

# Database System Internals

## Relational Model Review

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

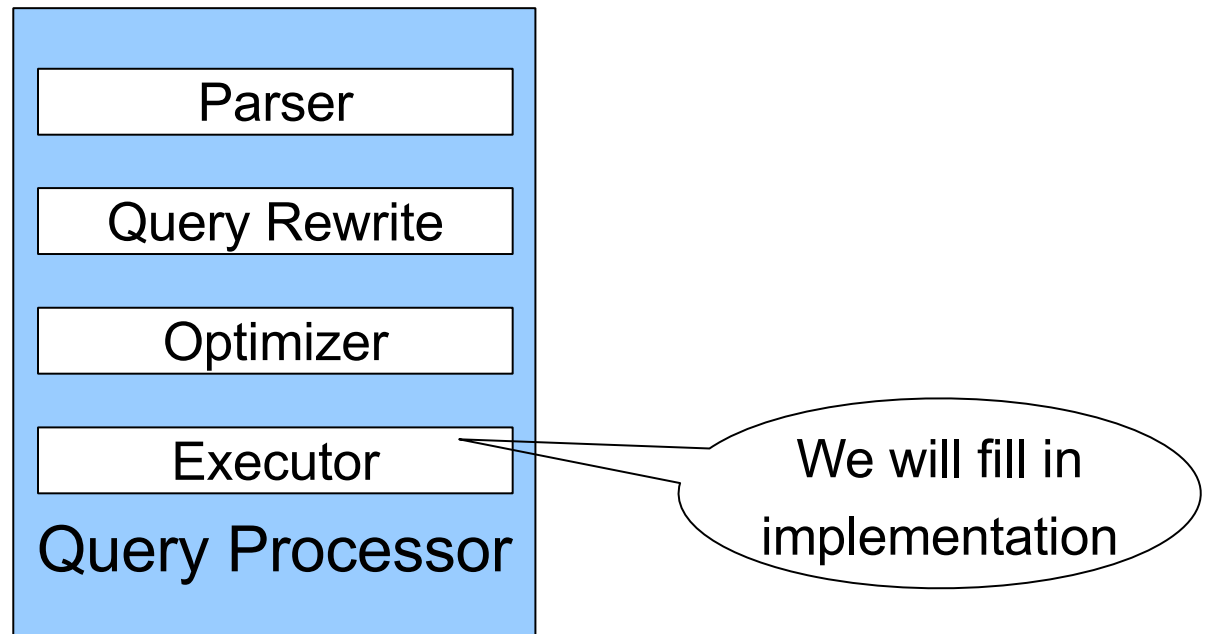
# Announcements

- Lab 1 part 1 is due on Tuesday, April 4
  - Repos not created yet due to technical issues with Gitlab, should be later today
  - Next week we will send a partner sign-up form
  - “git pull upstream master” before building
  - Thursday’s sections: SimpleDB repo and structure
- HW1 is due on April 7
  - Submit via gradescope
- 544M first paper review is due April 17
  - Submit via gradescope
  - (Not hard deadline)

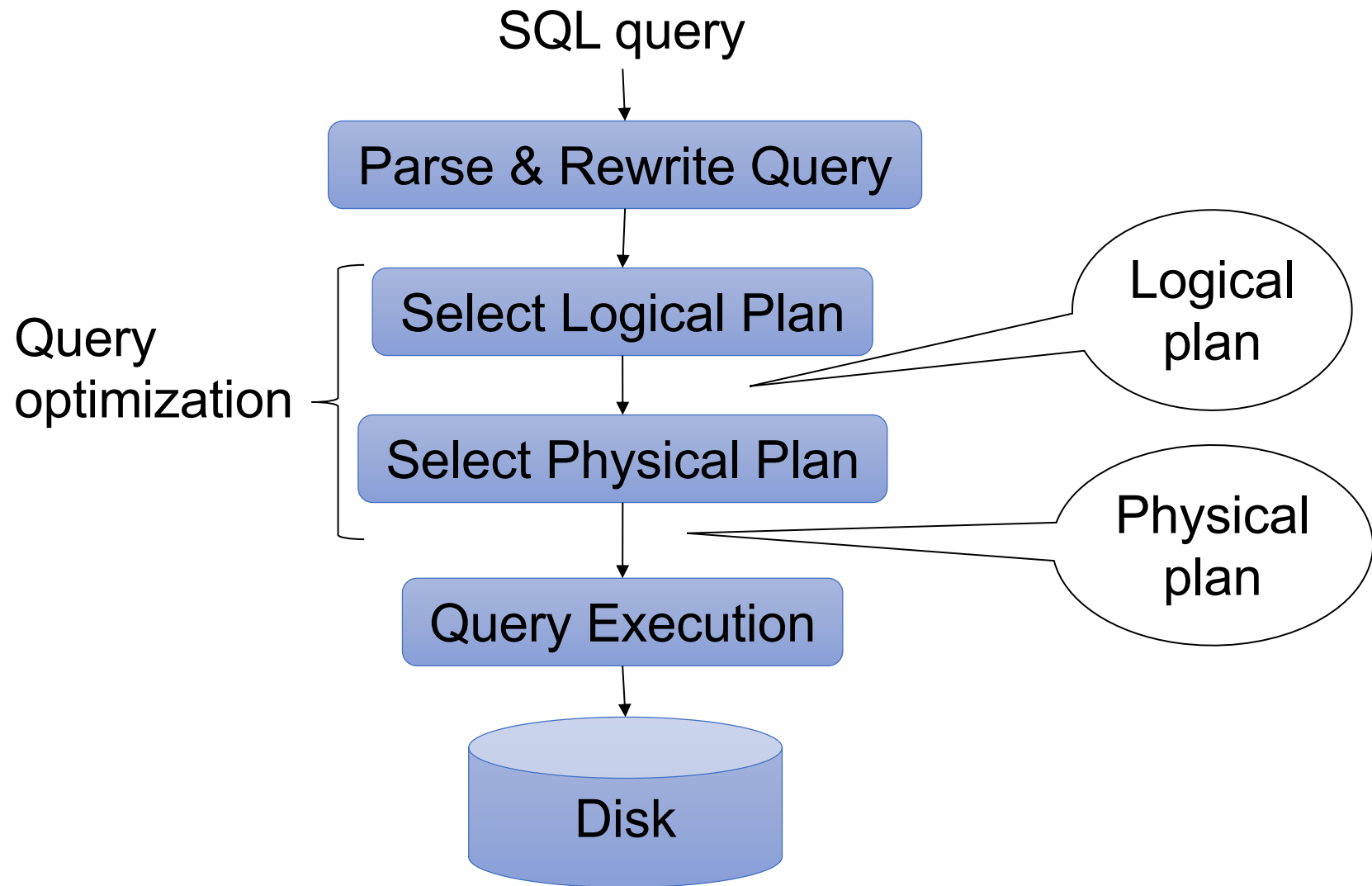
# Note

- We will go very quickly in class over the Relational Algebra and SQL
- Please review at home:
  - Read the slides that we skipped in class
  - Review material from 344 as needed

# DBMS Architecture



# Query Evaluation Steps Review



# Database/Relation/Tuple

- A **Database** is collection of relations
- A **Relation**  $R$  is subset of  $S_1 \times S_2 \times \dots \times S_n$ 
  - Where  $S_i$  is the domain of attribute  $i$
  - $n$  is number of attributes of the relation
  - A relation is a set of tuples
- A **Tuple**  $t$  is an element of  $S_1 \times S_2 \times \dots \times S_n$

Other names: relation = **table**; tuple = **row**

# Discussion

## ▪ Rows in a relation:

- Ordering immaterial (a relation is a set)
- All rows are distinct – **set semantics**
- Query answers may have duplicates – **bag semantics**

Data independence!

## ▪ Columns in a tuple:

- Ordering is significant (in theory, it shouldn't be)
- Applications refer to columns by their names

## ▪ Domain of each column is a primitive type

# Schema

- **Relation schema:** describes column heads
  - Relation name
  - Name of each field (or column, or attribute)
  - Domain of each field
- **Degree (or arity) of relation:** # attributes
- **Database schema:** set of all relation schemas

# Instance

- **Relation instance**: concrete table content
  - Set of tuples (also called records) matching the schema
- **Cardinality of relation instance**: # tuples
- **Database instance**: set of all relation instances

# What is the schema? What is the instance?

## Supplier

<b>sno</b>	<b>sname</b>	<b>scity</b>	<b>sstate</b>
1	s1	city 1	WA
2	s2	city 1	WA
3	s3	city 2	MA
4	s4	city 2	MA

# What is the schema? What is the instance?

## Relation schema

Supplier(sno: integer, sname: string, scity: string, sstate: string)

### Supplier

<b>sno</b>	<b>sname</b>	<b>scity</b>	<b>sstate</b>
1	s1	city 1	WA
2	s2	city 1	WA
3	s3	city 2	MA
4	s4	city 2	MA

} instance

# What is the schema? What is the instance?

Handled by SimpleDB  
Catalog

## Relation schema

Supplier(sno: integer, sname: string, scity: string, sstate: string)

### Supplier

sno	sname	scity	sstate
1	s1	city 1	WA
2	s2	city 1	WA
3	s3	city 2	MA
4	s4	city 2	MA

SimpleDB Storage  
Manager

instance

# Integrity Constraints

- Condition specified on a database schema
- Restricts data that can be stored in db instance
- DBMS enforces integrity constraints
  - Ensures only legal database instances exist
- Simplest form of constraint is domain constraint
  - Attribute values must come from attribute domain

# Key Constraints

- **Super Key:** “set of attributes that functionally determines all attributes”
- **Key:** Minimal super-key; a.k.a. “candidate key”
- **Primary key:** One minimal key can be selected as primary key

# Foreign Key Constraints

- A relation can refer to a tuple in another relation
- **Foreign key**
  - Field that refers to tuples in another relation
  - This field refers to the primary key of other relation

# Key Constraint SQL Examples

```
CREATE TABLE Part (  
    pno integer,  
    pname varchar(20),  
    psize integer,  
    pcolor varchar(20),  
    PRIMARY KEY (pno)  
);
```

# Key Constraint SQL Examples

```
CREATE TABLE Supply(  
    sno integer,  
    pno integer,  
    qty integer,  
    price integer  
);
```

```
CREATE TABLE Part (  
    pno integer,  
    pname varchar(20),  
    psize integer,  
    pcolor varchar(20),  
    PRIMARY KEY (pno)  
);
```

# Key Constraint SQL Examples

```
CREATE TABLE Supply(  
    sno integer,  
    pno integer,  
    qty integer,  
    price integer,  
    PRIMARY KEY (sno,pno)  
);
```

```
CREATE TABLE Part (  
    pno integer,  
    pname varchar(20),  
    psize integer,  
    pcolor varchar(20),  
    PRIMARY KEY (pno)  
);
```

# Key Constraint SQL Examples

```
CREATE TABLE Supply(  
    sno integer,  
    pno integer,  
    qty integer,  
    price integer,  
    PRIMARY KEY (sno,pno) ,  
    FOREIGN KEY (sno) REFERENCES Supplier,  
    FOREIGN KEY (pno) REFERENCES Part  
);
```

```
CREATE TABLE Part (  
    pno integer,  
    pname varchar(20),  
    psize integer,  
    pcolor varchar(20),  
    PRIMARY KEY (pno)  
);
```

# Key Constraint SQL Examples

```
CREATE TABLE Supply(  
  sno integer,  
  pno integer,  
  qty integer,  
  price integer,  
  PRIMARY KEY (sno,pno) ,  
  FOREIGN KEY (sno) REFERENCES Supplier  
      ON DELETE NO ACTION,  
  FOREIGN KEY (pno) REFERENCES Part  
      ON DELETE CASCADE  
);
```

```
CREATE TABLE Part (  
  pno integer,  
  pname varchar(20),  
  psize integer,  
  pcolor varchar(20),  
  PRIMARY KEY (pno)  
);
```

# General Constraints

- **Table constraints serve to express complex constraints over a single table**

```
CREATE TABLE Part (  
    pno integer,  
    pname varchar(20),  
    psize integer,  
    pcolor varchar(20),  
    PRIMARY KEY (pno),  
    CHECK ( psize > 0 )  
);
```

**Note: Also possible to create constraints over many tables**  
**Alternative: use database triggers for that purpose**



# Relational Query Languages

# Relational Query Language

- **Set-at-a-time:**
  - Query inputs and outputs are relations
- **Two variants of the query language:**
  - Relational algebra: specifies order of operations
  - Relational calculus / SQL: declarative

# Relational Algebra

- **Queries specified in an operational manner**
  - A query gives a step-by-step procedure
- **Relational operators**
  - Take one or two relation instances as argument
  - Return one relation instance as result
  - Easy to compose into **relational algebra expressions**

# Five Basic Relational Operators

- **Selection:**  $\sigma_{\text{condition}}(S)$ 
  - Condition is Boolean combination ( $\wedge, \vee$ ) of atomic predicates ( $<, \leq, =, \neq, \geq, >$ )
- **Projection:**  $\pi_{\text{list-of-attributes}}(S)$
- **Union** ( $\cup$ )
- **Set difference** ( $-$ ),
- **Cross-product/cartesian product** ( $\times$ ),  
**Join:**  $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$

# Selection & Projection Examples

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

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

zip
98120
98125

# Cross-Product Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \times V$

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

# Different Types of Join

- **Theta-join:**  $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$ 
  - Join of R and S with a join condition  $\theta$
  - Cross-product followed by selection  $\theta$
- **Equijoin:**  $R \bowtie_{\theta} S = \pi_A(\sigma_{\theta}(R \times S))$ 
  - Join condition  $\theta$  consists only of equalities
  - Projection  $\pi_A$  drops all redundant attributes
- **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

# Different Types of Join

Our focus in SimpleDB  
We have a class for the  
predicate  $\theta$

- **Theta-join:**  $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$ 
  - Join of R and S with a join condition  $\theta$
  - Cross-product followed by selection  $\theta$
- **Equijoin:**  $R \bowtie_{\theta} S = \pi_A(\sigma_{\theta}(R \times S))$ 
  - Join condition  $\theta$  consists only of equalities
  - Projection  $\pi_A$  drops all redundant attributes
- **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

# EqJoin Example

AnonPatient P

age	zip	disease
50	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie_{P.zip = V.zip \text{ and } P.age = V.age} V$

P.age	P.zip	P.disease	V.name	V.age	V.zip
20	98120	flu	p2	20	98120

# Theta-Join Example

AnonPatient P

age	zip	disease
50	98125	heart
19	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

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

P.age	P.zip	P.disease	V.name	V.age	V.zip
19	98120	flu	p2	20	98120

# Natural Join Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

$P \bowtie V$

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

Note:  
age, zip  
occur only  
once

# More Joins

- **Outer join**

- Include tuples with no matches in the output
- Use NULL values for missing attributes

- **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

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
33	98120	lung	null

# Logical Query Plans

Supplier(sno, sname, scity, sstate)

Supply(sno, pno, qty, price)

Part(pno, pname, psize, pcolor)

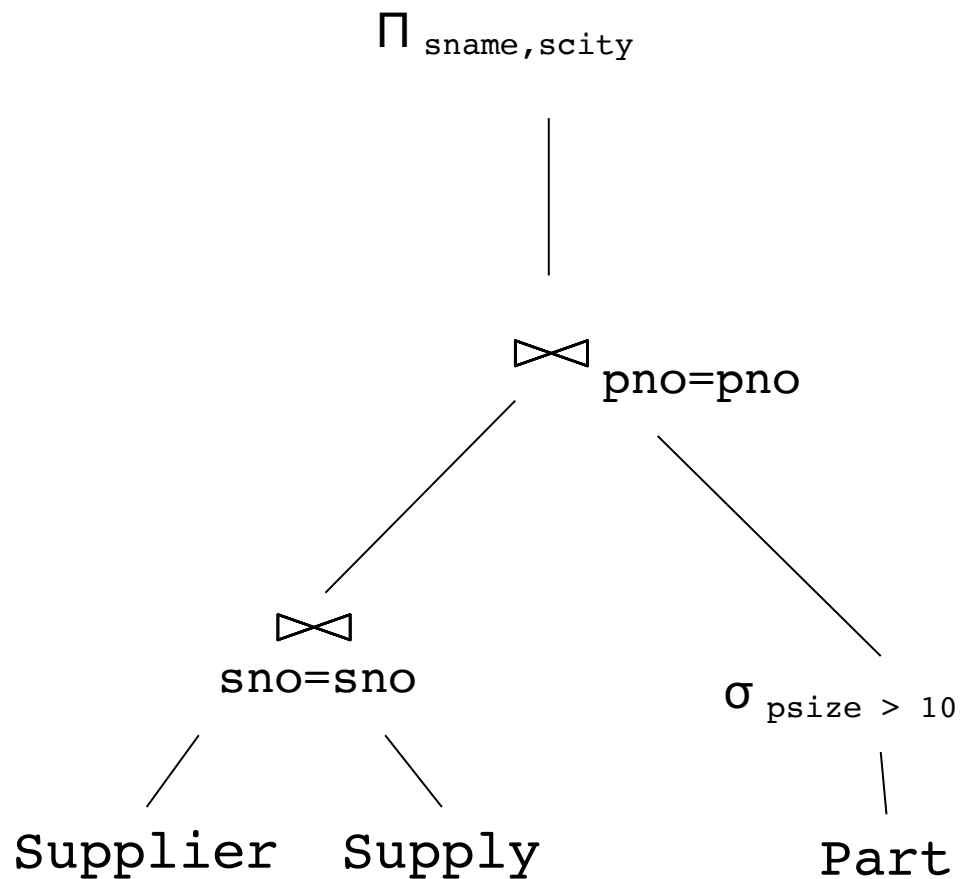
# Logical Query Plans

Supplier(sno, sname, scity, sstate)

Supply(sno, pno, qty, price)

Part(pno, pname, psize, pcolor)

Q: What does this query compute?

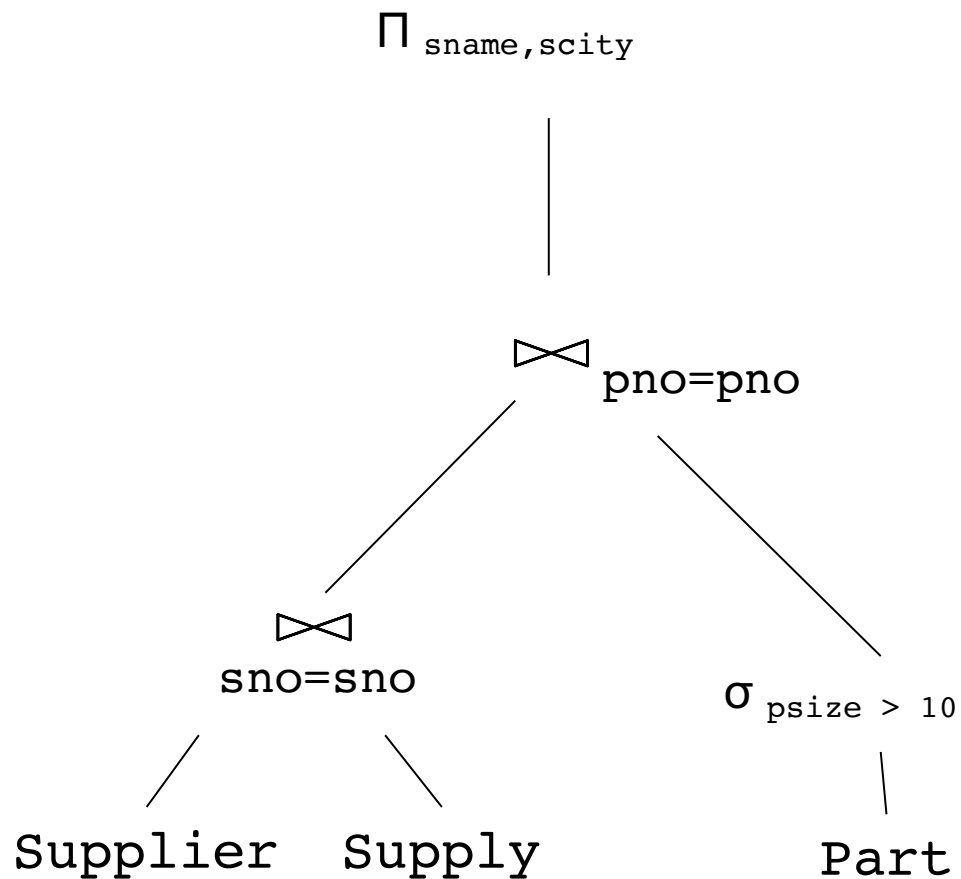


# Logical Query Plans

Supplier(sno, sname, scity, sstate)

Supply(sno, pno, qty, price)

Part(pno, pname, psize, pcolor)



**Q: What does this query compute?**

**A: The name and city of suppliers who make any parts with size > 10**

# Extended Operators of RA

- **Duplicate elimination ( $\delta$ )**
  - Since commercial DBMSs operate on multisets not sets
- **Aggregate operators ( $\gamma$ )**
  - Min, max, sum, average, count
- **Grouping operators ( $\gamma$ )**
  - Partitions tuples of a relation into “groups”
  - Aggregates can then be applied to groups
- **Sort operator ( $\tau$ )**

# Structured Query Language: SQL

- Declarative query language, based on the relational calculus (see 344)
- **Data definition language**
  - Statements to create, modify tables and views
- **Data manipulation language**
  - Statements to issue queries, insert, delete data

# SQL Query

Basic form: (plus many many more bells and whistles)

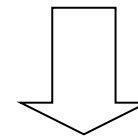
```
SELECT <attributes>  
FROM <one or more relations>  
WHERE <conditions>
```

# Simple SQL Query

Product

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

```
SELECT *  
FROM Product  
WHERE category='Gadgets'
```



PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks

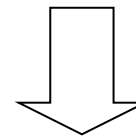
“selection”

# Simple SQL Query

Product

PName	Price	Category	Manufacturer
Gizmo	\$19.99	Gadgets	GizmoWorks
Powergizmo	\$29.99	Gadgets	GizmoWorks
SingleTouch	\$149.99	Photography	Canon
MultiTouch	\$203.99	Household	Hitachi

```
SELECT PName, Price, Manufacturer  
FROM Product  
WHERE Price > 100
```



“selection” and  
“projection”

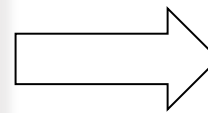
PName	Price	Manufacturer
SingleTouch	\$149.99	Canon
MultiTouch	\$203.99	Hitachi

# Details

- **Case insensitive:**
  - Same: SELECT Select select
  - Same: Product product
  - Different: 'Seattle' 'seattle'
  
- **Constants:**
  - 'abc' - yes
  - "abc" - no

# Eliminating Duplicates

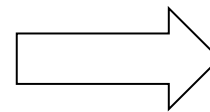
```
SELECT DISTINCT category  
FROM Product
```



Category
Gadgets
Photography
Household

Compare to:

```
SELECT category  
FROM Product
```



Category
Gadgets
Gadgets
Photography
Household

# Ordering the Results

```
SELECT pname, price, manufacturer
FROM Product
WHERE category='gizmo' AND price > 50
ORDER BY price, pname
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

# Joins

Product (pname, price, category, manufacturer)  
Company (cname, stockPrice, country)

Find all products under \$200 manufactured in Japan;  
return their names and prices.

```
SELECT PName, Price
FROM Product, Company
WHERE Manufacturer=CName AND Country='Japan'
AND Price <= 200
```

# Quick Review of SQL

Supplier(sno, sname, scity, sstate)

Supply(sno, pno, qty, price)

Part(pno, pname, psize, pcolor)

```
SELECT DISTINCT z.pno, z.pname
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno and y.pno = z.pno
      and x.scity = 'Seattle' and y.price < 100
```

What does  
this query  
compute?

# Quick Review of SQL

Supplier(sno, sname, scity, sstate)

Supply(sno, pno, qty, price)

Part(pno, pname, psize, pcolor)

What about  
this one?

```
SELECT z.pname, count(*) as cnt, min(y.price)
FROM Supplier x, Supply y, Part z
WHERE x.sno = y.sno and y.pno = z.pno
GROUP BY z.pname
```

# Aggregation

```
SELECT avg(price)
FROM Product
WHERE maker="Toyota"
```

```
SELECT count(*)
FROM Product
WHERE year > 1995
```

SQL supports several aggregation operations:  
sum, count, min, max, avg

Except count, all aggregations apply to a single attribute

# Grouping and Aggregation

```
SELECT S  
FROM R1,...,Rn  
WHERE C1  
GROUP BY a1,...,ak  
HAVING C2
```

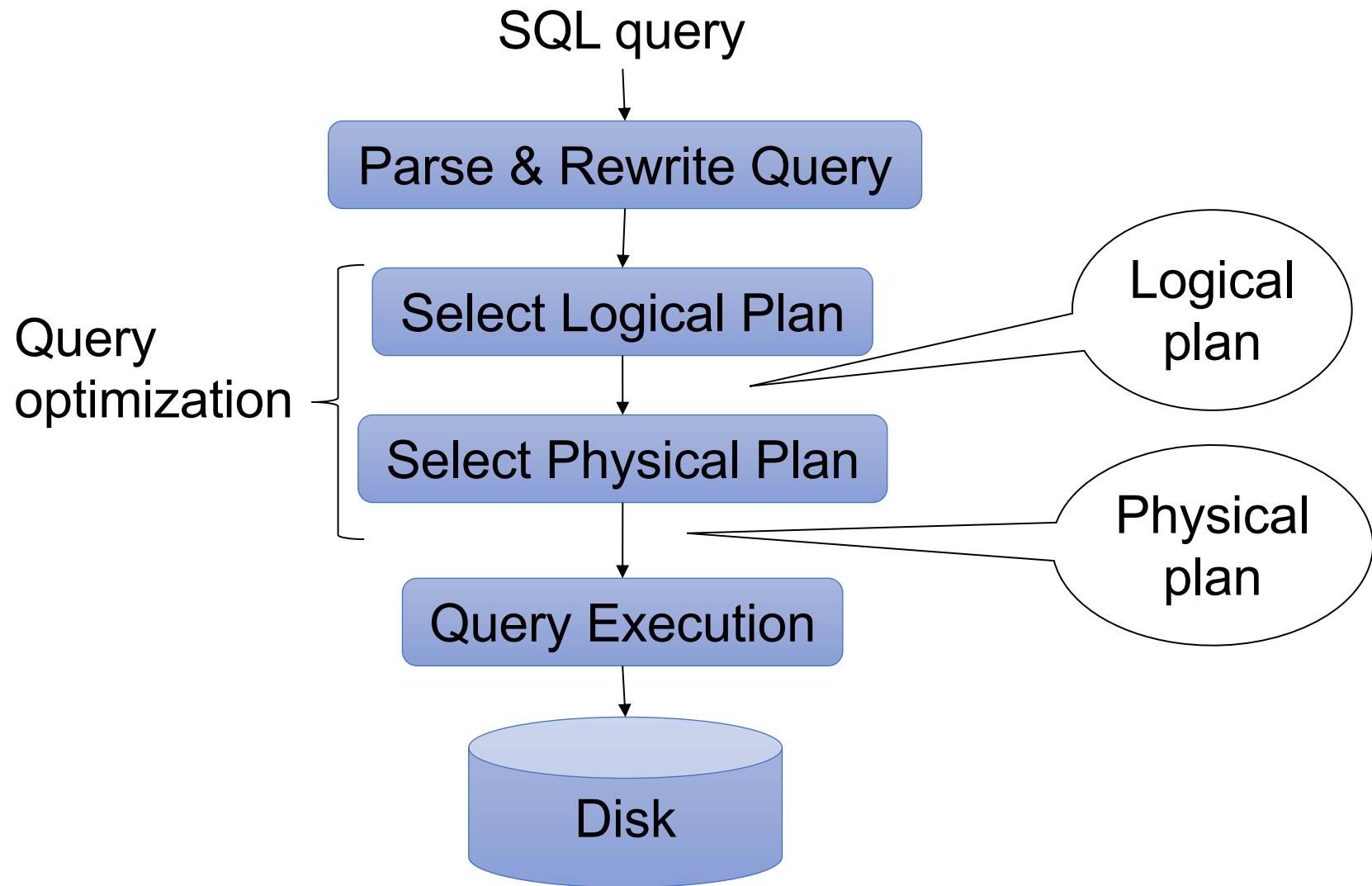
Conceptual evaluation steps:

1. Evaluate FROM-WHERE, apply condition C1
2. Group by the attributes  $a_1, \dots, a_k$
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

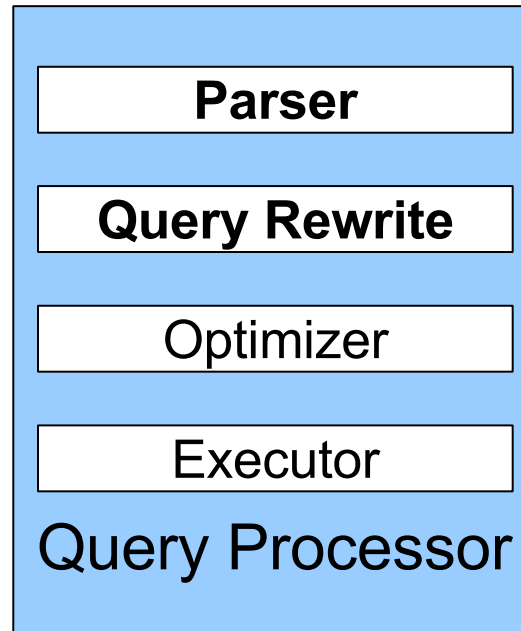
Read more about it in the book...

# From SQL to RA

# Query Evaluation Steps Review



# DBMS Architecture



# From SQL to RA

Product(pid, name, price)

Purchase(pid, cid, store)

Customer(cid, name, city)

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

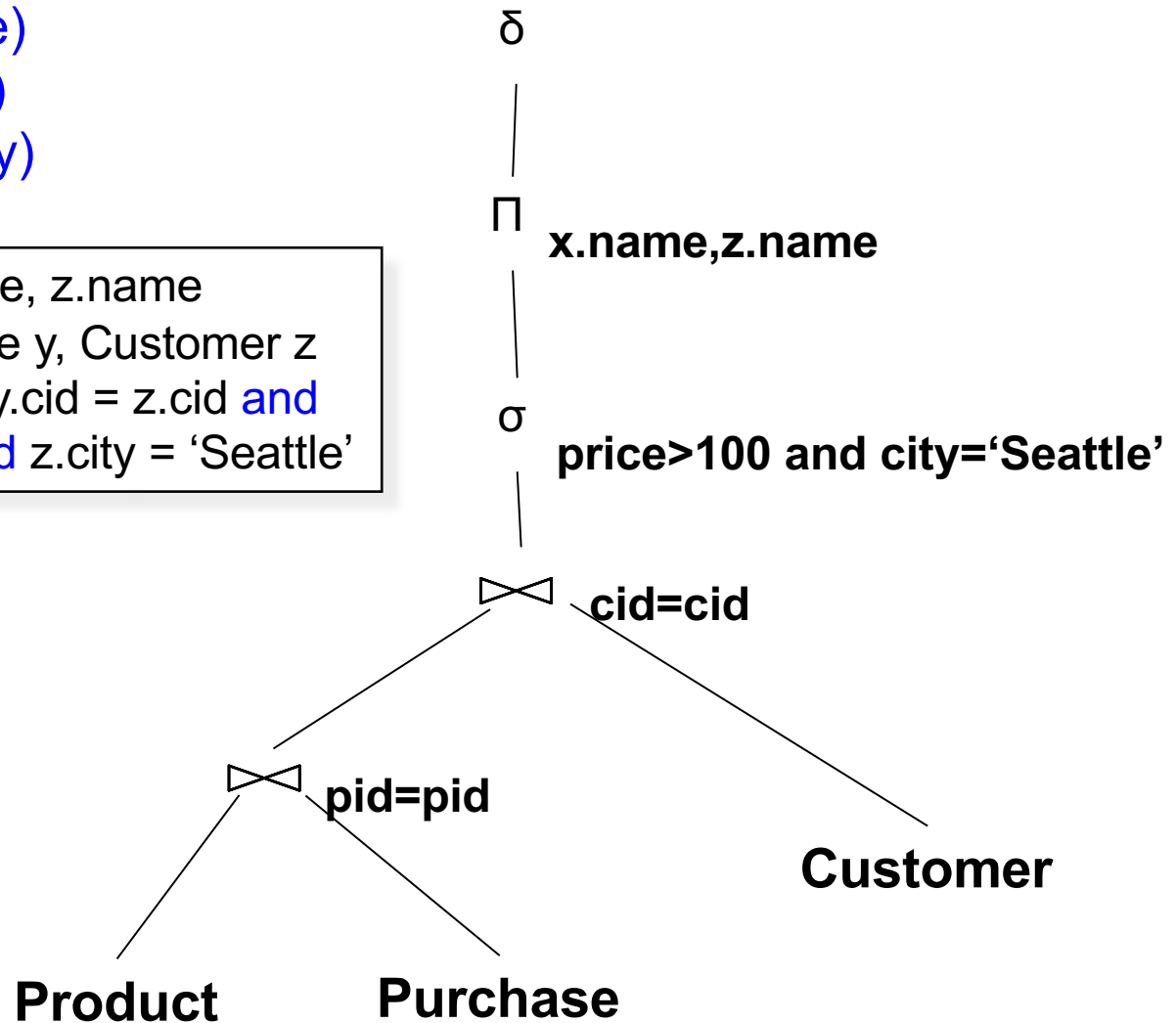
# From SQL to RA

Product(pid, name, price)

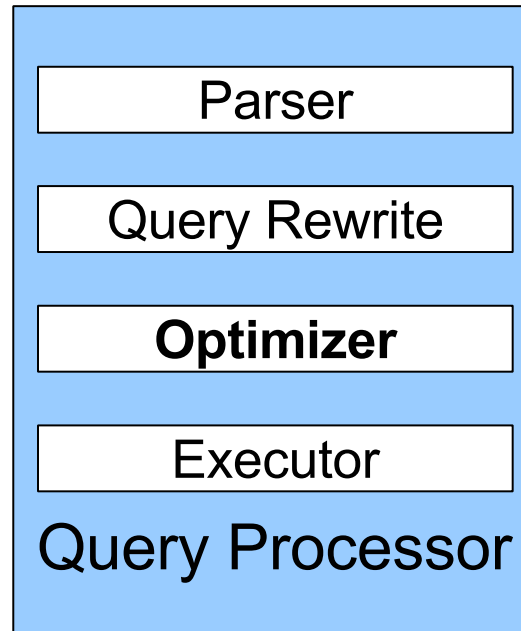
Purchase(pid, cid, store)

Customer(cid, name, city)

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



# DBMS Architecture



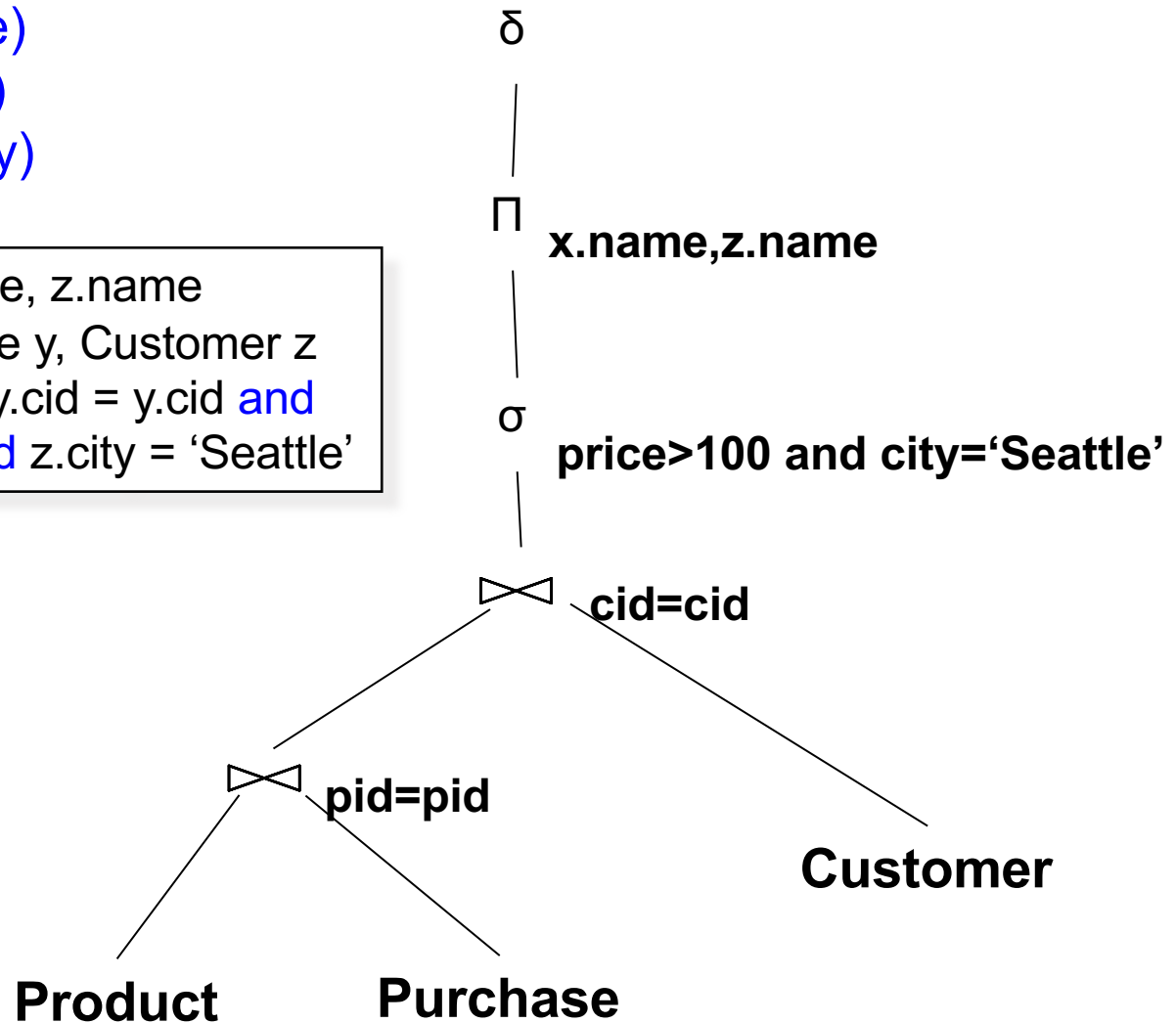
# From SQL to RA

Product(pid, name, price)

Purchase(pid, cid, store)

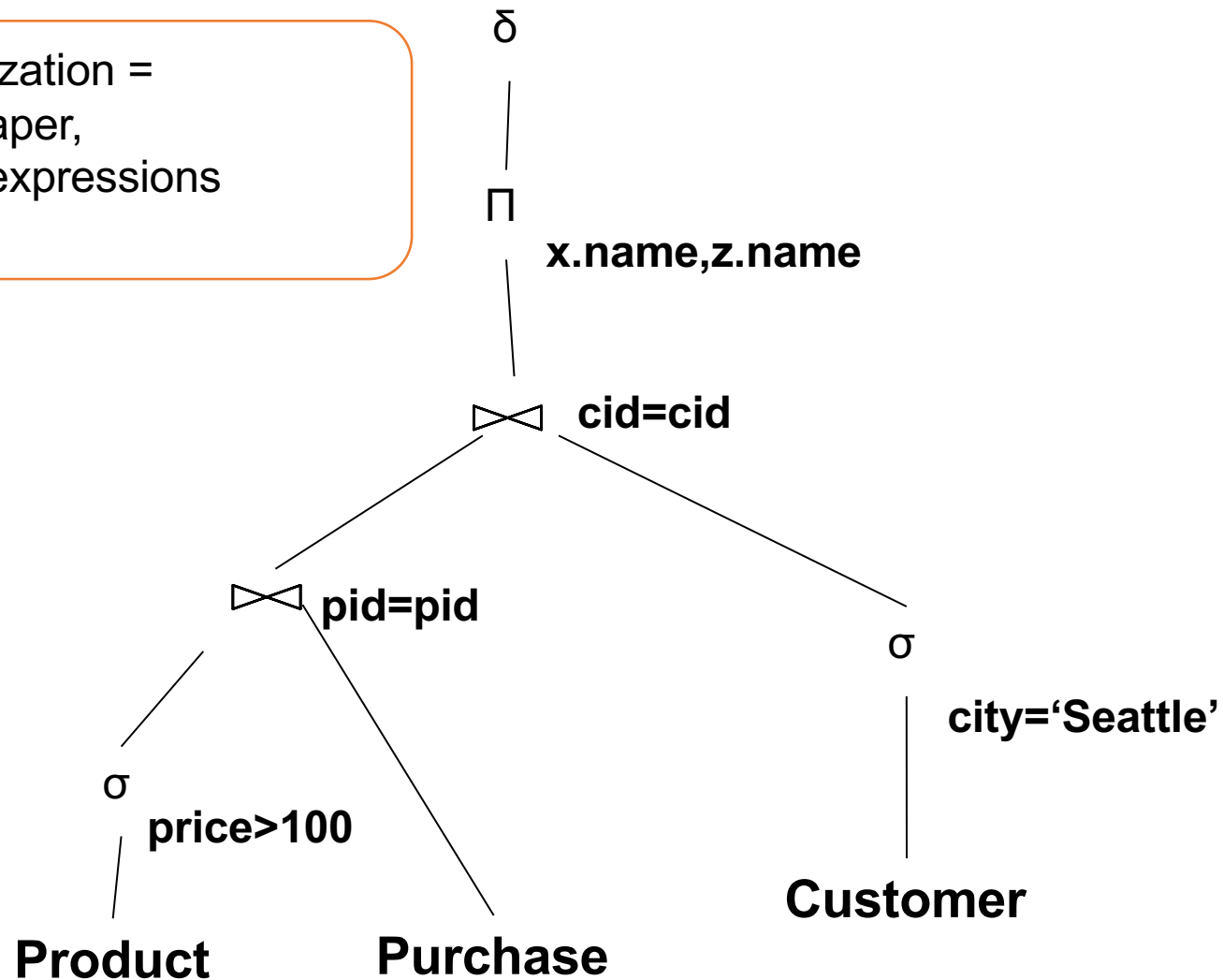
Customer(cid, name, city)

```
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'
```

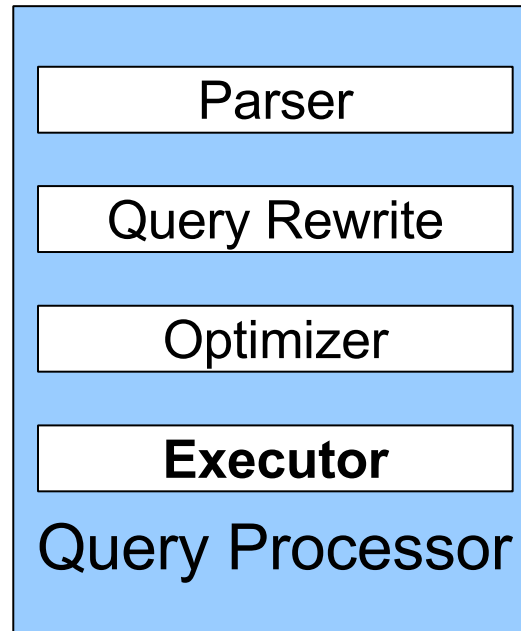


# An Equivalent Expression

Query optimization =  
finding cheaper,  
equivalent expressions

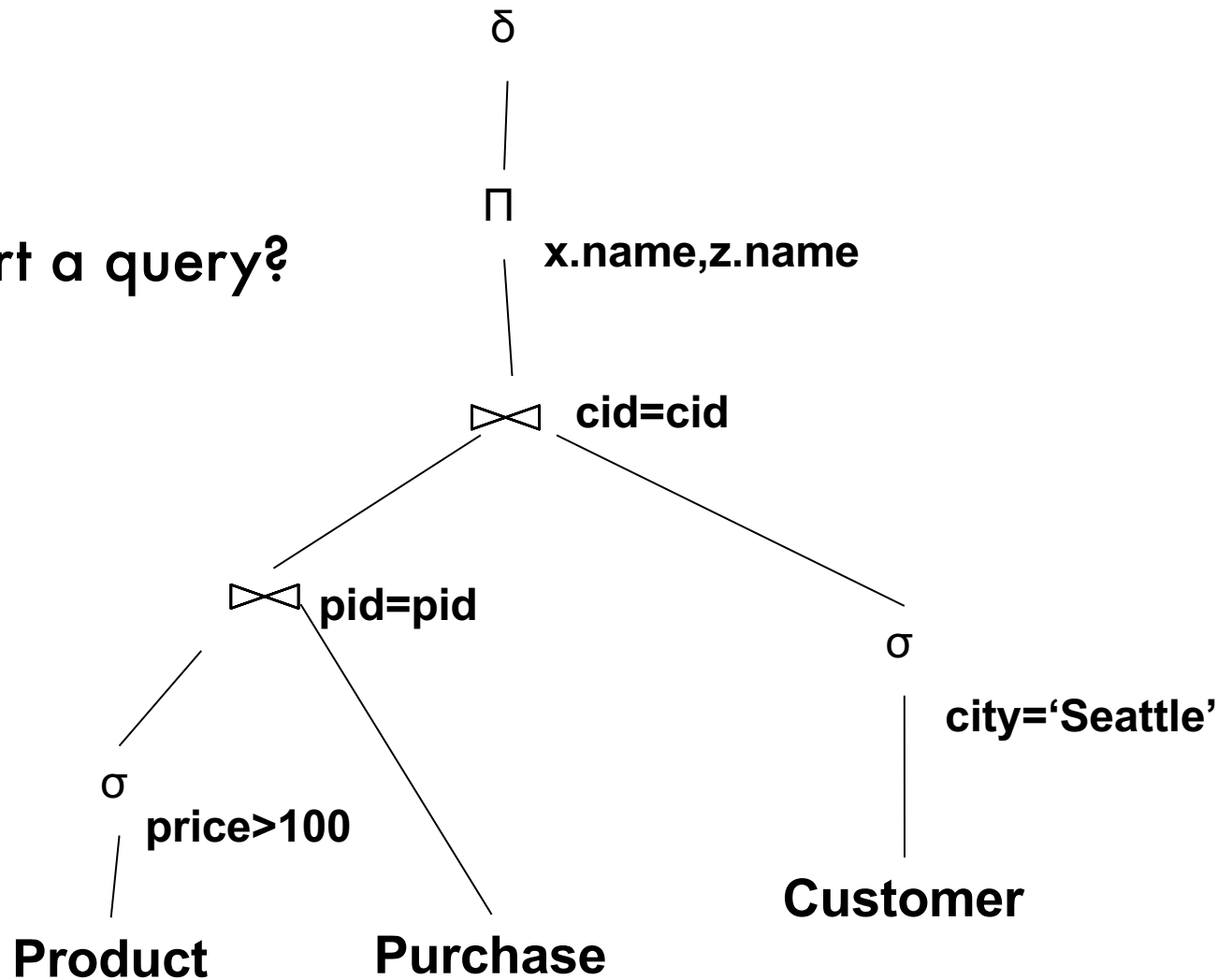


# DBMS Architecture



# Query Execution

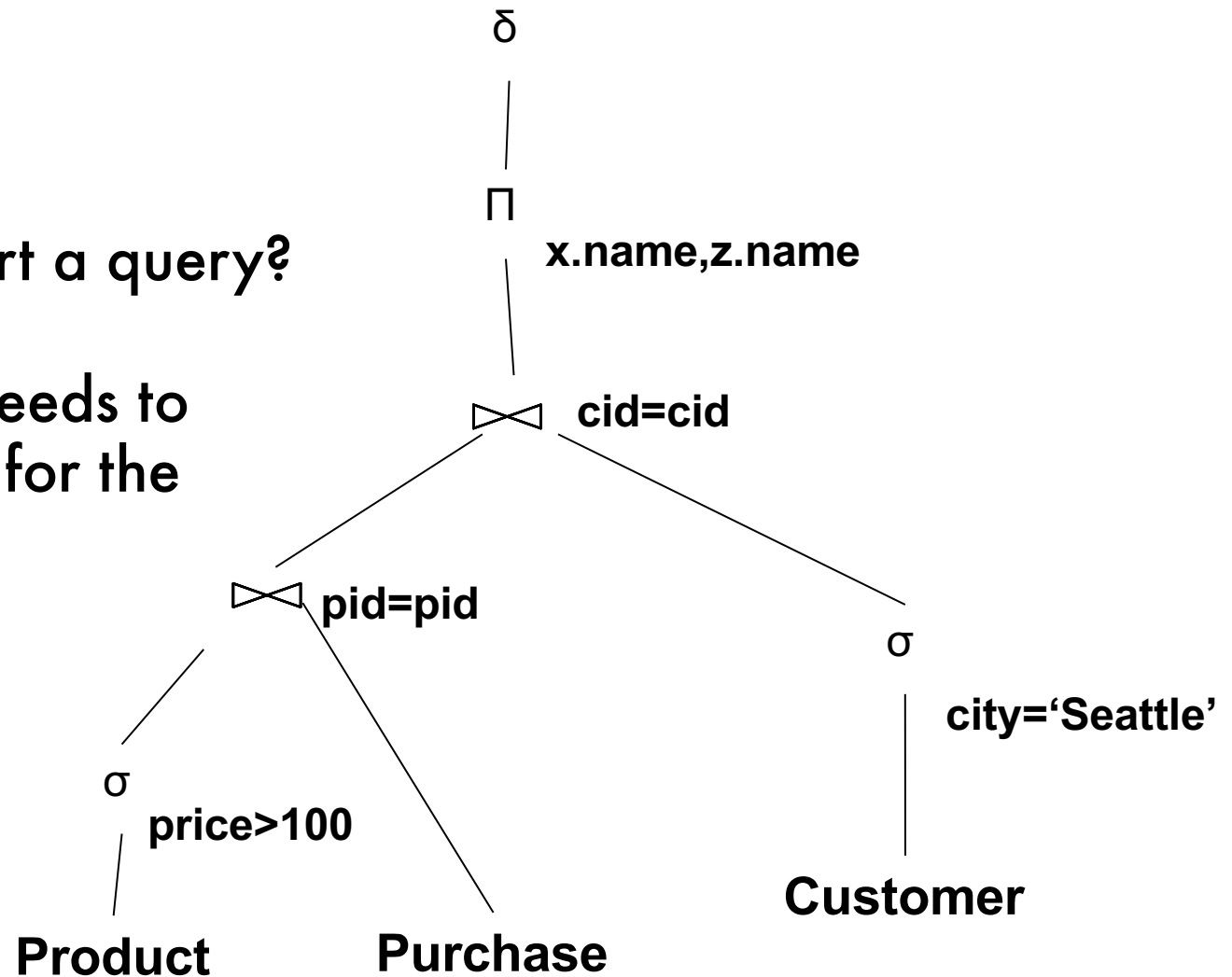
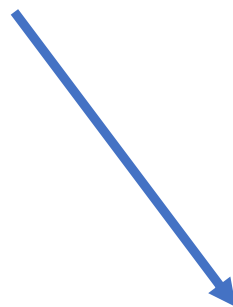
How do we start a query?



# Query Execution

How do we start a query?

Each relation needs to produce tuples for the plan to modify



# Query Evaluation Steps Review

