Segurança

- Tópicos:
  - Segurança e Autorizações em SQL
- Bibliografia:
  - Secção 4.6 do livro recomendado

Segurança

- **Segurança** – ao contrário das restrições de integridade, que pretendiam proteger a base de dados contra estragos acidentais, a segurança preocupa-se com proteger a base de dados de estragos propositados.
  - A nível do sistema operativo
  - A nível da rede
  - A nível físico
  - A nível humano
  - A nível da base de dados
    - Mecanismos de autenticação e autorização para permitir acessos seletivos de (certos) utilizadores a (certas) partes dos dados

Autorizações

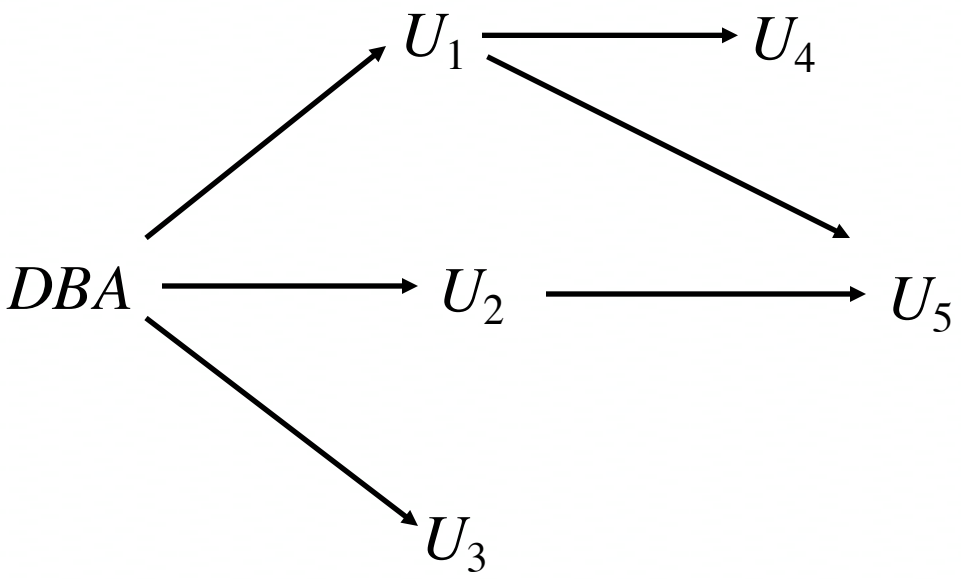
- **Autorização de leitura** – permite ler, mas não modificar dados.
- **Autorização de inserção** – permite inserir novos tuplos, mas não modificar tuplos existentes.
- **Autorização de modificação** – permite modificar tuplos, mas não apagá-los.
- **Autorização de remoção** – permite apagar tuplos
- **Autorização de resources** – permite criar novas relações.
- **Autorização de alteração** – permite criar e apagar atributos duma relação.
- **Autorização de drop** – permite apagar relações.
- **Autorização de index** – permite criar e apagar ficheiros de index.

Autorizações e Vistas

- Pode-se dar autorização a utilizadores sobre uma vista, sem se lhe dar autorização sobre as tabelas que a definem
  - Isto permite não só melhorar a segurança dos dados, como também tornar mais simples o seu uso
- Uma combinação de segurança a nível de tabelas, com segurança a nível de vistas, pode ser usada para limitar o acesso de um utilizador apenas aos dados de que ele necessita
- A criação de uma vista não requer autorização **resources** pois, de facto, nenhuma nova tabela é criada
- Quem cria uma vista , fica exatamente com os mesmo privilégios sobre esta que tinha sobre as tabelas.
  - E.g. o criador duma vista *cust_loan* sobre as tabelas *borrower* e *loan*, que só tenha autorização de leitura sobre estas tabelas, só fica com autorização de leitura sobre a vista que criou

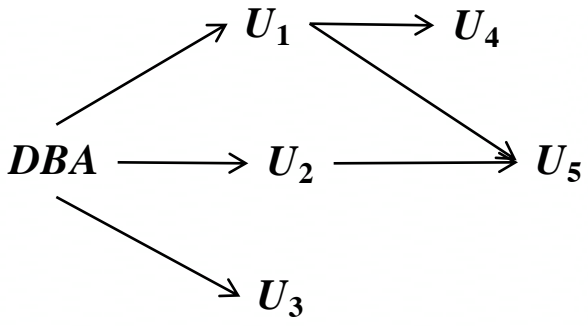
Atribuição de Privilégios

- A passagem de privilégios de um utilizador para outro pode ser representada por um grafo de autorizações.
- Os nós do grafo são utilizadores.
- A raiz é o administrador da base de dados.
- Considere o grafo abaixo, para e.g. escrita numa relação.
  - Um arco $U_i \rightarrow U_j$ indica que o utilizador U_i atribuiu ao utilizador U_j privilégio de escrita sobre essa relação.

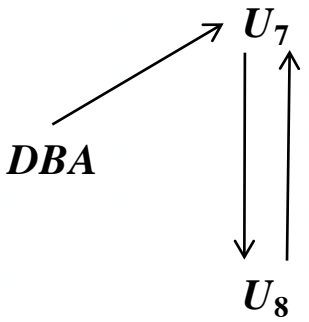


Grafo de atribuição de privilégios

- *Requisito*: Todos os arcos têm que fazer parte de algum caminho com origem no administrador.



- Se o administrador retira o privilégio a U_1:
  - Deve ser retirado privilégio a U_4 (pois U_1 já não tem autorização)
  - Não deve ser retirado a U_5 (pois U_5 tem autorização vinda de U_2)
- Devem ser prevenidos ciclos:
  - Administrador dá privilégios a U_7
  - U7 dá privilégios a U_8
  - U8 dá privilégios a U_7
  - DBA retira privilégios de U_7



- Deve retirar autorização de U_7 para U_8 e de U_8 para U_7 (pois já não há caminho do administrador nem para U_7 nem para U_8).

Especificações de Segurança em SQL

- O comando **grant** é usado para atribuir privilégios

  grant <lista de privilégios>

  on <nome de relação ou view> **to** <lista de utilizadores>

- <lista de utilizadores> é:
  - Um user-id
  - *public*, o que atribui o privilégios a todos os utilizadores
  - Um perfil (*role*) – veremos à frente
- Quem atribui o privilégio tem que o ter (ou ser o administrador da base de dados).

Privilégios em SQL

- **select:** permite acesso de leitura sobre a relação ou vista
  - Exemplo: dar a U_1, U_2, e U_3 autorização de leitura **na relação** *branch*:

    grant select on *branch* **to** U_1, U_2, U_3
- **insert**: permite inserir tuplos
- **update**: permite usar o comando **update** do SQL
- **delete**: permite apagar tuplos.
- **references**: permite a declaração de chaves externas.
- **all privileges**: forma sumária de atribuir todos os privilégios.

Privilégio de atribuir privilégios

- **with grant option**: autoriza um utilizador a passar um privilégio a outros utilizadores.
  - Exemplo:

    grant select on *branch* **to** U_1 **with grant option**

    dá a U_1 o privilégio **select** sobre a relação *branch* e autoriza U_1 a passar esse privilégio a qualquer outro utilizador

Perfis

- Um perfil permite atribuir, de apenas uma vez, privilégios iguais para uma classe de utilizadores
- Podem ser atribuídos e retirados privilégios a perfis de utilizadores, da mesma forma que a utilizadores isolados.
- Podem-se associar perfis a utilizadores, ou mesmo a outros perfis
- Exemplo:

create role *caixa*
create role *gerente*

grant select on *branch* **to** *caixa*
grant update (*balance***) on** *account* **to** *caixa*
grant all privileges on *account* **to** *gerente*

grant *caixa* **to** *gerente*

grant *caixa* **to** *maria, scott*
grant *gerente* **to** *ana*

Retirar de privilégios em SQL

- O comando **revoke** serve para retirar privilégios.

  revoke <privilégios>

  on <relação ou view> **from** <utilizadores> [**restrict**|**cascade**]

- Exemplo:

  revoke select on *branch* **from** U_1, U_2, U_3 **cascade**

- Se se colocar **cascade**, retirar privilégios de um utilizador também os pode retirar a outros, conforme descrito pelo grafo.

- Se se usar **restrict** só é retirado privilégio a esse utilizador

  revoke select on *branch* **from** U_1 **restrict**

  Com **restrict**, o comando **revoke** falha (dá erro) se a remoção do privilégio ao utilizador implicar a remoção do mesmo privilégio a outros utilizadores (e.g. por o utilizador ter dado o privilégio a outros utilizadores que não o detenham através de outros utilizadores)

Retirar de privilégios em SQL (Cont.)

- <privilégios> pode ser **all**. Nesse caso são retirados todos os privilégios *que foram atribuídos pelo utilizador* que deu o comando.
- Se <utilizadores> incluir **public** todos os utilizadores perdem esse privilégio, *a não ser que lhe tenha sido atribuído explicitamente*.
- Se o mesmo privilégio for atribuído duas vezes por utilizadores diferentes, então quem o tem pode ficar com ele mesmo depois dum **revoke** (cf. grafo).
- Todos os privilégios que dependem do privilégio retirado, são também retirados.

Limitação a autorizações em SQL

- SQL não permite autorizações a nível de tuplo
  - E.g. não se pode restringir de forma a que um aluno só possa ver as suas notas.
- Neste caso, a tarefa de autorização cai sobre as aplicações
  - É indesejável, mas o SQL aqui não ajuda
- Ou então definir vistas e dar autorizações apenas a essas vistas

Datalog e Recursão

- Tópicos:
  - Datalog para interrogação de Bases de Dados
  - Interrogações recursivas em SQL
- Bibliografia:
  - Secções C.3 e 5.4 do livro recomendado

Outras linguagens de Interrogação

- O SQL é, de longe, a linguagem mais usada para interrogação de bases de dados relacionais
  - Mas não é a única!
- Há outra linguagens de interrogação, que são baseadas em SQL
  - Linguagens visuais para "tornar mais fácil" o uso por utilizadores não especialistas (por exemplo o *QBE – Query By Example*)
    - Estas linguagens apenas "implementam" subconjuntos do SQL
    - O ser mais fácil é altamente discutível – especialmente para informáticos!
- O Datalog é uma linguagem que não se baseia em SQL, e que até é mais expressiva que o SQL
  - É baseada na programação em lógica
    - Linguagens de programação declarativas – Prolog
  - Não tem tanto uso comercial, mas serve de base teórica para várias extensões ao SQL, e a linguagens de interrogação em outros modelos que não o relacional (que verão e.g. em *Representação do Conhecimento e Sistemas de Raciocínio* e em *Modelação de Dados*)

Estrutura de Base de Datalog

- Um programa Datalog é um conjunto de regras, que definem relações (vistas).

- Exemplo: definir relação (*vista*) *v1* com todos os números de conta e respetivos saldos, para as contas de Perryridge com saldo superior a 700.

  *v1(A, B) :– account(A, "*Perryridge", *B), B* > 700.

- Qual o saldo da conta "A-217" na *vista v1?*

  *?- v1("*A-217", *B).*

- Quais as contas com saldo superior a 800?

  ?- account(A,_,B), B > 800

Exemplos de perguntas

- Cada regra define um conjunto de tuplos que pertencem à *vista*
  - E.g. *v1(A, B) :– account(A, "*Perryridge", *B), B* > 700
    lê-se como

    para todo *A, B*

    se (*A*, "Perryridge", *B*) ∈ *account* **e** *B* > 700

    então (*A, B*) ∈ *v1*
- O conjunto de tuplos duma *view* é definido como a união dos conjuntos de tuplos de cada uma das suas regras.
- Exemplo:

  interest_rate(N, 5*) :– account(N, _, B), B* < 10000
  interest_rate(N, 6*) :– account(N, _, B), B* >= 10000

Negação em Datalog

- Definir a *view c* contendo os nomes de todos os clientes que têm pelo menos um depósito mas não têm empréstimos:

  c(N) :– *depositor(N, A),* **not** *is_borrower(N).*
  is_borrower(N) :– *borrower (N,L).*

- NOTA: usando diretamente **not** *borrower (N, L)* na 1ª regra dava resultado diferente: clientes *N* que têm pelo menos um depósito e existe algum empréstimo *L* que não é de *N*
  - Para evitar confusão (e problemas de implementação), é usual exigir-se que as variáveis em literais negados apareçam na cabeça da regra ou em algum literal positivo do corpo da regra.

Regras por camadas (views sobre views)

- Quais os juros de cada uma das contas de Perryridge

  interest_rate(N,0) :– account(N, A, B), B < 2000.

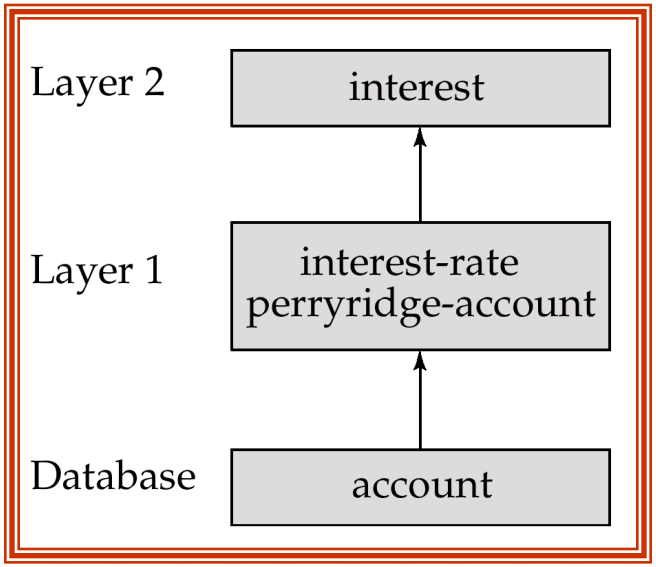
  interest_rate(N,5) :– account(N, A, B), B >= 2000.

  perryridge_account(A,B) :– *account*(*A,* “Perryridge”, *B).*

  interest(A, I) :– *perryridge_account*(*A,B),*

  *interest_rate(A,R), I = B * R*/100*.*

- Camadas das várias views:



Sintaxe de regras Datalog

- Um *literal positivo (ou átomo)* tem a forma

$$p(t_1, t_2 \ldots, t_n)$$

  - p é o nome da relação, com n atributos
  - cada t_i é uma constante ou uma variável
    - As variáveis começam por maiúsculas, ou por _
- Um *literal negativo (ou por default)* tem a forma

$$\textbf{not } p(t_1, t_2 \ldots, t_n)$$

- Operadores de comparação são entendidos como predicados
  - E.g. $X > Y$ é entendido como o átomo $>(X, Y)$
  - Conceptualmente “>” define uma relação (infinita) contendo todos os pares (X,Y) onde X é maior que Y.
- Operações aritméticas são entendidas como predicados
  - E.g. A is B + C é entendido como +(B, C, A), onde a relação (infinita) “+” tem todos os triplos em que o 3º valor é igual à soma dos primeiros 2.

Sintaxe de regras Datalog (Cont.)

- *Regras* são da forma:

$$p(t_1, t_2, ..., t_n) :- L_1, L_2, ..., L_m.$$

  - cada L_i é um literal (positivo ou negativo)
  - cabeça – átomo $p(t_1, t_2, ..., t_n)$
  - corpo – literais $L_1, L_2, ..., L_m$
- A uma regra com corpo vazio chama-se um *facto*:

$$p(v_1, v_2, ..., v_n).$$

  - afirma que o tuplo $(v_1, v_2, ..., v_n)$ pertence à relação p
- Um programa Datalog é um conjunto (finito) de regras
- Um programa diz-se definido, se nenhuma das suas regras contém literais negativos (i.e. se **not** não aparece no programa).

Expressividade de Datalog

- A linguagem de interrogação Datalog é pelo menos tão expressiva quanto a álgebra relacional
  - Para mostrar que isto é verdadeiro, basta mostrar que todos os operadores da álgebra relacional se podem expressar através de regras em Datalog

Operações Relacionais em Datalog

- Projeção do atributo *account-name* de *account*:

  query(Account-name) :– *account(Account-name, N, B).*

- Produto cartesiano das relações r_1 e r_2.

  $query(X_1, X_2, \ldots, X_n, Y_1, Y_2, \ldots, Y_m)$:–
  $r_1(X_1, X_2, \ldots, X_n), r_2(Y_1, Y_2, \ldots, Y_m).$

- União das relações r_1 e r_2.

  $query(X_1, X_2, \ldots, X_n)$:– $r_1(X_1, X_2, \ldots, X_n)$.
  $query(X_1, X_2, \ldots, X_n)$:– $r_2(X_1, X_2, \ldots, X_n)$.

- Seleção de tuplos de r_1 que obedecem a condição *cond*.

  $query(X_1, \ldots, X_n)$:– $r_1(X_1, \ldots, X_n)$, $cond(X_1, \ldots, X_n)$.

- Diferença entre r_1 e r_2.

  $query(X_1, X_2, \ldots, X_n)$:– $r_1(X_1, X_2, \ldots, X_n)$,
  not $r_2(X_1, X_2, \ldots, X_n)$.

Expressividade de Datalog

- A linguagem de interrogação Datalog é pelo menos tão expressiva quanto a álgebra relacional
  - Para mostrar que isto é verdadeiro, basta mostrar que todos os operadores da álgebra relacional se podem expressar através de regras em Datalog

- A linguagem Datalog é mais expressiva que a álgebra relacional
  - Há queries que se conseguem expressar em Datalog e que não se conseguem expressar em álgebra relacional (já veremos exemplos)
  - A maior expressividade vem do facto de em Datalog se poderem definir views recursivas

Recursão em Datalog

- Considere a relação:

  precedencia(X, Y)

  contendo pares X, Y de cadeiras onde X dá precedência direta para Y.

- Cada cadeira pode dar precedência para outras, direta ou indiretamente
  - Uma cadeira X dá precedência indiretamente a Y, se dá precedência direta a uma cadeira Z, que por sua vez dá precedência para Y
- Encontrar todas as cadeiras que dão precedência (direta ou indireta) a Bases de Dados. Podemos escrever o seguinte programa Datalog **recursivo**:

  precBD(X) :- precedencia(X, 'BD').

  precBD(X) :- precedencia(X, Y), precBD(Y).

Recursão em Datalog (Cont.)

- De forma mais geral: criar view *precs* com todos os pares *(X, Y)* onde *X* dá precedência directa ou indirecta para *Y:*

  precs(X, Y) :– *precedencia(X, Y).*
  precs(X, Z) :– *precedencia(X,Y), precs(Y,Z).*

- Quais as cadeiras que dão precedência a BD

  ?- precs(X, 'BD').

- E quais as cadeiras que têm precedência de BD

  ?- precs('BD', X).

Exemplo de pergunta complexa

- Questão 14 da ficha 5, de problema SQL:
  - Quais os nomes dos alunos que não seguiram pelo menos uma das precedências (diretas) aconselhadas, e em que cadeiras não as seguiram?

```
nao_fez(NAluno,CCurso,CCadeira) :- curso_cadeira(CCurso,CCadeira,_Semestre),
                                   not esta_inscrito(NAluno,CCurso,Ccadeira).
esta_inscrito(NA,CCr,CCd) :- inscricoes(NA,CCr,CCd,_DIn,_DAv,_Nt).

query(Nome,Cadeira) :- alunos(Num,Nome,_L,_Dt,_Sx,_Cr),
                       inscricoes(Num,CCurso,CCadeira,_DIn,_DAv,Nt),
                       precedencias(CCurso,CCadeira,CadP),
                       nao_fez(Num,CCurso,CadP),
                       cadeiras(CCadeira,Cadeira,_C,_D).

query(Nome,Cadeira) :- alunos(Num,Nome,_L,_Dt,_Sx,_Cr),
                       inscricoes(Num,CCurso,CCadeira,DIn,_DAv,_Nt),
                       precedencias(CCurso,CCadeira,CadP),
                       inscricoes(Num,CCurso,CadP,_DIn,DAv,_Nt),
                       DAv > DIn,
                       cadeiras(CCadeira,Cadeira,_C,_D).
```

- E se fossem precedências indiretas, bastava substituir *precedencias* por *precs*
  - Em álgebra relacional não se consegue expressar a query com precedências indiretas!

Semântica duma regra

- Uma *instância* duma regra é o resultado de substituir cada uma das variáveis da regra por alguma constante.
  - E.g. Sendo *v1*

    v1(A,B) :– *account (A,*"Perryridge", *B), B* > 700.
  - Uma instância de *v1* é:

    *v1("*A-217", 750) :–*account(*"A-217", "Perryridge", 750), 750 > 700.
- O corpo duma instância duma regra *R'* é *satisfeito* numa instância da base de dados, ou num conjunto de átomos *I* sse
  1. Para cada átomo $q_i(v_{i,1}, ..., v_{i,ni}) \in R'$, *I* contem $q_i(v_{i,1}, ..., v_{j,ni})$
  2. Para cada literal **not** $q_j(v_{j,1}, ..., v_{j,ni}) \in R'$, *I* não contem $q_j(v_{i,1}, ..., v_{j,ni})$

Restrições nas variáveis

- É possível escrever regras com um nº infinito de soluções.

  maior(X, Y) :– *X* > *Y*

  not_in_loan(B, L) :– **not** *loan(B, L)*

- Para evitar este problema, em Datalog:
  - Toda a variável que apareça na cabeça duma regra tem que aparecer também no corpo dessa regra, e num átomo que não seja expressão aritmética.
  - Toda a variável que apareça num literal negativo no corpo duma regra tem que aparecer também num átomo do corpo dessa mesma regra.

Semântica de Programas Definidos

- Seja *I* um conjunto de átomos. Define-se, para um programa definido *P*:

  $T_P(I) = \{p(t_1, t_2, ..., t_n)$: existe uma instância *R'* duma regra *R* de P com cabeça $p(t_1, t_2, ..., t_n)$ cujo corpo está satisfeito em *I*$\}$

- A semântica (significado) de um programa definido P sobre uma instância duma base de dados *I* é o resultado do menor ponto-fixo da sequência:
  - $I_0 = I$ (sendo P definido, esta sequência é não decrescente)
  - $I_{n+1} = T_P(I_n)$
- O resultado duma vista *v* com *k* argumentos, num programa P e base de dados *I* é o conjunto de todos os factos $v(t_1, t_2, ..., t_k)$ pertencentes ao ponto-fixo.

Monotonicidade

- Uma vista V é **monotónica** se para qualquer par de conjunto de factos I_1 e I_2 tais que $I_1 \subseteq I_2$, se tem $E_V(I_1) \subseteq E_V(I_2)$, onde E_V é a expressão que define V.
- Um programa Datalog P é monotónico sse para todo o I_1 e I_2:

  $I_1 \subseteq I_2$ implica $T_P(I_1) \subseteq T_P(I_2)$,

  - As expressões que se exprimem em álgebra relacional usando os operadores $\prod$, σ, $\times$, $\cup$ e ρ (bem como outros – e.g. joins – definidos à custa destes) são monotónicas.
  - As expressões que se exprimem com '–' podem não ser monotónicas.
- Da mesma forma, programas Datalog definidos são monotónicos, mas programas Datalog com negação podem não o ser

Não-monotonicidade

- A semântica à custa do operador T_P só funciona em programas monotónicos
  - Algumas conclusões numa iteração, podem deixar de o ser numa iteração posterior – possibilidade de não convergência para um ponto fixo. E.g.

    other_accounts(Num) :- **not** *perryridge_account(Num).*
    perryridge_account(N) :–*account*(*Num,* “Perryridge”, *_B).*

    Na 1ª iteração uma conta doutra sucursal aparece na view *other_accounts* mas desaparece na 2ª iteração.
- Pode-se estender a iteração do T_P para lidar com negação em programas estratificados (i.e. sem recursão sobre negação)

Programas Estratificados

- Um programa Datalog é estratificado se cada um dos seus predicados pode ser colocado num estrato de tal forma que:
  1. Para cada literal positivo q no corpo duma regra com cabeça p
     p(..) :- …., q(..), …
     o estrato de p é maior ou igual do que o estrato de q
  2. Dada uma regra com um literal negativo no corpo
     p(..) :- …, not q(..), …
     o estrato de p é estritamente maior que o estrato de q
- Por outras palavras, recursão e negação não se misturam

Programas Estratificados

- A semântica dum programa estratificado é definida estrato a estrato, começando pelo menor deles (que não tem negação).
- Em cada estrato usa-se a iteração T_P
- Calculado o resultado final da iteração de T_P num estrato n, usa-se para nos "livrarmos" da negação no estrato n+1:
  - Seja not q um literal negativo que aparece no corpo duma instância duma regra do estrato n+1:
    - Se q $\in$ T_P dum estrato anterior → apague-se a instância
    - Caso contrário → apague-se o not q do corpo dessa instância
- Como os estratos inferiores são calculados antes, os seus factos não mudam. Logo, os resultados de cada estrato são monotónicos, se fixados os resultados dos estratos anteriores

- Também é possível definir vistas recursivas em programas não estratificados. Mas isso fica para *Representação do Conhecimento*

Recursão em SQL

- O standard SQL foi estendido para também permite definição recursiva de vistas
- Exemplo: encontrar todos os pares empregado-chefe, onde o empregado responde directa ou indirectamente ao chefe (i.e. ao chefe do chefe do chefe, etc.)

```
with recursive empl (employee_name, manager_name ) as (
        select employee_name, manager_name
        from   manager
    union
        select manager.employee_name, empl.manager_name
        from   manager, empl
        where manager.manager_name = empl.employe_name)
select *
from   empl
```

Esta vista, empl, é o fecho transitivo da relação *manager*

O poder da recursão

- As vistas recursivas permitem a escrita de consultas, tais como as de fecho transitivo, que não podem ser escritas sem recursão (ou iteração).
  - Intuição: Sem recursão, um programa não-iterativo e não-recursivo só pode calcular um número fixo de junções de *manager* consigo própria.
    - Isto só pode fornecer um número fixo de níveis de chefias
- Cálculo do fecho transitivo
  - Cada passo do processo iterativo constroi uma versão estendida de *empl a partir da sua definição recursiva*.
  - O resultado final é chamado de *ponto-fixo da definição recursiva da vista*.
- As vistas recursivas têm que ser *monotónicas.* I.e. se acrescentarmos tuplos à relação *manager*, a vista contém todos os tuplos que continha anteriormente, e possivelmente mais
  - Em Datalog podia haver recursão mesmo em vistas não-monotónicas
  - Ou seja, mesmo com recursão o SQL é menos expressivo que Datalog

Exemplo de uma computação ponto-fixo

```
with recursive empl (emp_name, man_name) as
        select emp_name, man_name
        from manager
    union
        select manager.emp_name, empl.man_name
        from manager, empl
        where manager.man_name = empl.emp_name
select *
from empl
```

manager

emp_name	*man_name*
António	Bruno
Bruno	Eduardo
Carla	Duarte
Duarte	João
Eduardo	João
João	Lara
Rita	Lara

1ª iteração
2ª iteração
3ª iteração
4ª iteração
5ª iteração

Ponto Fixo

empl

emp_name	*man_name*
António	Bruno
Bruno	Eduardo
Carla	Duarte
Duarte	João
Eduardo	João
João	Lara
Rita	Lara
António	Eduardo
Bruno	João
Carla	João
Duarte	Lara
Eduardo	Lara
António	João
Bruno	Lara
Carla	Lara
António	Lara