

CSE 344: Intro to Data Management Subqueries and Relational Algebra

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

The WITH Clause

The WITH Clause

- Define temporary tables
- Use them in a query

The WITH Clause

What is the average salary of car drivers?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
SELECT avg(p.salary)
FROM payroll p, regist r
WHERE p.user_id = r.user_id;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
SELECT avg(p.salary)
FROM payroll p, regist r
WHERE p.user_id = r.user_id;
```

```
SELECT p.salary
FROM ...;
```



name	salary
Jack	50000
Magda	90000
Magda	90000

Duplicate!

Wrong!

avg(...)

76667



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
SELECT avg(DISTINCT p.salary)
FROM payroll p, regist r
WHERE p.user_id = r.user_id;
```

Does DISTINCT fix it?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
SELECT avg(DISTINCT p.salary)  
FROM payroll p, regist r  
WHERE p.user_id = r.user_id;
```

```
SELECT DISTINCT p.salary  
FROM ...;
```



salary
50000
90000



avg(...)
70000

Does DISTINCT fix it?

Correct answer:
63333

Wrong!

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	50000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
345	Tesla
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

We will solve this query by computing a temporary table using the WITH clause

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	50000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
345	Tesla
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.user_id, p.salary
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	50000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
345	Tesla
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.user_id, p.salary
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	50000
567	Magda	Prof	90000
789	Dan	Prof	100000


regist

user_id	car
123	Charger
345	Tesla
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.user_id, p.salary
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```



user_id	salary
123	50000
345	50000
567	90000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	50000
567	Magda	Prof	90000
789	Dan	Prof	100000


regist

user_id	car
123	Charger
345	Tesla
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.user_id, p.salary
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```



user_id	salary
123	50000
345	50000
567	90000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	50000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
345	Tesla
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.user_id, p.salary
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```

cardrivers

user_id	salary
123	50000
345	50000
567	90000

avg(...)
63333

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	50000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
345	Tesla
567	Civic
567	Pinto

The WITH Clause

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.user_id, p.salary
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```

cardrivers

user_id	salary
123	50000
345	50000
567	90000

avg(...)
63333

Correct

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	50000
567	Magda	Prof	90000
789	Dan	Prof	100000


regist

user_id	car
123	Charger
345	Tesla
567	Civic
567	Pinto

The WITH Clause

General form:

```
WITH tbl1 as (SELECT ...),  
      tbl2 as (SELECT ...),  
      . . .  
SELECT ...  
FROM .....  
WHERE ...  
...
```



Here we may use
tbl1, tbl2, ...

- A *WITH* construct is a simple form of a subquery
- We could also write the subquery in the *FROM* clause, but it is less readable

Views

- A view is a table that is defined using a SQL query
- The table content is computed only when used

- The view becomes part of the persistent database

- A view is a table that is defined using a SQL query
- The table content is computed only when used

Different from WITH

Same as WITH

- The view definition becomes part of the persistent database

Views

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Views

Persistent database

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Views

```
CREATE VIEW cardrivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Views

```
CREATE VIEW cardrivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

Persistent database

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

cardrivers

```
SELECT ...
FROM ...
```


Views

```
CREATE VIEW cardrivers AS
  SELECT DISTINCT p.*
  FROM payroll p, regist r
  WHERE p.user_id=r.user_id;
```

Persistent database

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

cardrivers

```
SELECT ...
FROM ...
```

Views

```
CREATE VIEW cardrivers AS
  SELECT DISTINCT p.*
  FROM payroll p, regist r
  WHERE p.user_id=r.user_id;
```

```
SELECT *
FROM cardrivers;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

cardrivers

```
SELECT ...
FROM ...
```

Views

```
CREATE VIEW cardrivers AS
  SELECT DISTINCT p.*
  FROM payroll p, regist r
  WHERE p.user_id=r.user_id;
```

```
SELECT *
FROM cardrivers;
```



user_id	name	job	salary
123	Jack	TA	50000
567	Magda	Prof	90000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

cardrivers

```
SELECT ...
FROM ...
```

Views

```
CREATE VIEW cardrivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

```
SELECT *
FROM cardrivers;
```



The view is computed at query time, with fresh data. Let's see that...

user_id	name	job	salary
123	Jack	TA	50000
567	Magda	Prof	90000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

cardrivers

```
SELECT ...
FROM ...
```

Views

```
CREATE VIEW cardrivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

```
INSERT INTO regist
VALUES (345, 'Tesla');
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

cardrivers

```
SELECT ...
FROM ...
```

Views

```
CREATE VIEW cardrivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

```
INSERT INTO regist
VALUES (345, 'Tesla');
```



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

cardrivers

```
SELECT ...
FROM ...
```

Views

```
CREATE VIEW cardrivers AS
  SELECT DISTINCT p.*
  FROM payroll p, regist r
  WHERE p.user_id=r.user_id;
```

```
SELECT *
FROM cardrivers;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

cardrivers

```
SELECT ...
FROM ...
```

Views

```
CREATE VIEW cardrivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

```
SELECT *
FROM cardrivers;
```



user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

cardrivers

```
SELECT ...
FROM ...
```


Views

```
CREATE VIEW cardrivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

```
SELECT *
FROM cardrivers;
```



user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000

Uses updated data

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

cardrivers

```
SELECT ...
FROM ...
```

Views

Virtual View: means computed at query time

All DBMS

Materialized View: computed at definition time

Not in Sqlite

What are their pros and cons?

Views

Virtual View: means computed at query time

All DBMS

Materialized View: computed at definition time

Not in Sqlite

Advantage of virtual views:

- Always contains fresh data
- Query-time optimization
(in class)

Disadvantages:

- Need to re-compute
every time it is queried

Views

Virtual View: means computed at query time

All DBMS

Materialized View: computed at definition time

Not in Sqlite

Advantage of virtual views:

- Always contains fresh data
- Query-time optimization (in class)

Disadvantages:

- Need to re-compute every time it is queried

Advantage of materialize views:

- Computed only once

Disadvantages:

- Need to be updated when the input data is updated
- Incremental View Maintenance (IVM)

Views

Virtual View: means computed at query time

All DBMS

Materialized View: computed at definition time

Not in Sqlite

Advantage of virtual views:

- Always contains fresh data
- Query-time optimization (in class)

Disadvantages:

- Need to re-compute every time it is queried

Advantage of materialize views:

- Computed only once

Disadvantages:

- Need to be updated when the input data is updated
- Incremental View Maintenance (IVM)

We don't discuss materialized views in this class

Materializing Query Outputs

Materializing Query Outputs

```
CREATE TABLE drivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Materializing Query Outputs

```
CREATE TABLE drivers AS  
SELECT DISTINCT p.*  
FROM payroll p, regist r  
WHERE p.user_id=r.user_id;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Materializing Query Outputs

```
CREATE TABLE drivers AS
  SELECT DISTINCT p.*
  FROM payroll p, regist r
  WHERE p.user_id=r.user_id;
```

Only the attributes
from payroll

payroll

user_id	name	job	salary	regist	
123	Jack	TA	50000	user_id	car
345	Allison	TA	60000	123	Charger
567	Magda	Prof	90000	567	Civic
789	Dan	Prof	100000	567	Pinto

Materializing Query Outputs

```
CREATE TABLE drivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

payroll

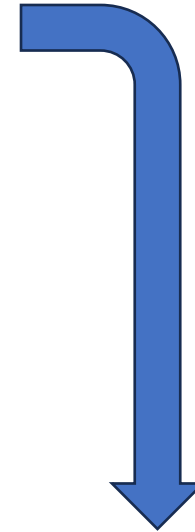
user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

drivers

user_id	...	salary
123	...	50000
567	...	90000



Materializing Query Outputs

```
CREATE TABLE drivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

Persistent database

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

drivers

user_id	...	salary
123	...	50000
567	...	90000

Materializing Query Outputs

```
CREATE TABLE drivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

How does this differ from a materialized view?

Persistent database

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

drivers

user_id	...	salary
123	...	50000
567	...	90000

Materializing Query Outputs

```
CREATE TABLE drivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

How does this differ from a materialized view?

System will not update automatically

Persistent database

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

drivers

user_id	...	salary
123	...	50000
567	...	90000

Materializing Query Outputs

```
CREATE TABLE drivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

```
SELECT *
FROM drivers;
```



user_id	...	salary
123	...	50000
567	...	90000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

drivers

user_id	...	salary
123	...	50000
567	...	90000

Materializing Query Outputs

```
CREATE TABLE drivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

```
INSERT INTO regist
VALUES (345, 'Tesla');
```



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

drivers

user_id	...	salary
123	...	50000
567	...	90000

Materializing Query Outputs

```
CREATE TABLE drivers AS
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id=r.user_id;
```

```
SELECT *
FROM drivers;
```



user_id	...	salary
123	...	50000
567	...	90000

Uses stale data

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

drivers

user_id	...	salary
123	...	50000
567	...	90000

More about GROUP BY

More GROUP BY

- So far, we grouped only by attributes

- We can also group by expressions!

More GROUP BY

Find the total revenue per company and decade

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

More GROUP BY

Find the total revenue per company and decade

We want this: 

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999	Acme	250000
1990	1999	IBM	...
2000	2009	Acme	...
...			
2010	2019	IBM	450000
...			

More GROUP BY

Find the total revenue per company and decade

```
SELECT year/10*10 AS Start, year/10*10 + 9 AS End,  
        name, sum(Revenue)  
FROM Company  
GROUP BY year/10*10, year/10*10 + 9, name;
```

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999	Acme	250000
1990	1999	IBM	...
2000	2009	Acme	...
...			
2010	2019	IBM	450000
...			

More GROUP BY

Find the total revenue per company and decade

```
SELECT year/10*10 AS Start, year/10*10 + 9 AS End,  
        name, sum(Revenue)  
FROM Company  
GROUP BY year/10*10, year/10*10 + 9, name;
```

Integer division
or use cast(...)

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999	Acme	250000
1990	1999	IBM	...
2000	2009	Acme	...
...			
2010	2019	IBM	450000
...			

More GROUP BY

Find the total revenue per company and decade

```
SELECT year/10*10 AS Start, year/10*10 + 9 AS End,  
        name, sum(Revenue)  
FROM Company  
GROUP BY year/10*10, year/10*10 + 9, name;
```

Beginning of decade

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999	Acme	250000
1990	1999	IBM	...
2000	2009	Acme	...
...			
2010	2019	IBM	450000
...			

More GROUP BY

Find the total revenue per company and decade

```
SELECT year/10*10 AS Start, year/10*10 + 9 AS End,  
        name, sum(Revenue)  
FROM Company  
GROUP BY year/10*10, year/10*10 + 9, name;
```

Beginning of decade

End of decade

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999	Acme	250000
1990	1999	IBM	...
2000	2009	Acme	...
...			
2010	2019	IBM	450000
...			

More GROUP BY

Find the total revenue per company and decade

Needs to occur
in GROUP BY

```
SELECT year/10*10 AS Start, year/10*10 + 9 AS End,  
        name, sum(Revenue)  
FROM Company  
GROUP BY year/10*10, year/10*10 + 9, name;
```

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999	Acme	250000
1990	1999	IBM	...
2000	2009	Acme	...
...			
2010	2019	IBM	450000
...			

More GROUP BY

Find the total revenue per company and decade

```
SELECT year/10*10 AS Start, year/10*10 + 9 AS End,  
        name, sum(Revenue)  
FROM Company  
GROUP BY Start, End, name;
```

Sqlite allows
this. Nice 😊

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999	Acme	250000
1990	1999	IBM	...
2000	2009	Acme	...
...			
2010	2019	IBM	450000
...			

More GROUP BY

Find the total revenue in a sliding window of 10 years

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

More GROUP BY

Find the total revenue in a sliding window of 10 years

We want this: 

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999		
1991	2000		
1992	2001		
...			
2013	2022		
...			

More GROUP BY

Find the total revenue in a sliding window of 10 years

```
SELECT X.year, X.year+9, X.name, sum(Y.Revenue)
FROM Company X, Company Y
WHERE X.name = Y.name
    and X.year <= Y.year and Y.year < X.year+10
GROUP BY X.year, X.year+9, X.name
ORDER BY X.year;
```

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999		
1991	2000		
1992	2001		
...			
2013	2022		
...			

More GROUP BY

Find the total revenue in a sliding window of 10 years

```
SELECT X.year, X.year+9, X.name, sum(Y.Revenue)
FROM Company X, Company Y
WHERE X.name = Y.name
        and X.year <= Y.year and Y.year < X.year+10
GROUP BY X.year, X.year+9, X.name
ORDER BY X.year;
```

Gaps if
year
not in
database

Company

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999		
1991	2000		
1992	2001		
...			
2013	2022		
...			

More GROUP BY

Find the total revenue in a sliding window of 10 years

```
SELECT X.year, X.year+9, X.name, sum(Y.Revenue)
FROM Company X, Company Y
WHERE X.name = Y.name
    and X.year <= Y.year and Y.year < X.year+10
GROUP BY X.year, X.year+9, X.name
ORDER BY X.year;
```

Gaps if
year
not in
database

Company

Postgres:
generate_series(1999,2024)

name	Revenue	year
Acme	100000	1995
IBM	200000	2012
Apple	300000	2012
IBM	250000	2019
...		

Start	End	name	Total
1990	1999		
1991	2000		
1992	2001		
...			
2013	2022		
...			

- GROUP-BY is versatile and powerful
- Optimizers can often find every efficient plans
- SQL also has “windows function”: very complex
 - Will not discuss in class

The Witness

The Witness

- SQL provides min/max, but not argmin/argmax
- Record that achieves min/max: [The Witness](#)
- Several ways to compute it:
 - WITH
 - Self-join and HAVING

The Witnessing Problem

Find the person with highest salary for each job

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

Desired answer:

job	name	salary
TA	Allison	60000
Prof	Dan	100000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT job, MAX(salary)
FROM payroll
GROUP BY job
```

job	salary
TA	60000
Prof	100000

Finding max is easy.

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT job, MAX(salary)
FROM payroll
GROUP BY job
```

job	salary
TA	60000
Prof	100000

Finding max is easy.

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

But we want argmax.
How do we find
the witness?

The Witnessing Problem

Find the person with highest salary for each job

Solution 1: Using WITH
Solution 2: Using HAVING

Plan:

1. Compute the max(salary) for each job
2. Join back with payroll on job
3. Return the users where salary = max(salary)

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
WITH jobSal AS
  (SELECT job, max(salary) AS M
   FROM payroll
   GROUP BY job)
SELECT j.job, p.name, p.salary
FROM jobSal j, payroll p
WHERE j.job = p.job
      and j.M = p.salary;
```


payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
WITH jobSal AS
  (SELECT job, max(salary) AS M
   FROM payroll
   GROUP BY job)
SELECT j.job, p.name, p.salary
FROM jobSal j, payroll p
WHERE j.job = p.job
      and j.M = p.salary;
```



job	M
TA	60000
Prof	100000

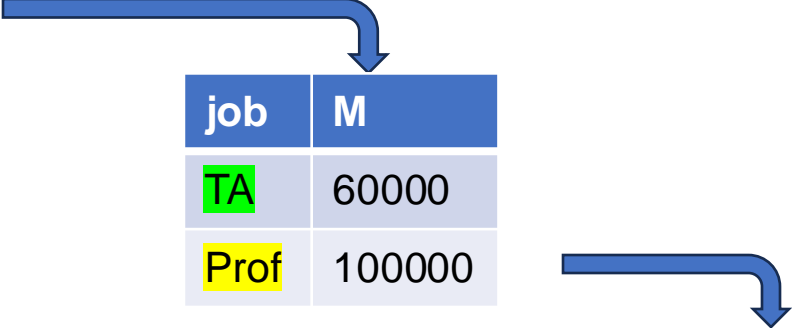
payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
WITH jobSal AS
  (SELECT job, max(salary) AS M
   FROM payroll
   GROUP BY job)
SELECT j.job, p.name, p.salary
FROM jobSal j, payroll p
WHERE j.job = p.job
      and j.M = p.salary;
```



job	M
TA	60000
Prof	100000

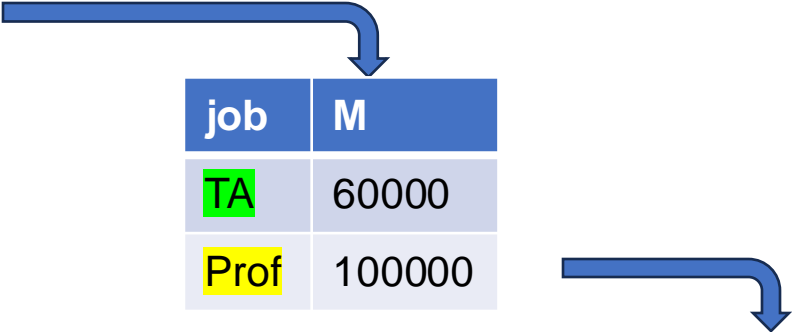
payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
WITH jobSal AS
  (SELECT job, max(salary) AS M
   FROM payroll
   GROUP BY job)
SELECT j.job, p.name, p.salary
FROM jobSal j, payroll p
WHERE j.job = p.job
      and j.M = p.salary;
```



job	M
TA	60000
Prof	100000

payroll

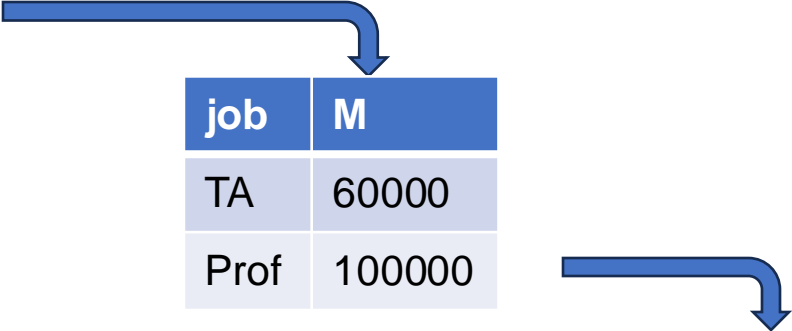
job	M	user_id	name	job	salary
TA	60000	123	Jack	TA	50000
TA	60000	345	Allison	TA	60000
Prof	100000	567	Magda	Prof	90000
Prof	100000	789	Dan	Prof	100000

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
WITH jobSal AS
  (SELECT job, max(salary) AS M
   FROM payroll
   GROUP BY job)
SELECT j.job, p.name, p.salary
FROM jobSal j, payroll p
WHERE j.job = p.job
      and j.M = p.salary;
```



job	M
TA	60000
Prof	100000

payroll

job	M	user_id	name	job	salary
TA	60000	123	Jack	TA	50000
TA	60000	345	Allison	TA	60000
Prof	100000	567	Magda	Prof	90000
Prof	100000	789	Dan	Prof	100000

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
WITH jobSal AS
  (SELECT job, max(salary) AS M
   FROM payroll
   GROUP BY job)
SELECT j.job, p.name, p.salary
FROM jobSal j, payroll p
WHERE j.job = p.job
      and j.M = p.salary;
```

job	M
TA	60000
Prof	100000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

job	M	user_id	name	job	salary
TA	60000	123	Jack	TA	50000
TA	60000	345	Allison	TA	60000
Prof	100000	567	Magda	Prof	90000
Prof	100000	789	Dan	Prof	100000

job	name	salary
TA	Allison	60000
Prof	Dan	100000

The Witnessing Problem

Find the person with highest salary for each job

Solution 1: Using WITH
Solution 2: Using HAVING

Plan:

1. Compute the max(salary) for each job
2. Join back with payroll on job
3. Return the users where salary = max(salary)

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

Plan:

1. Compute the $\max(\text{salary})$ for each job
2. Join back with payroll on job
3. Return the users where $\text{salary} = \max(\text{salary})$

We first join

Goes in HAVING

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, MAX(p1.salary)
FROM payroll AS p1

GROUP BY p1.job
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, MAX(p1.salary)
FROM payroll AS p1

GROUP BY p1.job
```

Similar to jobSal
in our first solution

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, MAX(p1.salary)  
FROM payroll AS p1  
  
GROUP BY p1.job
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job  
FROM payroll AS p1  
  
GROUP BY p1.job
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job
```

Similar to joining
jobSal with payroll

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job
```

Incorrect!

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
```

Correct; but not done!

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Which P2 should we return for each job?

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
HAVING P2.salary = MAX(p1.salary)
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
HAVING MAX(p1.salary) = P2.salary;
```

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
HAVING MAX(p1.salary) = P2.salary;
```



payroll join with payroll

P1				P2			
user_id	name	job	salary	user_id	name	job	salary
123	Jack	TA	50000	123	Jack	TA	50000
345	Allison	TA	60000	123	Jack	TA	50000
123	Jack	TA	50000	345	Allison	TA	60000
345	Allison	TA	60000	345	Allison	TA	60000
567	Magda	Prof	90000	567	Magda	Prof	90000
789	Dan	Prof	100000	567	Magda	Prof	90000
567	Magda	Prof	90000	789	Dan	Prof	100000
789	Dan	Prof	100000	789	Dan	Prof	100000

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
HAVING MAX(p1.salary) = P2.salary;
```

Group by

P1				P2			
user_id	name	job	salary	user_id	name	job	salary
123	Jack	TA	50000	123	Jack	TA	50000
345	Allison	TA	60000	123	Jack	TA	50000
123	Jack	TA	50000	345	Allison	TA	60000
345	Allison	TA	60000	345	Allison	TA	60000
567	Magda	Prof	90000	567	Magda	Prof	90000
789	Dan	Prof	100000	567	Magda	Prof	90000
567	Magda	Prof	90000	789	Dan	Prof	100000
789	Dan	Prof	100000	789	Dan	Prof	100000

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
HAVING MAX(p1.salary) = P2.salary;
```

Compute max(p1.salary)

P1				P2			
user_id	name	job	salary	user_id	name	job	salary
123	Jack	TA	50000	123	Jack	TA	50000
345	Allison	TA	60000	123	Jack	TA	50000
123	Jack	TA	50000	345	Allison	TA	60000
345	Allison	TA	60000	345	Allison	TA	60000
567	Magda	Prof	90000	567	Magda	Prof	90000
789	Dan	Prof	100000	567	Magda	Prof	90000
567	Magda	Prof	90000	789	Dan	Prof	100000
789	Dan	Prof	100000	789	Dan	Prof	100000

max(salary)=60000

max(salary)=60000

max(salary)=100000

max(salary)=100000

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
HAVING MAX(p1.salary) = P2.salary;
```



P1				P2			
user_id	name	job	salary	user_id	name	job	salary
123	Jack	TA	50000	123	Jack	TA	50000
345	Allison	TA	60000	123	Jack	TA	50000
123	Jack	TA	50000	345	Allison	TA	60000
345	Allison	TA	60000	345	Allison	TA	60000
567	Magda	Prof	90000	567	Magda	Prof	90000
789	Dan	Prof	100000	567	Magda	Prof	90000
567	Magda	Prof	90000	789	Dan	Prof	100000
789	Dan	Prof	100000	789	Dan	Prof	100000

max(salary)=60000

max(salary)=60000

max(salary)=100000

max(salary)=100000

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
HAVING MAX(p1.salary) = P2.salary;
```

P1				P2			
user_id	name	job	salary	user_id	name	job	salary
123	Jack	TA	50000	123	Jack	TA	50000
345	Allison	TA	60000	123	Jack	TA	50000
123	Jack	TA	50000	345	Allison	TA	60000
345	Allison	TA	60000	345	Allison	TA	60000
567	Magda	Prof	90000	567	Magda	Prof	90000
789	Dan	Prof	100000	567	Magda	Prof	90000
567	Magda	Prof	90000	789	Dan	Prof	100000
789	Dan	Prof	100000	789	Dan	Prof	100000

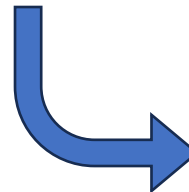
max(salary)=60000

max(salary)=60000

max(salary)=100000

max(salary)=100000

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



p1.job	P2.name	P2.salary
TA	Allison	60000
Prof	Dan	100000

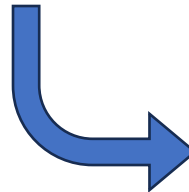
The Witnessing Problem

Find the person with highest salary for each job

```
SELECT p1.job, P2.name, P2.salary
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.job, P2.name, P2.salary
HAVING MAX(p1.salary) = P2.salary;
```

Final output has the witnesses

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



p1.job	P2.name	P2.salary
TA	Allison	60000
Prof	Dan	100000

Subqueries in FROM

Subqueries in FROM

What is the average salary of car drivers?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in FROM

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.*
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in FROM

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.*
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```

Side note:
This is called a
semi-join

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in FROM

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.*
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```

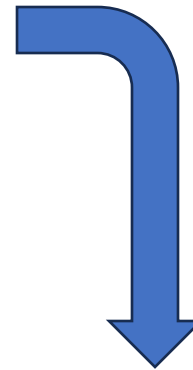
Side note:
This is called a
semi-join

A semi-join is a join of two relations,
followed by a projection on the attributes of the first relation

Subqueries in FROM

What is the average salary of car drivers?

```
WITH cardrivers AS  
  (SELECT DISTINCT p.*  
   FROM payroll p, regist r  
   WHERE p.user_id=r.user_id)  
SELECT avg(salary)  
FROM cardrivers;
```

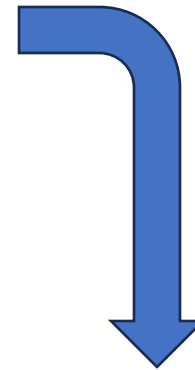


```
SELECT avg(C.salary)  
FROM (SELECT DISTINCT p.*  
       FROM payroll p, regist r  
       WHERE p.user_id=r.user_id) as C;
```

Subqueries in FROM

What is the average salary of car drivers?

```
WITH cardrivers AS
  (SELECT DISTINCT p.*
   FROM payroll p, regist r
   WHERE p.user_id=r.user_id)
SELECT avg(salary)
FROM cardrivers;
```



```
SELECT avg(C.salary)
FROM (SELECT DISTINCT p.*
       FROM payroll p, regist r
       WHERE p.user_id=r.user_id) as C;
```

Subquery in
the FROM clause

Subqueries in FROM

What is the average salary of car drivers?

```
WITH cardrivers AS  
  (SELECT DISTINCT p.*  
   FROM payroll p, regist r  
   WHERE p.user_id=r.user_id)  
SELECT avg(salary)  
FROM cardrivers;
```



Must have
an alias

```
SELECT avg(C.salary)  
FROM (SELECT DISTINCT p.*  
       FROM payroll p, regist r  
       WHERE p.user_id=r.user_id) as C;
```

Subquery in
the FROM clause

- Subquery in FROM is the same as one in WITH
- Sometimes WITH makes the query easier to read
- Some DBMS may not support one or the other

Subqueries in SELECT

Subqueries in SELECT

We can use subqueries in SELECT, but caveat:

- A subquery returns a set...
- ...while in SELECT we must list single values!
- Must ensure that our query returns a single value

Subqueries in SELECT

For each user, find the average salary of their job type

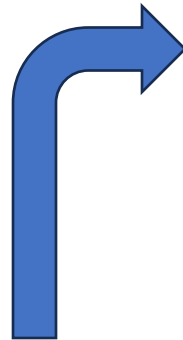
payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

For each user, find the average salary of their job type

We want this



name	salary	Avg
Jack	50000	55000
Allison	60000	55000
Magda	90000	95000
Dan	100000	95000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

For each user, find the average salary of their job type

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

For each user, find the average salary of their job type

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Semantics:
Nested for loops!

Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

	user_id	name	job	salary
→	123	Jack	TA	50000
	345	Allison	TA	60000
	567	Magda	Prof	90000
	789	Dan	Prof	100000

A single
FOR loop:
payroll p

Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

A single FOR loop: payroll p

For each P, compute a subquery

payroll p1

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

A single FOR loop: payroll p

For each P, compute a subquery

payroll p1

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

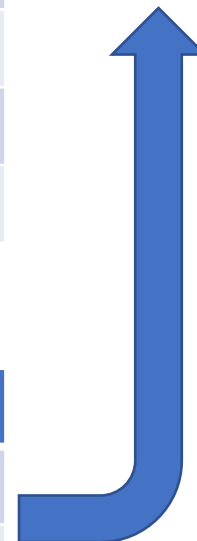
user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

For each P, compute a subquery

payroll p1

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

55000



Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

	user_id	name	job	salary	
	123	Jack	TA	50000	55000
	345	Allison	TA	60000	
	567	Magda	Prof	90000	
	789	Dan	Prof	100000	

Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

user_id	name	job	salary	
123	Jack	TA	50000	55000
→ 345	Allison	TA	60000	
567	Magda	Prof	90000	
789	Dan	Prof	100000	

payroll p1

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

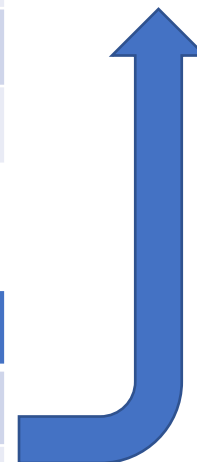
user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



payroll p1

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

55000



Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

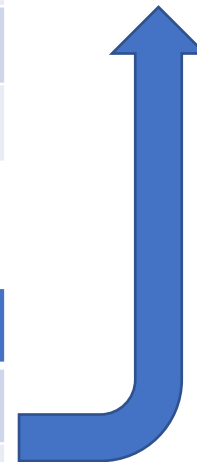


payroll p1

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

55000

55000



Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

user_id	name	job	salary	
123	Jack	TA	50000	55000
345	Allison	TA	60000	55000
→ 567	Magda	Prof	90000	95000
789	Dan	Prof	100000	

payroll p1

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

```
SELECT p.name, (SELECT AVG(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll p

user_id	name	job	salary	
123	Jack	TA	50000	55000
345	Allison	TA	60000	55000
567	Magda	Prof	90000	95000
→ 789	Dan	Prof	100000	95000

payroll p1

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

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 query, unnested

```
SELECT p1.name, AVG(P2.salary)
FROM payroll AS p1, payroll AS P2
WHERE p1.job = P2.job
GROUP BY p1.user_id, p1.name;
```

Subqueries in SELECT

- A subquery in SELECT can be unnested
- careful: sometimes it requires left outer joins

Subqueries in SELECT

For each person find the number of cars they drive

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.user_id =
                    r.user_id)
FROM payroll AS P;
```

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.user_id =
                      r.user_id)
FROM payroll AS P;
```



```
SELECT p.name, COUNT(r.car)
FROM payroll AS P, regist AS R
WHERE p.user_id = r.user_id
GROUP BY p.user_id, 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.user_id =
                      r.user_id)
FROM payroll AS P;
```

Not the same!
Why?



```
SELECT p.name, COUNT(r.car)
FROM payroll AS P, regist AS R
WHERE p.user_id = r.user_id
GROUP BY p.user_id, 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.user_id =
                      r.user_id)
FROM payroll AS P;
```

0-count case not covered!



```
SELECT p.name, COUNT(r.car)
FROM payroll AS P, regist AS R
WHERE p.user_id = r.user_id
GROUP BY p.user_id, 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.user_id =
                      r.user_id)
FROM payroll AS P;
```

name	Count
Jack	1
Allison	0
Magda	2
Dan	0

0-count case not covered!

```
SELECT p.name, COUNT(r.car)
FROM payroll AS P, regist AS R
WHERE p.user_id = r.user_id
GROUP BY p.user_id, p.name;
```

name	Count
Jack	1
Magda	2

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.user_id =
                    r.user_id)
FROM payroll AS P;
```



Still possible to 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.user_id =
                      r.user_id)
FROM payroll AS P;
```



Still possible to unnest

```
SELECT p.name, COUNT(r.car)
FROM payroll AS P LEFT OUTER JOIN
    regist AS R ON p.user_id = r.user_id
GROUP BY p.user_id, p.name;
```

Subqueries in SELECT

- Lesson:
 - Unnesting queries may require left outer join

- Another issue:
 - Subqueries in SELECT must return a single value
 - Otherwise, they produce an error (except Sqlite...)

Subqueries in SELECT

For each person list the cars that they drive

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

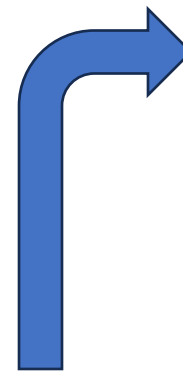
regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in SELECT

For each person list the cars that they drive

Intended answer



name	car
Jack	Charger
Magda	Civic
Magda	Pinto

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in SELECT

For each person list the cars that they drive

```
SELECT p.name, (SELECT r.car
                FROM regist r
                WHERE p.user_id=r.user_id)
FROM payroll p;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in SELECT

For each person list the cars that they drive

```
SELECT p.name, (SELECT r.car
                FROM regist r
                WHERE p.user_id=r.user_id)
FROM payroll p;
```

WRONG! Why?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in SELECT

For each person list the cars that they drive

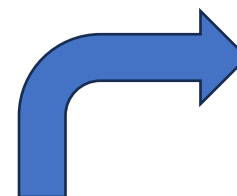
```
SELECT p.name, (SELECT r.car  
                FROM regist r  
                WHERE p.user_id=r.user_id)  
FROM payroll p;
```

Is not always
a single value

WRONG! Why?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto

name	car
Jack	Charger
Allison	...
Magda	Civic Pinto
Dan	...

Subqueries in SELECT

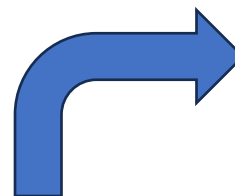
For each person list the cars that they drive

```
SELECT p.name, (SELECT r.car  
                FROM regist r  
                WHERE p.user_id=r.user_id)  
FROM payroll p;
```

Is not always
a single value

WRONG! Why?

Sqlite returns junk.
Better systems give an error



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto


name	car
Jack	Charger
Allison	...
Magda	Civic Pinto
Dan	...

Subqueries in SELECT

For each person list the cars that they drive

```
SELECT p.name, (SELECT r.car  
                FROM regist r  
                WHERE p.user_id=r.user_id)  
FROM payroll p;
```

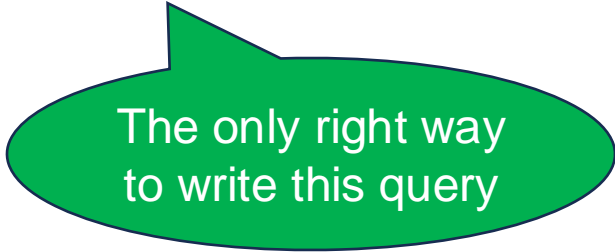
WRONG! Why?



```
SELECT p.name, r.car  
FROM payroll p, regist r  
WHERE p.user_id=r.user_id;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



The only right way
to write this query

Subqueries in SELECT

Final wrinkle:

- A query with a subquery in SELECT may introduce unwanted duplicates
- Need DISTINCT

Subqueries in SELECT

Compute the average salary for each job

Want this output:



job	avg(...)
TA	55000
Prof	95000

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

Compute the average salary for each job

```
SELECT p.job, (SELECT avg(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

Compute the average salary for each job

```
SELECT p.job, ( SELECT avg(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

How many records are
in the output?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in SELECT

Compute the average salary for each job

```
SELECT p.job, (SELECT avg(p1.salary)
                FROM payroll AS p1
                WHERE p.job = p1.job)
FROM payroll AS P;
```

How many records are
in the output?



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

job	avg(...)
TA	55000
TA	55000
Prof	95000
Prof	95000

Subqueries in SELECT

Compute the average salary for each job

```
SELECT DISTINCT p.job,  
    (SELECT avg(p1.salary)  
     FROM payroll AS p1  
     WHERE p.job = p1.job)  
FROM payroll AS P;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

job	avg(...)
TA	55000
TA	55000
Prof	95000
Prof	95000

Subqueries in SELECT

Compute the average salary for each job

```
SELECT DISTINCT p.job,  
    (SELECT avg(p1.salary)  
     FROM payroll AS p1  
     WHERE p.job = p1.job)  
FROM payroll AS P;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

job	avg(...)
TA	55000
Prof	95000



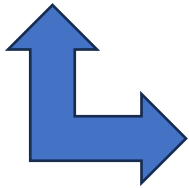
job	avg(...)
TA	55000
TA	55000
Prof	95000
Prof	95000

Subqueries in SELECT

Compute the average salary for each job

```
SELECT DISTINCT p.job,  
    (SELECT avg(p1.salary)  
    FROM payroll AS p1  
    WHERE p.job = p1.job)  
FROM payroll AS P;
```

Under the hood:
GROUP BY replaces
two loops with one
loop over some
hash table



```
SELECT p.job, avg(p.salary)  
FROM payroll AS P  
GROUP BY p.job;
```

Discussion

- Queries in SELECT must return single value
- Think about edge cases: zero matches, null values
- Best: avoid nested queries when possible
- Appreciate the utility of GROUP BY

Subqueries in WHERE/HAVING

Subqueries in WHERE/HAVING

- Can use subquery that returns single value; same as in SELECT

- Additional predicates:
 - EXISTS / NOT EXISTS
 - IN / NOT IN
 - ANY / ALL

Subqueries in WHERE/HAVING

Find all employees who earn less than the average of their job

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in WHERE/HAVING

Find all employees who earn less than the average of their job

```
SELECT p.name, p.salary
FROM payroll p
WHERE p.salary < (SELECT avg(p1.salary)
FROM payroll p1
WHERE p.job = p1.job);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in WHERE/HAVING

Find all employees who earn less than the average of their job

```
SELECT p.name, p.salary
FROM payroll p
WHERE p.salary < (SELECT avg(p1.salary)
FROM payroll p1
WHERE p.job = p1.job);
```

We can unnest
using HAVING

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Subqueries in WHERE/HAVING

Find all employees who earn less than the average of their job

```
SELECT p.name, p.salary
FROM payroll p
WHERE p.salary < (SELECT avg(p1.salary)
                   FROM payroll p1
                   WHERE p.job = p1.job);
```

We can unnest
using HAVING

```
SELECT p.name, p.salary
FROM payroll p, payroll p1
WHERE p.job = p1.job
GROUP BY p.name, p.salary
HAVING p.salary < avg(p1.salary);
```

Subqueries in WHERE/HAVING

SQL has a few predicates that apply to a subquery:

- **EXISTS (SELECT)** checks if it is not empty
- **NOT EXISTS (SELECT ...)** checks if it is empty

Subqueries in WHERE/HAVING

SQL has a few predicates that apply to a subquery:

- `EXISTS (SELECT)` checks if it is not empty
`NOT EXISTS (SELECT ...)` checks if it is empty
- `X in (SELECT Y FROM ...)` checks output has X
`X not in (SELECT Y ...)` checks if it doesn't have X

Subqueries in WHERE/HAVING

SQL has a few predicates that apply to a subquery:

- `EXISTS (SELECT)` checks if it is not empty
`NOT EXISTS (SELECT ...)` checks if it is empty
- `X in (SELECT Y FROM ...)` checks output has X
`X not in (SELECT Y ...)` checks if it doesn't have X
- `X > ALL(SELECT ...)`
`X > ANY(SELECT ...)`
checks if X is > than one or all values in output

Subqueries in WHERE/HAVING

SQL has a few predicates that apply to a subquery:

Next

- EXISTS (SELECT) checks if it is not empty
NOT EXISTS (SELECT ...) checks if it is empty

- X in (SELECT Y FROM ...) checks output has X
X not in (SELECT Y ...) checks if it doesn't have X

- X > ALL(SELECT ...)
X > ANY(SELECT ...)
checks if X is > than one or all values in output

Next lecture

Subqueries in WHERE/HAVING

Find people who **do** drive cars

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in WHERE/HAVING

Find people who **do** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
    (SELECT *
     FROM regist r
     WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in WHERE/HAVING

Find people who **do** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
  (SELECT *
   FROM regist r
   WHERE p.user_id = r.user_id);
```

Same as
a semi-join

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto



user_id	name
123	Jack
567	Magda

Subqueries in WHERE/HAVING

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
    (SELECT *
     FROM regist r
     WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in WHERE/HAVING

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Subqueries in WHERE/HAVING

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto



user_id	name
345	Allison
789	Dan

Subqueries in WHERE/HAVING

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```


Called an
anti-semijoin

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto



user_id	name
345	Allison
789	Dan

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

P →

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

P →

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=123

user_id	car

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

P →

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

R →

user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=123

user_id	car
123	Charger

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll



user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist



user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=123

user_id	car
123	Charger

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll



user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist



user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=123

user_id	car
123	Charger

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

P →

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=123

user_id	car
123	Charger

Output so far



user_id	name

Done user_id=123
Exists answers

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll



user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=123

user_id	car
123	Charger

Output so far



user_id	name

Skip user_id=123

Done user_id=123
Exists answers

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll



user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Output so far



user_id	name



Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

P →

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=345

user_id	car

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto



Compute subquery
for p.user_id=345

user_id	car

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto



Compute subquery
for p.user_id=345

user_id	car

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto



Compute subquery
for p.user_id=345

user_id	car

Output so far



user_id	name

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto



Compute subquery
for p.user_id=345

user_id	car

Output so far



user_id	name

Done user_id=345
Not exists answers

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=345

user_id	car

Output so far



user_id	name
345	Allison

Output user_id=345

Done user_id=345
Not exists answers

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Output so far



user_id	name
345	Allison



Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto

Output so far



user_id	name
345	Allison

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto



Compute subquery
for p.user_id=567

user_id	car

Output so far



user_id	name
345	Allison

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto



Compute subquery
for p.user_id=567

user_id	car
567	Civic

Output so far



user_id	name
345	Allison

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000



regist

user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=567

user_id	car
567	Civic
567	Pinto

Output so far



user_id	name
345	Allison

Skip user_id=567

Done user_id=567
Exists answers

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto



Output so far



user_id	name
345	Allison

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Output so far



user_id	name
345	Allison



Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=789

user_id	car

Output so far



user_id	name
345	Allison



Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

P →

regist

user_id	car
123	Charger
567	Civic
567	Pinto

R →

Compute subquery
for p.user_id=789

user_id	car

Output so far



user_id	name
345	Allison

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

P →

regist

user_id	car
123	Charger
567	Civic
567	Pinto

R →

Compute subquery
for p.user_id=789

user_id	car

Output so far



user_id	name
345	Allison

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

Output so far



user_id	name
345	Allison
789	Dan

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Compute subquery
for p.user_id=789

user_id	car

Output user_id= 567

Done user_id=567
Not exists answers

Nested Loop Semantics

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

user_id	name
345	Allison
789	Dan

Final answer

regist

user_id	car
123	Charger
567	Civic
567	Pinto



Summary

- Subquery can occur in **SELECT/FROM/WHERE**
- Sometimes (not always) it is possible to unnest
- Keep in mind edge cases: zero counts
- Most difficult: Existential / universal quantifiers
Next lecture!

Predicates on Subqueries

- EXISTS (SELECT) checks if it is not empty
NOT EXISTS (SELECT ...) checks if it is empty

- X in (SELECT Y FROM ...) checks output has X
X not in (SELECT Y ...) checks if it doesn't have X

- X > ALL(SELECT ...)
X > ANY(SELECT ...)
checks if X is > than one or all values in output

Recap: EXISTS

Find people who **do** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
    (SELECT *
     FROM regist r
     WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Recap: EXISTS

Find people who **do** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
    (SELECT *
     FROM regist r
     WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto



user_id	name
123	Jack
567	Magda

Recap: EXISTS

Find people who **do** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
  (SELECT *
   FROM regist r
   WHERE p.user_id = r.user_id);
```

Same as
a semi-join



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

user_id	name
123	Jack
567	Magda

Recap: NOT EXISTS

Find people who **do not** drive cars

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Recap: NOT EXISTS

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
  (SELECT *
   FROM regist r
   WHERE p.user_id != r.user_id);
```

Does this work?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Recap: NOT EXISTS

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
  (SELECT *
   FROM regist r
   WHERE p.user_id != r.user_id);
```

Does this work?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

NO!
It returns everybody

Recap: NOT EXISTS

Find people who **do not** drive cars

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Recap: NOT EXISTS

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Recap: NOT EXISTS

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto



user_id	name
345	Allison
789	Dan

Unnesting EXISTS

Find people who **do** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
    (SELECT *
     FROM regist r
     WHERE p.user_id = r.user_id);
```



```
SELECT DISTINCT p.user_id, p.name
FROM payroll p, regist r
WHERE p.user_id = r.user_id;
```

How do we unnest NOT EXISTS?

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
  (SELECT *
   FROM regist r
   WHERE p.user_id = r.user_id);
```



This doesn't work...



```
SELECT DISTINCT p.user_id, p.name
FROM payroll p, regist r
WHERE p.user_id != r.user_id;
```

How do we unnest NOT EXISTS?

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
      (SELECT *
       FROM regist r
       WHERE p.user_id = r.user_id);
```

This query cannot be unnested without aggregates.

Proof next

Monotone Functions

Definition

A function $f: R \rightarrow R$ is monotone if $x \leq y$ implies $f(x) \leq f(y)$

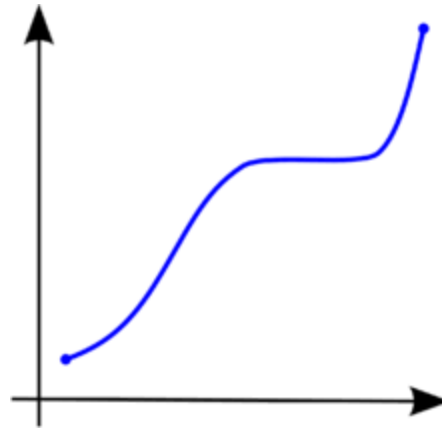
Monotone:

$$x^3 + x^2,$$

$$e^x,$$

$$\log(x),$$

...



Non-Monotone:

$$x^3 - x^2,$$

$$e^{-x},$$

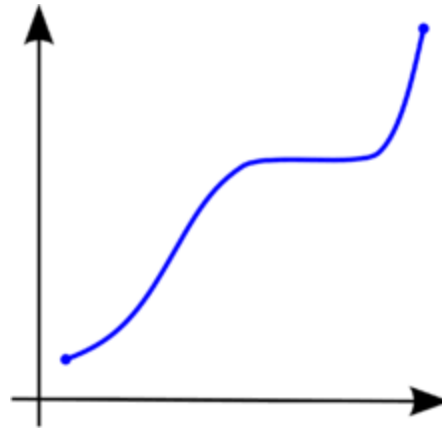
$$\frac{1}{x},$$

...

Monotone Queries

Definition

A query Q is monotone if $I \subseteq J$ implies $q(I) \subseteq q(J)$

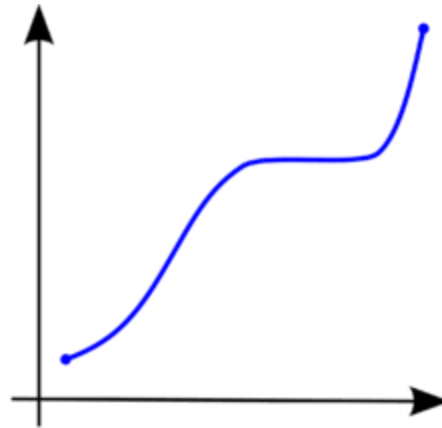


Monotone Queries

Definition

A query Q is monotone if $I \subseteq J$ implies $q(I) \subseteq q(J)$

Adding tuples to the input does
not remove tuples from the output



Monotone Queries

Find people who **do** drive cars

Is this query monotone?

Monotone Queries

Find people who **do** drive cars

Is this query monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

I

Monotone Queries

Find people who **do** drive cars

Is this query monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

I

user_id	name
123	Jack
567	Magda

$q(I)$

Monotone Queries

Find people who **do** drive cars

Is this query monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

I

user_id	name
123	Jack
567	Magda

$q(I)$

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

J

Monotone Queries

Find people who **do** drive cars

Is this query monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

I

user_id	name
123	Jack
567	Magda

$q(I)$

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

J

user_id	name
123	Jack
567	Magda
345	Allison

$q(J)$

Monotone Queries

Find people who **do** drive cars

Is this query monotone?

Yes, it is monotone

Monotone Queries

Find people who **do not** drive cars

Is this query monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

I

user_id	name
345	Allison
789	Dan

$q(I)$

Monotone Queries

Find people who **do not** drive cars

Is this query monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

I

user_id	name
345	Allison
789	Dan

$q(I)$

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

J

Monotone Queries

Find people who **do not** drive cars

Is this query monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

I

user_id	name
345	Allison
789	Dan

$q(I)$

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto
345	Tesla

J

user_id	name
345	Allison
789	Dan

$q(J)$

Monotone Queries

Find people who **do not** drive cars

Is this query monotone?

No, this query is not monotone

Monotone Queries

Theorem

Every SELECT-FROM-WHERE query without subqueries and without aggregates is monotone

Monotone Queries

Theorem

Every SELECT-FROM-WHERE query without subqueries and without aggregates is monotone

Proof. Consider a SQL query:

```
SELECT attrs
FROM T1, T2, ...
WHERE condition
```

Monotone Queries

Theorem

Every SELECT-FROM-WHERE query without subqueries and without aggregates is monotone

Proof. Consider a SQL query:

```
SELECT attrs
FROM T1, T2, ...
WHERE condition
```

Its nested loop semantics is:

```
for each r1 in T1:
  for each t2 in T2:
    for each t3 in T3:
      ...
      if (condition):
        output (r1, r2, ...)
```

If we insert a tuple into one of the input relations T_i , we will not remove any tuples from the output.

Consequence

The query “Find people who **do not** drive cars” cannot be unnested without aggregates

- The property whether the query is monotone or not does not depend on its SQL writeup
- Instead, it depends on the meaning of the query, regardless of how we write it in SQL

Monotone Queries

Count the number of employees.

```
SELECT count (*)  
FROM payroll
```

Monotone?

Monotone Queries

Count the number of employees.

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

I

```
SELECT count (*)  
FROM payroll
```

Monotone?

count

4

$q(I)$

Monotone Queries

Count the number of employees.

```
SELECT count (*)  
FROM payroll
```

Monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

I

count

4

q(I)

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000
555	Alice	TA	80000

J

count

5

q(J)

Monotone Queries

Count the number of employees.

```
SELECT count (*)  
FROM payroll
```

Monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

I

count

4

$q(I)$

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000
555	Alice	TA	80000

J

count

5

$q(J)$

$\{4\} \not\subseteq \{5\}$

Monotone Queries

Count the number of employees.

```
SELECT count (*)  
FROM payroll
```

Monotone?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

I

count

4

$q(I)$

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000
555	Alice	TA	80000

J

count

5

$q(J)$

Not
monotone

$\{4\} \not\subseteq \{5\}$

IN and NOT IN

Subqueries in WHERE/HAVING

X in (SELECT Y FROM ...)

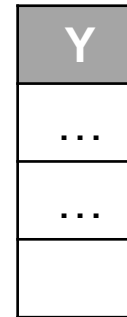
- Compute the subquery
- Check if $X \in Output$

Y
...
...

Subqueries in WHERE/HAVING

X in (SELECT Y FROM ...)

- Compute the subquery
- Check if $X \in Output$



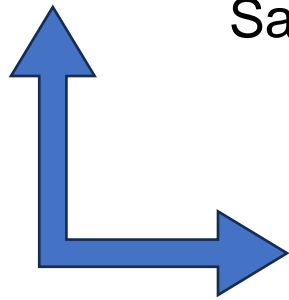
X not in (SELECT Y ...) not(X in (SELECT Y ...))

- Compute the subquery
- Check if $X \notin Output$

EXISTS v.s. IN

Find people who **do** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE exists
    (SELECT *
     FROM regist r
     WHERE p.user_id = r.user_id);
```



Same output

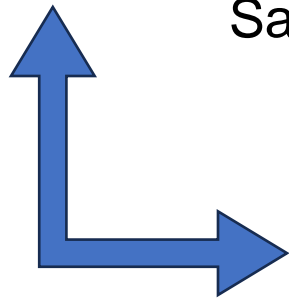
```
SELECT p.user_id, p.name
FROM payroll p
WHERE p.user_id in
    (SELECT r.user_id
     FROM regist r);
```

NOT EXISTS v.s. NOT IN

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
    (SELECT *
     FROM regist r
     WHERE p.user_id = r.user_id);
```

Same output



```
SELECT p.user_id, p.name
FROM payroll p
WHERE p.user_id not in
    (SELECT r.user_id
     FROM regist r);
```

Computing NOT IN

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE p.user_id not in
      (SELECT r.user_id
       FROM regist r);
```

1. Compute subquery

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Computing NOT IN

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE p.user_id not in
      (SELECT r.user_id
       FROM regist r);
```

1. Compute subquery

user_id
123
567
567

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Computing NOT IN

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE p.user_id not in
      (SELECT r.user_id
       FROM regist r);
```

1. Compute subquery

user_id
123
567
567

2. For each payroll,
check if user_id \notin subquery



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

user_id	name
345	Allison
789	Dan

NOT EXISTS v.s. NOT IN

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
    (SELECT *
     FROM regist r
     WHERE p.user_id = r.user_id);
```

Which query is
more efficient?

```
SELECT p.user_id, p.name
FROM payroll p
WHERE p.user_id not in
    (SELECT r.user_id
     FROM regist r);
```

NOT EXISTS v.s. NOT IN

Find people who **do not** drive cars

```
SELECT p.user_id, p.name
FROM payroll p
WHERE not exists
  (SELECT *
   FROM regist r
   WHERE p.user_id = r.user_id);
```

Correlated subquery:
computed repeatedly,
once for each payroll

Which query is
more efficient?

Computed only once

```
SELECT p.user_id, p.name
FROM payroll p
WHERE p.user_id not in
  (SELECT r.user_id
   FROM regist r);
```


ANY and ALL

ANY and ALL

< or <= or > or ...

$X < ANY (SELECT Y FROM ...)$

- Compute the subquery
- Check if there exists $Y \in Output$ s.t. $X < Y$

Y
...
...

ANY and ALL

< or <= or > or ...

$X < ANY (SELECT Y FROM ...)$

- Compute the subquery
- Check if there exists $Y \in Output$ s.t. $X < Y$

Y
...
...

$X < ALL (SELECT Y FROM ...)$

- Compute the subquery
- Check if for all $Y \in Output$, $X < Y$

ANY and ALL

Find people who drive **some** car made after 2017

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022

ANY and ALL

Find people who drive **some** car made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ANY (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022

ANY and ALL

Find people who drive **some** car made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ANY (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Jack:


year
2016
2018

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022



name	...
Jack	

ANY and ALL

Find people who drive **some** car made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ANY (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Allison:


year

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022



name	...
Jack	

ANY and ALL

Find people who drive **some** car made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ANY (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Magda:

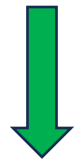
year
2020
2022

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022



name	...
Jack	
Magda	

ANY and ALL

Find people who drive **some** car made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ANY (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Dan:


year

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022



name	...
Jack	
Magda	

ANY and ALL

Find people who drive **some** car made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ANY (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Same as
a semi-join

```
SELECT DISTINCT p.*
FROM payroll p, regist r
WHERE p.user_id = r.user_id
      and r.year > 2017
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022

name	...
Jack	
Magda	

ANY and ALL

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022

ANY and ALL

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Do we output Jack?

Jack:

year
2016
2018

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022

A diagram showing a table with two columns: 'name' and '...'. An orange arrow points down to the table, indicating a result set or a specific row being highlighted.

name	...

ANY and ALL

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Do we output Jack?

Jack:

year
2016
2018

The test $2017 < \text{ALL}(\dots)$ fails for 2016

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022

name	...

ANY and ALL

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Do we output Allison?

Allison:


year

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022



name	...
Allison	

ANY and ALL

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Allison:

year

Do we output Allison?


The test $2017 < ALL(\dots)$ does not fail anywhere. Hence it holds!!

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022



name	...
Allison	

ANY and ALL

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Magda:

year
2020
2022


The test $2017 < \text{ALL}(\dots)$ does not fail anywhere.

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022



name	...
Allison	
Magda	

ANY and ALL

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Dan:

year


The test $2017 < ALL(\dots)$ does not fail anywhere. Hence it holds!!

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022



name	...
Allison	
Magda	
Dan	

ANY and ALL

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Can we unnest this query?

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022

name	...
Allison	
Magda	
Dan	

ANY and ALL

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

Can we unnest this query?

NO!
Non-monotone

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car	year
123	Charger	2016
123	Tesla	2018
567	Civic	2020
567	Pinto	2022

name	...
Allison	
Magda	
Dan	

Recap: Predicates on Subqueries

- EXISTS / NOT EXISTS
- IN / NOT IN
- ANY / ALL

The are “equivalent” meaning that a query that you can write using one, you can also write using the others

They express QUANTIFIERS

Quantifiers

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL(SELECT r.year
       FROM regist r
       WHERE p.user_id = r.user_id);
```

Quantifiers

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

```
SELECT p.*
FROM payroll p
WHERE NOT EXISTS
  (SELECT *
   FROM regist r
   WHERE p.user_id = r.user_id
        and r.year <= 2017);
```

Quantifiers

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

```
SELECT p.*
FROM payroll p
WHERE NOT EXISTS
  (SELECT *
   FROM regist r
   WHERE p.user_id = r.user_id
        and r.year <= 2017);
```

```
SELECT p.*
FROM payroll p
WHERE p.user_id NOT IN
  (SELECT r.user_id
   FROM regist r
   WHERE r.year <= 2017);
```

Quantifiers

Find people who drive **only** cars made after 2017

```
SELECT p.*
FROM payroll p
WHERE 2017 <
  ALL (SELECT r.year
        FROM regist r
        WHERE p.user_id = r.user_id);
```

All these
compute the
same thing

```
SELECT p.*
FROM payroll p
WHERE NOT EXISTS
  (SELECT *
   FROM regist r
   WHERE p.user_id = r.user_id
   and r.year <= 2017);
```

```
SELECT p.*
FROM payroll p
WHERE p.user_id NOT IN
  (SELECT r.user_id
   FROM regist r
   WHERE r.year <= 2017);
```


- SQL can express naturally queries that represent existential quantifiers

- To write a query that uses a universal quantifier, use DeMorgan's laws (next)

Quantifiers

Quantifiers

There are two types of quantifiers:

- **Exists** ($\exists x, \dots$) there is at least 1 that satisfies predicate
- **forall**: ($\forall x, \dots$) all elements satisfy the predicate

Quantifiers

There are two types of quantifiers:

- **Exists** ($\exists x, \dots$) there is at least 1 that satisfies predicate
- **forall**: ($\forall x, \dots$) all elements satisfy the predicate

SQL makes it easy to write **exists**

Quantifiers

There are two types of quantifiers:

- **Exists** ($\exists x, \dots$) there is at least 1 that satisfies predicate
- **forall**: ($\forall x, \dots$) all elements satisfy the predicate

SQL makes it easy to write **exists**

To write **forall**, use double negation

predicate holds **forall** elements
if and only if

not (**exists** element where **not**(predicate) holds)

Using First Order Logic

Query: persons **P** that drive **only** cars made after 2017:

$$\forall R \in \text{regist}, (\mathbf{P}. \text{user_id} = r. \text{user_id}) \Rightarrow (r. \text{year} > 2017)$$

Using First Order Logic

Query: persons **P** that drive **only** cars made after 2017:

$$\forall R \in \text{regist}, (\mathbf{P}. \text{user_id} = r. \text{user_id}) \Rightarrow (r. \text{year} > 2017)$$

Negation: persons **P** that drive **some** car made on/before 2017:

$$\exists R \in \text{regist}, (\mathbf{P}. \text{user_id} = r. \text{user_id}) \text{ and } (r. \text{year} \leq 2017)$$

Using First Order Logic

Query: persons **P** that drive **only** cars made after 2017:

$$\forall R \in \text{regist}, (\mathbf{P}. \text{user_id} = r. \text{user_id}) \Rightarrow (r. \text{year} > 2017)$$

Negation: persons **P** that drive **some** car made on/before 2017:

$$\exists R \in \text{regist}, (\mathbf{P}. \text{user_id} = r. \text{user_id}) \text{ and } (r. \text{year} \leq 2017)$$

Let's review this slowly

Brief Review of Logic

- Implication: $A \rightarrow B$ is same as: $\text{not}(A) \text{ or } B$

Brief Review of Logic

- Implication: $A \rightarrow B$ is same as: $\text{not}(A) \text{ or } B$
- DeMorgan's Laws:

$$\begin{aligned} \text{not}(A \text{ and } B) &= \text{not}(A) \text{ or } \text{not}(B) \\ \text{not}(A \text{ or } B) &= \text{not}(A) \text{ and } \text{not}(B) \end{aligned}$$

Brief Review of Logic

- Implication: $A \rightarrow B$ is same as: $\text{not}(A) \text{ or } B$
- DeMorgan's Laws:

$$\begin{aligned}\text{not}(A \text{ and } B) &= \text{not}(A) \text{ or } \text{not}(B) \\ \text{not}(A \text{ or } B) &= \text{not}(A) \text{ and } \text{not}(B)\end{aligned}$$

$$\begin{aligned}\text{not}(\exists x, P(x)) &= \forall x, \text{not}(P(x)) \\ \text{not}(\forall x, P(x)) &= \exists x, \text{not}(P(x))\end{aligned}$$

Brief Review of Logic

▪ Implication: $A \rightarrow B$ is same as: $\text{not}(A) \text{ or } B$

▪ DeMorgan's Laws:

$$\begin{aligned}\text{not}(A \text{ and } B) &= \text{not}(A) \text{ or } \text{not}(B) \\ \text{not}(A \text{ or } B) &= \text{not}(A) \text{ and } \text{not}(B)\end{aligned}$$

$$\begin{aligned}\text{not}(\exists x, P(x)) &= \forall x, \text{not}(P(x)) \\ \text{not}(\forall x, P(x)) &= \exists x, \text{not}(P(x))\end{aligned}$$

▪ Consequences

$$\text{not}(A \rightarrow B) = (A \text{ and } \text{not}(B))$$

Brief Review of Logic

▪ Implication: $A \rightarrow B$ is same as: $\text{not}(A) \text{ or } B$

▪ DeMorgan's Laws:

$$\begin{aligned}\text{not}(A \text{ and } B) &= \text{not}(A) \text{ or } \text{not}(B) \\ \text{not}(A \text{ or } B) &= \text{not}(A) \text{ and } \text{not}(B)\end{aligned}$$

$$\begin{aligned}\text{not}(\exists x, P(x)) &= \forall x, \text{not}(P(x)) \\ \text{not}(\forall x, P(x)) &= \exists x, \text{not}(P(x))\end{aligned}$$

▪ Consequences

$$\text{not}(A \rightarrow B) = (A \text{ and } \text{not}(B))$$

$$\text{not}(\forall x, (A(x) \rightarrow B(x))) = \exists x, (A(x) \wedge \text{not}(B(x)))$$

Using First Order Logic

Query: persons **P** that drive **only** cars made after 2017:

$$\forall R \in \text{regist}, (\mathbf{P}. \text{user_id} = r. \text{user_id}) \Rightarrow (r. \text{year} > 2017)$$

Negation: persons **P** that drive **some** car made on/before 2017:

$$\exists R \in \text{regist}, (\mathbf{P}. \text{user_id} = r. \text{user_id}) \text{ and } (r. \text{year} \leq 2017)$$

How to Write FORALL in SQL

Find person **P** drives **only** cars made after 2017

How to Write FORALL in SQL

Find person **P** drives **only** cars made after 2017

Negate: find the other persons

Find person **P** drives **some** car made on or before 2017

How to Write FORALL in SQL

Find person **P** drives **only** cars made after 2017

Negate: find the other persons

Find person **P** drives **some** car made on or before 2017

```
SELECT p.*
FROM payroll p
WHERE EXISTS
  (SELECT r.year
   FROM regist r
   WHERE p.user_id = r.user_id
    and r.year <= 2017);
```

How to Write FORALL in SQL

Find person **P** drives **only** cars made after 2017

Negate: find the other persons

Find person **P** drives **some** car made on or before 2017

```
SELECT p.*
FROM payroll p
WHERE EXISTS
  (SELECT r.year
   FROM regist r
   WHERE p.user_id = r.user_id
    and r.year <= 2017);
```

Negate again:
find the other other persons

```
SELECT p.*
FROM payroll p
WHERE NOT EXISTS
  (SELECT r.year
   FROM regist r
   WHERE p.user_id = r.user_id
    and r.year <= 2017);
```

How to Write FORALL in SQL

Find person **P** drives **only** cars made after 2017

Universal
quantifier

Existential
quantifier

Negate: find the other persons

Find person **P** drives **some** car made on or before 2017

```
SELECT p.*
FROM payroll p
WHERE EXISTS
  (SELECT r.year
   FROM regist r
   WHERE p.user_id = r.user_id
    and r.year <= 2017);
```

Negate again:
find the other other persons

```
SELECT p.*
FROM payroll p
WHERE NOT EXISTS
  (SELECT r.year
   FROM regist r
   WHERE p.user_id = r.user_id
    and r.year <= 2017);
```

Writing universally quantified queries in SQL requires creativity

- Try using DeMorgan's laws: not exists, not in
- Try using ALL
- Try using aggregates, checking count=0

Relational Algebra

Motivation

- SQL is a declarative language:
we say **what**, we don't say **how**
- The query optimizer needs to convert the query into some intermediate language that can be both optimized, and executed
- That language is Relational Algebra

The Five Basic Relational Operators

1. Selection $\sigma_{\text{condition}}(S)$
2. Projection $\Pi_{\text{attrs}}(S)$
3. Join $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
4. Union \cup
5. Set difference $-$

Rename ρ

Let's discuss them one by one

1. Selection

$\sigma_{\text{condition}}(T)$

Returns those tuples in T
that satisfy the condition:

```
SELECT *  
FROM T  
WHERE condition;
```


1. Selection

$\sigma_{\text{condition}}(T)$

Returns those tuples in T that satisfy the condition:

```
SELECT *  
FROM T  
WHERE condition;
```

$\sigma_{\text{salary} \geq 55000}(\text{payroll}) =$

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

1. Selection

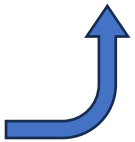
$\sigma_{\text{condition}}(T)$

Returns those tuples in T that satisfy the condition:

```
SELECT *  
FROM T  
WHERE condition;
```

user_id	name	job	salary
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

$\sigma_{\text{salary} \geq 55000}(\text{payroll}) =$



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

1. Selection

$\sigma_{\text{condition}}(T)$

Returns those tuples in T that satisfy the condition:

```
SELECT *  
FROM T  
WHERE condition;
```

$\sigma_{\text{salary} \geq 55000 \text{ and job} = \text{'TA'}}(\text{payroll}) =$

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000


1. Selection

$\sigma_{\text{condition}}(T)$

Returns those tuples in T that satisfy the condition:

```
SELECT *  
FROM T  
WHERE condition;
```

user_id	name	job	salary
345	Allison	TA	60000

$\sigma_{\text{salary} \geq 55000 \text{ and job} = \text{'TA'}}(\text{payroll}) =$ 

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

2. Projection

$$\Pi_{\text{attrs}}(T)$$

Returns all tuples in T keeping only the attributes in the subscript:

```
SELECT attrs  
FROM T;
```

2. Projection

$\Pi_{\text{attrs}}(T)$

Returns all tuples in T keeping only the attributes in the subscript:

$\Pi_{\text{name,salary}}(\text{payroll}) =$

```
SELECT attrs
FROM T;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

2. Projection

$\Pi_{\text{attrs}}(T)$

name	salary
Jack	50000
Allison	60000
Magda	90000
Dan	100000

Returns all tuples in T keeping only the attributes in the subscript:

$\Pi_{\text{name, salary}}(\text{payroll}) =$



```
SELECT attrs  
FROM T;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

2. Projection

 $\Pi_{\text{attrs}}(T)$

Returns all tuples in T keeping only the attributes in the subscript:

 $\Pi_{\text{job}}(\text{payroll}) =$

```
SELECT attrs  
FROM T;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

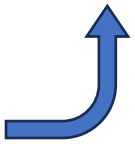
2. Projection

$\Pi_{\text{attrs}}(T)$

job
TA
TA
Prof
Prof

Returns all tuples in T keeping only the attributes in the subscript:

$\Pi_{\text{job}}(\text{payroll}) =$



```
SELECT attrs
FROM T;
```

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

2. Projection

$\Pi_{\text{attrs}}(T)$

Returns all tuples in T keeping only the attributes in the subscript:

```
SELECT attrs
FROM T;
```

job
TA
TA
Prof
Prof

RA can be defined using bag semantics or set semantics.

We always need to say which one we mean.

job
TA
Prof

$\Pi_{\text{job}}(\text{payroll}) =$



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

3. Join

$$S \bowtie_{\theta} T$$

Join S and T using condition θ

```
SELECT *  
FROM S, T  
WHERE  $\theta$ ;
```

3. Join

$$S \bowtie_{\theta} T$$

Join S and T using condition θ

```
SELECT *  
FROM S, T  
WHERE  $\theta$ ;
```

$\text{payroll} \bowtie_{\text{user_id}=\text{user_id}} \text{regist} =$

payroll


user_id	name	job	salary	regist	
123	Jack	TA	50000	user_id	car
345	Allison	TA	60000	123	Charger
567	Magda	Prof	90000	567	Civic
789	Dan	Prof	100000	567	Pinto

3. Join

$$S \bowtie_{\theta} T$$

user_id	name	job	salary	user_id	car
123	Jack	TA	50000	123	Charger
567	Magda	Prof	90000	567	Civic
567	Magda	Prof	90000	567	Pinto

Join S and T using condition θ

$$\text{payroll} \bowtie_{\text{user_id}=\text{user_id}} \text{regist} =$$


```
SELECT *  
FROM S, T  
WHERE  $\theta$ ;
```

payroll

user_id	name	job	salary	regist
123	Jack	TA	50000	user_id
345	Allison	TA	60000	123
567	Magda	Prof	90000	567
789	Dan	Prof	100000	567

user_id	car
123	Charger
567	Civic
567	Pinto

Many Variants of Join

- **Eq-join:** $\text{payroll} \bowtie_{\text{user_id}=\text{user_id}} \text{regist}$
- **Theta-join:** $\text{payroll} \bowtie_{\text{user_id}<\text{user_id}} \text{regist}$
- **cartesian product:** $\text{payroll} \times \text{regist}$
- **Natural Join:** $\text{payroll} \bowtie \text{regist}$

Many Variants of Join

▪ **Eq-join:** payroll $\bowtie_{\text{user_id}=\text{user_id}}$ regist

Only =

▪ **Theta-join:** payroll $\bowtie_{\text{user_id} < \text{user_id}}$ regist

Any condition

▪ **cartesian product:** payroll \times regist

▪ **Natural Join:** payroll \bowtie regist

Many Variants of Join

▪ **Eq-join:** payroll $\bowtie_{\text{user_id}=\text{user_id}}$ regist

Only =

▪ **Theta-join:** payroll $\bowtie_{\text{user_id} < \text{user_id}}$ regist

Any condition

▪ **cartesian product:** payroll \times regist

▪ **Natural Join:** payroll \bowtie regist

Next

cartesian Product / Cross Product

$S \times T$

Cross product of S and T

```
SELECT *  
FROM S, T
```

cartesian Product / Cross Product

$S \times T$

Cross product of S and T

```
SELECT *  
FROM S, T
```

payroll \times regist =

payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

cartesian Product / Cross Product

$S \times T$

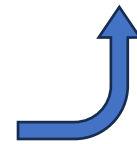
12 tuples

user_id	name	job	salary	user_id	car
123	Jack	TA	50000	123	Charger
123	Jack	TA	50000	567	Civic
			...		
789	Dan	Prof	100000	567	Pinto

Cross product of S and T

```
SELECT *  
FROM S, T
```

payroll \times regist =



payroll

user_id	name	job	salary	regist	
123	Jack	TA	50000	user_id	car
345	Allison	TA	60000	123	Charger
567	Magda	Prof	90000	567	Civic
789	Dan	Prof	100000	567	Pinto

cartesian Product / Cross Product

$S \times T$

Cross product of S and T

```
SELECT *  
FROM S, T
```

Join = cartesian product + selection

$$R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$$

Natural Join

$S \bowtie T$

Join S, T on
common attributes,
retain only one copy
of those attributes

Natural Join

$S \bowtie T$

Join S, T on
common attributes,
retain only one copy
of those attributes

$\text{payroll} \bowtie \text{regist} =$

payroll

user_id	name	job	salary	regist	car
123	Jack	TA	50000	123	Charger
345	Allison	TA	60000	567	Civic
567	Magda	Prof	90000	567	Pinto
789	Dan	Prof	100000		

Natural Join

$S \bowtie T$

Join S, T on
common attributes,
retain only one copy
of those attributes

user_id	name	job	salary	car
123	Jack	TA	50000	Charger
567	Magda	Prof	90000	Civic
567	Magda	Prof	90000	Pinto

Only one
user_id attr

payroll \bowtie regist =

payroll

user_id	name	job	salary	regist
123	Jack	TA	50000	
345	Allison	TA	60000	
567	Magda	Prof	90000	
789	Dan	Prof	100000	

user_id	car
123	Charger
567	Civic
567	Pinto

Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$

- $R(A, B) \bowtie S(C, D)$

- $R(A, B) \bowtie S(A, B)$

Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$

R	A	B	S	B	C
	1	10		10	8
	2	10		10	9
	2	20		20	8
				50	7

- $R(A, B) \bowtie S(C, D)$

- $R(A, B) \bowtie S(A, B)$

Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$
eqjoin on attribute B (5 tuples)

R	A	B	S	B	C
	1	10		10	8
	2	10		10	9
	2	20		20	8
				50	7

- $R(A, B) \bowtie S(C, D)$

- $R(A, B) \bowtie S(A, B)$

Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$
eqjoin on attribute B (5 tuples)

R	A	B	S	B	C
	1	10		10	8
	2	10		10	9
	2	20		20	8
				50	7

- $R(A, B) \bowtie S(C, D)$

R	A	B	S	C	D
	1	10		8	u
	2	10		9	v
	2	20		8	v
				7	w

- $R(A, B) \bowtie S(A, B)$

Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$
eqjoin on attribute B (5 tuples)
- $R(A, B) \bowtie S(C, D)$
cross product (12 tuples)
- $R(A, B) \bowtie S(A, B)$

R	A	B	S	B	C
	1	10		10	8
	2	10		10	9
	2	20		20	8
				50	7

R	A	B	S	C	D
	1	10		8	u
	2	10		9	v
	2	20		8	v
				7	w

Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$
eqjoin on attribute B (5 tuples)
- $R(A, B) \bowtie S(C, D)$
cross product (12 tuples)
- $R(A, B) \bowtie S(A, B)$

R	A	B	S	B	C
	1	10		10	8
	2	10		10	9
	2	20		20	8
				50	7

R	A	B	S	C	D
	1	10		8	u
	2	10		9	v
	2	20		8	v
				7	w

R	A	B	S	A	B
	1	10		1	10
	2	10		2	20
	2	20			

Natural Join

What do these natural joins output?

- $R(A, B) \bowtie S(B, C)$
eqjoin on attribute B (5 tuples)

R	A	B	S	B	C
	1	10		10	8
	2	10		10	9
	2	20		20	8
				50	7

- $R(A, B) \bowtie S(C, D)$
cross product (12 tuples)

R	A	B	S	C	D
	1	10		8	u
	2	10		9	v
	2	20		8	v
				7	w

- $R(A, B) \bowtie S(A, B)$
intersection (2 tuples)

R	A	B	S	A	B
	1	10		1	10
	2	10		2	20
	2	20			

Even More Joins

- Inner join \bowtie
 - Eq-join, theta-join, cross product, natural join
- Outer join
 - Left outer join $\bowtie\llcorner$
 - Right outer join $\lrcorner\bowtie$
 - Full outer join $\bowtie\ltimes$
- Semi join \ltimes

4. Union

$S \cup T$

The union of S and T

```
S UNION T;
```

SQL

4. Union

$S \cup T$

The union of S and T

$\text{regist} \cup \text{Bicycle} =$

`S UNION T;`

regist

user_id	model
123	Charger
567	Civic
567	Pinto

Bicycle

user_id	model
345	Schwinn
567	Sirrus

4. Union

$S \cup T$

The union of S and T

```
S UNION T;
```

regist \cup Bicycle =

regist

user_id	model
123	Charger
567	Civic
567	Pinto

Bicycle

user_id	model
345	Schwinn
567	Sirrus

Must have same schema

4. Union

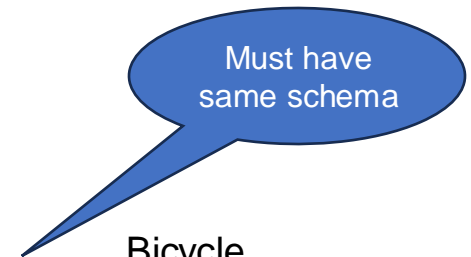
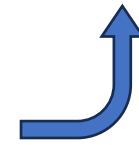
S U T

user_id	model
123	Charger
567	Civic
567	Pinto
345	Schwinn
567	Sirrus

The union of S and T

S UNION T;

regist U Bicycle =



regist

user_id	model
123	Charger
567	Civic
567	Pinto

Bicycle

user_id	model
345	Schwinn
567	Sirrus

5. Difference

$$S - T$$

The set difference of S and T

```
S EXCEPT T;
```

SQL

5. Difference

$S - T$

The set difference of S and T

`S EXCEPT T;`

`regist - Bicycle =`

regist

user_id	model
123	Charger
567	Civic
567	Pinto

Bicycle

user_id	model
345	Schwinn
567	Civic

Must have same schema

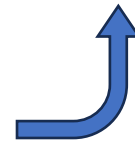
5. Difference

$S - T$

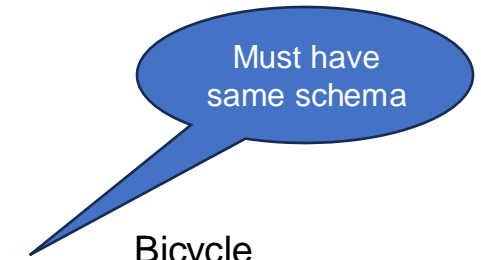
user_id	model
123	Charger
567	Pinto

The set difference of S and T

regist - Bicycle =



S **EXCEPT** T;



regist

user_id	model
123	Charger
567	Civic
567	Pinto

Bicycle

user_id	model
345	Schwinn
567	Civic

Renaming

$\rho_{attrs'}(T)$

Rename attributes

```
SELECT a1 as a1',  
        a2 as a2',  
        ...  
FROM T;
```

Renaming

 $\rho_{attrs'}(T)$

Rename attributes

```
SELECT a1 as a1',  
        a2 as a2',  
        ...  
FROM T;
```

 $\rho_{user_id,model}(regist) =$

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Renaming

 $\rho_{attrs'}(T)$

Rename attributes

```
SELECT a1 as a1',  
        a2 as a2',  
        ...  
FROM T;
```

user_id	model
123	Charger
567	Civic
567	Pinto

 $\rho_{user_id,model}(\text{regist}) =$

regist

user_id	car
123	Charger
567	Civic
567	Pinto




Renaming

 $\rho_{attrs'}(T)$

Rename attributes

```
SELECT a1 as a1',  
        a2 as a2',  
        ...  
FROM T;
```

user_id	model
123	Charger
567	Civic
567	Pinto

$\rho_{user_id,model}(\text{regist}) =$ 

regist

user_id	car
123	Charger
567	Civic
567	Pinto

Corrected union:

$\rho_{user_id,model}(\text{regist}) \cup \text{Bicycle}$

The Five Basic Relational Operators

1. Selection $\sigma_{\text{condition}}(S)$
 2. Projection $\Pi_{\text{attrs}}(S)$
 3. Join $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
 4. Union \cup
 5. Set difference $-$
- Rename ρ

Which operators are monotone?

The Five Basic Relational Operators

1. Selection $\sigma_{\text{condition}}(S)$
2. Projection $\Pi_{\text{attrs}}(S)$
3. Join $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
4. Union \cup
5. Set difference $-$

Monotone

Non-monotone

Rename ρ Monotone, but doesn't do anything

Which operators are monotone?

Query Plans

Relational Algebra Plan, or Query Plan

```
SELECT p.name  
FROM payroll p, regist r  
WHERE p.user_id = r.user_id  
       and p.job = 'TA';
```

payroll

user_id	name	job	salary	regist	
123	Jack	TA	50000	user_id	car
345	Allison	TA	60000	123	Charger
567	Magda	Prof	90000	567	Civic
789	Dan	Prof	100000	567	Pinto

Relational Algebra Plan, or Query Plan

```
SELECT p.name  
FROM payroll p, regist r  
WHERE p.user_id = r.user_id  
       and p.job = 'TA';
```



$\Pi_{\text{name}}(\sigma_{\text{job}='TA'}(\text{payroll} \bowtie \text{regist}))$

payroll


user_id	name	job	salary	regist	
123	Jack	TA	50000	user_id	car
345	Allison	TA	60000	123	Charger
567	Magda	Prof	90000	567	Civic
789	Dan	Prof	100000	567	Pinto

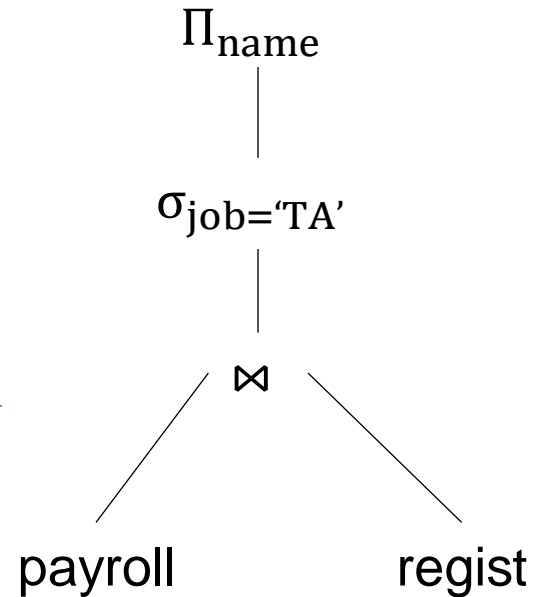
Relational Algebra Plan, or Query Plan

```
SELECT p.name  
FROM payroll p, regist r  
WHERE p.user_id = r.user_id  
and p.job = 'TA';
```



$\Pi_{\text{name}}(\sigma_{\text{job}='TA'}(\text{payroll} \bowtie \text{regist}))$

We write it as

a query plan



payroll

user_id	name	job	salary
123	Jack	TA	50000
345	Allison	TA	60000
567	Magda	Prof	90000
789	Dan	Prof	100000

regist

user_id	car
123	Charger
567	Civic
567	Pinto

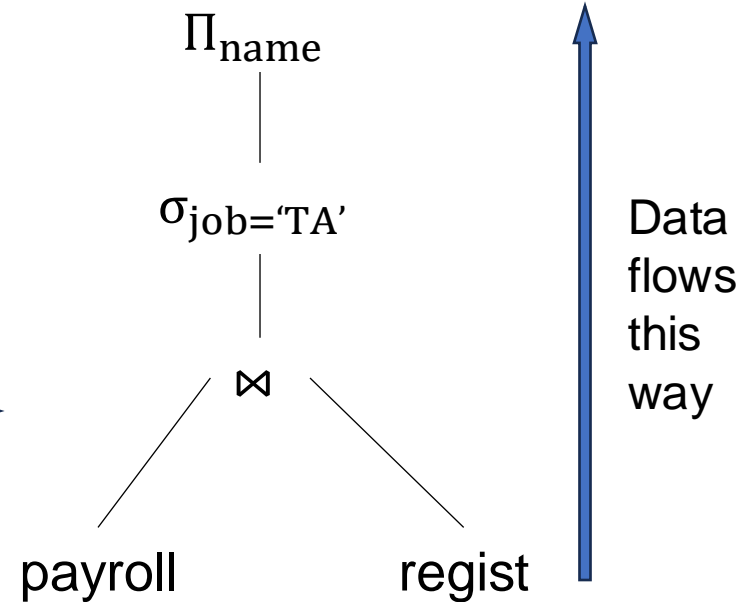
Relational Algebra Plan, or Query Plan

```
SELECT p.name
FROM payroll p, regist r
WHERE p.user_id = r.user_id
       and p.job = 'TA';
```



$\Pi_{\text{name}}(\sigma_{\text{job}='TA'}(\text{payroll} \bowtie \text{regist}))$

We write it as
a query plan

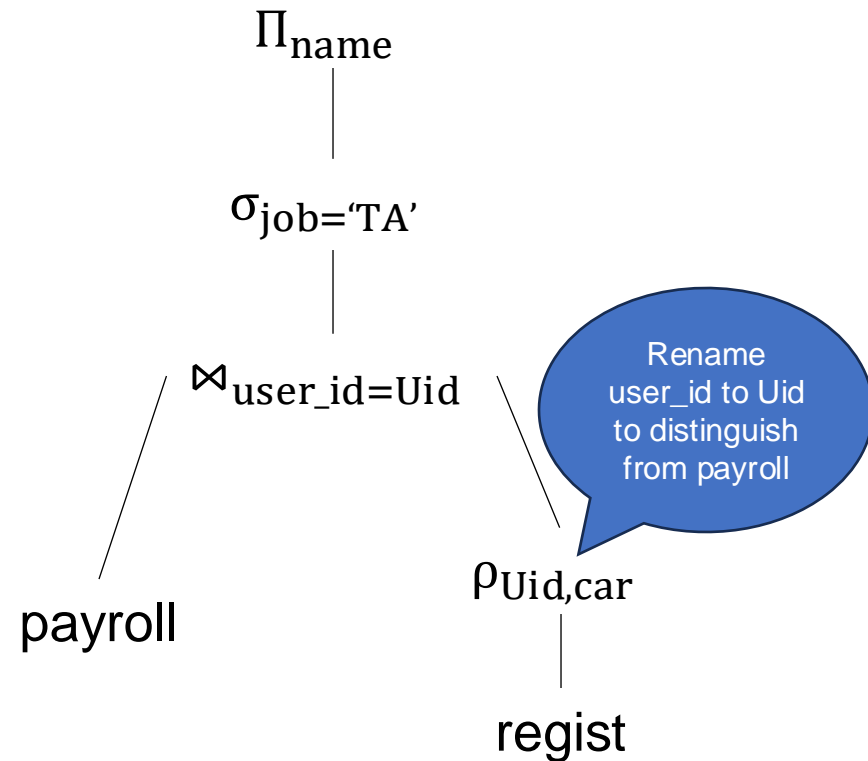


payroll

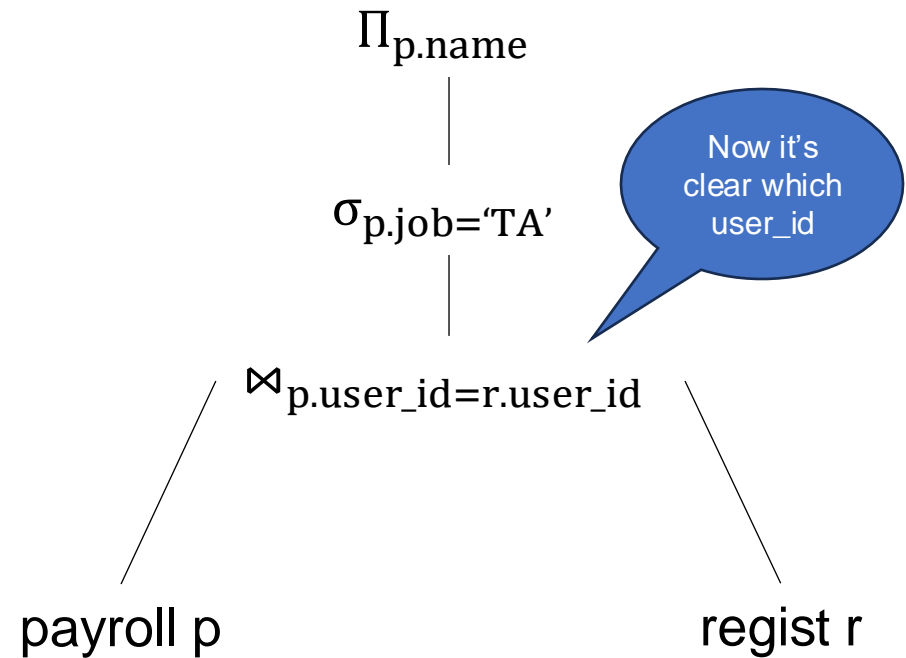
user_id	name	job	salary	regist	
123	Jack	TA	50000	user_id	car
345	Allison	TA	60000	123	Charger
567	Magda	Prof	90000	567	Civic
789	Dan	Prof	100000	567	Pinto

Query Plan: Attribute names

Managing attribute names correctly is tedious



Better: use aliases, much like in SQL



Query Plan: Execution Order

```
SELECT p.name  
FROM payroll p, regist r  
WHERE p.user_id = r.user_id  
       and p.job = 'TA';
```

We say **what** we want,
not **how** to get it

Query Plan: Execution Order

```
SELECT p.name
FROM payroll p, regist r
WHERE p.user_id = r.user_id
       and p.job = 'TA';
```

We say **what** we want,
not **how** to get it

One way
how to get it

$\Pi_{p.name}$

$\sigma_{p.job='TA'}$

$\bowtie_{p.user_id=r.user_id}$

payroll p

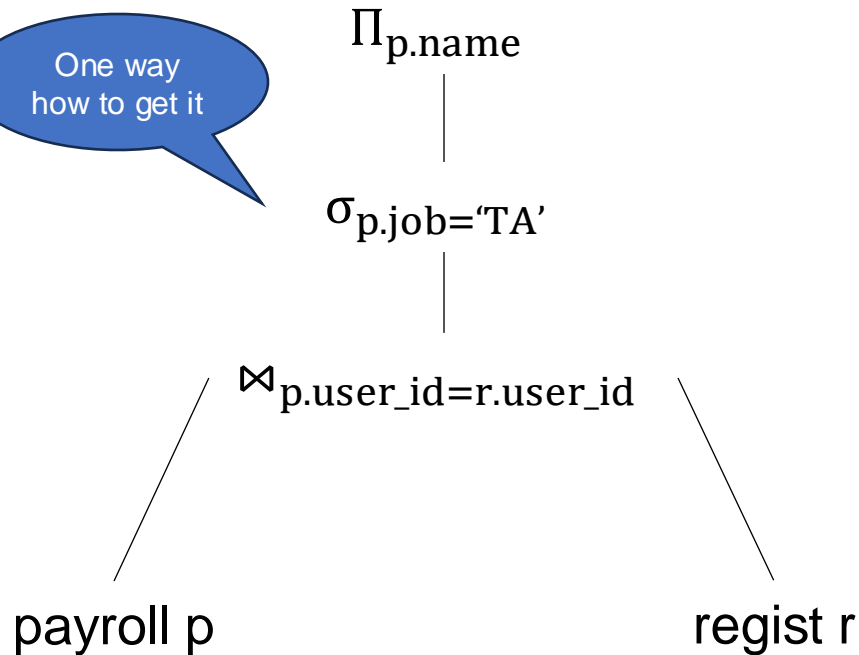
regist r

Query Plan: Execution Order

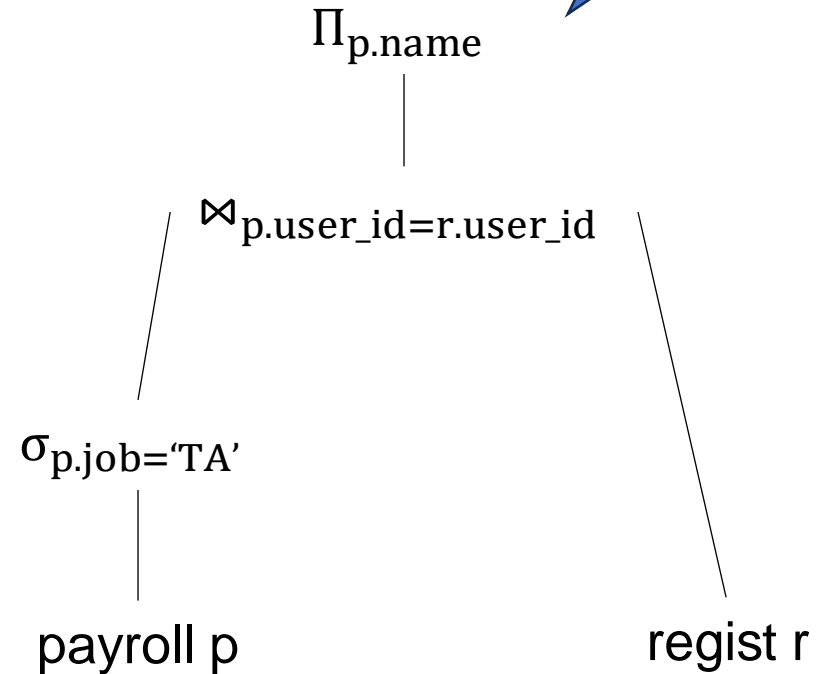
```
SELECT p.name
FROM payroll p, regist r
WHERE p.user_id = r.user_id
       and p.job = 'TA';
```

We say **what** we want,
not **how** to get it

One way
how to get it



Another way
how to get it



Query Plan: Execution Order

```
SELECT p.name  
FROM payroll p, regist r  
WHERE p.user_id = r.user_id  
and p.job = 'TA';
```

We say **what** we want,
not **how** to get it

Which one
is more
efficient?

One way
how to get it

Another way
how to get it

$\Pi_{p.name}$

$\sigma_{p.job='TA'}$

$\bowtie_{p.user_id=r.user_id}$

payroll p

regist r

$\Pi_{p.name}$

$\bowtie_{p.user_id=r.user_id}$

$\sigma_{p.job='TA'}$

payroll p

regist r

Query Plan: Execution Order

```
SELECT p.name
FROM payroll p, regist r
WHERE p.user_id = r.user_id
       and p.job = 'TA';
```

We say **what** we want,
not **how** to get it

One way
how to get it

$\Pi_{p.name}$

$\sigma_{p.job='TA'}$

$\bowtie_{p.user_id=r.user_id}$

payroll p

regist r

Which one
is more
efficient?

$\Pi_{p.name}$

$\bowtie_{p.user_id=r.user_id}$

$\sigma_{p.job='TA'}$

payroll p

regist r

Another way
how to get it

Most likely
this one

Discussion

- Database system converts a SQL query to a Relational Algebra Plan

Discussion

- Database system converts a SQL query to a Relational Algebra Plan
- Then it optimizes the plan by exploring equivalent plans, using simple algebraic identities:

$$R \bowtie S = S \bowtie R$$

$$R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$$

$$\sigma_{\theta}(R \bowtie S) = \sigma_{\theta}(R) \bowtie S$$

... many others*

*over 500 rules in SQL Server

Discussion

- Database system converts a SQL query to a Relational Algebra Plan
- Then it optimizes the plan by exploring equivalent plans, using simple algebraic identities:
 - $R \bowtie S = S \bowtie R$
 - $R \bowtie (S \bowtie T) = (R \bowtie S) \bowtie T$
 - $\sigma_{\theta}(R \bowtie S) = \sigma_{\theta}(R) \bowtie S$
 - ... many others*
- Next lecture: how to convert SQL to RA plan

*over 500 rules in SQL Server