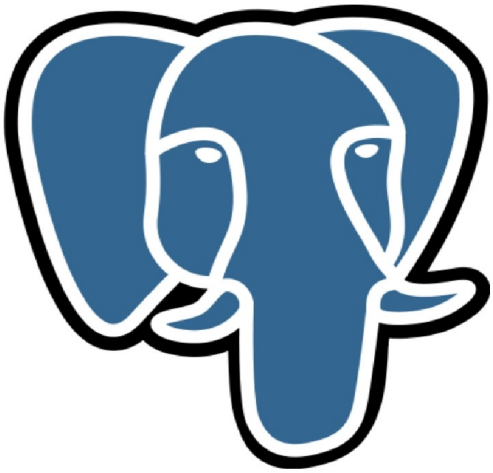

PostgreSQL Essentials

Objetivos do Curso

- **Outros objetos de banco de dados**
  - **Visões(VIEW)**
  - **Sequências(SEQUENCE)**
  - **Índices(INDEX)**
  - **Tabelas clusterizadas**
  - **Cursores**
  - **Manipulação de Cursores**
- **Controle de transações**
  - **Transações no PostgreSQL**
  - **Padrão ACID**
  - **Mecanismo MVCC**
  - **Níveis de isolamento**
  - **Read Commited**
  - **Serializable**
  - **Visualização das diferenças**
  - **SAVEPOINT**
- **BLOBs**
  - **Conceitos**
  - **BLOBs no PostgreSQL**
  - **Manipulação de BLOBs**

Notação Utilizada no Curso

Itálico **: Utilizamos para nomes de objetos de banco de dados, arquivos e outros elementos que devem ser fornecidos pelo usuário.**

`Courier New` **: Os comandos, palavras-chave e saídas de comandos são grafados no tipo Courier.**

`Courier New` **: Utilizamos Courier New em negrito para enfatizar uma palavra-chave apresentada pela primeira vez e valores padrão.**

[ ] : Utilizamos colchetes sempre que uma palavra-chave for opcional.

{ } : Utilizamos chaves para delimitar uma lista de itens onde um deles deva ser escolhido.

... : Utilizamos três pontos para indicar que aquele item pode se repetir. Quando utilizado nos exemplos, indica uma parte não importante da informação foi removida.

Capítulo 15

Outros Objetos de Banco de Dados

Views

VIEWs podem ser vistas como consultas armazenadas e, uma vez criadas, são tratadas como tabelas.

Sintaxe:

CREATE [OR REPLACE] [TEMP | TEMPORARY] VIEW name
[(column_name [, ...])]
AS query

Exemplo:

```
CREATE VIEW view_pedidos_de_hoje AS
SELECT u_conexao,
       u_codmaq,
       u_pkey,
       u_numero_polibras,
       t_data,
       t_data_conexao,
       exporta_data,
       exporta_login
FROM pedcab
WHERE t_data_conexao > CURRENT_DATE
;
```

Para remover uma VIEW:

DROP VIEW [IF EXISTS] name [, ...] [CASCADE |
RESTRICT]

Sequences

Seqüências são utilizadas para gerar identificadores únicos (números inteiros) para linhas de tabelas.

Sintaxe básica:
CREATE [TEMPORARY | TEMP] SEQUENCE name [INCREMENT [BY] increment] [MINVALUE minvalue | NO MINVALUE] [MAXVALUE maxvalue | NO MAXVALUE] [START [WITH] start] [CACHE cache] [[NO] CYCLE] [OWNED BY { table.column | NONE }]

Exemplo:
CREATE SEQUENCE seq INCREMENT 2 MINVALUE 100 MAXVALUE 110 CYCLE;

Para evitar o bloqueio de transações concorrentes que acessem uma mesma SEQUENCE, uma operação NEXTVAL nunca é desfeita. Isto é, uma vez que um valor foi pego na SEQUENCE, ele é considerado utilizado. O resultado disto é que uma transação abortada pode deixar buracos numa seqüência de valores na tabela.

A função NEXTVAL é utilizada para pegar o próximo valor da seqüência:

```
SELECT nextval('seq');
nextval
---------
100
```

A função CURRVAL retorna o último valor tomado por NEXTVAL naquela sessão:

```
SELECT currval('seq');
currval-
--------
100
```

A função SETVAL muda o valor corrente da seqüência para o argumento passado (será descontinuada):

```
SELECT setval('seq', 101);
setval
---------
101
```

Comando ALTER SEQUENCE

A partir da 7.4 esse comando foi implementado para alterar as propriedades de uma SEQUENCE.

Sintaxe:

ALTER SEQUENCE name [INCREMENT [BY] increment]
[MINVALUE minvalue | NO MINVALUE] [MAXVALUE
maxvalue | NO MAXVALUE] [RESTART [WITH] start]
[CACHE cache] [[NO] CYCLE] [OWNED BY
{ table.column | NONE }]

Exemplo:

ALTER SEQUENCE seq INCREMENT 1 MINVALUE 1 MAXVALUE 10
RESTART WITH 1 NO CYCLE;

```
SELECT nextval('seq');
nextval
---------
1
```

Criar uma coluna com o tipo SERIAL é equivalente a criar uma seqüência e colocar o resultado da função nextval como default da coluna:

```
CREATE TABLE exemplo1 (
codigo serial,
nome text
);
```

É equivalente à:

```
CREATE SEQUENCE codigo_seq;
CREATE TABLE exemplo2 (
codigo int NOT NULL DEFAULT NEXTVAL ('codigo_seq'),
nome text
);
```

Índices

Um índice é um objeto de banco do dados que pode aumentar sensivelmente a performance de execução das queries.

Índices sempre são associados a tabelas e a uma ou mais colunas. Um tabela pode ter vários índices.

Um índice rastreia os dados de uma ou mais colunas num mecanismo próprio (árvore ou hash), permitindo que as cláusulas WHERE obtenham seus valores mais rapidamente.

As colunas a serem indexadas são aquelas frequentemente utilizadas nas cláusulas WHERE e com uma grande variação no conteúdo dos dados.

Para que o índice seja utilizado, não é necessário especificar ao PostgreSQL, ele próprio decide automaticamente se um índice vai ou não ser utilizado.

Sintaxe :

CREATE [UNIQUE] INDEX [CONCURRENTLY] name ON table
[USING method] [WHERE predicate]
DROP INDEX [IF EXISTS] name [, ...] [CASCADE |
RESTRICT]

Existem três tipos básicos de índices no PostgreSQL:

Btree: implementação de árvores B. É a opção padrão e recomendada.
Gist: Utilizado para tipos de dados geométricos.
Hash: implementa uma função de hash.
Gin: Utilizado para valores do tipo array.

Exemplo :

```
CREATE UNIQUE INDEX idx_agente_nome ON agente (nome);
```

Tabelas Clusterizadas

Uma tabela clusterizada é uma tabela ordenada fisicamente no banco baseando-se nas chaves de um índice já existente.

O uso de tabelas clusterizadas pode trazer ganhos de performance ao banco de dados.

Sempre que novos dados são adicionados ou modificados na tabela, deve-se executar novamente a operação de clusterização.

Sintaxe:

CLUSTER tablename [USING indexname]

Exemplo:

```
CLUSTER aposta_numeros USING aposta_numeros_pkey;
```

Cursores Explícitos

O PostgreSQL oferece a utilização de cursores para a execução de consultas no banco de dados.

Cursores são ponteiros para leitura de um SELECT executado no banco de dados. Os cursores necessitam de conexões persistentes com o banco.

Usando cursores, uma aplicação pode gerenciar de maneira mais eficiente uma consulta executada, não sendo necessário executa-la várias vezes (usando LIMIT e OFFSET) e poupando recursos
de memória.

Cursores podem ser utilizados tanto em SQL como em PL/pgSQL (ou outra linguagem procedural aceita pelo PostgreSQL). No SQL, a declaração do cursor (DECLARE) automaticamente o abre para execução.

Sintaxe de Declaração de Cursores

Sintaxe para criação de cursores:

DECLARE name [BINARY] [INSENSITIVE] [[NO] SCROLL]
CURSOR [{ WITH | WITHOUT } HOLD] FOR query

A palavra-chave WITH HOLD possibilita a criação e o acesso de um cursor fora de um bloco de transação.

A palavra-chace SCROLL faz com que o cursor acesse dados de forma não sequencial.

Exemplo:

```
BEGIN;
DECLARE apostas CURSOR
FOR SELECT * FROM aposta_numeros;
```

Até a versão 8.1 não existe cursor para atualização.

Recuperando Linhas do Cursor

O comando FETCH é utilizado para recuperar as linhas do cursor.
Sintaxe:
FETCH [direction { FROM | IN }] cursorname

Um cursor sempre aponta para a posição corrente da consulta executada. A palavra chave ABSOLUTE faz com que a primeira posição do cursor seja usada como base.

O cursor pode tanto andar para frente (FORWARD) como para trás (BACKWARD) na recuperação dos dados.

É possível especificar a quantidade de linhas recuperada pelo cursor (o padrão é 1).

Exemplos:

```
FETCH 10 FROM apostas;
FETCH BACKWARD FROM apostas;
FETCH PRIOR FROM apostas;
FETCH -1 FROM apostas;
```

Movendo e Fechando um Cursor

É possível mover o ponteiro do cursor para uma posição especificada sem recuperar as linhas. Isto é feito com o comando MOVE.

A sintaxe do MOVE é bastante semelhante à do FETCH:

MOVE [direction { FROM | IN }] cursorname

Exemplo:

MOVE FORWARD 4 FROM apostas;
Um cursor pode ser fechado de duas maneiras:
Explicitamente através do comando CLOSE:
CLOSE apostas;
Implicitamente quando o bloco de transação é encerrado (COMMIT ou ROLLBACK

Controle de Transações

Transações no PostgreSQL

Uma transação é uma seqüência indivisível de comandos SQL que modifica o conteúdo do banco de dados.

O PostgreSQL utiliza um mecanismo chamado Multiversion Concurrency Control (MVCC) para garantir as transações do banco de dados.

O mecanismo MVCC faz com que cada transação veja um instante do banco. Isso faz com que um atualizador não bloqueie um escritor.

Caso a transação não seja explicitamente aberta, o comportamento padrão do PostgreSQL é o autocommit, ou seja, o banco de dados é sincronizado (COMMIT) com cada comando logo após a sua execução.

As modificações efetuadas no decorrer de uma transação por um usuário só são visíveis a outros usuários após a efetivação da transação.

Padrão ACID

Atomicidade

- O resultado da execução de uma transação é totalmente completado, ou não é executada nenhuma das operações que fazem parte da transação. Ou todas as mudanças tem efeito e são realizadas, ou nenhuma delas é executada.

- Consistência
  A execução de uma transação isolada preserva a consistência do banco de dados.

- IIsolamento

  Cada transação não toma conhecimento de outras transações concorrentes no sistema.

- Durabilidade
  Depois de terminada com sucesso, as alterações feitas pela transação no banco de dados persistem.

Transações no PostgreSQL

A seqüência abaixo não seria adequada sem o uso de transações:

Retira o dinheiro da conta A:

```
UPDATE contas SET saldo = saldo - 100
WHERE n_conta = 7534;
```

Deposita o dinheiro na conta B:

```
UPDATE contas SET saldo = saldo + 100
WHERE n_conta = 12345;
```

Para que a operação acima seja segura, ou os dois passos devem ser executados ou nenhum deles pode ocorrer.

Uma transação ou é executada totalmente ou não é executada. Não existe transação parcialmente executada.

O início de uma transação é determinado pelo comando BEGIN e utiliza o nível de isolamento padrão read committed.

O fim de uma transação é determinado por:

- Execução do comando COMMIT;
- Execução do comando ROLLBACK;
- Término da sessão (equivalente ao ROLLBACK);
- Quando uma sessão é interrompida por fatores externos (ROLLBACK).

Sintaxe de um bloco de transação:
BEGIN [WORK | TRANSACTION] ;
comandos sql;
{ COMMIT | END | ROLLBACK };

Transações no PostgreSQL

Reescrevendo a seqüência da transferência entre contas:

```
BEGIN;
UPDATE contas SET saldo = saldo - 100
WHERE n_conta = 7534;
UPDATE contas SET saldo = saldo + 100
WHERE n_conta = 12345;
COMMIT;
```

Caso algum erro ocorra na execução dos comandos, a transação é abortada (ROLLBACK).

Mas e em relação a transações concorrentes?

Níveis de Isolamento

O padrão SQL define quatro níveis de isolamento e três problemas que devem ser evitados:

Dirty Read (leitura suja)

Uma transação lê dados ainda não efetivados por uma transação concorrente.

Nonrepeatable Read (leitura que não se repete)

A transação lê uma segunda vez os dados, e descobre que os dados foram modificados por outra transação.

Phantom Read (leitura fantasma)

A transação executa pela segunda vez uma mesma consulta e descobre que a quantidade de registros resultante é diferente, devido a uma transação recentemente efetivada.

dirty read — A transação SQL T1 altera uma linha. Em seguida a transação SQL T2 lê esta linha antes de T1 executar o comando COMMIT. Se depois T1 executar o comando ROLLBACK, T2 terá lido uma linha que nunca foi efetivada e que, portanto, pode ser considerada como nunca tendo existido.

nonrepeatable read — A transação SQL T1 lê uma linha. Em seguida a transação SQL T2 altera ou exclui esta linha e executa o comando COMMIT. Se T1 tentar ler esta linha novamente, pode receber o valor alterado ou descobrir que a linha foi excluída.

Phantom read — A transação SQL T1 lê um conjunto de linhas N que satisfazem a uma condição de procura. Em seguida a transação SQL T2 executa comandos SQL que geram uma ou mais linhas que satisfazem a condição de procura usada pela transação T1. Se depois a transação SQL T1 repetir a leitura inicial com a mesma condição de procura, será obtida uma coleção diferente de linhas.

Níveis de Isolamento

A tabela abaixo mostra os níveis de isolamento e os possíveis fenômenos:

Nível de isolamento	Dirty Read	Nonrepeatable Read	Phantom Read
Read uncommitted	Possível	Possível	Possível
Read committed	Impossível	Possível	Possível
Repeatable read	Impossível	Impossível	Possível
Serializable	Impossível	Impossível	Impossível

Uma transação pode ser iniciada nos quatro níveis.

Entretanto, internamente para o PostgreSQL, somente existem o Read Commited e Serializable.

O Read Uncommitted equivale ao Read Committed.
O Repeatable Read equilave ao Serializable.

Transações Read Committed

O Read Comitted é o nível de isolamento padrão do PostgreSQL.

Sobre este nível de isolamento, uma transação:

Enxerga dados efetivados antes do início de cada comando. Nunca enxerga dados ainda não efetivados por transações concorrentes ou mudanças ocorridas durante a execução de uma consulta.

Enxerga as alterações feitas na própria transação, mesmo que ainda não tenham sido efetivados.

Os comandos de UPDATE, DELETE, SELECT e SELECT FOR UPDATE somente encontram linhas efetivadas até o início do comando.

Caso duas transações tentem alterar o mesmo registro o segundo atualizador aguarda a transação que iniciou primeiro efetivar (COMMIT) ou desfazer (ROLLBACK):

Se o primeiro atualizador desfizer (ROLLBACK) então a linha original é recuperada e é usada pelo segundo atualizador.

Se o primeiro atualizador efetivar (COMMIT) o segundo atualizador ignora a linha se esta foi excluída, senão tenta aplicar a operação com a versão atualizada da linha.

locks

```
Lock type             Acquired by system for      Conflicts with
1 AccessShareLock     SELECT                      7
2 RowShareLock        SELECT FOR UPDATE           6,7
3 RowExclusiveLock    UPDATE, INSERT, DELETE      4,5,6,7
4 ShareLock           CREATE INDEX                3,5,6,7
5 ShareRowExclusiveLock                           3,4,5,6,7
6 ExclusiveLock                                   2,3,4,5,6,7
7 AccessExclusiveLock DROP TABLE, ALTER TABLE, VACUUM all
```

Transações Serializable

O Serializable é o nível de isolamento mais restritivo.

Sobre este nível de isolamento, uma transação:

É executada de modo serializado, ou seja, como se uma transação fosse executada após a outra.

Enxerga o mesmo instante do início da transação. Entretanto enxerga dados alterados dentro da própria transação mesmo que não tenham sido efetivados.

As aplicações que usam esse tipo de isolamento devem estar preparadas para submeter novamente a transação devido a falhas na serialização.

Os comandos de UPDATE, DELETE, SELECT e SELECT FOR UPDATE somente encontram linhas efetivadas até o início da transação.

Caso duas transações tentem alterar o mesmo registro o segundo atualizador aguarda a transação que iniciou primeiro efetivar (COMMIT) ou desfazer (ROLLBACK):

Se o primeiro atualizador desfizer (ROLLBACK) então a linha original é recuperada e é usada pelo segundo atualizador.

Se o primeiro atualizador efetivar (COMMIT) então a transação do segundo atualizador é abortada com a seguinte mensagem de erro:
ERROR: could not serialize access due to concurrent update

Transações no PostgreSQL

É possível trocar o nível de isolamento padrão para toda uma sessão:

SET SESSION CHARACTERISTICS AS TRANSACTION
[ISOLATION LEVEL { SERIALIZABLE | REPEATABLE |
READ COMMITTED | READ UNCOMMITTED }]

Para iniciar transações explicitando o nível de isolamento.

Sintaxe:

{ START TRANSACTION | BEGIN [TRANSACTION | WORK] }
[ISOLATION LEVEL { SERIALIZABLE | REPEATABLE READ |
READ COMMITTED | READ UNCOMMITTED }]

DEADLOCK

```
create table a1 (a1 int, b1 text);
INSERT INTO a1 VALUES (1, 'emanuel'),(1, 'rafael'),(1, 'deborah'),(1,
'gledson') ;

-- ses1
begin;
     update a1 set b1 = 'emanuel araujo' where a1 = 1;

-- ses2
begin;
     update a1 set b1 = 'rafael nogueira' where a1 = 2;

-- ses1
     update a1 set b1 = 'rafael nog' where a1 = 2;

-- ses2
     update a1 set b1 = 'emanuel ara' where a1 = 1;
```

SAVEPOINT

O comando SAVEPOINT cria uma marca especial dentro de uma transação (ponto de salvamento), permitindo que os comandos executados após essa marca sejam desfeitos (ROLLBACK), voltando ao estado anterior a esse ponto de salvamento.

Sintaxe:

SAVEPOINT savepoint_name

Para retornar ao estado anterior a um ponto de salvamento, utilizamos o comando:

ROLLBACK TO SAVEPOINT savepoint_name

Exemplo:

```
BEGIN;
INSERT INTO agente VALUES (18, 4, 'Coelho Verde');
SAVEPOINT my_savepoint;
INSERT INTO agente VALUES (19, 3, 'Super Trevo');
ROLLBACK TO SAVEPOINT my_savepoint;
SELECT * FROM agente;
COMMIT;
```

Também é possível liberar um ponto de salvamento através do comando:
RELEASE SAVEPOINT savepoint_name;

Exemplo:

```
BEGIN;
INSERT INTO agente VALUES (19, 3, 'Super Trevo');
SAVEPOINT my_savepoint;
RELEASE my_savepoint;
ROLLBACK TO SAVEPOINT my_savepoint;
ERROR: no such savepoint
```

Sempre que um ponto de salvamento não é encontrado pelo comando ROLLBACK TO SAVEPOINT um erro aborta a transação corrente.

Quando um novo SAVEPOINT é criado com o mesmo nome, o antigo é mantido, mas não mais acessível, a menos que uma operação de RELEASE seja feita neste novo SAVEPOINT, permitindo então acesso ao antigo.

BLOBS

Blobs no PostgreSQL

BLOB é a sigla para Binary Large OBject.
Os BLOBs possibilitam a gravação de dados no banco em formato binário como: imagem, som, vídeo, arquivos executáveis, etc.
Geralmente são dados multimídia que ficam gravados nesses campos.
O tipo bytea é usado para representar os dados de um BLOB.
O uso de BLOBs pode dificultar o porte para outros gerenciadores de banco de dados. Uma maneira de contornar esse problema é gravar no campo da tabela somente o diretório do sistema operacional para o arquivo binário

BLOBS

Existem duas funções para a inserção e recuperação de blobs no PostgreSQL:

lo_import(text): Essa função recebe como parâmetro o local onde se encontra o arquivo e retorna o identificador do arquivo importado para o banco.

lo_export (oid, text): Essa função recebe como parâmetro o OID do BLOB a ser exportado e o local onde o arquivo deverá ser gravado.

Quando uma importação é feita o campo na tabela receberá apenas o OID identificador), que é uma referência para os dados do BLOB que estão na tabela pg_largeobject:

pg_largeobject		
campo	**tipo**	**descrição**
loid	oid	Identificador do blob.
pageno	int4	O número da página do blob.
data	bytea	Os dados do blob.

Exemplos

Criando a tabela com uma coluna do tipo OID que terá a referência para a tabela pg_largeobject:

```
CREATE TABLE imagem (
nome_imagem varchar(20),
oid_imagem oid);
```

Inserindo o dado no formato binário:

```
INSERT INTO imagem (nome_imagem, oid_imagem)
VALUES ('lago', lo_import('/tmp/lago.jpg'));
```

Recuperando o dado

```
SELECT lo_export(oid_imagem, '/tmp/lago_export.jpg')
FROM imagem
WHERE nome_imagem = 'lago';
```

Apagando BLOBS

Mesmo que usemos o DELETE na tabela imagem (no exemplo da transparência anterior) o largeobject não será apagado da tabela

pg_largeobject. Para remover o objeto da base usamos a função lo_unlink e passamos como parâmetro o OID do BLOB.

Exemplo:

```
SELECT lo_unlink(33441);
```

ou

```
SELECT lo_unlink(oid_imagem)
FROM imagem WHERE nome_imagem = 'lago';
```

Agora podemos apagar o registro da tabela imagem:

```
DELETE FROM imagem
WHERE nome_imagem='lago';
```

OBRIGADO!