SQL Data Manipulation Language

- primarily declarative query language
- starting point: relational calculus

  aka first-order predicate logic
- many additions, bells and whistles…
- corresponding procedural language: relational algebra

will discuss relational calculus and algebra later

Running example: Movie database

Movie	Title	Director	Actor

Schedule	Theater	Title

Running example: Movie database

Movie	Title	Director	Actor
	Star Wars	Lucas	Ford
	Star Wars	Lucas	Fischer
	Mad Max	Miller	Hardy
	………………………………		

Schedule	Theater	Title
	Hillcrest	Star Wars
	Hillcrest	Mad Max
	Paloma	Rocky Horror
	…………………………..	

Find titles of currently playing movies

```
SELECT Title
FROM Schedule
```

Find the titles of all movies by “Lucas”

```
SELECT Title
FROM Movie
WHERE Director=“Lucas”
```

Find the titles and the directors of all currently playing movies

```
SELECT Movie.Title, Director
FROM Movie, Schedule
WHERE Movie.Title=Schedule.Title
```

Basic form

SELECT $a_1, \ldots, a_n$
FROM $R_1, \ldots, R_m$
WHERE *condition*

WHERE clause is optional

Informal semantics of basic form

```
SELECT $a_1, \dots, a_n$
FROM  $R_1, \dots, R_m$
WHERE $condition$
```

```
for each tuple $t_1$ in $R_1$
    for each tuple $t_2$ in $R_2$
        …….
            for each tuple $t_m$ in $R_m$

                if condition($t_1, t_2, \dots, t_m$) then output in answer
                attributes $a_1, \dots, a_n$ of $t_1, \dots, t_m$
```

Examples revisited

```
SELECT Title
FROM Movie
WHERE Director= “Lucas”
```

Semantics:

```
for each tuple m in Movie
if m(Director) = “Lucas”  then output m(Title)
```

Examples revisited

```
SELECT Movie.Title, Director
FROM Movie, Schedule
WHERE Movie.Title=Schedule.Title
```

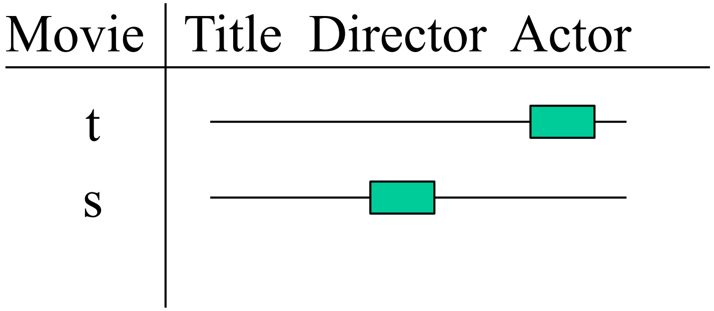
Semantics:

```
for each tuple m in Movie
   for each tuple s in Schedule
      if m(title) = s(title)  then output <m(Title),m(Director)>
```

SQL Queries: Tuple variables

- Needed when using the same relation more than once in the FROM clause
- Example: find actors who are also directors

```
SELECT  t.Actor
FROM  Movie  t, Movie s
WHERE t.Actor = s.Director
```



Semantics:

```
for each t in Movie
  for each s in Movie
     if  t(Actor) = s(Director) then output t(Actor)
```

Previous examples using tuple variables

```
SELECT Title
FROM Movie
WHERE Director= “Lucas”
```

```
SELECT m.Title
FROM Movie m
WHERE m.Director = “Lucas”
```

Previous examples using tuple variables

```
SELECT Movie.Title, Director
FROM Movie, Schedule
WHERE Movie.Title=Schedule.Title
```

```
SELECT m.Title,  m.Director
FROM Movie m, Schedule s
WHERE m.Title = s.Title
```

SQL Queries: * and LIKE

- Select all attributes using *

- Pattern matching conditions
  - *<attr>* LIKE *<pattern>*

Retrieve all movie attributes of currently playing movies

```
SELECT Movie.*
FROM Movie, Schedule
WHERE Movie.Title=Schedule.Title
```

Retrieve all movies where the title starts with “Ta”

```
SELECT *
FROM Movie
WHERE Title LIKE “Ta%”
```

Forgot if “Polanski” is spelled with “i” or “y”:

```
SELECT *
FROM Movie
WHERE Director LIKE “Polansk_”
```

SQL Queries: duplicate elimination

- *Default: answers to queries contain duplicates*

- *Duplicate elimination* must be explicitly requested
  - SELECT DISTINCT … FROM … WHERE …

```
SELECT Title
FROM Movie
```

Title
Tango
Tango
Tango

```
SELECT DISTINCT Title
FROM Movie
```

Title
Tango

Ordering the Display of Tuples

- List all titles and actors of movies by Fellini, in alphabetical order of titles

  select *Title, Actor*
  from *Movie*
  where *Director = 'Fellini'*
  ORDER BY *Title*

- We may specify **desc** for descending order or **asc** for ascending order, for each attribute;
  - ascending order is the default.
  - Example: **order by** *Title* **desc**

Renaming attributes in result

Done using the **as** construct:

Find titles of movies by Bertolucci, under attribute Berto-titles:

```
select title as Berto-title
from movie
where director = ‘Bertolucci’
```

Aggregate Functions

These functions operate on the multiset of values of a column of a relation, and return a value

avg: average value
min: minimum value
max: maximum value
sum: sum of values
count: number of values

Aggregate Functions (Cont.)

-- Find the average account balance at the La Jolla branch.

select avg *(balance)*
from *account*
where *branch_name* = 'La Jolla'

-- Find the number of tuples in the *customer* relation.

select count (*)
from *customer*

-- Find the number of depositors in the bank.

select count (distinct *customer_name)*
from *depositor*

Aggregate Functions (Cont.)

- Find the maximum salary, the minimum salary, and the average salary among all employees for the Company database

```
SELECT      MAX(SALARY),
            MIN(SALARY), AVG(SALARY)
            FROM  EMPLOYEE
```

Obs. Some SQL implementations *may not allow more than one aggregate function* in the SELECT-clause!

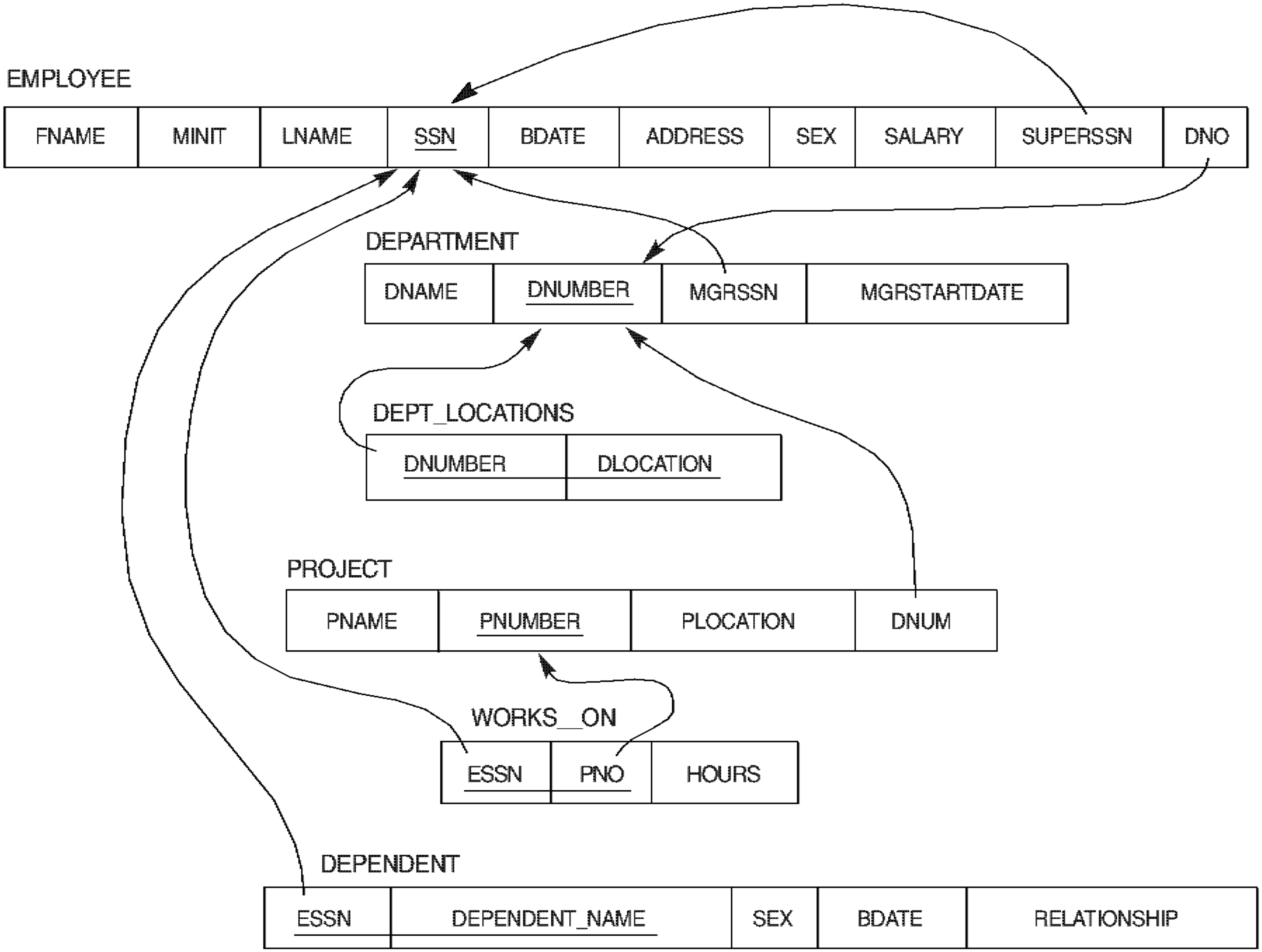
Aggregate Functions (Cont.)

- Find the maximum salary, the minimum salary, and the average salary among employees who work for the 'Research' department.

```
SELECT      MAX(SALARY), MIN(SALARY),
            AVG(SALARY)
FROM        EMPLOYEE, DEPARTMENT
WHERE       DNO=DNUMBER AND
            DNAME='Research'
```

Note: the aggregate functions are applied to the relation consisting of all pairs of tuples from Employee and Department satisfying the condition in the WHERE clause

(Reminder: company schema)



EMPLOYEE

FNAME | MINIT | LNAME | SSN | BDATE | ADDRESS | SEX | SALARY | SUPERSSN | DNO

DEPARTMENT

DNAME | DNUMBER | MGRSSN | MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER | DLOCATION

PROJECT

PNAME | PNUMBER | PLOCATION | DNUM

WORKS__ON

ESSN | PNO | HOURS

DEPENDENT

ESSN | DEPENDENT_NAME | SEX | BDATE | RELATIONSHIP

Grouping (example)

Employee

Name	Dept	Salary
Joe	Toys	45
Nick	PCs	50
Jim	Toys	35
Jack	PCs	40

Find average salary of all employees

SELECT Avg(Salary) AS AvgSal
FROM Employee

AvgSal
42.5

Find the average salary for each department

SELECT Dept, Avg(Salary) AS AvgSal
FROM Employee
GROUP BY Dept

Dept	AvgSal
Toys	40
PCs	45

Grouping

- Allows to apply the aggregate functions *to subgroups of tuples in a relation*
- Each subgroup of tuples consists of the set of tuples that have *the same value* for the *grouping attribute(s)*
- The function is applied to each subgroup independently
- SQL has a **GROUP BY**-clause for specifying the grouping attributes
- Most systems require the grouping attributes to appear in the SELECT-clause
- Non-grouping attributes may only appear in the SELECT clause as arguments to aggregate functions

R	A	B	C
	a	b	0
	a	b	1
	a	b	2
	a	c	1
	a	c	3

```
select   A, B,  MAX(C) as  M
from  R
group by A , B
```

A	B	M

```
select   A,  MAX(C) as  M
from  R
group by A , B
```

A	M

```
select   MAX(C) as  M
from  R
group by A , B
```

M

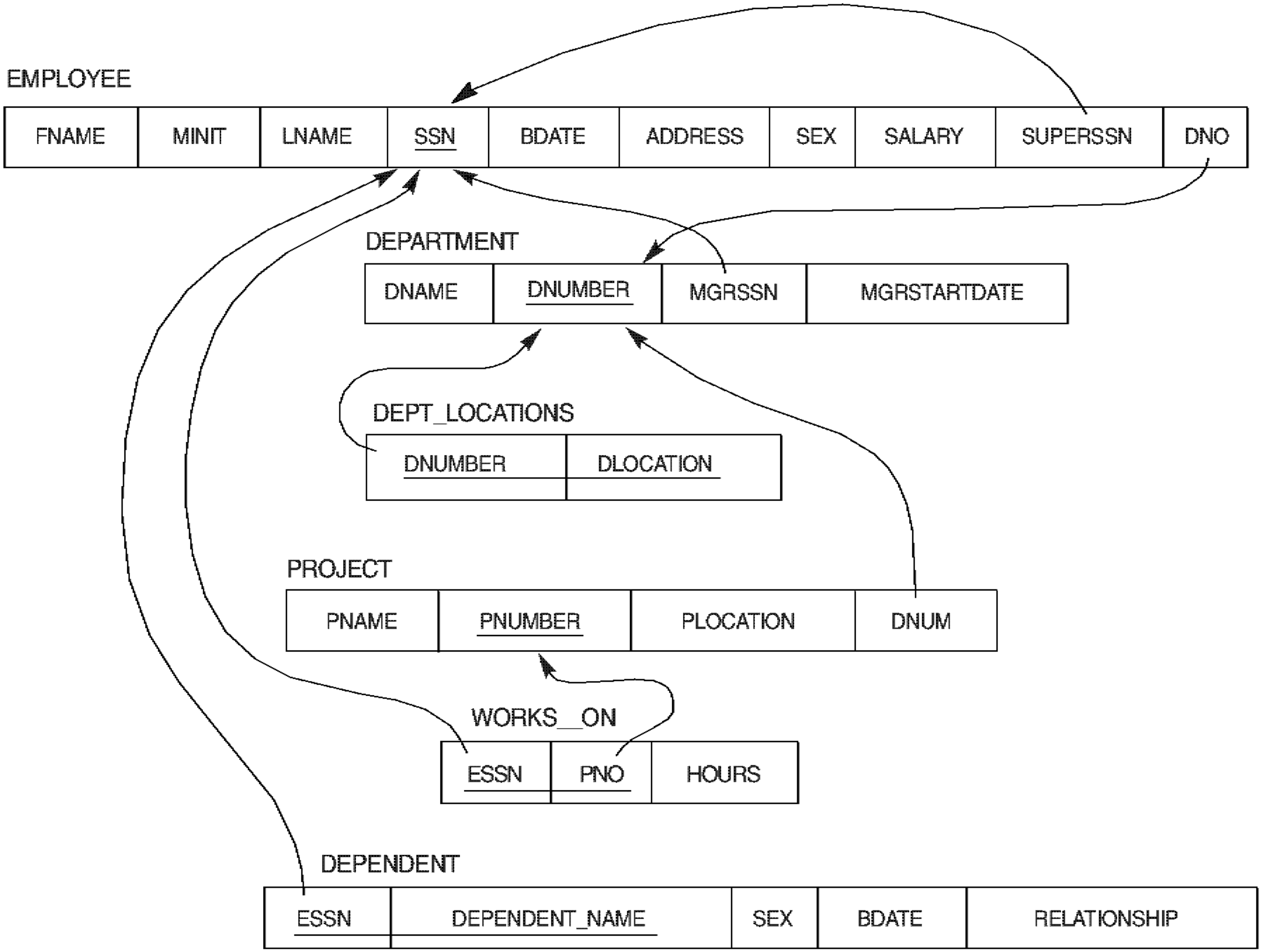
Grouping (cont.)

- For each department, retrieve the department number, the number of employees in the department, and their average salary.

```
SELECT  DNO, COUNT (*) AS NUMEMP,  AVG (SALARY) AS AVGSAL
FROM                  EMPLOYEE
GROUP BY              DNO
```

  - The EMPLOYEE tuples are divided into groups--each group having the same value for the grouping attribute DNO
  - The COUNT and AVG functions are applied to each such group of tuples separately
  - The SELECT-clause includes only the grouping attribute and the aggregate functions to be applied on each group of tuples

(Reminder: company schema)



EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS__ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

GROUPING Example

- For each project, retrieve the project number, project name, and the number of employees who work on that project.

```
SELECT      PNUMBER, PNAME, COUNT (*)
FROM        PROJECT, WORKS_ON
WHERE       PNUMBER=PNO
GROUP BY    PNUMBER, PNAME
```

  – Note: the grouping and functions are applied **on pairs of tuples from** **PROJECT,   WORKS_ON**

Subtlety: suppose PNO and ESSN do not form a key for WORKS_ON
Problem: will get duplicate employees

Works_on	ESSN	PNO	HOURS
	111-11-1111	001	20
	111-11-1111	001	10
	22-22-2222	002	25

PROJECT	PNAME,	PNUMBER
	Wiki	001
	Geo	002

Fix:

```
SELECT PNUMBER, PNAME, COUNT (DISTINCT ESSN)
FROM  PROJECT, WORKS_ON
WHERE  PNUMBER=PNO
GROUP BY  PNUMBER, PNAME
```

THE HAVING-CLAUSE

- Sometimes we want to retrieve the values of aggregate functions for only those *groups that satisfy certain conditions*
- The HAVING-clause is used for specifying a selection condition on groups (rather than on individual tuples!)

Aggregate Functions – Having Clause

- Find the name and average balance of all branches where the average account balance is more than $1,200.

```
select      branch_name, avg (balance)
from        account
group by    branch_name
HAVING      avg (balance) > 1200
```

Condition in HAVING clause use values of attributes in group-by clause and aggregate functions on the other attributes

THE HAVING-CLAUSE (cont.)

- For each project *on which more than two employees work* , retrieve the project number, project name, and the number of employees who work on that project.

```
SELECT          PNUMBER, PNAME, COUNT (*)
FROM            PROJECT, WORKS_ON
WHERE           PNUMBER=PNO
GROUP BY        PNUMBER, PNAME
HAVING          COUNT (*) > 2
```

Note: predicates in the having clause are applied after the formation of groups whereas predicates in the where clause are applied before forming groups

Another example

For each currently playing movie having more than 100 actors, find the number of theaters showing the movie

```
SELECT m.Title,   COUNT(distinct s.Theater) AS number
FROM Schedule s, Movie m
WHERE s.Title = m.Title
GROUP BY m.Title
HAVING COUNT(DISTINCT m.Actor)   > 100
```

Aggregate is taken over pairs <s,m> with same Title

Schedule	Theater	Title
	Hillcrest	Star Wars
	Paloma	Star Wars

Movie	Title	Director	Actor
	Star Wars	Lucas	Ford
	Star Wars	Lucas	Fischer

FROM Schedule s, Movie m
WHERE s.Title = m.Title

	Theater	Title	Director	Actor
	Hillcrest	Star Wars	Lucas	Ford
	Paloma	Star Wars	Lucas	Ford
	Hillcrest	Star Wars	Lucas	Fischer
	Paloma	Star Wars	Lucas	Fischer

GROUP BY m.Title

Title	Theater	Director	Actor
Star Wars	Hillcrest	Lucas	Ford
	Paloma	Lucas	Ford
	Hillcrest	Lucas	Fischer
	Paloma	Lucas	Fischer

SQL Queries: Nesting

- The WHERE clause can contain predicates of the form
  - *attr/value* IN *<SQL query>*
  - *attr/value* NOT IN *<SQL query>*
- The IN predicate is satisfied if the *attr* or *value* appears in the result of the nested *<SQL query>*

Examples:

find directors of current movies

```
SELECT director FROM Movie
WHERE title IN
          (SELECT title
          FROM schedule)
```

The nested query finds currently playing movies

More examples

Find actors playing in some movie by Bertolucci

```
SELECT actor FROM Movie
WHERE title IN
        (SELECT title
        FROM Movie
        WHERE director = “Bertolucci”)
```

The nested query finds the titles of movies by Bertolucci

In this case, can eliminate nesting:

```
SELECT actor FROM Movie
WHERE title IN
           (SELECT title
           FROM Movie
           WHERE director = “Bertolucci”)
```

```
SELECT m1. actor
FROM Movie m1, Movie m2
WHERE m1.title = m2.title
          and m2.director = “Bertolucci”
```

Question: is nesting syntactic sugar? Can it always be eliminated?

A: yes B: no

Question: is nesting syntactic sugar? Can it always be eliminated?

A: yes B: no

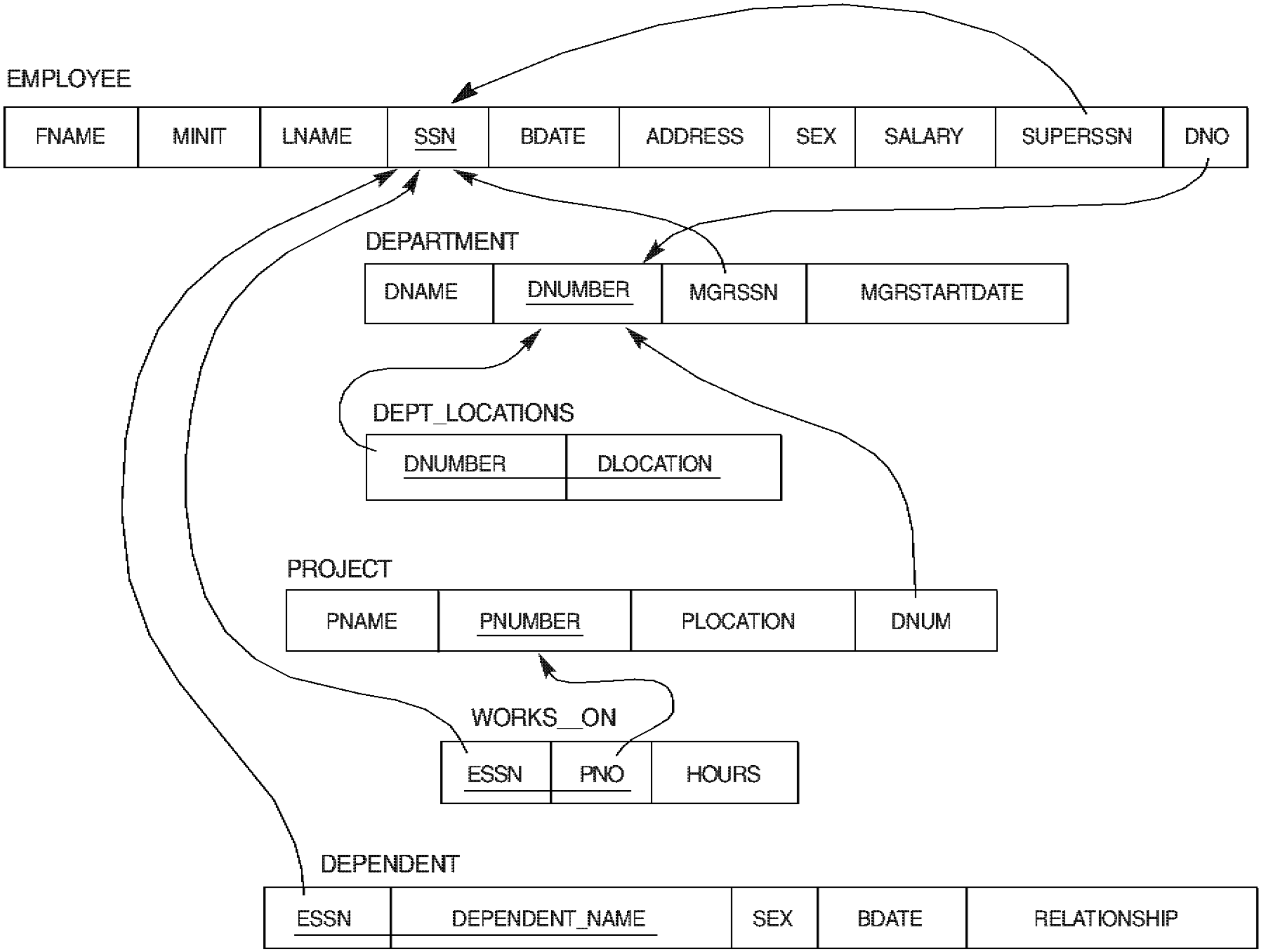
Queries involving nesting but no negation can always be un-nested, unlike queries with nesting and negation

Correlated nested queries

- If a condition in the WHERE-clause of a *nested query* references an attribute of a relation declared in the *outer query* , the two queries are said to be ***correlated***
- The result of a correlated nested query may be different for each tuple (or combination of tuples) of the relation(s) the outer query
- E.g. DB Company: *Retrieve the name of each employee who has a dependent with the same first name as the employee.*

```
SELECT   E.FNAME, E.LNAME
FROM      EMPLOYEE  E
WHERE    E.SSN  IN
              (SELECT ESSN
               FROM     DEPENDENT
               WHERE   ESSN=E.SSN
                AND       E.FNAME=DEPENDENT_NAME)
```

(Reminder: company schema)



EMPLOYEE

FNAME	MINIT	LNAME	SSN	BDATE	ADDRESS	SEX	SALARY	SUPERSSN	DNO

DEPARTMENT

DNAME	DNUMBER	MGRSSN	MGRSTARTDATE

DEPT_LOCATIONS

DNUMBER	DLOCATION

PROJECT

PNAME	PNUMBER	PLOCATION	DNUM

WORKS__ON

ESSN	PNO	HOURS

DEPENDENT

ESSN	DEPENDENT_NAME	SEX	BDATE	RELATIONSHIP

Correlated nested queries (cont.)

- Correlated queries using just the IN comparison operators can ***still be unnested***
- For example, the previous query could be un-nested as follows:

```
SELECT    E.FNAME, E.LNAME
FROM   EMPLOYEE  E, DEPENDENT  D
WHERE  E.SSN=D.ESSN  AND
E.FNAME=D.DEPENDENT_NAME
```

Use of NOT IN tests increases expressive power

Simple use of NOT IN

Find all movies in which Hitchcock does not act:

```
SELECT  title FROM Movie
Where  title NOT IN
            (SELECT title FROM Movie
              WHERE actor  = ‘Hitchcock’)
```

Simple use of NOT IN

Find all movies that are not currently playing:

```
SELECT  title FROM Movie
WHERE title NOT IN
            (SELECT title FROM Schedule)
```

Why can’t these be flattened?

Hand-waving “proof” :

1. Basic queries with no nesting are monotonic

the answer never decreases when the database increases

$DB1 \subseteq DB2$ implies $Query(DB1) \subseteq Query(DB2)$

2. But queries using NOT IN are usually not monotonic

```
SELECT  title FROM Movie
          WHERE title NOT IN
                   (SELECT title FROM Schedule)
```

If Schedule increases, the answer may decrease

Recall semantics of basic queries:

```
SELECT $a_1, …, a_n$
FROM  $R_1, …, R_m$
WHERE condition
```

```
for each tuple $t_1$ in $R_1$
    for each tuple $t_2$ in $R_2$
        …….
            for each tuple $t_m$ in $R_m$

                if  condition($t_1,t_2, … ,t_m$) then output in answer
                attributes $a_1,…,a_n$ of $t_1,…,t_m$
```

This is monotonic if condition has no nested queries

Monotonic (A) or non-monotonic (B) ?

1. Find the theaters showing some movie by Fellini
2. Find the theaters showing only movies by Fellini
3. Find the actors who are also directors
4. Find the actors playing in some movie showing at Paloma
5. Find the actors playing in every movie by Bertolucci

Schedule	Theater	Title

Movie	Title	Director	Actor

More complex use of NOT IN

Find the names of employees with the maximum salary

```
SELECT  name FROM Employee
WHERE salary NOT IN
               (SELECT  e.salary
                FROM Employee e, Employee f
                WHERE   e.salary < f.salary)
```

Intuition: salary is maximum if it is not among salaries e.salary lower than some f.salary

More complex use of NOT IN:

Find actors playing in every movie by "Berto"

SQL's way of saying this:

find the actors for which there is no movie by Bertolucci in which they do not act

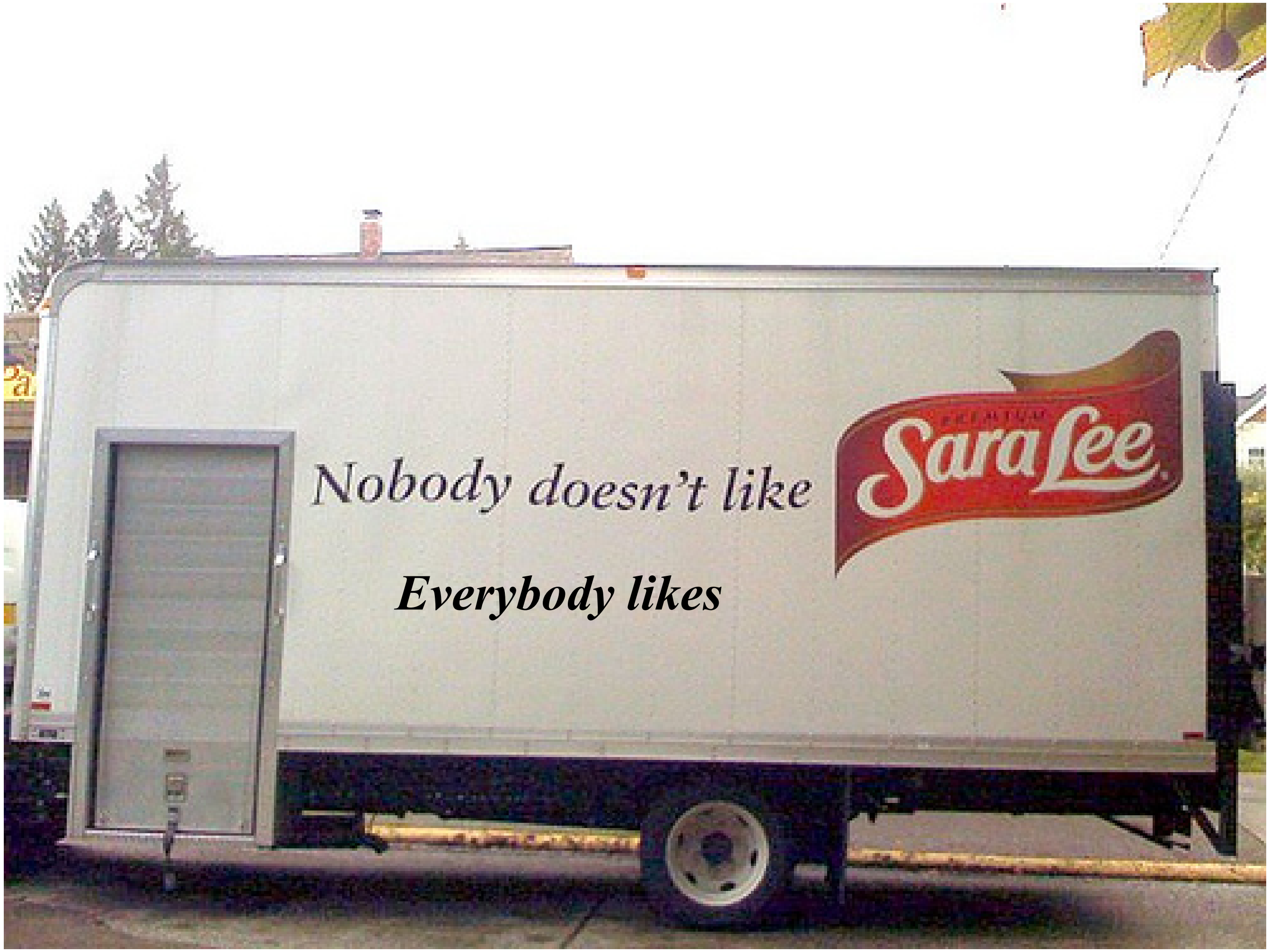
OR equivalently:

find the actors not among the actors for which there is some movie by Bertolucci in which they do not act

```
SELECT Actor FROM Movie
WHERE Actor NOT IN
      (SELECT m1.Actor
        FROM Movie m1, Movie m2,
        WHERE m2.Director="Berto"
          AND m1.Actor NOT IN
                  (SELECT Actor
                    FROM Movie
                    WHERE Title=m2.Title))
```

The shaded query finds actors for which there is some movie by "Berto" in which they do not act

The top lines complement the shaded part



Everybody likes

- Another construct used with nesting: EXISTS

SELECT …
FROM…
WHERE … EXISTS (<query>)

Meaning of EXISTS:

EXISTS (<query>) is true iff the result of <query> is not empty.

NOT EXISTS (<query>) is true iff the result of <query> is empty.

Examples:

R	A	B
	1	1
	3	2
	5	7

EXISTS (select A from R where A < 4)

NOT EXISTS (select A from R where A > 10)

EXISTS (select * from R where A < B)

NOT EXISTS (select * from R where A <B)

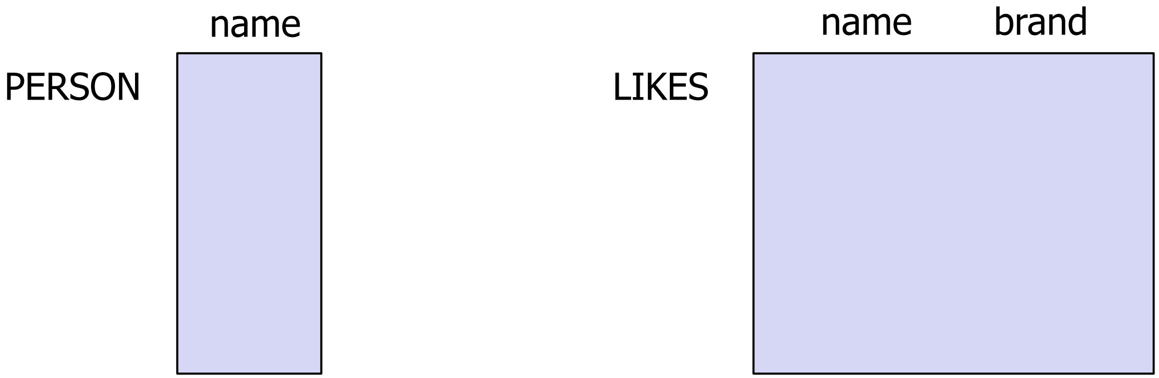
Example: Find theaters showing a movie directed by Berto:

```
SELECT s.theater
FROM  schedule s
WHERE EXISTS (SELECT * FROM movie
                            WHERE movie.title = s.title AND
                                          movie.director = ‘Berto’ )
```

Schedule	Theater	Title

Movie	Title	Director	Actor

Example (Boolean predicate): Everybody likes Sara Lee



```
NOT EXISTS
    (SELECT * FROM PERSON
      WHERE  NOT EXISTS
         (SELECT * FROM LIKES
          WHERE PERSON.name = LIKES.name
                        AND brand = ‘Sara Lee’
```

Example: Find the actors playing in every movie by Berto

```
SELECT a.actor FROM movie a
WHERE NOT EXISTS
    (SELECT * FROM movie m
    WHERE m.director = ‘Berto’ AND
       NOT EXISTS
           (SELECT *
           FROM movie t
           WHERE m.title = t.title
           AND t.actor = a.actor))
```

Movie	Title	Director	Actor

Nested Queries: ANY and ALL

- *A op* ANY <*nested query*> is satisfied if ***there is*** a value *X* in the result of the <*nested query*> and the condition *A op X* is satisfied
  - ANY aka SOME
- *A op* ALL <*nested query*> is satisfied if ***for every*** value *X* in the result of the <*nested query*> the condition *A op X* is satisfied

Find directors of currently playing movies

```
SELECT Director
FROM Movie
WHERE Title = ANY
    SELECT Title
    FROM Schedule
```

Find the employees with the highest salary

```
SELECT Name
FROM Employee
WHERE Salary >= ALL
    SELECT Salary
    FROM Employee
```

Nested Queries: Set Comparison

- *<nested query 1>* CONTAINS *<nested query 2>*

The original SQL as specified for SYSTEM R had a CONTAINS operator. This was dropped from the language, possibly because of the difficulty in implementing it efficiently

Find actors playing in every movie by "Bertolucci"

```
SELECT m1.Actor
FROM Movie m1
WHERE
            (SELECT  Title
             FROM Movie
             WHERE  Actor = m1.Actor)
            CONTAINS
            (SELECT Title
             FROM Movie
             WHERE Director = "Berto")
```

Nested queries in FROM clause

SQL allows nested queries in the FROM clause

Find directors of movies showing in Hillcrest:

```
select  m.director
from movie m,
(select title from schedule
where theater = ‘Hillcrest’) t
where m.title = t.title
```

Note: this is syntactic sugar and can be eliminated

SQL:Union, Intersection, Difference

- *Union*
  - *<SQL query 1>* UNION *<SQL query 2>*
- *Intersection*
  - *<SQL query 1>* INTERSECT *<SQL query 2>*
- *Difference*
  - *<SQL query 1>* EXCEPT *<SQL query 2>*

These operations eliminate duplicates

Find all actors or directors

```
(SELECT Actor as Name
 FROM Movie)
UNION
(SELECT Director as Name
 FROM Movie)
```

Find all actors who are not directors

```
(SELECT Actor as Name
 FROM Movie)
EXCEPT
(SELECT Director as Name
 FROM Movie)
```

SQL:Union, Intersection, Difference

- *Union*
  - *<SQL query 1>* UNION ALL *<SQL query 2>*
- *Intersection*

  <SQL query 1> INTERSECT ALL *<SQL query 2>*
- *Difference*
  - *<SQL query 1>* EXCEPT ALL *<SQL query 2>*

To keep duplicates : ALL

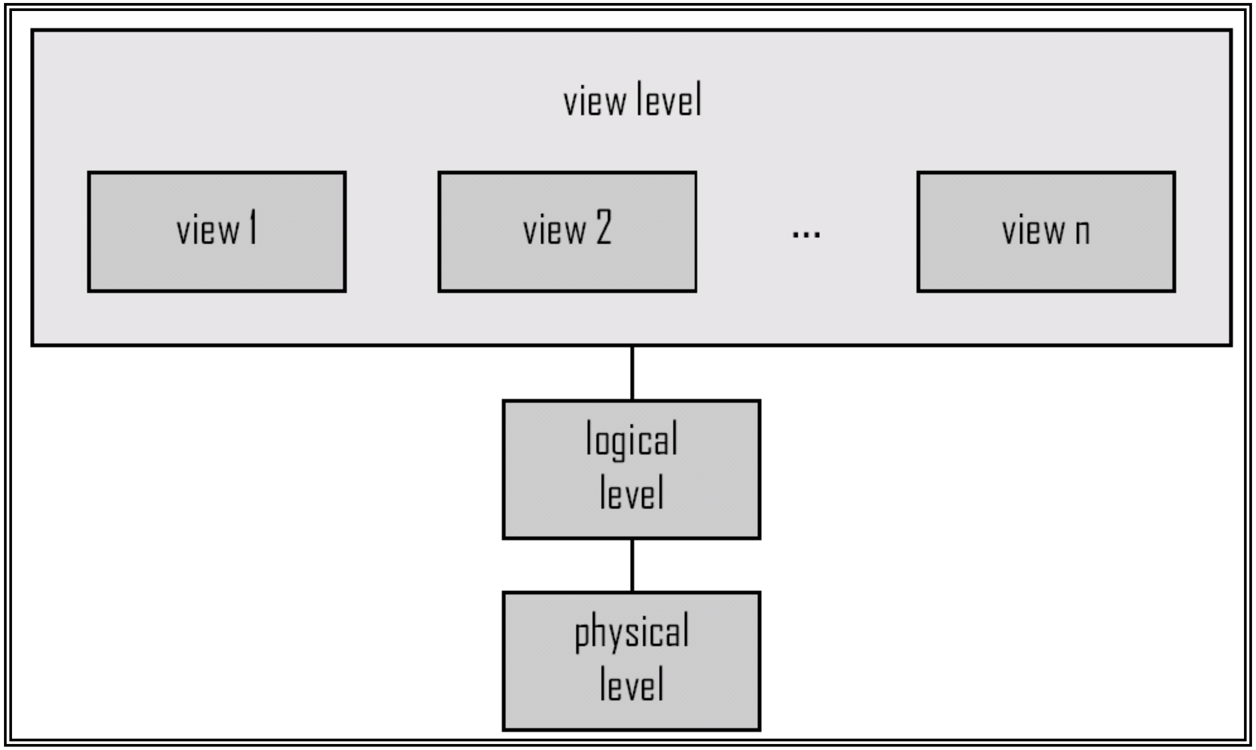
Example (union):

for each title in movie, find the number of theaters showing that title

schedule	theater	title

movie	title	director	actor

Basic Views (more later)



Views are a mechanism for customizing the database; also used for creating temporary virtual tables

Views

- In some cases, it is not desirable for all users to see the entire logical model (i.e, all the actual relations stored in the database.)
- Consider a person who needs to know customers' loan numbers but has no need to see the loan amounts. This person should see a relation described, in SQL, by

> (**select** *customer_name, loan_number*
> **from** *customer c, borrower b*
> **where** *c.customer_id* = *b.customer_id*)

- A **view** provides a mechanism to hide or restructure data for certain users.
- Any relation that is not in the database schema but is made visible to a user as a "virtual relation" is called a **view**.

Bank relational schema

- *branch* = (*branch_name*, *branch_city*, *assets*)
- *loan* = (*loan_number*, *branch_name*, *amount*)
- *account* = (*account_number*, *branch_name* , *balance*)
- *borrower* = (*customer_id*, *loan_number*)
- *depositor* = (*customer_id*, *account_number*)
- *customer = (customer_id, customer_name)*

View Definition

- A view is defined using the **create view** statement which has the form

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

  where *V* is the view name and <query expression> is any legal SQL query. A list of attribute names for *V* is optional.
- Once a view is defined, the view name can be used in queries
- Only limited updates can be applied to the view (more later)
- View definition is not the same as creating a new relation by evaluating the query expression: the view contents is refreshed automatically when the database is updated

Examples

- A view consisting of bank branches and all their customers

```
create view all_customers  as
          (select branch_name, customer_id
           from depositor d, account a
          where  d.account_number = a.account_number)
          union
          (select branch_name, customer_id
          from  borrower b, loan l
          where   b.loan_number = l.loan_number)
```

- Find all customers of the La Jolla branch

```
select customer_id
from all_customers
where branch_name = 'La Jolla'
```

Views can simplify complex queries

Example

find actors playing in every movie by “Berto”:

```
SELECT Actor FROM Movie
WHERE Actor NOT IN
          (SELECT m1.Actor
            FROM Movie m1, Movie m2,
            WHERE m2.Director=“Berto”
              AND m1.Actor NOT IN
                     (SELECT Actor
                       FROM Movie
                       WHERE Title=m2.Title))
```

The shaded query finds actors NOT playing in some movie by “Berto”

Same query using views

```
CREATE VIEW Berto-Movies AS
 SELECT title FROM Movie WHERE director = “Bertolucci” ;
```

```
CREATE VIEW Not-All-Berto AS
SELECT m.actor FROM Movies m, Berto-Movies
WHERE  Berto-Movies.title NOT IN
                    (SELECT title FROM Movies
                    WHERE actor = m.actor);
```

```
SELECT actor FROM Movies WHERE actor NOT IN
                                        (SELECT * FROM Not-All-Berto)
```

Another syntax: the **with** clause

```
WITH  Berto-Movies   AS
 SELECT title FROM Movie WHERE director = “Bertoucci”

WITH   Not-All-Berto  AS
SELECT m.actor FROM Movies m, Berto-Movies
WHERE  Berto-Movies.title NOT IN
                 (SELECT title FROM Movies
                 WHERE actor = m.actor)

SELECT actor FROM Movies WHERE actor NOT IN
                   (SELECT * FROM Not-All-Berto) ;
```

Note: Berto-Movies and Not-All-Berto are temporary tables, not views

Another example:

Employee	SSN	salary	dept

Find the employees working in the departments with the highest average salary

```
create view Avg-sal  as
select  dept,  AVG(salary) as average
from Employee
group by dept;

select e.SSN from Employee e,  Avg-sal a
where e.dept  = a.dept and a.average =
(select MAX(average) from Avg-sal)
```