

Mais sobre SQL

Profa. Dra. Sarajane Marques Peres, 2010
Escola de Artes, Ciências e Humanidades da USP

<http://each.uspnet.usp.br/sarajane/>

Roteiro

- ▶ Asserções
- ▶ Visões
- ▶ Gatilhos





Assertões



Asserções

- ▶ Utilizadas para especificar restrições genéricas, via asserções declarativas, usando a declaração CREATE ASSERTION (comando DDL da SQL padrão).
- ▶ Exemplo (Navathe – modelo na pág. 38)
 - ▶ O salário de um empregado não pode ser maior do que o salário do gerente do departamento em que ele trabalhar.

```
CREATE ASSERTION Limite_Salario  
CHECK (NOT EXISTS  
    (SELECT *  
     FROM empregado E, empregado M, departamento D  
     WHERE E.salario > M.salario AND  
           E.dno = D.dnumero AND  
           D.ssnger = M.ssn));
```

Se existir uma ou mais tuplas satisfazendo essa consulta (empregado ganhando mais que gerente), o SELECT retornará **Verdadeiro** para o EXIST, que é negado pelo NOT e retorna um **Falso** para o Check. ASSERTION violada!!!

Asserções

- ▶ Explorando a declaração da asserção:
 - ▶ A condição entre parênteses (condição do CHECK) precisa ser verdadeira em todos os estados do banco de dados para que a asserção seja satisfeita.
 - ▶ O SGBD é responsável por garantir que a condição não seja violada.
 - ▶ Na asserção modelada:
 - ▶ Sempre que alguma tupla no BD fizer com que a condição evolua para FALSE, a restrição será violada;
 - ▶ Se a restrição é violada então o SGBD deve impedir que tal tupla entre ou permaneça* do banco de dados
 - ▶ A restrição será satisfeita para um dado estado do BD se nenhuma combinação de tuplas no BD violar a restrição.

* Se a asserção é criada no SGBD depois que o banco de dados já está populado, é preciso verificar na “tecnologia” utilizada se a averiguação no estado atual do banco de dados será realizada.

Asserções

- ▶ Técnica básica para criação de asserções:
 - ▶ Especificar uma consulta que selecione as tuplas que VIOLEM a condição desejada;
 - ▶ Incluir esta consulta em uma cláusula NOT EXIST;
 - ▶ Assim a asserção exigirá que o resultado dessa consulta seja vazio e será violada se o resultado NÃO for vazio.
- ▶ CREATE ASSERTION x cláusula CHECK
 - ▶ Podemos usar a cláusula CHECK para impor condições para domínios e tuplas.
 - ▶ Asserções são usadas apenas quando a cláusula CHECK não é suficiente para atender à uma restrição do domínio sob modelagem.



Cláusula CHECK no PostGreSQL

`CHECK (expression)`

The `CHECK` clause specifies an expression producing a Boolean result which new or updated rows must satisfy for an insert or update operation to succeed. Expressions evaluating to `TRUE` or `UNKNOWN` succeed. Should any row of an insert or update operation produce a `FALSE` result an error exception is raised and the insert or update does not alter the database. A check constraint specified as a column constraint should reference that column's value only, while an expression appearing in a table constraint can reference multiple columns.

Currently, `CHECK` expressions cannot contain subqueries nor refer to variables other than columns of the current row.



Experimente!!!



Verifique em
outros SGBDs!!!



CREATE ASSERTION no PostgreSQL?

- ▶ O PostgreSQL não suporta o comando CREATE ASSERTION.

- ▶ Veja:

- ▶ <http://www.htmlstaff.org/postgresqlmanual/sql-createtable.html>




Asserções

Uma asserção (assertion) é um tipo especial de restrição de integridade e compartilha o mesmo espaço de nomes como outras restrições. Entretanto, uma asserção não é necessariamente dependente de uma tabela em particular como as restrições são, por isso o SQL92 fornece a declaração `CREATE ASSERTION` como uma forma alternativa para definir restrições:

```
CREATE ASSERTION nome CHECK ( condição )
```

O PostgreSQL não implementa asserções atualmente.



SQL:2003
SQL:2008

- ▶ Para satisfazer essa restrição você pode implementar RULES ou TRIGGERS.
- ▶ Exemplo: próximo slide



Exemplo de alternativa para ASSEÇÃO

-- Restricao de integridade que nao permite a existencia de marcas que fabriquem pcs e portateis (ambos).
-- NOTA: nao funciona em POSTGRESQL

```
CREATE ASSERTION pcportatil CHECK (NOT EXISTS
    (SELECT marca FROM produto WHERE tipo='pc'
    INTERSECT
    SELECT marca FROM produto WHERE tipo='portatil'));
```

-- Alternativa (aceita em POSTGRESQL)

```
CREATE RULE pcportatil_i AS
ON INSERT TO produto WHERE NEW.tipo='pc' AND
    NEW.marca IN (SELECT marca
        FROM produto
        WHERE tipo='portatil')
    OR
    NEW.tipo='portatil' AND
    NEW.marca IN (SELECT marca
        FROM produto
        WHERE tipo='pc')
DO INSTEAD NOTHING;

CREATE RULE pcportatil_u AS
ON UPDATE TO produto WHERE NEW.tipo='pc' AND
    NEW.marca IN (SELECT marca
        FROM produto
        WHERE tipo='portatil')
    OR
    NEW.tipo='portatil' AND
    NEW.marca IN (SELECT marca
        FROM produto
        WHERE tipo='pc')
DO INSTEAD NOTHING;
```

Fonte: http://w3.ualg.pt/~mzacaria/bd/bd_pcs.html

Obs.: É preciso testar!



Visões



Visões

- ▶ Uma visão (na terminologia SQL) é uma tabela única derivada de outra tabela, que pode ser uma tabela básica ou uma visão previamente definida.
- ▶ Uma visão não existe de forma física, ela é considerada uma tabela virtual.
 - ▶ A menos que seja materializada
- ▶ A não materialização limita as operações de atualização possíveis para as visões, embora não imponha nenhuma limitação para as consultas.
- ▶ A criação de uma visão é motivada por:
 - ▶ Consultas freqüentes
 - ▶ Consultas complexas
 - ▶ Segurança

Consulte no SGBD em uso se o *default* é materializar ou não!



Visões

- ▶ Exemplo (Navathe, modelo na pág. 38)
 - ▶ Visão que realiza a consulta de recuperação dos nomes dos empregados e os projetos nos quais eles trabalham.

```
CREATE VIEW trabalha_em1  
AS SELECT pnome, unome, pjnome, horas  
FROM empregado, projeto, trabalha_em  
WHERE ssn = essn AND pno = pnumero;
```

- ▶ Obs.: Veja que ao invés de formular uma junção entre as tabelas **empregado**, **trabalha_em** e **projeto** toda vez que isso for necessário, o resultado da visão já nos fornece esta relação.



Visões

- ▶ Exemplo (Navathe, modelo na pág. 38)

```
CREATE VIEW dept_info (dept_nome, no_emps, total_sal)
AS SELECT dnome, COUNT(*), SUM(salário)
   FROM departamento, empregado
   WHERE dnumero = dno
   GROUP BY dnome;
```

- ▶ Observe que aqui foram criados e nomeados novos atributos para a visão.
- ▶ departamento e empregado são as tabelas de definição (ou tabelas primárias/básicas) da visão.
- ▶ O nome da visão pode ser usado em consultas SQL da mesma forma que as tabelas básicas são usadas.



Visões

- ▶ Supõe-se que uma visão esteja sempre atualizada.
- ▶ Se as tuplas das tabelas básicas, sobre as quais a visão foi criada, forem modificadas, a visão deverá, automaticamente, refletir essas alterações.

Verifique no SGBD em uso!!!

- ▶ Conseqüentemente, a visão não é, teoricamente, realizada no instante de sua definição, mas quando especificar-se uma consulta sobre ela.
- ▶ **DROP VIEW**: comando para apagar a visão.
 - ▶ DROP VIEW trabalhar_em_1;



Visões

- ▶ Implementações de visões:
 - ▶ **Modificação da consulta que usa visão:** implica em transformar uma consulta sobre uma visão em uma consulta sobre tabelas básicas
 - ▶ na primeira execução da visão em uma determinada sessão da aplicação de banco de dados, ou
 - ▶ a cada vez que a visão for solicitada e, por algum motivo, o SGBD não a tem disponível – (liberação de memória, por exemplo).
 - ▶ **Materialização da visão:** criar fisicamente uma tabela temporária a partir da primeira consulta para a visão e mantê-la além da sessão atual da aplicação de banco de dados e além das decisões de gerenciamento de memória do SGBD.
 - ▶ Neste caso é necessário estabelecer uma estratégia de atualização da visão



Visões

- ▶ Atualização incremental (atualização da visão):
 - ▶ Determina-se quais novas tuplas precisam ser inseridas, apagadas ou modificadas na tabela materializada da visão, quando uma alteração for aplicada para alguma(s) das tabelas básicas da visão.
- ▶ Atualização a partir da visão:
 - ▶ Cuidado: atualizações de visão são complicadas e podem ser ambíguas.
 - ▶ Em geral, a atualização de uma visão definida em uma ÚNICA TABELA, sem FUNÇÕES AGREGADAS, pode ser mapeada na atualização de uma tabela básica subjacente sob certas condições.



Visões (atualizando dados a partir das visões)

- ▶ Exemplo de interpretações ambíguas (Navathe, modelo na pág. 38):
- ▶ Considere a visão **trabalha_em1** e suponha que seja emitido um comando de atualização do atributo **pjnome** de '**produto X**' para '**produto Y**' em relação ao empregado '**john smith**'.

Atualização a partir da visão	Reflexão nas tabelas básicas
<pre>UPDATE trabalha_em1 SET pjnome = 'produto Y' WHERE unome = 'smith' AND pnome = 'john' AND pjnome = 'produto X';</pre>	<pre>a) UPDATE trabalha_em SET pno = (SELECT pnumero FROM projeto WHERE pjnome = 'produto Y') WHERE essn IN (SELECT ssnn FROM empregado WHERE unome = 'smith' AND pnome = 'john'); AND pno = (SELECT pnumero FROM projeto WHERE pjnome = 'produto X');</pre>
	<pre>b) UPDATE projeto SET pjnome = 'produto Y' WHERE pjnome = 'produto X';</pre>

Visões (atualizando dados a partir das visões)

▶ Resumo:

- ▶ Uma visão de uma única tabela de definição é atualizável se ela contiver, entre os seus atributos:
 - ▶ A chave primária da relação básica;
 - ▶ Todos os atributos com restrição NOT NULL que não contiverem valores default especificado;
- ▶ As visões definidas a partir de diversas tabelas utilizando-se as junções, em geral, não são atualizáveis;
- ▶ As visões definidas usando-se as funções de agrupamento e agregadas não são atualizáveis.
- ▶ Na SQL, a cláusula **WITH CHECK OPTION** precisa ser adicionada no final da definição da visão se ela puder ser atualizada.
- ▶ Teoricamente, isto permite que o sistema cheque a capacidade de atualização da visão e planeje a estratégia de execução das atualizações.



Visões X PostgreSQL

CREATE VIEW

Name

CREATE VIEW -- define a new view

Synopsis

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

Description

CREATE VIEW defines a view of a query. The view is not physically materialized. Instead, the query is run every time the view is referenced in a query.



O comando [CREATE VIEW](#) cria uma tabela fictícia (sem armazenamento subjacente) e associa uma regra ON SELECT a esta tabela. O sistema não permite atualizações na visão, porque sabe que não existe uma tabela real. Pode ser criada a ilusão de uma visão atualizável definindo-se regras para ON INSERT, ON UPDATE e ON DELETE (ou qualquer subconjunto destes comandos que for suficiente para atender as necessidades) para substituir as ações de atualização na visão por atualizações apropriadas em outras tabelas.

Examples

Create a view consisting of all comedy films:

```
CREATE VIEW comedies AS  
    SELECT *  
    FROM films  
    WHERE kind = 'Comedy';
```





Gatilhos



Gatilhos

- ▶ Regras que especificam ações disparadas automaticamente por meio de eventos.
- ▶ consideradas um importante avanço para os sistemas de banco de dados.
- ▶ Consideradas um tipo de regra ativa, caracterizando o banco de dados ativo



Gatilhos

- ▶ Modelo ECA: Modelo: **evento-condição-ação**
 - ▶ O **evento** (ou eventos) que ativa (ativam) a regra:
 - ▶ Esses eventos normalmente são operações de atualização de banco de dados aplicadas explicitamente.
 - ▶ Porém, no modelo geral, eles poderiam também ser eventos temporais* ou outros tipos de eventos externos;

* Exemplo: dê este comando diariamente as 5:30.



Gatilhos

- ▶ A **condição** que determina se a ação da regra deve ser executada:
 - ▶ uma vez ocorrido o evento, uma condição **opcional** pode ser avaliada.
 - ▶ Se nenhuma condição for especificada, a ação será disparada pelo evento.
 - ▶ Se uma condição for especificada, primeiro ela será avaliada, e somente se seu resultado for verdadeiro a ação da regra será executada.



Gatilhos

- ▶ A **ação** a ser executada:
 - ▶ normalmente é uma sucessão de declarações SQL.
 - ▶ mas também poderia ser uma transação de banco de dados ou um programa externo que será executado automaticamente.



Gatilhos

- ▶ Exemplo (Navathe)
- ▶ Considere o seguinte esquema simplificado:
 - ▶ **Empregado** (nome, ssn, salario, *dno, *supervisor_ssn)
 - ▶ Considere que é permitido o valor **null** para dno, indicando que um empregado pode estar temporariamente sem departamento.
 - ▶ **Departamento** (dnome, dno, total_sal, *gerente_ssn)
 - ▶ Note que o atributo total_sal é derivado, e seu valor corresponde à soma de salários dos empregados locados naquele departamento em particular. A manutenção do valor correto de tal atributo pode ser tarefa de uma regra ativa.



Gatilhos

- ▶ Primeiro é necessário determinar os eventos que possam causar uma mudança no valor de `total_sal`, que podem ser:
 1. Inserindo uma ou mais tuplas de novos empregados;
 2. Mudando o salário de um ou mais empregados existentes;
 3. Realocando um ou mais empregados entre os departamentos;
 4. Excluindo uma ou mais tuplas de empregados.



Gatilhos

- ▶ **Evento 1:** será necessário apenas recalcular **total_sal** se o empregado novo for imediatamente nomeado para um departamento - isto é, se o valor do atributo de **dno** para a tupla do novo empregado não for **null**. Conseqüentemente, essa será a **condição** a ser checada.
- ▶ Uma condição semelhante poderia ser atribuída ao **evento 2 e 4:** determinar se o empregado cujo salário é alterado (ou que está sendo excluído) está alocado no departamento.
- ▶ Para o **evento 3**, sempre será executada uma ação que mantém o valor correto de **total_sal**, assim, nenhuma condição é necessária (a ação sempre será executada).



Gatilhos

- ▶ A ação para os eventos 1, 2 e 4 é atualizar o valor de **total_sal** automaticamente para que o departamento do empregado reflita as recentes inserções, atualizações e remoções de salário dos empregados.
- ▶ No caso do evento 3, é necessária uma ação dupla: uma para atualizar o **total_sal** do departamento antigo e outra para atualizar o **total_sal** do novo departamento.



Gatilhos

► Sintaxe Oracle (PL/SQL) – Navathe

```
<trigger> ::= create trigger <nome gatilho>  
              (after | before) <eventos ativadores>  
              on <nome tabela>  
              [ for each row ]  
              [ when <condição> ]  
              <ações disparadas>
```

```
<eventos ativadores> ::= <evento ativador> {or <evento ativador>}  
<evento ativador> ::= insert | delete | update [of <nome coluna>  
              {, <nome coluna>}]  
<ação disparada> ::= <bloco PL/SQL>
```



Gatilhos

```
R1 : create trigger totalsal1  
      after insert on empregado  
      for each row  
      when (new.dno is not null)  
        update departamento  
        set total_sal = total_sal + new.salario  
        where dno = new.dno;
```

New: palavra chave usada para se referir à tupla que foi inserida ou atualizada.



Gatilhos

R2 : **create trigger** totalsal2

after update of salario **on** empregado

for each row

when (**new.dno is not null**)

update departamento

set total_sal = total_sal + **new**.salario – **old**.salario

where dno = **new**.dno;

Old: palavra chave usada para se referir à tupla excluída ou à tupla antes de sua atualização.



Gatilhos

```
R3 : create trigger totalsal3
      after update of dno on empregado
      for each row
      begin
        update departamento
        set total_sal = total_sal + new.salario
        where dno = new.dno;
        update departamento
        set total_sal = total_sal - old.salario
        where dno = old.dno;
      end;
```



Gatilhos

```
R4 : create trigger totalsal4  
      after delete on empregado  
      for each row  
      when (old.dno is not null)  
        update departamento  
        set total_sal = total_sal + new.salario – old.salario  
        where dno = old.dno;
```



Um exemplo do *help* do PostgreSQL

- ▶ Este exemplo garante que se uma linha é inserida ou alterada na tabela, o nome de usuário e hora atual são marcados na linha.
- ▶ Também checa se o nome do empregado é dado e se o salário dele é um valor positivo.

```
CREATE TABLE emp (  
    empname text,  
    salary integer,  
    last_date timestamp,  
    last_user text  
);
```



**Funções sem
argumento!**

**Retorno do tipo
“trigger”.**

```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$  
BEGIN
```

Corpo da função!

```
END;  
$emp_stamp$ LANGUAGE plpgsql;
```

**Linguagem usada
no corpo da
função.**

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

**Argumentos que possam vir do gatilho são
passados via TG_ARGV.**



```
CREATE FUNCTION emp_stamp() RETURNS trigger AS $emp_stamp$
BEGIN
    -- Check that empname and salary are given
    IF NEW.empname IS NULL THEN
        RAISE EXCEPTION 'empname cannot be null';
    END IF;
    IF NEW.salary IS NULL THEN
        RAISE EXCEPTION '% cannot have null salary', NEW.empname;
    END IF;

    -- Who works for us when she must pay for it?
    IF NEW.salary < 0 THEN
        RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
    END IF;

    -- Remember who changed the payroll when
    NEW.last_date := current_timestamp;
    NEW.last_user := current_user;
    RETURN NEW;
END;
$emp_stamp$ LANGUAGE plpgsql;
```



Um gatilho para auditoria, analise ...

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



```
CREATE OR REPLACE FUNCTION process_emp_audit() RETURNS TRIGGER AS $emp_audit$  
    BEGIN
```

```
        END;
```

```
$emp_audit$ LANGUAGE plpgsql;
```

```
CREATE TRIGGER emp_audit
```

```
AFTER INSERT OR UPDATE OR DELETE ON emp
```

```
    FOR EACH ROW EXECUTE PROCEDURE process_emp_audit();
```



```

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

```



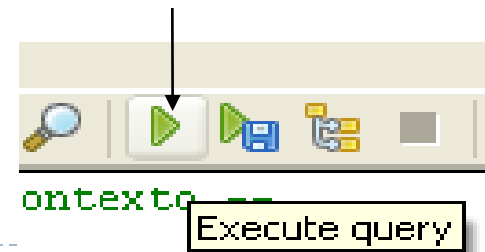
Utilizando o primeiro exemplo do PostgreSQL

```
insert into emp values ('João',100,'2007-02-02','wft');  
insert into emp values ('Maria',300,'2007-02-02','wft');  
insert into emp values ('José',600,'2007-02-02','wft');  
insert into emp values ('Carlos',200,'2007-02-02','wft');
```

Se for necessário criar a linguagem:

```
Create language plpgsql;
```

Execute o código do gatilho!



Object browser

Servers (1)

PostgreSQL Database Server 8.2 (localhost:5432)

Databases (3)

Laboratorio2

postgres

Casts (0)

Languages (0)

Schemas (1)

public

Aggregates (0)

Conversions (0)

Domains (0)

Functions (0)

Trigger Functions (1)

emp_stamp()

Procedures (0)

Operators (0)

Operator Classes (0)

Sequences (0)

Tables (4)

Types (0)

Views (0)

Replication (0)

template_postgis

Tablespaces (2)

Group Roles (0)

Login Roles (1)

Properties

Statistics

Dependencies

Dependents

Trigger Function

Comment

emp_stamp()

SQL pane

-- Function: emp_stamp()

-- DROP FUNCTION emp_stamp();

CREATE OR REPLACE FUNCTION emp_stamp()
 RETURNS "trigger" AS
\$BODY\$
 BEGIN
 -- Check that empname and salary are given
 IF NEW.empname IS NULL THEN
 RAISE EXCEPTION 'empname cannot be null';
 END IF;
 IF NEW.salary IS NULL THEN
 RAISE EXCEPTION '% cannot have null salary', NEW.empname;
 END IF;

 -- Who works for us when she must pay for it?
 IF NEW.salary < 0 THEN
 RAISE EXCEPTION '% cannot have a negative salary', NEW.empname;
 END IF;

 -- Remember who changed the payroll when
 NEW.last_date := current_timestamp;
 NEW.last_user := current_user;
 RETURN NEW;

 END;
\$BODY\$
 LANGUAGE 'plpgsql' VOLATILE;
ALTER FUNCTION emp_stamp() OWNER TO postgres;

Dê um Refresh
no
esquema!!!!

```
insert into emp values ('Lucas',1100,null,null);|
```

	empname text	salary integer	last_date timestamp without time zone	last_user text
1	João	100	2007-02-02 00:00:00	wft
2	Maria	300	2007-02-02 00:00:00	wft
3	José	600	2007-02-02 00:00:00	wft
4	Carlos	200	2007-02-02 00:00:00	wft
5	Lucas	1100	2007-06-06 15:56:33.468	postgres

```
insert into emp values (null,1100,null,null);|
```

Output pane

Data Output

Explain

Messages

History

ERROR: empname cannot be null
SQL state: P0001

Transações

Examples

The following example traverses a table using a cursor:

BEGIN WORK;

-- Set up a cursor:

DECLARE liahona SCROLL CURSOR FOR SELECT * FROM films;

-- Fetch the first 5 rows in the cursor liahona:

FETCH FORWARD 5 FROM liahona;

code	title	did	date_prod	kind	len
BL101	The Third Man	101	1949-12-23	Drama	01:44
BL102	The African Queen	101	1951-08-11	Romantic	01:43
JL201	Une Femme est une Femme	102	1961-03-12	Romantic	01:25
P_301	Vertigo	103	1958-11-14	Action	02:08
P_302	Becket	103	1964-02-03	Drama	02:28

-- Fetch the previous row:

FETCH PRIOR FROM liahona;

code	title	did	date_prod	kind	len
P_301	Vertigo	103	1958-11-14	Action	02:08

-- Close the cursor and end the transaction:

CLOSE liahona;

COMMIT WORK;

Propriedades ACID:
atomicidade
consistência
isolamento
durabilidade



Trabalho T3: SGBDs X Conceitos

- ▶ Escolha um SGBD para realizar suas pesquisas durante o semestre.
- ▶ Estes são os conceitos da primeira pesquisa:
 - ▶ Asserções
 - ▶ Checks
 - ▶ Visões
 - ▶ Gatilhos
- ▶ Verifique se tais comandos são implementados na tecnologia escolhida. Verifique como podemos usá-los. Teste-os. Crie um pequeno tutorial explicando as características do comando mediante os exemplos testados.
- ▶ Armazene essa pesquisa e os produtos criados, pois eles serão entregues e discutidos em aula – últimas semanas de aula.



Bibliografia

- ▶ Site oficial do PostGreSQL:
 - ▶ <http://www.postgresql.org/docs>
- ▶ Elmasri, R. & Navathe, S. B.. Sistemas de Banco de Dados. São Paulo: Addison Wesley, 2005.

