Introduction to Data Management

Subqueries

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Announcements

- Azure credits have been issued ($50)
  - Sent to @uw.edu emails
  - Post on Piazza if you have issues accepting it
  - Will use for HW3

Accept your Azure lab assignment

You have a pending lab assignment. Please accept your assignment to get started with your course.

Accept lab assignment >

This email is generated from an unmonitored alias; please do not reply. If you have questions, please submit a request.
Recap: Grouping

```
SELECT Product, SUM(quantity)
FROM Purchases
GROUP BY Product
HAVING SUM(quantity) > 20
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
<th>Month</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
<td>Jan</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
<td>Feb</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
<td>Feb</td>
</tr>
<tr>
<td>Banana</td>
<td>5</td>
<td>10</td>
<td>March</td>
</tr>
<tr>
<td>Apple</td>
<td>4</td>
<td>10</td>
<td>March</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>SUM(quantity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>40</td>
</tr>
<tr>
<td>Banana</td>
<td>60</td>
</tr>
</tbody>
</table>
Recap: Semantics

First evaluate the FROM clause
Next evaluate the WHERE clause
Group the attributes in the GROUPBY
Eliminate groups based on HAVING
Sort the results based on ORDER BY
Last evaluate the SELECT clause

FWGHOS™
**Recap - General form**

```
SELECT S
FROM R₁, ..., Rₙ
WHERE C₁
GROUP BY a₁, ..., aₚ
HAVING C₂
ORDER BY O
```

S, O = any attributes a₁, ..., aₚ and/or any aggregates, but no other attributes
C₁ = any condition on the attributes in R₁, ..., Rₙ
C₂ = any condition on the aggregate expressions and attributes a₁, ..., aₚ
Goals for Today

- We’ve completed our general form of a query
- Use SQL queries to assist other SQL queries
- Conclude our unit on SQL queries
  - After today you’ll have essentially all the building blocks of most all queries you can think of
Outline

▪ Subquery mechanics
  • SELECT
  • FROM
  • WHERE/HAVING

▪ Decorrelation and unnesting along the way
▪ The Witness Problem
A subquery is a SQL query nested inside a larger query.

A subquery may occur in:
- A SELECT clause
- A FROM clause
- A WHERE or HAVING clause

Rule of thumb:
Avoid nested queries when possible...
...but sometimes it’s impossible.
Subqueries in SELECT

- Must return a single value
  - A 1x1 relation – single row, single column
- Uses:
  - Compute an associated value
Subqueries in SELECT

- Must return a single value
  - A 1x1 relation – single row, single column
- Uses:
  - Compute an associated value

```sql
SELECT P.Name, (SELECT AVG(P1.Salary) FROM Payroll AS P1 WHERE P.Job = P1.Job) FROM Payroll AS P
```
Subqueries in SELECT

- Must return a single value
  - A 1x1 relation – single row, single column

- Uses:
  - Compute an associated value

```sql
SELECT P.Name, (SELECT AVG(P1.Salary)
FROM Payroll AS P1
WHERE P.Job = P1.Job)
FROM Payroll AS P
```

**Correlated subquery!**
Inner query refers to attributes from the outer query
Subqueries in SELECT

- Must return a single value
  - A 1x1 relation – single row, single column
- Uses:
  - Compute an associated value

**SELECT P.Name, (SELECT AVG(P1.Salary) FROM Payroll AS P1 WHERE P.Job = P1.Job) FROM Payroll AS P**

**Correlated subquery!**
Semantics are that the entire subquery is recomputed for each tuple
Subqueries in SELECT

For each person find the average salary of their job

```
SELECT  P.Name, (SELECT  AVG(P1.Salary)
               FROM    Payroll AS P1
               WHERE   P.Job = P1.Job)
FROM     Payroll AS P
```

```
SELECT  P1.Name, AVG(P2.Salary)
FROM    Payroll AS P1, Payroll AS P2
WHERE   P1.Job = P2.Job
GROUP BY P1.UserID, P1.Name
```
FWGHOS recall

```sql
SELECT P1.Name, AVG(P2.Salary)
FROM Payroll AS P1, Payroll AS P2
WHERE P1.Job = P2.Job
GROUP BY P1.UserID, P1.Name
```

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
</tbody>
</table>
FWGHOS recall

**SELECT** P1.Name, AVG(P2.Salary)
**FROM** Payroll AS P1, Payroll AS P2
**WHERE** P1.Job = P2.Job
**GROUP BY** P1.UserID, P1.Name

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
</tbody>
</table>
**Subqueries**

### SELECT

```
SELECT P1.Name, AVG(P2.Salary)
FROM Payroll AS P1, Payroll AS P2
WHERE P1.Job = P2.Job
GROUP BY P1.UserID, P1.Name
```

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
</tbody>
</table>
Subqueries in SELECT

For each person find the number of cars they drive

```sql
SELECT P.Name, (SELECT COUNT(R.Car) FROM Regist AS R WHERE P.UserID = R.UserID)
FROM Payroll AS P
```

Same? Discuss!

```sql
SELECT P.Name, COUNT(R.Car)
FROM Payroll AS P, Regist AS R
WHERE P.UserID = R.UserID
GROUP BY P.UserID, P.Name
```
Subqueries in SELECT

For each person find the number of cars they drive

```
SELECT P.Name, (SELECT COUNT(R.Car) FROM Regist AS R WHERE P.UserID = R.UserID)
FROM Payroll AS P
```

```
SELECT P.Name, COUNT(R.Car)
FROM Payroll AS P, Regist AS R
WHERE P.UserID = R.UserID
GROUP BY P.UserID, P.Name
```

0-count case not covered!
For each person find the number of cars they drive

```
SELECT P.Name, (SELECT COUNT(R.Car) FROM Regist AS R WHERE P.UserID = R.UserID)
FROM Payroll AS P
```

Still possible to decorrelate and unnest
Subqueries in SELECT

For each person find the number of cars they drive

```
SELECT P.Name, (SELECT COUNT(R.Car) 
                  FROM Regist AS R 
                  WHERE P.UserID = R.UserID) 
FROM Payroll AS P
```

Still possible to decorrelate and unnest

```
SELECT P.Name, COUNT(R.Car) 
FROM Payroll AS P LEFT OUTER JOIN 
    Regist AS R ON P.UserID = R.UserID 
GROUP BY P.UserID, P.Name
```
The Witnessing Problem

- Also known as argmax/argmin
- Ex: Return the person with the highest salary for each job type

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
</tbody>
</table>
The Witnessing Problem

- Also known as argmax/argmin
- Ex: Return the person with the highest salary for each job type

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
</tbody>
</table>

**SELECT** Name, MAX(Salary) **FROM** Payroll **GROUP** BY Job

Easy right?
The Witnessing Problem

- Also known as argmax/argmin
- Ex: Return the person with the highest salary for each job type

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
</tbody>
</table>

```
SELECT Name, MAX(Salary) 
FROM Payroll 
GROUP BY Job 
```
### The Witnessing Problem

**SELECT** Name, MAX(Salary) **FROM** Payroll **GROUP BY** Job
## The Witnessing Problem

Return the person with the highest salary for each job type

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
</tbody>
</table>

How do we witness the maxima for a group? **Discuss!**

Conceptual ideas are great
The Witnessing Problem

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Jack</td>
<td>TA</td>
<td>50000</td>
</tr>
<tr>
<td>345</td>
<td>Allison</td>
<td>TA</td>
<td>60000</td>
</tr>
<tr>
<td>567</td>
<td>Magda</td>
<td>Prof</td>
<td>90000</td>
</tr>
<tr>
<td>789</td>
<td>Dan</td>
<td>Prof</td>
<td>100000</td>
</tr>
</tbody>
</table>

Return the person with the highest salary for each job type

Main idea:  
we need to join the respective maxima to each row
Return the person with the highest salary for each job type

Main idea:
we need to join the respective maxima to each row
Subqueries in FROM

- Uses:
  - Solve subproblems that can be later joined/evaluated

```sql
SELECT P.Name, P.Salary
FROM Payroll AS P,
     (SELECT P1.Job AS Job,
         MAX(P1.Salary) AS Salary
     FROM Payroll AS P1
     GROUP BY P1.Job) AS Pmax
WHERE P.Job = Pmax.Job AND
     P.Salary = Pmax.Salary
```
Subqueries in FROM

- Equivalent to a WITH subquery

**WITH** MaxPay AS

```
(SELECT P1.Job AS Job,
    MAX(P1.Salary) AS Salary
FROM Payroll AS P1
GROUP BY P1.Job)
```

**SELECT** P.Name, P.Salary

**FROM** Payroll AS P, MaxPay AS MP

**WHERE** P.Job = MP.Job AND
    P.Salary = MP.Salary
Subqueries in WHERE/HAVING

- Can return a single value
- Uses:
  - Compare with another value

```sql
SELECT   P.Name, P.Salary
FROM     Payroll AS P
WHERE    P.Salary =
          (SELECT MAX(P1.Salary) AS Salary
           FROM    Payroll AS P1
           WHERE   P1.Job = P.Job)
```
Subqueries in WHERE/HAVING

- Can return a single value
- Uses:
  - Compare with another value

```
SELECT P.Name, P.Salary
FROM Payroll AS P
WHERE P.Salary =
    (SELECT MAX(P1.Salary) AS Salary
     FROM Payroll AS P1
     WHERE P1.Job = P.Job)
```

Correlated subquery alert!
Return the person with the highest salary for each job type

\[
\text{SELECT} \quad \text{P.Name,} \\
\quad \text{MAX(Pmax.Salary) as MaxSalary} \\
\text{FROM} \quad \text{Payroll AS P, Payroll AS Pmax} \\
\text{WHERE} \quad \text{P.Job = Pmax.Job} \\
\text{GROUP BY} \quad \text{Pmax.Job, P.Name, P.Salary} \\
\text{HAVING} \quad \text{P.Salary = MaxSalary}
\]
Witnessing Unnested

```sql
SELECT P1.Name, MAX(Pmax.Salary)
FROM Payroll AS P1, Payroll AS Pmax
WHERE P1.Job = Pmax.Job
GROUP BY Pmax.Job, P1.Salary, P1.Name
HAVING P1.Salary = MAX(Pmax.Salary)
```
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
    - (NOT) EXISTS, (NOT) IN, ANY, ALL

```sql
SELECT .......... WHERE EXISTS (subquery);
SELECT .......... WHERE NOT EXISTS (subquery);
SELECT .......... WHERE attr IN (subquery);
SELECT .......... WHERE attr NOT IN (subquery);
SELECT .......... WHERE const > ANY (subquery);
SELECT .......... WHERE const > ALL (subquery);
```
Subqueries in WHERE/HAVING

▪ Can return a relation

▪ Uses:
  • Use with an existential or universal quantifier
    • (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive some car made before 2017.
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
  - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive some car made before 2017.

```sql
SELECT P.Name
FROM Payroll AS P
WHERE EXISTS (SELECT * FROM Regist R
  WHERE R.UserID = P.UserID
  AND R.Year < 2017)
```

`EXISTS (subquery) returns true iff cardinality of subquery > 0`
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
  - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive some car made before 2017.

attr IN (subquery) returns true iff value of attr is contained in subquery

```
SELECT P.Name
FROM Payroll AS P
WHERE P.UserID IN (SELECT R.UserID
                    FROM Regist R
                    WHERE R.Year < 2017)
```
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
    - \((\text{NOT})\) EXISTS, \((\text{NOT})\) IN, ANY, ALL

Ex: Find all people who drive some car made before 2017.

```sql
SELECT P.Name
FROM Payroll AS P
WHERE P.UserID IN (SELECT R.UserID FROM Regist R WHERE R.Year < 2017)
```

attr IN (subquery) returns true iff value of attribute in relation

Decorrelated!
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
  - \((\text{NOT}) \text{ EXISTS}, (\text{NOT}) \text{ IN}, \text{ANY}, \text{ALL}\)

**Ex:** Find all people who drive some car made before 2017.

\[
\text{SELECT} \quad \text{P.Name} \\
\text{FROM} \quad \text{Payroll AS P} \\
\text{WHERE} \quad 2017 > \text{ANY} (\text{SELECT R.Year} \\
\text{FROM} \quad \text{Regist R} \\
\text{WHERE} \\
\text{P.UserID} = \text{R.UserID})
\]

const > ANY (sub) returns true iff const > value for at least one value in sub
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
  - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive some car made before 2017.

```
SELECT P.Name
FROM Payroll AS P
WHERE 2017 > ANY (SELECT R.Year
FROM Regist AS R
WHERE P.UserID = R.UserID)
```

Not supported in sqlite :(

Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
  - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive some car made before 2017.

```
SELECT DISTINCT P.Name
FROM Payroll AS P, Regist R
WHERE P.UserID = R.UserID AND
  R.Year < 2017
```

Unnesting existential quantifiers is easy!
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
    - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive some car made before 2017.

```
SELECT  DISTINCT P.Name
FROM    Payroll AS P, Regist R
WHERE   P.UserID = R.UserID AND
        R.Year < 2017
```

Unnesting existential quantifiers is easy!

Expert SQL style
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
    - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive only cars older than 2017.
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
    - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive only cars older than 2017.

Universal quantifier

Not easy :(
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
    - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive only cars older than 2017.

(VERIFY: Find all the other people, the ones who DO drive newer cars)

(SELECT R.UserID
     FROM Regist AS R
     WHERE R.Year >= 2017)

(VERIFY)
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
  - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive only cars older than 2017.

```
SELECT P.Name
FROM Payroll AS P
WHERE P.UserID NOT IN (SELECT R.UserID
FROM Regist AS R
WHERE R.Year >= 2017)
```

Find all the other people, the ones who DO drive newer cars.
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
  - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive only cars older than 2017.

```sql
SELECT P.Name
FROM Payroll AS P
WHERE NOT EXISTS (SELECT *
FROM Regist AS R
WHERE R.Year >= 2017
AND P.UserID = R.UserID)
```
Subqueries in WHERE/HAVING

- Can return a relation

- Uses:
  - Use with an existential or universal quantifier
  - (NOT) EXISTS, (NOT) IN, ANY, ALL

Ex: Find all people who drive only cars older than 2017.

```
SELECT P.Name
FROM Payroll AS P
WHERE 2017 > ALL (SELECT R.Year
                    FROM Regist AS R
                    WHERE P.UserID = R.UserID)
```

\( \text{const} > \text{ALL(sub)} \) returns true iff \( \text{const} > \text{value for all values in sub} \)
Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
  - \( \text{(NOT) EXISTS, (NOT) IN, ANY, ALL} \)

Ex: Find all people who drive only cars older than 2017.

```sql
SELECT P.Name
FROM Payroll AS P
WHERE 2017 > ALL (SELECT R.Year
                      FROM Regist AS R
                      WHERE P.UserID = R.UserID)
```

Not supported in sqlite :(

```sql
SELECT P.Name
FROM Payroll AS P
WHERE 2017 > ALL (SELECT R.Year
                      FROM Regist AS R
                      WHERE P.UserID = R.UserID)
```
Can we unnest the universal quantifier query?

First, a discussion on the concept of monotonicity....
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

Monotone queries can be similar to monotonically increasing functions when considering cardinalities of results.
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

```
SELECT P.Name, P.Car
FROM Payroll AS P, Regist AS R
WHERE P.UserID = R.UserID
```

Is this query monotone?
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

**Example:**

```sql
SELECT P.Name, P.Car 
FROM Payroll AS P, Regist AS R 
WHERE P.UserID = R.UserID 
```

Is this query monotone? **Yes!**
A **Monotonic** query is one that obeys the following rule where \( I \) and \( J \) are data instances and \( q \) is a query:

\[
I \subseteq J \rightarrow q(I) \subseteq q(J)
\]

That is, for any superset of \( I \), the query over that superset must contain at least the query results of \( I \).

**SELECT** P.Name, P.Car  
**FROM** Payroll AS P, Regist AS R  
**WHERE** P.UserID = R.UserID

Is this query monotone? **Yes!**
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

```
SELECT P.Name
FROM Payroll AS P
WHERE P.Salary >= ALL (SELECT Salary FROM Payroll)
```

Is this query monotone?
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

**SELECT**  
P.Name  
**FROM**  
Payroll AS P  
**WHERE**  
P.Salary >= ALL (**SELECT**  
Salary  
**FROM**  
Payroll)

Is this query monotone? **No!**
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

**SELECT** P.Name  
**FROM** Payroll AS P  
**WHERE** P.Salary >= ALL (SELECT Salary FROM Payroll)

Is this query monotone? **No!**
A Monotonic query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

**SELECT**  
P.Job, COUNT(*)  
**FROM**  
Payroll AS P  
**GROUP BY**  
P.Job

Is this query monotone?
A Monotonic query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

```
SELECT P.Job, COUNT(*)
FROM Payroll AS P
GROUP BY P.Job
```

Is this query monotone? No!
A **Monotonic** query is one that obeys the following rule where $I$ and $J$ are data instances and $q$ is a query:

$$I \subseteq J \rightarrow q(I) \subseteq q(J)$$

That is for any superset of $I$, the query over that superset must contain at least the query results of $I$.

```sql
SELECT P.Job, COUNT(*)
FROM Payroll AS P
GROUP BY P.Job
```

Is this query monotone? **No!**
Monotonicity

Theorem:
If Q is a SELECT–FROM–WHERE query that does not have subqueries or aggregates, then it is monotone.
Monotonicity

Theorem:
If Q is a SELECT–FROM–WHERE query that does not have subqueries or aggregates, then it is monotone.

Proof:
We use nested loop semantics. If we insert a tuple in relation R, this will not remove any tuples from the answer.

```
SELECT a1, a2, ..., ak
FROM R1 AS x1, R2 AS x2, ..., Rn AS xn
WHERE Conditions
```

for x1 in R1 do
  for x2 in R2 do
    ...
    for xn in Rn do
      if Conditions
        output (a1,...,ak)
Theorem:
The query “Find all people who drive only cars older than 2017” is not monotone.

Proof:
We use example. For user 123 who previously only drove a car made in 2009, we add another car made in 2018. Now user 123 does not appear in the results. Thus, the query is not monotone.
Monotonicity

Theorem:
The query “Find all people who drive only cars older than 2017” is not monotone.

Proof:
We use example. For user 123 who previously only drove a car made in 2009, we add another car made in 2018. Now user 123 does not appear in the results. Thus, the query is not monotone.

If a query is not monotonic, then we can’t write it as a SELECT-FROM-WHERE query without subqueries.
Queries That Cannot Be S-F-W

- Queries with universal quantifiers or negation

```sql
SELECT P.Name
FROM Payroll AS P
WHERE P.UserID NOT IN (SELECT R.UserID
                        FROM Regist AS R
                        WHERE R.Year < 2017)
```

```sql
SELECT P.Name
FROM Payroll AS P
WHERE P.Salary >= ALL (SELECT Salary
                        FROM Payroll)
```
Bonus: Set Operations

- SQL mimics set theory in many ways
  - Bag = duplicates allowed
  - **UNION (ALL)** □ set union (bag union)
  - **INTERSECT (ALL)** □ set intersection (bag intersection)
  - **EXCEPT (ALL)** □ set difference (bag difference)

- SQL Server Management Studio 2017
  - INTERSECT ALL not supported
  - EXCEPT ALL not supported
Set Operations

- SQL set-like operators basically slap two queries together (not really a subquery...)

\[
(\text{SELECT } * \text{ FROM T1}) \cup (\text{SELECT } * \text{ FROM T2})
\]

Is this a subquery?
Subqueries let us express some problems more easily

Many subqueries can be unnested

Some cannot be (think non-monotonic queries – universal quantifiers, negation, aggregates)

SQL set operations behave much like set theory