

Databases

Advanced SQL

Danilo Montesi

danilo.montesi@unibo.it



Other Constraints: Check

- Check specifies constraints over tuples (and possibly, even more complex ones that aren't always supported in all SQL implementations)

CHECK (*Predicate*)



Check: Example (1)

```
create table EMPLOYEE (  
    Number          integer primary key,  
    Surname         character(20),  
    Name            character(20),  
    Gender          character not null  
        CHECK (Gender in ('M', 'F')),  
    Salary          integer  
        CHECK (Salary >= 0),  
    Supervisor      integer,  
        CHECK (Salary <= (  
                                select Salary  
                                from EMPLOYEE J  
                                where Supervisor =  
                                J.Number))  
)
```

- A **select** clause within the **CHECK** constraint is not fully supported in every SQL implementation



Check: Example (2)

```
create table EMPLOYEE (  
    Number          integer primary key,  
    Surname         character(20),  
    Name            character(20),  
    Gender          character not null  
        CHECK (Gender in ('M', 'F')),  
    Salary          integer,  
    Withholding     integer,  
    Net             integer,  
    Supervisor     character(6),  
        CHECK (Net = Salary - Withholding)  
)
```

■ Ok



Check: Example (3)

```
insert into EMPLOYEE values  
  (1 , 'Doe' , 'John' , ' ' , 100 , 20 , 8  
    0);
```

Sex is neither 'M'
nor 'F'

```
insert into EMPLOYEE values  
  (2 , 'Lee' , 'Jim' , 'M' , 100 , 10 , 8  
    0);
```

Net (80) is not
100-10

```
insert into EMPLOYEE values  
  (3 , 'Hill' , 'Sam' , 'M' , 70 , 20 , 50  
    );
```

This update is
allowed



Other Constraints: Assertion

- Defines constraints at the schema level

```
create assertion NameAs check  
    (Predicate)
```

```
create assertion AtLeastOneEmployee  
    check (1 <= (select count(*)  
                    from EMPLOYEE))
```

- A **select** clause within the **check** constraint is not fully supported in every SQL implementation

View

```
create view ViewName [ ( AttList ) ]  
  as SelectStatement  
[ with [ local | cascaded ] check  
  option ]
```

```
create view ADMINEMPLOYEES  
  (Name, Surname, Salary) as  
  select Name, Surname, Salary  
  from EMPLOYEE  
  where Dept = 'Administration' and  
         Salary > 10
```

View Update

- Usually, such updates are allowed for views defined on a single relation
- We can force the DB to perform some checks



View Update: Check Option

```
create view P00RADMINEMPLOYEES as  
  select *  
  from ADMINEMPLOYEES  
  where Salary < 50  
with check option
```

- **check option** allows to update the view, but only if the inserted tuple belongs to the view (the user cannot have a salary greater than 50)



Not Allowed Operation: an Example

```
create view P00RADMINEMPLOYEES as  
  select *  
  from ADMINEMPLOYEES  
  where Salary < 50  
with check option
```

```
update P00RADMINEMPLOYEES  
  set Salary = 60  
  where Name = 'Ann'
```



Altering a View: Local and Cascaded

- **local** (in the case of views over views): the tuple update has to be performed only at the last level of the view
- **cascaded** (in the case of views over views): the tuple update has to be performed over all the underlying views and relations

Querying a View

- Views could be queried like any other relation within the database, for instance:

```
select * from ADMINEMPLOYEES
```

is like performing the following query (and it is run as):

```
select Name, Surname, Salary  
from EMPLOYEE  
where Dept = 'Administration' and  
       Salary > 10
```



A Wrong SQL Query (1)

- Provide the average number of offices per department

- Wrong query:

```
select avg(count(distinct  
Office))  
from EMPLOYEE  
group by Dept
```

- The query is wrong because the SQL syntax does not allow to nest aggregating operators



A Correct SQL Query (1)

- Provide the average number of offices per department
- Using an intermediate view:

create view

```
DEPTOFFICES(NameDept,OffNum) as  
  select Dept, count(distinct Office)  
  from EMPLOYEE  
  group by Dept;
```

```
select avg(OffNum)  
from DEPTOFFICES
```



A Wrong SQL Query (2)

- Provide the departments having the greatest employee's salary sum
- A solution wrong for some systems

```
select Dept
from EMPLOYEE
group by Dept
having sum(Salary)>=all(select
    sum(Salary)
                        from EMPLOYEE
                        group by Dept)
```

- A nested **having** is not allowed in some SQL implementations



A Correct SQL Query (2)

- Provide the departments having the greatest employee's salary sum

```
create view  
  BUDGETSALARY(Dept,TotalSalary) as  
  select Dept, sum(Salary)  
  from EMPLOYEE  
  group by Dept
```

```
select Dept  
from BUDGETSALARY  
where TotalSalary = (select  
  max(TotalSalary)
```

```
  from BUDGETSALARY)
```


Recursive Queries (1)

- For each person provide his/her ancestors, having:

FATHERHOOD(Father, Child)

FATHERHOOD	
Father	Child
Carl	Frank
Louis	Olga
Louis	Bob
Frank	Alex
Frank	Alfred

Recursive Queries (2)

■ We have to use recursion; in Datalog:

ANCESTORS(Ancessor: f , Descendant: c) \leftarrow
FATHERHOOD(Father: f , Child: c)

ANCESTORS(Ancessor: a , Descendant: d) \leftarrow
FATHERHOOD(Father: a , Child: c),
ANCESTORS(Ancessor: c , Descendant: d)



Recursive Queries in SQL:1999

with recursive

```
ANCESTORS(Ancestor,Descendant) as (  
    select Father, Son  
    from FATHERHOOD  
    union all  
    select Ancestor, Son  
    from ANCESTORS, FATHERHOOD  
    where Descendant = Father  
)
```

```
select *  
from ANCESTORS
```

- **with** defines the **ANCESTORS** view, which is built up recursively using **FATHERHOOD**

Example

- Return all John Doe's supervisors

```
with recursive INCHARGE(Num, Supervisor)
as (
    select Num, Supervisor
    from EMPLOYEE
    union
    select Employee.Num,
           INCHARGE.Supervisor
    from EMPLOYEE, INCHARGE
    where EMPLOYEE.Supervisor =
INCHARGE.Num
)
select Name, Surname, INCHARGE.Supervisor
from EMPLOYEE join INCHARGE
on (EMPLOYEE.Num = INCHARGE.Num)
where Name = 'John' and Surname = 'Doe'
```



Scalar Functions (1)

- Functions at tuple level providing one single value per tuple
- Temporal
 - **current_date()** returns the present date
 - **extract(*yearExpression*)** extracts part of a date from a given expression (e.g., month, day, hour, etc.)
 - Return the order year from the orders that were issued today

```
select extract(year from OrderDate) as  
    OrderYear  
from ORDERS  
where date(OrderDate)=current_date()
```



Scalar Functions (2)

- String editing
 - **char_length** returns the string's length
 - **lower** converts the string to lower case
- Casting
 - **cast** allows the casting of a value into another domain
- Conditional
 - ... *stay tuned*



Conditional Expression: COALESCE (1)

coalesce takes several expressions as an input and returns the first of them which is not **NULL**

■ Given the following relation

EMPLOYEE(Number, Dept, Mobile,
PhoneHome)

For each employee return either a valid Mobile phone number, or its phone number

```
select number, coalesce(Mobile,  
PhoneHome)  
from EMPLOYEE
```



Conditional Expression: COALESCE (2)

coalesce could be used to provide a default value for replacing null values. In the following example, a NULL department is replaced with “None”

```
select Name, Surname,  
        coalesce(Dept, 'None')  
        from EMPLOYEE
```




Conditional Expression: NULLIF

nullif compares the first argument (e.g. an Attribute of a given relation) with the second one (e.g. a constant value). It returns NULL if the two arguments match, otherwise it returns the value from the first argument

- Example: Extract the surnames and departments to which the employees belong, returning a NULL value for the department when the Dept attribute has the value 'Unknown'

```
select Surname,  
nullif(Dept, 'Unknown')  
from EMPLOYEE
```



Conditional Expression: CASE (1)

The **case** function allows to specify conditional structures, whose results depends on the evaluation of the tables' content

It is used to provide an “if-then-else” kind of logic to the SQL language



Conditional Expression: CASE (2)

- Evaluate the vehicle taxes by its type and registration year (since 1975)

VEHICLE(PlateNum, Type, Year, KWatt, Length)

```
select PlateNum,  
        (case Type  
          when 'Car' then 2.58 * KWatt  
          when 'Moto' then (22.00 + 1.00 *  
KWatt)  
          else null  
        end) as Tax  
from VEHICLE  
where Year > 1975
```

Database Security

- SQL allows to grant for each user specific **privileges** (reading, writing, ...) for either the whole DB or part of it
- **Privileges** could be **granted** to either relations, attributes, views or domains
- There is at least one default admin (e.g., **_system**) for which all the privileges are granted
- Any user creating a specific “resource” automatically grants every privilege on it

Privileges

A privilege is described by:

- a specific ***resource***
- the user ***granting*** the privilege
- the user ***to which*** the privilege is granted
- a specific ***operation***
- whether the user can ***propagate*** the privilege or not

SQL Privilege Types

- **insert**: allows to insert new tuples
- **update**: allows to edit pre-existing tuples
- **delete**: allows to delete tuples
- **select**: allows to read a resource
- **references**: allows to define referential integrity constraints
- **usage**: allows to use a definition (e.g., a custom data type)



Granting Privileges

- Granting privileges:

```
grant < Privileges | all privileges  
>  
    on Resource  
    to Users    [ with grant option ]
```

- **grant option** allows the user to propagate his/her privilege to other users

```
grant select on DEPARTMENT to Jack
```



Revoking Privileges

■ Revoking privileges

***revoke Privileges on Resource
from Users [restrict | cascade]***

■ The privileges have to be revoked by the same user that granted them in the first place

■ **restrict** (default) the revoke does not involve other users that received the grant from the current user

■ **cascade** the revoke is extended to the other users

Be careful: **cascade** causes a “chain reaction”

Privileges Discussion (1)

- The SQL implementation has to hide (without giving a clue) the part of the database that are not accessible to the user. For instance, either:
 - Table **EMPLOYEE** doesn't exist, or
 - Table **EMPLOYEE** does exist, but the user cannot access it
- ⇒ the user has to receive the same message from the system



Privileges Discussion (2)

- We could use a view for showing only specific tuples to a user:
 - The view is defined with a selection predicate
 - The privilege is granted for the specific view

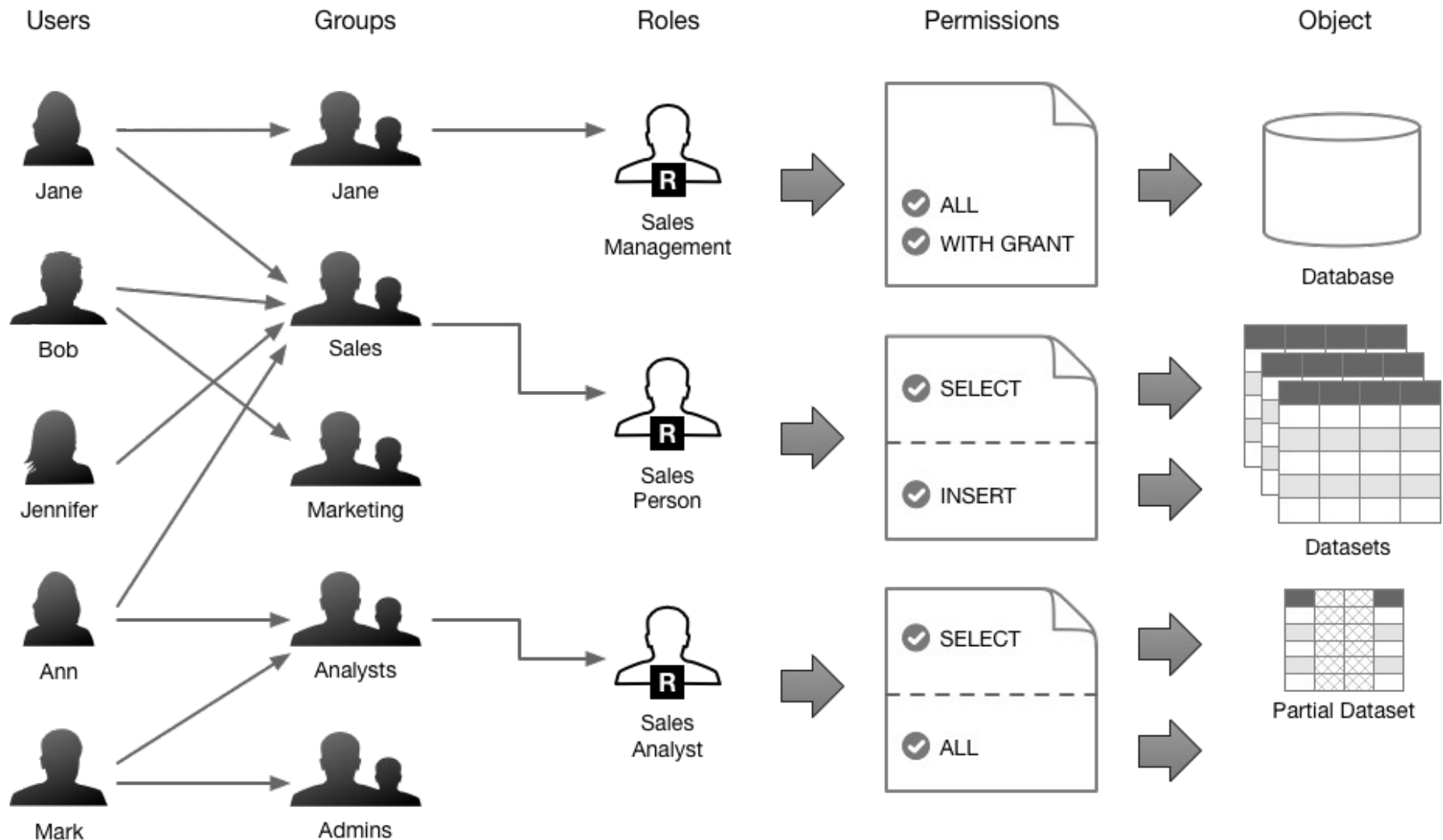


Authorizations: RBAC (1)

SQL-3 has introduced RBAC for the first time
(Role-Based Access Control, **RBAC**)

- Each **role** acts as a container of several privileges that can be granted via the **grant** command
- At any moment, each user has
 - “individual” granted privileges, associated directly to him
 - privileges granted to its role through RBAC

Authorizations: RBAC (2)





Authorizations: RBAC (3)

- The following command creates a new *Name* role

create role *Name*

- This command grants the role *Name* to the current user

set role *Name*

RBAC: an Example

■ Grant **create table** to a specific user through the role **Employee**:

- Create the new role:

create role Employee;

- Grant the privilege to the previously defined role:

grant create table to Employee;

- Grant the privilege to a specific user:

grant Employee to user;

- Revoke the previously granted privilege:

revoke create table from Employee;

Transactions

- A transaction is an executing program that forms a logical unit of database processing (atomic operation)
- Properties:
 - **Atomicity**
 - **Consistency preservation**
 - **Isolation**
 - **Durability (permanency)**

Transactions are ... Atomic

- A transaction is an atomic unit of processing: it should either be performed or not performed at all
- *Money transfer from a bank account A to B: either the money is withdrawn from A and transferred to B, or no operation at all is performed*



Transactions are ... Consistent

- A transaction should be consistency preserving: if it is completely executed from beginning to end with no interference, the database moves between two consistent states
- During the execution of a transaction some violations may occur, but they cannot stay when the transaction ends: if in the end of a transaction there are still violations, the transaction must be cancelled (abort) completely

Transactions are ... Isolated

- Even if many transactions are executed concurrently, the execution of the current transaction should not be interfered with others that are currently running
- *For example, the case of two money transfers over the same bank account happening at the “same” time*



Transactions have Durable Effects

- The changes applied to the database by a correct (**committed**) transaction must persist and must not be lost, even when serious faults (hardware or software) occur and in parallel execution



Transaction Support in SQL

- Once the connection to the database is established, the first transaction starts with **start transaction**, either before the first operation, or after the last one (the command is optional in most RDBMSs)
- The transaction could terminate with
 - **commit [work]**: the operations are saved within the database
 - **rollback [work]**: the operations are discarded and the DB returns to a previous safe state
- Most RDBMSs have **autocommit**, where each statement is a different transaction

Transaction Support in SQL: an Example

- Withdraw 10€ from the account 42177 and transfer it to the account 12202

start transaction

update BANKACCOUNT

set Balance = Balance – 10

where AccountNumber = 42177;

update BANKACCOUNT

set Balance = Balance + 10

where AccountNumber = 12202;

commit work;