor não ligado (ou seja, como uma variável refcursor simples). A consulta deve ser feita por meio do SELECT, ou qualquer outro comando que retorne linhas (como EXPLAIN).

A outra opção seria OPEN FOR EXECUTE (sintáxe: OPEN unbound_cursorvar [[NO] SCROLL] FOR EXECUTE query_string [USING expression [, ...]];). E a última opção seria OPEN BOUND CURSOR (OPEN bound_cursorvar [([argument_name: =] argument_value [, ...])];) Esta forma de OPEN é usada para abrir uma variável cursor cuja consulta foi vinculada a ela quando foi declarada. Uma lista de expressões com valores de argumentos deve aparecer se e somente se o cursor foi declarado com estes argumentos. Estes valores serão substituídos na consulta.

Uma vez que o cursor tenha sido aberto, ele pode ser manipulado com as instruções descritas abaixo. Estas manipulações podem não ocorrer na mesma função que abriu o cursor. Você pode retornar um valor *refcursor* fora da função e deixar outra função operar o cursor. São três os principais comandos:

1. FETCH - recupera a próxima linha do cursor no *target*, que pode ser uma linha, um registro, ou uma lista de variáveis simples, separadas por vírgulas, da mesma forma que o

comando SELECT INTO. Se não houver nenhuma linha seguinte, o alvo é definido como NULL(s), tal como acontece com SELECT INTO. A variável especial FOUND pode verificar se uma linha foi obtida ou não.

2. MOVE - reposiciona um cursor sem recuperar os dados. O MOVE funciona exatamente como o comando FETCH, exceto que só reposiciona o cursor e não retorna a linha para qual mudou o ponteiro. Aqui também é possível usar a variável especial FOUND para verificar se uma linha será obtida pelo próximo comando MOVE.

3. CLOSE - fecha o cursor aberto. Pode ser utilizado para libertar recursos antes do final da operação, ou para liberar a variável cursor para ser aberta novamente.

Vejam que o exposto acima apenas confirma nossa resposta na alternativa D.

Gabarito: D.

54. BANCA: FCC ANO: 2012 ORGÃO: MPE-AP - Analista Ministerial PROVA: Tecnologia da Informação

No banco de dados *PostgreSQL,* o comando utilizado para efetivar a transação corrente é chamado

A END.

B ROLLBACK.

C TRANSFER.

D EFFECTIVE.

E SELECT.

Comentário. São seis os principais comandos utilizados para trabalhar com transações dentro do PostgreSQL: START TRANSACTION, BEGIN, COMMIT, ROLLBACK, SAVEPOINT e SET TRANSACTION. Vejamos a sintaxe e a definição de cada uma delas.

START TRANSACTION - Este comando inicia um novo bloco de transação. Se o nível de isolamento, modo de leitura/gravação, ou o modo deferrable forem especificados, a nova transação tem essas características, como se SET TRANSACTION fosse executado. Este é o mesmo que o comando BEGIN.

```
START TRANSACTION [ transaction_mode [, ...] ]

where transaction_mode is one of:

    ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ | READ COMMITTED | READ UNCOMMITTED }
    READ WRITE | READ ONLY
    [ NOT ] DEFERRABLE
```

O BEGIN tem a mesma função do START TRANSACTION. Ou seja, o BEGIN inicia um bloco de transação, todas as declarações após o comando BEGIN serão executadas em uma única transação até que um COMMIT ou ROLLBACK explícito. Por padrão (sem o

BEGIN), o PostgreSQL executa as transações em modo "autocommit", isto é, cada comando é executado em sua própria transação e uma efetivação é implicitamente realizada ao final do comando (se a execução for bem-sucedida; caso contrário, um rollback é realizado).

Os comandos são executados mais rapidamente em um bloco de transação, porque a transação iniciar/commit requer uso significativo de CPU e do disco. A execução de vários comandos em uma única transação também é útil para garantir a consistência ao fazer várias alterações relacionadas: outras sessões não serão capazes de ver os estados intermediários das atualizações relacionadas.

O comando de COMMIT efetiva a transação corrente. Todas as alterações feitas pela transação se tornam visíveis para os outros e são garantidas para serem duráveis, mesmo que uma falha aconteça. Sua sintaxe é bem simples:

COMMIT [WORK | TRANSACTION]

Uma opção ao COMMIT é o END, que também confirma a transação atual. Este comando é uma extensão do PostgreSQL, que é equivalente ao COMMIT.

ROLLBACK reverte a transação atual e faz com que todas as modificações realizadas pela transação sejam descartadas. Sua sintaxe é semelhante à do commit.

ROLLBACK [WORK | TRANSACTION]

O SAVEPOINT estabelece um novo savepoint dentro da transação corrente. Um savepoint é uma marca especial dentro de uma transação, permitindo que todos os comandos que são executados antes da sua criação sejam revertidos, restaurando o estado de transação para o momento do savepoint.

Observem que, pelo exposto, podemos marcar a alternativa A, visto que COMMIT e END provocam o mesmo resultado para o PostgreSQL.

Gabarito: A.

---

55. BANCA: FCC ANO: 2012 ORGÃO: MPE-AP - Técnico Ministerial PROVA: Informática

Em bancos de dados *PostgreSQL*, o comando *DECLARE* é utilizado para

A criar uma classe de operadores que define como um determinado tipo de dado pode ser usado em um índice.

B criar cursores, que podem ser utilizados para retornar, de cada vez, um pequeno número de linhas em uma consulta.

C criar uma tabela, inicialmente vazia, no banco de dados corrente.

D registrar um novo tipo de dado para uso no banco de dados corrente.

E registrar uma nova linguagem procedural a ser utilizada em consultas ao banco de dados.

Comentário. Vejam que podemos pensar em duas opções para o uso do DECLARE: quando queremos criar um cursor; ou quando estamos utilizando PL/pgSQL, e utilizamos o DECLARE no bloco para declaramos as variáveis a serem utilizadas pelo comando. A questão optou por questionar o uso do DECLARE na criação de cursores. Observem que as demais alternativas não fazem sentido. Como tarefa, tente achar o comando que executa cada uma das operações descritas nas demais alternativas.

Gabarito: B.

---

56. BANCA: FCC ANO: 2012 ORGÃO: MPE-AP - Técnico Ministerial PROVA: Informática

Quando o nível de isolamento de uma transação em *SQL* no banco de dados *PostgreSQL* é definido como serializável *(Serializable*), o comando *SELECT* enxerga apenas os dados efetivados

A durante a transação, desde que as transações concorrentes tenham feito *COMMIT.*
B por transações simultâneas.
C após o início da transação, desde que as transações simultâneas tenham efetivado as alterações no banco de dados.
D antes de a transação começar.
E durante a transação, desde que as transações concorrentes não tenham feito *COMMIT.*

Comentário. Sabemos que o nível de isolamento serializável faz as operações serem executadas isoladamente. Nestes casos, só podemos visualizar dados efetivados antes de a transação começar.

Gabarito: D.

---

57. BANCA: FCC ANO: 2012 ORGÃO: MPE-PE - Técnico Ministerial PROVA: Informática

Em *PostgreSQL*, um conjunto de funções e expressões estão disponíveis para a geração de arquivos XML. A função, similar a função *xmlconcat*, que concatena as colunas xml entre linhas de uma tabela é denominada de

A *xmllist.*

B *xmlrowcat.*

C *xmlgrep.*

D *xmlagg.*

E *xmlconcr.*

Comentário. Vejamos as ações feitas por cada uma das funções xml da questão. Antes, porém, precisamos eliminar da nossa analise as alternativas que não são funções XML. Eliminamos as letras A, B, C e E.

A função xmlagg é, ao contrário das outras funções descritas na documentação, uma função de agregação. Ela concatena os valores de entrada para a chamada de função agregada, tal como XMLCONCAT. A diferença é que a concatenação ocorre em todas as linhas, em vez de em uma única linha.

Para a lista completa das funções XML, sugiro olhar a documentação oficial aqui.

Gabarito: D.

LISTA DE QUESTÕES – MULTIBANCAS

1. DIRENS Aeronáutica - Estágio de Adaptação à Graduação de Sargento (EEAR)/Informática/2019/EAGS 2020

Em se tratando do utilitário psql, para a criação de um banco de dados no PostgreSQL, devemos recorrer a esse utilitário para saber quais são os bancos de dados que já foram criados e quais são as tabelas e os índices existentes em cada um.

Considerando essa proposição, relacione as colunas de acordo com cada definição e assinale a alternativa correta.

1 – \?

2 – \h

3 – \di

4 – \dt

( ) Para visualizar os índices.

( ) Para obter ajuda na sintaxe de comandos SQL.

( ) Para visualizar o nome das tabelas existentes.

( ) Para obter acesso a todas as opções oferecidas pelo gerenciador.

a) 3 – 1 – 4 – 2

b) 2 – 1 – 4 – 3

c) 3 – 2 – 4 – 1

d) 4 – 1 – 2 – 3

2. CEBRASPE (CESPE) - Auxiliar Judiciário (TJ PA)/Programador de Computador/2020

No sistema de gerenciamento de banco de dados PostgreSQL, para criar uma tabela contendo uma coluna com um tipo de dados inteiro e com a propriedade de autoincremento, é correto o uso de dados dos tipos

a) boolean e bigint.

b) character varying e cidr.

c) inet e integer.

d) smallint e real.

e) serial e bigserial.

3. IBFC - Técnico Judiciário (TRE PA)/Apoio Especializado/Operação de Computadores/2020

Quanto às principais características do PostgreSQL, analise as afirmativas abaixo e dê valores Verdadeiro (V) ou Falso (F).

( ) não impõe limites no tamanho de armazenamento dos tipos de dados.

( ) suporta um único tipo de índice, denominado índice em cluster (clustered index).

( ) o PostgreSQL possui o maior TCO (Total Cost of Ownership) dos SGBD's.

Assinale a alternativa que apresenta a sequência correta de cima para baixo.

a) V, F, F

b) V, V, F

c) F, V, V

d) F, F, V

4. NC-UFPR - Profissional Nível Universitário Jr (ITAIPU)/Gestão da Informação/2019

Considere a seguinte instrução SQL:

```
WITH RECURSIVE cte(n) AS (
  SELECT 1
  UNION ALL
  SELECT n+1 FROM cte WHERE n<5
)
SELECT * FROM cte;
```

Ao ser executada no PostgreSQL, ela produz como resultado:

a) 0,1,2,3,4

b) 1,1,2,3,4

c) 1,2,3,4,5

d) 1,2,3,4

e) 1

5. NC-UFPR - Profissional Nível Universitário Jr (ITAIPU)/Gestão da Informação/2019

Em relação à busca por texto utilizando os operadores LIKE, SIMILAR TO, expressão regular, Full Text Search (FTS), funções e operadores relacionados à busca textual no PostgreSQL, é correto afirmar:

a) O operador LIKE realiza busca por semelhança de palavras, resolvendo o problema de ortografia incorreta.

b) O operador SIMILAR TO realiza busca baseada em expressões regulares, realizando o ranqueamento de semelhança entre as palavras do resultado em relação às palavras da busca.

c) Erros de ortografia podem ser tratados pelo mecanismo de busca FTS, por semelhança entre os termos.

d) O operador @@ do PostgreSQL é equivalente ao operador RLIKE do MySQL.

e) Os tipos tsquery e tsvector, criados respectivamente pelas funções to_tsquery e to_tsvector, são conjuntos de trigrans das strings informadas, que são comparadas em ordem alfabética durante a busca.

6. VUNESP - Técnico de Tecnologia da Informação (UFABC)/2019

No Sistema Gerenciador de Bancos de Dados PostgreSQL (versão 9), há um comando que exibe o plano que o gerenciador irá utilizar para realizar uma determinada consulta.

O comando descrito é o

a) EXPLAIN.

b) OFFSET.

c) NOTIFY.

d) LOCK.

e) FETCH.

Comentário: Vamos analisar cada uma das alternativas:

a) CORRETA. O comando EXPLAIN é utilizado para exibir o plano de execução de uma determinada consulta.

b) ERRADA. A cláusula OFFSET serve para retornar apenas uma parte do resultado de uma consulta, indicando quantas linhas a partir do início do resultado serão ignoradas.

c) ERRADA. O comando NOTIFY envia uma notificação aos clientes que se inscreveram para receber notificações através do comando LISTEN.

d) ERRADA. LOCK é um comando utilizado para travar uma tabela.

e) ERRADA. O comando FETCH é utilizado para retornar linhas de uma consulta utilizando um cursor criado previamente.

Gabarito: A.

---

7. CEBRASPE (CESPE) - Analista de Gestão de Resíduos Sólidos (SLU DF)/Informática/2019

No que diz respeito a ferramentas de desenvolvimento, julgue o item a seguir.

O PostgreSQL é um sistema gerenciador de banco de dados objeto-relacional (ORDBMS) que oferece suporte a tipos de dados especializados como o JSon e o JSonb.

8. VUNESP - Analista de Tecnologia da Informação (Pref Olímpia)/2019

O sistema gerenciador de bancos de dados PostgreSQL (versão 9.5) possui os seguintes modos de desligamento (shutdown):

a) Hard, Premium e Single.

b) Hibernate, Full e Soft.

c) Initial, Intermediate e Final.

d) Partial, Permanent e Semi-permanent.

e) Smart, Fast e Immediate.

9. CEBRASPE (CESPE) - Analista de Tecnologia da Informação (TCE-RO)/Desenvolvimento de Sistemas/2019

Em geral, a sintaxe para a criação de índice em banco de dados relacional segue uma estrutura- padrão, como demonstra, por exemplo, a seguinte estrutura no banco relacional PostgreSQL, em versão 9 ou superior.

CREATE [UNIQUE] INDEX [CONCURRENTLY] [name] ON table [USING method]

Tendo como referência essas informações, assinale a opção correta.

a) CREATE INDEX constrói uma linha de índice de acordo com uma coluna específica da tabela.

b) O parâmetro method depende do tamanho da tabela e não deve ser utilizado se o tamanho da tabela for menor que 1 MB.

c) Um campo de índice não pode ser uma expressão calculada a partir dos valores de uma ou mais colunas da tabela.

d) O método de indexação btree armazena dados de forma que cada nó contenha chaves em ordem crescente.

e) Quando a cláusula WHERE está presente, um índice total é criado, porque a cláusula já é restritiva na operação de selecionar dados ou de inserir dados.

10. CEBRASPE (CESPE) - Analista Judiciário (TJ AM)/Analista de Sistemas/2019

A respeito de bancos de dados relacionais, julgue o item a seguir.

Em um banco de dados PostgreSQL, a manipulação de ROLES é feita exclusivamente por comandos CREATE e DROP fornecidos com o banco de dados.

11. IDECAN - Técnico (IF Baiano)/Tecnologia da Informação /2019

Sobre o PostgreSQL, assinale a alternativa correta.

a) Só é possível instalar o PostgreSQL com privilégios de super usuário, ou seja, é necessário o acesso de usuário root do sistema.

b) Existem três abordagens fundamentais de backup no PostgreSQL: SQL dump, backup a nível de arquivos e arquivamento contínuo.

c) O comando para se efetuar um dump de um banco de dados chamado dbname gerando um arquivo dumpfile é o seguinte: psql dbname < dumpfile.

d) O comando createdb mydb, considerando que o PostgreSQL está instalado corretamente, cria uma tabela de nome mydb.

e) O comando dropdb mydb remove todos os arquivos associados a um banco de dados. No entanto, o PostgreSQL cria um backup antes.

12. BANCA: CESPE ANO: 2014 ÓRGÃO: ANATEL PROVA: ANALISTA ADMINISTRATIVO - SUPORTE E INFRAESTRUTURA DE TI

A respeito de banco de dados, julgue os itens que se seguem.

O PostgreSQL 9.3, ao gerenciar o controle de concorrência, permite o acesso simultâneo aos dados. Internamente, a consistência dos dados é mantida por meio do MVCC (*multiversion concurrency control*), que impede que as transações visualizem dados inconsistentes.

13. BANCA: FCC ANO: 2012 ÓRGÃO: TCE-AM PROVA: ANALISTA TÉCNICO DE CONTROLE EXTERNO - TECNOLOGIA DA INFORMAÇÃO

Sobre os fundamentos arquiteturais do banco de dados PostgreSQL, considere:

I. Utiliza um modelo cliente/servidor, consistindo de um processo servidor que gerencia os arquivos do banco de dados, controla as conexões dos clientes ao banco dados e efetua ações no banco de dados em favor dos clientes.

II. A aplicação cliente, que irá efetuar as operações no banco de dados, poderá ser de diversas naturezas, como uma ferramenta em modo texto, uma aplicação gráfica, um servidor web que acessa o banco de dados para exibir as páginas ou uma ferramenta de manutenção especializada.

III. A aplicação cliente pode estar localizada em uma máquina diferente da máquina em que o servidor está instalado. Neste caso, a comunicação entre ambos é efetuada por uma conexão TCP/IP. O servidor pode aceitar diferentes conexões dos clientes ao mesmo tempo.

Está correto o que se afirma em

A I, II e III.

B I e II, apenas.

C I e III, apenas.

D II e III, apenas.

E III, apenas.

14. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Localização refere-se ao fato de uma aplicação respeitar as preferências culturais sobre alfabetos, classificação, formatação de números etc. PostgreSQL usa o padrão ISO C e POSIX fornecidos pelo sistema operacional do servidor para aplicar as regras de localização. O suporte à localização é automaticamente inicializado quando um cluster de banco de dados é criado usando o comando

A create cluster.

B create database.

C initdb.

D ccluster.

E locale init.

15. BANCA: FCC ANO: 2014 ÓRGÃO: TJ-AP PROVA: ANALISTA JUDICIÁRIO - BANCO DE DADOS - DBA

Um dos itens da administração do sistema gerenciador de bancos de dados PostgreSQL (V.9.3.4) refere-se a gerenciar informações sobre os bancos de dados por ele controlados. O PostgreSQL contém algumas visões que auxiliam nessa tarefa, dentre elas, a visão pg_settings que contém dados sobre

A os parâmetros run-time do servidor.

B estatísticas das tabelas do servidor.

C os usuários dos bancos de dados.

D a lista de bloqueios impostos.

E a lista das funções presentes no banco de dados.

16. BANCA: CESPE ANO: 2014 ÓRGÃO: ANATEL PROVA: ANALISTA ADMINISTRATIVO - SUPORTE E INFRAESTRUTURA DE TI

Julgue o item abaixo:

A conexão com o PostgreSQL 9.3 é realizada, por padrão, na porta TCP 5432. Uma das configurações de segurança permitida é o acesso por meio de SSL que é true, por padrão, e é aceito, neste caso, com o uso dos protocolos TCP, IP ou NTP.

17. BANCA: FCC - Analista em Gestão (DPE AM)/Especializado em Tecnologia da Informação de Defensoria/Analista de Sistema/2018

No PostgreSQL 9.0, para efetuar o backup e a restauração de um banco de dados utilizam-se, respectivamente, os comandos

a) bman e rman.

b) backup e restore.

c) sqlbac e sqlrest.

d) pg_dump e psql.

e) sql_backup e sql_restore.

18. BANCA: FCC ANO: 2014 ÓRGÃO: TRT - 13ª REGIÃO (PB) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Paulo utiliza o pg_dump do PostgreSQL para fazer cópia de segurança de um banco de dados. Normalmente faz cópias de segurança no formato tar e utiliza o pg_restore para reconstruir o banco de dados, quando necessário. O pg_restore pode selecionar o que será restaurado, ou mesmo reordenar os itens antes de restaurá-los, além de permitir salvar e restaurar objetos grandes. Certo dia Paulo fez uma cópia de segurança do banco de dados chamado trt13 para o arquivo tribunal.tar, incluindo os objetos grandes. Paulo utilizou uma instrução que permitiu a seleção manual e reordenação de itens arquivados durante a restauração, porém, a ordem relativa de itens de dados das tabelas não pôde ser alterada durante o processo de restauração. Paulo utilizou, em linha de comando, a instrução

A pg_dump -Ec -h trt13 > tribunal.tar

B pg_dump -Ft -b trt13 > tribunal.tar

C pg_dump -tar -a trt13 > tribunal.tar

D pg_dump -tar -c trt13 > tribunal.tar

E pg_dump -Fp -b trt13 > tribunal.tar

19. BANCA: FCC ANO: 2013 ORGÃO: TRT - 12ª Região (SC) PROVA: Analista Judiciário - Tecnologia da Informação

O comando em SQL capaz de serializar dados de uma tabela para um arquivo em disco, ou efetuar a operação contrária, transferindo dados de um arquivo em disco para uma tabela de um banco de dados, é o comando:

(a) COPY.

(b) TRANSFER.

(c) SERIALIZE.

(d) FILE TRANSFER.

(e) EXPORT.

20. BANCA: CESPE - Oficial Técnico de Inteligência/Área 9/2018

Julgue o próximo item, a respeito de conceitos e comandos PostgreSQL e MySQL.

No programa psql do PostgreSQL, a instrução \h permite mostrar o histórico de comandos SQL na sessão atual.

Certo

Errado

21. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

A instrução SQL em PostgreSQL abaixo está mal formulada.

```
CREATE VIEW vista AS SELECT 'Hello World';
```

Isto aconteceu, porque

A a criação de uma visualização requer a utilização da cláusula WHERE para a restrição dos dados.

B não é possível criar uma VIEW sem a identificação do tipo de dado e sem a determinação da quantidade de registros selecionados.

C o comando CREATE VIEW deve utilizar a cláusula FROM para o nome da tabela.

D a criação de uma visualização (VIEW) requer a definição de um gatilho (trigger) correspondente ao nome da coluna.

E por padrão, o tipo de dado será considerado indefinido (unknown) e a coluna irá utilizar o nome padrão ?column?.

22. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Considere o trecho em PostgreSQL abaixo.

INSERT INTO products (product_no, name, price) VALUES (1, 'Cheese', 9.99), (2,'Bread',1.99), (3,'Milk', 2.99);

Considerando a existência prévia da tabela products contendo as colunas product_no, name e price, e desconsiderando os tipos de dados, esse trecho irá resultar:

A na adição de 3 novas colunas na tabela products.

B na adição de 3 novas linhas na tabela products.

C em erro, pois não é possível múltiplas inserções em um único comando SQL.

D em erro, pois para se realizar múltiplas inserções é necessário a utilização da cláusula SELECT.

E em erro, pois múltiplas inserções são possíveis somente com a utilização de colchetes para a limitação dos registros.

23. BANCA: FCC - Analista Judiciário (TRT 23ª Região)/Apoio Especializado/Tecnologia da Informação/2016

São vários os tipos de dados numéricos no PostgreSQL. O tipo

a) smallint tem tamanho de armazenamento de 1 byte, que permite armazenar a faixa de valores inteiros de −128 a 127.

b) bigint é a escolha usual para números inteiros, pois oferece o melhor equilíbrio entre faixa de valores, tamanho de armazenamento e desempenho.

c) integer tem tamanho de armazenamento de 4 bytes e pode armazenar valores na faixa de −32768 a 32767.

d) numeric pode armazenar números com precisão variável de, no máximo, 100 dígitos.

e) serial é um tipo conveniente para definir colunas identificadoras únicas, semelhante à propriedade auto incremento.

24. BANCA: FCC - Técnico Judiciário (TRE PB)/Apoio Especializado/Programação de Sistemas/2015

No PostgreSQL, o tipo de dados numérico considerado meramente uma notação conveniente para definir colunas identificadoras únicas, semelhante à propriedade auto incremento em alguns Sistemas Gerenciadores de Banco de Dados, é o tipo

a) serial.

b) smallint.

c) byte.

d) bit.

e) blob.

25. BANCA: FCC ANO: 2014 ÓRGÃO: TRT - 1ª REGIÃO (RJ) PROVA: TÉCNICO DO JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

No sistema gerenciador de Banco de Dados PostgreSQL (v. 9.1), a forma para se declarar um atributo com o tipo de dados Array, com duas dimensões, tendo o nome teste é

A . . . teste [[ ]] . . .

B . . . teste 2 [ ] . . .

C . . . teste [2] . . .

D . . . teste [ ] [ ] . . .

E . . . teste [ ] x [ ] . . .

26. BANCA: FCC ANO: 2012 ÓRGÃO: TCE-AM PROVA: ANALISTA TÉCNICO DE CONTROLE EXTERNO - TECNOLOGIA DA INFORMAÇÃO

Em PostgreSQL, a função que converte a primeira letra da string informada em letra maiúscula, alterando todas as letras subsequentes dessa string para minúsculas se chama

A chgstr.

B altertext.

C initcap.

D upper.

E toupper.

27. BANCA: FCC ANO: 2012 ORGÃO: MPE-PE - Analista Ministerial PROVA: Informática

No banco de dados *PostgreSQL*, a função COALESCE

A retorna verdade ou falso conforme o padrão corresponda ou não à cadeia de caracteres fornecida. Este operador é muito semelhante ao LIKE, exceto por interpretar o padrão utilizando a definição de expressão regular do padrão SQL.

B é uma expressão condicional genérica, semelhante às declarações *if/else* de outras linguagens.

C é uma declaração SELECT arbitrária, ou uma subconsulta. A subconsulta é processada para determinar se retorna alguma linha.

D retorna o primeiro de seus argumentos que não for nulo. Só retorna nulo quando todos os seus argumentos são nulos.

E permite a conversão do carimbo do tempo (*time stamp*) para uma zona horária diferente.

28. BANCA: CESPE - Analista Judiciário (STM)/Apoio Especializado/Análise de Sistemas/2018

Julgue o item subsequente, a respeito do Postgres 9.6.

Ao se criar uma trigger, a variável especial TG_OP permite identificar que operação está sendo executada, por exemplo, DELETE, UPDATE, INSERT ou TRUNCATE.

Certo

Errado

29. Ano: 2018 Banca: CESPE Órgão: STM Cargo: Analista de Sistemas Questão: 71 e 72

Um sistema gerenciador de banco de dados (SGBD) instalado no Linux deve ser configurado de modo a permitir os seguintes requisitos:

I no máximo, 1000 conexões simultâneas;

II somente conexões originadas a partir do servidor de

aplicação com IP 10.10.10.2.

Tendo como referência essas informações, julgue os seguintes itens.

71 Caso o SGBD instalado seja o Postgres 9.6, para atendimento do requisito I, deve-se modificar o arquivo postgres.conf para o referido cluster; alterar o parâmetro max_connections para 1000; e reiniciar o serviço do SGBD.

72 Caso o SGBD instalado seja o Postgres 9.6, para atendimento do requisito II, deve-se modificar o arquivo pg_hba.conf para o referido cluster; alterar o parâmetro listen_addresses para o IP fornecido; e reiniciar o serviço do SGBD.

30. Ano: 2018 Banca: CESPE Órgão: STM Cargo: Analista de Sistemas Questão: 82

Julgue os itens subsequentes, a respeito do Postgres 9.6.

82 Nas instruções seguintes, a palavra-chave IMMUTABLE indica que a função criada não pode modificar o banco de dados. CREATE FUNCTION add(integer, integer) RETURNS integer

AS 'select $1 + $2;'

LANGUAGE SQL

IMMUTABLE

RETURNS NULL ON NULL INPUT;

31. Ano: 2018 Banca: CESPE Órgão: STM Cargo: Programação de Sistemas Questão: 66 a 70

Julgue os próximos itens, que dizem respeito aos SGBDs Oracle, MySQL e PostgreSQL.

69 Uma desvantagem do PostgreSQL em relação aos demais SGBDs é que ele não oferece recursos necessários para se realizar a replicação de dados.

32. Ano: 2018 Banca: CESPE Órgão: STM Cargo: Analista de Sistemas Questões: 86 e 88

Julgue os itens que se seguem, a respeito do processamento de transações e otimização de desempenho do SGBD e de consultas SQL.

86 O controle de nível de isolamento de transações é importante para gerenciar a forma como as transações concorrentes se comportarão no SGBD. No Postgres 9.6, o nível de isolamento padrão é READ COMMITTED, mas pode ser alterado para

SERIAZABLE por meio do comando SET TRANSACTION ISOLATION LEVEL SERIAZABLE.

88 No Oracle 12C, a Automatic Workload Repository (AWR) é uma funcionalidade similar ao autovacuum no Postgres 9.6, haja vista que ambos processam e mantêm estatísticas de desempenho para detecção de problemas e manutenção automática do banco de dados, por exemplo, reusando, ajustando e excluindo dados temporários e reusando espaço em blocos por linhas excluídas.

33. BANCA: CESPE - Oficial Técnico de Inteligência/Área 8/2018

A respeito de sistemas gerenciadores de banco de dados, julgue o próximo item.

No arquivo pg_hba.conf de configuração do PostgreSQL, as diretivas são avaliadas a partir da linha superior, para a linha inferior.

Certo

Errado

34. BANCA: FCC - Técnico Judiciário (TRT 2ª Região)/Apoio Especializado/Tecnologia da Informação/2018

Considere que um Técnico de TI deseja criar as tabelas abaixo em um banco de dados PostgreSQL 8 aberto e em condições ideais.

```
CREATE TABLE departamento (
        codDep     varchar(10) primary key,
        local point
);

CREATE TABLE funcionario (
        codDep      varchar(10)  I ,
        salario        real,
        dataAdm    date
);
```

Para que codDep na tabela funcionario seja definido como chave estrangeira com relação à tabela departamento, a lacuna I deve ser preenchida com

a) foreign key departamento(codDep)

b) constraint foreign key REF departamento(codDep)

c) foreign key IN departamento(codDep)

d) constraint departamento(codDep)

e) references departamento(codDep)

35. BANCA: CESPE - Analista Judiciário (TRT 7ª Região)/Apoio Especializado/Tecnologia da Informação/2017

No sistema gerenciador de banco de dados PostgreSQL, a restrição de acesso pelo endereço IP do cliente é feita mediante alteração do arquivo de configuração

a) pg_subtrans.

b) pg_hba.conf.

c) postmaster.opts.

d) pg_ctl.

36. BANCA: FCC - Técnico de Nível Superior (PGM Teresina)/Analista de Sistemas/2016

Uma das recomendações na prática do tuning no PostgreSQL como forma de melhorar o desempenho das tabelas com grandes quantidades de registros e especialmente com muitos acessos é a inserção de

a) índices em todos os campos das tabelas.

b) valores nulos em campos que compõem a cláusula WHERE ou que fazem parte de cláusulas ORDER BY, GROUP BY.

c) valores nulos em campos que compõem o statement SELECT sem cláusula WHERE.

d) índices em campos que compõem a cláusula WHERE ou que fazem parte de cláusulas ORDER BY, GROUP BY.

e) índices em todos os campos que compõem um statement SELECT DISTINCT e/ou cláusula CONSTRAINT.

37. BANCA: FCC - Analista de Sistemas (DPE RR)/2015

Um Analista de Sistemas deseja fazer uma cópia de segurança consistente de um banco de dados PostgreSQL, mesmo que ele esteja sendo utilizado por outros usuários, gerando um arquivo texto contendo comandos SQL. Estes comandos, ao serem processados pelo servidor, recriam o banco de dados no mesmo estado em que este se encontrava quando o arquivo foi gerado. O Analista deve usar o utilitário

a) pg_dump.

b) psql.

c) pgbackup.

d) gunzip.

e) sql_backup.

38. BANCA: CESPE ANO: 2015 ÓRGÃO: CGE-PI PROVA: AUDITOR GOVERNAMENTAL - TECNOLOGIA DA INFORMAÇÃO

Acerca de bancos de dados, julgue os itens a seguir.

No PostgreSQL 9.3, os arquivos de WAL (write-ahead logging), que armazenam as transações do SGBD na forma de segmentos de log, são gravados por padrão no diretório pg_wal abaixo do diretório data.

39. Ano: 2018 Banca: FCC Órgão: DPE-AM Cargo: Analista Área: Banco de Sistemas Questão: 46

Considere as instruções SQL a seguir, digitadas no PostgreSQL 9.0, em condições ideais.

CREATE TABLE processo (proc_num character(24));

INSERT INTO processo SELECT '0000125–40.' || '1981.403.6100';

SELECT proc_num, octet_length(proc_num) FROM processo;

Será exibido na tela

(A) 0000125–40.1981.403.6100 e 24.

(B) uma mensagem de erro, pois para concatenar valores, no lugar de || deve ser usado &&.

(C) 1981.403.6100 e 24.

(D) uma mensagem de erro, já que o comando octet_length não existe.

(E) uma mensagem de buffer overflow.

40. Ano: 2018 Banca: FCC Órgão: TRT-06 Cargo: Analista Judiciário Área: Tecnologia da Informação Questão: 34

Em um banco de dados PostgreSQL aberto e em condições ideais, um Analista especializado em Tecnologia da Informação executou as instruções abaixo em uma tabela chamada funcionario.

BEGIN;

UPDATE funcionario SET salario = salario - 1000.00

WHERE nome = 'João';

SAVEPOINT ps1;

UPDATE funcionario SET salario = salario + 1000.00

WHERE nome = 'Paulo';

I

UPDATE salario SET salario = salario + 1000.00

WHERE nome = 'Marcos';

COMMIT;

Na segunda instrução UPDATE, o Analista aumentou o salário do funcionário Paulo em 1000.00, quando deveria aumentar o salário do funcionário Marcos nesse valor. Para cancelar a operação realizada, a lacuna I deve ser preenchida pela instrução

(A) ROLLBACK -1;

(B) ROLLBACK TO ps1;

(C) CANCEL OPERATION;

(D) RESTORE TO ps1;

(E) CANCEL UPDATE;

41. Ano: 2017 Banca: FCC Órgão: TRT-11 Cargo: Técnico Judiciário de TI – Q. 46

Um comando SQL, cuja sintaxe é válida no PostgreSQL 9.3, está apresentado em:

(A) SELECT nome1, nome2, nome 3 FROM tab2TRT ORDER FOR nome1 AFTER nome2;

(B) DELETE FROM tabTRT WHERE data processo= “13-AGO-2013”;

(C) UPDATE trabTRT SETTING mes = 10 INSTEAD mes = 5;

(D) CREATE TABLE tab2 TRT { id integer, nome texts, salario numerical};

(E) INSERT INTO tabTRT DEFAULT VALUES;

42. BANCA: FCC ANO: 2017 ÓRGÃO: TST PROVA: ANALISTA JUDICIÁRIO – SUPORTE EM TECNOLOGIA DA INFORMAÇÃO

[70] Um Analista de Suporte que utiliza o PostgreSQL possui uma tabela chamada employee, com os campos id, name e salary. Deseja executar uma consulta que exiba todos os nomes e salários dos funcionários, de forma que, se o salário for nulo, exiba o valor 0 (zero). Para realizar a consulta terá que utilizar a instrução SELECT name,

a) NVDL(salary, 0) FROM employee;

b) IFNL(salary, 0) FROM employee;

c) COALESCE(salary, 0) FROM employee;

d) IFNULL(salary; '0') FROM employee;

e) NVL(salary; 0) FROM employee;

43. BANCA: FCC ANO: 2017 ÓRGÃO: DPE-RS PROVA: ANALISTA – BANCO DE DADOS

[49] No sistema gerenciador de banco de dados PostgreSQL 8 (versão 8.3) há duas bases de dados padrão utilizadas quando se cria uma nova base de dados. Se o

objetivo for utilizar a base de dados padrão denominada template0, o comando a ser empregado na criação da base de dados teste, é CREATE DATABASE

a) USEDB template0

b) STANDARDDB template0

c) WITH template0

d) HAVING template0

e) teste TEMPLATE template0

44. BANCA: FCC ANO: 2017 ÓRGÃO: DPE-RS PROVA: ANALISTA – BANCO DE DADOS

[53] Alguns sistemas gerenciadores de bancos de dados, como o PostgreSQL versão 8 permitem a definição de parâmetros de armazenamento de tabelas e de índices. Um desses parâmetros é o FILLFACTOR, sendo que no caso de

a) índices B-Tree, seu valor padrão é 50.

b) tabelas, pode variar entre os valores 50 e 100.

c) índices pouco atualizados, o melhor valor é 10.

d) tabelas muito atualizadas seu valor mais indicado é 100.

e) tabelas, seu valor padrão é 70.

45. BANCA: FCC ANO: 2016 ÓRGÃO: TRT - 14ª REGIÃO (RO E AC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

[34] Para conceder, a todos os usuários de um banco de dados PostgreSQL, o privilégio de inserção na tabela Clientes utiliza-se a instrução

a) SET GRANT INSERT ON Clientes ALL;

b) SET ROLE ALL PRIVILEGES TO INSERT ON Clientes;

c) SET GRANT ALL TO Clientes ON INSERT;

d) GRANT INSERT ON Clientes TO ALL GROUP;

e) GRANT INSERT ON Clientes TO PUBLIC;

46. BANCA: FCC ANO: 2016 ÓRGÃO: TRT - 23ª REGIÃO (MT) PROVA: TÉCNICO JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

[33] Um Técnico digitou as instruções abaixo.

No PostgreSQL:

SELECT TRIM (0 FROM 0004872348400) AS "Exemplo";

No Oracle:

SELECT TRIM (0 FROM 0004872348400) "Exemplo" FROM sys.dual;

O valor exibido foi

a) 4872348400

b) 0048432784000

c) 48432784

d) 48723484

e) 00048723484

47. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 5ª REGIÃO (BA) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

No PostgreSQL, a função utilizada para obter informações sobre arquivos é chamada

A pg_header_info

B pg_file_info

C pg_stat_file

D pg_read_file

E pg_file_access.

48. BANCA: FCC ANO: 2013 ÓRGÃO: TRT - 12ª REGIÃO (SC) PROVA: ANALISTA JUDICIÁRIO - TECNOLOGIA DA INFORMAÇÃO

Em PostgreSQL, se uma função de um gatilho (trigger) executar comandos SQL, existe a possibilidade destes comandos também executarem outros gatilhos. Este processo é conhecido como

A cascading triggers.

B recursive triggers.

C sub-triggers.

D indented triggers.

E multi-trigger.

49. BANCA: FCC ANO: 2007 ORGÃO: MPU PROVA: Analista de Informática - Banco de Dados

NÃO é um tipo de junção suportado pelos gerenciadores de banco de dados PostgreSQL:

A INNER OUTER JOIN ON.

B LEFT OUTER JOIN ON.

C RIGHT OUTER JOIN ON.

D NATURAL JOIN.

E CROSS JOIN.

50. BANCA: FCC ANO: 2013 ORGÃO TRT - 12ª Região (SC) PROVA: Técnico Judiciário - Tecnologia da Informação

No banco de dados PostgreSQL, após uma operação de CROSS JOIN entre uma determinada tabela 1 e uma determinada tabela 2, a tabela resultante irá conter

A ambas as linhas das tabelas 1 e 2, porém somente as colunas que possuam correspondência em ambas as tabelas.

B as colunas da tabela 1 que possuam uma correspondência na tabela 2. O número de linhas será determinado pelas colunas que tenham ocorrência em uma das duas tabelas.

C apenas as linhas da tabela 1 e ambas as colunas da tabela 1 e da tabela 2.

D apenas as colunas da tabela 1 e ambas as linhas da tabela 1 e da tabela 2.

E todas as colunas da tabela 1, seguidas por todas as colunas da tabela 2. Caso as tabelas possuam N e M linhas, respectivamente, a tabela resultante irá conter o produto de N e M (N × M) linhas.

51. BANCA: FCC ANO: 2013 ORGÃO: TRT - 12ª Região (SC) PROVA: Técnico Judiciário - Tecnologia da Informação

Em PostgreSQL, uma função permite que o processamento da query seja interrompido por um determinado número de segundos. Este comando é chamado de

A until

B delay

C wait

D pg_sleep

E stop

52. BANCA: FCC ANO: 2013 ORGÃO: TRT - 12ª Região (SC) PROVA: Analista Judiciário - Tecnologia da Informação

Considere o trecho do comando em SQL abaixo.

CREATE USER MAPPING FOR

Este comando é utilizado para a criação de um mapeamento do usuário para

A um trecho de pesquisa específico.

B uma tabela remota.

C um banco de dados do tipo *cluster*.

D uma visualização (VIEW).

E um servidor estrangeiro (*foreign server*).

53. BANCA: FCC ANO: 2012 ORGÃO: MPE-AP – Prova: Analista Ministerial - Tecnologia da Informação

No banco de dados *PostgreSQL*, o comando MOVE é utilizado para reposicionar

A uma tabela em um banco de dados.

B uma linha em uma tabela.

C uma coluna em uma linha.

D o cursor sem trazer dados.

E uma view em uma tabela

54. BANCA: FCC ANO: 2012 ORGÃO: MPE-AP - Analista Ministerial PROVA: Tecnologia da Informação

No banco de dados *PostgreSQL,* o comando utilizado para efetivar a transação corrente é chamado

A END.

B ROLLBACK.

C TRANSFER.

D EFFECTIVE.

E SELECT.

55. BANCA: FCC ANO: 2012 ORGÃO: MPE-AP - Técnico Ministerial PROVA: Informática

Em bancos de dados *PostgreSQL*, o comando *DECLARE* é utilizado para

A criar uma classe de operadores que define como um determinado tipo de dado pode ser usado em um índice.

B criar cursores, que podem ser utilizados para retornar, de cada vez, um pequeno número de linhas em uma consulta.

C criar uma tabela, inicialmente vazia, no banco de dados corrente.

D registrar um novo tipo de dado para uso no banco de dados corrente.

E registrar uma nova linguagem procedural a ser utilizada em consultas ao banco de dados.

56. BANCA: FCC ANO: 2012 ORGÃO: MPE-AP - Técnico Ministerial PROVA: Informática

Quando o nível de isolamento de uma transação em *SQL* no banco de dados *PostgreSQL* é definido como serializável *(Serializable*), o comando *SELECT* enxerga apenas os dados efetivados

A durante a transação, desde que as transações concorrentes tenham feito *COMMIT.*
B por transações simultâneas.
C após o início da transação, desde que as transações simultâneas tenham efetivado as alterações no banco de dados.
D antes de a transação começar.
E durante a transação, desde que as transações concorrentes não tenham feito *COMMIT.*

57. BANCA: FCC ANO: 2012 ORGÃO: MPE-PE - Técnico Ministerial PROVA: Informática

Em *PostgreSQL*, um conjunto de funções e expressões estão disponíveis para a geração de arquivos XML. A função, similar a função *xmlconcat*, que concatena as colunas xml entre linhas de uma tabela é denominada de

A *xmllist.*

B *xmlrowcat.*

C *xmlgrep.*

D *xmlagg.*

E *xmlconcr.*

GABARITO

1. C	2. E	3. A	4. C
5. C	6. A	7. C	8. E
9. D	10. E	11. B	12. C
13. A	14. C	15. A	16. E
17. D	18. B	19. A	20. E
21. E	22. B	23. E	24. A
25. D	26. C	27. D	28. C
29. E E	30. C	31. E	32. E
33. C	34. E	35. B	36. D
37. A	38. E	39. A	40. B
41. E	42. C	43. E	44. B
45. E	46. D	47. C	48. A
49. A	50. E	51. D	52. E
53. D	54. A	55. B	56. D
57. D			

MongoDB

O MongoDB é um SGBD No-SQL **open-source e orientado a documentos**. Alguns de seus diferenciais são:

> (1) Alto desempenho: documentos embutidos e índices atuando sobre eles.
> (2) Rica linguagem de consulta: permite operações CRUD, agregações de dados, busca por texto e consultas geoespaciais.
> (3) Alta disponibilidade: replica set.
> (4) Escalabilidade horizontal: *sharding*.

O MongoDB é escrito em C ++ e desenvolvido ativamente pela MongoDB, Inc. O projeto é compilado para todos os principais sistemas operacionais, incluindo o Mac OS X, o Windows, o Solaris e a maioria dos tipos de Linux. Binários pré-compilados estão disponíveis para cada uma dessas plataformas em http://mongodb.org . O MongoDB é licenciado sob a Licença Pública Geral GNU-Affero (AGPL). O código-fonte está disponível gratuitamente no GitHub e as contribuições da comunidade são frequentemente aceitas. Mas o projeto é guiado pela equipe de servidores centrais do MongoDB, Inc., e a esmagadora maioria dos *commits* vem desse grupo.

O MongoDB é um banco de dados de propósito geral poderoso, flexível e escalável. Ele combina a capacidade de ser escalável com recursos como índices secundários, consultas de intervalo, classificação, agregações e índices geoespaciais. Nesta aula mostraremos as principais características do MongoDB.

Documentos são o conceito chave para o MongoDB. O banco de dados armazena e recupera documentos, os quais podem ser XML, JSON, BSON, entre outros. Esses documentos são estruturas de dados descritas na forma de árvores hierárquicas auto descritivas, constituídas de mapas, coleções e valores escalares.

Vejamos uma comparação rápida entre a terminologia usada no MongoDB e no Oracle:

MongoDB	Oracle
Instância MongoDB	Instância de banco de dados
Banco de dados	Esquema
Coleção	Tabela
Documento	Linha
_id	Rowid

DBRef	Junção

O campo _id é um campo especial encontrado em todos os documentos no Mongo, assim como o ROWID no Oracle. O valor deste atributo pode ser atribuído pelo usuário, desde que seja único.

Vamos agora entender um aspecto importante do MongoDB: JSON x BSON.

Entendendo o Banco de dados de documentos

Um banco de dados é definido em grande parte por seu modelo de dados. Nesta seção, você aprenderá sobre o modelo de dados orientado a documento e, em seguida, verá os recursos do MongoDB que permitem que você opere efetivamente nesse modelo. Se você não estiver familiarizado com documentos no contexto de bancos de dados, o conceito pode ser mais facilmente apresentado por um exemplo, para tal observe o quadro abaixo.

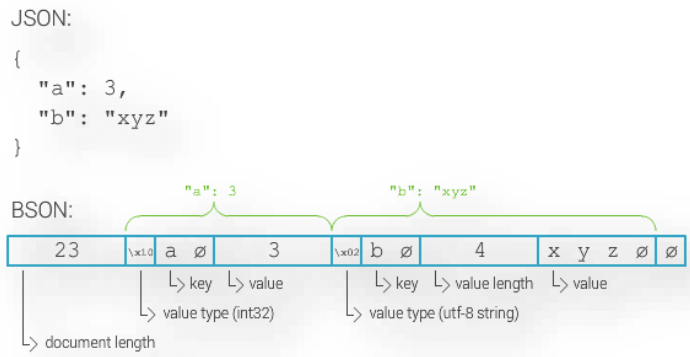
O *JavaScript Object Notation* (JSON) é um padrão aberto, humano e legível por máquina que facilita o intercâmbio de dados e, juntamente com o XML, é o principal formato de intercâmbio de dados usado na Web moderna. O JSON é compatível com todos os tipos de dados básicos que você espera: números, sequências de caracteres e valores booleanos, bem como matrizes e hashes.

O formato de documento do MongoDB é baseado no JSON, um esquema popular para armazenar estruturas de dados arbitrárias. As estruturas JSON consistem em chaves e valores que podem ser aninhadas arbitrariamente em diferentes profundidades. Bancos de dados de documentos, como o MongoDB, usam documentos JSON para armazenar registros, assim como tabelas e linhas armazenam registros em um banco de dados relacional. Aqui está um exemplo de um documento JSON:

```
{
   '_id' : 1,
   'name' : { 'first' : 'John', 'last' : 'Backus' },
   'contribs' : [ 'Fortran', 'ALGOL', 'Backus-Naur Form', 'FP' ],
   'awards' : [
      {
         'award' : 'W.W. McDowell Award',
         'year' : 1967,
         'by' : 'IEEE Computer Society'
      }, {
         'award' : 'Draper Prize',
         'year' : 1993,
         'by' : 'National Academy of Engineering'
      }
   ]
}
```

Um documento JSON precisa de aspas simples em todos os valores, exceto para valores numéricos, como podemos observar na figura acima. Um banco de dados que retorna um JSON como resultado de consulta permite que os dados sejam facilmente analisados, com pouca ou nenhuma transformação, diretamente pelo JavaScript e pelas linguagens de programação mais populares, reduzindo a quantidade de lógica necessária para criar sua camada de aplicativo.

O MongoDB representa documentos JSON em formato codificado binário chamado BSON nos bastidores. O BSON estende o modelo JSON para fornecer tipos de dados adicionais, campos ordenados e ser eficiente para codificação e decodificação em diferentes idiomas.



Ao contrário de outros bancos de dados que armazenam dados JSON como strings e números simples, a codificação BSON estende a representação JSON para incluir tipos adicionais, como int, long, date, ponto flutuante e decimal (128) - o último é especialmente importante para alta precisão, sem perdas financeiras e cálculos científicos. Isso torna muito mais fácil para os aplicativos processar, classificar e comparar dados de forma confiável. Os documentos BSON contêm um ou mais campos e cada campo contém um valor de um tipo de dados específico, incluindo matrizes, dados binários e subdocumentos.

Você já tem o conhecimento a respeito das diferenças entre JSON e BSON. Neste momento vamos voltar nossa atenção para as características da arquitetura do MongoDB. Espero que você esteja acompanhando nossa aula.

Principais Características do MongoDB

O MongoDB é um banco de dados ***orientado a documentos***, não um banco de dados relacional. A principal razão para se afastar do modelo relacional é **facilitar o dimensionamento**, mas também há outras vantagens. Um banco de dados orientado a documentos **substitui o conceito de "linha" por um modelo mais flexível, o "documento"**.

Ao permitir documentos e matrizes incorporados, a abordagem orientada a documentos possibilita **representar relações hierárquicas complexas com um único registro**. Isso se encaixa naturalmente no modo como os desenvolvedores em linguagens modernas orientadas a objetos pensam sobre seus dados.

Também **não há esquemas predefinidos**: as chaves e valores de um documento não são de tipos ou tamanhos fixos. Sem um esquema fixo, adicionar ou remover campos conforme a demanda fica mais fácil. Geralmente, isso torna o desenvolvimento mais rápido pois os desenvolvedores podem fazer iterações rapidamente. Assim, os desenvolvedores podem tentar dezenas de modelos para os dados e escolher o melhor.

Os desenvolvedores podem ainda começar **a escrever códigos e persistir objetos à medida que são criados**. Utilizando o MongoDB, quando eles precisam adicionar mais recursos, o MongoDB continua armazenando os objetos atualizados sem a necessidade de executar operações ALTER TABLE dispendiosas - ou pior, sem ter que reprojetar o esquema de dados do zero.

RICA LINGUAGEM DE CONSULTA

Foge do nosso objetivo apresentar detalhes da linguagem para manipulação e definição de dados do MongoBD. Caso você tenha curiosidade, sugiro que acesse o link[1]. Abaixo apresentamos algumas ações que podem ser realizadas com a linguagem de conuslta e que demonstram o poder da linguagem associada ao MongoDB. É possível realizar algumas ações, descritas nos retângulos, em cada um dos contextos apresentados.

Consultas expressivas

- Encontrar alguém com o telefone # “1-212…”
- Verificar se a pessoa com o número “555…” está na lista “não ligar”

Geoespacial

- Encontrar a melhor oferta para o cliente nas coordenadas geográficas da 42nd St. e 6th Ave

Pesquisa de texto

- Encontrar todos os tweets que mencionam a empresa nos últimos dois dias

Navegação facetada

- Filtrar resultados para mostrar apenas produtos cujos preços seja menores que cinquenta dólares (<$50), de tamanho grande e fabricados pela ExampleCo

Agregação

- Contar e classificar o número de clientes por cidade, calcular o gasto mínimo, máximo e médio

Suporte JSON Binário Nativo

- Adicionar um número de telefone adicional ao registro de Ricardo Vale sem reescrever o documento no cliente
- Atualizar apenas 2 números de telefone de 10

[1] http://wiki.icmc.usp.br/images/3/30/SCC0542012017mongoDB.pdf

• Ordenar de acordo com a data de modificação

Operações de matriz de granulação fina

• No conjunto de pontuações dos testes de Thiago Cavalcanti, atualizar cada pontuação <70 para 0.

JOIN ($ lookup)

• Consultar todas as residências de San Francisco, pesquisar suas transações e somar a quantia por pessoa.

Consultas de gráfico ($ graphLookup)

• Consultar todas as pessoas dentro de 3 graus de separação do Marcos em um grafo.

TRANSAÇÕES

Como os documentos podem reunir dados relacionados que seriam modelados em tabelas pai-filho separadas em um esquema tabular, **as operações atômicas de documento único do MongoDB fornecem semânticas de transação que atendem às necessidades de integridade de dados da maioria dos aplicativos**. Um ou mais campos podem ser gravados em uma única operação, incluindo atualizações para vários subdocumentos e elementos de uma matriz. As garantias fornecidas pelo MongoDB **garantem isolamento completo** à medida que um documento é atualizado. Qualquer erro faz com que a operação seja revertida para que os clientes recebam uma visão consistente do documento.

O **MongoDB 4.0** adiciona **suporte a transações ACID de vários documentos**, tornando ainda mais fácil para os desenvolvedores abordarem uma gama completa de casos de uso com o MongoDB. Eles podem agir como os desenvolvedores de transações estão familiarizados com bancos de dados relacionais - multi-instruções, sintaxe semelhante e fácil de adicionar a qualquer aplicativo. Por meio do isolamento de snapshot, as transações fornecem uma visualização consistente dos dados, reforçam a execução de tudo ou nada e não afetam o desempenho das cargas de trabalho que não as exigem. Saiba mais visitando a página de transações de documentos múltiplos do MongoDB, onde você pode ver exemplos de códigos, documentação, giz e palestras dos engenheiros que os implementaram.

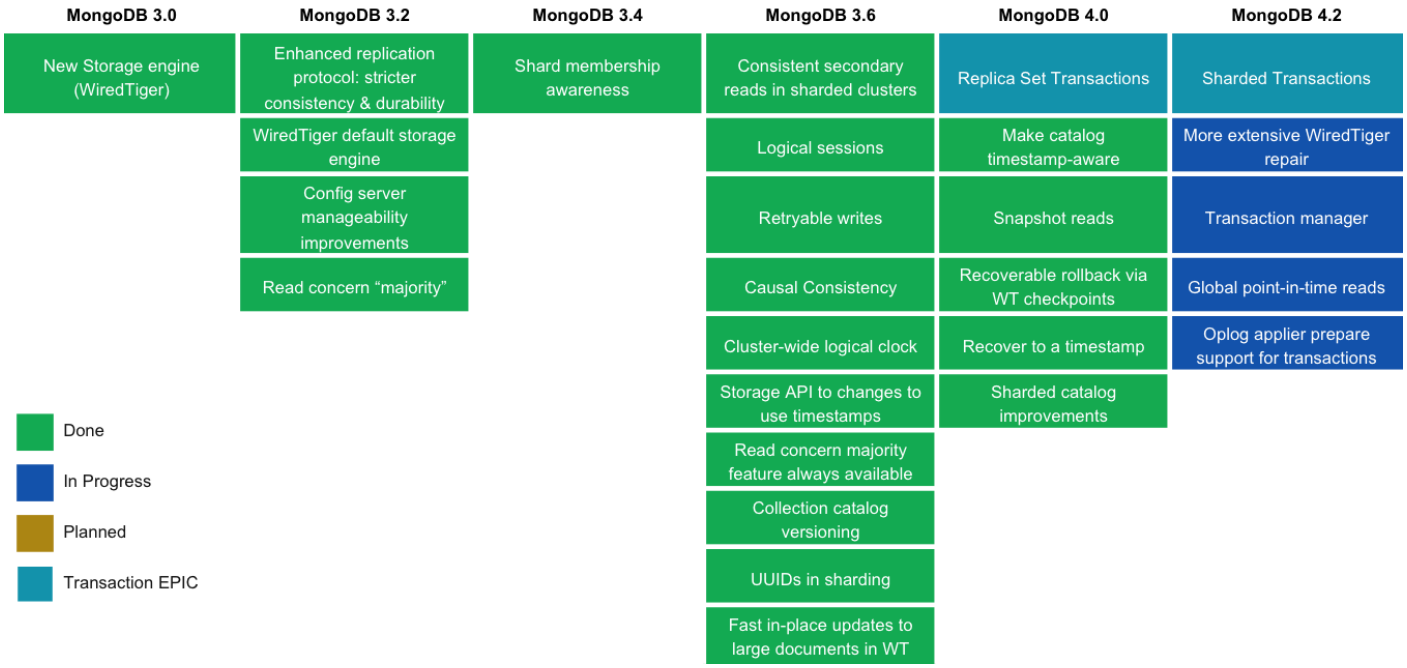
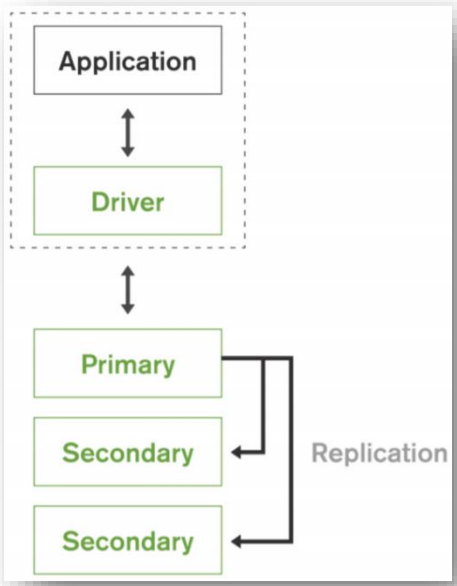


Figura 1 - O caminho para as transações - Junho de 2018

DISPONIBILIDADE



O MongoDB mantém **várias cópias de dados usando conjuntos de réplicas**. Ao contrário dos bancos de dados relacionais, os conjuntos de réplicas são recuperados automaticamente, pois o *failover* e a recuperação são totalmente automatizados. Portanto, não é necessário intervir manualmente para restaurar um sistema em caso de falha ou para adicionar estruturas e agentes de cluster adicionais. Conjuntos de réplicas também fornecem flexibilidade operacional, fornecendo uma maneira de executar a manutenção de sistemas (ou seja, atualizar hardware e software subjacente) usando reinicializações de réplica "deslizante" que preservam a continuidade do serviço.

Um conjunto de réplicas consiste em várias réplicas de banco de dados. Para manter a consistência de dados forte, um membro assume a **função da réplica primária** na qual todas as operações de gravação são aplicadas (o MongoDB fragmenta automaticamente o conjunto de dados em vários nós para dimensionar operações de gravação). Os outros membros do conjunto de réplicas agem como **secundários**, replicando todas as alterações de dados do *oplog* (log de operações). O oplog contém um conjunto ordenado de operações que serão replicadas nos secundários.

Se o membro do conjunto de réplicas primário sofrer uma falha (por exemplo, falta de energia, falha de hardware, partição de rede), um dos membros secundários será

automaticamente eleito para primário, geralmente dentro de alguns segundos, e as conexões do cliente serão automaticamente recuperadas (*failover*) para esse novo primário. Qualquer gravação que não consiga ser atendida durante a eleição pode ser submetida novamente de forma automática assim que um novo nó primário for estabelecido.

O processo de eleição do conjunto de réplicas é controlado por algoritmos sofisticados baseados em uma implementação estendida do *Raft consensus protocol*. Isso não só permite um *failover* rápido para maximizar a disponibilidade do serviço, como garante que apenas os membros secundários mais adequados sejam avaliados para eleição para primária e reduz o risco de *failovers* desnecessários (também conhecidos como "falsos positivos"). Antes de uma réplica secundária ser promovida, os algoritmos de eleição avaliam uma série de parâmetros, incluindo:

• Análise de identificadores de eleição, *timestamps* e *journal persistence* para identificar os membros do conjunto de réplicas que aplicaram as atualizações mais recentes do membro primário.
• Status de pulsação (heartbeat) e conectividade com a maioria dos outros membros do conjunto de réplicas.
• Prioridades definidas pelo usuário atribuídas aos membros do conjunto de réplicas. Por exemplo, os administradores podem configurar todas as réplicas localizadas em uma região remota para serem candidatas à eleição somente se toda a região primária falhar.

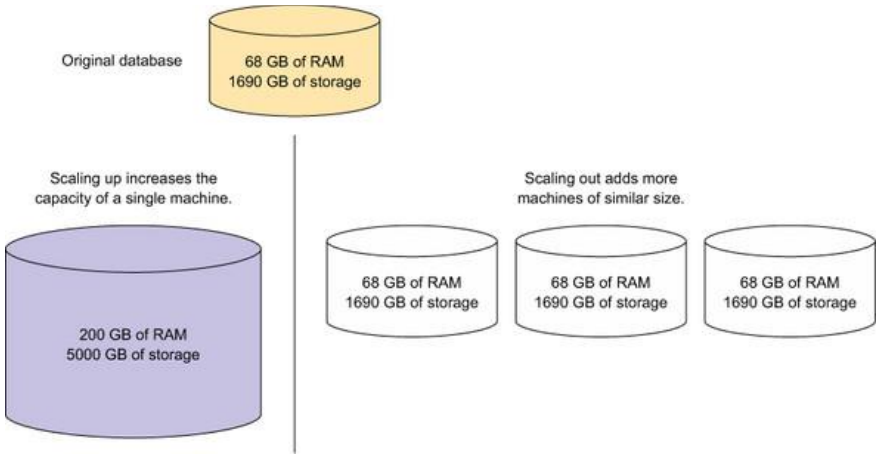
Uma vez que o processo de eleição tenha determinado o novo nó primário, os membros secundários começam a replicar automaticamente a partir dele. Quando a réplica primária original volta a ficar on-line, ela reconhece sua mudança de estado e automaticamente assume a função de secundária, aplicando todas as operações de gravação que ocorreram enquanto estava inativa.

ESCALABILIDADE

A maneira mais fácil de escalar a maioria dos bancos de dados é atualizar o hardware. Se seu aplicativo estiver sendo executado em um único nó, geralmente é possível adicionar uma combinação de discos mais rápidos, mais memória e uma CPU mais robusta para remover os gargalos do banco de dados. A técnica de aumentar o hardware de um único nó para redimensionamento é conhecida como *escalonamento vertical*. O redimensionamento vertical tem as vantagens de ser simples, confiável e econômico (até certo ponto), pois você chega em uma situação na qual não é mais possível migrar para uma máquina melhor.

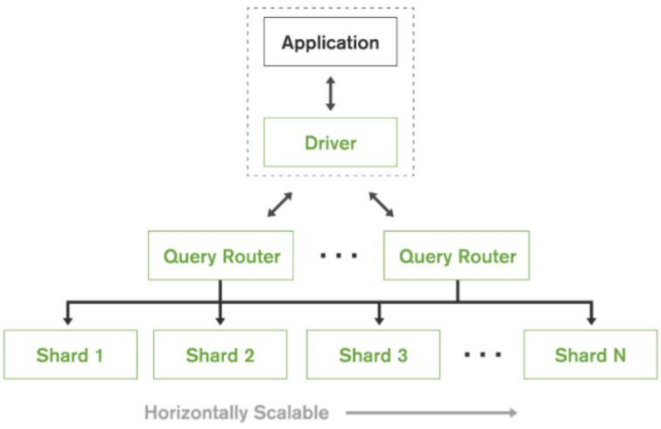
Então, faz sentido considerar o redimensionamento *horizontal* ou *scale out* (veja a figura abaixo). Em vez de reforçar um único nó, escalar horizontalmente significa **distribuir o banco de dados em várias máquinas**. Uma arquitetura desta natureza pode ser executada em muitas máquinas menores e mais baratas, geralmente reduzindo seus custos. Além disso, a distribuição de dados entre as máquinas atenua as consequências de uma falha.

As máquinas inevitavelmente falharão de tempos em tempos. Se você escalou verticalmente e a máquina falha, então você precisa lidar com a falha de uma máquina da qual depende a maior parte do seu sistema. Isso pode não ser um problema se existir uma cópia dos dados em um servidor escravo replicado, mas ainda assim é interessante que um servidor não consiga derrubar todo o sistema. Compare isso com falha dentro de uma arquitetura dimensionada horizontalmente.



O MongoDB foi projetado para tornar o escalonamento horizontal gerenciável. Ele faz isso por meio de **um mecanismo de particionamento baseado em intervalo**, conhecido como **sharding**, que gerencia automaticamente a distribuição de dados entre nós. Há também um mecanismo de sharding baseado em hash e tag, mas é apenas outra forma de implementar sharding baseado em intervalo.

O sistema de sharding manipula a adição de nós fragmentados e facilita o failover automático. Os fragmentos individuais são compostos de um conjunto de réplicas cujos dados existem em pelo menos dois nós, garantindo a recuperação automática sem nenhum ponto de falha. Isso significa que nenhum código de aplicativo precisa lidar com essas logísticas de processamento; seu código de aplicativo se comunica com um cluster sharded. O cluster, para a aplicação, pode ser visto como um servidor de apenas um nó.



O sharding é transparente para os aplicativos. **Se há um ou mil shards, o código do aplicativo para consultar o MongoDB permanece o mesmo**. Os aplicativos emitem consultas para **um roteador de consulta** que envia a consulta para os shards apropriados. Para consultas de chave-valor baseadas na chave do fragmento (shard key), o roteador de consulta enviará a consulta ao shard que gerencia o documento com a chave solicitada.

Ao usar o sharding baseado em intervalo, as consultas que especificam intervalos na chave de shard são despachadas apenas para shards que contêm documentos com valores dentro do intervalo. Para consultas que não usam a chave de fragmento, o roteador de consulta

transmitirá a consulta para todos os fragmentos, agregando e classificando os resultados conforme apropriado. Vários roteadores de consulta podem ser usados em um cluster MongoDB, onde o número apropriado é estabelecido pelos requisitos de desempenho e disponibilidade do aplicativo.

Muitos bancos de dados distribuídos aleatoriamente pulverizam dados em nós de um cluster, impondo penalidades de desempenho quando os dados são consultados ou adicionando complexidade de aplicativo quando os dados precisam ser localizados para nós específicos. Ao expor várias políticas de sharding aos desenvolvedores, o MongoDB oferece uma abordagem melhor. Os dados podem ser distribuídos de acordo com padrões de consulta ou requisitos de posicionamento de dados, oferecendo aos desenvolvedores uma escalabilidade muito maior em um conjunto diversificado de cargas de trabalho:

• **Ranged Sharding**. Os documentos são particionados em shards de acordo com o valor da chave de shard. É provável que os documentos com valores de chave de fragmento próximos um do outro sejam colocados no mesmo fragmento. Essa abordagem é bem adaptada para aplicativos que precisam **otimizar consultas baseadas em intervalos**, como a agrupar os dados de todos os clientes de uma região específica em um fragmento específico.

• **Fragmentação com Hash**. Os documentos são distribuídos de acordo com um hash MD5 do valor da chave de fragmento. Essa abordagem garante uma distribuição uniforme de gravações em shards, o que geralmente é ideal para **ingerir fluxos de séries temporais e dados de eventos**.

• **Fragmentação por zonas**. Fornece a capacidade dos desenvolvedores definirem regras específicas que controlam o posicionamento de dados em um cluster fragmentado.

SEGURANÇA DOS DADOS

Ter a liberdade de colocar os dados onde são necessários permite que os desenvolvedores criem novas e poderosas classes de aplicativos. No entanto, eles também devem ter certeza de que seus dados estão seguros, onde quer que estejam armazenados. Em vez de criar controles de segurança no aplicativo, eles devem poder **confiar no banco de dados para implementar os mecanismos necessários para proteger dados confidenciais e atender às necessidades** dos aplicativos em setores regulamentados. O MongoDB possui recursos para defender, detectar e controlar o acesso aos dados:

• **Autenticação**. Simplificando o controle de acesso ao banco de dados, o MongoDB oferece integração com mecanismos de segurança externos, incluindo certificados LDAP, Windows Active Directory, Kerberos e x.509. Além disso, a lista de permissões de IP permite que as equipes de DevOps configurem o MongoDB para aceitar apenas conexões de endereços IP aprovados.

• **Autorização**. Controles de acesso baseados em função (RBAC) permitem que as equipes de DevOps configurem permissões para um usuário ou um aplicativo com base nos privilégios necessários para executar suas tarefas. Eles podem ser definidos no MongoDB ou em um servidor LDAP. Além disso, os desenvolvedores podem definir visões que

exponham apenas um subconjunto de dados de uma coleção, por exemplo, uma visão que filtra ou mascara campos específicos, como Informações de identificação pessoal de dados de clientes. As *views* também podem ser criadas para expor apenas dados agregados.

• **Auditoria.** Para conformidade regulatória, os administradores de segurança podem usar **o log de auditoria nativo do MongoDB** para rastrear quaisquer operações de banco de dados - seja DML ou DDL.

• **Criptografia**. Os dados do MongoDB podem **ser criptografados na rede, no disco e nos backups**. Com o mecanismo de armazenamento criptografado, a proteção de dados armazenados é um recurso integrando ao banco de dados. Ao criptografar nativamente arquivos de banco de dados no disco, os desenvolvedores eliminam a sobrecarga de gerenciamento e desempenho dos mecanismos externos de criptografia. Somente os funcionários que possuem as credenciais de autorização de banco de dados apropriadas podem acessar os dados criptografados, fornecendo níveis adicionais de defesa.

Outros Recursos

O MongoDB é um banco de dados de propósito geral, portanto, além de criar, ler, atualizar e excluir dados, ele fornece a maioria dos recursos que você esperaria de um SGBD e muitos outros que o diferenciam:

Indexação

O MongoDB suporta **índices secundários genéricos** e fornece recursos de indexação exclusivos, compostos, geoespaciais e de texto completo (full-text). Índices secundários em estruturas hierárquicas, como documentos e *arrays* aninhados, também são suportados e permitem que os desenvolvedores aproveitem ao máximo a capacidade de desenhar índices que melhor atendam às suas aplicações.

Agregação

O MongoDB fornece uma estrutura de agregação baseada no conceito de pipelines de processamento de dados. Os pipelines de agregação permitem que você construa mecanismos analíticos complexos processando dados por meio de uma série de estágios relativamente simples no lado do servidor e com a vantagem total das otimizações do banco de dados.

Tipos especiais de coleção e índice

O MongoDB suporta coleções TTL (time-to-live) para dados que devem expirar em um determinado momento, como sessões e fixed-size (capped) collections, para armazenar dados recentes, como logs. O MongoDB também suporta **índices parciais** limitados aos documentos que correspondem a um filtro de critérios para aumentar a eficiência e reduzir a quantidade de espaço de armazenamento necessário.

Armazenamento de arquivo

O MongoDB suporta um protocolo fácil de usar para armazenar arquivos grandes e metadados de arquivos.

Alguns recursos comuns aos bancos de dados relacionais não estão presentes no MongoDB, principalmente transações complexas em múltiplas linhas. O MongoDB também suporta joins apenas de uma forma muito limitada através do uso do operador de agregação **$lookup** introduzido na versão 3.2. O tratamento do MongoDB para transações e junções foram decisões de arquitetura para permitir maior escalabilidade, porque esses dois recursos são difíceis de fornecer com eficiência em um sistema distribuído.

É verdade que as novas releases já estão avançando em termos de transações, conforme visto anteriormente em nossa aula. Para saber mais sobre o estado das transações no MongoDB acesse o link[2].

[2] https://www.mongodb.com/blog/post/mongodb-multi-document-acid-transactions-general-availability

ESSA LEI TODO MUNDO CONHECE:
PIRATARIA É CRIME.

Mas é sempre bom revisar o porquê e como você pode ser prejudicado com essa prática.

1. 

Professor investe seu tempo para elaborar os cursos e o site os coloca à venda.

2. 

Pirata divulga ilicitamente (grupos de rateio), utilizando-se do anonimato, nomes falsos ou laranjas (geralmente o pirata se anuncia como formador de "grupos solidários" de rateio que não visam lucro).

3. 

Pirata cria alunos fake praticando falsidade ideológica, comprando cursos do site em nome de pessoas aleatórias (usando nome, CPF, endereço e telefone de terceiros sem autorização).

4. 

Pirata compra, muitas vezes, clonando cartões de crédito (por vezes o sistema anti-fraude não consegue identificar o golpe a tempo).

5. 

Pirata fere os Termos de Uso, adultera as aulas e retira a identificação dos arquivos PDF (justamente porque a atividade é ilegal e ele não quer que seus fakes sejam identificados).

6. 

Pirata revende as aulas protegidas por direitos autorais, praticando concorrência desleal e em flagrante desrespeito à Lei de Direitos Autorais (Lei 9.610/98).

7. 

Concurseiro(a) desinformado participa de rateio, achando que nada disso está acontecendo e esperando se tornar servidor público para exigir o cumprimento das leis.

8. 

O professor que elaborou o curso não ganha nada, o site não recebe nada, e a pessoa que praticou todos os ilícitos anteriores (pirata) fica com o lucro.

Deixando de lado esse mar de sujeira, aproveitamos para agradecer a todos que adquirem os cursos honestamente e permitem que o site continue existindo.