Writing Queries involving more than one Relation

CSC365
Spring 2019

Tuple Variables (Renaming Relations ρ)

- Sometimes, you want to alias the name of a relation to a short string

  SELECT **m.**mid, title, year, genre

  FROM Movies **as m**, Genre **as g**

  WHERE **m.**mid=**g.**mid;

- This means that each tuple in Movies is aliased as m, each tuple in Genre is aliased as g.
- A tuple variable -> A variable for each tuple in a relation.

Interpreting Multirelation Queries as Nested Loops

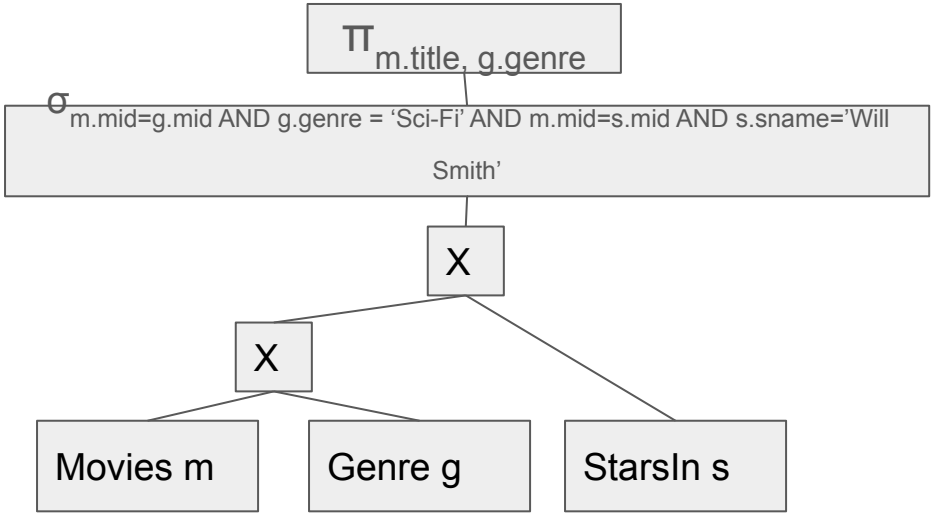
```
SELECT m.title, g.genre

FROM Movies m, Genre g, StarsIn s

WHERE m.mid = g.mid

 AND g.genre = 'Sci-Fi'

 AND m.mid = s.mid

 AND s.sname = 'Will Smith';
```

```
For each tuple m in Movies:
    For each tuple g in Genre:
        For each tuple s in StarsIn:
            If m.mid == g.mid
                AND g.genre == ‘Sci-Fi’
                AND m.mid == s.mid
                AND s.sname == ‘Will Smith’:
                    Produce m.title, g.genre
```

Converting Multirelation Queries to Relational Algebra

```
SELECT m.title, g.genre

FROM Movies m, Genre g, StarsIn s

WHERE m.mid = g.mid

 AND g.genre = 'Sci-Fi'

 AND m.mid = s.mid

 AND s.sname = 'Will Smith';
```

$$\pi_{\text{m.title, g.genre}}(\sigma_{\text{m.mid=g.mid AND g.genre = 'Sci-Fi' AND m.mid=s.mid AND s.sname='Will Smith'}}(\rho_m(\text{Movies}) \times \rho_g(\text{Genre}) \times \rho_s(\text{StarsIn})))$$



Subqueries

A query constructed out of the results of other queries - Nesting queries

1. Subqueries can return a single constant, which can be used in **WHERE** clause.
2. Subqueries can return relations that can be used in various ways in **WHERE** clauses.
3. Subqueries can appear in **FROM** clauses, followed by a tuple variable that represents the tuples in the result of the subquery.

Subqueries that produce Scalar Values

```
SELECT MAX(imdb) FROM Movies;
```

```
+-----------+
| MAX(imdb) |
+-----------+
|         9 |
+-----------+
```

```
SELECT * FROM Movies WHERE imdb = 9;
```

```
SELECT *

FROM Movies

WHERE imdb = (SELECT MAX(imdb) FROM Movies);
```

Conditions Involving Relations

- R - A one column relation produced by a subquery.
- EXISTS R
  - TRUE if and only if R is not empty.
- S IN R
  - TRUE if and only if S is equal to one of the values in R.
- S NOT IN R
  - TRUE if and only if S is not equal to any values in R.
- S > ALL R
  - TRUE if and only if S is greater than every value in R.
- S > ANY R
  - TRUE if and only if S is greater than at least one of the values in R.

S IN / NOT IN R

- *s* IN *R* is true if and only if *s* is equal to one of the values in *R*.
  - s may be a single value or single column name, in which case R must be a single-column relation
  - If s has more than one element, the number of elements must match the number of columns in *R*

- *s* NOT IN *R* is true if any only if *s* is equal to *no value* in *R*

IN / NOT IN Example

```
#Print the team which did not win a single game.

SELECT t.id, t.name

FROM Team t

WHERE t.id not in (

SELECT t.id From Game g JOIN Team t ON t.id = g.home_team_id and
g.score_home > score_away or t.id = g.away_team_id and g.score_away >
g.score_home

);
```

IN / NOT IN Example

```
-- Students without a minor who are in a college other than OCOB
SELECT *
FROM Student
WHERE MinorCode IS NULL
AND MajorCode NOT IN (
  SELECT Code
  FROM Department
  WHERE College = 'OCOB'
)
```

How does this query change if we use `JOIN` rather than a subquery?

ALL / ANY

s <comparator> ALL *R*
s <comparator> ANY *R*
(comparator may be any of our six basic comparisons: >, <, >=, <=, =, <>)

s > ALL *R* is true if any only if *s* is greater than <u>every</u> value in *unary* relation *R*. *s* <> ALL *R* is the same as *s* NOT IN *R*

s > ANY *R* is true if and only if *s* is greater than <u>at least one</u> value in *unary* relation *R*. *s* = ANY *R* is the same as *s* IN *R*

NOT ALL / ANY

ALL / ANY may be negated just like any other boolean expression. For example:

NOT *s* >= ALL *R* is true if and only if *s* is not the maximum value in *R*

NOT *s* > ANY *R* is true if any only if *s* is the minimum value in *R*

ALL Example

```
-- Find departments established before every department in Engineering
SELECT *
FROM Department
WHERE DateEstablished < ALL (
  SELECT DateEstablished
  FROM Department WHERE College = 'CENG'
)
```

ANY Example

```
-- Find the earliest-established department
SELECT *
FROM Department
WHERE NOT DateEstablished > ANY (
  SELECT DateEstablished
  FROM Department
)
```

(NOT) EXISTS R

- EXISTS *R* is true if and only if *R* is not empty
- NOT EXISTS *R* is true if and only if *R* is empty

(NOT) EXISTS Example

```
-- Find students who are the only students in their major
SELECT *
FROM Student o
WHERE MajorCode IS NOT NULL
AND NOT EXISTS (
  SELECT StudentID
  FROM Student i
  WHERE o.MajorCode = i.MajorCode AND o.StudentID <> i.StudentID
)

-- Find students who share a major with at least one other student
… (change NOT EXISTS to EXISTS) …
```

EXISTS / NOT EXISTS Transformation

Subqueries in the `WHERE` clause using IN can be transformed into equivalent expressions using `EXISTS` & `NOT EXISTS`.

```
SELECT *
FROM Student
WHERE MajorCode IN (
  SELECT Code
  FROM Department
  WHERE College = 'OCOB'
)
```

```
SELECT *
FROM Student
WHERE EXISTS (
  SELECT Code
  FROM Department
  WHERE College = 'OCOB'
  AND Code = Student.MajorCode
)
```

EXISTS / NOT EXISTS Transformation

List students who were enrolled on/after all departments in CENG were established.

```
SELECT *
FROM Student
WHERE DateEnrolled >= ALL (
  SELECT DateEstablished
  FROM Department
  WHERE College = 'CENG'
)
```

```
SELECT *
FROM Student
WHERE NOT EXISTS (
  SELECT DateEstablished
  FROM Department
  WHERE College = 'CENG'
  AND Student.DateEnrolled < DateEstablished
)
```

EXISTS / NOT EXISTS Transformation

Consider relations R(A, B) and S(C)

```
SELECT C
FROM S
WHERE C IN (
  SELECT SUM(B)
  FROM R
  GROUP BY A
)
```

```
SELECT C
FROM S
WHERE EXISTS (
  SELECT SUM(B) FROM R
  GROUP BY A
  HAVING SUM(B) = S.C
)
```

Conditions Involving Tuples

We've seen how IN, ANY, and ALL work with one-column relations. It is also possible to perform tuple-based comparisons using subqueries, as long as the *degree* matches on both sides of the comparison.

```
-- Students who share the same major & minor as another student
SELECT *
FROM Student SO
WHERE (MajorCode, MinorCode) IN (
  SELECT MajorCode, MinorCode FROM Student SI
  WHERE SI.StudentID <> SO.StudentID
)
-- Anybody missing?
```

Nested Queries in the FROM clause

Since a SELECT statement returns a relational table, we can use a nested SELECT statement in the FROM clause of a query.

SELECT <column list>
FROM **(SELECT query) [AS] <alias>**
[WHERE <condition>]
[GROUP BY <attribute list>
[HAVING <group condition>]

Nested Queries in the FROM clause

Two requirements to consider when using a nested `SELECT` statement in the `FROM` clause of a query:

1. Nested `SELECT` must be enclosed in parentheses and *must have an alias*

2. All computed columns (aggregates, scalar functions, `CASE`, etc.) *must have aliases*

Subqueries in the FROM clause

```
-- Print duplicate records in Movies table

SELECT m1.title, m1.year, m1.gross, m1.imdb

FROM Movies m1,

(SELECT title, year, count(*) as num FROM Movies GROUP BY title, year) m2

WHERE m1.title = m2.title AND m1.year = m2.year AND m2.num > 1

ORDER BY m1.title, m1.year;
```

Joins with Subquery Results

SELECT a1, a2, …, an

FROM R

JOIN S ON jc

WHERE c;

R or S can be substituted with a relation resulting from a subquery.

#List all actors who co-starred with Tom Hanks

Select distinct s.sname

From StarsIn s

JOIN (select * from StarsIn where sname='Tom Hanks') as kb

ON kb.mid=s.mid AND kb.sname != s.sname;

Nested Queries in the FROM clause

```
#List all actors who co-starred with Harrison Ford

select distinct s.sname

From StarsIn s JOIN (

select * from StarsIn where sname='Harrison Ford'

) as hf

ON hf.mid=s.mid AND hf.sname != s.sname;
```

Nested Queries in the FROM clause

```
#List Movies both Harrison Ford and Sylvester Stallone appeared

SELECT m.*

FROM Movies m JOIN (

    select DISTINCT s.mid FROM StarsIn s JOIN (

        select * from StarsIn where sname='Harrison Ford'

    ) as hf ON hf.mid=s.mid WHERE s.sname='Sylvester Stallone'

) as hs ON m.mid=hs.mid;
```

Nesting Subqueries

Subqueries can be deeply nested, up to practical limits imposed by RDBMS implementations. Inner queries can reference outer columns, but cannot reference siblings.

```
-- Non-CENG departments with students who enrolled
-- prior to the date the department was established
SELECT *
FROM Department d1
WHERE EXISTS (
  SELECT StudentID FROM Student s
  WHERE s.MajorCode = d1.Code AND DateEnrolled < (
    SELECT DateEstablished FROM Department d2
    WHERE d2.Code = s.MajorCode
  ) AND s.MajorCode NOT IN (
    SELECT Code FROM Department d3
    WHERE d3.College = 'CENG'
  )
)
-- How would we list the corresponding students?
```

Nesting Subqueries

Subqueries can be deeply nested, up to practical limits imposed by RDBMS implementations. Inner queries can reference outer columns, but cannot reference siblings.

```
select name from (
select name, count(name) as countNum
from (select name
 from (select * from
   (select player_id as id
   from Event
   where type = "Goals scored" and
   count >= 3) as players
   natural join Player) as playersAndTeam, Team
 where playersAndTeam.team_id = Team.id) as
teamName
group by name having countNum >= 1) as NameNums;
```

Correlated Subqueries

The subquery examples seen thus far have been uncorrelated: subqueries that can be executed once for the entire outer SELECT.

It is also possible to refer to values in the outer query from within a subquery, causing the subquery to be re-evaluated multiple times during execution of the outer query.

This second form of subquery is called a **correlated subquery**

Correlated Subqueries Example

```
-- Check if there are movies with the same title but made earlier

SELECT m1.title, m1.year, m1.gross, m1.imdb

FROM Movies m1

WHERE m1.year < ANY (SELECT year FROM Movies WHERE title = m1.title)

ORDER BY m1.title, m1.year;
```

Correlated Subqueries Example

```
SELECT name, (SELECT COUNT(*) FROM StarsIn WHERE sname=name) as
num_movies

FROM Stars;
```

Can we convert this correlated subquery to a regular join?

```
SELECT name, num_movies

FROM Stars s

JOIN (SELECT sname, COUNT(*) as num_movies FROM StarsIn group by
sname) as n ON n.sname=name;
```

Subquery Recap

- Considerable flexibility, deep nesting allowed
- Can appear throughout SELECT statement (with some exceptions)
- Subquery variations:
  - Uncorrelated
  - Derived table in FROM clause
  - Correlated
    - No correlation permitted for subqueries that appear FROM clause
- Always consider performance (more on this later)

Set Operations in SQL

- ∪ : UNION
- ∩ : INTERSECT
- - : EXCEPT

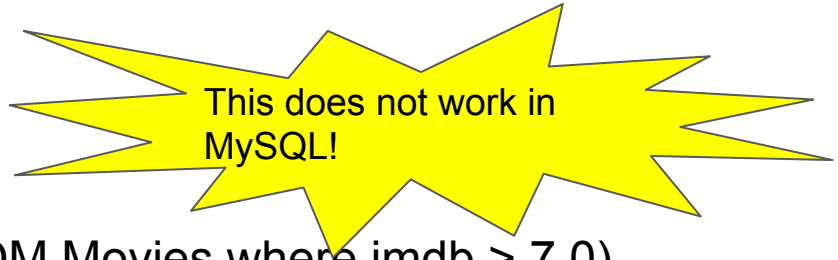
UNION

```
(SELECT title, year, gross, imdb FROM Movies where imdb > 7.0)

UNION

(SELECT title, year, gross, imdb FROM Movies where gross > 10000000);
```

INTERSECT

Unfortunately, MySQL does not support the INTERSECT operator.

However, you can simulate INTERSECT operator.




```
(SELECT title, year, gross, imdb FROM Movies where imdb > 7.0)
INTERSECT
(SELECT title, year, gross, imdb FROM Movies where gross > 10000000);
```

Simulating INTERSECT

simulate INTERSECT operator using DISTINCT operator and INNER JOIN

```
(SELECT title, year, gross, imdb FROM Movies where imdb > 7.0)
INTERSECT
(SELECT title, year, gross, imdb FROM Movies where gross > 10000000);
```

```
SELECT DISTINCT m1.title, m1.year, m1.gross, m1.imdb FROM Movies m1
JOIN
(SELECT title, year, gross, imdb FROM Movies where gross > 10000000) as m2
USING (title, year)
WHERE m1.imdb > 7.0;
```

Simulating INTERSECT

Simulate MySQL INTERSECT operator using IN operator and subquery

```
SELECT title, year, gross, imdb FROM Movies
WHERE imdb > 7.0 AND mid IN
(SELECT mid FROM Movies where gross > 10000000);
```

EXCEPT

Unfortunately, MySQL does not support the EXCEPT operator.

However, you can simulate EXCEPT operator using NOT IN operator, LEFT JOIN, and NOT EXISTS operator.

Simulating EXCEPT

```
SELECT title, year, gross, imdb FROM Movies

WHERE imdb > 7.0 AND mid NOT IN

(SELECT mid FROM Movies where gross > 10000000);
```

```
SELECT title, year, gross, imdb FROM Movies

WHERE imdb > 7.0 AND (title,year) NOT IN

(SELECT title,year FROM Movies where gross > 10000000);
```

Simulating EXCEPT

```
SELECT m1.title, m1.year, m1.gross, m1.imdb FROM Movies m1
LEFT JOIN
(SELECT mid FROM Movies where gross > 10000000) as m2 USING (mid)
WHERE imdb > 7.0 AND m2.mid is NULL;
```

Simulating EXCEPT

```
SELECT title, year, gross, imdb FROM Movies m

WHERE imdb > 7.0 AND NOT EXISTS

(SELECT title,year FROM Movies where mid=m.mid AND gross > 10000000);
```

Duplicates in Unions, Intersections, and Differences

- Basically, set operations produce SET of tuples, not BAG of tuples.
- This means that duplicates will be automatically eliminated.
- This is True with UNION in MySQL.
- When simulating INTERSECT and EXCEPT (MINUS) in MySQL, you have to eliminate duplicate tuples by yourself.

UNION

```
(SELECT title, year, gross, imdb FROM Movies where imdb > 7.0)

UNION ALL

(SELECT title, year, gross, imdb FROM Movies where gross > 10000000);
```

Control Row Count Returned by SELECT

ANSI SQL does not define a standard way to limit the number of rows returned by a `SELECT` query.

Each RDBMS has its own syntax. In MySQL, we use `LIMIT`:

```
SELECT *
FROM Student
ORDER BY LastName
LIMIT 2
```

Sorting (`ORDER BY`) is performed **before** `LIMIT`, so we see the first two students, based on A-Z ordering of last name.

MySQL LIMIT Variations

Single-argument LIMIT returns up to the specified number of rows:

```
LIMIT 5     -- top 5 rows in result set
```

Two-argument LIMIT returns up to the requested number of rows, after a given offset (zero indexed):

```
LIMIT 5,10  -- rows 6-15
```

Row Limiting in other RDBMSs

RDBMS	Syntax
MySQL & PostgreSQL	`SELECT * FROM table LIMIT 10`
Microsoft SQL Server & Access	`SELECT TOP 10 * FROM table`

Oracle	`SELECT * FROM`
	` (SELECT * FROM table)`
	`WHERE rownum <= 10`

IBM DB2	`SELECT * FROM table` `FETCH FIRST 10 ROWS ONLY`
Informix	`SELECT FIRST 10 * FROM table`

Using LIMIT in Subqueries

```
SELECT Code, Name, DateEstablished,
  DATEDIFF((SELECT DateEstablished
            FROM Department
            ORDER BY DateEstablished DESC
            LIMIT 1),
           DateEstablished) /  365 AS YearsOlderThanNewestDept
FROM Department
```

Alternative way to write this query, without LIMIT?