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
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 – The Witnessing Problem

- A question pattern that asks for data associated with a maxima of some value
  - “Self join” on values you find the maxima for
  - GROUP BY to deduplicate one side of the join
  - HAVING to compare values with respective maxima
Recap: The Witnessing Problem

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

<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>
Goals for Today

- Conclude our unit on SQL queries
  - After today you’ll have essentially all the building blocks of most all queries you can think of
- Use SQL queries to assist other SQL queries
Subqueries

- Subquery mechanics
  - SELECT
  - FROM
  - WHERE/HAVING

- Decorrelation and unnesting along the way
- Witnessing revisited
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
▪ Uses:
  • Compute an associated value
Subqueries in SELECT

- Must return a single value
- 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
- 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
```
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
```

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

```sql
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

```sql
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
```
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

```sql
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!
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.
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

```sql
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
  - \((\text{NOT}) \ EXIST \), \((\text{NOT}) \ IN\), \(\text{ANY}\), \(\text{ALL}\)

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

```
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 this subquery

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.

```
const > \text{ANY} (\text{sub}) \quad \text{returns true iff} \quad const > \text{value for at least one value in sub}
```

```
\begin{verbatim}
SELECT P.Name
FROM Payroll AS P
WHERE 2017 > \text{ANY} (SELECT R.Year
FROM Regist R
WHERE P.UserID = R.UserID)
\end{verbatim}
```
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 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.

```sql
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 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.
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 only cars older than 2017.

\[
\text{(SELECT R.UserID FROM Regist AS R WHERE R.Year} \geq \text{2017)}
\]

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 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

```
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.

```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.

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

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

```
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.
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.
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.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 $I$.

```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**  
\[ P.\text{Name} \]

**FROM**  
\[ \text{Payroll AS } P \]

**WHERE**  
\[ P.\text{Salary} \geq \text{ALL} \left( \text{SELECT } \text{Salary} \right) \]

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**  
\textbf{P.Job, COUNT(*)}  
**FROM**  
Payroll AS P  
**GROUP BY**  
\textbf{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.

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

Is this query monotone? **No!**

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

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

```
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.
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 Must Be Nested

- 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)** $\sqcup$ set union (bag union)
- **INTERSECT (ALL)** $\sqcap$ set intersection (bag intersection)
- **EXCEPT (ALL)** $\setminus$ 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...)

```
(SELECT * FROM T1)
UNION
(SELECT * 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