discussion1 SQL

丛培珊

Table Schemas

- The **schema** of a table is the table name, its attributes, and their types:

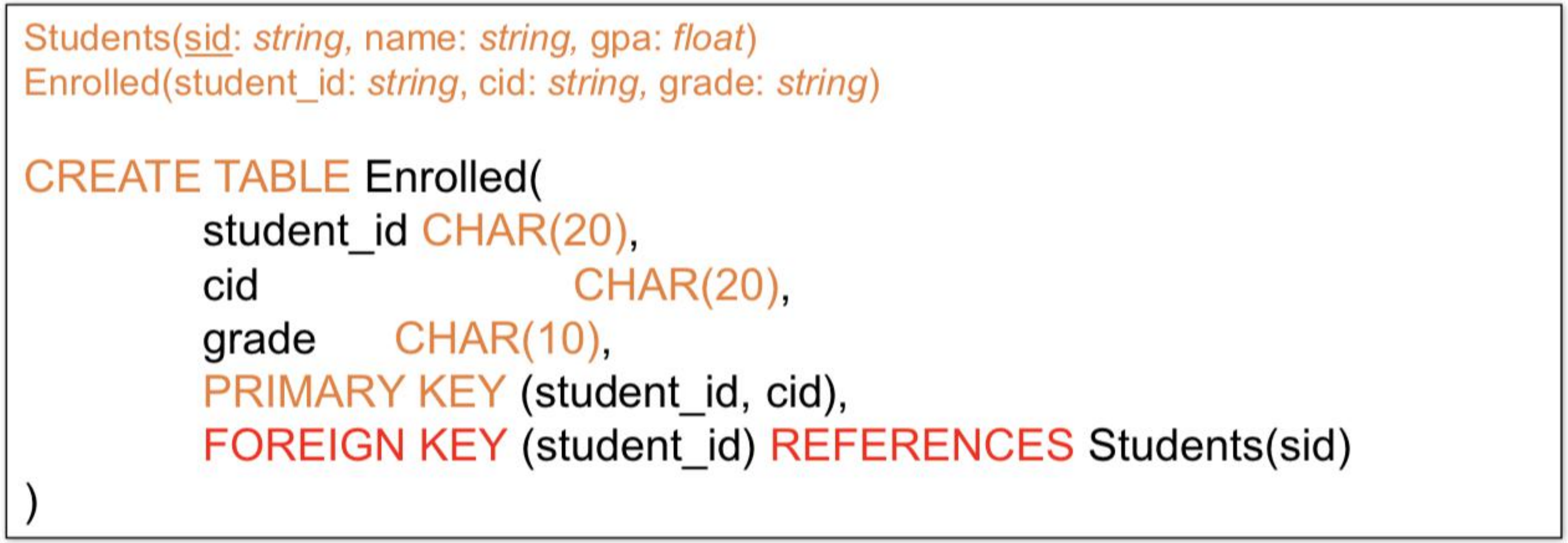
Product(Pname: *string*, Price: *float*, Category: *string*, Manufacturer: *string*)

- Attribute (Column, Field);   Tuple (Record, Row)

- A **key** is an attribute whose values are unique; we underline a key

Product(Pname: *string*, Price: *float*, Category: *string*, Manufacturer: *string*)

create a table



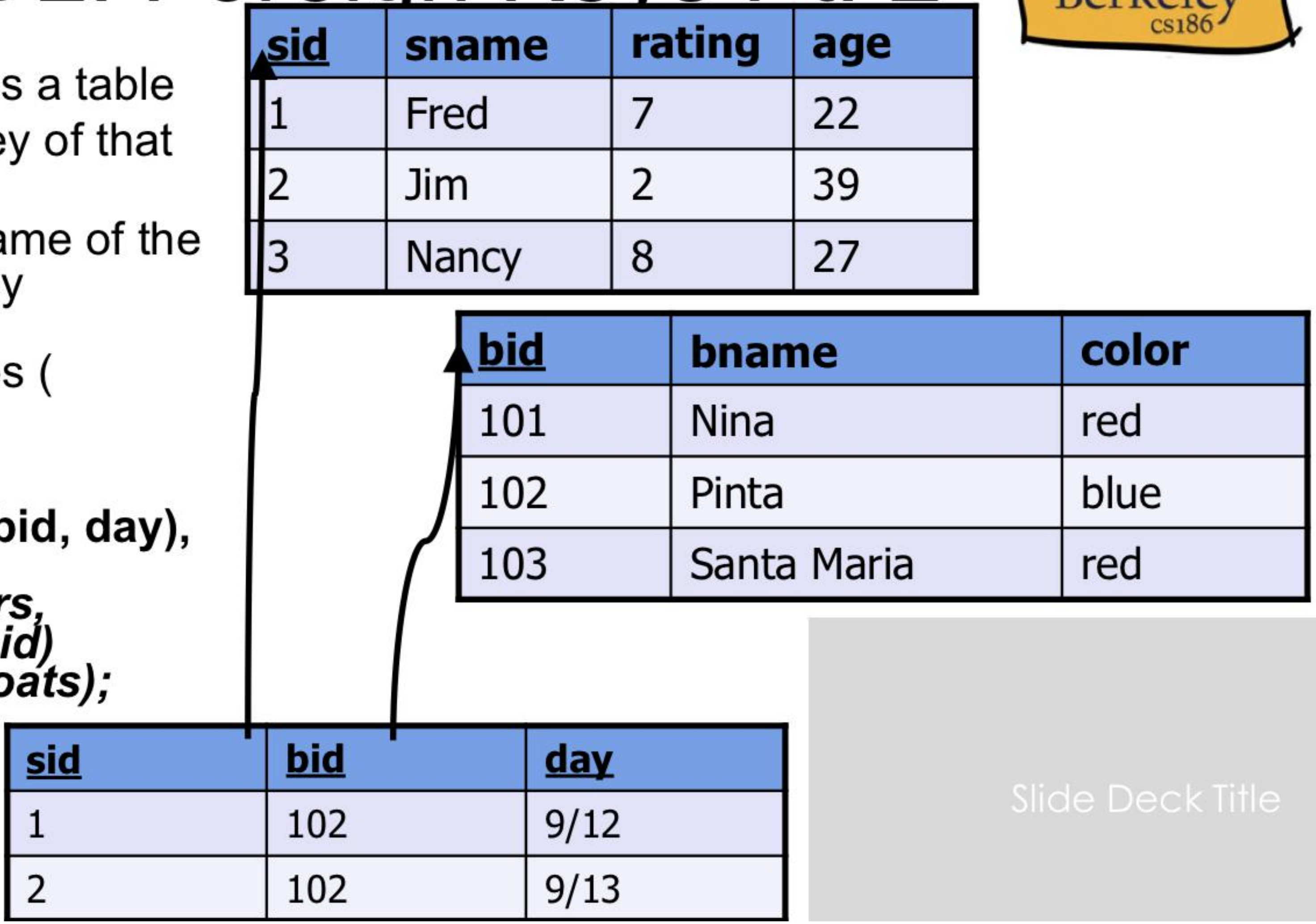
```
Students(sid: string, name: string, gpa: float)
Enrolled(student_id: string, cid: string, grade: string)

CREATE TABLE Enrolled(
        student_id CHAR(20),
        cid                  CHAR(20),
        grade     CHAR(10),
        PRIMARY KEY (student_id, cid),
        FOREIGN KEY (student_id) REFERENCES Students(sid)
)
```

The SQL DDL: Foreign Keys Pt. 2

- Foreign key references a table
  - Via the primary key of that table
- Need not share the name of the referenced primary key

```
CREATE TABLE Reserves (
    sid INTEGER,
    bid INTEGER,
    day DATE,
    PRIMARY KEY (sid, bid, day),
    FOREIGN KEY (sid)
    REFERENCES Sailors,
        FOREIGN KEY (bid)
        REFERENCES Boats);
```

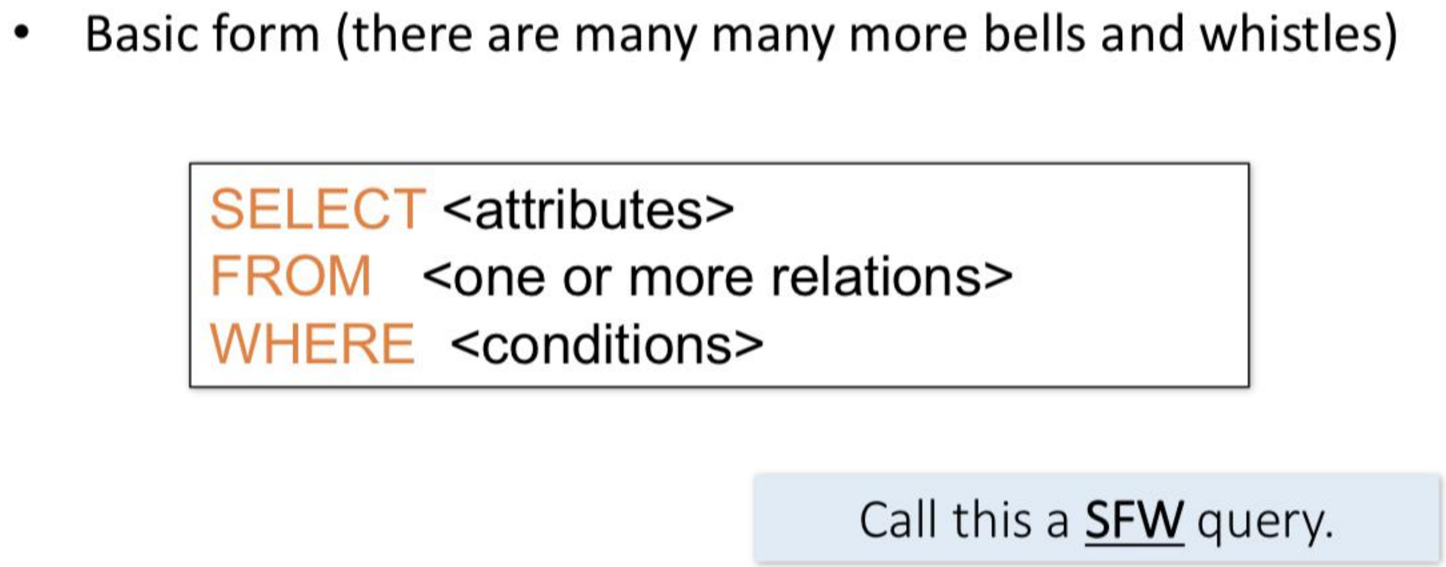


sid	sname	rating	age
1	Fred	7	22
2	Jim	2	39
3	Nancy	8	27

bid	bname	color
101	Nina	red
102	Pinta	blue
103	Santa Maria	red

sid	bid	day
1	102	9/12
2	102	9/13

basic form

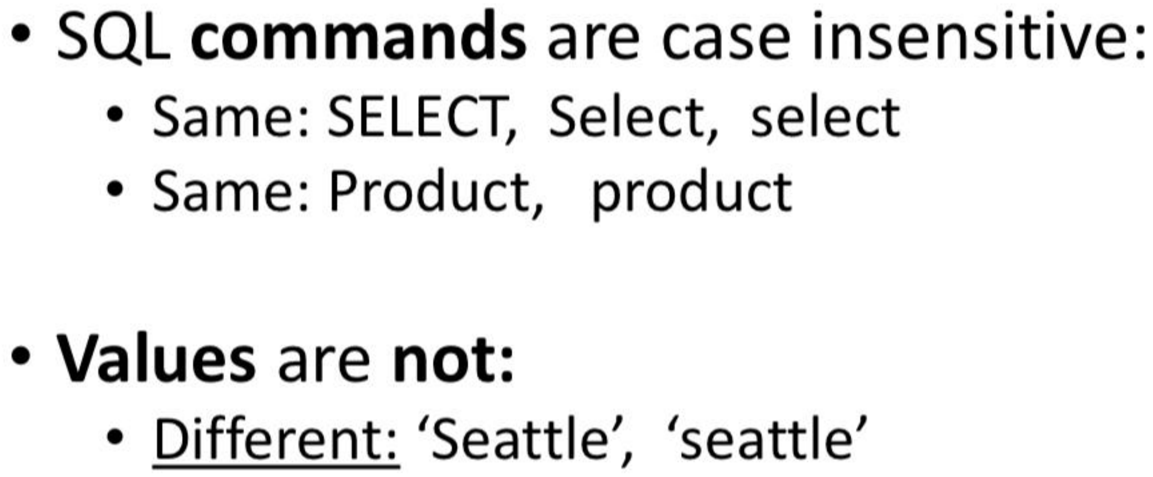


- Basic form (there are many many more bells and whistles)

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

Call this a <u>SFW</u> query.

A few detail



- SQL **commands** are case insensitive:
  - Same: SELECT, Select, select
  - Same: Product, product
- **Values** are **not:**
  - Different: 'Seattle', 'seattle'

- more than one key   ok
- NULL（not primary key） ok

- primary key column(s)

– Provides a unique “lookup key” for the relation

– Cannot have any duplicate values

– Can be made up of >1 column • E.g. (firstname, lastname)

Null Values

- *For numerical operations,* NULL -> NULL:
  - If x = NULL then 4*(3-x)/7 is still NULL

- *For boolean operations,* in SQL there are three values:

FALSE	**=**	**0**
UNKNOWN	**=**	**0.5**
TRUE	**=**	**1**

  - If x= NULL then x="Joe" is UNKNOWN

Null Values

Can test for NULL explicitly:

- x IS NULL
- x IS NOT NULL

```
SELECT *
FROM   Person
WHERE  age < 25 OR age >= 25
   OR age IS NULL
```

Now it includes all Persons!

Multi-table queries

Joins

Product(PName, Price, Category, Manufacturer)

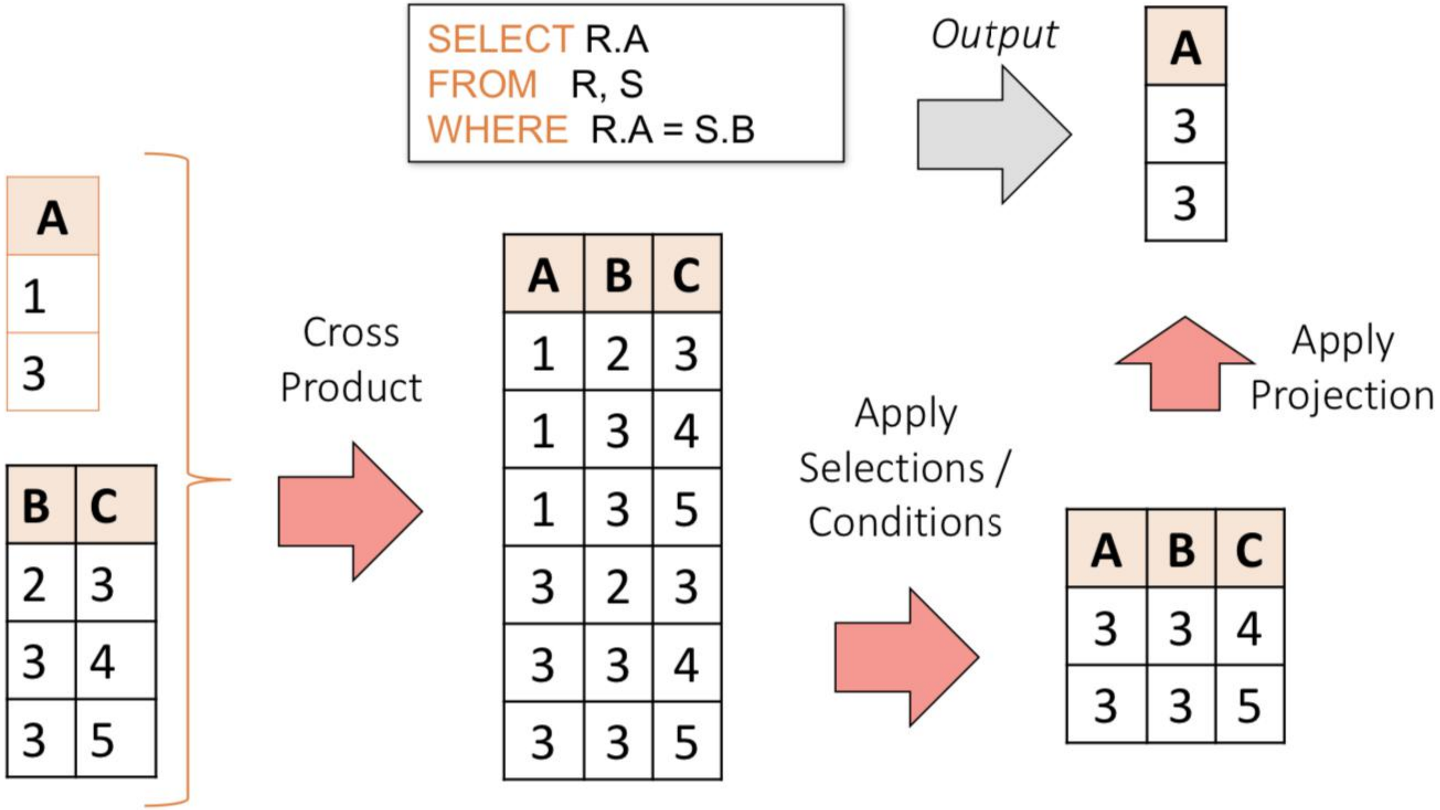
Company(CName, StockPrice, Country)

Several equivalent ways to write a basic join in SQL:

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

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

A few more later on…



```
SELECT R.A
FROM   R, S
WHERE  R.A = S.B
```

A
1
3

B	C
2	3
3	4
3	5

Cross Product

A	B	C
1	2	3
1	3	4
1	3	5
3	2	3
3	3	4
3	3	5

Apply Selections / Conditions

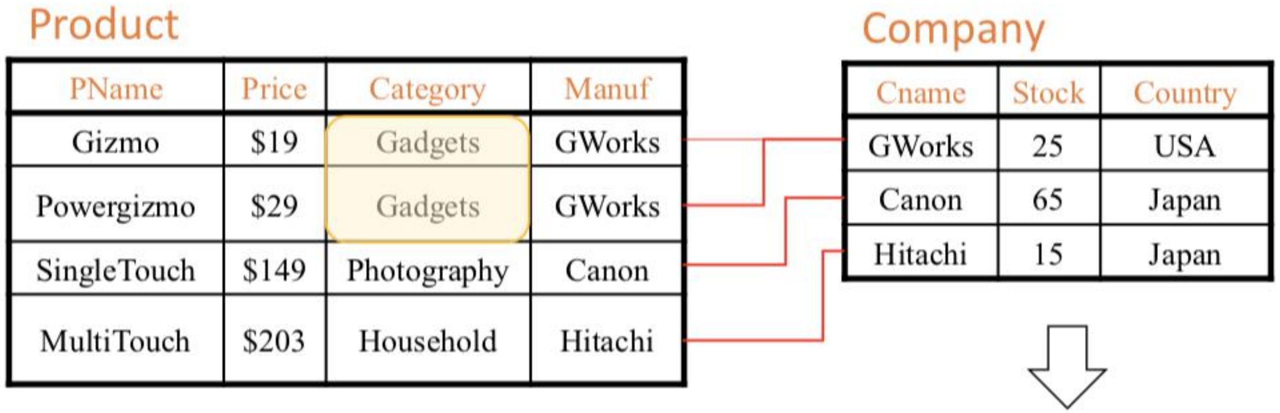
A	B	C
3	3	4
3	3	5

Apply Projection

A
3
3

Output

A subtlety about Joins



Product

PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

Company

Cname	Stock	Country
GWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

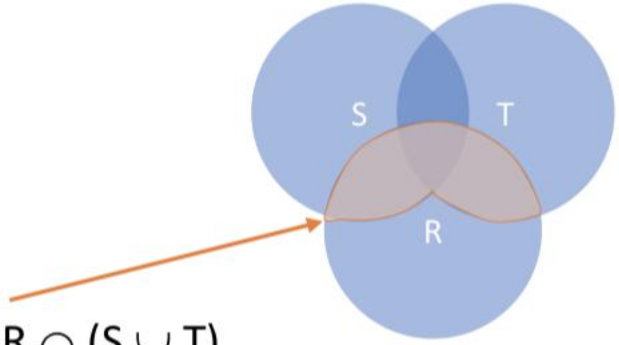
```
SELECT Country
FROM   Product, Company
WHERE  Manufacturer=Cname
  AND Category='Gadgets'
```

Country
?
?

What is the problem ?
What's the solution ?

adding "distinct"

```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```



Computes R ∩ (S ∪ T)

But what if S = ϕ?

Go back to the semantics!

- If S = {}, then the cross product of R, S, T = {}, and the query result = {}!

Multiset Operations in SQL
nested query

sub-query---in

Company(name, hq_city)
Product(pname, maker, factory_loc)

```
SELECT DISTINCT hq_city
FROM   Company, Product
WHERE  maker = name
      AND name IN (
                         SELECT maker
                         FROM   Product
                         WHERE  factory_loc = ‘US’)
            AND name IN (
                         SELECT maker
                         FROM   Product
                         WHERE  factory_loc = ‘China’)
```

*“Headquarters of companies which make gizmos in US **AND** China”*

Note: If we hadn’t used DISTINCT here, how many copies of each hq_city would have been returned?

sub-query---ALL

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

ANY and ALL not supported by SQLite.

Ex: Product(name, price, category, maker)

```
SELECT name
FROM   Product
WHERE  price > ALL(
            SELECT price
    FROM   Product
    WHERE  maker = 'Gizmo-Works')
```

Find products that are more expensive than all those produced by "Gizmo-Works"

sub-query---exists

You can also use operations of the form:

- s > ALL R
- s < ANY R
- EXISTS R

Ex:

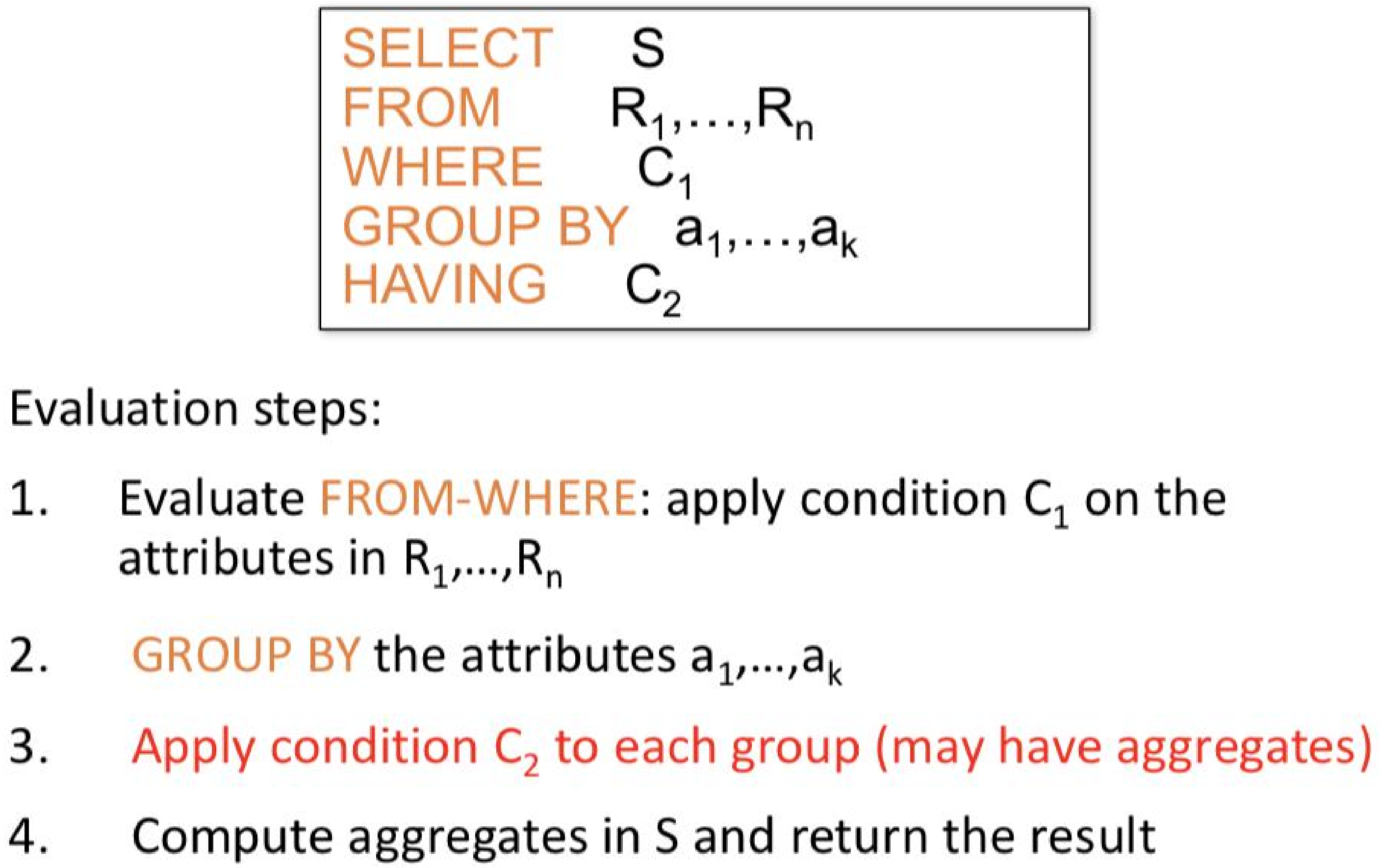
```
Product(name, price, category, maker)
```

```
SELECT p1.name
FROM   Product p1
WHERE  p1.maker = ‘Gizmo-Works’
  AND EXISTS(
            SELECT p2.name
      FROM   Product p2
      WHERE  p2.maker <> ‘Gizmo-Works’
                AND p1.name = p2.name)
```

<> means !=

Find ‘copycat’ products, i.e. products made by competitors with the same names as products made by “Gizmo-Works”

General form of Grouping and Aggregation



SELECT S
FROM $R_1,\ldots,R_n$
WHERE C_1
GROUP BY $a_1,\ldots,a_k$
HAVING C_2

Evaluation steps:

1. Evaluate FROM-WHERE: apply condition C_1 on the attributes in $R_1,\ldots,R_n$
2. GROUP BY the attributes $a_1,\ldots,a_k$
3. Apply condition C_2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

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

Aggregation: COUNT

- COUNT applies to duplicates, unless otherwise stated

```
SELECT COUNT(category)
FROM   Product
WHERE  year > 1995
```

Note: Same as COUNT(*). Why?

We probably want:

```
SELECT COUNT(DISTINCT category)
FROM   Product
WHERE  year > 1995
```

Grouping and Aggregation

Purchase(product, date, price, quantity)

```
SELECT   product,
           SUM(price * quantity) AS TotalSales
FROM     Purchase
WHERE    date > ‘10/1/2005’
GROUP BY product
```

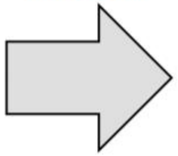
Find total sales after 10/1/2005 per product.

Let’s see what this means…

1. Compute the FROM and WHERE clauses

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

FROM

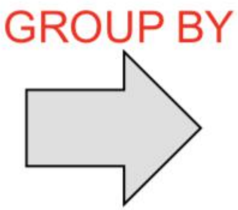


Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

2. Group by the attributes in the GROUP BY

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10



Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

3. Compute the SELECT clause: grouped attributes and aggregates

```
SELECT   product, SUM(price*quantity) AS TotalSales
FROM     Purchase
WHERE    date > '10/1/2005'
GROUP BY product
```

Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

SELECT →

Product	TotalSales
Bagel	50
Banana	15

HAVING Clause

```
SELECT   product, SUM(price*quantity)
FROM     Purchase
WHERE    date > ‘10/1/2005’
GROUP BY product
HAVING   SUM(quantity) > 100
```

Same query as before, except that we consider only products that have more than 100 buyers

HAVING clauses contains conditions on **aggregates**

*Whereas WHERE clauses condition on **individual tuples…***

ARGMAX?

- The sailor with the highest rating
  - what about ties for highest?!

```
SELECT *
FROM   Sailors S
WHERE  S.rating >= ALL
   (SELECT  S2.rating
    FROM  Sailors S2)
```

```
SELECT *
FROM   Sailors S
WHERE  S.rating =
(SELECT  MAX(S2.rating)
    FROM  Sailors S2)
```

```
SELECT *
FROM   Sailors S
ORDER BY rating DESC
LIMIT 1;
```

Median in SQL (odd cardinality)

```
SELECT c AS median FROM T
 WHERE
 (SELECT COUNT(*) from T AS T1
   WHERE T1.c <= T.c)
 =
 (SELECT COUNT(*) from T AS T2
   WHERE T2.c >= T.c);
```

Faster Median in SQL (odd cardinality)

```
SELECT x.c as median
  FROM T x, T y
 GROUP BY x.c
HAVING
 SUM(CASE WHEN y.c <= x.c THEN 1 ELSE 0 END)
  >= (COUNT(*)+1)/2 -- ceiling(N/2)
AND
 SUM(CASE WHEN y.c >= x.c THEN 1 ELSE 0 END)
  >= (COUNT(*)/2)+1 -- floor(N/2) +1
```