Bases de Dados

INTRODUÇÃO AO SQL (PARTE 3)

Adaptado de Silberschatz, Korth e Sudarschan

Joins

Operações de join

Três tipos de junções:

- Natural join
- Inner join
- Outer join

Natural join em SQL

- O natural join seleciona apenas os tuplos que têm os mesmos valores para todos os atributos comuns (mantém apenas uma cópia de cada atributo comum).

- Exemplo:

  Liste os nomes dos professores e o ID das disciplinas que eles ministraram

  - **select** *name*, *course_id*
    from *students, takes*
    where *student.ID = takes.ID*;

- A mesma query SQL pode ser escrita usando “natural join”
  - **select** *name*, *course_id*
    from *student* **natural join** *takes*;

Natural join em SQL (continuação)

A cláusula **from** pode ter múltiplas relações combinadas usando **natural join**:

select $A_1, A_2, \ldots A_n$
from r_1 **natural join** r_2 **natural join .. natural join** r_n
where P ;

Relação student

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

Relação takes

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

student natural join takes

ID	*name*	*dept_name*	*tot_cred*
00128	Zhang	Comp. Sci.	102
12345	Shankar	Comp. Sci.	32
19991	Brandt	History	80
23121	Chavez	Finance	110
44553	Peltier	Physics	56
45678	Levy	Physics	46
54321	Williams	Comp. Sci.	54
55739	Sanchez	Music	38
70557	Snow	Physics	0
76543	Brown	Comp. Sci.	58
76653	Aoi	Elec. Eng.	60
98765	Bourikas	Elec. Eng.	98
98988	Tanaka	Biology	120

ID	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	CS-101	1	Fall	2017	A
00128	CS-347	1	Fall	2017	A-
12345	CS-101	1	Fall	2017	C
12345	CS-190	2	Spring	2017	A
12345	CS-315	1	Spring	2018	A
12345	CS-347	1	Fall	2017	A
19991	HIS-351	1	Spring	2018	B
23121	FIN-201	1	Spring	2018	C+
44553	PHY-101	1	Fall	2017	B-
45678	CS-101	1	Fall	2017	F
45678	CS-101	1	Spring	2018	B+
45678	CS-319	1	Spring	2018	B
54321	CS-101	1	Fall	2017	A-
54321	CS-190	2	Spring	2017	B+
55739	MU-199	1	Spring	2018	A-
76543	CS-101	1	Fall	2017	A
76543	CS-319	2	Spring	2018	A
76653	EE-181	1	Spring	2017	C
98765	CS-101	1	Fall	2017	C-
98765	CS-315	1	Spring	2018	B
98988	BIO-101	1	Summer	2017	A
98988	BIO-301	1	Summer	2018	*null*

ID	*name*	*dept_name*	*tot_cred*	*course_id*	*sec_id*	*semester*	*year*	*grade*
00128	Zhang	Comp. Sci.	102	CS-101	1	Fall	2017	A
00128	Zhang	Comp. Sci.	102	CS-347	1	Fall	2017	A-
12345	Shankar	Comp. Sci.	32	CS-101	1	Fall	2017	C
12345	Shankar	Comp. Sci.	32	CS-190	2	Spring	2017	A
12345	Shankar	Comp. Sci.	32	CS-315	1	Spring	2018	A
12345	Shankar	Comp. Sci.	32	CS-347	1	Fall	2017	A
19991	Brandt	History	80	HIS-351	1	Spring	2018	B
23121	Chavez	Finance	110	FIN-201	1	Spring	2018	C+
44553	Peltier	Physics	56	PHY-101	1	Fall	2017	B-
45678	Levy	Physics	46	CS-101	1	Fall	2017	F
45678	Levy	Physics	46	CS-101	1	Spring	2018	B+
45678	Levy	Physics	46	CS-319	1	Spring	2018	B
54321	Williams	Comp. Sci.	54	CS-101	1	Fall	2017	A-
54321	Williams	Comp. Sci.	54	CS-190	2	Spring	2017	B+
55739	Sanchez	Music	38	MU-199	1	Spring	2018	A-
76543	Brown	Comp. Sci.	58	CS-101	1	Fall	2017	A
76543	Brown	Comp. Sci.	58	CS-319	2	Spring	2018	A
76653	Aoi	Elec. Eng.	60	EE-181	1	Spring	2017	C
98765	Bourikas	Elec. Eng.	98	CS-101	1	Fall	2017	C-
98765	Bourikas	Elec. Eng.	98	CS-315	1	Spring	2018	B
98988	Tanaka	Biology	120	BIO-101	1	Summer	2017	A
98988	Tanaka	Biology	120	BIO-301	1	Summer	2018	*null*

Natural join pode ser “perigoso”

Cuidado com atributos com o mesmo nome que são igualados incorretamente

Devem ver sempre com detalhe todos os nomes dos atributos das relações envolvidas.

Apesar de poder “poupar” na escrita da query, é conveniente escrever uma query equivalente sem natural join

Condição de junção

- A condição **on** permite um predicado geral para definir a condição de junção
- Este predicado é escrito como um predicado de cláusula **where,** exceto pelo uso da palavra-chave **on**
- Exemplo

```
select *
from  student join takes on student_ID  = takes_ID
```

  - A condição **on** acima especifica que um tuplo de *student* corresponde a um tuplo de *takes* se seus valores *de ID* forem iguais.

- Equivalente a:

```
select *
from  student , takes
where  student_ID  = takes_ID
```

Outer join

- Tal como vimos em álgebra relacional, é uma extensão da operação de junção que evita perda de dados.
- Calcula a junção e adiciona ao resultado tuplos de uma relação que não têm correspondência com tuplos da outra relação.
- Usa valores *null*.
- Três formas de outer join:
  - left outer join
  - right outer join
  - full outer join

Exemplos

Course

course_id	*title*	*dept_name*	*credits*
BIO-301	Genetics	Biology	4
CS-190	Game Design	Comp. Sci.	4
CS-315	Robotics	Comp. Sci.	3

Prereq

course_id	*prereq_id*
BIO-301	BIO-101
CS-190	CS-101
CS-347	CS-101

- Observe que:
  - Falta informação sobre a disciplina de código CS-347
  - Não há prerequisitos para a disciplina de código CS-315

Left outer join

course **natural left outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*

- Em álgebra relacional: *course* ⟕ *prereq*

Right outer join

course **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

- Em álgebra relacional: : *course* ⟖ *prereq*

Full outer join

course **natural full outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Em álgebra relacional: : *course* ⟗ *prereq*

Tipos e condições de joins

Condição de junção – define os critérios para que os tuplos das relações se juntem.

Tipo de junção – define a forma de tratamento para os tuplos que não têm correspondência com base nas condições de junção

Join types
inner join
left outer join
right outer join
full outer join

Join conditions
natural
on <predicate>
using $(A_1, A_2, \ldots, A_n)$

Exemplos

course **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

course **full outer join** *prereq* **using** (*course_id*)

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Exemplos

course **inner join** *prereq* **on**
course.course_id = prereq.course_id

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190

Qual é a diferença entre o anterior e um natural join?

course **left outer join** *prereq* **on**
course.course_id = prereq.course_id

course_id	*title*	*dept_name*	*credits*	*prereq_id*	*course_id*
BIO-301	Genetics	Biology	4	BIO-101	BIO-301
CS-190	Game Design	Comp. Sci.	4	CS-101	CS-190
CS-315	Robotics	Comp. Sci.	3	*null*	*null*

Exemplos

course **natural right outer join** *prereq*

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-347	*null*	*null*	*null*	CS-101

course **full outer join** *prereq* **using** (*course_id*)

course_id	*title*	*dept_name*	*credits*	*prereq_id*
BIO-301	Genetics	Biology	4	BIO-101
CS-190	Game Design	Comp. Sci.	4	CS-101
CS-315	Robotics	Comp. Sci.	3	*null*
CS-347	*null*	*null*	*null*	CS-101

Views

Views

- Em alguns casos, não é desejável que todos os utilizadores vejam o modelo lógico completo
- Considere um utilizador que precisa de saber o nome e o departamento de um professor, mas não deve ter acesso ao seu salário. Essa pessoa deve apenas aceder ao resultado da query:

  select *ID*, *name*, *dept_name*
  from instructor

- Uma **view** fornece um mecanismo para ocultar certos dados aos utilizadores.
- Qualquer relação que não provém do modelo conceptual, mas que seja tornada visível como uma “relação virtual”, é chamada de **view** .

Definição de *views*

- Uma view é definida usando a instrução **create view** que tem o formato

  create view *v* **as** < query expression >

  onde <query expression> é qualquer expressão SQL válida. O nome da *view* é representado por *v.*

- Depois de criada, uma view pode ser usada para se referir à relação virtual que a view representa.

- A criação de uma view não é o mesmo que criar uma nova relação avaliando a <query expression>
  - Em vez disso, view guarda a query expression, que é substituída nas queries que usam a view.

Definição e uso de views

- Uma view dos professores sem seus salários

```
create view faculty as
    select ID, name, dept_name
    from instructor
```

- Encontre todos os professores do departamento de Biologia

```
select name
from faculty
where dept_name = 'Biology'
```

- Uma view com os totais de salários do departamento

```
create view departments_total_salary(dept_name, total_salary) as
    select dept_name, sum (salary)
    from instructor
    group by dept_name;
```

Views definidas usando outras views

- Uma view pode ser usada na expressão que define outra view
- Diz-se que uma view v_1 ***depende diretamente*** de uma view v_2 se v_2 for usada na expressão que define v_1
- Diz-se que uma view v_1 ***depende de*** uma view v_2 se v_1 depende diretamente de v_2 ou há um caminho de dependências de v_1 para v_2

Views definidas usando outras views

```
create view physics_fall_2017 as
  select course.course_id, sec_id, building, room_number
  from course, section
  where course.course_id = section.course_id
        and course.dept_name = 'Physics'
        and section.semester = 'Fall'
        and section.year = '2017';


create view physics_fall_2017_watson as
   select course_id, room_number
   from physics_fall_2017
   where building= 'Watson';
```

Expansão de views

- Expandir a view:

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from physics_fall_2017
    where building= 'Watson'
```

- Para:

```
create view physics_fall_2017_watson as
    select course_id, room_number
    from (select course.course_id, building, room_number
        from course, section
        where course.course_id = section.course_id
            and course.dept_name = 'Physics'
            and section.semester = 'Fall'
            and section.year = '2017')
    where building= 'Watson';
```

Expansão de views (cont.)

- Uma forma de definir o significado de views definidas a partir de outras views.
- Seja a view v_1 definida por uma expressão e_1 que pode usar outras views.
- A expansão da expressão de uma view repete as seguinte etapas de substituição:

repeat
  Encontre qualquer view v_i em e_1
  Substitua a view v_i pela expressão que define v_i
until não haja mais views em e_1

Materialized views

Certos SGBD permitem que as *views* sejam armazenadas fisicamente.

- Cópia física criada quando a *view* é definida.
- Essas views são chamadas de **materialized views (vistas materializadas)**

Se as relações usadas na view forem atualizadas, o resultado da vista materializada fica desatualizado

- É necessário **manter** a view, atualizando-a sempre que as relações subjacentes forem atualizadas.

Atualização de uma view

- Considere a *view faculty*

create view *faculty* as
select *ID*, *name*, *dept_name*
from *instructor*

- Adicione um novo tuplo à view *faculty*:

insert into faculty
values ('30765', 'Green', 'Music'**);**

- Esta inserção deve ser representada pela inserção na relação *instrutor*
  - Deve ter um valor para o salário.

Duas abordagens
  - Rejeitar a inserção
  - Insirir o tuplo ('30765', Green', Music', nulo) na relação *do instrutor*

Algumas atualizações não podem ser traduzidas exclusivamente

```
create view instructor_info as
    select ID, name, building
     from instructor, department
     where instructor.dept_name = department.dept_name;
```

```
insert into instructor_info
        values ('69987', 'White', 'Taylor');
```

Problemas

- Qual departamento, se houver vários departamentos no edifício Taylor?
- E se não houver nenhum departamento no edifício Taylor?

Atualizações de *views* em SQL

A maioria das implementações de SQL permitem apenas atualizações em *views* simples

- A cláusula **from** tem apenas uma relação.
- A cláusula **select** contém apenas nomes de atributos da relação e não possui nenhuma expressão, agregação ou **distinct** .
- Qualquer atributo não listado na cláusula **select** pode ser definido como nulo
- A query não tem uma cláusula **group** by ou **having** .

Transações

Transações

- Uma **transação** consiste numa sequência de instruções que é considerada uma "unidade" de trabalho
- A norma SQL especifica que uma transação começa implicitamente quando uma instrução SQL é executada.
- A transação deve terminar com uma das seguintes declarações:
  - **Commit**. As atualizações realizadas pela transação tornam-se permanentes na BD.
  - **Rollback** . Todas as atualizações realizadas pelas instruções SQL na transação são desfeitas.
- Transação atómica
  - As transações são totalmente executadas ou totalmente revertidas como se nunca tivessem ocorrido
- Isolamento de transações simultâneas

Transações

- Iremos falar sobre transações com detalhe (mais à frente)

Restrições de integridade

Restrições de integridade

- As Restrições de Integridade protegem a BD contra danos, garantindo que alterações autorizadas na BD não resultem na perda de consistência dos dados.
- Exemplos:
  - Uma conta corrente deve ter um saldo superior a €10.000,00
  - O salário de um funcionário bancário deve ser de pelo menos €10,00 por hora
  - Um cliente deve ter um número de telefone (não nulo)

Restrições numa única relação

- **not null**
- **primary key**
- **unique**
- **check** (P), onde P é um predicado

Restrições Not Null

Not Null

- Declarar *name* e budget como **não nulos**

> *name* **varchar**(20) **not null**
> *budget* **numeric**(12,2) **not null**

Tal como vimos nas aulas práticas, idealmente devemos dar nomes a todas as restrições implementadas na BD.

No exemplo anterior, considerando que os atributos pertencem à relação department poderíamos ter:

> *name* **varchar**(20) constraint NN_department_name **not null**
> *budget* **numeric**(12,2) constraint NN_department_budget **not null**

NOTA: atribuir nomes às restrições de integridade é uma boa prática que se deve aplicar em todas as restrições definidas numa BD

Restrições Unique

unique (A_1 , A_2 , …, A_m)

- A especificação unique indica que os atributos A_1 , A_2 , …, A_m são únicos, i.e., não se podem repetir

A cláusula check

- A cláusula de **check** (P) especifica um predicado P que deve ser satisfeito por cada tuplo numa relação.
- Exemplo: garantir que o semestre seja de 'Fall', 'Winter', 'Spring' ou 'Summer

```
create table section
        (course_id varchar (8),
         sec_id varchar (8),
         semester varchar (6),
         year numeric (4,0),
         building varchar (15),
         room_number varchar (7),
         time slot id varchar (4),
         primary key (course_id, sec_id, semester, year),
         check (semester in ('Fall', 'Winter', 'Spring', 'Summer')))
```

Integridade referencial

Garante que um valor que aparece numa relação para um determinado conjunto de atributos também apareça para um determinado conjunto de atributos noutra relação.

- Exemplo: Se “Biologia” for um nome de departamento que aparece num dos tuplos na relação *instrutor* , então existe um tuplo na relação *department* para “Biologia”.

Seja A um conjunto de atributos e R e S duas relações que contêm atributos A, onde A é a chave primária de S. A é dito ser uma **chave estrangeira** de R se para quaisquer valores de A que aparecem em R esses valores também aparecem em S.

Integridade referencial (cont.)

- *Chaves* estrangeiras podem ser especificadas como parte da instrução SQL de **create table**

  foreign key (*dept_name*) **references** *department*

- Por omissão, uma chave estrangeira faz referência aos atributos de chave primária da tabela referenciada.

- O SQL permite definir os atributos explicitamente:

  chave estrangeira (*dept_name*) **faz referência** *ao departamento* (*dept_name*)

Ações em cascata na integridade referencial

- Quando uma restrição de integridade referencial é violada, o procedimento normal é rejeitar a ação que a causou.
- Uma alternativa, em caso de exclusão ou atualização, é operar em cascata

```
create table course (
        (…
         dept_name varchar(20),
         foreign key (dept_name) references department
             on delete cascade
             on update cascade,
          . . .)
```

Em vez de cascade, podemos usar:

- **set null**
- **set default**

Violação de Restrição de Integridade Durante Transações

- Considerar:

  create table *person* (

  ID **char**(10),
  name **char**(40),
  mother **char**(10),
  father **char**(10),
  primary key *ID,*
  foreign key *father* **references** *person,*
  foreign key *mother* **references** *person*)

  - Como inserir um tuplo sem violar a restrição?
    - Insirir o pai e a mãe de uma pessoa antes de inserir a pessoa
    - OU, definir o pai e mãe como nulos inicialmente, atualizar após inserir todas as pessoas (não é possível se os atributos father e mother forem declarados como **not null**)
    - OU adiar a verificação da restrição (**defer)**

Tipos de dados e domínios

Tipos de objetos (grandes)

- Objetos grandes (fotos, vídeos, etc.) são armazenados como *large objects*:
  - **blob** : binary large object -- coleção de dados binários não interpretados (cuja interpretação é deixada para uma aplicação fora do SGBD)
  - **clob** : character large object -- coleção de dados de caráteres
- Quando uma query retorna um objeto grande, um ponteiro é retornado em vez do objeto em si.

Tipos definidos pelo utilizador

- **create type** em SQL permite criar tipos definidos pelo utilizador

  create type *Dollars* **as numeric (12,2) final**

  Exemplo:

create table *department*
(*dept_name* **varchar** (20),
building **varchar** (15),
budget Dollars);

Domínios

- **create domain** em SQL-92 cria domínio definido pelo utilizador

  create domain *person_name* **char**(20) **not null**

- Tipos e domínios são semelhantes. Domínios podem ter restrições de integridade associadas na sua especificação
- Exemplo:

  create domain *degree_level* **varchar**(10)
  constraint *degree_level_test*
  check (**value in** ('Bachelors', 'Masters', 'Doctorate'));

Índices

Criação de Índices

- Muitas queries fazem referência apenas a uma pequena proporção dos registos numa tabela.
- É ineficiente para o sistema ler toda a tabela para encontrar um registo com valores específicos
- Um **índice** num atributo é uma estrutura de dados que permite que o SGBD encontre tuplos que têm um valor especificado para esse atributo de forma eficiente, sem precisar ler todos os tuplos.
- Criamos um índice com o comando **create index**

  create index <name> **on** <relation-name> (attribute);

Exemplo de criação de índice

- **create table** *student*
  (*ID* **varchar** (5),
  name **varchar** (20) **not null**,
  dept_name **varchar** (20),
  tot_cred **numeric** (3,0) **default** 0,
  primary key (*ID*))

- **create index studentID_index on student(ID)**

- A query:

```
select *
from  student
where  ID = '12345'
```

pode ser executada usando o índice, sem necessidade de ler todos os registos

Índices

- Falaremos de índices com mais detalhe (mais à frente)

Autorização

Autorização

- Podemos atribuir a um utilizador diversas formas de autorização em partes da BD
  - **Read** - permite a leitura, mas não a modificação de dados.
  - **Insert** - permite a inserção de novos dados, mas não a modificação de dados existentes.
  - **Update** - permite modificação, mas não a eliminação de dados.
  - **Delete** - permite a eliminação de dados.
- Cada um destes tipos de autorização é chamado de **privilégio**. Podemos autorizar o utilizador com todos, nenhum ou uma combinação destes tipos de privilégios em partes específicas de uma BD, como uma tabela ou uma *view.*

Autorização (Cont.)

Formas de autorização para modificar o esquema da BD

- **Index** - permite a criação e eliminação de índices.
- **Resources** - permite a criação de novas tabelas.
- **Alteration** - permite adicionar ou excluir atributos de uma tabela.
- **Drop** - permite eliminar tabelas.

- NOTA: Existem muitos mais privilégios que podem ser atribuídos ou removidos a cada utilizador.

Especificação de autorização em SQL

A declaração **grant** é usada para conferir autorização

grant <privilege list> **on** <relation or view > **to** <user list>

< user list > é:

- um utilizador da BD
- **public**, que concede o privilégio a todos os utilizadores válidos
- Um papel (role)

Exemplo:

- **grant select on** *department* **to** Amit, Satoshi

- Conceder um privilégio numa view não implica conceder privilégios nas relações subjacentes.

Privilégios em SQL

- **select** : permite acesso de leitura à tabela ou à view
  - Exemplo: conceder aos utilizadores U_1, U_2 e U_3 autorização de select na tabela *instrutor* :

    grant select on *instructor* **to** U_1, U_2, U_3

- **insert** : a capacidade de inserir registos
- **update** : a capacidade de atualizar usando a instrução update em SQL
- **delete** : a capacidade de apagar registos.
- **All privileges** : usado como uma forma abreviada para todos os privilégios permitidos

Revogar autorização em SQL

- A declaração **revoke** é usada para revogar autorização.

  revoke <privilege list> **on** <relation or view> **from** <user list>

- Exemplo:

  revoke select on *student* **from** U_1, U_2, U_3

- <privilege-list> pode ser **usado** para revogar todos os privilégios que o utilizador possa ter.

- Se a lista de utilizadores incluir **public,** todos os utilizadores perdem o privilégio, exceto aqueles que o receberam explicitamente.

Roles

- Um **role** é uma forma de agrupar vários privilégios que, posteriormente, podem ser atribuídos a diferentes perfis de utilizador.
- Para criar um role usamos:

  create a role <name>

- Exemplo:
  - **create role** instructor
- Depois de um role estar criado, podemos atribuir esse role aos utilizadores:
  - **grant** <role> **to** <users>

Exemplo de roles

- **create role** instructor;
- **grant** *instructor* **to** Amit**;**
- Privilégios podem ser concedidos aos roles:
  - **grant select on** *takes* **to** *instructor*;

Os roles podem ser concedidos aos utilizadores, bem como a outros roles

- **create role** *teaching_assistant*
- **grant** *teaching_assistant* **to** *instructor*;
  - *O instrutor* herda todos os privilégios do *teaching_assistant*

Cadeia de roles

- **create role** *dean*;
- **grant** *instructor* **to** *dean*;
- **grant** *dean* **to** Satoshi;

Funções e procedimentos

Funções e Procedimentos

- Funções e procedimentos permitem que a “lógica de negócio” seja armazenada na BD e executada a partir de instruções SQL.
- A sintaxe apresentada nos slides é a definida pelo padrão SQL.
  - A maioria dos SGBD implementa versões **não padronizadas** desta sintaxe.
  - É necessário ajustar a sintaxe ao SGBD em utilização.

Funções em SQL

- Defina uma função que, dado o nome de um departamento, retorne a contagem do número de professores desse departamento.

```
create function dept_count (dept_name varchar(20))
        returns integer
        begin
        declare d_count  integer;
                select count (* ) into d_count
                from instructor
                where instructor.dept_name = dept_name
        return d_count;
end
```

- Após a definição da função, ela pode ser usada no contexto de queries SQL. Por exemplo:
  - Lista os nomes dos departamentos e o orçamento de todos os departamentos com mais de 12 professores .

```
select dept_name, budget
from department
where dept_count (dept_name ) > 12
```

Funções de tabela

- O padrão SQL permite que uma função possa retornar tabelas como resultado
- Exemplo: Retornar todos os professores de um determinado departamento

```
create function instructor_of (dept_name char(20))
        returns table  (
             ID varchar(5),
             name varchar(20),
            dept_name varchar(20),
            salary numeric(8,2))
         return table
               (select ID, name, dept_name, salary
                from instructor
                where instructor.dept_name = instructor_of.dept_name)
```

- Uso

```
select *
from table (instructor_of ('Music'))
```

Procedimentos em SQL

- A função *dept_count* poderia ser escrita como procedimento (stored procedure):

```
create procedure dept_count_proc (in dept_name varchar(20), out d_count integer)
    begin
        select count(*) into d_count
        from instructor
        where instructor.dept_name = dept_count_proc.dept_name
    end
```

- As palavras-chave **in** e **out** são parâmetros que se espera que tenham valores atribuídos (se forem **in**, na chamada do procedimento; se forem **out**, após a chamada do procedimento; **inout** em ambos)
- Os procedimentos podem ser invocados a partir de um procedimento SQL ou de SQL embebido, usando a instrução **call.**

```
declare d_count integer;
call dept_count_proc( 'Physics', d_count);
```

Construções de linguagem para procedimentos e funções

- SQL suporta vários componentes que fornecem uma capacidade semelhante à de uma linguagem de programação de uso geral.
  - Aviso: a maioria dos SGBD implementa sua própria variante da sintaxe padrão.

- Composição de instruções (bloco): **begin** … **end**
  - Pode conter múltiplas instruções SQL entre **begin** e **end**
  - Variáveis locais podem ser declaradas

- Ciclos:
  - **while** boolean expression **do**
    sequence of statements ;
    end while

  - **repeat**
    sequence of statements ;
    until boolean expression
    end repeat

Construções de Linguagem (Cont.)

- **For** loop
  - Permite iteração sobre todos os resultados de uma consulta

- Exemplo: Encontre o orçamento de todos os departamentos

```
declare n integer default 0;
    for r as
      select budget from department
      where dept_name = 'Music'
    do
       set n = n + r.budget
    end for
```

Construções de linguagem – if-then-else

- Declarações condicionais (**if-then-else**)

if *boolean expression*
  then *statement or compound statement*
elseif *boolean expression*
  then *statement or compound statement*
else *statement or compound statement*
end if

Triggers

Triggers

- Um **trigger** é uma instrução executada automaticamente pelo sistema como efeito colateral de uma modificação na BD.
- Para projetar um trigger, devemos:
  - Especificar as condições sob as quais o trigger será executado.
  - Especificar as ações a serem tomadas quando o trigger for executado.
- Os triggers foram introduzidos no padrão SQL:1999, mas eram suportados usando sintaxe não padrão pela maioria dos SGBD.

Acionando eventos e ações em SQL

- O evento desencadeador pode ser **insert** , **delete** ou **update**
- Os triggers no update podem ser restritos a atributos específicos
  - Por exemplo, **after update of** *takes* **on** *grade*
- Valores de atributos antes e depois de uma atualização podem ser referenciados
  - **referencing old row as :** para deletes e updates
  - **referencing new row as :** para inserts updates
- Os triggers podem ser ativados antes de um evento. Por exemplo, converta notas em branco (string vazia) em nulas.

```
create trigger setnull_trigger before update of takes
referencing new row as nrow
for each row
    when (nrow.grade = ' ')
        begin atomic
            set nrow.grade = null;
end;
```

Trigger para manter o valor credits_earned

```
create trigger credits_earned after update of takes on (grade)
referencing new row as nrow
referencing old row as orow
for each row
when nrow.grade <> 'F' and nrow.grade is not null
   and (orow.grade = 'F' or orow.grade is null)
begin atomic
    update student
    set tot_cred= tot_cred +
        (select credits
         from course
         where course.course_id= nrow.course_id)
    where student.id = nrow.id;
end;
```

Triggers ao nível da instrução

- Em vez de executar uma ação separada para cada linha afetada, uma única ação pode ser executada para todas as linhas afetadas por uma transação
  - Use **for each statement** em vez de **for each row**
  - Use **referencing old table** ou **referencing new table** para fazer referência a tabelas temporárias (chamadas ***tabelas de transição***) que contêm as linhas afetadas
  - Pode ser mais eficiente quando lidamos com instruções SQL que atualizam um grande número de linhas

Quando não usar triggers

Os triggers foram usados anteriormente para tarefas como

- Manter dados resumidos (por exemplo, salário total de cada departamento)
- Replicar a BD, copiando os registos para outra BD

Existem formas mais adequadas de realizar estas tarefas:

- Os SGBD atuais fornecem recursos integrados como vistas materializadas para manter dados resumidos
- Os SGBD atuais fornecem mecanismos avançados de replicação

Em muitos casos, devemos usar encaplusalmento, em vez de triggers

- Definir métodos para atualizar campos
- Executar ações como parte dos métodos de atualização em vez de um trigger

Quando não usar triggers (cont.)

Risco de execução não intencional de triggers, por exemplo, quando

- Carregamos dados de uma cópia de backup
- Replicamos atualizações de uma BD remota
- A execução do trigger pode ser desativada antes destas ações, mas o risco existe

Genericamente, os trigers podem ser usados para

- Manter a integridade dos dados (apenas quando não é possível através de mecanismos mais simples, como unique, not null, foreign key ou check)
- Garantir regras de negócio (quando não é possível usar encapsulamento)
- Automatizar tarefas (quando não é possível usar encapsulamento ou mecanismos específicos para a tarefa)