Introduction to Relational Databases

- Bachelor Computer Science, Lille 1 University
- Nov 14th, 2012 (lecture 10/12)
- Today's lecturer: C. Kuttler
- Topic: Introduction to SQL
  - Subqueries:
    - Comparison of operators
    - Aggregate functions for generalized AND/OR in having clause
    - Variable visibility
  - Other definitions of data in SQL
    - Views
    - Generic integrity constraints
    - Access control

Equivalence of expressive power

- IN, =ANY, EXISTS have the same expressive power, and can also be expressed through a join
- NOT IN, < >ALL, NOT EXISTS have the same expressive power, and can be expressed by a difference
- *comp* SOME, if there are no duplicates, can be rewritten as theta-joins (not as equi-joins)
- *comp* ALL can be rewritten by queries combining grouping and extraction of a minimum and maximum

Having clause: generalized AND

- EVERY(test): true, if test is true for all lines of group
- Supplier whose articles all cost at least 10 euro

```
select c.fid
from catalogue c
where
not exists(select *
          from catalogue c2
          where c2.fid=c.fid and c.prix<=10)
```

Having clause: generalized AND

- EVERY(test): true, if test is true for all lines of group
  - True for line 1 AND true for line 2 AND … AND true for line n
  - Other syntax: BOOL_AND(test)
- Supplier whose articles all cost at least 10 euro

```
select fid
from catalogue
group by fid
having every(prix >10)
```

Having clause: generalized OR

- bool_or(test): true, if test is true for at least one line of group
- Supplier offering both green and red articles

```
select c.fid
from catalogue c
where
  exists (select * from articles a where
  a.aid=c.aid and a.acoul=‘green’)
and
exists (select * from articles a where
  a.aid=c.aid and a.acoul=‘red’)
```

Having clause: generalized OR

- bool_or(test): true, if test is true for at least one line of group
- Supplier offering both green and red articles

```
select c.fid
from catalogue c, articles a
where a.aid=c.aid
group by c.fid
  having bool_or(acoul=‘vert’)
          and
  bool_or(acoul=‘red’)
```

Tuple construction

- The comparison with the embedded query can involve more than one attribute.
- The attributes must be enclosed by a pair of parentheses (tuple constructor)
- Our previous query can be rewritten as:

```
select *
from Person P
where (Name,LastName) in
        (select Name, LastName
         from Person P1
         where P1.NumSecu <> P.NumSecu)
```

Comments on subqueries

- Embedded queries can be ‘less declarative’, but are mostly easier to read
- Complex queries with variables can be hard to understand.
- The embedded queries can not contain set operations, mostly (take home lesson: “only do unions on top level”). This limitation is not significant, and not present in all DBMS.

Comments on subqueries

- The use of variables must respect rules of visibility
  - a variable can only be used in the query where it is introduced, or within subqueries embedded therein
  - If a variable name is ambiguous, the system assumes we are referring to the closer one

Visibility of variables

- Incorrect query:

```
select *
from Customer
where Cus_ID in
           (select Cus_ID
            from Contract O1
            where Con_ID = 'AZ1020')
   or Cus_ID in
           (select Cus_ID
            from Contract O2
            where O2.Date = O1.Date)
```

- The query is incorrect, because the variable `O1` is not visible within the second embedded query.

Subqueries in modification commands

Modifation commands with `in`

- Increase by 5 euro the VALUE of all contracts that contain the product 456

```
update Contract
  set VALUE = VALUE + 5
  where Con_ID in
     (select Con_ID
      from Detail
      where Prod_ID = '456')
```

Embedded queries in modifications

- Setting TotalPieces to the sum of quantities of all lines of a contract.

```
update Contract O
  set TotalPieces =
    (select sum(Qt)
     from Detail D
     where D.Con_ID = O.Con_ID)
```

Next topics

- Views
- Generic constraints
- Access control

Views

- Offer the "view" of virtual tables (external schemas)
- Classified into:
  - simple (selection and projection from only one table)
  - complex
- Syntax:

  create view *ViewName* [(*AttributeList*)]
  as *Subquery*
  [**with** [**local** | **cascaded**] **check option**]

Views

- Their definition may contain other views, that were previously defined, but without mutual dependency (recursion was introduced in SQL:1999)
- Can be used to write complex queries
  - Query decomposition
- Are sometimes needed to express certain queries
  - Namely such queries that combine and embed several aggregate operations

Composition of views and queries

- View creation:

```
create view MainContracts as
    select *
    from Contract
    where VALUE > 10000
```

- Query:

```
select Cus_ID
from MainContracts
```

- Composition of both:

```
select Cus_ID
from MainContracts
where VALUE > 100000
```

Views and queries

- Extract the customer with the highest total bill (without view):

```
select Cus_ID
from Contract
group by Cus_ID
having sum(VALUE) >= all
          (select sum(VALUE)
           from Contract
           group by Cus_ID)
```

- Works with Postgresql, but not accepted by all SQL systems.

Views and queries

- Extract the customer with the highest bill (via view):

```
create view CustomerBill(Cus_ID,TotalBill)
as
   select Cus_ID, sum(VALUE)
   from Contract
   group by Cus_ID;

select Cus_ID
from CustomerBill
where TotalBill = (select max(TotalBill)
                   from CustomerBill);
```

Views and queries

- Extract the average number of contracts per customer:
  - Incorrect query (aggregate functions can not be nested):

```
select avg(count(*))
from Contract
group by Cus_ID
```

  - Correct query (with a view):

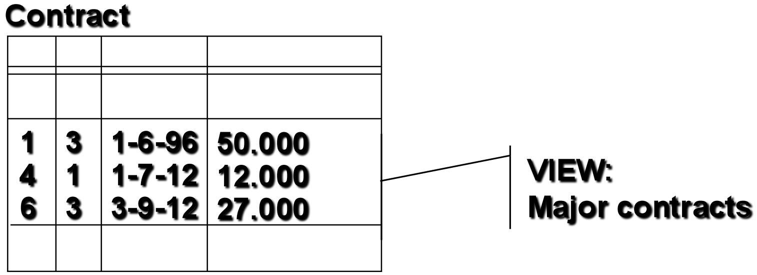
```
create view CustomerStat=(Cus_ID,ConNumber) as
select Cus_ID, count(*)
from Contract
group by Cus_ID;

select avg(ConNumber)
from CustomerStat;
```

Example of simple view

- Contracts with VALUE over 10.000

```
create view MajorContracts as
       select *
       from Contract
       where VALUE > 10000
```



Contract

1	3	1-6-96	50.000
4	1	1-7-12	12.000
6	3	3-9-12	27.000

VIEW:
Major contracts

Simple views in a cascade

```
create view Administrators
   (Sid,Name,LastName,Income) as
select Sid, Name, LastName, Income
from Employee
where Department = 'Administration'


create view JuniorAdministrators as
select *
from Administrators
where Income < 50
with check option
```

Modifications through views

- View:

```
create view MajorContracts as
  select *
  from Contract
  where VALUE > 10000
```

- Modification:

```
update MajorContracts
  set VALUE = VALUE * 1.05
  where Cus_ID = '45'
```

- Composition of both:

```
update Contract
  set VALUE = VALUE * 1.05
  where Cus_ID = '45'
    and VALUE > 10000
```

Check option: updating views

- The **`check option`** acts when the content of a view is modified.
  - Pre-condition: inserted/ updated tuple must be part of the view.
  - Post-condition: the tuple must remain in the view

- **`local:`**control only with respect to the view that is invoking the command.
- **`cascaded:`** the control is made in all involved views, recursively.

Check option: example

-
  ```
  create view MajorContracts70 as
  select *
  from MajorContracts
  where Cus_ID = '70'
  with local check option
  ```
- **Dependencies:**
  - `MajorContracts: Contracts with VALUE>10000`
  - `MajorContracts70: MajorContracts with Cus_ID=70`

Check option

-
  ```
  update MajorContracts70
  set Cus_ID = '71'
  where Con_ID = '754'
  ```

  `is refused with` check option **`local`** and **`cascaded`**

-
  ```
  update MajorContracts70
  set VALUE = 5000
  where Con_ID = '754'
  ```

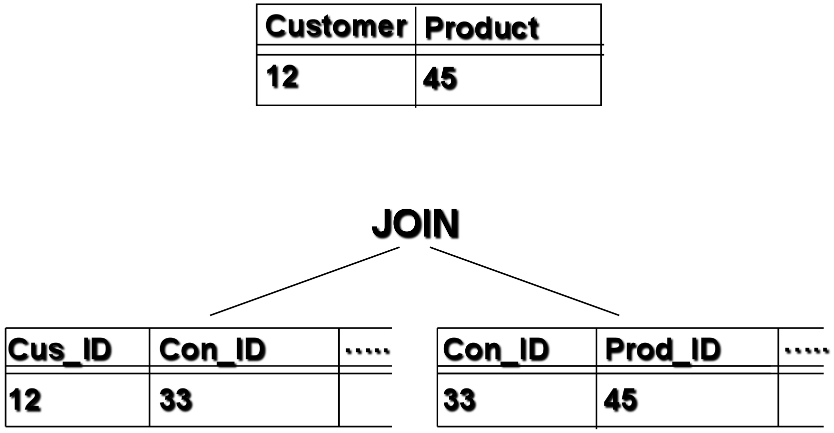
  is accepted with **`local`**, but refused with **`cascaded`**

Complex view

What else is possible, beyond selection and projection?

```
create view CusPro(Customer,Product) as
   select Cus_ID, Prod_ID
   from Contract join Detail
     on Contract.Con_ID = Detail.Con_ID
```

Complex view (JOIN)



Customer	Product
12	45

JOIN

Cus_ID	Con_ID	.....
12	33	

Con_ID	Prod_ID	.....
33	45	

Query on complex view

- Query:

```
select Customer
from CusPro
where Product = ’45’
```

- Combining both:

```
select Cus_ID
from Contract join Detail
  on Contract.Con_ID = Detail.Con_ID
where Prod_ID = ’45’
```

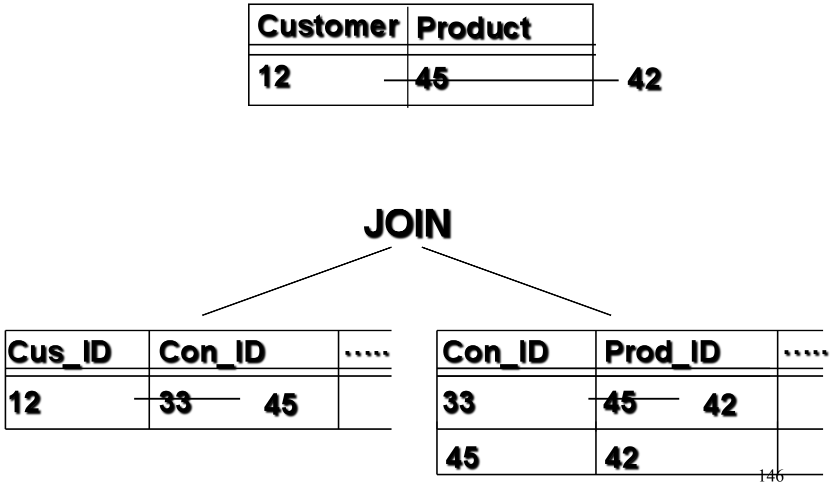
Modifications of the complex view

- It is impossible to modify the original table through the view, because the interpretation is ambiguous:
- Ex.:

```
update CusPro
     set Product = ’42’
     where Customer = ’12’
```

- Ambiguity for the modification of the original tables
  - The customer has changed his contract
  - The product’s identifier has changed

Complex view (JOIN)



Customer	Product
12	~~45~~ 42

JOIN

Cus_ID	Con_ID	.....
12	~~33~~ 45	

Con_ID	Prod_ID	.....
33	~~45~~ 42	
45	42	

Recursion in SQL:1999

```
with recursive Raggiungibile (Orig,Dest,Costo) as
( select Orig, Dest, Costo
  from Volo where Orig = ‘Milano’
  union
  select V.Orig, R.Dest, V.Costo+R.Costo
  from Volo V join Raggiungibile R
       on V.Dest = R.Orig
  where V.Dest not in
        select V.Dest in Raggiungibile)

select distinct Dest, Costo
from Raggiungibile R
where Costo = (select min(Costo)
               from Raggiungibile R1
               where R.Dest = R1.Dest)
```

Constraints in the Data Definition Language (DDL)

Data quality

- Data quality
  - Correctness, completeness, up-to-date?
  - Quality of real data is often poor (5- 40% incorrect)
- To improve the data quality:
  - Integrity rules
  - Data manipulation by predefined programs (procedures and triggers)

Generic integrity constraints

- Predicates that must hold on correct (legal) instances of the database
- Expressed in two ways:
  - in the table's schema
  - as separate assertions

Check clause

- Allows to express arbitrary constraints in the schema definition.
- It appears immediately after the attribute, within the **create table** command.
- Syntax:

  `check` (*Condition*)
- *Condition* is what can appear in a `where` clause (including embedded queries), i.e.its evaluation returns a boolean value

Example

- Employee(Emp_ID,FirstName,LastName,Dept,Superior)
  - Managers, whose ID starts with digit 1, may not have a superior
  - Otherwise, an employee’s superior must be from the same department
- Example: constraints for the attribute *Superior* in the schema of the table *Employee*:

```
Superior character(6)
check (Emp_ID like “1%”
       or
       Department = (select Department
                     from Employee E
                     where E.Sid = Superior))
```

Assertions

- Assertions allow to define constraints outside of table definitions, by giving a name to a check clause
- Useful in many situations, for example, to express generic constraints between tables
- Syntax:

  create assertion *AssertionName* **check** (*Condition*)
- Ex: the table Employee must contain at least one tuple:

```
create assertion AlwaysOneEmployee
  check (1 <= (select count(*)
               from Employee))
```

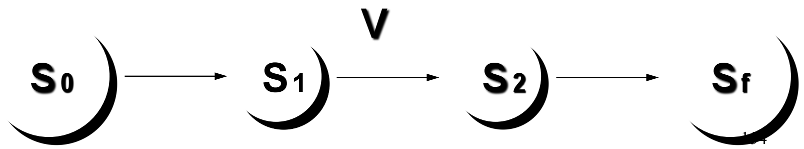
When are constraints checked?

immediate :

violation cancels the last modification

deferred (later):

violation cancels the whole application



Dynamic modification of the meaning of constraints

• Each constraint is defined as of a certain type (usually "immediate")

•The application can modify the intial type of constraints:

```
    - set constraints immediate
    - set constraints deferred
```

•Sooner or later, all constraints are checked.

Example: managing a shop

Shop

Prod_ID	QtDisp	QtOrder
1	150	100
3	130	80
4	170	50
5	500	150

Order

Prod_ID	Date	QtaOrd

Example: definition of the shop

```
create table Shop as
  ( Prod_ID    char(2) primary key,
    QtDisp     integer not null
                    check(QtDisp > 0),
    QtOrd      integer not null
               check(QtaOrd > 10)
               check(QtaDisp>QtaOrd)
)
```

Access control

Access control

- Privacy: protection of the DB in order to guarantee that only authorized users may access it
- Mechanisms to identify the user (by *password*):
  - When she connects to the computer system
  - When she connects to the DBMS
- Individual users, and user groups

Permissions

- Each component of a scheme can be protected (tables, attributes, views, domains, etc)
- A resources's owner (its creator) assigns **privileges (permissions)** to other users
- A pre-defined user **`_system`** represents the administrator, and has full access to all resources
- A privilege is specified by:
  - The resource
  - The user giving the privilege
  - The user receiving the privilege
  - The action that is allowed on the resource
  - The possibility to pass on the permission to other users

6 types of privileges in SQL

- **`insert`**: add a new object to the resource
- **`update`**: modify the resource's content
- **`delete`**: remove an object from the resource
- **`select`**: acces the resource's content in queries
- **`references`**: create a referential integrity constraint that involves the resource (may restrict the possibility to modify the resource!)
- **`usage`**: use the resource in a schema definition (particularly, a domain)

- **`all privileges:`** summarizes all 6 types

`Grant` and `revoke`

- Syntax to give a privilege to a user:

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

  - **`grant option`** indicates if the grant can be propagated to other users.

- To withdraw a privilege:

  `revoke` *Privileges* **`on`** *Resource* **`from`** *User*
  [**`restrict`** | **`cascade`**]

Examples

```
grant all privileges on Contract to User1
grant update(VALUE) on Contract to User2
grant select on Contract to User2, User3

revoke update on Contract from User1
revoke select on Contract from User3
```

Example of grant option

1 Database administrator

```
grant all privileges on Contract to User1
  with grant option
```

2 User1

```
grant select on Contract to User2
  with grant option
```

3 User2

```
grant select on Contract to User3
```

Withdrawing a privilege with cascade

1 Database administrator

```
grant select on Contract to User1
    with grant option
```

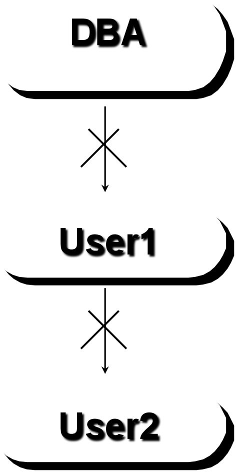
2 User1

```
grant select on Contract to User2
```

3 Database administrator

```
revoke select on Contract from User1 cascade
```

Withdrawing a privilege with cascade

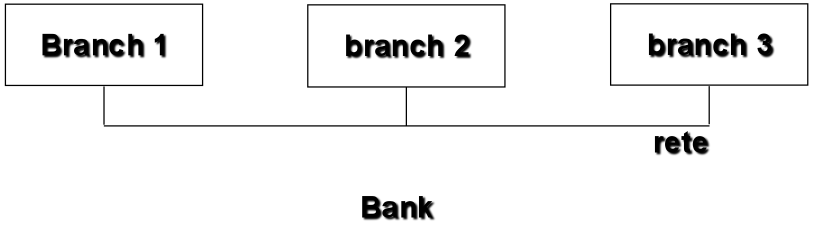


Views and access control

Views = unit of permission

- Allows the optimal management of privacy.

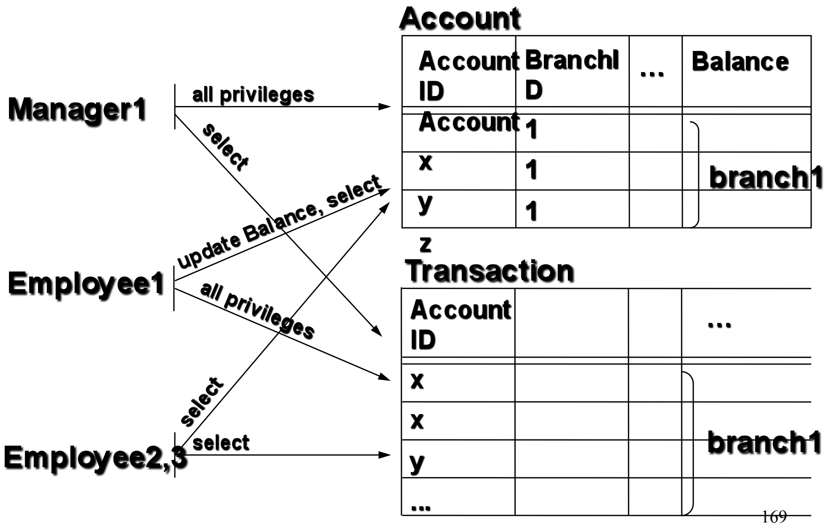
Example: managing bank accounts



Account(<u>AccountID</u>, BranchID, …, Balance)

Transaction(<u>AccountID, …</u>)

Access needs



Account

Account ID	BranchID	…	Balance
Account	1		
x	1		
y	1		

z

Transaction

Account ID			…
x			
x			
y			
…			

Views relative to the first branch

```
create view Account1 as
( select *
  from Account
  where branch = 1)

create view Transaction1 as
( select *
  from Transaction
  where AccountID in
         ( select AccountID
            from Account1 ) )
```

Permissions relative to data of the first branch

```
grant all privileges on Account1
        to Manager1
grant update(Balance) on Account1
        to Employee1
grant select on Account1
        to Employee1, Employee2, Employee3
grant select on Transaction1
        to Manager1
grant all privileges on Transaction1
        to Employee1
grant select on Transaction1
        to Employee2, Employee3
```