

Banco de Dados - PostgreSQL

Prof. Washington Almeida, MSc, ISF 27002

APRESENTAÇÃO

- O **PostgreSQL** é um sistema de gerenciamento de banco de dados relacional de objetos (SGBDRO) baseado no POSTGRES, desenvolvido na Universidade da Califórnia no Departamento de Ciência da Computação de Berkeley.
- De acordo com o ranking da DB-Engines (<https://db-engines.com/en/ranking>) é o 4.º SGBD mais utilizado do mundo.



Nov 2019	Rank		DBMS
	Oct 2019	Nov 2018	
1.	1.	1.	Oracle +
2.	2.	2.	MySQL +
3.	3.	3.	Microsoft SQL Server +
4.	4.	4.	PostgreSQL +
5.	5.	5.	MongoDB +

APRESENTAÇÃO

- O POSTGRES foi pioneiro em muitos conceitos que só ficaram disponíveis em alguns sistemas comerciais de banco de dados muito mais tarde.
- O **PostgreSQL** é um descendente de código aberto do código original do POSTGRES de Berkeley.
- Suporta grande parte do padrão SQL e oferece muitos recursos modernos: Querys (consultas) complexas; Chaves estrangeiras; Triggers (gatilhos); Views (visões) atualizáveis; Integridade transacional; Multiversion Concurrency Control (controle de concorrência de multiversão).

HISTÓRIA

- O projeto POSTGRES, liderado pelo professor Michael Stonebraker, foi patrocinado pela Defense Advanced Research Projects Agency (DARPA), pelo Army Research Office (ARO), pela National Science Foundation (NSF) e pela ESL, Inc.
- O POSTGRES foi utilizado para implementar diferentes aplicações de pesquisa e produção.
- Em 1994, Andrew Yu e Jolly Chen, lançaram um novo nome ao projeto, Postgres95, adicionando um interpretador de linguagem SQL ao POSTGRES.
- Em 1996, um novo nome foi escolhido, **PostgreSQL**, refletindo o relacionamento entre o POSTGRES e a linguagem SQL.

HISTÓRIA

- Muitas pessoas continuam a se referir ao **PostgreSQL** como "Postgres" devido a tradição ou porque é mais fácil de pronunciar.
- Por causa da licença liberal, o **PostgreSQL** pode ser usado, modificado e distribuído gratuitamente por qualquer pessoa, para qualquer fim, seja ele privado, comercial ou acadêmico.
- Como o **PostgreSQL** é um projeto de código aberto, depende da comunidade de usuários para suporte contínuo.

LICENÇA

- O **PostgreSQL** é lançado sob a Licença **PostgreSQL**, uma licença liberal de código-fonte aberto, semelhante às licenças BSD ou MIT.
- As pessoas frequentemente perguntam por que o **PostgreSQL** não é liberado sob a **GNU General Public License**. A resposta simples é: "gostamos da nossa licença e não queremos alterá-la".
- No momento, não há planos de alterar a licença do **PostgreSQL** ou liberar o **PostgreSQL** sob uma licença diferente.

CARACTERÍSTICAS

- Compatível com várias plataformas, usando todos os principais idiomas e middleware disponíveis.
- Oferece um dos mais sofisticados mecanismos de travamento/bloqueio.
- Suporte para Multiversion Concurrency Control (controle de concorrência de multiversão).
- Write-Ahead Logging (WAL) para garantir a integridade dos dados e logging de transações.
- Suporta múltiplos tipos de índices (B-Tree, rTree e Hash) permitindo a escolha do índice mais eficiente para cada aplicação.
- Constraints/Foreign Keys (restrições/chaves estrangeiras).

CARACTERÍSTICAS

- Linguagem Procedural em várias linguagens (PL/pgSQL, PL/Python, PL/Java, PL/Perl) para procedimentos armazenados.
- Integridade transacional.
- Facilidade de acesso.
- Excelente escalabilidade vertical.
- Compatível com o padrão ANSI SQL.
- Suporte completo à arquitetura de rede cliente-servidor.
- Replicação Master-Slave (mestre-escravo) e High Availability (alta disponibilidade).

CARACTERÍSTICAS

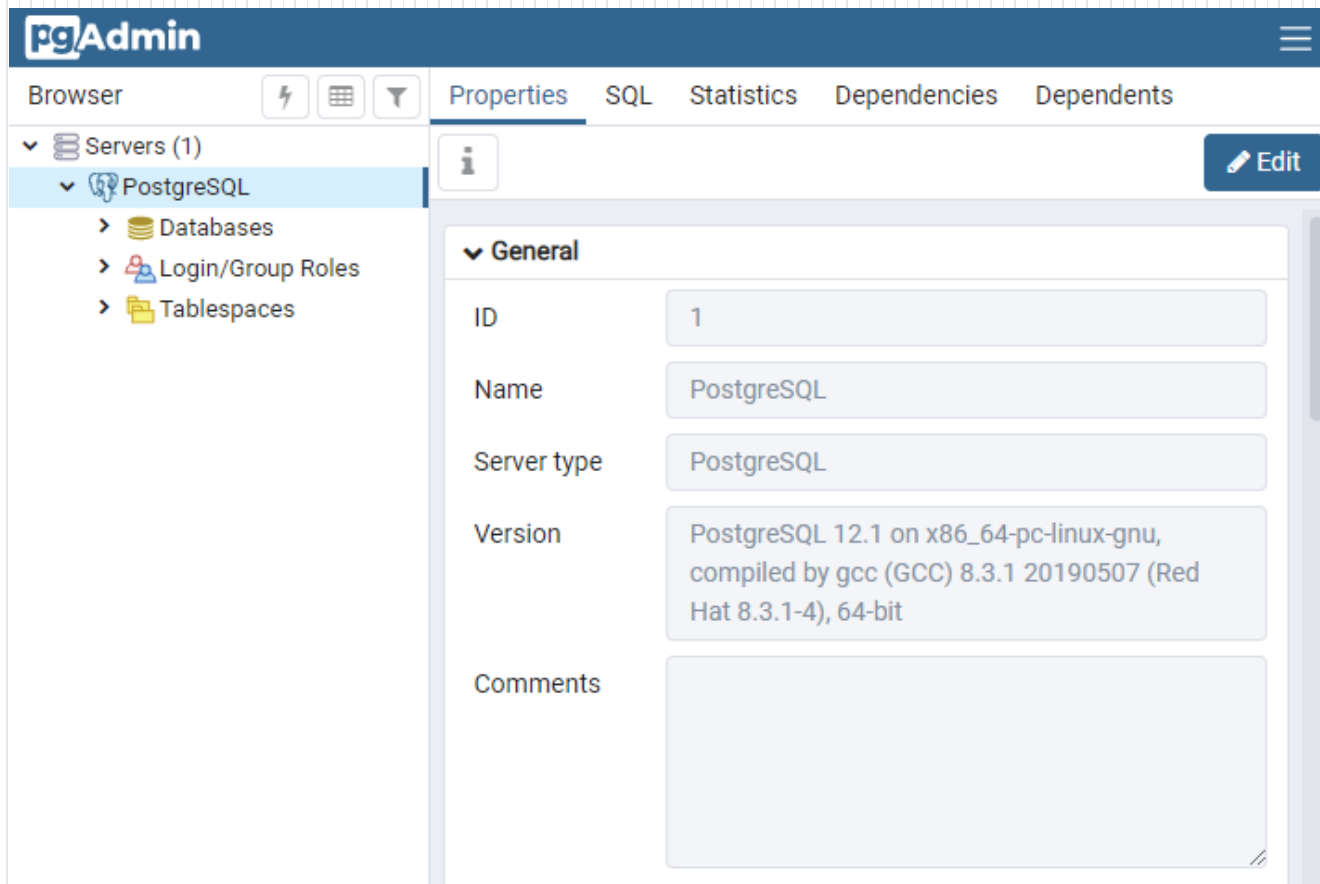
- **IMPORTANTE:**
 - No mecanismo de **MVCC (controle de concorrência de multiversão)**, processos de leitura não bloqueiam processos de escrita e vice-versa, reduzindo drasticamente (às vezes, eliminando) a contenção entre transações concorrentes e paralisação parcial ou completa (deadlock);
 - Os mecanismos de bloqueio são uma maneira dos bancos de dados produzirem saída de dados sequenciais sem as etapas sequenciais. Os bloqueios fornecem um método para proteger os dados que estão sendo usados, para que não ocorram anomalias como dados perdidos ou dados adicionais que podem ser adicionados devido à perda de uma transação.

INSTALAÇÃO

- Existem 3 formas de realizar a instalação do **PostgreSQL**:
 - Instalando a partir do gerenciador de pacotes da distribuição GNU/Linux que está sendo utilizado, como o **yum** do CentOS.
 - Instalando a partir dos repositórios oficiais.
 - Instalando a partir do código fonte.
- As duas primeiras formas são as mais fáceis de instalar.
- Como uma desvantagem, é provável que a versão presente na distribuição não seja a última versão, como alternativa seria instalar a partir do pacote oficial.
- A terceira forma é a mais radical e flexível, onde é necessário compilar todos os binários e pacotes a partir do código fonte, além de decidir em detalhes como será a instalação.

INSTALAÇÃO

- Para auxiliar na gerência do **PostgreSQL** é possível instalar a ferramenta de interface gráfica **PGAdmin**.



INSTALAÇÃO

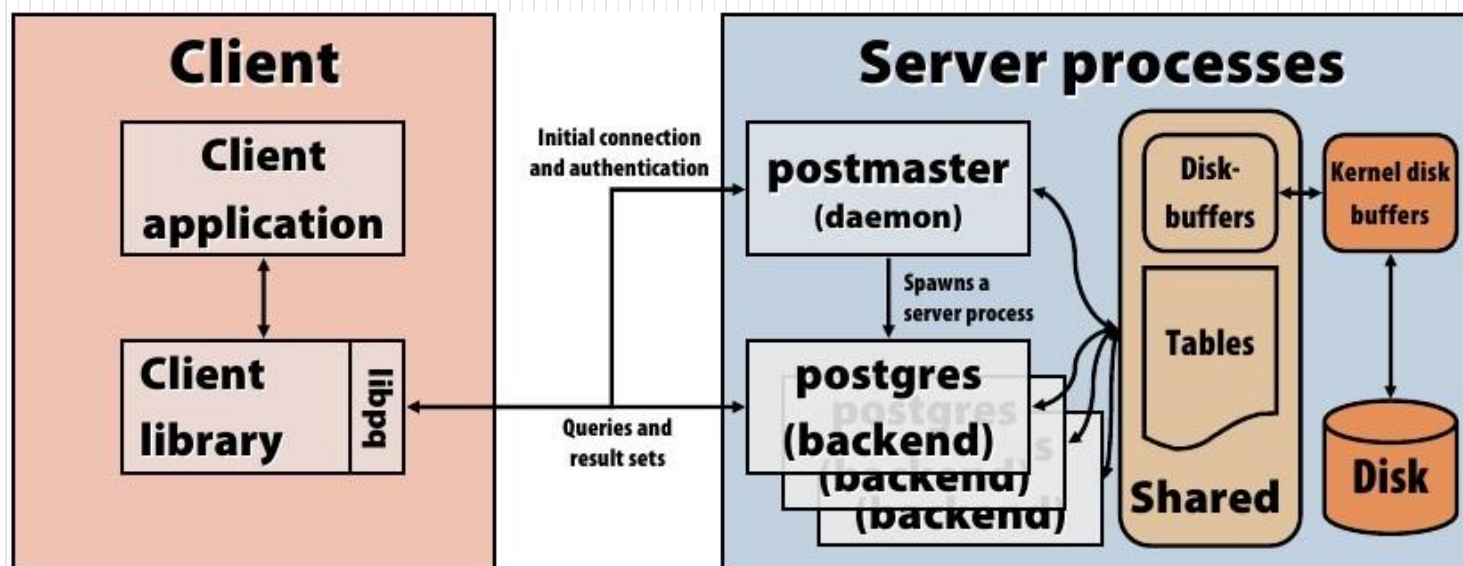
- A porta **TCP** padrão para o **PostgreSQL** é geralmente **5432**, no entanto, isso pode ser facilmente alterado.
- O **PostgreSQL** pode funcionar com essas arquiteturas de CPU: x86, x86_64, IA64, PowerPC, PowerPC 64, S / 390, S / 390x, Sparc, Sparc 64, ARM, MIPS, MIPSEL e PA-RISC.
- O **PostgreSQL** pode funcionar nesses sistemas operacionais: Linux (todas as distribuições recentes), Windows (Win2000 SP4 e posterior), FreeBSD, OpenBSD, NetBSD, macOS, AIX, HP / UX e Solaris.

ARQUITETURA

- O **PostgreSQL** usa um modelo de cliente/servidor.
- Uma sessão do **PostgreSQL** consiste nos seguintes processos/programas:
 - Um processo de servidor, que será responsável por gerenciar os arquivos do banco de dados, as conexões das aplicações clientes ao banco de dados e executar ações no banco de dados em nome das aplicações clientes. O processo do servidor de banco de dados é chamado de **postgres**.
 - A aplicação cliente que deseja executar operações no banco de dados. Essas aplicações possuem natureza muito diversas, pode ser um aplicativo gráfico, um servidor web que acessa o banco de dados para exibir páginas web ou uma ferramenta especializada em manutenção de banco de dados.
- Tipicamente de uma arquitetura cliente/servidor, o cliente e o servidor podem estar em hosts diferentes, se comunicando através de uma rede TCP/IP.

ARQUITETURA

- O servidor **PostgreSQL** pode lidar com várias conexões simultâneas de clientes, onde um novo processo é iniciado para cada conexão.
- O processo **postgres** do servidor de banco de dados sempre está em execução aguardando conexões de clientes.
- Os processos de comunicação entre o cliente e o servidor vêm e vão.



NOMEAÇÃO

- Ao criar um objeto, como um **database**, uma **coluna** ou uma **tabela**, um nome deve ser atribuído.
- O PostgreSQL usa um tipo de dado único para definir as regras dos nomes desses objetos: o tipo **name**.
- Um valor tipo **name** deve possuir uma sequência de 63 caracteres ou menos.
- Deve começar uma letra, ou um sublinhado (_), e o restante da string pode conter letras, dígitos e sublinhados.
- Pode ser inserido mais de 63 caracteres para um nome de um objeto, mas o **PostgreSQL** só irá armazenar os primeiros 63 caracteres, sendo o restante descartado.

Regras de Nomeação

- O SQL e o **PostgreSQL** reservam algumas palavras, e, normalmente, não é possível utilizá-las para nomear objetos. Exemplos de palavras reservadas:
- Não é possível criar uma **tabela** chamada INTEGER ou uma coluna chamada BETWEEN.
- Parar criar um objeto que não atenda a essas regras, caso necessário, pode colocar entre aspas duplas. Por exemplo: "**3.14159**".
- Ao criar um objeto onde foi incluído aspas duplas no nome, é necessário usar aspas duplas também quando for se referir a esse objeto. Por exemplo:

```
SELECT filling, topping, crust FROM "3.14159";
```

1	ANALYZE
2	BETWEEN
3	CHARACTER
4	INTEGER
5	CREATE

NOMEAÇÃO

- Os nomes criados com aspas são case-sensitive. Por exemplo: **"1020Home"** e **"1020HOME"**.
- Nomes criados sem aspas, todos os seus caracteres alfabéticos são convertidos para minúsculos.
- Exemplos de nomes válidos e inválidos:

```
my_table      -- válido
my_2nd_table  -- válido
échéanciers   -- válido: acentuação e letras não latinas são permitidas
"2nd_table"   -- válido: identificador de aspas
"createtable" -- válido: identificador de aspas
"1040Forms"   -- válido: identificador de aspas
2nd_table     -- inválido: não inicia com uma letra com sublinhado
```

- Todo banco de dados deve ter um nome exclusivo.

CREATE DATABASE

- Um servidor **PostgreSQL** pode gerenciar muitos databases, normalmente para cada projeto ou usuário é criado um database separado.
- Para criar um novo database, neste exemplo chamaremos de mydatabase, pode ser realizado de duas formas:
 - Executando o comando **createdb mydatabase** diretamente no shell da distribuição GNU/Linux que está sendo utilizada, para isso é necessário estar logado com o usuário **postgres**.
 - Executando a sintaxe SQL **CREATE DATABASE mydatabase;** diretamente conectado ao PostgreSQL utilizando o utilitário **psql**.
- O PostgreSQL permite criar qualquer número de banco de dados em um determinado servidor.

CREATE DATABASE - createdb

- Conecte com o usuário **postgres**:

```
[root@postgresql ~]# su -l postgres
```

- Crie o database com o comando **createdb**:

```
[postgres@postgresql ~]$ createdb mydatabase
```

- Com o database criado, vamos utilizar o **psql** para verificar se está mesmo criado:

```
postgres=# \l
```

List of databases

Name	Owner	Encoding	Collate	Ctype	Access privileges
mydatabase	postgres	UTF8	en_US.UTF-8	en_US.UTF-8	

CREATE DATABASE - psql

- Conectado ao usuário postgres, execute o utilitário psql:

```
[postgres@postgresql ~]$ psql
psql (12.1)
Type "help" for help.

postgres=# \1
```

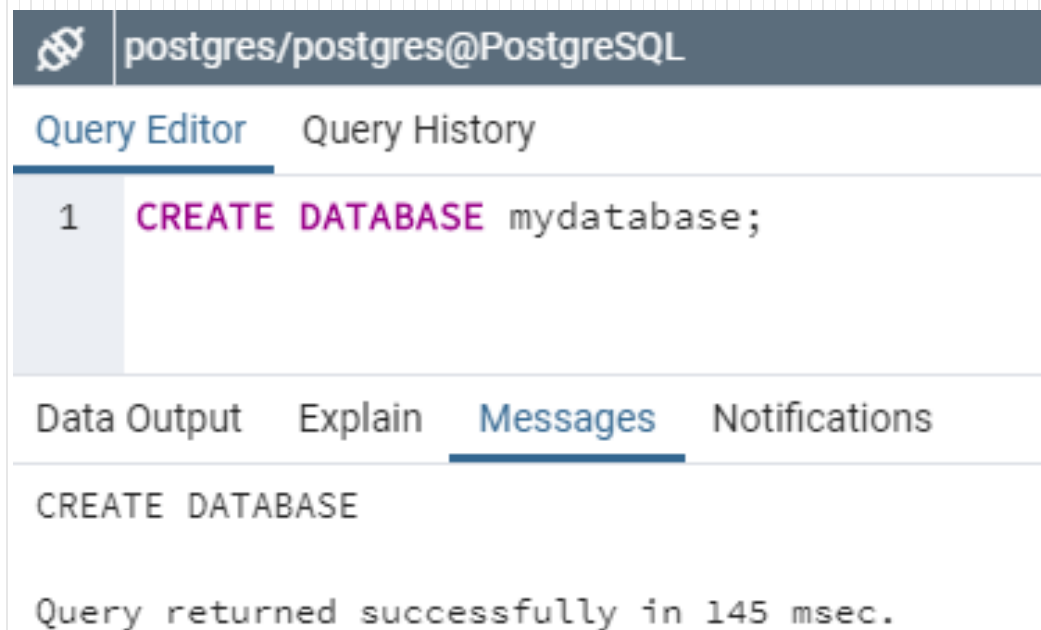
- Crie o database com o comando **CREATE DATABASE**:

```
postgres=# CREATE DATABASE mydatabase;
CREATE DATABASE
```

- Para verificar se o database foi criado corretamente, só executar o comando **\1** dentro do **psql**, conforme explicado anteriormente.

CREATE DATABASE - PGAdmin

- Como já informado, é possível gerenciar o servidor **PostgreSQL** através da ferramenta do **PGAdmin**.
- No **PGAdmin**, deve sempre utilizar a sintaxe SQL para execução dos comandos.
- Para criar o database no **PGAdmin**, utilize os mesmos comandos executados no **psql**.



DROP DATABASE - dropdb

- Se você não deseja mais usar o banco de dados criado, é possível removê-lo, ou executar um **drop** nesse database.
- Como já mostrado, esteja conectado com o usuário **postgres** para executar os comandos.
- Execute o comando **dropdb**:

```
[postgres@postgresql ~]$ dropdb mydatabase
```
- Para verificar se o database foi corretamente removido, utilize o **psql** e o comando **\l**, como já mostramos.

DROP DATABASE - psql

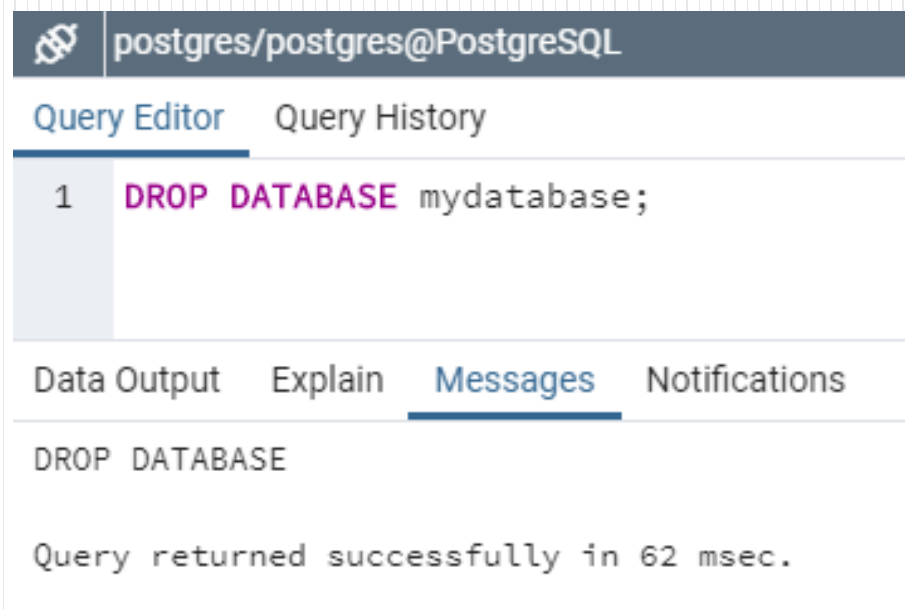
- Com usuário **postgres** conectado, execute o **psql**, como já mostramos.
- Com o **psql** aberto, execute o comando DROP DATABASE:

```
postgres=# DROP DATABASE mydatabase;  
DROP DATABASE
```

- Para verificar se o database foi corretamente removido, utilize o **psql** e o comando **\l**, como já mostramos.

DROP DATABASE - PGAdmin

- Para remover o database no **PGAdmin**, utilize os mesmos comandos executados no **psql**.



psql

- Para acessar o database que acabamos de criar (mydatabase), execute o utilitário **psql**:

```
[postgres@postgresql ~]$ psql mydatabase
psql (12.1)
Type "help" for help.

mydatabase=#
```

- Executando **psql mydatabase**, a conexão é feita diretamente no database criado, ao invés conectar ao servidor.
- A última linha de saída do **psql** (**mydatabase=#**) indica que o **psql** está aguardando as consultas SQL que se deseja executar.

psql

- Após a conexão ser realizado com sucesso, teste os seguintes comandos:
- **SELECT version ();** para consultar a versão do PostgreSQL instalada.

```
mydatabase=# SELECT version();
              version
-----
PostgreSQL 12.1 on x86_64-pc-linux-gnu, compiled by gcc (GCC) 8.3.1 20190507 (Red Hat 8.3.1-4), 64-bit
(1 row)
```

- **SELECT current_date;** para consultar a data atual do seu database.

```
mydatabase=# SELECT current_date;
 current_date
-----
2019-11-16
(1 row)
```

psql

- O **psql** possui alguns comandos internos que não são comandos **SQL**. Começam com o caracter da barra invertida "\".
- Por exemplo, para obter ajuda sobre comandos **SQL** execute **\h**:

```
mydatabase=# \h
Available help:
  ABORT                CREATE USER
  ALTER AGGREGATE      CREATE USER MAPPING
  ALTER COLLATION      CREATE VIEW
  ALTER CONVERSION     DEALLOCATE
  ALTER DATABASE      DECLARE
  ALTER DEFAULT PRIVILEGES DELETE
  ALTER DOMAIN         DISCARD
```

Questão

Ano: 2018 **Banca:** CCV-UFC **Órgão:** UFC **Prova:** CCV-UFC - 2018 - UFC - Analista de Tecnologia da Informação


Sob qual tipo de licença o banco de dados *PostgreSQL* é lançado atualmente?

- a) GPL license.
- b) BSD license.
- c) LGPL license.
- d) X11(MIT) license.
- e) PostgreSQL license.

Questão

Ano: 2018 **Banca:** CCV-UFC **Órgão:** UFC **Prova:** CCV-UFC - 2018 - UFC - Analista de Tecnologia da Informação

Sob qual tipo de licença o banco de dados *PostgreSQL* é lançado atualmente?

- a) GPL license.
- b) BSD license.
- c) LGPL license.
- d) X11(MIT) license.
-  PostgreSQL license.

Justificativa:

PostgreSQL is released under the **PostgreSQL license** , a liberal Open Source license, similar to the BSD or MIT licenses.

Why not the GNU General Public License?

People often ask why PostgreSQL is not released under the **GNU General Public License**.

The simple answer is: we like our license and do not want to change it!

<https://www.postgresql.org/about/licence/>

CONCEITOS

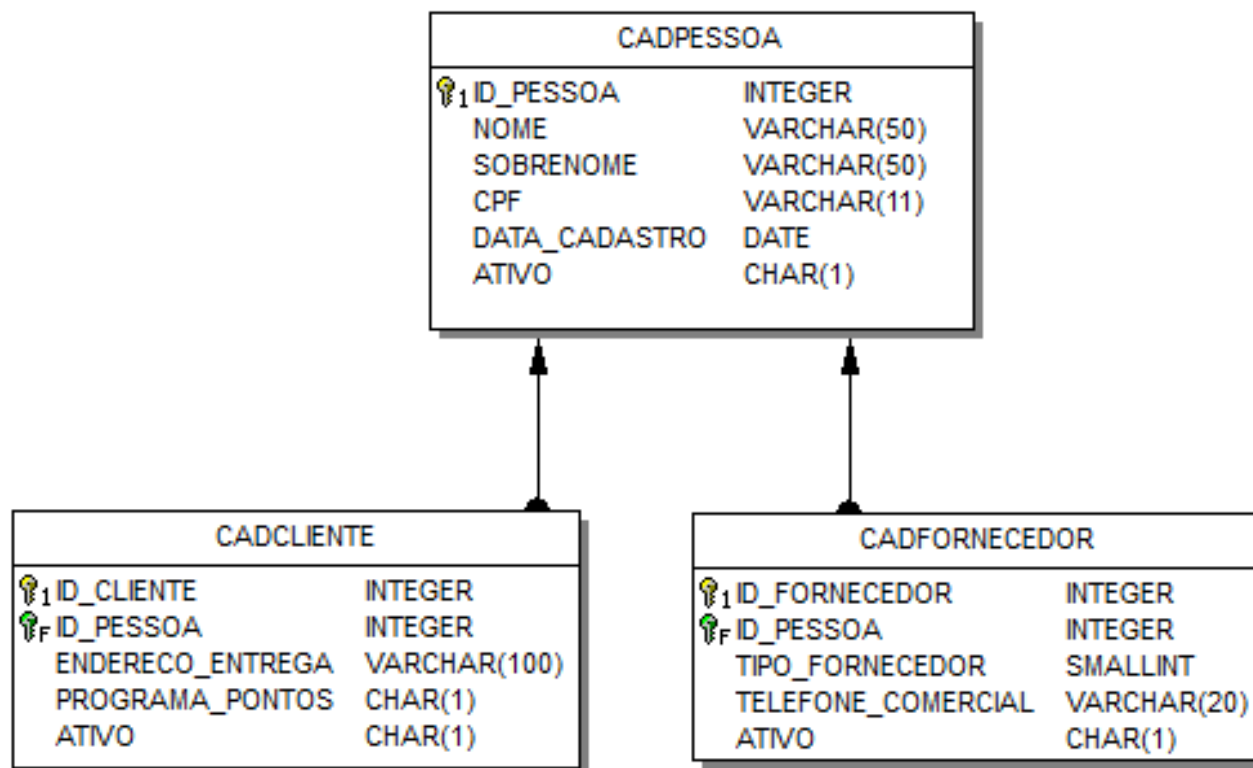
- O **PostgreSQL** é um sistema de gerenciamento de banco de dados relacional.
- O gerenciamento dos dados se dá em forma de **tabelas** ou **relações**.
- Uma coleção de **databases** gerenciados por uma única instância **PostgreSQL** constitui um cluster de banco de dados.
- Cada instalação do **PostgreSQL** cria uma área de dados onde são armazenados todos os **databases**.
- Um **database** do **PostgreSQL** pode contar **tabelas**, **views**, **triggers**, **stored procedures**, **domínios** e etc.
- Esses objetos podem ser agrupados em **schemas**, que são subdivisões dentro de um **database**.

BANCO DE DADOS RELACIONAL

- Um Banco de Dados Relacional segue o **Modelo Relacional**, cujo princípio se baseia em que todos os dados estão armazenados em **tabelas**.
- Uma **tabela** é uma simples estrutura de linhas e colunas.
- As **tabelas** associam-se entre si por meio de **regras de relacionamentos**, ou seja, associar vários atributos de uma tabela com vários atributos da outra tabela.
- As **linhas** formadas por uma lista ordenada de **colunas** representam um registro, ou **tuplas**.
- Um **registro** é uma instância de uma **tabela**, ou **entidade**.
- As **colunas** de uma **tabela** são chamadas de atributos.

BANCO DE DADOS RELACIONAL

- As **tabelas** se relacionam entre si através de **chaves**.
- Uma **chave** é um conjunto de um ou mais atributos.
- **Chave Primária** é um identificador exclusivo de todas as informações de cada registro.
- **Chave Estrangeira** é uma chave formada através de um relacionamento com a **chave primária** de outra **tabela**.



Questão

Ano: 2019 **Banca:** CESPE **Órgão:** TJ-AM **Prova:** CESPE - 2019 - TJ-AM - Assistente Judiciário - Programador

Julgue o próximo item, relativo a sistema gerenciador de banco de dados (SGBD).

Um SGBD trata do acesso ao banco e pode ser executado independentemente pelo Oracle, MySQL ou PostgreSQL; no entanto, cada SGBD utiliza DML (*data manipulation language*) e DDL (*data definition language*) específicas.

Certo

Errado

Questão

Ano: 2019 **Banca:** CESPE **Órgão:** TJ-AM **Prova:** CESPE - 2019 - TJ-AM - Assistente Judiciário - Programador

Julgue o próximo item, relativo a sistema gerenciador de banco de dados (SGBD).

Um SGBD trata do acesso ao banco e pode ser executado independentemente pelo Oracle, MySQL ou PostgreSQL; no entanto, cada SGBD utiliza DML (*data manipulation language*) e DDL (*data definition language*) específicas.



Certo

Errado

Justificativa:

Correto, por isso é necessário conhecer a linguagens e comandos específicas de cada um desses SGBDs.

Questão

Ano: 2019 **Banca:** CS-UFG **Órgão:** IF Goiano **Prova:** CS-UFG - 2019 - IF Goiano - Técnico de Tecnologia da Informação


Sistemas de Gerenciamento de Bancos de Dados (SGBDs) oferecem serviços para armazenamento de dados. MySQL, SQL Server e PostgreSQL são alguns exemplos. Especificamente, o PostgreSQL

- a) impede a criação de índices em dados do tipo texto e numérico.
- b) oferece os tipos de dados JSON e XML.
- c) é uma tecnologia que foi criada há menos de uma década.
- d) é incompatível com uso em nuvem (cloud computing).

Questão

Ano: 2019 **Banca:** CS-UFG **Órgão:** IF Goiano **Prova:** CS-UFG - 2019 - IF Goiano - Técnico de Tecnologia da Informação

Sistemas de Gerenciamento de Bancos de Dados (SGBDs) oferecem serviços para armazenamento de dados. MySQL, SQL Server e PostgreSQL são alguns exemplos. Especificamente, o PostgreSQL

- a) impede a criação de índices em dados do tipo texto e numérico.
-  oferece os tipos de dados JSON e XML.
- c) é uma tecnologia que foi criada há menos de uma década.
- d) é incompatível com uso em nuvem (cloud computing).

Justificativa:

- a) ERRADO, possibilita.
- b) CERTO
- c) ERRADO, Já tem mais de uma década, como vimos.
- d) ERRADO, totalmente compatível.

CREATE TABLE

- Para criar uma nova tabela se pode especificar o nome da tabela, juntamente com todos os nomes de colunas e seus tipos.
- Vamos criar a tabela abaixo no database já criado:

```
CREATE TABLE clima (  
    cidade varchar(80),  
    temp_min int, --temperatura mínima  
    temp_max int, --temperatura máxima  
    prpc real, --precipitação  
    dia date  
);
```

- O segundo exemplo armazenará cidades e sua localização geográfica associada:

```
CREATE TABLE cidades (  
    nome varchar(80),  
    localizacao point  
);
```

DROP TABLE

- Caso não precise mais de uma tabela ou deseja recriar uma maneira diferente, com outros atributos, a tabela pode ser removida usando o seguinte comando:

```
DROP TABLE nome_tabela;
```

ALTER TABLE

- Para adicionar uma **coluna**, use o comando:

```
ALTER TABLE nome_tabela ADD COLUMN nome_coluna text;
```

- A nova **coluna** é preenchida inicialmente com qualquer valor padrão fornecido.
- Para remover uma **coluna**, use o comando:

```
ALTER TABLE nome_tabela DROP COLUMN nome_coluna;
```

- Quaisquer dados que estiverem na **coluna** irão desaparecer.
- Caso a **coluna** esteja referenciada por uma restrição de **chave estrangeira** de outra **tabela**, o **PostgreSQL** não eliminará silenciosamente essa restrição.
- Você pode autorizar a eliminação de tudo o que depende da **coluna** adicionando **CASCADE**.

```
ALTER TABLE nome_tabela DROP COLUMN nome_coluna CASCADE;
```

ALTER TABLE

- Para definir um novo valor padrão para uma **coluna**, use o comando:

```
ALTER TABLE nome_tabela ALTER COLUMN nome_coluna SET DEFAULT 10;
```

- Observe que isso não afeta nenhuma **linha** existente na **tabela**, apenas altera o padrão para futuros comandos **INSERT**.
- Para remover qualquer valor padrão, use:

```
ALTER TABLE nome_tabela ALTER COLUMN nome_coluna DROP DEFAULT;
```
- Isso é efetivamente o mesmo que definir o padrão como nulo.

ALTER TABLE

- Para converter uma **coluna** em um tipo de dados diferente, use o comando:

```
ALTER TABLE nome_tabela ALTER COLUMN nome_coluna TYPE numeric(10,2);
```
- Só executará com sucesso apenas se cada entrada existente na **coluna** puder ser convertida no novo tipo por uma conversão implícita.
- Para renomear uma **coluna**:

```
ALTER TABLE nome_tabela RENAME COLUMN nome_coluna TO novo_nome_coluna;
```
- Para renomear uma **tabela**:

```
ALTER TABLE nome_tabela RENAME TO novo_nome_tabela;
```

TIPOS DE DADOS

- O **PostgreSQL** suporta os tipos de dados padrões do **SQL** além de outros tipos de utilidade geral e um rico conjunto de tipos geométricos.
- A seguir iremos passar por um overview dos tipos mais utilizados (Numéricos, Monetários, Caracteres, Binários, Data e Hora, Boolean).
- Outros tipos de dados também são suportados (Enumerados, Geométricos, Endereço de rede, Sequência de bits, Pesquisa de texto, UUID, XML, JSON, Matrizes, Compostos, Intervalo, Domínio, Identificador de objeto, pg_Isn e Pseudo-tipos), tire um tempo para entender cada um desses tipos.



TIPOS DE DADOS - Numéricos

- Lista dos tipos numéricos disponíveis no **PostgreSQL**:

Name	Storage Size	Description	Range
smallint	2 bytes	small-range integer	-32768 to +32767
integer	4 bytes	typical choice for integer	-2147483648 to +2147483647
bigint	8 bytes	large-range integer	-9223372036854775808 to 9223372036854775807
decimal	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
numeric	variable	user-specified precision, exact	up to 131072 digits before the decimal point; up to 16383 digits after the decimal point
real	4 bytes	variable-precision, inexact	6 decimal digits precision
double precision	8 bytes	variable-precision, inexact	15 decimal digits precision
serial	4 bytes	autoincrementing integer	1 to 2147483647
bigserial	8 bytes	large autoincrementing integer	1 to 9223372036854775807

TIPOS DE DADOS - Monetários

- O tipo de dado **money** armazena um valor em moeda com uma precisão fracionária fixa.
- O intervalo mostrado na tabela abaixo assume que há dois dígitos fracionários.
- A precisão fracionária é determinada pela configuração da database **lc_monetary** (define o local a ser usado para formatar valores monetários).
- A entrada é aceita em vários formatos, incluindo números inteiros, bem como formatação de moeda típica. Por exemplo: "**R\$ 1.000,00**"

Name	Storage Size	Description	Range
money	8 bytes	currency amount	-92233720368547758.08 to +92233720368547758.07

TIPOS DE DADOS - Caracteres

- O **SQL** define dois tipos principais de tipos de dados para caracteres: **character varying(n)** e **character(n)**, em que **n** é um número inteiro positivo.
- Esses dois tipos podem armazenar sequências de caracteres com até **n** caracteres, caso a string seja mais longa que **n** caracteres, resultará em erro.
- **varchar(n)** e **char(n)** são aliases para **character varying(n)** e **character(n)**.
- O **PostgreSQL** fornece um outro tipo de dado chamado **text**, que armazena sequências de caracteres de qualquer tamanho, mas não é padrão do **SQL**.

Name	Description
character varying(n), varchar(n)	variable-length with limit
character(n), char(n)	fixed-length, blank padded
text	variable unlimited length

TIPOS DE DADOS - Caracteres

- Existem outros dois tipos de dados de tamanho fixo no **PostgreSQL**.
- O tipo **name** existe apenas para o armazenamento de identificadores nos catálogos internos do sistema.
- Seu comprimento é definido como 64 bytes (63 caracteres utilizáveis mais o terminador)
- O tipo **"char"** (observe as aspas) é diferente de char (1), pois ele usa apenas um byte de armazenamento.

Name	Storage Size	Description
"char"	1 byte	single-byte internal type
name	64 bytes	internal type for object names

TIPOS DE DADOS - Binários

- Uma string binária é uma sequência de octetos (ou bytes).
- As cadeias binárias são diferentes das cadeias de caracteres de duas formas.
- Cadeias binárias são apropriadas para armazenar dados que o programador considera "bytes brutos" (raw bytes), enquanto cadeias de caracteres são apropriadas para armazenar texto.
- O tipo de dados **bytea** suporta dois formatos externos para entrada e saída: o histórico "**escape**" do PostgreSQL e o formato "**hex**".

Name	Storage Size	Description
bytea	1 or 4 bytes plus the actual binary string	variable-length binary string

TIPOS DE DADOS - Binários

- O formato "**hex**" codifica os dados binários como dois dígitos hexadecimais por byte, o mais significativo primeiro.
- O formato **hex** é compatível com diversos aplicativos e protocolos externos e tende a ser mais rápido para converter do que o formato de **escape**.
- O formato "**escape**" é o formato tradicional do **PostgreSQL** para o tipo **bytea**.
- É adotada uma abordagem de representar uma sequência de caracteres binários como uma sequência de caracteres ASCII.

Decimal Octet Value	Description	Escaped Output Representation	Example	Output Result
92	backslash	\\	SELECT E'\\134'::bytea;	\\
0 to 31 and 127 to 255	"non-printable" octets	\xxx (octal value)	SELECT E'\\001'::bytea;	\001
32 to 126	"printable" octets	client character set representation	SELECT E'\\176'::bytea;	~

TIPOS DE DADOS - Data e Hora

- Entrada de data e hora são aceitas em quase qualquer formato razoável.
- O **PostgreSQL** é bastante flexível no tratamento das entradas de data e hora do que o padrão **SQL** exige.

Name	Storage Size	Description	Low Value	High Value	Resolution
timestamp [(p)] [without time zone]	8 bytes	both date and time (no time zone)	4713 BC	294276 AD	1 microsecond / 14 digits
timestamp [(p)] with time zone	8 bytes	both date and time, with time zone	4713 BC	294276 AD	1 microsecond / 14 digits
date	4 bytes	date (no time of day)	4713 BC	5874897 AD	1 day
time [(p)] [without time zone]	8 bytes	time of day (no date)	00:00:00	24:00:00	1 microsecond / 14 digits
time [(p)] with time zone	12 bytes	times of day only, with time zone	00:00:00+1459	24:00:00-1459	1 microsecond / 14 digits
interval [fields] [(p)]	16 bytes	time interval	-178000000 years	178000000 years	1 microsecond / 14 digits

- **time**, **timestamp** e **interval** aceitam um valor de precisão opcional **p** que especifica o número de dígitos fracionários no campo de segundos.

TIPOS DE DADOS - Data

- Algumas entradas possíveis para o tipo **date**:

Example	Description
1999-01-08	ISO 8601; January 8 in any mode (recommended format)
January 8, 1999	unambiguous in any datestyle input mode
1/8/1999	January 8 in MDY mode; August 1 in DMY mode
1/18/1999	January 18 in MDY mode; rejected in other modes
01/02/03	January 2, 2003 in MDY mode; February 1, 2003 in DMY mode; February 3, 2001 in YMD mode
1999-Jan-08	January 8 in any mode
Jan-08-1999	January 8 in any mode
08-Jan-1999	January 8 in any mode
99-Jan-08	January 8 in YMD mode, else error
08-Jan-99	January 8, except error in YMD mode
Jan-08-99	January 8, except error in YMD mode
19990108	ISO 8601; January 8, 1999 in any mode

TIPOS DE DADOS - Hora

- Algumas entradas possíveis para o tipo **time**:

Example	Description
04:05:06.789	ISO 8601
04:05:06	ISO 8601
04:05	ISO 8601
040506	ISO 8601
04:05 AM	same as 04:05; AM does not affect value
04:05 PM	same as 16:05; input hour must be <= 12
04:05:06.789-8	ISO 8601
04:05:06-08:00	ISO 8601
04:05-08:00	ISO 8601
040506-08	ISO 8601
04:05:06 PST	time zone specified by abbreviation
2003-04-12 04:05:06 America/New_York	time zone specified by full name

TIPOS DE DADOS - Boolean

- O **PostgreSQL** fornece o tipo **boolean** padrão do **SQL**.
- O tipo **boolean** pode ter vários estados: "**true**", "**false**" e um terceiro estado, "**unknown**", que é representado pelo valor nulo (null) do **SQL**.

Name	Storage Size	Description
boolean	1 byte	state of true or false

- O tipo **boolean** aceita as seguintes funções como entrada para representar um estado "**true**": "**true**", "**yes**", "**on**", "**1**".
- E essas funções para representar um estado "**false**": "**false**", "**no**", "**off**", "**0**".
- A função de saída para o tipo **boolean** sempre emite "**t**" ou "**f**".

TIPOS DE DADOS - Geométricos

- Os tipos de dados **geométricos** representam objetos espaciais bidimensionais.

Name	Storage Size	Description	Representation
point	16 bytes	Point on a plane	(x,y)
line	32 bytes	Infinite line	{A,B,C}
lseg	32 bytes	Finite line segment	((x1,y1),(x2,y2))
box	32 bytes	Rectangular box	((x1,y1),(x2,y2))
path	16+16n bytes	Closed path (similar to polygon)	((x1,y1),...)
path	16+16n bytes	Open path	[(x1,y1),...]
polygon	40+16n bytes	Polygon (similar to closed path)	((x1,y1),...)
circle	24 bytes	Circle	<(x,y),r> (center point and radius)

- Um rico conjunto de funções e operadores está disponível para executar várias operações **geométricas**, como escala, translação, rotação e determinação de interseções.

INSERT

- O comando **INSERT** é usada para preencher uma **tabela** com **linhas**:

```
INSERT INTO clima VALUES (  
    'Brasilia',  
    14,  
    32,  
    0.10,  
    '2019-11-16'  
);
```

- Observe que todos os tipos de dados usam formatos de entrada óbvios. As constantes que não são valores números devem estar entre aspas simples.
- O tipo **point** requer um par de coordenadas como entrada, como mostrado aqui:

```
INSERT INTO cidades VALUES (  
    'Brasilia',  
    '(-15.7801,-47.9292)'  
);
```

INSERT

- A sintaxe que usamos requer que você lembre a ordem das **colunas** na **tabela**. Uma sintaxe alternativa permite listar explicitamente as colunas:

```
INSERT INTO clima (cidade, temp_min, temp_max, prcp, dia)
VALUES ('Brasilia', 14, 32, 0.10, '2019-11-16');
```

- Também é possível listar as **colunas** em uma ordem diferente ou mesmo omitir algumas **colunas**, por exemplo, a precipitação é desconhecida:

```
INSERT INTO clima (dia, cidade, temp_max, temp_min)
VALUES ('2019-11-16', 'Brasilia', 32, 14);
```

- Muitos usuários consideram uma boa prática sempre listar os nomes das **colunas**.

INSERT

- É possível inserir várias **linhas** em um único comando:

```
INSERT INTO clima (cidade, temp_min, temp_max, prcp, dia) VALUES  
  ('Sao Paulo', 9, 27, 0.20, '2019-11-16'),  
  ('Rio de Janeiro', 15 33, 0.35, '2019-11-16'),  
  ('Belo Horizonte', 14, 29, 0.40, '2019-11-16');
```

- Também é possível inserir o resultado de uma **consulta** (query), que pode ser sem **linhas**, uma **linha** ou várias **linhas**:

```
INSERT INTO clima (cidade)  
SELECT nome FROM cidades;
```

UPDATE

- Você pode atualizar **linhas** individuais, todas as **linhas** de uma **tabela** ou um subconjunto de todas as **linhas**.
- Para atualizar **linhas** existentes, use o comando **UPDATE**. Requer três dados:
 - O nome da tabela e coluna a serem atualizadas
 - O novo valor da coluna
 - Quais linhas atualizar

```
UPDATE clima
SET temp_max = temp_max - 2,
    temp_min = temp_min - 2;
```

DELETE

- As **linhas** podem ser removidas de uma **tabela** usando o comando **DELETE**.
- A sintaxe é muito semelhante ao comando **UPDATE**.

```
DELETE FROM clima WHERE city = 'Brasília';
```

- Com isso, deletamos todas as informações de clima da cidade de Brasília.
- Tenha cuidado com os comandos **SQL** que incluem a instrução **DELETE**.

```
DELETE FROM clima;
```

- Sem uma qualificação, a instrução **DELETE** removerá todas as linhas da **tabela** especificada, deixando-a em branco.

SELECT - FROM

- Para recuperar dados de uma tabela, a tabela é "**queried**".
- A instrução **SQL** usada para fazer isso é o **SELECT**.
- A instrução é dividida em uma lista de seleção:
 - A parte que lista as **colunas** a serem retornadas
 - A parte que lista as **tabelas** das quais se deseja recuperar os dados
 - Uma qualificação opcional, especificar quaisquer restrições
- Para recuperar todas as **linhas** da **tabela clima**, execute:

```
SELECT * FROM clima;
```
- O * é uma abreviação para **todas as colunas**, teremos o mesmo resultado com:

```
SELECT cidade, temp_min, temp_max, prcp, dia FROM weather;
```

SELECT - WHERE, DISTINCT e ORDER BY

- Uma **consulta** pode ser "qualificada" adicionando uma cláusula **WHERE** que especifica quais **linhas** são desejadas.

```
SELECT * FROM clima WHERE city = 'Brasilia';
```

- A cláusula **WHERE** contém uma expressão booleana e somente as **linhas** para as quais a expressão booleana é verdadeira são retornadas.
- Você pode solicitar que os resultados de uma **consulta** sejam retornados em ordem classificada:

```
SELECT * FROM clima ORDER BY cidade;
```

- Você pode garantir resultados consistentes usando **DISTINCT** e **ORDER BY** juntos:

```
SELECT DISTINCT cidade FROM clima ORDER BY cidade;
```

Questão

Ano: 2018 **Banca:** COMPERVE **Órgão:** UFRN **Prova:** COMPERVE - 2018 - UFRN - Analista de Tecnologia da Informação – 103

Para responder a questão considere as seguintes tabelas de dados a serem armazenadas em um banco de dados relacional PostgreSQL.

Tabela ALUNO:

id	Nome	idade	matricula
1	João Pedro	22	1234
2	Bernardo Freire	24	2312
...

Tabela AMIZADES:

id_amigo_1	Id_amigo_2
1	2
...	...

Para selecionar os nomes dos alunos por ordem alfabética crescente, deve-se executar no PostgreSQL o seguinte comando SQL:

- a) SELECT NOME FROM aluno ORDER BY nome DESC
- b) SELECT NOME FROM aluno ORDER BY nome ASC
- c) SELECT NOME FROM aluno SORTED BY nome DESC
- d) SELECT NOME FROM aluno SORTED BY nome ASC

Questão

Ano: 2018 **Banca:** COMPERVE **Órgão:** UFRN **Prova:** COMPERVE - 2018 - UFRN - Analista de Tecnologia da Informação – 103

Para responder a questão considere as seguintes tabelas de dados a serem armazenadas em um banco de dados relacional PostgreSQL.


Tabela ALUNO:

id	Nome	idade	matricula
1	João Pedro	22	1234
2	Bernardo Freire	24	2312
...

Tabela AMIZADES:

id_amigo_1	Id_amigo_2
1	2
...	...

Para selecionar os nomes dos alunos por ordem alfabética crescente, deve-se executar no PostgreSQL o seguinte comando SQL:

- a) SELECT NOME FROM aluno ORDER BY nome DESC
-  b) SELECT NOME FROM aluno ORDER BY nome ASC
- c) SELECT NOME FROM aluno SORTED BY nome DESC
- d) SELECT NOME FROM aluno SORTED BY nome ASC

Justificativa:

ASC – ASCEDENTE

DESC - DESCENDENTE

SELECT - Concatenation Operator

- Sequências de caracteres com tipo não especificado são correspondidas com prováveis candidatos a operadores de concatenação.
- O operador de concatenação é "||".

```
SELECT text 'abc' || 'def' AS "texto e desconhecido";
```

```
texto e desconhecido
```

```
-----  
abcdef
```

- Nesse caso, como o primeiro argumento é do tipo texto, o analisador assume que o segundo argumento deve ser do tipo texto também.
- Aqui está uma concatenação de dois valores de tipos não especificados:

```
SELECT 'abc' || 'def' AS "desconhecido";
```

```
desconhecido
```

```
-----  
abcdef
```

SELECT - SUBQUERY

- Em casos que queremos uma maneira de passar o resultado da primeira **consulta** para a segunda em uma **consulta**. A solução é usar uma **subquery**.
- Uma subquery é uma **consulta** aninhada dentro de outra **consulta**, como **SELECT**, **INSERT**, **DELETE** e **UPDATE**.
- Vamos imaginar que temos uma **tabela** com dados de locadora de filmes, observe o exemplo de uma **subquery**:

```
SELECT film_id, title, rental_rate FROM film WHERE  
    rental_rate > (  
        SELECT AVG (rental_rate) FROM film  
    );
```

- A **query** dentro dos colchetes é chamada de **subquery** ou **inner query**. A **query** que contém a **subquery** é conhecida como **outer query**.

JOIN

- As **consultas** podem acessar várias **tabelas** ao mesmo tempo ou acessar a mesma **tabela** de modo que várias **linhas** da **tabela** estejam sendo processadas ao mesmo tempo.
- Uma **consulta** que acessa várias **linhas** da mesma ou de diferentes **tabelas** ao mesmo tempo é chamada de **join query**.
- Uma **tabela unida (joined table)** é uma **tabela** derivada de duas outras **tabelas** (reais ou derivadas) de acordo com as regras do tipo de união escolhido:
 - INNER, junções internas
 - OUTER, junções externas
 - CROSS JOIN, junções cruzadas

JOIN

- Observe a seguinte **query**:

```
SELECT * FROM clima, cidades WHERE cidade = nome;
```

- O resultado será informações das cidades que constam nas duas tabelas, **cidades** e **clima**, a junção ignora linhas não correspondentes.
- Como todas as **colunas** possuem nomes diferentes, a boa prática, como já vimos, é qualificar os nomes das **colunas** conforme o comando:

```
SELECT clima.cidade clima.temp_min, clima.temp_max,  
       clima.prcp, clima.dia, cidades.localizacao  
FROM clima, cidades  
WHERE cidades.nome = clima.cidade;
```

JOIN

- As palavras **INNER** e **OUTER** são opcionais em todas as formas. **INNER** é o padrão. **LEFT**, **RIGHT** e **FULL** implicam em uma junção externa.
- A condição de **junção** é especificada na cláusula **ON** ou **USING**, ou implicitamente pela palavra **NATURAL**.
- A condição de **junção** determina quais linhas das duas tabelas de origem são consideradas "correspondentes".
- Para explicar os tipos de **JOIN**, vamos usar as seguintes **tabelas** como exemplo:

t1:	
num	name
1	a
2	b
3	c

t2:	
num	value
1	xxx
3	yyy
5	zzz

INNER JOIN

- Para cada linha R1 de T1, a **tabela** unida possui uma **linha** para cada **linha** em T2 que satisfaz a condição de **junção** com R1.

```
SELECT * FROM t1 INNER JOIN t2 ON t1.num = t2.num;
```

num	name	num	value
1	a	1	xxx
3	c	3	yyy

```
SELECT * FROM t1 INNER JOIN t2 USING (num);
```

num	name	value
1	a	xxx
3	c	yyy

```
SELECT * FROM t1 NATURAL INNER JOIN t2;
```

num	name	value
1	a	xxx
3	c	yyy

LEFT OUTER JOIN

- Primeiro, uma **junção** interna é realizada.
- Para cada **linha** em T1 que não satisfaça a condição de **junção** com qualquer **linha** em T2, uma **linha** é adicionada com valores nulos nas **colunas** de T2.
- Assim, a **tabela** unida tem pelo menos uma **linha** para cada **linha** em T1.

```
SELECT * FROM t1 LEFT JOIN t2 ON t1.num = t2.num;
```

num	name	num	value
1	a	1	xxx
2	b		
3	c	3	yyy

```
SELECT * FROM t1 LEFT JOIN t2 USING (num) ;
```

num	name	value
1	a	xxx
2	b	
3	c	yyy

RIGHT OUTER JOIN

- Primeiro, uma **junção** interna é realizada.
- Para cada **linha** em T2 que não satisfaça a condição de **junção** com qualquer **linha** em T1, uma **linha** é adicionada com valores nulos nas **colunas** de T1.
- A **tabela** unida tem pelo uma **linha** para cada **linha** em T2, sendo o inverso de uma junção a esquerda.

```
SELECT * FROM t1 RIGHT JOIN t2 ON t1.num = t2.num;
```

num	name	num	value
1	a	1	xxx
3	c	3	yyy
...		5	zzz

FULL OUTER JOIN

- Primeiro, uma **junção** interna é realizada.
- Para cada **linha** em T1 que não satisfaça a condição de **junção** com qualquer **linha** em T2, uma **linha** é adicionada com valores nulos nas **colunas** de T2.
- Além disso, para cada **linha** em T2 que não satisfaça a condição de **junção** com qualquer **linha** em T1, uma **linha** é adicionada com valores nulos nas **colunas** de T1.

```
SELECT * FROM t1 FULL JOIN t2 ON t1.num = t2.num;
```

num	name	num	value
1	a	1	xxx
2	b		
3	c	3	yyy
...		5	zzz

CROSS JOIN

- Para todas as combinações possíveis de **linhas** de T1 e T2, a **tabela** unida conterá uma **linha** que consiste em todas as **colunas** em T1 seguidas por todas as **colunas** em T2.

```
SELECT * FROM t1 CROSS JOIN t2;
```

num	name	num	value
1	a	1	xxx
1	a	3	yyy
1	a	5	zzz
2	b	1	xxx
2	b	3	yyy
2	b	5	zzz
3	c	1	xxx
3	c	3	yyy
3	c	5	zzz

GROUP BY

- Usado para agrupar **linhas** em uma **tabela** que possuem os mesmos valores em todas as **colunas** listadas.
- O efeito é combinar cada conjunto de **linhas** com valores comuns em uma **linha** do **grupo** que representa todas as **linhas** do **grupo**.

```
SELECT * FROM test1;
```

x	y
a	3
c	2
b	5
a	1

```
SELECT x FROM test1 GROUP BY x;
```

x
a
b
c

GROUP BY - HAVING

- Se uma **tabela** foi agrupada usando **GROUP BY**, mas apenas alguns **grupos** são de interesse, a cláusula **HAVING** pode ser usada, bem como uma cláusula **WHERE**, para eliminar **grupos** do resultado.

```
SELECT x, sum(y) FROM test1 GROUP BY x HAVING sum(y) > 3;
```

x	sum
a	4
b	5

```
SELECT x, sum(y) FROM test1 GROUP BY x HAVING x < 'c';
```

x	sum
a	4
b	5

- Aqui, **sum** é uma função agregada que calcula um único valor em todo o **grupo**.

PRIMARY KEYS

- Uma restrição de **chave primária** indica que uma **coluna**, ou grupo de **colunas**, pode ser usada como um identificador exclusivo para **linhas** na tabela.
- Requer que os valores sejam únicos e não nulos.

```
CREATE TABLE produtos (  
    produto_nr integer UNIQUE NOT NULL,  
    nome text,  
    preco numeric  
);
```

```
CREATE TABLE produtos (  
    produto_nr integer PRIMARY KEY,  
    nome text,  
    preco numeric  
);
```

- As chaves primárias podem abranger mais de uma coluna;

```
CREATE TABLE exemplo (  
    a integer,  
    b integer,  
    c integer,  
    PRIMARY KEY (a, c)  
);
```

PRIMARY KEYS - SERIAL e BIGSERIAL

- A técnica mais simples e comum para adicionar uma **chave primária** no **PostgreSQL** é usar os tipos de dados **SERIAL** ou **BIGSERIAL**.
- **SERIAL** não é um tipo de dado verdadeiro, mas é simplesmente uma notação abreviada que instrui o **PostgreSQL** a criar um identificador exclusivo e incrementado automaticamente para a coluna especificada.

```
CREATE TABLE produtos (  
    produto_nr SERIAL PRIMARY KEY,  
    nome text,  
    preco numeric  
);
```

PRIMARY KEYS - SEQUENCE

- Em alguns casos raros, o padrão incremental utilizado pelos tipos de dados **SERIAL** e **BIGSERIAL** pode não atender as necessidades.
- É possível executar a mesma funcionalidade de chave primária incrementada automaticamente, criando uma **SEQUENCE** personalizada.

```
CREATE SEQUENCE produtos_sequence start 2 increment 2;
```

- Neste exemplo, estamos criando uma **SEQUENCE** que inicia com o número 2 e é incrementada de 2 em 2.
- Quando inserimos um novo registro em nossa **tabela**, precisamos usar o próximo valor da sequência com **nextval** ("produtos_sequence") como ID.

```
INSERT INTO produtos (produto_nr, nome, preco)  
VALUES (nextval('produtos_sequence'), 'Arroz', '5.00');
```

Questão

Ano: 2018 **Banca:** COMPERVE **Órgão:** UFRN **Prova:** COMPERVE - 2018 - UFRN - Analista de Tecnologia da Informação – 103

Para responder a questão considere as seguintes tabelas de dados a serem armazenadas em um banco de dados relacional PostgreSQL.

Tabela ALUNO:

id	Nome	idade	matricula
1	João Pedro	22	1234
2	Bernardo Freire	24	2312
...

Tabela AMIZADES:

id_amigo_1	Id_amigo_2
1	2
...	...

Para se criar a tabela ALUNO em um banco de dados PostgreSQL, o comando SQL que deve ser executado é:

- a) CREATE TABLE ALUNO (id SERIAL, nome VARCHAR(250), idade INT, matrícula VARCHAR(10), primary key(id))
- b) CREATE TABLE ALUNO (id AUTO_INCREMENT, nome VARCHAR(250), idade INT, matrícula VARCHAR(10), primary key(id))
- c) CREATE TABLE ALUNO (id AUTO_INCREMENT, nome STRING, idade INT, matrícula STRING, primary key(id))
- d) CREATE TABLE ALUNO (id SERIAL, nome STRING, idade INT, matrícula STRING, primary key(id))

Questão

Ano: 2018 **Banca:** COMPERVE **Órgão:** UFRN **Prova:** COMPERVE - 2018 - UFRN - Analista de Tecnologia da Informação – 103

Para responder a questão considere as seguintes tabelas de dados a serem armazenadas em um banco de dados relacional PostgreSQL.


Tabela ALUNO:

id	Nome	idade	matricula
1	João Pedro	22	1234
2	Bernardo Freire	24	2312
...

Tabela AMIZADES:

id_amigo_1	Id_amigo_2
1	2
...	...

Para se criar a tabela ALUNO em um banco de dados PostgreSQL, o comando SQL que deve ser executado é:

-  **a)** CREATE TABLE ALUNO (id SERIAL, nome VARCHAR(250), idade INT, matrícula VARCHAR(10), primary key(id))
- b)** CREATE TABLE ALUNO (id AUTO_INCREMENT, nome VARCHAR(250), idade INT, matrícula VARCHAR(10), primary key(id))
- c)** CREATE TABLE ALUNO (id AUTO_INCREMENT, nome STRING, idade INT, matrícula STRING, primary key(id))
- d)** CREATE TABLE ALUNO (id SERIAL, nome STRING, idade INT, matrícula STRING, primary key(id))

Justificativa:

LETRA B e C estão erradas porque o campo "id" não recebeu o data type.

LETRA D está errada pois "string" não é um data type.

FOREIGN KEYS

- Uma restrição de **chave estrangeira** especifica que os valores em uma **coluna**, ou um grupo de **colunas**, devem corresponder aos valores que aparecem em alguma **linha** de outra **tabela**.
- Isso mantém a integridade referencial entre duas **tabelas** relacionadas.

```
CREATE TABLE pedidos (  
    pedido_nr integer PRIMARY KEY,  
    produto_nr integer REFERENCES produtos (produto_nr),  
    quantidade integer  
);
```

- Com isso é impossível criar pedidos com entradas que não sejam NULL produto_nr que não apareçam na **tabela** de produtos.

FOREIGN KEYS

- Nessa situação a **tabela** de pedidos é a **tabela** que está referenciando e a **tabela** de produtos é a **tabela** que é referenciada.
- Existem **colunas** que são referenciadas e que referenciam.
- Podemos reduzir o comando anterior para:

```
CREATE TABLE pedidos (  
    pedido_nr integer PRIMARY KEY,  
    produto_nr integer REFERENCES produtos,  
    quantidade integer  
);
```
- Na ausência de uma **lista** de **colunas**, a **chave primária** da **tabela** referenciada é usada como **coluna(s)** referenciada(s).

FOREIGN KEYS - RESTRICT e CASCADE

- Uma **tabela** pode ter mais de uma restrição de **chave estrangeira**. Isso é usado para implementar relacionamentos muitos para muitos entre **tabelas**.
- Sabemos que as **chaves estrangeiras** não permitem a criação de pedidos que não estejam relacionados a nenhum produto, mas e se um produto for removido após a criação de um pedido que faça referência a ele?
- O **SQL** lida com isso usando **RESTRICT** e **CASCADE**.

```
CREATE TABLE itens_pedidos (  
    produto_nr integer REFERENCES produtos ON DELETE RESTRICT,  
    pedido_nr integer REFERENCES pedidos ON DELETE CASCADE,  
    quantidade integer,  
    PRIMARY KEY (produto_nr, pedido_nr)  
);
```

- Restrições e exclusões em cascata são as duas opções mais comuns.

FOREIGN KEYS - RESTRICT e CASCADE

- **RESTRICT** impede a exclusão de uma **linha** referenciada.
- **CASCADE** especifica que quando uma **linha** referenciada é excluída, as **linhas** que fazem referência a ela também devem ser excluídas automaticamente.
- Análogo ao **ON DELETE**, também existe **ON UPDATE**, que é chamado quando uma **coluna** referenciada é alterada (atualizada).

VIEWS

- As **visualizações** no **PostgreSQL** são implementadas usando o sistema de regras, não há essencialmente nenhuma diferença entre:

```
CREATE VIEW myview AS SELECT * FROM mytab;
```

- Não há nenhuma diferença entre uma **tabela** e uma **visualização**, elas são a mesma coisa: relações.
- As **visualizações** permitem encapsular os detalhes da estrutura de suas **tabelas**, que podem mudar à medida que seu aplicativo evolui.
- As **visualizações** podem ser usadas em praticamente qualquer lugar em que uma **tabela** real possa ser usada.
- As regras **ON SELECT** são aplicadas a todas as consultas como a última etapa, mesmo que o comando fornecido seja **INSERT**, **UPDATE** ou **DELETE**.

VIEWS

- O benefício de implementar **visualizações** com o sistema de regras é que o desenvolvedor possui todas as informações sobre quais **tabelas** devem ser varridas, além dos **relacionamentos** entre essas **tabelas**, além das qualificações restritivas das **visualizações** e das qualificações da consulta original em uma única árvore de **consulta**.
- O sistema de regras implementado no **PostgreSQL** garante que todas essas informações estejam disponíveis sobre a **consulta** até aquele momento.

TRANSACTIONS

- As **transações** são um conceito fundamental de todos os **SGBD**.
- O ponto essencial de uma **transação** é que ela agrupa várias etapas em uma única operação de tudo ou nada.
- Garantir que, uma vez que uma **transação** seja concluída e reconhecida pelo **SGBD**, ela tenha sido permanentemente registrada e não será perdida, mesmo que ocorra uma falha logo em seguida.
- Quando várias **transações** estão sendo executadas simultaneamente, cada uma delas não deve ser capaz de ver as alterações incompletas feitas por outras pessoas.

TRANSACTIONS - ACID

- **ACID** é um conceito que se refere às quatro propriedades de **transação** de um SGBD: **Atomicidade**, **Consistência**, **Isolamento** e **Durabilidade**.
 - **Atomicidade:** Em uma transação envolvendo duas ou mais partes de informações discretas, ou a transação será executada totalmente ou não será executada, garantindo assim que as transações sejam atômicas.
 - **Consistência:** A transação cria um novo estado válido dos dados ou em caso de falha retorna todos os dados ao seu estado antes que a transação foi iniciada.
 - **Isolamento:** Uma transação em andamento deve permanecer isolada de qualquer outra operação, ou seja, garantimos que a transação não será interferida por nenhuma outra transação concorrente.
 - **Durabilidade:** Dados validados são registrados pelo sistema de tal forma que mesmo no caso de uma falha e/ou reinício do sistema, os dados estão disponíveis em seu estado correto.

TRANSACTIONS - BEGIN, COMMIT e ROLLBACK

- No **PostgreSQL**, uma **transação** é configurada envolvendo os comandos **SQL** da **transação** com os comandos **BEGIN** e **COMMIT**.

```
BEGIN;  
UPDATE contas SET saldo = saldo - 100.00  
    WHERE nome = 'Alice';  
-- etc etc  
COMMIT;
```

- Então uma **transação** SQL de um banco poderia ser descrita da forma acima.
- Se, no meio da **transação**, decidirmos que não queremos confirmar (talvez tenhamos notado que o saldo de Alice foi negativo), podemos emitir o comando **ROLLBACK** em vez de **COMMIT** e todas as nossas atualizações até o momento serão canceladas.

TRANSACTIONS - BEGIN, COMMIT e ROLLBACK

- O **PostgreSQL** trata cada **instrução** SQL como sendo executada dentro de uma **transação**. Se você não emitir um comando **BEGIN**, cada instrução individual terá um **BEGIN** implícito e (se bem-sucedido) um **COMMIT** envolvido nele.
- Um grupo de **instruções** cercado por **BEGIN** e **COMMIT** é chamado de bloco de transação (**transaction block**).
- Algumas bibliotecas emitem comandos **BEGIN** e **COMMIT** automaticamente, para que você possa obter o efeito de blocos de transações sem perguntar.

TRANSACTIONS - SAVEPOINT e ROLLBACK TO

- É possível controlar as **instruções** em uma **transação** de maneira mais granular através do uso de pontos de **SAVEPOINT**.
- Um **SAVEPOINT** permite descartar seletivamente partes da **transação**, enquanto confirma o restante.
- Depois de definir um ponto de salvamento com **SAVEPOINT**, você pode, se necessário, reverter para o ponto de salvamento com **ROLLBACK TO**.
- Depois de reverter para um ponto de salvamento, ele continua a ser definido, então você pode reverter várias vezes para esse ponto de salvamento.

TRANSACTIONS - SAVEPOINT e ROLLBACK TO

```
BEGIN;  
UPDATE contas SET saldo = saldo - 100.00  
    WHERE nome = 'Alice';  
SAVEPOINT my_savepoint;  
UPDATE contas SET saldo = saldo + 100.00  
    WHERE nome = 'Bob';  
-- nesse ponto escolhes o nome errado  
ROLLBACK TO my_savepoint;  
UPDATE contas SET saldo = saldo + 100.00  
    WHERE nome = 'Wally';  
COMMIT;
```

STORED PROCEDURES

- No **PostgreSQL** temos três tipos de **Stored Procedures**, também conhecidas como **Functions**:
 - **Funções em Linguagem SQL**: vimos no slide anterior, não possuem variáveis e estruturas de comando (if, for, etc). Consistem em uma lista de comandos **SQL** (**SELECT**, **INSERT**, **DELETE** ou **UPDATE**). Essas funções são carregadas junto com o serviço do **PostgreSQL**.
 - **Funções de Linguagens Procedurais**: utiliza variáveis e estruturas de comando, além de executar instruções SQL. A linguagem **PL/pgSQL** é a mais utilizada, estruturada e fácil de aprender. A linguagem **PL/PgSQL** é semelhante a linguagem PL/SQL, do Oracle.
 - **Funções Externas**: é possível utilizar funções desenvolvidas em uma linguagem externa, como C++. Principal vantagem é poder contar com o poder de uma linguagem de programação completa. As funções devem ser empacotadas em bibliotecas compartilhadas e registradas no **SGBD**.

PL/pgSQL

- **PL/pgSQL** é uma linguagem procedural carregável desenvolvida para o sistema de banco de dados **PostgreSQL**.
- As **funções** escritas em **PL/pgSQL** podem:
 - aceitar como argumento qualquer tipo de dado escalar ou matriz suportado pelo servidor, e podem retornar como resultado qualquer um destes tipos.
 - aceitar e retornar qualquer tipo composto (tipo linha) especificado por nome.
 - retornam *record*, significando que o resultado é um tipo linha, cujas colunas são determinadas pela especificação no comando que faz a chamada
 - ser declaradas como recebendo ou retornando os tipos polimórficos *anyelement* e *anyarray*.
 - ser declaradas como retornando "set" (conjunto), ou tabela, de qualquer tipo de dado para o qual pode ser retornada uma única instância.
 - ser declaradas como retornando *void* se não produzir nenhum valor de retorno útil.

CREATE OR REPLACE FUNCTION

- Para funções implementadas em **SQL**, o comando **CREATE OR REPLACE FUNCTION** tem a seguinte estrutura:

```
CREATE [OR REPLACE] FUNCTION nome_funcao([parametro 1, parametro 2, parametro n])  
  RETURNS retorno_tipo_dado AS '  
    |      --descrição da função;  
  '  
  LANGUAGE 'SQL';
```

- **CREATE FUNCTION**: define o nome da **função** e seus parâmetros, caso existam.
- **RETURNS**: indica o tipo de dado de retorno da **função**, podendo retornar tipos simples como INT, VARCHAR, etc.
- **Descrição da função**: contém o que será implementado e deve estar entre aspas simples.
- **LANGUAGE 'SQL'**: indica que a linguagem implementada é **SQL**, se fossêmos utilizar PL/pgSQL, deveria ser escrito LANGUAGE 'PL/pgSQL'.

CREATE OR REPLACE FUNCTION

- IMPORTANTE:
 - A **função** é de propriedade do usuário que a criou, caso seja necessário acessar essa função com outro usuário do banco, é necessário executar **GRANT EXECUTE ON nome_funcao TO GROUP nome_usuario**.
- Para criar uma **função** no **PostgreSQL** é necessário salvar o conteúdo dessa função em um arquivo **.sql** em um diretório qualquer do servidor.
- Vamos criar a seguinte **função** como exemplo e salvar em **/tmp/funcao.sql**.

```
CREATE FUNCTION incrementar(INTEGER)
RETURNS INTEGER AS '
    SELECT $1 + 1 ;
'
LANGUAGE 'SQL';
```

CREATE OR REPLACE FUNCTION

- Para carregar a **função** que acabamos de criar no **PostgreSQL** é necessário usar o utilitário **psql**. Execute: **\i /tmp/funcao.sql**.
- Para executar a função, é utilizado a instrução **SELECT**.

```
SELECT incrementar(10);
```

```
incrementar
-----
|          | 11
```

- A **função** executou as instruções passadas no corpo da **função** (**SELECT \$1 + 1**) e o resultado é o parâmetro com valor 10 somado com 1.

CREATE OR REPLACE FUNCTION - SETOF

- Em **funções** SQL é possível retornar um conjunto de **linhas**.
- Acrescentar o parâmetro **SETOF** antes do tipo de dado a ser retornado.

- Observe o exemplo:

```
CREATE FUNCTION quemdeve() RETURNS SETOF INTEGER AS '  
    SELECT clientes.id FROM clientes  
    INNER JOIN contas ON clientes.id = contas.fkcliente  
    INNER JOIN movimentos ON contas.id = movimentos.fkconta  
    GROUP BY clientes.id  
    HAVING SUM(movimentos.credito - movimentos.debito) < 0;  
'  
LANGUAGE 'SQL';
```

- Utilize o comando **SELECT** quemdeve() para executar a função.

```
quemdeve  -----  
          1  
          2
```

- A tabela **pg_proc** no **PostgreSQL** armazena todas as **FUNCTIONS** criadas.

Procedures

- Como você deve saber em todas as versões até o PostgreSQL 10, não era possível criar uma **procedure** no PostgreSQL.
- No **PostgreSQL 11**, **PROCEDURE** foi adicionada como um novo objeto de esquema, que é um objeto semelhante a **FUNCTION**, mas sem um valor de retorno.
- Ao longo dos anos, muitas pessoas estavam ansiosas por ter a funcionalidade e ela foi finalmente adicionada no **PostgreSQL 11**.
- Tradicionalmente, o **PostgreSQL** forneceu todos os meios para escrever funções (chamadas de procedimentos armazenados), em uma função que você não pode executar transações.
- Tudo o que você pode realmente usar são exceções, que são basicamente pontos de salvamento. Dentro de um corpo de função, você não pode simplesmente confirmar uma transação ou abrir uma nova.
- O novo comando **CREATE PROCEDURE** mudará tudo isso e fornecerá uma funcionalidade para executar transações dentro do código procedural.

Benefícios

- Controle de transação que permite **COMMIT** e **ROLLBACK** dentro dos procedimentos.
- Muito útil para a migração do Oracle para o **PostgreSQL**, a nova funcionalidade de **procedure** pode economizar bastante tempo.
- Como você pode ver, existem algumas semelhanças entre **CREATE FUNCTION** e **CREATE PROCEDURE**, portanto as coisas devem ser realmente fáceis para a maioria dos usuários finais.

Como usar ?

- Use **CREATE PROCEDURE** para criar um novo procedimento no **PostgreSQL 11**;
- Isso permitirá que você escreva o procedimento exatamente como outros bancos de dados.
- **PROCEDURE** é quase o mesmo que **FUNCTION** sem um valor de retorno.
- **PROCEDURE** é criado com a instrução **CREATE PROCEDURE** no PostgreSQL 11.
- Diferentemente da instrução **CREATE FUNCTION**, não há cláusula **RETURNS**, cláusula **ROWS** etc.

Sintaxe

```
1 postgres=# \h CREATE PROCEDURE
2 Command:      CREATE PROCEDURE
3 Description:  define a new procedure
4 Syntax:
5 CREATE [ OR REPLACE ] PROCEDURE
6     name ( [ [ argmode ] [ argname ] argtype [ { DEFAULT | = } default_expr ] [, ...]
7     { LANGUAGE lang_name
8       | TRANSFORM { FOR TYPE type_name } [, ... ]
9       | [ EXTERNAL ] SECURITY INVOKER | [ EXTERNAL ] SECURITY DEFINER
10      | SET configuration_parameter { TO value | = value | FROM CURRENT }
11      | AS 'definition'
12      | AS 'obj_file', 'link_symbol'
13    } ...
```

Exemplo

```
1 CREATE PROCEDURE procedure1(INOUT p1 TEXT)
2 AS $$
3 BEGIN
4     RAISE NOTICE 'Procedure Parameter: %', p1 ;
5 END ;
6 $$
7 LANGUAGE plpgsql ;
```

Executando...

- Para executar **PROCEDURE** no **PostgreSQL**, use a instrução **CALL** em vez da instrução **SELECT**. Essa é uma das diferenças entre **PROCEDURE** e **FUNCTION**.

```
1 postgres=# CALL procedure1 (' CREATE PROCEDURE functionality supported in PostgreSQL 11!
2 NOTICE: Procedure Parameter: CREATE PROCEDURE functionality supported in PostgreSQL 11!
3                               p1
4 -----
5 CREATE PROCEDURE functionality supported in PostgreSQL 11!
6 (1 row)
```

Você também pode especificar o nome do parâmetro na instrução **CALL**.

```
postgres=# CALL procedure1 (p1=>'CREATE PROCEDURE functionality supported in PostgreSQL 11!');
NOTICE: Procedure Parameter: CREATE PROCEDURE functionality supported in PostgreSQL 11!
                               p1
-----
CREATE PROCEDURE functionality supported in PostgreSQL 11!
(1 row)
```

Exibir definição de PROCEDURE

Use '\sf' para exibir a definição de PROCEDURE criado.

```
1 postgres=# \sf procedure1
2 CREATE OR REPLACE PROCEDURE public.procedure1(INOUT p1 text)
3     LANGUAGE plpgsql
4 AS $procedure$
5 BEGIN
6     RAISE NOTICE 'Procedure Parameter: %', p1 ;
7 END ;
8 $procedure$
```

Controle de Transação

- Controle de transação que permite **COMMIT** e **ROLLBACK** dentro dos procedimentos.
- **CREATE FUNCTION** não suporta transações dentro da função. Esta é a principal diferença entre **FUNCTION** e **PROCEDURE** no **PostgreSQL**.

Transaction

```
1 CREATE OR REPLACE PROCEDURE transaction_test()  
2 LANGUAGE plpgsql  
3 AS $$  
4 DECLARE  
5 BEGIN  
6     CREATE TABLE committed_table (id int);  
7     INSERT INTO committed_table VALUES (1);  
8     COMMIT;  
9     CREATE TABLE rollback_table (id int);  
10    INSERT INTO rollback_table VALUES (1);  
11    ROLLBACK;  
12 END $$;
```

```
1 postgres=# CALL transaction_test();  
2 CALL
```

Checando a execução...

```
1 postgres=# \d
2          List of relations
3 Schema |          Name          | Type  | Owner
4 -----+-----+-----+-----
5 public | committed_table       | table | postgres
6 (1 row)
7
8 postgres=# SELECT * FROM committed_table;
9 id
10 ----
11      1
12 (1 row)
```

Finalizando...

- **CREATE PROCEDURE** é definitivamente um dos recursos importantes e desejáveis do **PostgreSQL 11**.
- Esse recurso é muito útil para a migração do **Oracle** para o **PostgreSQL** e para muitos casos de uso diferentes.

TRIGGERS

- São operações realizadas de forma espontânea para eventos específicos, onde um evento pode ser um **INSERT**, um **UPDATE** ou até mesmo um **DELETE**.
- **Trigger (gatilho)** é definida com a instrução **CREATE TRIGGER**.
- Deve ser associada com alguma **tabela**, **visualização** ou **tabela estrangeira** e irá executar uma função específica quando determinadas operações forem executadas nessa tabela.
- Pode ser especificado para disparar antes da tentativa de operação em uma linha; ou após a conclusão da operação; ou ao invés da operação.
- Se vários **gatilhos** do mesmo tipo forem definidos para o mesmo evento, eles serão disparados em ordem alfabética por nome.

TRIGGERS

- Um **gatilho** que está marcado **FOR EACH ROW** é chamado uma vez para cada linha que a operação modifica.
- Um **gatilho** marcado como **FOR EACH STATEMENT** é executado apenas uma vez para qualquer operação, independentemente de quantas linhas ele modifica.
- Um **gatilhos** que é especificado para disparar **INSTEAD OF** do evento de **gatilho** deve ser marcado **FOR EACH ROW** e só pode ser definido nas **visualizações**.
- Um **gatilho** marcado com **BEFORE** e **AFTER** de uma visualização deve ser marcado como **FOR EACH STATEMENT**.

TRIGGERS

Quando	Evento	Row-level	Statement-level
BEFORE	INSERT/UPDATE/DELETE	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables
AFTER	INSERT/UPDATE/DELETE	Tables and foreign tables	Tables, views, and foreign tables
	TRUNCATE	—	Tables
INSTEAD OF	INSERT/UPDATE/DELETE	Views	—
	TRUNCATE	—	—

TRIGGERS

- Alguns exemplos de **gatilhos**:
- 1. Execute uma **função** name_function criada no **PostgreSQL** sempre que uma **linha** da **tabela** name_table estiver prestes a ser atualizada:

```
CREATE TRIGGER name_trigger  
  BEFORE UPDATE ON name_table  
  FOR EACH ROW  
  EXECUTE FUNCTION name_function();
```

- Execute uma função view_insert_row para cada **linha** para inserir **linhas** nas **tabelas** subjacentes a uma **visualização**:

```
CREATE TRIGGER view_insert  
  INSTEAD OF INSERT ON my_view  
  FOR EACH ROW  
  EXECUTE FUNCTION view_insert_row();
```

Questão

Ano: 2018 **Banca:** CESPE **Órgão:** STM **Prova:** CESPE - 2018 - STM - Analista Judiciário -
Análise de Sistemas

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

Questão

Ano: 2018 **Banca:** CESPE **Órgão:** STM **Prova:** CESPE - 2018 - STM - Analista Judiciário -
Análise de Sistemas

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

Justificativa:

Quando uma função que retorna o tipo *trigger* é executada, o PostgreSQL cria algumas variáveis especiais na memória. Essas variáveis podem ser usadas no corpo das nossas funções.

Pode ser consultada uma lista em <https://www.postgresql.org/docs/9.6/static/plpgsql-trigger.html>

Questão

Ano: 2018 Banca: CESPE Órgão: STM Prova: CESPE - 2018 - STM - Analista
Judiciário - Análise de Sistemas

Julgue o item subsequente, a respeito do Postgres 9.6.

```
CREATE FUNCTION add(integer, integer) RETURNS integer
AS 'select $1 + $2;'
LANGUAGE SQL
IMMUTABLE
RETURNS NULL ON NULL INPUT;
```

Nas instruções seguintes, a palavra-chave IMMUTABLE indica que a função criada não pode modificar o banco de dados.

Certo

Errado

Questão

Ano: 2018 **Banca:** CESPE **Órgão:** STM **Prova:** CESPE - 2018 - STM - Analista
Judiciário - Análise de Sistemas

Julgue o item subsequente, a respeito do Postgres 9.6.

```
CREATE FUNCTION add(integer, integer) RETURNS integer  
AS 'select $1 + $2;'  
LANGUAGE SQL  
IMMUTABLE  
RETURNS NULL ON NULL INPUT;
```

Nas instruções seguintes, a palavra-chave IMMUTABLE indica que a função criada não pode modificar o banco de dados.

 Certo
Errado

Justificativa:

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.

INDEXES

- A criação de um **índice** em uma **tabela**, possui o objetivo de facilitar a localização de **linhas** correspondents de uma **consulta** qualquer.
- Para criar um **índice** utilize a instrução **CREATE INDEX**.
- Para remover um **índice** utiliza a instrução **DROP INDEX**.

```
CREATE INDEX test1_id_index ON test1 (id);
```

- Índices podem ser adicionados e removidas de uma tabela a qualquer momento.
- Depois que um **índice** é criado, nenhuma intervenção adicional é necessária.
- O sistema atualizará o **índice** quando a **tabela** for modificada e o usará nas **consultas** quando achar que isso será mais eficiente do que uma varredura sequencial da **tabela**.

INDEXES

- O **PostgreSQL** suporta os seguintes tipos de **índices**: **B-tree**, **Hash**, **GiST**, **SP-GiST**, **GIN** e **BRIN**. Cada um utiliza um algoritmo diferente que é mais adequado para diferentes tipos de consultas
- Um **índice** pode ser definido em mais de uma **coluna** de uma **tabela**. Atualmente, apenas os tipos de **índice B-tree**, **GiST**, **GIN** e **BRIN** suportam **índices** de várias **colunas**.
- O **PostgreSQL** possui a capacidade de combinar vários **índices** para lidar com casos em que não são possíveis de implementar varreduras de **índice** único.
- Os **índices** também podem ser usados para impor exclusividade ao valor de uma **coluna** específica, somente os **índices B-tree** podem ser declarados únicos.

postgresql.conf e pg_hba.conf

- As principais configurações de um servidor PostgreSQL são realizadas nos arquivos **postgresql.conf** e **pg_hba.conf**.
- Os parâmetros de configuração no arquivo **postgresql.conf** especificam o comportamento do servidor com relação a auditoria, autenticação, criptografia e outros pontos.
- O arquivo **pg_hba.conf** gerencia quais máquinas, ou outros servidores, terão acesso ao **PostgreSQL**, além da sua forma de autenticação (trust, md5, crypt etc)

BACKUP - pg_dump

- Para exportar o banco de dados **PostgreSQL**, precisamos usar a ferramenta **pg_dump**, que despeja todo o conteúdo de um banco de dados selecionado em um único arquivo.
- Precisamos executar o **pg_dump** na linha de comando no computador em que o banco de dados está armazenado.

```
pg_dump -U db_user -W -F t db_name > /path/to/your/file/dump_name.tar
```

- Para ver uma lista de todas as opções disponíveis, use **pg_dump -?**

RESTORE - pg_restore

- Se você escolher o formato personalizado, de diretório ou de arquivamento ao criar um arquivo de backup, precisará usar o `pg_restore` para restaurar seu banco de dados:

```
pg_restore -d db_name /path/to/your/file/dump_name.tar -c -U db_user
```

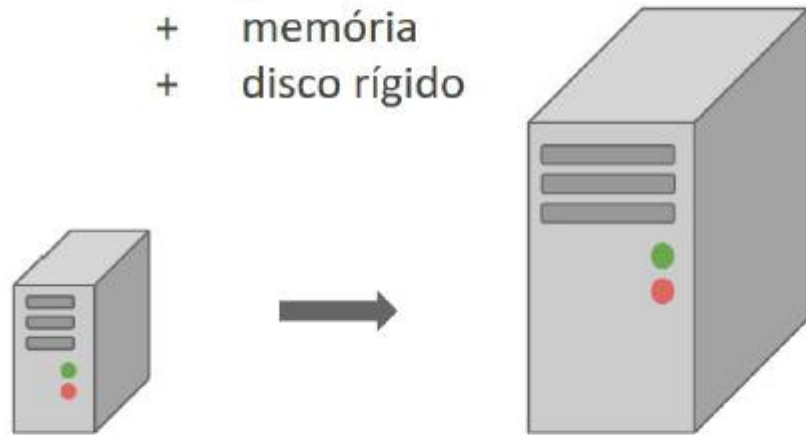
- Use **`pg_restore -?`** para obter a lista completa de opções disponíveis.

Escalabilidade

Escalabilidade vertical

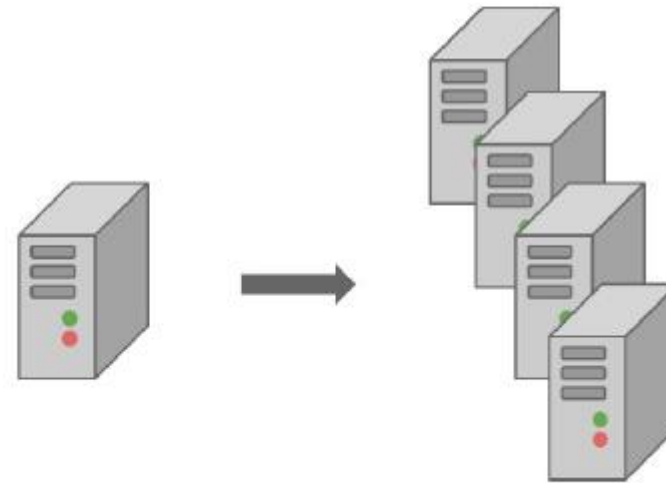
(Adição de capacidade para um único recurso)

- + processador
- + memória
- + disco rígido



Escalabilidade horizontal

(Adição de recursos ao ambiente computacional)



CREATE USER e DROP USER

- Existem **usuários** que são independentes do sistema operacional, que servem apenas para manipular objetos de um banco de dados.
- No **PostgreSQL** é possível criar um novo usuário no banco de dados utilizando a instrução **CREATE USER**.
- Atualmente, a instrução **CREATE USER** é apenas um alias para o comando **CREATE ROLE**.

```
CREATE USER nome [ [ WITH ] opção [ ... ] ];
```

- Uma vez o **usuário** criado, o comando **DROP USER** irá remover o **usuário**. Entretanto as **tabelas**, **visualizações** e outros objetos que pertençam ao **usuário** não são removidas.

```
DROP USER nome_usuario;
```

ALTER USER

- Em alguns casos, pode ser mais conveniente alterar um **usuário**, do que remover para criar um novo, seja para alterar sua senha ou para incluir a permissão de criação de **usuários** ou retirá-la, caso exista etc.

- O comando para modificar um **usuário** é o **ALTER USER**.

```
ALTER USER nome [ [ WITH ] opção [ ... ] ]
```

- Um exemplo exibido abaixo, altera o nome do **usuário**.

```
ALTER USER nome RENAME TO novo_nome
```

- É importante salientar que apenas um **super-usuário** pode mudar o nome de outro **usuário**.

CREATE GROUP e DROUP GROUP

- No **PostgreSQL**, os **grupos** são uma forma lógica de juntar **usuários** para facilitar o gerenciamento de seus privilégios.
- Privilégios podem ser concedidos, ou revogados, para o **grupo** como um todo.
- Para criar um **grupo**, utilize o comando **CREATE GROUP**.

```
CREATE GROUP nome_do_grupo;
```

- Uma vez criado um **grupo**, é possível removê-lo. Utilize o comando **DROP GROUP**.

```
DROP GROUP nome_do_grupo;
```

ALTER GROUP

- Assim como podemos modificar um **usuário** criado no banco de dados, podemos, da mesma forma, modificar um **grupo**.
- O comando para modificar um **grupo** é o **ALTER GROUP**.
- Com o **ALTER GROUP** podemos adicionar, remover e alterar **usuários** de um **grupo**, além de poder alterar o próprio nome do **grupo**.

```
ALTER GROUP nome_do_grupo ADD USER nome_do_usuario;
```

```
ALTER GROUP nome_do_grupo DROP USER nome_do_usuario;
```

```
ALTER GROUP nome_do_grupo RENAME TO novo_nome;
```

CREATE ROLE

- O comando **CREATE ROLE** adiciona um novo **papel** (role) ao agrupamento de bancos de dados do **PostgreSQL**.
- O **papel** é uma **entidade** que pode possuir objetos do banco de dados e possuir **privilégios** do banco de dados.
- Pode ser considerado como um "**usuário**", um "**grupo**", ou ambos, dependendo da forma que for utilizado.
- O comando **CREATE ROLE** é o substituto dos comandos **CREATE USER** e **CREATE GROUP** por possuir mais recursos que os mesmos.

```
CREATE ROLE nome [ [WITH] opção [...] ] ;
```

- Cuidado com o **privilégio CREATE ROLE**.
- Não existe conceito de herança para os **privilégios** do **papel** com **CREATE ROLE**.

DROP ROLE e ALTER ROLE

- Por mais que um **papel** não possua um determinado **privilégio**, mas possua permissão para criar outros **papéis**, é possível criar um **papel** com **privilégios** diferentes do próprio **papel**, exceto **privilégios** de **super-usuário**.
- Para remover um papel, execute o comando **DROP ROLE**.

```
DROP ROLE [ IF EXISTS ] nome [, ...]
```

- O **papel** não poderá ser removido se ainda estiver sendo referenciado em qualquer banco de dados.
- Antes de remover o **papel** é necessário remover todos os objetos pertencentes a esse **papel**, ou mudar o dono, além de revogar todos os **privilégios**.
- O comando ALTER ROLE altera os atributos de um papel do PostgreSQL.

```
ALTER ROLE nome [ [ WITH ] opção [ ... ] ]
```

PRIVILÉGIOS

- Quando um objeto do banco de dados é criado, é atribuído um **dono**.
- O **dono** é o **usuário** que executou o comando de criação do objeto.
- Para mudar o **dono** de uma **tabela**, **índice**, **sequência** ou **visão** deve ser utilizado o comando **ALTER TABLE**.

```
ALTER TABLE nome_da_tabela OWNER TO novo_dono
```

- Por padrão, somente o **dono**, ou um **super-usuário**, pode fazer alterações no objeto.
- Para permitir que outros **usuários** façam alterações, existe o comando **GRANT**.
- O **GRANT** possui duas funcionalidades básicas:
 - Conceder **privilégios** para um objeto do banco de dados;
 - Conceder o **privilégio** de ser membro de um **papel**.

PRIVILÉGIOS

- Existem vários **privilégios** distintos: **SELECT**, **INSERT**, **UPDATE**, **DELETE**, **RULE**, **REFERENCES**, **TRIGGER**, **CREATE**, **TEMPORARY**, **EXECUTE**, **USAGE** e **ALL PRIVILEGES**.
- Em alguns casos é necessário revogar alguns privilégios de acesso a usuários e grupos de usuários. Para isso, execute o comando **REVOKE**.
- Os **super-usuários** do banco de dados podem acessar todos os objetos, independentemente dos **privilégios** definidos para o objeto.
- Existe uma opção chamada **SECURITY DEFINER** que é utilizada na criação de **funções**.
- Serve para permitir que todos os **usuários** acessem a função, que tiver a opção **SECURITY DEFINER**, com a mesma permissão de quem criou.

SQL

- A linguagem **SQL** é dividida em quatro tipos de instruções, com base em sua funcionalidade: DDL, DML, DCL e TCL.
 - DDL-Data Definition Language
 - DML-Data Manipulation Language
 - DCL-Data Control Language
 - TCL-Transactional Control Language
- **DDL: DROP, RENAME, CREATE, ALTER, TRUNCATE, etc.**
- **DML: SELECT, UPDATE, INSERT, DELETE.**
- **DCL: GRANT, REVOKE.**
- **TCL: COMMIT, ROLLBACK, SAVEPOINT.**

Questão

Ano: 2019 **Banca:** CESPE **Órgão:** TJ-AM **Prova:** CESPE - 2019 - TJ-AM -
Analista Judiciário - Analista de Sistemas

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.

Certo

Errado

Questão

Ano: 2019 **Banca:** CESPE **Órgão:** TJ-AM **Prova:** CESPE - 2019 - TJ-AM -
Analista Judiciário - Analista de Sistemas

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.

Certo

 Errado

Justificativa:

ALTER ROLE e DROP ROLE

Questão

Ano: 2018 **Banca:** FCC **Órgão:** DPE-AM **Prova:** FCC - 2018 - DPE-AM - Analista em Gestão Especializado de Defensoria - Analista de Sistema

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*.

Questão

Ano: 2018 **Banca:** FCC **Órgão:** DPE-AM **Prova:** FCC - 2018 - DPE-AM - Analista em Gestão Especializado de Defensoria - Analista de Sistema

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*.

Justificativa:

Função OCTET_LENGTH Retorna o tamanho da string especificada como número de bytes.

Questão

Ano: 2018 **Banca:** FCC **Órgão:** DPE-AM **Prova:** FCC - 2018 - DPE-AM - Analista em Gestão Especializado de Defensoria - Analista de Sistema


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.

Questão

Ano: 2018 **Banca:** FCC **Órgão:** DPE-AM **Prova:** FCC - 2018 - DPE-AM - Analista em Gestão Especializado de Defensoria - Analista de Sistema

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.

Justificativa:

pg_dump:

Terminal de comando

Realiza cópia de segurança do PostgreSQL

psql:

Ferramenta de administração do PostgreSQL em modo texto (terminal de comandos) do mesmo fornecedor do banco de dados PostgreSQL. Por padrão, este programa já é disponibilizado junto com o servidor PostgreSQL.

pg_restore:

Restaura um banco de dados PostgreSQL a partir de um arquivo gerado pelo pg_dump.

Questão

Ano: 2017 **Banca:** CESPE **Órgão:** TRF - 1ª REGIÃO **Prova:** CESPE - 2017 - TRF - 1ª REGIÃO - Analista Judiciário - Informática

Considerando o SGBD Postgresql, julgue o próximo item.

```
begin;  
    savepoint primeiro;  
    create table tbl_conceito (id int, nome  
varchar);  
    rollback to savepoint primeiro;  
commit;
```

Após a execução do trecho de código SQL a seguir, a tabela tbl_conceito não será criada.

Certo

Errado

Questão

Ano: 2017 **Banca:** CESPE **Órgão:** TRF - 1ª REGIÃO **Prova:** CESPE - 2017 - TRF - 1ª REGIÃO - Analista Judiciário - Informática

Considerando o SGBD Postgresql, julgue o próximo item.

```
begin;  
    savepoint primeiro;  
    create table tbl_conceito (id int, nome  
varchar);  
    rollback to savepoint primeiro;  
commit;
```

Após a execução do trecho de código SQL a seguir, a tabela tbl_conceito não será criada.



Certo

Errado

Justificativa:

O roolback trará tudo ao estado inicial do savepoint.

Questão

Ano: 2017 **Banca:** FCC **Órgão:** TST **Prova:** FCC - 2017 - TST - Analista Judiciário – Suporte em Tecnologia da Informação

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;`

Questão

Ano: 2017 **Banca:** FCC **Órgão:** TST **Prova:** FCC - 2017 - TST - Analista Judiciário – Suporte em Tecnologia da Informação

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;`

Justificativa:

A função `COALESCE` retorna o primeiro dos seus argumentos que não é nulo. Nulo é retornado somente se todos os argumentos forem nulos. Então se *salary* for nulo, ele irá retornar o 0, próximo argumento não nulo.

Questão

Ano: 2017 **Banca:** CESPE **Órgão:** TRT - 7ª Região (CE) **Prova:** CESPE - 2017 - TRT - 7ª Região (CE) - Analista Judiciário - Tecnologia da Informação


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.

Questão

Ano: 2017 **Banca:** CESPE **Órgão:** TRT - 7ª Região (CE) **Prova:** CESPE - 2017 - TRT - 7ª Região (CE) - Analista Judiciário - Tecnologia da Informação

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.

Justificativa:

No arquivo pg_hba.conf, é possível restringir o acesso ao banco de dados PostgreSQL por IP.

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

<https://www.postgresql.org/docs/9.3/static/auth-pg-hba-conf.html>

Questão

Ano: 2018 **Banca:** FGV **Órgão:** Câmara de Salvador - BA **Prova:** FGV - 2018 - Câmara de Salvador - BA - Analista de Tecnologia da Informação

Analise o script a seguir, no âmbito do PostgreSQL.

```
CREATE TABLE T ( chave serial NOT NULL PRIMARY KEY, dados json NOT NULL ); INSERT INTO T (dados) VALUES ('{ "nome": "Maria", "notas": {"disciplina": "Fisica", "nota": 10}}'), ('{ "nome": "Pedro", "notas": {"disciplina": "Calculo", "nota": 9}}');
```

O comando SQL que produz corretamente uma lista dos alunos, com a matrícula, nome e respectivas disciplinas e notas é:

- a) `SELECT chave matricula, dados.nome AS aluno, dados.notas.disciplina disc, dados.notas.nota grau FROM T;`
- b) `SELECT chave matricula, dados!'nome' AS aluno, dados!'notas'>>'disciplina' disc, dados!'notas'>>'nota' grau FROM T;`
- c) `SELECT chave matricula, dados -> 'nome' AS aluno, dados -> 'notas' ->> 'disciplina' disc, dados -> 'notas' ->> 'nota' grau FROM T;`
- d) `SELECT chave matricula, nome AS aluno, notas ->> 'disciplina' disc, notas ->> 'nota' grau FROM T;`
- e) `SELECT chave matricula, dados -> 'nome' AS aluno, dados -> 'notas.disciplina' disc, dados -> 'notas.nota' grau FROM T;`


Questão

Ano: 2018 **Banca:** FGV **Órgão:** Câmara de Salvador - BA **Prova:** FGV - 2018 - Câmara de Salvador - BA - Analista de Tecnologia da Informação

Analise o script a seguir, no âmbito do PostgreSQL.

```
CREATE TABLE T ( chave serial NOT NULL PRIMARY KEY, dados json NOT NULL ); INSERT INTO T (dados) VALUES ({
"nome": "Maria", "notas": {"disciplina":"Fisica","nota": 10}}), ({ "nome": "Pedro", "notas":
{"disciplina":"Calculo","nota": 9}});
```

O comando SQL que produz corretamente uma lista dos alunos, com a matrícula, nome e respectivas disciplinas e notas é:

- a) `SELECT chave matricula, dados.nome AS aluno, dados.notas.disciplina disc, dados.notas.nota grau FROM T;`
- b) `SELECT chave matricula, dados!'nome' AS aluno, dados!'notas'>>'disciplina' disc, dados!'notas'>>'nota' grau FROM T;`
-  c) `SELECT chave matricula, dados -> 'nome' AS aluno, dados -> 'notas' ->> 'disciplina' disc, dados -> 'notas' ->> 'nota' grau FROM T;`
- d) `SELECT chave matricula, nome AS aluno, notas ->> 'disciplina' disc, notas ->> 'nota' grau FROM T;`
- e) `SELECT chave matricula, dados -> 'nome' AS aluno, dados -> 'notas.disciplina' disc, dados -> 'notas.nota' grau FROM T;`

Justificativa:


Inserção de Dados JSON

Questão

Ano: 2021 **Banca:** VUNESP **Órgão:** TJM-SP **Prova:** VUNESP - 2021 - TJM-SP - Analista em Comunicação e Processamento de Dados Judiciário (Analista de Redes)

No sistema gerenciador de bancos de dados PostgreSQL (versão 12), o comando para recuperar um banco de dados de nome “primeiro”, a partir do arquivo “green”, criado pelo comando pg_dump é:

Alternativas

- a) pgsql green > primeiro
- b) pgsql primeiro (green)
- c) psql primeiro (green)
- d) psql_pg primeiro \$green
-  e) psql primeiro < green

Questão

Ano: 2021 Banca: FCC Órgão: TJ-SC Prova: FCC - 2021 - TJ-SC - Analista de Sistemas

No PostgreSQL, para fazer backup full de todo o cluster, incluindo schemas, bancos de dados, tabelas, templates, usuários e roles, assim como suas permissões, utiliza-se o comando cuja sintaxe está abaixo:

<comando> -p <porta> -U <usuario> -h <host> -f <path/nome_arquivo_backup>

Na sintaxe apresentada, <comando> refere-se à

Alternativas

- a) SQL-dump.
- b) pg_dump.
- c) psql.
- ☒ d) pg_dumpall.
- e) pg_rmain.

Questão

Ano: 2021 Banca: VUNESP Órgão: TJM-SP Prova: VUNESP - 2021 - TJM-SP - Técnico em Comunicação e Processamento de Dados Judiciário (Desenvolvedor)

O sistema gerenciador de bancos de dados PostgreSQL (versão 12) possui diversos catálogos de sistema, sendo que o catálogo que armazena informações sobre a hierarquia de heranças entre tabelas é

Alternativas

- a) pg_policy.
- b) pg_depend.
- ☒ c) pg_inherits.
- d) pg_trigger.
- e) pg_database (bases de dados disponíveis).

PL/pgSQL

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

```
[ <<rótulo>> ]  
[ DECLARE  
    declarações ]  
BEGIN  
    instruções  
END;
```

- Todas as declarações e instruções dentro do bloco devem ser terminadas por ponto-e-vírgula.
- Um bloco contido dentro de outro bloco deve conter um ponto-e-vírgula após o END, conforme mostrado acima; entretanto, o END final que conclui o corpo da função não requer o ponto-e-vírgula.



PL/pgSQL

- Todas as palavras chave e identificadores podem ser escritos misturando letras maiúsculas e minúsculas.
- As letras dos identificadores são convertidas implicitamente em minúsculas, a menos que estejam entre aspas.

```
CREATE FUNCTION sum(int[]) RETURNS int8 AS $$  
DECLARE  
    s int8 := 0;  
    x int;  
BEGIN  
    FOREACH x IN ARRAY $1  
    LOOP  
        s := s + x;  
    END LOOP;  
    RETURN s;  
END;  
$$ LANGUAGE plpgsql;
```

PL/pgSQL

- Existem dois tipos de comentários no **PL/pgSQL**. O hífen duplo (--) começa um comentário que se estende até o final da linha. O /* começa um bloco de comentário que se estende até a próxima ocorrência de */.
- Os blocos de comentário não podem ser aninhados, mas comentários de hífen duplo podem estar contidos em blocos de comentário, e os hífen duplos escondem os delimitadores de bloco de comentário /* e */.
- Qualquer instrução na seção de instruções do bloco pode ser um **sub-bloco**. Os sub-blocos podem ser utilizados para agrupamento lógico, ou para limitar o escopo de variáveis a um pequeno grupo de instruções.

Exemplo

```
CREATE FUNCTION func_escopo() RETURNS integer AS $$
DECLARE
    quantidade integer := 30;
BEGIN
    RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 30
    quantidade := 50;
    --
    -- Criar um sub-bloco
    --
    DECLARE
        quantidade integer := 80;
    BEGIN
        RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 80
    END;

    RAISE NOTICE 'Aqui a quantidade é %', quantidade; -- A quantidade aqui é 50

    RETURN quantidade;
END;
$$ LANGUAGE plpgsql;
```

```
=> SELECT func_escopo();
```

```
NOTA: Aqui a quantidade é 30
```

```
NOTA: Aqui a quantidade é 80
```

```
NOTA: Aqui a quantidade é 50
```

```
func_escopo
```

```
-----
```

```
50
```

```
(1 linha)
```

- O **BEGIN/END** da **PL/pgSQL** é apenas para agrupamento; não começam nem terminam transações.
- Os procedimentos de funções e de gatilhos são sempre executados dentro da transação estabelecida pelo comando externo — não podem iniciar ou efetivar transações, porque não haveria contexto para estas serem executadas.

Declarações

- Todas as variáveis utilizadas em um bloco devem ser declaradas na seção de declarações do bloco (A única exceção é a variável de laço do **FOR** interagindo sobre um intervalo de valores inteiros, que é automaticamente declarada como sendo do tipo inteiro).
- As variáveis da linguagem **PL/pgSQL** podem possuir qualquer tipo de dado da linguagem SQL, como *integer*, *varchar* e *char*.
- Abaixo seguem alguns exemplos de declaração de variáveis:

```
id_usuario    integer;  
quantidade    numeric(5);  
url           varchar;  
minha_linha   nome_da_tabela%ROWTYPE;  
meu_campo     nome_da_tabela.nome_da_coluna%TYPE;  
uma_linha     RECORD;
```

Declarações

- A cláusula **DEFAULT**, se for fornecida, especifica o valor inicial atribuído à variável quando o processamento entra no bloco. Se a cláusula **DEFAULT** não for fornecida, então a variável é inicializada com o valor nulo do SQL.
- A opção **CONSTANT** impede que seja atribuído valor a variável e, portanto, seu valor permanece constante pela duração do bloco. Se for especificado **NOT NULL**, 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.
- O valor padrão é avaliado toda vez que a execução entra no bloco. Portanto, por exemplo, atribuir **now()** a uma variável do tipo **timestamp** faz com que a variável possua a data e hora da chamada corrente à função, e não de quando a função foi pré-compilada.
- ***Exemplos:***

```
quantidade integer DEFAULT 32;  
url         varchar := 'http://meu-site.com';  
id_usuario  CONSTANT integer := 10;
```

Cópia de tipo

- A expressão **%TYPE** fornece o tipo de dado da variável ou da coluna da tabela. Pode ser utilizada para declarar variáveis que armazenam valores do banco de dados.
- Por exemplo, supondo que exista uma coluna chamada **id_usuario** na tabela **usuarios**, para declarar uma variável com o mesmo tipo de dado de **usuarios.id_usuario** deve ser escrito:
 - ***variável%TYPE***

```
id_usuario usuarios.id_usuario%TYPE;
```

- Utilizando **%TYPE** não é necessário conhecer o tipo de dado da estrutura.

Tipos linha

- Uma variável de tipo composto é chamada de variável *linha* (ou variável *tipo-linha*).
- Este tipo de variável pode armazenar toda uma linha de resultado de um comando **SELECT** ou **FOR**, desde que o conjunto de colunas do comando corresponda ao tipo declarado para a variável.
 - *nome nome_da_tabela%ROWTYPE;*
 - *nome nome_do_tipo_composto;*
- Os campos individuais do valor linha são acessados utilizando a notação usual de ponto como, por exemplo, **variável_linha.campo**.

FOR

- Esta forma do **FOR** cria um laço que interage num intervalo de valores inteiros.
- A variável ***nome*** é definida automaticamente como sendo do tipo ***integer***, e somente existe dentro do laço.
- As duas expressões que fornecem o limite inferior e superior do intervalo são avaliadas somente uma vez, ao entrar no laço.
- Normalmente o passo da interação é 1, mas quando **REVERSE** é especificado se torna -1.
- Alguns exemplos de laços FOR inteiros:

```
FOR i IN 1..10 LOOP
    -- algum processamento
    RAISE NOTICE 'i é %', i;
END LOOP;

FOR i IN REVERSE 10..1 LOOP
    -- algum processamento
END LOOP;
```

Aliases para parâmetros de função

- Os parâmetros passados para as funções recebem como nome os identificadores \$1, \$2, etc.
- Opcionalmente, para melhorar a legibilidade do código, podem ser declarados **aliases** para os nomes dos parâmetros \$*n*. Para fazer referência ao valor do parâmetro, pode ser utilizado tanto o aliás quanto o identificador numérico.
- Existem duas maneiras de criar um aliás. A forma preferida é fornecer nome ao parâmetro no comando **CREATE FUNCTION** como, por exemplo:

```
CREATE FUNCTION taxa_de_venda(subtotal real) RETURNS real AS $$  
BEGIN  
    RETURN subtotal * 0.06;  
END;  
$$ LANGUAGE plpgsql;
```

Aliases para parâmetros de função

- A outra maneira, que era a única disponível antes da versão 8.0 do PostgreSQL, é declarar explicitamente um aliás utilizando a sintaxe de declaração:

nome ALIAS FOR \$n;

```
CREATE FUNCTION taxa_de_venda(real) RETURNS real AS $$  
DECLARE  
    subtotal ALIAS FOR $1;  
BEGIN  
    RETURN subtotal * 0.06;  
END;  
$$ LANGUAGE plpgsql;
```

Obtenção do status do resultado

- Existem diversas maneiras de determinar o efeito de um comando. O primeiro método é utilizar o comando **GET DIAGNOSTICS**, que possui a forma:
 - ***GET DIAGNOSTICS** variável = item [, ...] ;*
- O segundo método para determinar os efeitos de um comando é verificar a variável especial **FOUND**, que é do tipo *boolean*. A variável **FOUND** é iniciada como falso dentro de cada chamada de função **PL/pgSQL**.
- **FOUND** é uma variável local dentro de cada função **PL/pgSQL**; qualquer mudança feita na mesma afeta somente a função corrente.

Cursors

- Em vez de executar toda a consulta de uma vez, é possível definir um ***cursor*** encapsulando a consulta e, depois, ler umas poucas linhas do resultado da consulta de cada vez.
- Um dos motivos de se fazer desta maneira, é para evitar o uso excessivo de memória quando o resultado contém muitas linhas (Entretanto, normalmente não há necessidade dos usuários da linguagem **PL/pgSQL** se preocuparem com isto, uma vez que os laços **FOR** utilizam internamente um cursor para evitar problemas de memória, automaticamente).
- Uma utilização mais interessante é retornar a referência a um cursor criado pela função, permitindo a quem chamou ler as linhas. Esta forma proporciona uma maneira eficiente para a função retornar conjuntos grandes de linhas.

Declaração de variável cursor

- Todos os acessos aos cursores na linguagem **PL/pgSQL** são feitos através de variáveis cursor, que sempre são do tipo de dado especial ***refcursor***.
- Uma forma de criar uma variável cursor é simplesmente declará-la como sendo do tipo ***refcursor***.
- Outra forma é utilizar a sintaxe de declaração de cursor, cuja forma geral é:

```
nome CURSOR [ ( argumentos ) ] FOR comando ;
```

- Os ***argumentos***, quando especificados, são uma lista separada por vírgulas de pares ***nome tipo_de_dado***. Esta lista define nomes a serem substituídos por valores de parâmetros na consulta.
- Os valores verdadeiros que substituirão estes nomes são especificados posteriormente, quando o cursor for aberto.

```
DECLARE
  curs1 refcursor;
  curs2 CURSOR FOR SELECT * FROM tenk1;
  curs3 CURSOR (chave integer) IS SELECT * FROM tenk1 WHERE unico1 = chave;
```

Erros e mensagens

- A instrução **RAISE** é utilizada para gerar mensagens informativas e causar erros.

```
RAISE nível 'formato' [, variável [, ...]];
```

- Os níveis possíveis são **DEBUG**, **LOG**, **INFO**, **NOTICE**, **WARNING**, e **EXCEPTION**. O nível **EXCEPTION** causa um erro (que normalmente interrompe a transação corrente); os outros níveis apenas geram mensagens com diferentes níveis de prioridade.
- Se as mensagens de uma determinada prioridade são informadas ao cliente, escritas no *log* do servidor, ou as duas coisas, é controlado pelas variáveis de configuração log_min_messages e client_min_messages.
- Dentro da cadeia de caracteres de formatação, o caractere % é substituído pela representação na forma de cadeia de caracteres do próximo argumento opcional. Deve ser escrito %% para produzir um % literal. Deve ser observado que atualmente os argumentos opcionais devem ser variáveis simples, e não expressões, e o formato deve ser um literal cadeia de caracteres simples.
- Neste exemplo o valor de **v_job_id** substitui o caractere % na cadeia de caracteres:

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

- Este exemplo interrompe a transação com a mensagem de erro fornecida:

```
RAISE EXCEPTION 'ID inexistente --> %', id_usuario;
```

Questão 1

Ano: 2020 **Banca:** CESPE **Órgão:** TJ-PA **Prova:** CESPE - 2020 - TJ-PA - Analista Judiciário - Programador

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.

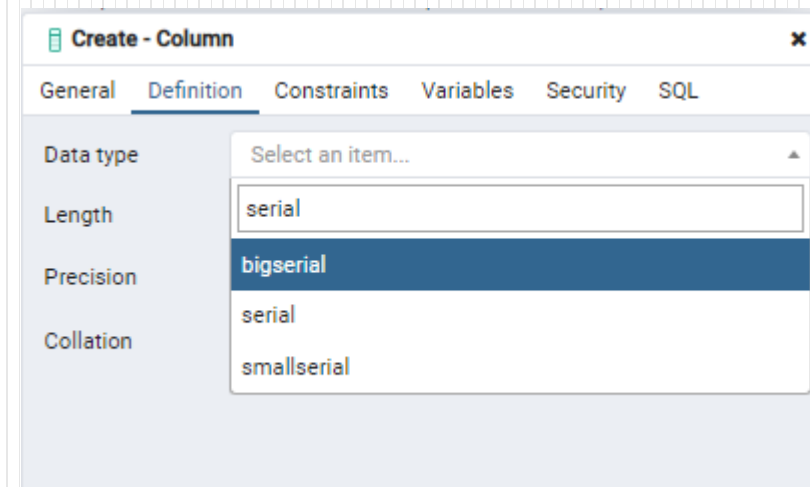
Questão 1

Ano: 2020 **Banca:** CESPE **Órgão:** TJ-PA **Prova:** CESPE - 2020 - TJ-PA - Analista Judiciário - Programador

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.

LETRA E



Justificativa:

Tipos de dados do PostgreSQL para auto numeração.

Questão 1

Behind the scenes, the following statement:

```
1 CREATE TABLE table_name(  
2     id SERIAL  
3 );
```

is equivalent to the following statements:

```
1 CREATE SEQUENCE table_name_id_seq;  
2  
3 CREATE TABLE table_name (  
4     id integer NOT NULL DEFAULT nextval('table_name_id_seq')  
5 );  
6  
7 ALTER SEQUENCE table_name_id_seq  
8 OWNED BY table_name.id;
```

Name	Storage Size	Range
SMALLSERIAL (v 9.2)	2 bytes	1 to 32,767
SERIAL	4 bytes	1 to 2,147,483,647
BIGSERIAL	8 bytes	1 to 9,223,372,036,854,77 5 807

Examples

Create an ascending sequence called `serial`, starting at 101:

```
CREATE SEQUENCE serial START 101;
```

Select the next number from this sequence:

```
SELECT nextval('serial');  
  
nextval  
-----  
101
```

Select the next number from this sequence:

```
SELECT nextval('serial');  
  
nextval  
-----  
102
```

Use this sequence in an `INSERT` command:

```
INSERT INTO distributors VALUES (nextval('serial'), 'nothing');
```

Questão 2

Ano: 2019 **Banca:** CESPE **Órgão:** TCE-RO **Prova:** CESPE - 2019 - TCE-RO - Analista de Tecnologia da Informação - Desenvolvimento de Sistemas

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.

Questão 2

Ano: 2019 **Banca:** CESPE **Órgão:** TCE-RO **Prova:** CESPE - 2019 - TCE-RO - Analista de Tecnologia da Informação - Desenvolvimento de Sistemas

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.

LETRA D

- 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.

Justificativa:

- a) **ERRADA** A redação da alternativa está meio confusa. O comando `CREATE INDEX` constrói um índice (e não uma linha de índice) de acordo com uma coluna ou conjunto de colunas de uma tabela específica.
- 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**. Um campo de índice pode sim ser uma expressão calculada a partir de valores de uma ou mais colunas. Isso é útil quando se deseja ter acesso rápido a um dado que seja calculados em função da(s) coluna(s).
- d) **CORRETA**. *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*).

Questão 3

Ano: 2018 **Banca:** CESPE **Órgão:** ABIN **Prova:** CESPE - 2018 - ABIN - Oficial
Técnico de Inteligência - Área 9

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

Questão 3

Ano: 2018 **Banca:** CESPE **Órgão:** ABIN **Prova:** CESPE - 2018 - ABIN - Oficial
Técnico de Inteligência - Área 9

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

ERRADO

Justificativa:

O psql é um cliente no modo terminal do PostgreSQL. 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 ERRADA.

Questão 4

Ano: 2018 **Banca:** CESPE **Órgão:** ABIN **Prova:** CESPE - 2018 - ABIN - Oficial Técnico de Inteligência - Área 8

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

Questão 4

Ano: 2018 **Banca:** CESPE **Órgão:** ABIN **Prova:** CESPE - 2018 - ABIN - Oficial Técnico de Inteligência - Área 8
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

CERTO

Justificativa:

`pg_hba.conf` 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 registros, 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.

Questão 5

Ano: 2018 **Banca:** CESPE **Órgão:** CGM de João Pessoa - PB **Prova:** CESPE - 2018 - CGM de João Pessoa - PB - Auditor Municipal de Controle Interno - Desenvolvimento de Sistemas

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

Ao ser iniciado, o PostgreSQL executa o processo master, que, por sua vez, inicia dois processos auxiliares, stats collector e autovacuum, exibidos como instâncias postgres nas ferramentas de monitoramento de processos mais comuns em Linux.

Certo

Errado

Questão 5

Ano: 2018 **Banca:** CESPE **Órgão:** CGM de João Pessoa - PB **Prova:** CESPE - 2018 - CGM de João Pessoa - PB - Auditor Municipal de Controle Interno - Desenvolvimento de Sistemas

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

Ao ser iniciado, o PostgreSQL executa o processo master, que, por sua vez, inicia dois processos auxiliares, stats collector e autovacuum, exibidos como instâncias postgres nas ferramentas de monitoramento de processos mais comuns em Linux.

ERRADO

Certo

E Justificativa:

O processo do **stats collector** coleta estatísticas sobre o banco de dados. É um processo opcional com o valor padrão ativado. O processo controla o acesso a tabelas e índices em termos de bloco de disco e de linha individuais. Ele também controla as contagens de registros das tabelas e rastreia o vácuo e analisa as ações. Os dados coletados são armazenados em um conjunto de tabelas e podemos acessá-lo através de várias visualizações fornecidas.

As visualizações começam com pg_stat.

\d pg_stat

O daemon AUTOVACUUM é composto de vários processos que recuperam o armazenamento removendo dados ou tuplas obsoletos do banco de dados. Ele verifica as tabelas que possuem um número significativo de registros inseridos, atualizados ou excluídos e esvazia essas tabelas. Pode ser configurado no postgresql.conf

Questão 6

Ano: 2018 **Banca:** FCC **Órgão:** DPE-AM **Prova:** FCC - 2018 - DPE-AM - Analista em Gestão Especializado de Defensoria - Analista de Banco de Dados

No sistema gerenciador de banco de dados PostgreSQL 8 é possível configurar diversos parâmetros de funcionamento por meio da edição do arquivo

- a) postgresql.data.
- b) pgsql.gz.
- c) pgsql.java.
- d) pgsql.prog.
- e) postgresql.conf.

Questão 6

Ano: 2018 **Banca:** FCC **Órgão:** DPE-AM **Prova:** FCC - 2018 - DPE-AM - Analista em Gestão Especializado de Defensoria - Analista de Banco de Dados

No sistema gerenciador de banco de dados PostgreSQL 8 é possível configurar diversos parâmetros de funcionamento por meio da edição do arquivo

- a) postgresql.data.
- b) pgsql.gz.
- c) pgsql.java.
- d) pgsql.prog.
- e) postgresql.conf.

LETRA E

Justificativa:

Os parâmetros de configuração no arquivo **postgresql.conf** especificam o comportamento do servidor com relação a auditoria, autenticação, criptografia e outros comportamentos.

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).

Questão 7

Ano: 2018 **Banca:** FCC **Órgão:** TRT - 6ª Região (PE) **Prova:** FCC - 2018 - TRT - 6ª Região (PE) - Analista Judiciário - Tecnologia da Informação

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) CANCEL OPERATION;
- b) RESTORE TO ps1;
- c) CANCEL UPDATE;
- d) ROLLBACK -1;
- e) ROLLBACK TO ps1;

Questão 7

Ano: 2018 **Banca:** FCC **Órgão:** TRT - 6ª Região (PE) **Prova:** FCC - 2018 - TRT - 6ª Região (PE) - Analista Judiciário - Tecnologia da Informação

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) CANCEL OPERATION;
- b) RESTORE TO ps1;
- c) CANCEL UPDATE;
- d) ROLLBACK -1;
- e) ROLLBACK TO ps1;

LETRA E

Gabarito

Questão	Resposta
1	<i>E</i>
2	<i>D</i>
3	<i>ERRADO</i>
4	<i>CERTO</i>
5	<i>ERRADO</i>
6	<i>E</i>
7	<i>E</i>



STUDY **HARD**

Referências

- https://wiki.postgresql.org/wiki/PostgreSQL_Tutorials
- <https://www.postgresql.org/files/documentation/pdf/12/postgresql-12-A4.pdf>
- <https://www.postgresql.org/docs/12/>