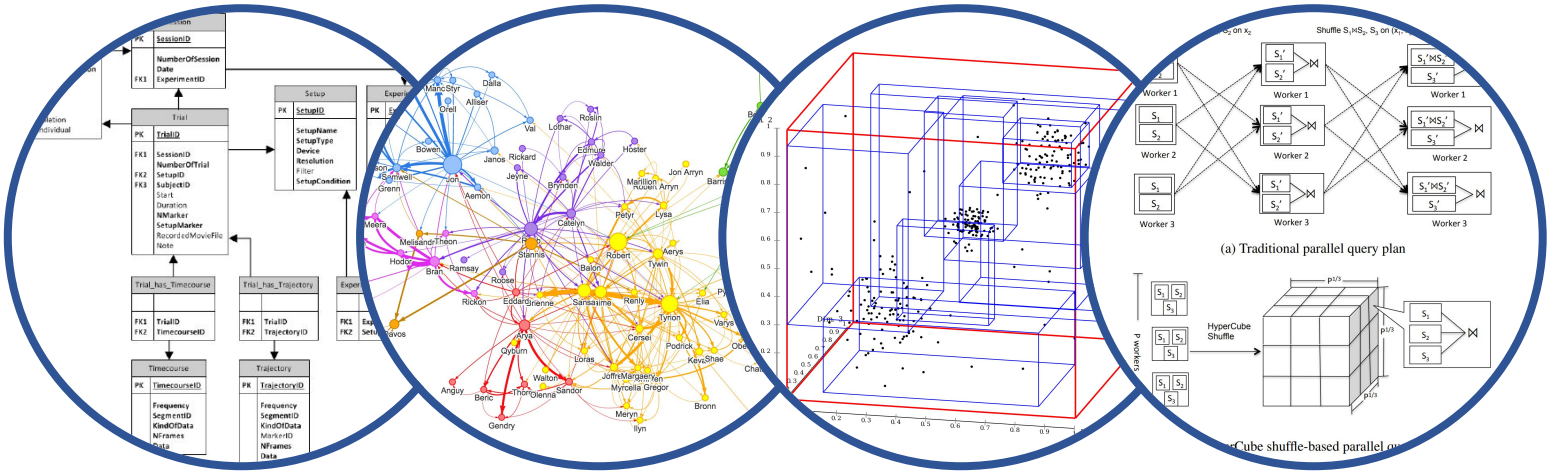



# Introduction to Data Management

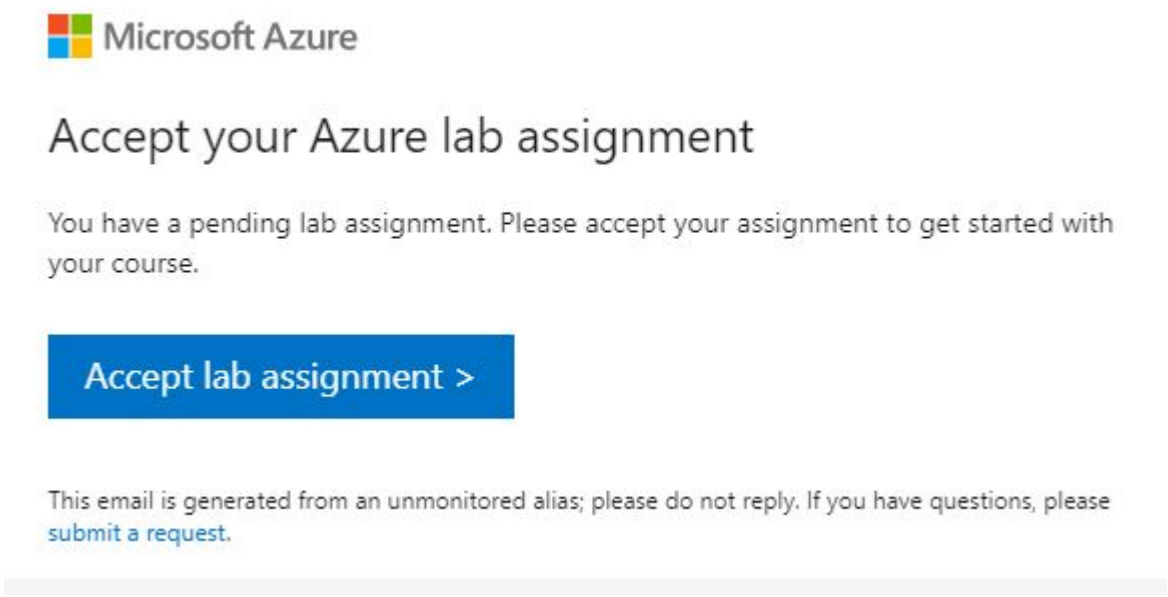
## Subqueries

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



# Recap: Grouping

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

| Product | Price | Quantity | Month |
|---|---|---|---|
| Bagel | 3 | 20 | Jan |
| Bagel | 1.50 | 20 | Feb |
| Banana | 0.5 | 50 | Feb |
| Banana | 5 | 10 | March |
| Apple | 4 | 10 | March |

| Product | SUM(quantity) |
|---|---|
| Bagel | 40 |
| Banana | 60 |

# 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, …, ak
   HAVING C2
ORDER BY O
```

S, O = any attributes $a_1$, ..., $a_k$ and/or any aggregates, but no other attributes

C1 = any condition on the attributes in $R_1$, ..., $R_n$

C2 = any condition on the aggregate expressions and attributes $a_1$, ..., $a_k$

# 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

# Nested Queries

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

```
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

```
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
```

Same (decorrelated and unnested)

```
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

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

P1 P2

| UserID | Name | Job | Salary | UserID | Name | Job | Salary |
|---|---|---|---|---|---|---|---|
| 123 | Jack | TA | 50000 | 123 | Jack | TA | 50000 |
| 123 | Jack | TA | 50000 | 345 | Allison | TA | 60000 |
| 345 | Allison | TA | 60000 | 345 | Allison | TA | 60000 |
| 345 | Allison | TA | 60000 | 123 | Jack | TA | 50000 |
| 567 | Magda | Prof | 90000 | 567 | Magda | Prof | 90000 |
| 567 | Magda | Prof | 90000 | 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 | 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 | 567 | Magda | Prof | 90000 |

# 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
```

P1 P2

| UserID | Name | Job | Salary | UserID | Name | Job | Salary |
|---|---|---|---|---|---|---|---|
| 123 | Jack | TA | 50000 | 123 | Jack | TA | 50000 |
| 123 | Jack | TA | 50000 | 345 | Allison | TA | 60000 |
| 345 | Allison | TA | 60000 | 345 | Allison | TA | 60000 |
| 345 | Allison | TA | 60000 | 123 | Jack | TA | 50000 |
| 567 | Magda | Prof | 90000 | 567 | Magda | Prof | 90000 |
| 567 | Magda | Prof | 90000 | 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 | 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 | 567 | Magda | Prof | 90000 |

# FWGHOS recall

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

| p1.Name | AVG |
|---|---|
| Jack | 55000 |
| Allison | 55000 |
| Magda | 95000 |
| Dan | 95000 |

P1 P2

| UserID | Name | Job | Salary | UserID | Name | Job | Salary |
|---|---|---|---|---|---|---|---|
| 123 | Jack | TA | 50000 | 123 | Jack | TA | 50000 |
| 123 | Jack | TA | 50000 | 345 | Allison | TA | 60000 |
| 345 | Allison | TA | 60000 | 345 | Allison | TA | 60000 |
| 345 | Allison | TA | 60000 | 123 | Jack | TA | 50000 |
| 567 | Magda | Prof | 90000 | 567 | Magda | Prof | 90000 |
| 567 | Magda | Prof | 90000 | 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 | 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 | 567 | Magda | Prof | 90000 |

# 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
```

Same? **Discuss!**

```
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
```

0-count case not covered!

```
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
```

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

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

# The Witnessing Problem

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

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

Easy right?

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

# The Witnessing Problem

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


| UserID | Salary |
|---|---|
| 123 | 50000 |
| 345 | 60000 |
| 567 | 90000 |
| 789 | 100000 |

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

# The Witnessing Problem

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

| Name | MAX(Salary) |
|---|---|
| ??? | 60000 |
| ??? | 100000 |

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

# The Witnessing Problem

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

Return the person with the highest salary for each job type

How do we witness the maxima for a group?

**Discuss!**

Conceptual ideas are great

# The Witnessing Problem

| UserID | Name | Job | Salary |
|---|---|---|---|
| 123 | Jack | TA | 50000 |
| 345 | Allison | TA | 60000 |
| 567 | Magda | Prof | 90000 |
| 789 | Dan | Prof | 100000 |

Return the person with the highest salary for each job type

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

# The Witnessing Problem

| UserID | Name | Job | Salary | maxima |
|---|---|---|---|---|
| 123 | Jack | TA | 50000 | 60000 |
| 345 | Allison | TA | 60000 | 60000 |
| 567 | Magda | Prof | 90000 | 100000 |
| 789 | Dan | Prof | 100000 | 100000 |

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

```
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

```
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!**

# Witnessing Unnested

| UserID | Name | Job | Salary | maxima |
|---|---|---|---|---|
| 123 | Jack | TA | 50000 | 60000 |
| 345 | Allison | TA | 60000 | 60000 |
| 567 | Magda | Prof | 90000 | 100000 |
| 789 | Dan | Prof | 100000 | 100000 |

Return the person with the highest salary for each job type

```
SELECT P.Name,
       MAX(Pmax.Salary) as MaxSalary
  FROM Payroll AS P, Payroll AS Pmax
 WHERE P.Job = Pmax.Job
GROUP BY Pmax.Job, P.Name, P.Salary
HAVING P.Salary = MaxSalary
```

# Witnessing Unnested

```
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)
```

P1 Pmax

| UserID | Name | Job | Salary | UserID | Name | Job | Salary |
|---|---|---|---|---|---|---|---|
| 123 | Jack | TA | 50000 | 123 | Jack | TA | 50000 |
| 123 | Jack | TA | 50000 | 345 | Allison | TA | 60000 |
| 345 | Allison | TA | 60000 | 345 | Allison | TA | 60000 |
| 345 | Allison | TA | 60000 | 123 | Jack | TA | 50000 |
| 567 | Magda | Prof | 90000 | 567 | Magda | Prof | 90000 |
| 567 | Magda | Prof | 90000 | 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 | 789 | Dan | Prof | 100000 |
| 789 | Dan | Prof | 100000 | 567 | Magda | Prof | 90000 |

# Subqueries in WHERE/HAVING

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

```
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.

**Existential quantifier**

# 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.

EXISTS (subquery) returns true iff cardinality of subquery > 0

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

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
    - (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 at

Decorrelated!

```
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
    - (NOT) EXISTS, (NOT) IN, ANY, ALL

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

const > ANY (sub) returns true iff
const > value for at least one value in sub

```
SELECT P.Name
  FROM Payroll AS P
 WHERE 2017 > ANY (SELECT R.Year
                   FROM Regist R
                   WHERE
                     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 some car made before 2017.

```
SELECT P.Name
  FROM Payroll A P
 WHERE 2017 > ANY (SELECT R.Year
                     FROM Regist 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.

Unnesting existential quantifiers is easy!

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

Unnesting existential quantifiers is easy!

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


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

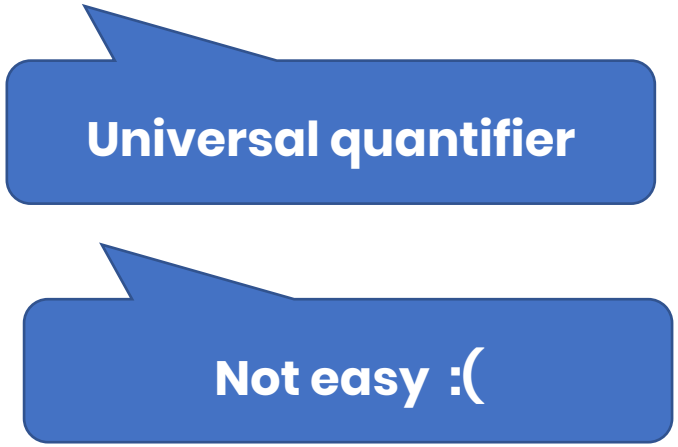
Ex: Find all people who drive only cars older than 2017.

**Universal quantifier**

# 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 o r than 2017.

Find all the other people, the ones who DO drive newer cars

```
(SELECT R.UserID
   FROM Regist AS R
 WHERE R.Year >= 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 o r than 2017.

Find all the other people, the ones who DO drive newer cars

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

```
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.

const > ALL(sub) returns true iff
const > value for all values in sub

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

Not supported in sqlite :(

# Unnesting

Can we unnest the universal quantifier query?

First, a discussion on the concept of monoticity....

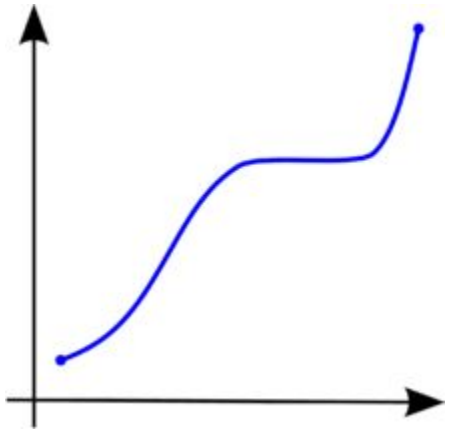
# Monotonicity

**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.



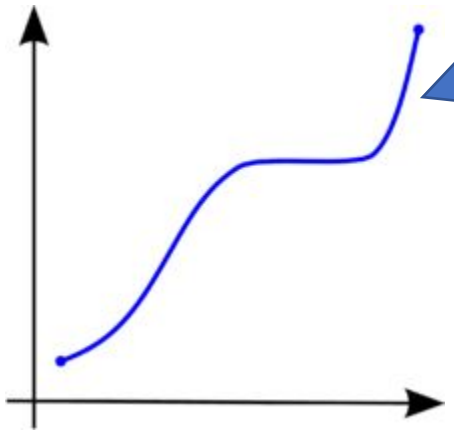
# Monotonicity

**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.



Monotone queries can be similar to monotonically increasing functions when considering cardinalities of results

# Monotonicity

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

Is this query monotone?

# Monotonicity

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

Is this query monotone? Yes!

# Monotonicity

**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.

I can't add tuples to Payroll or Regist that would "remove" a previous result

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

Is this query monotone? Yes!

# Monotonicity

**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?

# Monotonicity

**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!

# Monotonicity

**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.

I can add a tuple to Payroll that has a higher salary value than any other

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

Is this query monotone? No!

# Monotonicity

**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?

# Monotonicity

**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!

# Monotonicity

**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 tho
at least the query results of I.

Aggregates generally are sensitive to any new tuples since the aggregate value will change

```
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)
```

# 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.

# Monotonicity

Theorem:

The query "Find all people who drive only cars older than 2017" is not monotonic.

Proof:

We use a previously only add another row user 123 does not appear. Thus, the query is not monotonic.

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

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

```
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…)

# Takeaways

- 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