# DML SQL II

More complex queries

# Subqueries

- So far all the constraint implemented using `WHERE` involve only numbers (or strings) but not relations.
- But it is possible to compare the value of an attribute with a relation (IN, EXISTS, ALL, ANY, BETWEEN...)

# “Traditional” Subqueries

- 'Star Wars' cast (movie_id= 1)

```
SELECT name
FROM actor natural join
  casting
WHERE movie_id=1;
```

# "Compare" and attribute with a relation

- Instead of using a natural join we may compare the actor id with the output of a query

```
SELECT name
FROM actor
WHERE  actor_id =
   (SELECT DISTINCT actor_id
   FROM casting
   WHERE movie_id = 1
   );
```

| actor_id |
| --- |
| 12345 |

**Run-time error, if subquery returns more than one element**

# How to use relations in the "where" clause

1. `EXISTS` ***R*** : TRUE if R is an non empty relation
2. ***s*** `IN` ***R*** : TRUE if s ∈ R
3. ***s*** **op** `ALL` ***R***, op = {<,>,<>,=, …}: TRUE if s is greater than, less than, etc all elements ∈ R.
4. ***s*** **op** `ANY` ***R*** : TRUE if ***s*** is greater , less, … than any element ∈ R

# Example: IN

- Movies in which ‘Harrison Ford’ is the star
- **1:** use cartesian product

```
SELECT title
  FROM actor natural join movie natural join
  casting
  WHERE name = 'Harrison Ford';
```

# Example (Cont)

- **Opción 2:** Usando consultas anidadas y el operador IN

```
SELECT title
FROM movie
WHERE movie_id IN
   (SELECT movie_id
    FROM casting
    WHERE (actor_id) IN
          (SELECT actor_id
          FROM actor
          WHERE name = 'Harrison Ford'
          )
   );
```

# Example I

Title (and star) of all movies in which 'Julie Andrews' worked

- Movie_id of all movies

```
SELECT movie_id
FROM casting natural join actor
WHERE name='Julie Andrews';
```

- Titles and stars
- EXPLAIN

```
SELECT title, name
FROM movie natural join casting natural join
actor
WHERE ord=1
AND movie_id IN (SELECT movie_id
                 FROM casting natural join
actor
                 WHERE name='Julie Andrews');
```

# Example 2 (Cont)

```
EXPLAIN
SELECT title, Actor1.name
FROM movie, casting Casting1,
                casting Casting2,
                actor Actor1,
                actor Actor2
WHERE Casting1.movie_id=movie.movie_id
  AND Casting2.movie_id=movie.movie_id
  AND Casting1.actor_id=Actor1.actor_id
  AND Casting2.actor_id=Actor2.actor_id
  AND Casting1.ord=1
  AND Actor2.name='Julie Andrews'
;
```

# Subqueries than are evaluated several times

- Sometimes you need to evaluate the subquery for each value of the attribute
- Let us reuse this query (consistency check)

```
SELECT Star1.name, Star1.actor_id
FROM actor Star1, actor Star2
WHERE Star1.name = Star2.name
AND Star1.actor_id <
  Star2.actor_id;
```

# Correlated Subqueries

- Consistency Check

```
SELECT name, actor_id
FROM actor Star1
WHERE name = ANY
(SELECT actor.name
 FROM actor
 WHERE actor.actor_id <
       Star1.actor_id AND
          actor.name = Star1.name
);
```

# Ejemplos rewrite

## Actores who have work with 'Orson Welles'

- Orson welles' movie_id

```
SELECT movie_id
FROM casting natural join actor
WHERE name = 'Orson Welles';
```

- Name of the actor who have work in the above movies

```
SELECT name, title
FROM actor natural join  casting natural join
movie
WHERE movie_id IN (SELECT movie_id
FROM casting natural join  actor WHERE
name='Orson Welles');
```

```
SELECT title, Actor1.name
FROM movie, casting casting1,
            casting casting2,
            actor Actor1,
            actor Actor2
WHERE casting1.movie_id=movie.movie_id
  AND casting2.movie_id=movie.movie_id
  AND casting1.actor_id=Actor1.actor_id
  AND casting2.actor_id=Actor2.actor_id
  AND Actor2.name='Orson Welles'
;
```

# Another example

- *Actors who have worked in a movie before 1930*

*Re-write as*

- *Find those actors that satisfy the constraint: there is a movie in which they have work and the movie premiere happened before 1930*

```
SELECT DISTINCT name
FROM casting natural join actor
WHERE EXISTS (
    SELECT *
    from movie
    WHERE year < 1930 AND movie.movie_id =
casting.movie_id);
```

```
SELECT DISTINCT name
  FROM casting natural join
       actor natural join movie
  WHERE
        year < 1930;
```

# Subqueries in FROM clause

- It is possible to use a subquery as input relation
- Must have an alias (postgres)
- Movies that have their premiere in 1978 sorted by cast size (number of actor)

```
SELECT title, Temp.cc
FROM (SELECT movie_id, COUNT(actor_id)
      AS cc FROM casting natural join  movie
      WHERE year=1978
      GROUP BY movie_id) AS Temp natural join movie
ORDER BY Temp.cc;
```

COUNT: devuelve el numero de tuplas

# Aggregation Functions

- Same as in Relation algebra
  - SUM, AVG, MIN, MAX, COUNT
- *How many entries are in the data base?*

```
SELECT  COUNT(*)
FROM movie;
```

- Remember that SQL does not drop duplications

```
SELECT  COUNT(name) FROM actor;
SELECT  COUNT(DISTINCT name)FROM actor;
```

# Grouping Tuples

- As in algebra, many times we want to apply the operation to a subsets of tuples
- Table with the number of times each actor has been star
- We want something that looks like:

| name | Count (ord=1) |
|---|---|
| Pedro Pérez | 1 |
| Joe Dalton | 3 |

- GROUP BY is used for grouping

# GROUP BY

```
SELECT A1, SUM(A2)
FROM R1
GROUP BY A1;
```

- IMPORTANT: in SELECT you ONLY may find
  (1) Aggregations
  (2) Attributes that appear in the clause GROUP BY

# Example GROUP BY redo

- *Table with actors sorted by the number of times they are stars (just for those with more than 10 movies)*

```
CREATE VIEW ranking AS
    SELECT actor_id, count(*) AS stars
    FROM casting
    WHERE ord=1
    GROUP BY actor_id;

SELECT name, stars
FROM actor natural join  ranking
WHERE  stars > 10
ORDER BY stars;
```

# HAVING

- Allows to select subgroups

```
SELECT x, sum(y)
FROM test1
GROUP BY x
HAVING sum(y) > 3;
```

# SQL query: summary

SELECT
FROM

compulsory

optional

WHERE
GROUP BY
HAVING
ORDER BY

# How a query is processed

1. Create relation described in `FROM`
2. Apply constrains described in `WHERE`
3. If there is NO "group-by", project the relation as described in `SELECT`. Then do `ORDER BY`. The End.
4. If there is `GROUP-BY` clause. Group the original relation in subsets
5. Apply `HAVING`
6. And then `SELECT`.
7. `ORDER BY`.

# Procesamiento de una consulta con funciones de agregacion