DATA 514
Lecture 3

SQL Wrap-up

Relational Algebra

Announcements

- HW2 – deadline extended until tomorrow
- WQ3 is open, due on Tuesday

- Homework 3 will be posted tomorrow, due on
- Feb 3
  - We are using Microsoft Azure Cloud services!
  - Wait for instructions to be posted

Recap from last lectures

- Subqueries can occur in every clause:
  - SELECT
  - FROM
  - WHERE
- Monotone queries: SELECT-FROM-WHERE
  - Existential quantifier
- Non-monotone queries
  - Universal quantifier
  - Aggregation

Examples of Complex Queries

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

1. Find drinkers that frequent <u>some</u> bar that serves <u>some</u> beer they like.

2. Find drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they don’t like.

3. Find drinkers that frequent <u>only</u> bars that serves <u>some</u> beer they like.

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 1

Find drinkers that frequent <u>some</u> bar that serves <u>some</u> beer they like.

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 1

Find drinkers that frequent some bar that serves some beer they like.

```
SELECT DISTINCT X.drinker
FROM Frequents X, Serves Y, Likes Z
WHERE X.bar = Y.bar
AND Y.beer = Z.beer
AND X.drinker = Z.drinker
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 1

Find drinkers that frequent some bar that serves some beer they like.

```
SELECT DISTINCT X.drinker
FROM Frequents X, Serves Y, Likes Z
WHERE X.bar = Y.bar
AND Y.beer = Z.beer
AND X.drinker = Z.drinker
```

What happens if we didn't write DISTINCT?

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 2

Find drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they don’t like

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 2

Find drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they don’t like

Let’s check if the drinker frequents one of the other bars

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 2

Find drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they don't like

Let's check if the drinker frequents one of the other bars

Drinkers that frequent <u>some</u> bars that serves <u>some</u> beer they like.

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 2

Find drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they don't like

Let's check if the drinker frequents one of the other bars

Drinkers that frequent <u>some</u> bars that serves <u>some</u> beer they like.

That's the previous query… but let's write it with a subquery:

```
SELECT DISTINCT X.drinker
FROM Frequents X
WHERE       EXISTS (SELECT *
                    FROM Serves Y, Likes Z
                    WHERE X.bar=Y.bar
                          AND X.drinker=Z.drinker
                          AND Y.beer = Z.beer)
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 2

Find drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they don't like

Let's check if the drinker frequents one of the other bars

Drinkers that frequent <u>some</u> bars that serves <u>some</u> beer they like.

That's the previous query… but let's write it with a subquery:

Now negate!

```
SELECT DISTINCT X.drinker
FROM Frequents X
WHERE NOT EXISTS (SELECT *
                  FROM Serves Y, Likes Z
                  WHERE X.bar=Y.bar
                       AND X.drinker=Z.drinker
                       AND Y.beer = Z.beer)
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 3

Find drinkers that frequent <u>only</u> bars that serves <u>some</u> beer they like.

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 3

Find drinkers that frequent <u>only</u> bars that serves <u>some</u> beer they like.

Let's find the other drinkers

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 3

Find drinkers that frequent only bars that serves some beer they like.

Let's find the other drinkers

Drinkers that frequent some bar that serves only beers they don't like

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 3

Find drinkers that frequent only bars that serves some beer they like.

Let's find the other drinkers

Drinkers that frequent some bar that serves only beers they don't like

That's the previous query!

```
SELECT X.drinker
FROM Frequents X
WHERE NOT EXISTS (SELECT *
                  FROM Serves Y, Likes Z
                  WHERE X.bar=Y.bar
                        AND X.drinker=Z.drinker
                        AND Y.beer = Z.beer
```

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Example 3

Find drinkers that frequent <u>only</u> bars that serves <u>some</u> beer they like.

Let's find the other drinkers

Drinkers that frequent <u>some</u> bar that serves <u>only</u> beers they don't like

That's the previous query!

Now find the other drinkers:

```
SELECT DISTINCT U.drinker
FROM Frequents U
WHERE U.drinker NOT IN
      (SELECT X.drinker
       FROM Frequents X
       WHERE NOT EXISTS (SELECT *
                         FROM Serves Y, Likes Z
                         WHERE X.bar=Y.bar
                               AND X.drinker=Z.drinker
                               AND Y.beer = Z.beer)
```

Product (pname, price, cid)

Company(cid, cname, city)

Unnesting Aggregates

Find the number of companies in each city

Product (pname, price, cid)

Company(cid, cname, city)

Unnesting Aggregates

Find the number of companies in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
                          FROM Company Y
                          WHERE X.city = Y.city)
FROM  Company X
```

Product (pname, price, cid)

Company(cid, cname, city)

Unnesting Aggregates

Find the number of companies in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
                         FROM Company Y
                         WHERE X.city = Y.city)
FROM  Company X
```

```
SELECT city,  count(*)
FROM   Company
GROUP BY city
```

Equivalent queries

Product (pname, price, cid)

Company(cid, cname, city)

Unnesting Aggregates

Find the number of companies in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
                         FROM Company Y
                         WHERE X.city = Y.city)
FROM  Company X
```

```
SELECT city,  count(*)
FROM   Company
GROUP BY city
```

Equivalent queries

Note: no need for DISTINCT
(DISTINCT *is the same* as GROUP BY)

Product (pname, price, cid)

Company(cid, cname, city)

Unnesting Aggregates

Find the number of products made in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
                         FROM Product Y, Company Z
                         WHERE Z.cid=Y.cid
                         AND Z.city = X.city)
FROM  Company X
```

Product (<u>pname</u>, price, cid)

Company(<u>cid</u>, cname, city)

Unnesting Aggregates

Find the number of products made in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
                         FROM Product Y, Company Z
                         WHERE Z.cid=Y.cid
                         AND Z.city = X.city)
FROM  Company X
```

```
SELECT X.city, count(*)
FROM Company X, Product Y
WHERE X.cid=Y.cid
GROUP BY X.city
```

NOT equivalent !

You should know why!

Purchase(pid, product, quantity, price)

Unnesting Aggregates

```
SELECT     product, Sum(quantity) AS TotalSales
FROM       Purchase
WHERE      price > 1
GROUP BY   product
```

Purchase(pid, product, quantity, price)

Unnesting Aggregates

```
SELECT      product, Sum(quantity) AS TotalSales
FROM        Purchase
WHERE       price > 1
GROUP BY    product
```

```
SELECT DISTINCT  x.product, (SELECT Sum(y.quantity)
                             FROM    Purchase y
                             WHERE x.product = y.product
                             AND y.price > 1)
                             AS TotalSales
FROM        Purchase x
WHERE       x.price > 1
```

Purchase(pid, product, quantity, price)

Unnesting Aggregates

```
SELECT      product, Sum(quantity) AS TotalSales
FROM        Purchase
WHERE       price > 1
GROUP BY    product
```

```
SELECT DISTINCT  x.product, (SELECT Sum(y.quantity)
                             FROM    Purchase y
                             WHERE x.product = y.product
                             AND y.price > 1)
                             AS TotalSales
FROM        Purchase x
WHERE       x.price > 1
```

Why twice ?

Author(login,name)

Wrote(login,url)

More Unnesting

Find authors who wrote ≥ 100 documents:

Attempt 1: with nested queries

```
SELECT x.login, x.name
FROM Author x,
        (SELECT login, count(*) as c
         FROM Wrote
         GROUP BY login) y
WHERE x.login = y.login and y.c > 100
```

This is SQL by a novice

Author(login,name)
Wrote(login,url)

More Unnesting

Find authors who wrote ≥ 100 documents:

Attempt 1: with nested queries

Attempt 2: using GROUP BY and HAVING

```
SELECT      Author.login, Author.name
FROM        Author, Wrote
WHERE       Author.login=Wrote.login
GROUP BY    Author.login, Author.name
HAVING      count(wrote.url) >= 100
```

This is SQL by an expert

Product (pname, price, cid)

Company(cid, cname, city)

Finding Witnesses

For each city, find the most expensive product made in that city

Product (pname, price, cid)

Company(cid, cname, city)

Finding Witnesses

For each city, find the most expensive product made in that city

Finding the maximum price is easy…

```
SELECT x.city, max(y.price)
FROM Company x, Product y
WHERE x.cid = y.cid
GROUP BY x.city;
```

But we need the *witnesses*, i.e. the products with max price

Product (pname, price, cid)

Company(cid, cname, city)

Finding Witnesses

To find the witnesses, compute the maximum price in a subquery

```
SELECT DISTINCT u.city, v.pname, v.price
FROM Company u, Product v,
    (SELECT x.city, max(y.price) as maxprice
     FROM Company x, Product y
     WHERE x.cid = y.cid
     GROUP BY x.city) w
WHERE u.cid = v.cid
      and u.city = w.city
      and v.price=w.maxprice;
```

Product (<u>pname</u>, price, cid)

Company(<u>cid</u>, cname, city)

Finding Witnesses

To find the witnesses, compute the maximum price in a subquery

Or using the <u>*with*</u> clause:

```
WITH MaxPrices AS
   (SELECT x.city, max(y.price) as maxprice
    FROM Company x, Product y
    WHERE x.cid = y.cid
    GROUP BY x.city)
SELECT DISTINCT u.city, v.pname, v.price
FROM Company u, Product v, MaxPrices w
WHERE u.cid = v.cid
     and u.city = w.city
     and v.price=w.maxprice;
```

Product (pname, price, cid)

Company(cid, cname, city)

Finding Witnesses

Or we can use a subquery in where clause

```
SELECT u.city, v.pname, v.price
FROM Company u, Product v
WHERE u.cid = v.cid
  and v.price >= ALL (SELECT y.price
                      FROM Company x, Product y
                      WHERE u.city=x.city
                        and x.cid=y.cid);
```

Product (pname, price, cid)

Company(cid, cname, city)

Finding Witnesses

There is a more concise solution here:

```
SELECT u.city, v.pname, v.price
FROM Company u, Product v, Company x, Product y
WHERE u.cid = v.cid and u.city = x.city and x.cid = y.cid
GROUP BY u.city, v.pname, v.price
HAVING v.price = max(y.price);
```

Summary of SQL

- What you learn from this class:
  - Write complex SQL queries (done)
  - Tune the database, create indices
  - Define constraints
- What you don’t learn in this class
  - The rest of the SQL ecosystem
  - Learn-as-you go (manual, google)

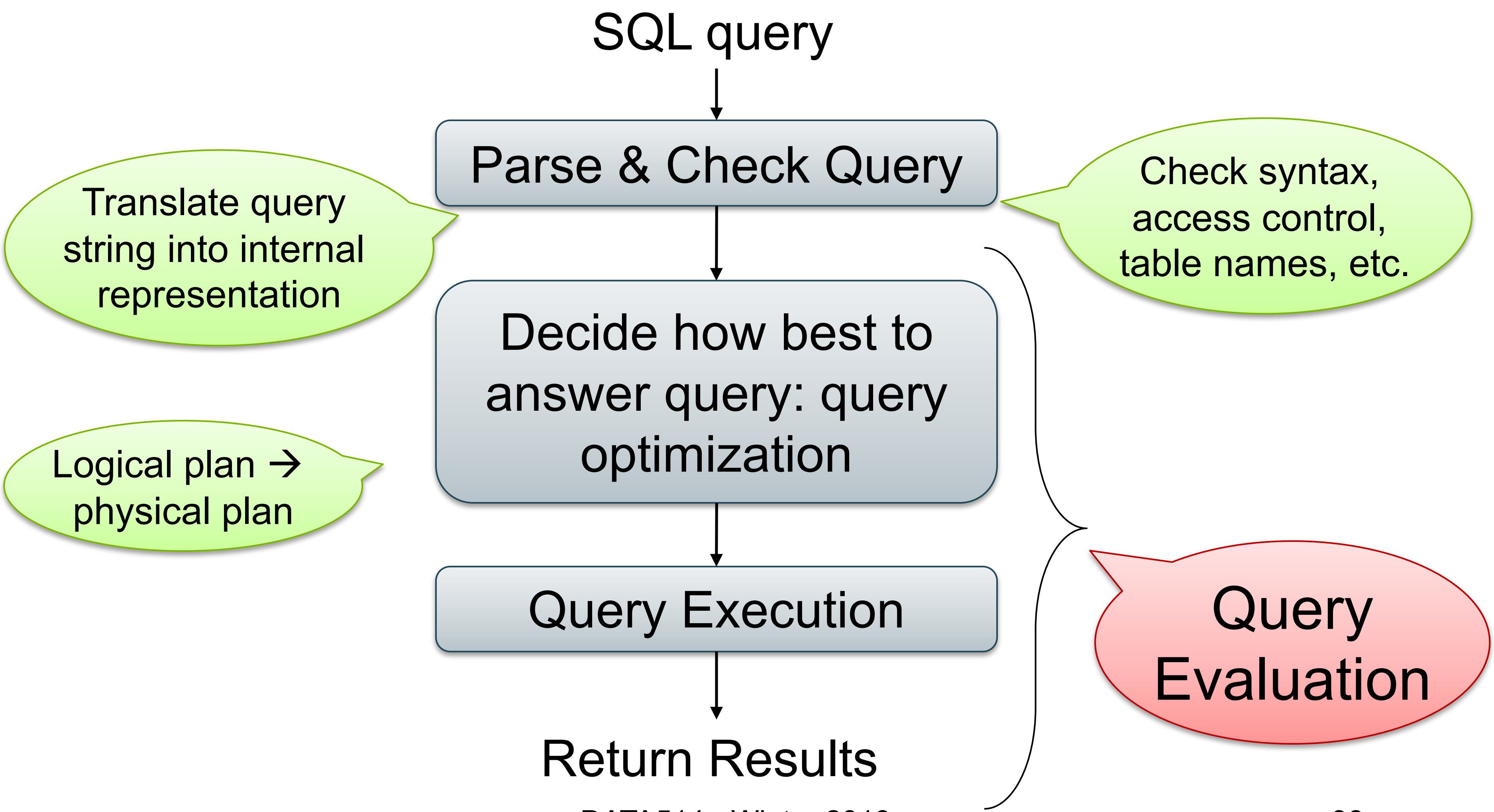
Relational Algebra

Where We Are

- Motivation for using a DBMS for managing data
- SQL:
  - Declaring the schema for our data (CREATE TABLE)
  - Inserting data one row at a time or in bulk (INSERT/.import)
  - Modifying the schema and updating the data (ALTER/UPDATE)
  - Querying the data (SELECT)

- Next step: More knowledge of how DBMSs work
  - Relational algebra and query execution
  - Client-server architecture

Query Evaluation Steps



The WHAT and the HOW

- SQL = WHAT we want to get form the data
- Relational Algebra = HOW to get the data we want
- The passage from WHAT to HOW is called query optimization
  - SQL -> Relational Algebra -> Physical Plan
  - Relational Algebra = Logical Plan

Relational Algebra

Sets v.s. Bags

So far, we have said that relational algebra and SQL operate on relations that are sets of tuples.

- Sets: {a,b,c}, {a,d,e,f}, { }, . . .
- Bags: {a, a, b, c}, {b, b, b, b, b}, . . .

Relational Algebra has two semantics:

- Set semantics = standard Relational Algebra
- Bag semantics = extended Relational Algebra

DB systems implement bag semantics (Why?)

Relational Algebra Operators

- Union $\cup$, intersection $\cap$, difference $-$
- Selection σ
- Projection Π
- Cartesian product $\times$, join $\bowtie$
- Rename ρ
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ

RA

Extended RA

Union and Difference

$$R1 \cup R2$$
$$R1 - R2$$

What do they mean over bags ?

What about Intersection ?

- Derived operator using minus

$$\boxed{R1 \cap R2 = R1 - (R1 - R2)}$$

- Derived using join (will explain later)

$$\boxed{R1 \cap R2 = R1 \bowtie R2}$$

Selection

- Returns all tuples which satisfy a condition

$$\boxed{\sigma_c(R)}$$

- Examples
  - $\sigma_{\text{Salary > 40000}}$ (Employee)
  - $\sigma_{\text{name = “Smith”}}$ (Employee)
- The condition c can be =, <, ≤, >, ≥, <> combined with AND, OR, NOT

Employee

SSN	Name	Salary
1234545	John	20000
5423341	Smith	60000
4352342	Fred	50000

$\sigma_{\text{Salary} > 40000}$ (Employee)

SSN	Name	Salary
5423341	Smith	60000
4352342	Fred	50000

Projection

- Eliminates columns

$$\boxed{\Pi_{A1,\ldots,An}(R)}$$

- Example: project social-security number and names:
  - $\Pi_{\text{SSN, Name}}$ (Employee)
  - Answer(SSN, Name)

Different semantics over sets or bags! Why?

Employee

SSN	Name	Salary
1234545	John	20000
5423341	John	60000
4352342	John	20000

$\Pi_{\text{Name,Salary}}$ (Employee)

Name	Salary
John	20000
John	60000
John	20000

Bag semantics

Name	Salary
John	20000
John	60000

Set semantics

Which is more efficient?

Composing RA Operators

Patient

no	name	zip	disease
1	p1	98125	flu
2	p2	98125	heart
3	p3	98120	lung
4	p4	98120	heart

$\pi_{\text{zip,disease}}(\text{Patient})$

zip	disease
98125	flu
98125	heart
98120	lung
98120	heart

$\sigma_{\text{disease='heart'}}(\text{Patient})$

no	name	zip	disease
2	p2	98125	heart
4	p4	98120	heart

$\pi_{\text{zip,disease}}\ (\sigma_{\text{disease='heart'}}(\text{Patient}))$

zip	disease
98125	heart
98120	heart

Cartesian Product

- Each tuple in R1 with each tuple in R2

$$\boxed{R1 \times R2}$$

- Rare in practice; mainly used to express joins

Cross-Product Example

Employee

Name	SSN
John	999999999
Tony	777777777

Dependent

EmpSSN	DepName
999999999	Emily
777777777	Joe

Employee $\times$ Dependent

Name	SSN	EmpSSN	DepName
John	999999999	999999999	Emily
John	999999999	777777777	Joe
Tony	777777777	999999999	Emily
Tony	777777777	777777777	Joe

Renaming

- Changes the schema, not the instance

$$\rho_{B1,\ldots,Bn}(R)$$

- Example:
  - R = $\rho_{N,\,S}$(Employee) makes R be a relation with attributes N, S and the same tuples as Employee.

Not really used by systems, but needed on paper

Natural Join

$$\boxed{R1 \bowtie R2}$$

- Meaning: $R1 \bowtie R2 = \Pi_A(\sigma_\theta(R1 \times R2))$
- Where:
  - Selection σ checks equality of all common attributes (attributes with same names)
  - Projection eliminates duplicate common attributes

Natural Join Example

R

A	B
X	Y
X	Z
Y	Z
Z	V

S

B	C
Z	U
V	W
Z	V

R ⋈ S =

$\Pi_{ABC}(\sigma_{R.B=S.B}(R \times S))$

A	B	C
X	Z	U
X	Z	V
Y	Z	U
Y	Z	V
Z	V	W

Natural Join Example 2

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie V$

age	zip	disease	name
54	98125	heart	p1
20	98120	flu	p2

Natural Join

- Given schemas R(A, B, C, D), S(A, C, E), what is the schema of R $\bowtie$ S ?
- Given R(A, B, C), S(D, E), what is R $\bowtie$ S ?
- Given R(A, B), S(A, B), what is R $\bowtie$ S ?

AnonPatient (age, zip, disease)
Voters (name, age, zip)

Theta Join

- A join that involves a predicate

$$R1 \bowtie_{\theta} R2 \;=\; \sigma_{\theta}\,(R1 \times R2)$$

- Here θ can be any condition
- For our voters/patients example:

$$P \bowtie_{\text{P.zip = V.zip and P.age >= V.age -1 and P.age <= V.age +1}} V$$

Equijoin

- A theta join where θ is an equality predicate
- By far the most used variant of join in practice

Equijoin Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie_{P.age=V.age} V$

P.age	P.zip	P.disease	P.name	V.zip	V.age
54	98125	heart	p1	98125	54
20	98120	flu	p2	98120	20

Join Summary

- **Theta-join**: $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
  - Join of R and S with a join condition θ
  - Cross-product followed by selection θ
- **Equijoin**: $R \bowtie_{\theta} S = \pi_A (\sigma_{\theta}(R \times S))$
  - Join condition θ consists only of equalities
- **Natural join**: $R \bowtie S = \pi_A (\sigma_{\theta}(R \times S))$
  - Equijoin
  - Equality on **all** fields with same name in R and in S
  - Projection π_A drops all redundant attributes

So Which Join Is It ?

When we write R $\bowtie$ S we usually mean an equijoin, but we often omit the equality predicate when it is clear from the context

More Joins

- **Outer join**
  - Include tuples with no matches in the output
  - Use NULL values for missing attributes
  - Does not eliminate duplicate columns

- Variants
  - Left outer join
  - Right outer join
  - Full outer join

Outer Join Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu
33	98120	lung

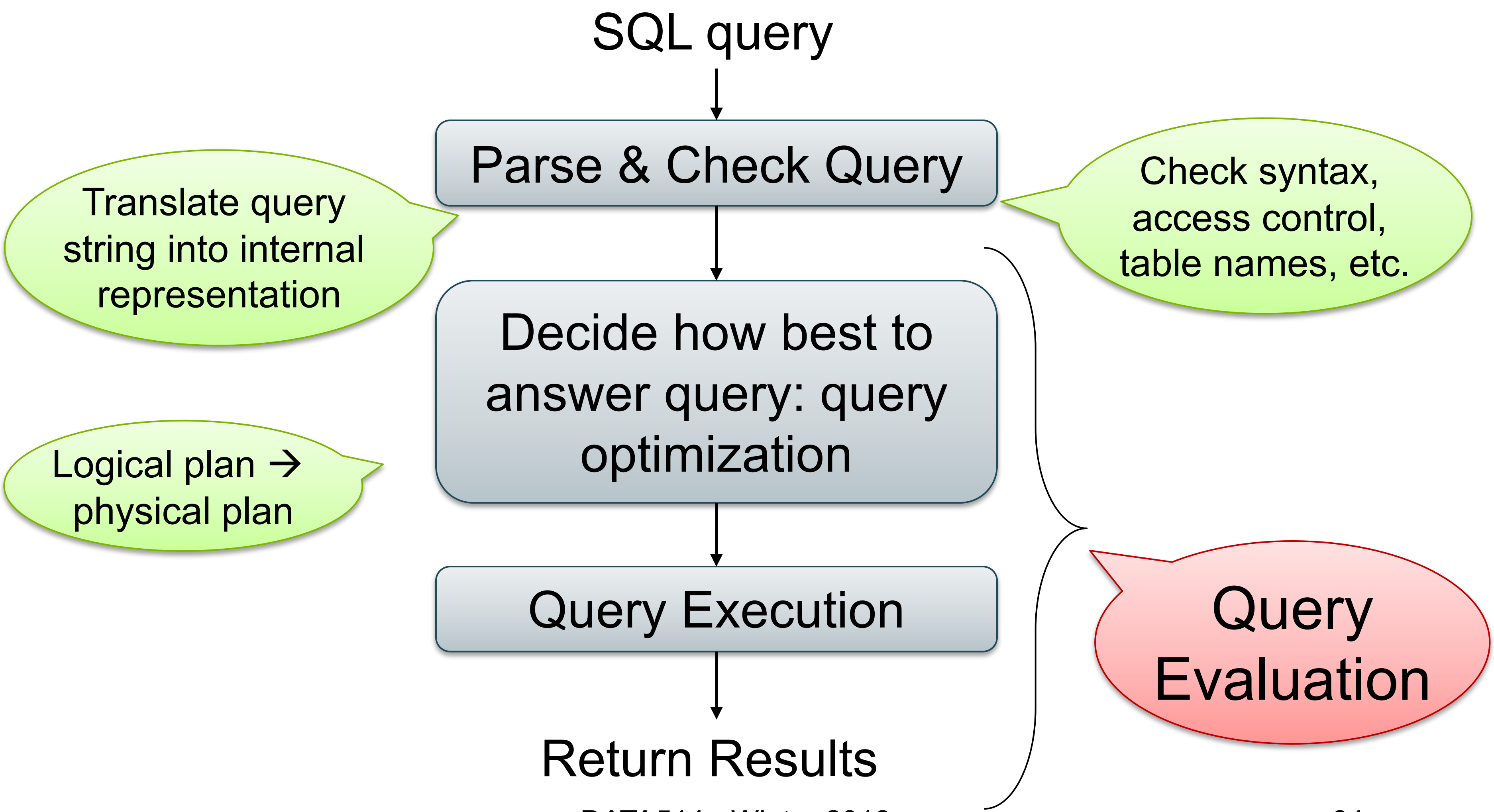
AnnonJob J

job	age	zip
lawyer	54	98125
cashier	20	98120

P ⟕ J

P.age	P.zip	disease	job	J.age	J.zip
54	98125	heart	lawyer	54	98125
20	98120	flu	cashier	20	98120
33	98120	lung	null	33	98120

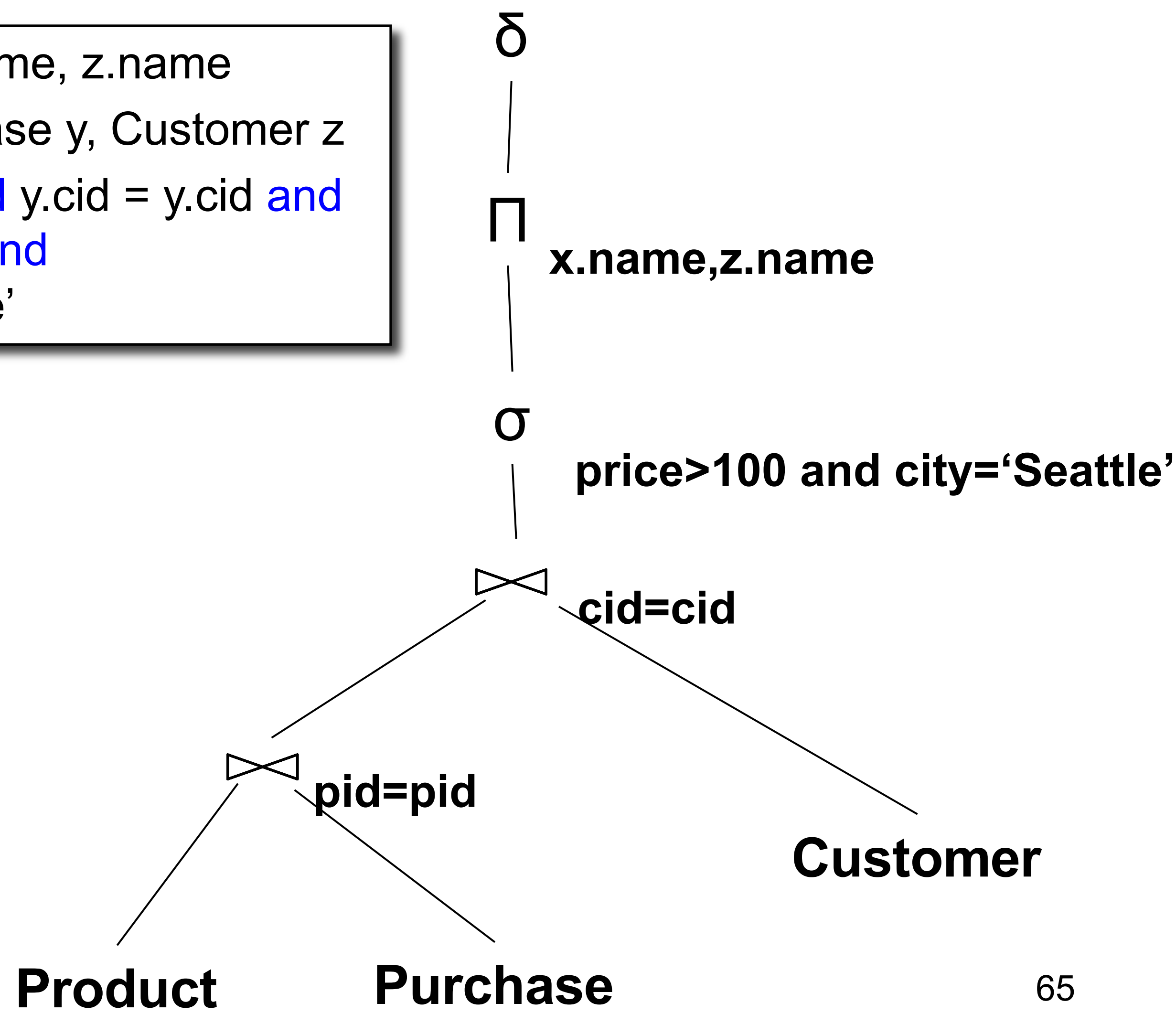
Query Evaluation Steps



Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

From SQL to RA

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
          x.price > 100 and
         z.city =  ‘Seattle’
```

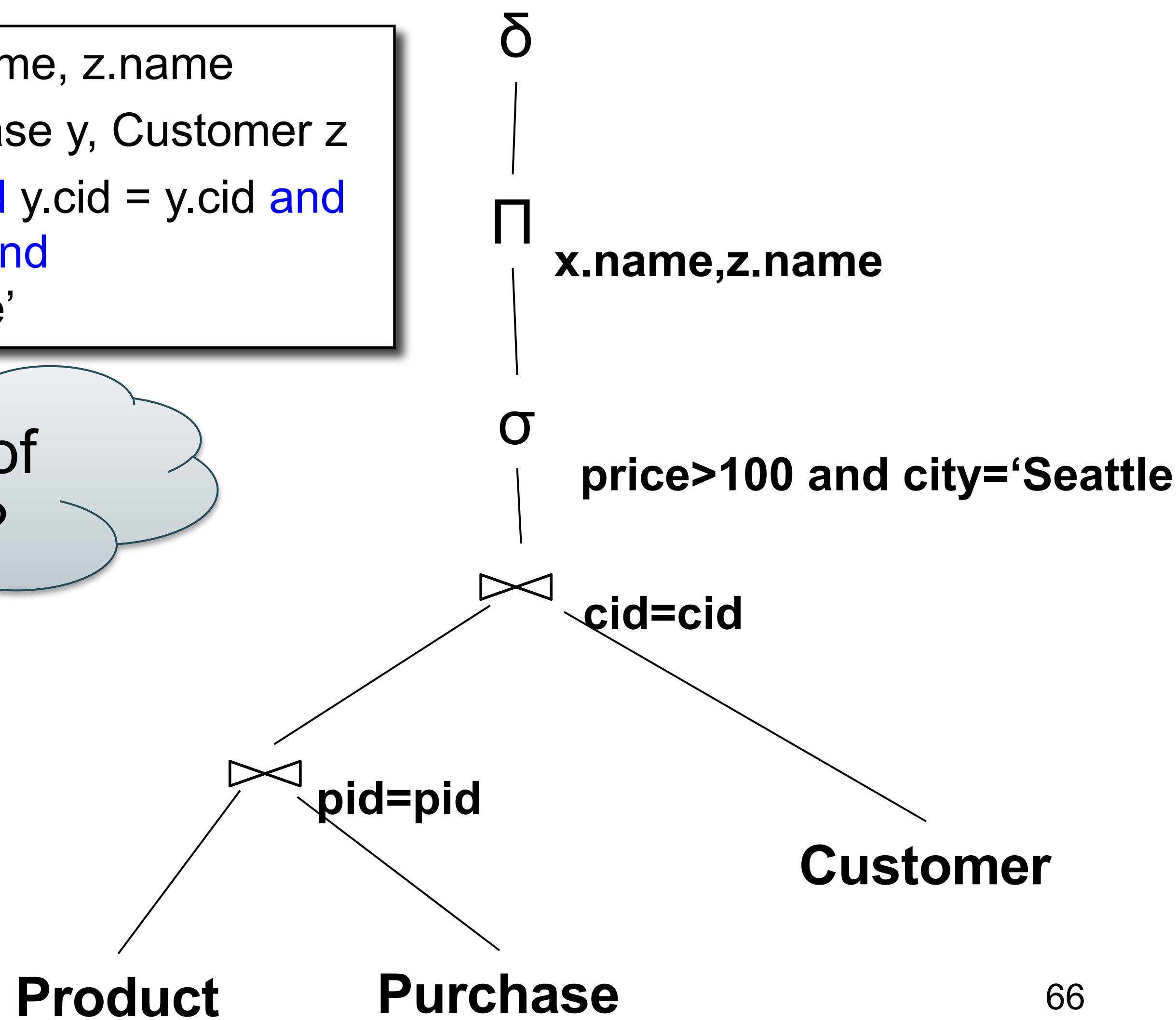


Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

From SQL to RA

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
          x.price > 100 and
          z.city = 'Seattle'
```

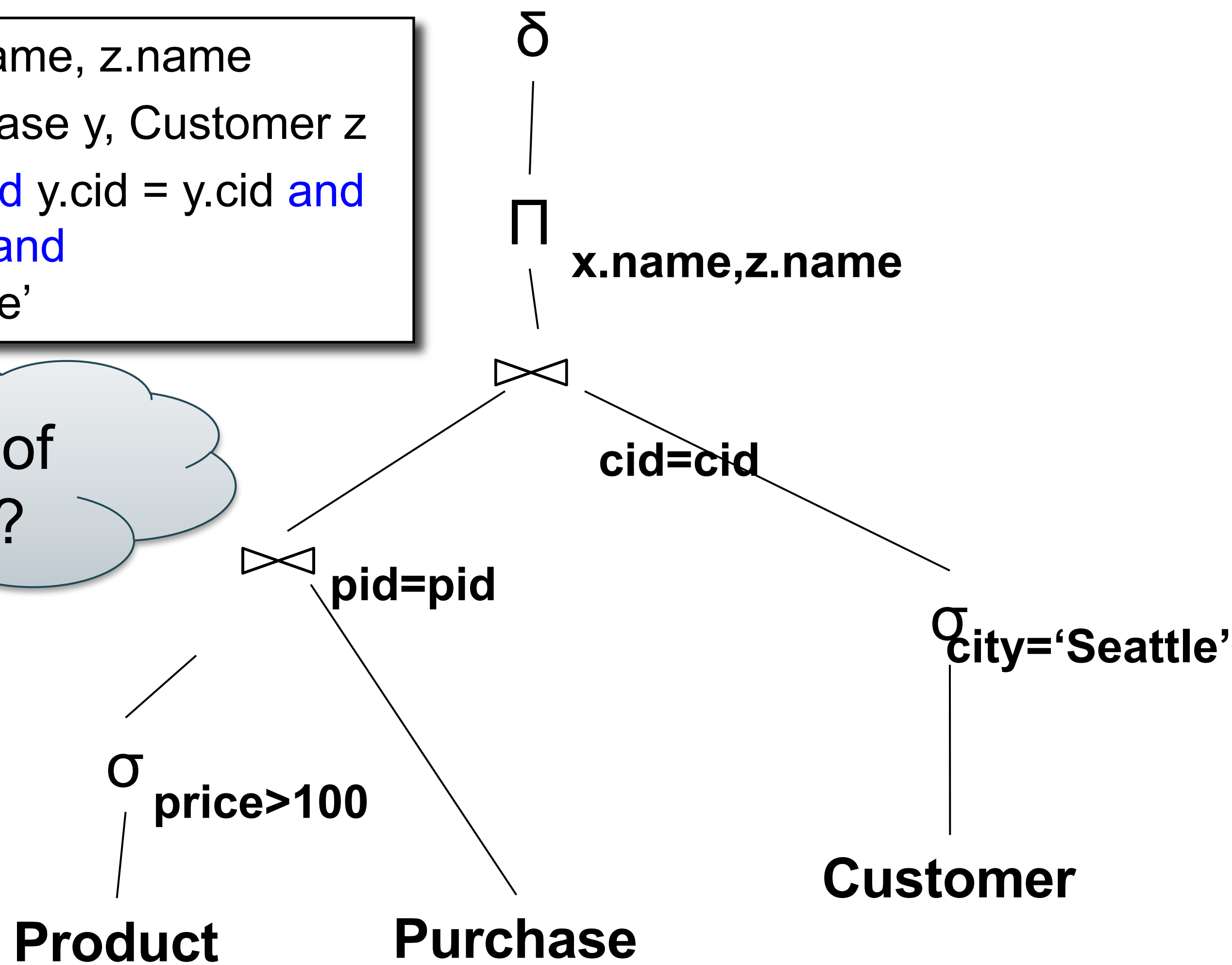
Can you think of a "better" plan?



Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

From SQL to RA

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
          x.price > 100 and
          z.city = 'Seattle'
```



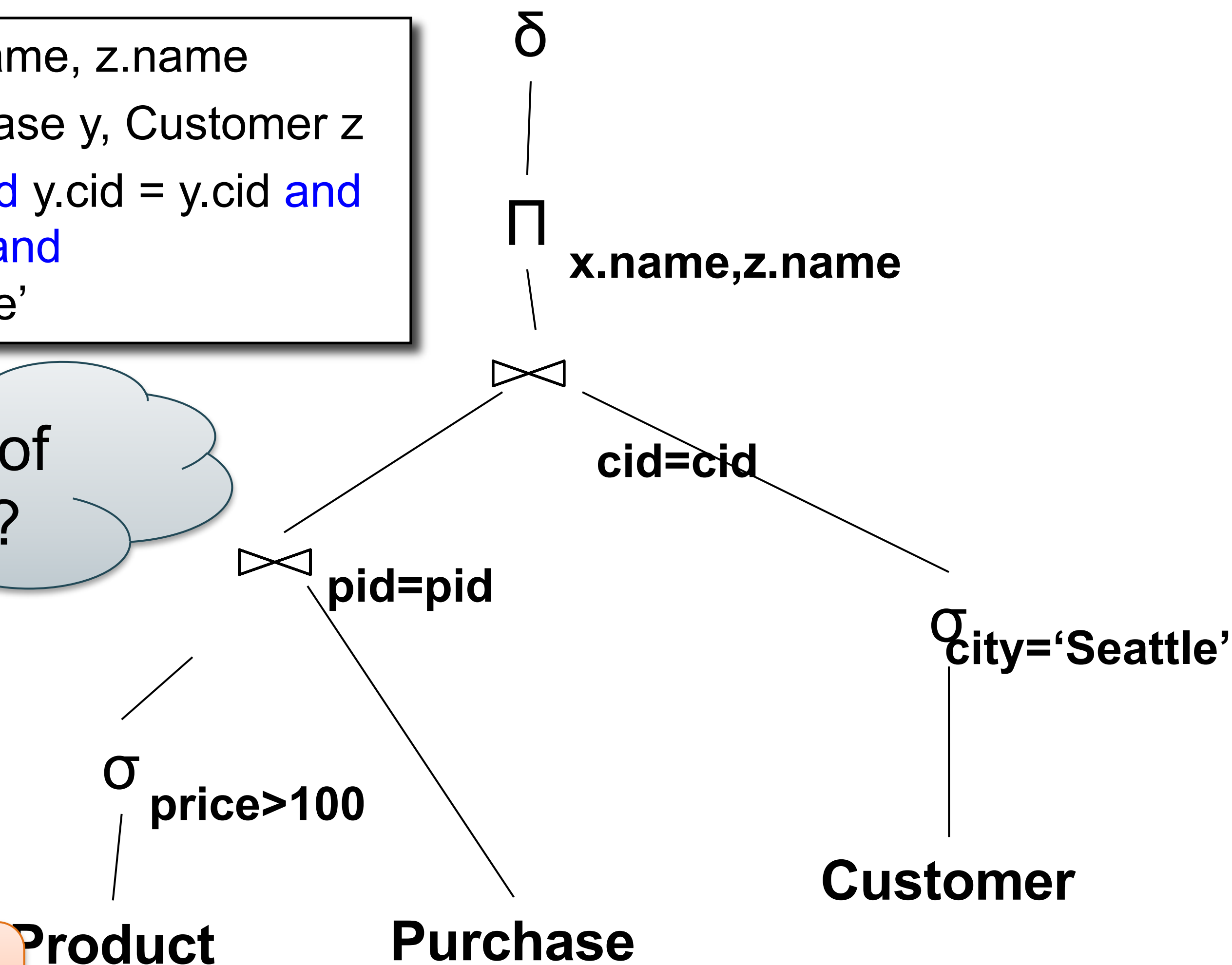
Can you think of a "better" plan?

Push selections down the query plan!

Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

From SQL to RA

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
          x.price > 100 and
          z.city = 'Seattle'
```



Can you think of a "better" plan?

Push selections down the query plan!

Query optimization: find an equivalent optimal plan

Extended RA: Operators on Bags

- Duplicate elimination δ
- Grouping γ
- Sorting τ

Logical Query Plan

```
SELECT city, count(*)
FROM sales
GROUP BY city
HAVING sum(price) > 100
```

T3(city, c)

$\Pi_{\text{city, c}}$

T2(city,p,c)

$\sigma_{\text{p > 100}}$

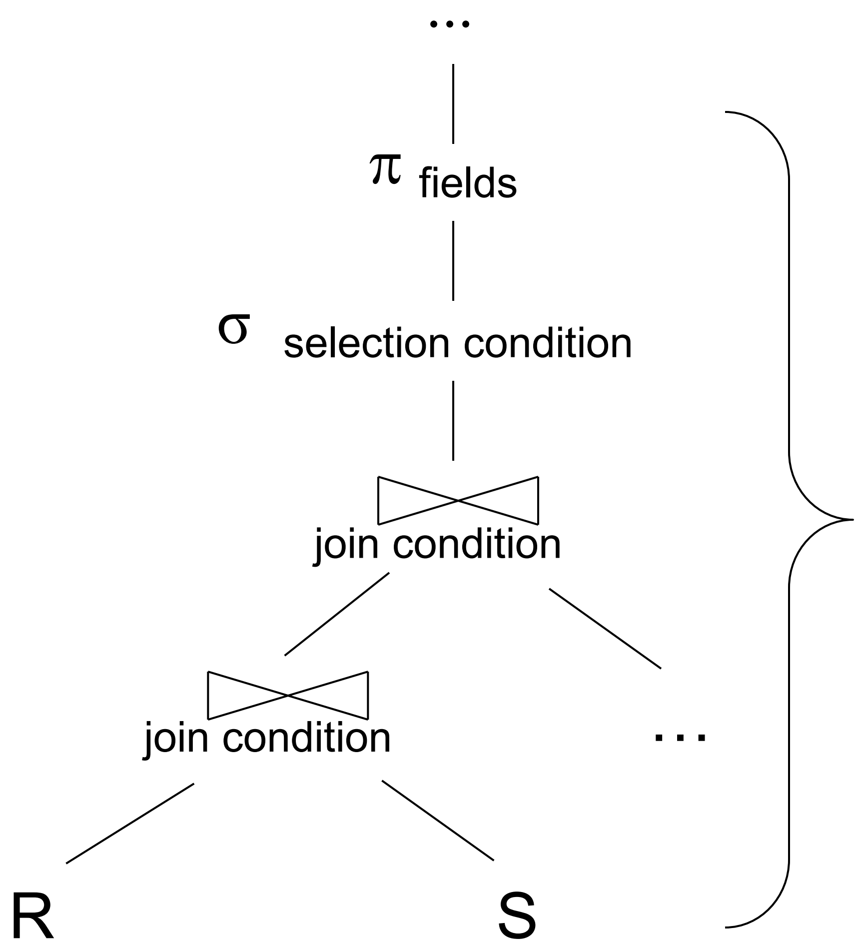
T1(city,p,c)

$\gamma_{\text{city, sum(price)} \rightarrow \text{p, count(*)} \rightarrow \text{c}}$

sales(product, city, price)

T1, T2, T3 = temporary tables

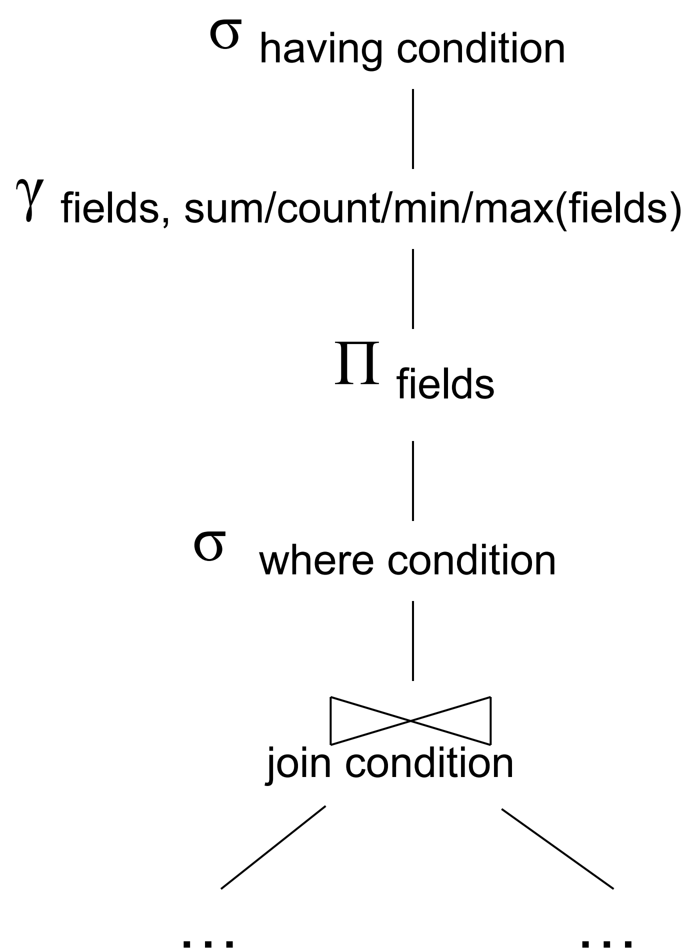
Typical Plan for Block (1/2)



SELECT fields
FROM R, S, …
WHERE condition

SELECT-PROJECT-JOIN
Query

Typical Plan For Block (2/2)



SELECT fields
FROM R, S, …
WHERE condition
GROUP BY fields
HAVING condition

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)

How about Subqueries?

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
   and not exists
     (SELECT *
      FROM Supply P
      WHERE P.sno = Q.sno
           and P.price > 100)
```

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)

How about Subqueries?

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
   and not exists
     (SELECT *
      FROM Supply P
      WHERE P.sno = Q.sno
          and P.price > 100)
```

Correlation !

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)

How about Subqueries?

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
    and not exists
      (SELECT *
       FROM Supply P
       WHERE P.sno = Q.sno
            and P.price > 100)
```

De-Correlation

```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
    and Q.sno not in
      (SELECT P.sno
       FROM Supply P
       WHERE P.price > 100)
```

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)

How about Subqueries?

Un-nesting

```
(SELECT  Q.sno
 FROM Supplier Q
 WHERE  Q.sstate = 'WA')
   EXCEPT
 (SELECT P.sno
  FROM Supply P
  WHERE P.price > 100)
```

EXCEPT = set difference

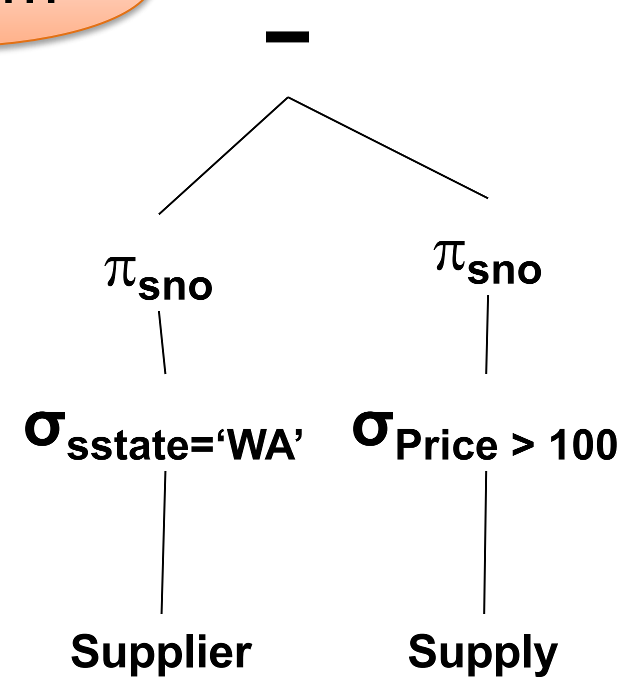
```
SELECT  Q.sno
FROM Supplier Q
WHERE  Q.sstate = 'WA'
    and Q.sno not in
      (SELECT P.sno
       FROM Supply P
       WHERE P.price > 100)
```

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)

How about Subqueries?

```
(SELECT  Q.sno
 FROM Supplier Q
 WHERE  Q.sstate = 'WA')
   EXCEPT
 (SELECT P.sno
   FROM Supply P
   WHERE P.price > 100)
```

Finally…



From Logical Plans to Physical Plans

Physical Operators

Each of the logical operators may have one or more implementations = physical operators

Will discuss several basic physical operators, with a focus on join

Product(<u>pid</u>, name, price)
Purchase(<u>pid, cid</u>, store)

Main Memory Algorithms

Logical operator:

Product(<u>pid</u>, name, price) $\bowtie_{pid=pid}$ Purchase(<u>pid, cid</u>, store)

Propose three physical operators for the join, assuming the tables are in main memory:

1.
2.
3.

Main Memory Algorithms

Logical operator:

Product(pid, name, price) $\bowtie_{pid=pid}$ Purchase(pid, cid, store)

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join O(??)
2. Merge join O(??)
3. Hash join O(??)

Main Memory Algorithms

Logical operator:

Product(pid, name, price) $\bowtie_{pid=pid}$ Purchase(pid, cid, store)

Propose three physical operators for the join, assuming the tables are in main memory:

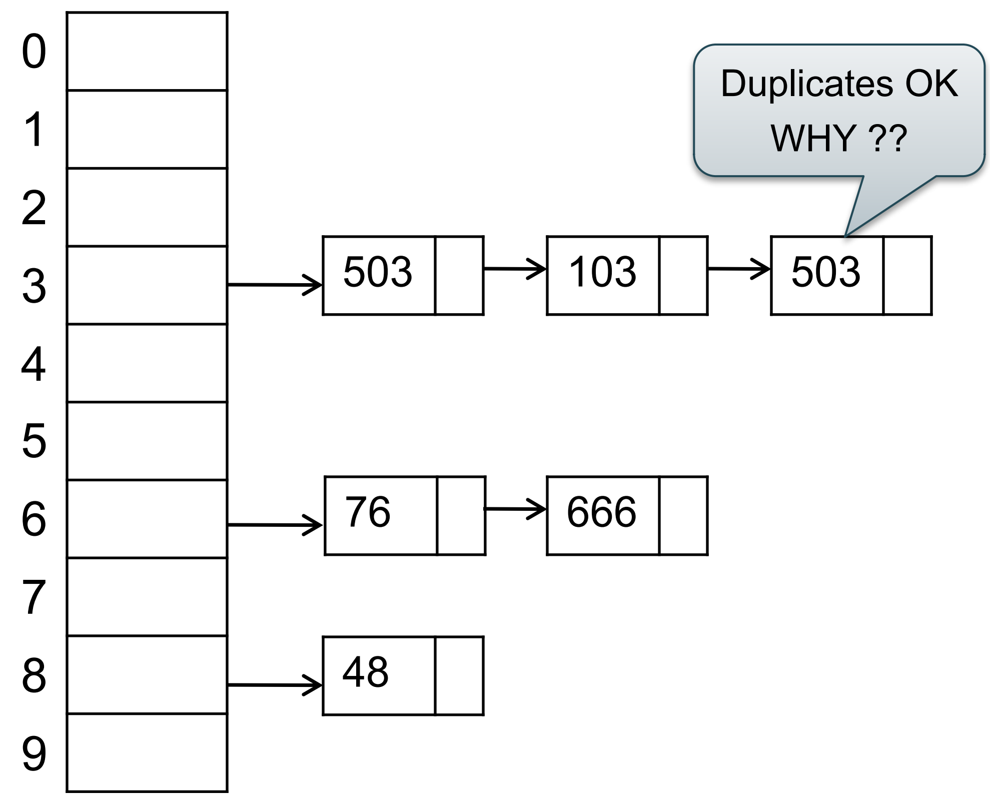
1. Nested Loop Join — $O(n^2)$
2. Merge join — $O(n \log n)$
3. Hash join — $O(n) \ldots O(n^2)$

BRIEF Review of Hash Tables

Separate chaining:

A (naïve) hash function:

$$h(x) = x \bmod 10$$



Operations:

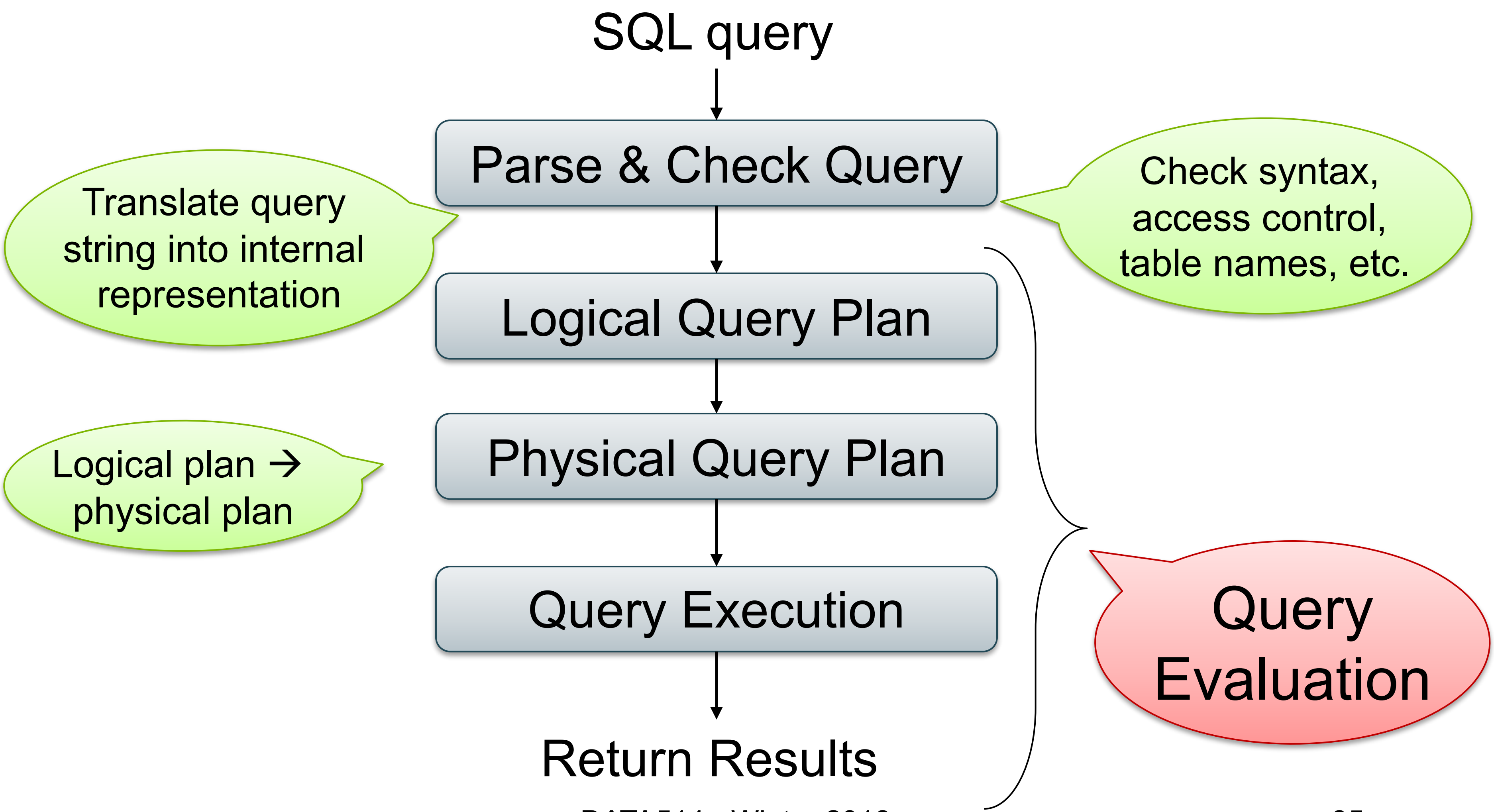
find(103) = ??

insert(488) = ??

BRIEF Review of Hash Tables

- insert(k, v) = inserts a key k with value v
- Many values for one key
  - Hence, duplicate k's are OK
- find(k) = returns the ***list*** of all values v associated to the key k

Query Evaluation Steps Review



Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Relational Algebra

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
   and  y.pno = 2
   and x.scity = 'Seattle'
   and x.sstate = 'WA'
```

Give a relational algebra expression for this query

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Relational Algebra

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
   and  y.pno = 2
   and x.scity = 'Seattle'
   and x.sstate = 'WA'
```

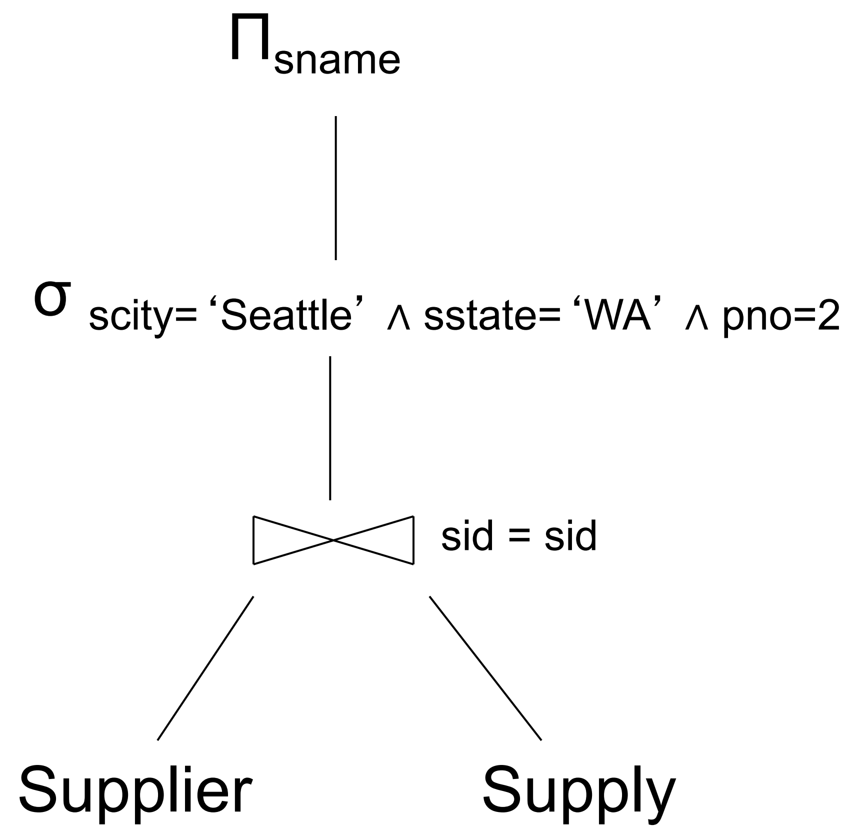
$$\Pi_{sname}(\sigma_{scity= \text{'Seattle'} \wedge sstate= \text{'WA'} \wedge pno=2}(\text{Supplier} \bowtie_{sid = sid} \text{Supply}))$$

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Relational Algebra

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
   and  y.pno = 2
   and x.scity = 'Seattle'
   and x.sstate = 'WA'
```

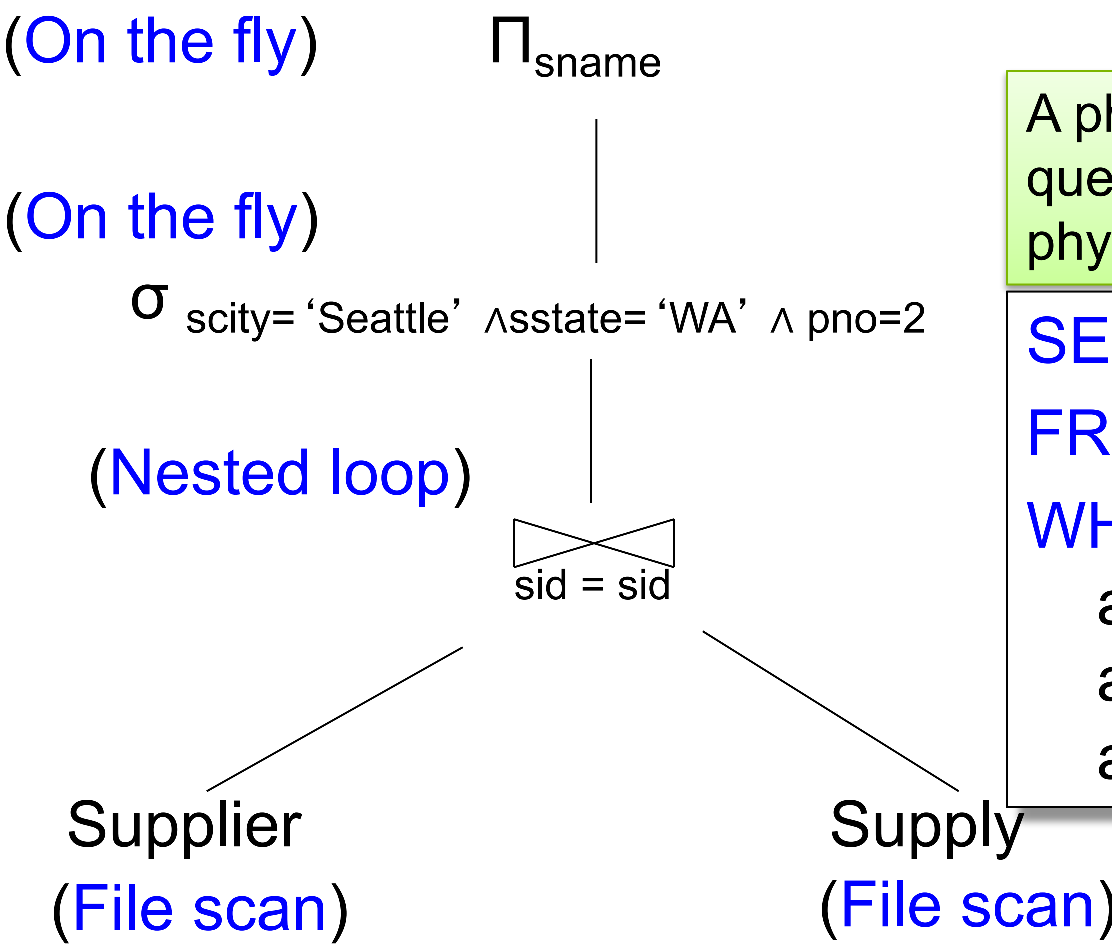


Relational algebra expression is also called the "logical query plan"

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Query Plan 1



A physical query plan is a logical query plan annotated with physical implementation details

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
   and  y.pno = 2
   and x.scity = 'Seattle'
   and x.sstate = 'WA'
```

Supplier(sid, sname, scity, sstate)

Supply(sid, pno, quantity)

Physical Query Plan 2

(On the fly) Π_{sname}

(On the fly) $\sigma_{scity='Seattle' \wedge sstate='WA' \wedge pno=2}$

(Hash join) $\bowtie_{sid = sid}$

Supplier (File scan)

Supply (File scan)

Same logical query plan
Different physical plan

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
   and  y.pno = 2
   and x.scity = 'Seattle'
   and x.sstate = 'WA'
```

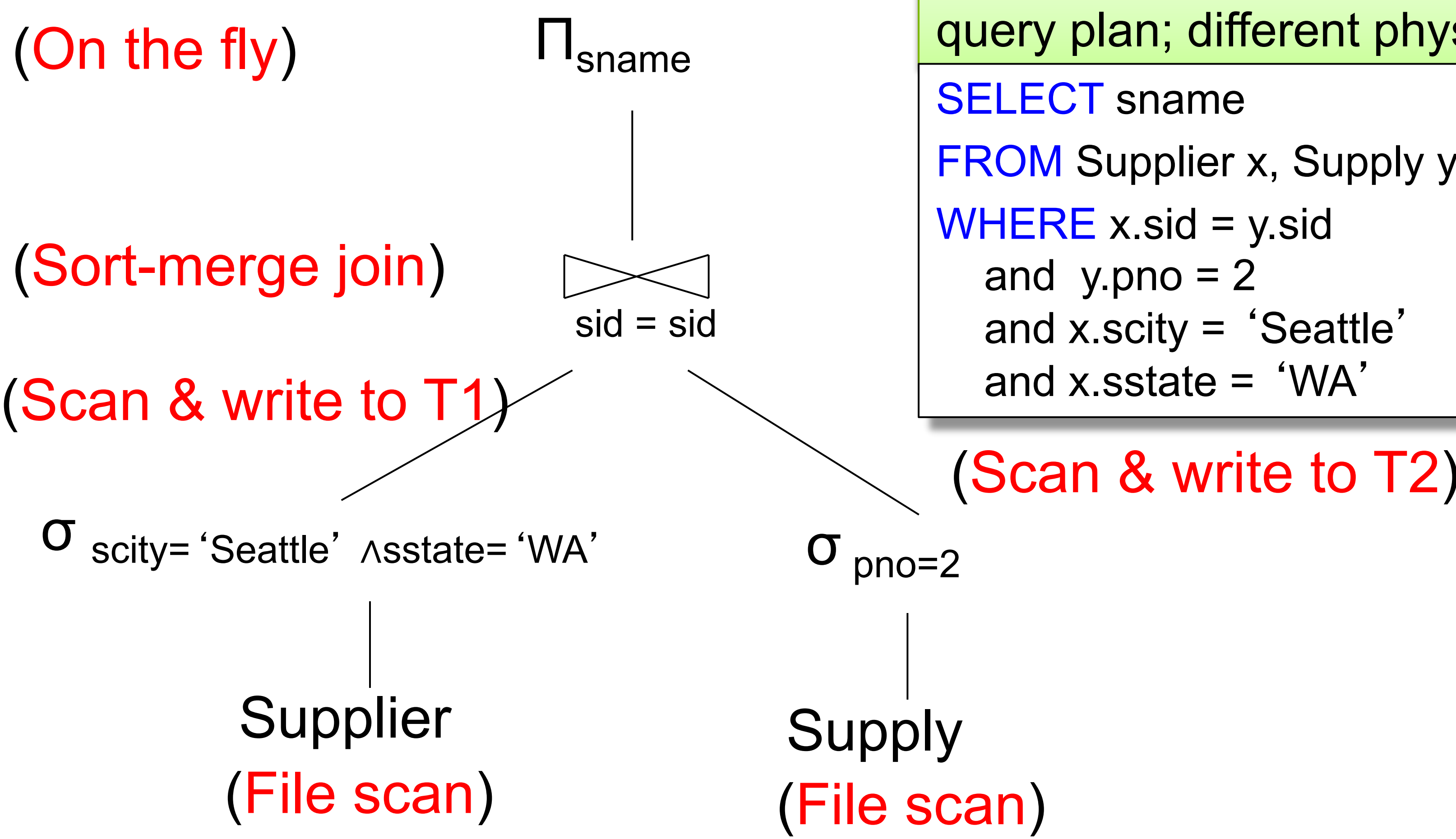
Supplier(<u>sid</u>, sname, scity, sstate)

Supply(<u>sid, pno</u>, quantity)

Physical Query Plan 3

Different but equivalent logical query plan; different physical plan

```
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
   and  y.pno = 2
   and x.scity = 'Seattle'
   and x.sstate = 'WA'
```



Query Optimization Problem

- For each SQL query… many logical plans
- For each logical plan… many physical plans
- How do find a fast physical plan?
  - Will discuss in a few lectures