

Bases de Dados

T17 - Transações & Vistas

Prof. Daniel Faria

Prof. Flávio Martins

Sumário

- Recapitulação Breve
- Transações
- Vistas

Recapitulação Breve

Divisão

- Que **porcos** realizaram vendas de produtos de **todas as espécies** de animais?

```
SELECT id
FROM pig p
WHERE NOT EXISTS (
  SELECT species
    FROM nonpig
  EXCEPT
  SELECT species
    FROM nonpig JOIN produce ON (id=producer) JOIN buys USING (code)
    WHERE seller=p.id
);
```

Quociente (porcos a retornar)

Divisor (espécies)

Dividendo
(espécies das vendas dos porcos)

Divisão: Solução Alternativa

- Que **porcos** realizaram vendas de produtos de **todas as espécies** de animais?

```
SELECT seller
```

```
FROM nonpig JOIN produce ON (id=producer) JOIN buys USING (code)
```

```
GROUP BY seller
```

```
HAVING COUNT(DISTINCT species) =
```

```
(SELECT COUNT(DISTINCT species) FROM nonpig; ← Divisor (espécies)
```



Dividendo (espécies das vendas dos porcos)



Divisor (espécies)



- Mais eficiente, mas mais arriscada
 - Basta não usar DISTINCT ou definir mal o divisor (não ter todos os valores possíveis) e o resultado é errado
- Usem a versão “textbook” para avaliação

SQL/PSM

- SQL/PSM (Persistent Stored Modules):
 - Linguagem de programação para funções e procedimentos armazenados no SGBD
 - Implementada no PostgreSQL sob o nome PL/pgSQL
 - Cobre todos os aspectos básicos de linguagens de programação tradicionais
 - Variáveis, if-else, ciclos, iteradores, ...

Funções

- Blocos de código (SQL ou PSM) que produzem um output
- Tipicamente não podem manipular dados, apenas consultá-los
 - Exceção: funções-trigger (cujo output é um trigger)
- Declaradas com [CREATE FUNCTION](#)
- Invocadas com SELECTs
 - Pode invocar-se só a função ou no meio de uma query mais complexa

Procedimentos

- Blocos de código (SQL ou PSM) que não podem produzir um output
- Podem manipular dados (inserir, atualizar, remover)
- Declarados com [CREATE PROCEDURE](#)
- Invocados com CALL
 - Não podem ser invocados em queries (SELECT)
 - Podem invocar funções mas não podem ser invocados por funções

Triggers

- Objetos que “vigiam” uma tabela especificada, despoletando quando são realizadas modificações nessa tabela e executando uma função-trigger
- Permitem implementar restrições de integridade sofisticadas bem como procedimentos que “resolvem” problemas no estado da BD
- Declarados com [CREATE TRIGGER](#)
- Não podem ser invocados (despoletam automaticamente)

Transações

Transação

- Conjunto de operações de um programa que formam uma unidade lógica de trabalho que pode aceder e atualizar vários dados
- Exemplos:
 - Transferir dinheiro da conta bancária A para a B
 - Reservar uma viagem

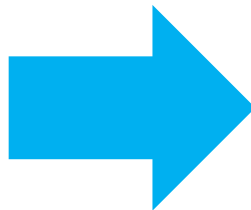
Transferir €50 da conta bancária A para a B:

1	T_i : read (A)
2	$A := A - 50$
3	write (A)
4	read (B)
5	$B := B + 50$
6	write (B)

Transação

- Nem todo o código de uma transação num sistema de informação é necessariamente SQL
 - Mas só nos vamos focar no SQL

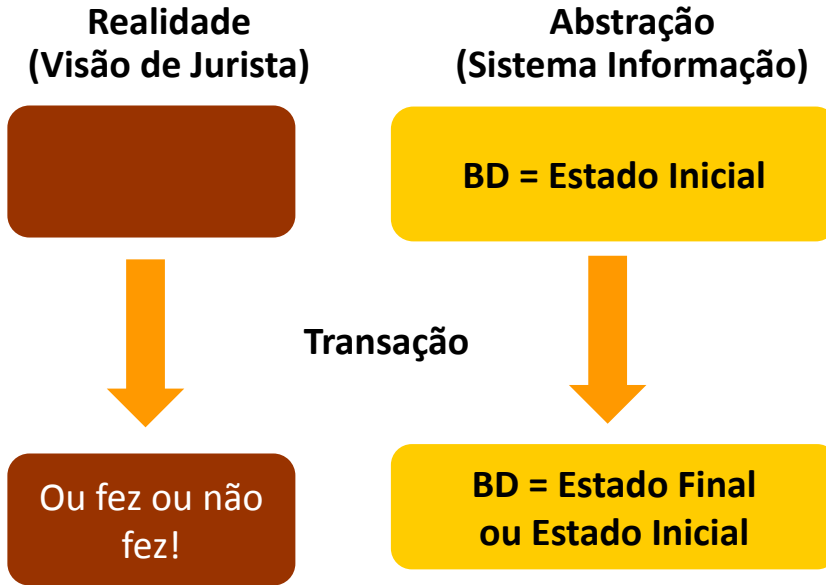
1	T_i : read (A)
2	$A := A - 50$
3	write (A)
4	read (B)
5	$B := B + 50$
6	write (B)



Pseudocódigo resultante de instruções SQL:

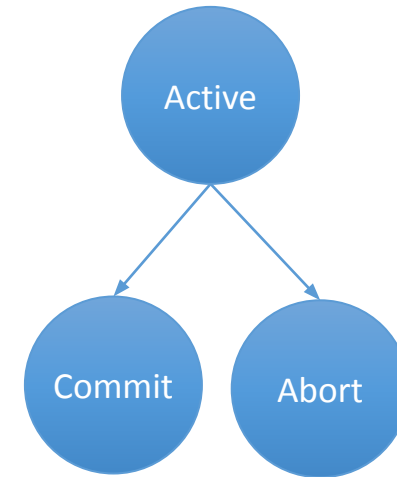
- SELECT
- UPDATE
- INSERT
- SELECT... INTO...

Abstração de um Sistema Transaccional



Modelo da Transação

BD = Estado Inicial



Tudo correu bem!

Nada foi feito!

BD = Estado Final

BD = Estado Inicial

2 Fins Possíveis

Duas Questões a Resolver

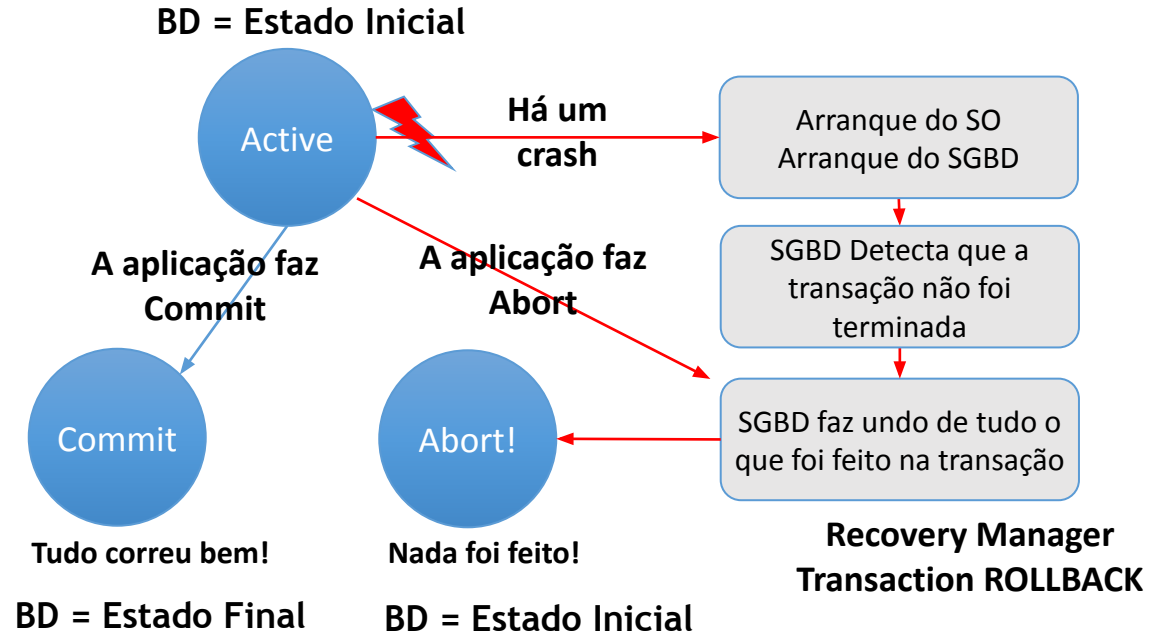
- **Concorrência:**

- Execução concorrente de várias transações
- SGBD suportam:
 - Múltiplos utilizadores simultâneos
 - Múltiplos processadores disponíveis

- **Integridade:**

- Lidar com falhas de vários tipos, nomeadamente de *hardware* e crashes do sistema operativo ou do *software* do SGBD
- SGBD asseguram integridade dos dados

Caminhos para a Conclusão



Propriedades ACID

- **Atomicidade:**

- E.g. se a transação falhar entre os passos 4–6, os passos 1–3 ficam sem efeito

- **Consistência**

- E.g. a soma $A+B$ tem de ser igual antes e depois da transação

- **Isolamento**

- E.g. nenhuma outra operação deve ler os valores de A e B entre os passos 3 e 6 (veria valores inconsistentes)

- **Durabilidade**

- Se a transação termina com sucesso, as alterações são definitivas, mesmo que haja falhas de *hardware* ou *software*

1	T_i : read (A)
2	$A := A - 50$
3	write (A)
4	read (B)
5	$B := B + 50$
6	write (B)

[A] Atomicidade

- Numa transacção, as alterações ao estado são **atómicas**
 - Ou todas se realizam ou nenhuma se realiza
- **Função do Sistema:**
 - Manter informação sobre as alterações efetuadas por cada transacção activa e, em caso de “crash” ou de “abort” explícito, desfazer as alterações realizadas desde o início da transacção até ao ponto em que falhou

1	T_i : read (A)
2	$A := A - 50$
3	write (A)
4	read (B)
5	$B := B + 50$
6	write (B)

[C] Consistência (Integridade)

- Uma transação é uma **transformação correta** do estado:
 - O conjunto das ações da transação não viola nenhuma das restrições de integridade do sistema, explícitas ou implícitas
 - Embora no decurso da transação o estado da BD possa ser temporariamente inconsistente
- **Função do Sistema:**
 - Assegurar que a BD evolui de um estado consistente para outro estado consistente, como definido pela lógica aplicacional

1	T_i : read (A)
2	$A := A - 50$
3	write (A)
4	read (B)
5	$B := B + 50$
6	write (B)

[I] Isolamento (Serializabilidade)

- Embora as transações se executem concorrentemente, os estados intermédios de uma transação devem ser invisíveis a todas as restantes transações
 - Estas apenas vêm o estado inicial ou o estado final
- **Função do Sistema:**
 - Garantir que uma transação apenas “vê” (*reads/writes*) alterações realizadas por transações *committed*

1	T_i : read (A)
2	$A := A - 50$
3	write (A)
4	read (B)
5	$B := B + 50$
6	write (B)

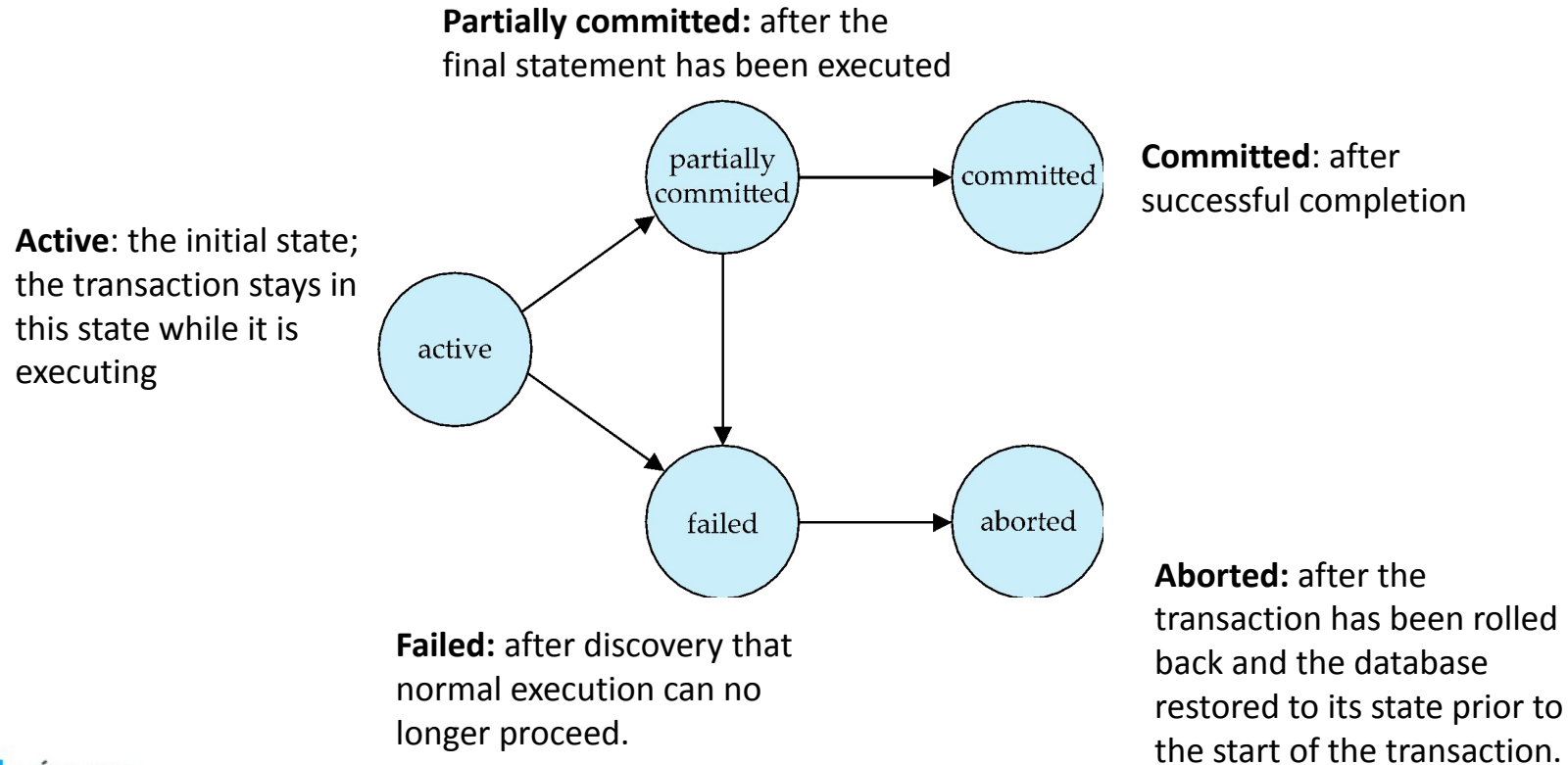
[D] Durabilidade (Persistência)

- Uma vez completada uma transação (*commit* concluído), todas as alterações ao estado são imutáveis, sobrevivendo a qualquer tipo de falta do sistema
- **Função do Sistema:**
 - Manter informação sobre alterações efectuadas por cada uma das transações *committed* e, em caso de “crash”, refazer as alterações que ainda não se encontravam registadas em disco
 - O “crash” dos discos é recuperado de outras formas (e.g. backups, redundância activa/passiva)

Modelo de Storage Pressuposto

- **Volatile Storage:**
 - Não sobrevive ao “crash” e “boot” do sistema.
- **Non-Volatile Storage:**
 - Sobrevive ao “crash” e “boot” do sistema.
- **Stable Storage:**
 - Tem redundância de *hardware* para tolerar N falhas
 - E.g. os vários tipos de RAID, com backup e mecanismos de recuperação

Transação: Diagrama de Estados



Definição de Transações em SQL

- Em SQL, qualquer consulta inicia implicitamente uma transacção, mas com uma única instrução
- Para incluir várias instruções SQL na mesma transação, deve-se preceder o grupo de instruções com
 - START TRANSACTION (ou BEGIN)
- E terminar com:
 - COMMIT: torna os resultados permanentes
 - ROLLBACK: desfaz todas as alterações feitas

Exemplo

```
START TRANSACTION;

--Verificar saldo
SELECT balance FROM account WHERE account_number = 'A-101';

--Transferir 350€ da conta A-101 para a conta A-102:
UPDATE account SET balance = balance - 350
    WHERE account_number = 'A-101';
UPDATE account SET balance = balance + 350
    WHERE account_number = 'A-102';

COMMIT;
```

<https://www.postgresql.org/docs/current/tutorial-transactions.html>

Exemplo

- Executar uma transacção
 - start transaction;**
 - ...
 - commit;** ou **rollback;**
- Vários sistemas usam *autocommit* por pré-definição
 - se **start transaction** for omitido
 - cada consulta é uma transacção
 - se houver erros, **rollback** automático
 - se não houver erros, **commit** automático

Transações em SQL

- Se START TRANSACTION for omitido
 - Cada consulta é uma transacção
 - Se houver erros, ROLLBACK automático
 - Se não houver erros, COMMIT automático
- Erros a meio de uma transação causam *abort*
 - Deixa de ser possível continuar a transação
- Podemos usar SAVEPOINT para gravar um ponto seguro e ROLLBACK TO para regressar a esse ponto
 - É a única maneira de recuperar do estado de *abort* sem fazer ROLLBACK total



Vistas

Vista

- Tabela virtual, definida através de uma query
- Uma vez criada, vale como uma tabela para efeito de outras queries, mas...
- Não tem materialização física
 - A query é corrida cada vez que a vista é referenciada
 - A vista pode mudar entre chamadas se as tabelas das quais deriva forem atualizadas

Vista

- Criar Vista: **CREATE VIEW**

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

- Remover Vista: **DROP VIEW**
- Aceder a Vista: **SELECT * FROM** name
 - Acede-se como a qualquer tabela

Vistas: Exemplo 1

```
CREATE VIEW account_stats AS
SELECT customer_name, COUNT(*) AS num_accts
FROM depositor
GROUP BY customer_name;
```

```
SELECT * FROM account_stats;
```

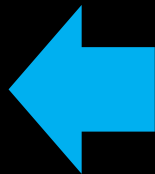
customer_name	num_accts
Johnson	2
Oliver	1
Brown	2
Evans	1
Flores	1
Cook	2
Iacocca	1

Vistas: Exemplo 2

```
CREATE VIEW customer_money AS
SELECT customer_name, SUM(balance) AS money
FROM depositor JOIN account USING(account_number)
GROUP BY customer_name;
```

```
SELECT customer_name
FROM customer_money
WHERE money >= ALL (
    SELECT money
    FROM customer_money);
```

```
customer_name
-----
Brown
```



Podemos fazer queries sobre a vista como qualquer tabela!

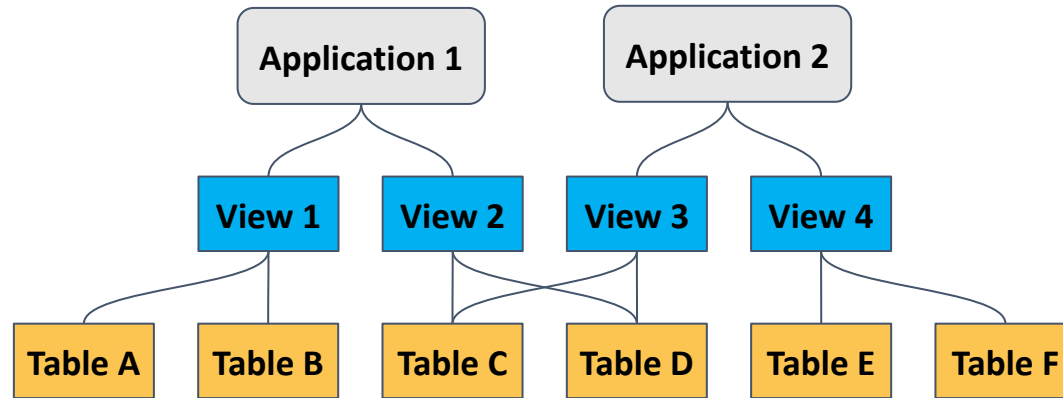
Vistas: Exemplo 2

- Referências à vista em queries são substituídas pela sua definição
 - Na realidade o que acontece no exemplo é que a vista é substituída por um select encadeado:

```
WITH customer_money AS
  (SELECT customer_name, SUM(balance) AS money
   FROM depositor JOIN account USING(account_number)
   GROUP BY customer_name)
SELECT customer_name
FROM customer_money
WHERE money >= ALL (
  SELECT money
  FROM customer_money) ;
```

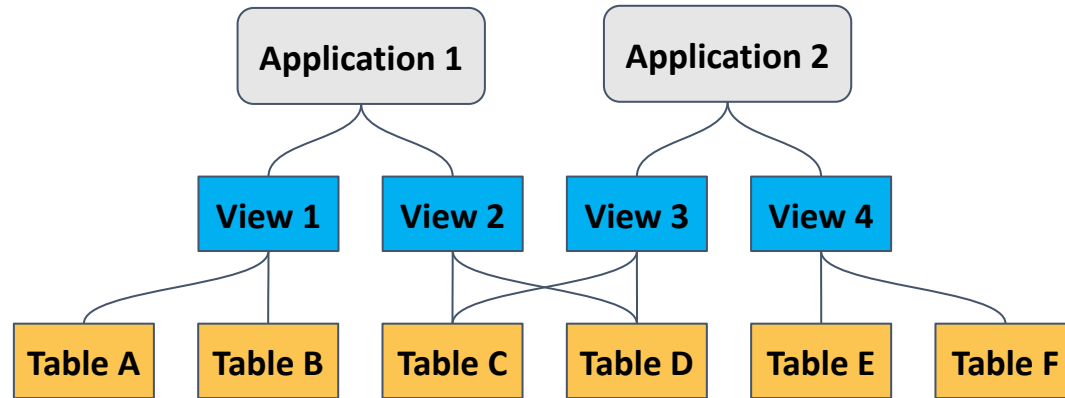

Vistas e Independência Lógica

- Vistas mapeiam as tabelas (modelo físico) para um novo modelo lógico que pode ser adaptado às necessidades das aplicações
- Permitem desenvolver aplicações com independência do modelo físico
- Dão a ilusão de tabelas distintas a aplicações distintas



Vistas e Segurança

- Vistas permitem acesso diferenciado aos dados por diferentes aplicações/utilizadores (ou grupos deles)
- Aplicações não têm acesso aos dados físicos



Vistas Atualizáveis / Updatable Views

- Uma vista é atualizável (*automatically updatable*) quando há uma correspondência direta entre a vista e uma única tabela (ou vista atualizável) na qual a vista é baseada
- Se a vista é atualizável, o SGBD permite operações INSERT, UPDATE e DELETE sobre a vista:
 - Estas operações são propagadas para a tabela em que a vista se baseia

Vistas Atualizáveis / Updatable Views

- Condições para uma vista ser atualizável:
 - Tem exactamente uma entrada (tabela ou vista atualizável) no FROM
 - Não contém cláusulas WITH, DISTINCT, GROUP BY, HAVING ou LIMIT no nível superior (i.e. SELECT externo)
 - Não contém operações de conjuntos (UNION, INTERSECT ou EXCEPT) no nível superior
 - O SELECT não pode conter agregados, ou funções que devolvem conjuntos

Vistas Atualizáveis / Updatable Views

- Uma vista atualizável pode conter colunas atualizáveis (referências simples a colunas atualizáveis) e outras não atualizáveis
 - Colunas não atualizáveis são “read-only” e tentativas de INSERT ou UPDATE resultam em erro
- Se uma vista atualizável contém um WHERE, isso restringe as linhas da tabela original que estão disponíveis para UPDATE e DELETE
 - Mas o UPDATE pode mudar a linha de forma a que não satisfaça o WHERE e deixe de ser visível
 - E é possível fazer INSERTs que não são visíveis na vista

Vista Atualizável: Exemplo

Vista atualizável

```
CREATE VIEW senior_employees AS  
SELECT e.eid, e.name AS emp_name, e.birthdate  
FROM employee e  
WHERE birthdate < '01-01-1980';
```

Podemos inserir, apagar ou
atualizar linhas

```
INSERT INTO senior_employees VALUES (7, 'Grace', '01-12-1979');  
(1 row affected)
```

```
UPDATE senior_employees SET birthdate = '01-12-1999' WHERE eid=7;  
(1 row affected)
```


Mesmo que isso leve a que as linhas
deixem de ser visíveis na vista

Vistas Atualizáveis / Updatable Views

- Vistas não atualizáveis são “read-only” por defeito
 - INSERT, UPDATE ou DELETE não são possíveis
- É possível fazer atualizações em vistas não atualizáveis através de triggers INSTEAD OF, que convertem operações sobre a vista em operações apropriadas nas tabelas das quais ela depende

Vista Não Atualizável: Exemplo

Vista não atualizável



```
CREATE VIEW customer_money AS
SELECT customer_name, SUM(balance) AS money
FROM depositor JOIN account USING(account_number)
GROUP BY customer_name;

UPDATE customer_money SET money = 1000 WHERE customer_name = 'Brown';
ERROR:  cannot update view "customer_money"
DETAIL:  Views containing GROUP BY are not automatically updatable.
HINT:   To enable updating the view, provide an INSTEAD OF UPDATE trigger
or an unconditional ON UPDATE DO INSTEAD rule.
```


Vistas Temporárias

- Extensão do PostgreSQL
- Vistas que só estão disponíveis durante uma sessão
 - São removidas automaticamente quando a sessão termina
- Podem ter o mesmo nome que uma relação (substituindo-a em todas as queries durante a sessão)

Vistas Materializadas

- O custo (CPU) de computar vistas “on-the-fly” pode ser demasiado elevado para queries complexas
- Podemos melhorar o desempenho com recurso a cache, i.e., manter a vista em memória ou escrevê-la em disco
- Uma vista materializada é uma vista que em vez de ser recomputada cada vez que é invocada, é materializada numa tabela quando criada, e só é recomputada quando atualizada explicitamente

Vistas Materializadas

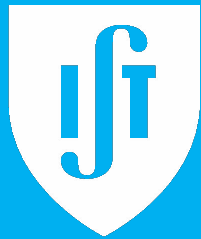
- Criar Vista: **CREATE MATERIALIZED VIEW**

```
CREATE MATERIALIZED VIEW [IF NOT EXISTS] table_name [(column_name [,...])]  
    AS query  
    [WITH [NO] DATA]
```

- Semelhante a **CREATE TABLE... AS**, exceto que a tabela memoriza a query que a criou
- Opção “**WITH NO DATA**” cria apenas a definição da vista sem a popular
- Atualizar Vista: **REFRESH MATERIALIZED VIEW**

Vistas vs. Vistas Materializadas

- **Vistas:**
 - Não ocupam storage mas requerem CPU para computar cada vez que invocadas
 - Estão sempre atualizadas
- **Vistas Materializadas:**
 - Ocupam storage mas só requerem CPU para atualizar
 - Podem não estar atualizadas
- Vistas são preferíveis para queries simples ou sobre as quais apenas serão feitas queries simples; vistas materializadas são melhores em casos de queries complexas



TÉCNICO LISBOA