



Instituto Federal de Educação, Ciência e Tecnologia da Bahia - IFBA
Departamento de Informática
Análise e Desenvolvimento de Sistemas / Licenciatura em Computação

Views SQL, Stored Procedures e Triggers

André L. R. Madureira <andre.madureira@ifba.edu.br>
Doutorando em Ciência da Computação (UFBA)
Mestre em Ciência da Computação (UFBA)
Engenheiro da Computação (UFBA)

Porque criar consultas em nível de view?

- Dificuldade em escrever, reescrever e entender consultas
 - Consultas SQL grandes, complexas, e derivadas de várias tabelas (joins e subconsultas)

```
SQLQuery1.sql - Q2...ROD\BPetrovi (53))*  X
1 CREATE VIEW vTop3SalesByQuantity
2 AS
3     SELECT TOP 3 --will only return first 3 records from query
4     Sales.ProductID,
5     Name AS ProductName,
6     SUM(Sales.Quantity) AS TotalQuantity
7     FROM Sales
8     JOIN Products ON Sales.ProductID = Products.ProductID
9     GROUP BY Sales.ProductID,
10             Name
11     ORDER BY SUM(Sales.Quantity) DESC;
```

Views SQL

Views não podem ter atributos com mesmo nome, logo devemos utilizar **nomes qualificados** sempre que acessamos algum atributo

- Consultas podem ser armazenadas como **Views**
 - **View** = consultas armazenadas, tabelas virtuais ou pseudotabelas
- **Objetivo:**
 - Facilitar o acesso a dados contidos em várias tabelas
 - Facilitar o uso de consultas grandes ou complexas
 - Implementar segurança nos dados de uma tabela
 - Restringir acesso de usuários do DB a apenas alguns dados (instâncias) ou atributos

Views SQL

- **Criar view (sintaxe):**
 - **CREATE VIEW** <nome_da_view>[(atributos_view)] **AS** <consulta_SQL>
- **Verificar se a view foi criada:**
 - **SHOW TABLES**
- **Alterar View:**
 - **ALTER VIEW** <nome_da_view> **AS** <nova_consulta_SQL>
- **Excluir View:**
 - **DROP VIEW** <nome_da_view>

Views podem ser usadas em qualquer lugar da consulta SQL que aceite o nome de uma relação (tabela)

Exemplo de View - Controle de Acesso a Dados

- **CREATE VIEW** devedor_banco **AS** (
 SELECT nome, quantia, tx_juros
 FROM Devedor
);

Tabela base: tabela real (modelo lógico) usada para construir a view

Ex: *Devedor* é a **tabela base** da view *devedor_banco*

- **SELECT** nome, quantia **FROM** devedor_banco;
 - Mostre somente o nome e quantia do cliente que tomou um empréstimo no banco (devedor)
 - Perceba que a view serve para controlar quais atributos da tabela Devedor temos acesso

Exemplo de View - Simplificação de Consultas SQL Complexas

- **CREATE VIEW** func_max_salario(depto, max_salario) **AS** (
 SELECT depto, **MAX**(salario)
 FROM Funcionario
 GROUP BY depto
);
- **SELECT** F.nome, F.depto
 FROM Funcionario **AS** F, func_max_salario **AS** M
 WHERE F.salario = M.max_salario **AND** F.depto = M.depto;
 - Encontre o funcionário com maior salário de cada departamento

Delete / Update em Views SQL

- É possível realizar operações de atualizações em views, que são realizadas na **tabela base** (modelo lógico), desde que:
 - A cláusula **FROM** possua apenas uma relação
 - **SELECT** possui apenas atributos (sem funções de agregação, expressões, ou **DISTINCT**)
 - Atributos não listados no **SELECT** podem ser definidos como **NULL**
 - **SELECT** não possui **HAVING** ou **GROUP BY**
- Além dessas restrições, precisamos também criar a view usando o comando **WITH CHECK OPTION**

Exemplo de Delete / Update em Views SQL

- **CREATE VIEW** vw_empregado1 **AS SELECT * FROM** Empregado **WHERE** nome_categoria = "Cat A" **WITH CHECK OPTION ;**
- **UPDATE** vw_empregado1 **SET** nome = "João da Silva" **WHERE** nome = "João" ;

Apesar de ser possível a atualização de algumas views,
esse procedimento é ALTAMENTE DESACONSELHÁVEL

Views devem ser utilizadas **majoritariamente para consultas SELECT**

Comando **WITH CHECK OPTION**

- Impor restrições na atualização de Views
- Atualizações que são emitidas sobre a view terão que se encaixar às condições definidas na cláusula **WHERE** do **SELECT**
 - **Ex: CREATE VIEW** vw_empregado2 **AS SELECT * FROM** Empregado **WHERE** nome_categoria = "Cat B" **WITH CHECK OPTION ;**
 - ~~○ **UPDATE** vw_empregado2 **SET** nome_categoria = "Cat A" **WHERE** nome = "Ricardo" ;~~

Implementação de Views no SQL

- Views podem ser implementadas nos SGBDs através de duas técnicas:
 - **Modificação de consulta**
 - SGBD substitui o nome das views pela consulta SQL inteira que compõem a view
 - **Materialização de view**
 - SGBD armazena as views no banco de dados (como se fosse uma tabela comum)
 - Views materializadas atuam como uma espécie de “cache” para consultas complexas em bancos de dados SQL

Modificação de Consulta com View

- A maioria dos DBMS implementa as views como uma substituição simples do nome da view pela consulta SQL inteira da view
 - Onde há o nome da tabela virtual (view), o DBMS substitui pela consulta SQL da view
 - Isso permite a **expansão de view** (uma view definida a partir de outra)
 - **CREATE VIEW** vw_empregado1 **AS**
SELECT * FROM Empregado **WHERE** nome_categoria = 'Cat A' ;
 - **CREATE VIEW** vw_emp_nome **AS**
SELECT nome **FROM** vw_empregado1;

Expansão de view

- O DBMS substitui as views sempre que eles forem utilizadas em consultas SQL até que nenhuma view exista mais na consulta
 - **CREATE VIEW** vw_empregado1 **AS**
SELECT * FROM Empregado **WHERE** cat = 'A' ;
 - **CREATE VIEW** vw_emp_nome **AS**
SELECT nome **FROM** vw_empregado1;

SELECT * FROM vw_emp_nome

SELECT * FROM (**SELECT** nome **FROM** vw_empregado1)

SELECT * FROM (**SELECT** nome **FROM** (**SELECT * FROM** Empregado **WHERE** cat = 'A'))

Views Materializadas

- Para obter maior desempenho, alguns DBMS implementam **views materializadas**
 - O DBMS não precisa executar a consulta SQL da view sempre que a view é utilizada
 - O DBMS executa a consulta SQL uma vez e cria uma tabela dentro do DB, como uma espécie de “cache” da view
- **Desvantagem:** necessidade de atualizar a view materializada sempre que uma das tabelas base forem atualizadas

View por Modificação x Views Materializada

- **View por Modificação**

- Consome pouco espaço do DB
- Apenas a consulta é armazenada
- Consulta SQL é realizada somente quando uma consulta é feita usando a view

- **View Materializada**

- Consome mais espaço do DB
- O resultado da consulta é armazenado em uma nova tabela
- Consulta SQL realizada sempre que uma das tabelas usadas para construir a view é atualizada

Quando o SGBD utiliza cada tipo de view?

- Na prática, a maioria dos SGBDs usa ambas as views (por modificação e materializadas)
 - SGBD cria views materializadas e as mantém armazenadas enquanto as views estão sendo consultadas
 - Se a view não for consultada por certo período, o sistema remove automaticamente a view
 - É responsabilidade do SGBD manter as views atualizadas, independente da implementação

Comando **WITH**

- Cria uma VIEW temporária (***Common Table Expression – CTE***)
 - Facilita o gerenciamento de consultas SQL complexas, semelhante a uma função de uma linguagem de programação
- O comando **WITH** só é válido para as cláusulas/comandos SQL que são executados junto com o **WITH**
- **Sintaxe:**
 - **WITH** <nome_da_view> (<atributos_view>) **AS** (
 <consulta_SQL>
) <consulta_SQL_que_usa_o_VIEW> ;

Exemplo do Comando **WITH**

- **WITH** saldo_max(valor) **AS** (

SELECT MAX(saldo)

FROM conta

)

SELECT num_conta

FROM conta, saldo_max

WHERE conta.saldo = saldo_max.valor

;

Note que não há ; (ponto e vírgula) aqui.
**Logo tudo está sendo executado como
uma consulta SQL única.**

Exemplo do Comando **WITH**

- **WITH** saldo_max(valor) **AS** (

SELECT MAX(saldo)

FROM conta

)

SELECT num_conta

FROM conta, saldo_max

WHERE conta.saldo = saldo_max.valor

;

Note que não há ; (ponto e vírgula) aqui.
**Logo tudo está sendo executado como
uma consulta SQL única.**

A consulta SQL termina aqui

Exemplo do Comando **WITH**

- **WITH** saldo_max (valor) **AS** (
 SELECT MAX(saldo)
 FROM conta
)
 SELECT num_conta
 FROM conta, saldo_max
 WHERE conta.saldo = saldo_max.valor
;
 - Encontre o número da conta que contém o maior saldo dentre todas as contas

Exemplo 02 - Comando **WITH**

- **WITH** total_agencia (nome_agencia, soma_saldo) **AS** (
 SELECT nome_agencia, **SUM**(saldo)
 FROM conta
 GROUP BY nome_agencia
)
SELECT nome_agencia
FROM total_agencia
;

Limitações das VIEWS

- Views são como tabelas virtuais (SELECTs)
 - Views **SEMPRE** retornam alguma informação (tabela)
- Existem operações no SQL que não retornam dados
 - INSERT INTO
 - DROP / DELETE
 - CREATE
 - etc
- **Como facilitar o uso desses comandos?**

Procedimentos SQL (*Stored Procedures*)

- Facilitam a escrita de comandos SQL complexos
- **Procedures não precisam retornar dados** (ao contrário das VIEWS)
- Procedures são como funções de uma linguagem de programação
 - Podem ter parâmetros e retorno de dados
 - Podem não retornar dados (função *void* ou procedimento)
 - Executam uma sequência de operações no DB
 - Permitem executar comandos DDL (CREATE, DROP, etc)
 - Permitem executar comandos DML (SELECT, INSERT, DELETE, etc)

Procedimentos SQL (*Stored Procedures*)

- **Sintaxe:**
 - **DELIMITER \$\$**
CREATE PROCEDURE nome_procedimento (parametros)
BEGIN
 /*CORPO DO PROCEDIMENTO*/
END \$\$
DELIMITER ;
- **Como chamar um stored procedure?**
 - **CALL** nome_procedimento(parametros) ;

Procedimentos SQL (*Stored Procedures*)

- **Sintaxe:**

- **DELIMITER \$\$**

CREATE PROCEDURE nome_procedimento (parametros)

BEGIN

/*CORPO DO PROCEDIMENTO*/

END \$\$

DELIMITER ;

- **Sintaxe dos parâmetros:**

- (MOD0 nome TIPO, MOD0 nome TIPO, ...)

Parâmetros de *Stored Procedures*

- **Sintaxe dos parâmetros:**
 - (MODO nome TIPO, MODO nome TIPO, ...)
- **MODO:**
 - **IN:** envia dados para o procedimento
 - **OUT:** retorna dados ao final do procedimento (**ex:** ponteiro do C++)
 - **INOUT:** envia dados para o procedimento, e também retorna dados
- **TIPO:** Domínio de cada parâmetro
 - **Ex:** INT, VARCHAR(25), DOUBLE, BOOLEAN, etc

Exemplo de *Stored Procedure*

- Liste todos os primeiros **N** produtos, onde **N** é definido pelo parâmetro **quantidade**

```
DELIMITER $$  
CREATE PROCEDURE Selecionar_Produtos(IN quantidade INT)  
BEGIN  
    SELECT * FROM PRODUTOS  
    LIMIT quantidade ;  
END $$  
DELIMITER ;
```

Exemplo de *Stored Procedure*

- Liste todos os primeiros **N** produtos, onde **N** é definido pelo parâmetro **quantidade**

```
DELIMITER $$  
CREATE PROCEDURE Selecionar_Produtos(IN quantidade INT)  
BEGIN  
    SELECT * FROM PRODUTOS  
    LIMIT quantidade ;  
END $$  
DELIMITER ;
```

```
CALL Selecionar_Produtos(2);
```

Utilização de *Stored Procedure*

- Podemos usar o *procedure* quantas vezes desejarmos, usando diferentes parâmetros
 - *“Isto é, um stored procedure é uma função para bancos de dados”*

```
CALL Selecionar_Produtos(5);  
CALL Selecionar_Produtos(10);
```

Exemplo de *Stored Procedure*

- Conte quantos os produtos existem no sistema

```
DELIMITER $$

CREATE PROCEDURE Verificar_Quantidade_Produtos(OUT quantidade INT)
BEGIN
SELECT COUNT(*) INTO quantidade FROM PRODUTOS;
END $$
DELIMITER ;
```

Exemplo de *Stored Procedure*

- Conte quantos os produtos existem no sistema

```
DELIMITER $$

CREATE PROCEDURE Verificar_Quantidade_Produtos(OUT quantidade INT)
BEGIN
SELECT COUNT(*) INTO quantidade FROM PRODUTOS;
END $$

DELIMITER ;
```

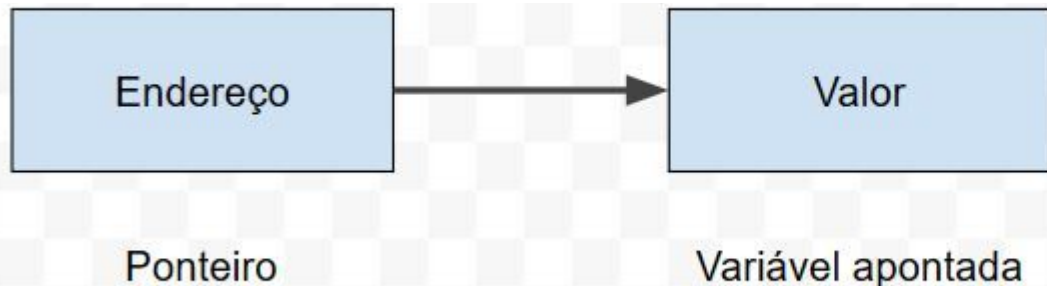
```
CALL Verificar_Quantidade_Produtos(@total);
SELECT @total;
```

Parâmetro OUT em *Stored Procedure*

- Um parâmetro OUT em um stored procedure é chamado usando @

```
CALL Verificar_Quantidade_Produtos(@total);
```

- O @ é como se fosse o & da linguagem C
 - Isto é, “**@total**” significa pegue o ponteiro para o endereço de memória da variável “**total**”



Parâmetro OUT em *Stored Procedure*

- Após o **CALL** abaixo, iremos fazer um **SELECT** para acessar “@total”

```
CALL Verificar_Quantidade_Produtos(@total);  
SELECT @total;
```

- “**SELECT @total**” é como se fosse um **printf(&total)** do C
 - Ou **print(total)** do Python
- Veremos uma analogia entre *Procedure (SQL)* <-> *Função (Python)* no próximo slide para facilitar nosso entendimento

Definição de Stored Procedure (SQL) x Função (Python)

SQL:

```
DELIMITER $$

CREATE PROCEDURE Verificar_Quantidade_Produtos(OUT quantidade INT)
BEGIN
SELECT COUNT(*) INTO quantidade FROM PRODUTOS;
END $$
DELIMITER ;
```

Python:

```
def Verificar_Quantidade_Produtos(total):
    total = 0
    for produto in produtos:
        total = total + 1
```

Chamada de Stored Procedure (SQL) x Função (Python)

SQL:

```
CALL Verificar_Quantidade_Produtos(@total);  
SELECT @total;
```

Python:

```
Verificar_Quantidade_Produtos(total)  
print(total)
```

Gatilhos (*Triggers*) SQL

- Permitem que uma ação seja executada quando uma determinada condição (evento) ocorrer
- *Triggers* podem ser usados para **monitorar** o banco de dados
 - Por isso, bancos de dados que contém triggers são chamados de **bancos de dados ativos** ou **orientados a eventos**
 - São usados em conjunto com as restrições de integridade para impor regras sobre os dados

Gatilhos (*Triggers*) SQL

- Sintaxe:

- **delimiter \$\$**

Comando **delimiter** altera o carácter que usamos para terminar o trigger.

```
CREATE TRIGGER <nome_trigger>  
[BEFORE | AFTER] [INSERT | DELETE | UPDATE]  
ON <tabela>  
[FOR EACH ROW]  
BEGIN  
<comandos_SQL>  
END $$
```

delimiter ;

Gatilhos (*Triggers*) SQL

- **Sintaxe:**

- **delimiter \$\$**

```
CREATE TRIGGER <nome_trigger>
[BEFORE | AFTER] [INSERT | DELETE | UPDATE]
ON <tabela>
[FOR EACH ROW]
BEGIN
<comandos_SQL>
END $$

delimiter ;
```

Comando **delimiter** altera o carácter que usamos para terminar o trigger.

DELIMITER é necessário pois iremos usar comandos SQL dentro do bloco **BEGIN END**, sendo que cada um deles precisa ser terminado com ;

Gatilhos (*Triggers*) SQL

- **Sintaxe:**

- **delimiter \$\$**

```
CREATE TRIGGER <nome_trigger>  
[BEFORE | AFTER] [INSERT | DELETE | UPDATE]  
ON <tabela>  
[FOR EACH ROW]  
BEGIN  
<comandos_SQL>  
END $$
```

delimiter ; ←

Após a definição do trigger, definimos o caractere ; como fim dos comandos SQL

Exemplo de Gatilhos (*Triggers*) SQL

- Desejamos que ao inserir e remover registro da tabela *ItensVenda*, o estoque do produto referenciado seja alterado na tabela *Produtos* do seguinte DB:

- **CREATE TABLE** Produtos (
 Referencia **VARCHAR(3) PRIMARY KEY**,
 Descricao **VARCHAR(50) UNIQUE**,
 Estoque **INT NOT NULL DEFAULT 0**
);

- CREATE TABLE** ItensVenda (
 Venda **INT PRIMARY KEY**,
 Produto **VARCHAR(3)**,
 Quantidade **INT**
);

INSERT INTO Produtos **VALUES** ("001", "Feijão", 10);
INSERT INTO Produtos **VALUES** ("002", "Arroz", 5);
INSERT INTO Produtos **VALUES** ("003", "Farinha", 15);

Exemplo de *Trigger* AFTER INSERT

- delimiter \$\$

```
CREATE TRIGGER Tgr_ItensVenda_Insert  
AFTER INSERT ON ItensVenda  
FOR EACH ROW  
BEGIN  
    UPDATE Produtos  
    SET Estoque = Estoque - NEW.Quantidade  
    WHERE Referencia = NEW.Produto ;  
END $$
```

```
delimiter ;
```


Exemplo de *Trigger* AFTER INSERT

- delimiter \$\$

```
CREATE TRIGGER Tgr_ItensVenda_Insert
```

```
AFTER INSERT ON ItensVenda
```

```
FOR EACH ROW
```

```
BEGIN
```

```
    UPDATE Produtos
```

```
    SET Estoque = Estoque - NEW.Quantidade
```

```
    WHERE Referencia = NEW.Produto ;
```

```
END $$
```

```
delimiter ;
```

Depois de INSERIR uma tupla na tabela
ItensVenda


Exemplo de *Trigger* AFTER INSERT

- delimiter \$\$

```
CREATE TRIGGER Tgr_ItensVenda_Insert  
AFTER INSERT ON ItensVenda
```

```
FOR EACH ROW  
BEGIN
```

Execute as seguintes instruções para cada tupla inserida



```
    UPDATE Produtos  
    SET Estoque = Estoque - NEW.Quantidade  
    WHERE Referencia = NEW.Produto ;  
END $$
```

```
delimiter ;
```

Exemplo de *Trigger* AFTER INSERT

- delimiter \$\$

```
CREATE TRIGGER Tgr_ItensVenda_Insert  
AFTER INSERT ON ItensVenda  
FOR EACH ROW  
BEGIN
```

```
    UPDATE Produtos  
    SET Estoque = Estoque - NEW.Quantidade  
    WHERE Referencia = NEW.Produto ;
```

```
END $$
```

```
delimiter ;
```

Defina Estoque = Estoque - **NEW**.Quantidade
para o produto cujo referencia é **NEW**.Produto

O comando **NEW** se refere a tupla que foi
inserida em *ItensVenda*

Exemplo de *Trigger* AFTER DELETE

- delimiter \$\$

```
CREATE TRIGGER Tgr_ItensVenda_Delete  
AFTER DELETE ON ItensVenda  
FOR EACH ROW  
BEGIN  
    UPDATE Produtos SET Estoque = Estoque + OLD.Quantidade  
    WHERE Referencia = OLD.Produto ;  
END $$
```

```
delimiter ;
```

Exemplo de *Trigger* AFTER DELETE

- delimiter \$\$

```
CREATE TRIGGER Tgr_ItensVenda_Delete
```

```
AFTER DELETE ON ItensVenda
```

```
FOR EACH ROW
```

```
BEGIN
```

```
    UPDATE Produtos SET Estoque = Estoque + OLD.Quantidade
```

```
    WHERE Referencia = OLD.Produto ;
```

```
END $$
```

```
delimiter ;
```

Depois de REMOVER uma tupla na tabela
ItensVenda


Exemplo de *Trigger* AFTER DELETE

- delimiter \$\$

```
CREATE TRIGGER Tgr_ItensVenda_Delete  
AFTER DELETE ON ItensVenda
```

```
FOR EACH ROW  
BEGIN
```

Execute as seguintes instruções para cada tupla removida



```
    UPDATE Produtos SET Estoque = Estoque + OLD.Quantidade  
    WHERE Referencia = OLD.Produto ;
```

```
END $$
```

```
delimiter ;
```

Exemplo de *Trigger* AFTER DELETE

- delimiter \$\$

```
CREATE TRIGGER Tgr_ItensVenda_Delete  
AFTER DELETE ON ItensVenda  
FOR EACH ROW  
BEGIN
```

```
    UPDATE Produtos SET Estoque = Estoque + OLD.Quantidade  
    WHERE Referencia = OLD.Produto ;
```

```
END $$
```

delimiter ;

O comando **OLD** se refere a tupla que foi removida de *ItensVenda*

Aumente o valor em estoque pela quantidade do item que havia sido vendido, mas foi removido da tabela *ItensVenda*

Exemplo de Testes com *Triggers*

- Vamos testar os triggers inserindo alguns produtos vendidos no DB:
 - **INSERT INTO** ItensVenda **VALUES** (1, "001", 3);
INSERT INTO ItensVenda **VALUES** (1, "002", 1);
INSERT INTO ItensVenda **VALUES** (1, "003", 5);
- Como ficou o DB ? Faça alguns SELECTs nas tabelas para ver o estado do DB.
- Agora vamos fazer o estorno da venda (remoção do produto vendido):
 - **DELETE FROM** ItensVenda **WHERE** Venda = 1 **AND** Produto = "001";
- Como ficou o estado do DB ?

Mostrando e Removendo Gatilhos (*Triggers*) SQL

- Mostrar triggers:
 - **SHOW TRIGGERS ;**
- Remover trigger:
 - **DROP TRIGGER** nome_do_trigger ;

Tutoriais sobre Triggers:

- <https://www.devmedia.com.br/mysql-basico-triggers/37462>
- <https://www.dolthub.com/blog/2023-06-09-writing-mysql-triggers/>
- <https://dev.mysql.com/doc/refman/8.4/en/trigger-syntax.html>

O que acontece se o gatilho (*Trigger*) SQL falhar?

- No MySQL, se qualquer comando SQL entre o **BEGIN** e o **END** do trigger falhar, a transação que causou o disparo da trigger também irá falhar
 - Isto pode ser algo interessante para garantir condições especiais como:
 - Estoque > 0 para poder vender um produto
 - Saldo > 0 para realizar uma transferência bancária
 - Dentre outros

Quando usar gatilhos (*Triggers*) SQL ?

- Apesar de ser tentador usar triggers para facilitar o desenvolvimento do DB, é necessário tomar alguns cuidados:
 - **Quanto mais triggers são usados, maior a sobrecarga no sistema**
 - Várias transações sendo executadas uma após a outra, cada uma com suas condições próprias (**CHECK, IF ELSE**, etc)
 - **Não há técnicas para verificar se os triggers estão corretos**
 - Por exemplo, não temos garantia que um trigger não viola alguma condição de uma tabela do DB

Referencial Bibliográfico

- KORTH, H.; SILBERSCHATZ, A.; SUDARSHAN, S. **Sistemas de bancos de dados**. 5. ed. Rio de Janeiro: Ed. Campus, 2006.
- DATE, C. J. **Introdução a sistemas de bancos de dados**. Rio de Janeiro: Ed. Campus, 2004. Tradução da 8ª edição americana.