

Lecture 4: SQL Part III

Copyright: These slides are the modified version of the slides used in CS145 Introduction to Databases course at Stanford by Dr. Peter Bailis

Today's Lecture

1. Set operators & nested queries
 - ACTIVITY: Set operator subtleties
2. Aggregation & GROUP BY
 - ACTIVITY: Fancy SQL Part I
3. Advanced SQL-izing
 - ACTIVITY: Fancy SQL Part II

1. Set Operators & Nested Queries

What you will learn about in this section

1. Multiset operators in SQL
2. Nested queries
3. ACTIVITY: Set operator subtleties

What does this look like in Python?

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

- Semantics:

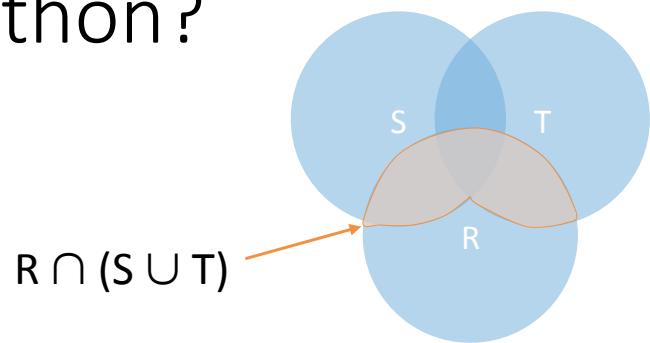
1. Take cross-product

Joins / cross-products are just **nested for loops** (in simplest implementation)!

2. Apply selections / conditions

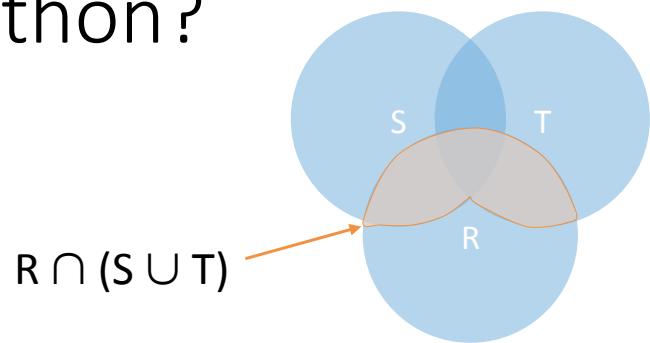
If-then statements!

3. Apply projection



What does this look like in Python?

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



```
output = []  
  
for r in R:  
    for s in S:  
        for t in T:  
            if r['A'] == s['A'] or r['A'] == t['A']:  
                output.add(r['A'])  
return list(output)
```

Can you see now what happens if $S = []$?

Multiset Operations

Recall Multisets

Multiset X

Tuple
(1, a)
(1, a)
(1, b)
(2, c)
(2, c)
(2, c)
(1, d)
(1, d)



Equivalent
Representations
of a Multiset

$\lambda(X)$ = “Count of tuple in X”
(Items not listed have
implicit count 0)

Multiset X

Tuple	
(1, a)	2
(1, b)	1
(2, c)	3
(1, d)	2

Note: In a set all
counts are {0,1}.

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

∩

Multiset Y

Tuple	
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

=

Multiset Z

Tuple	
(1, a)	2
(1, b)	0
(2, c)	2
(1, d)	0

$$\lambda(Z) = \min(\lambda(X), \lambda(Y))$$

For sets, this is intersection

Generalizing Set Operations to Multiset Operations

Multiset X

Tuple	
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

U

Multiset Y

Tuple	
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

=

Multiset Z

Tuple	
(1, a)	7
(1, b)	1
(2, c)	5
(1, d)	2

$$\lambda(Z) = \lambda(X) + \lambda(Y)$$

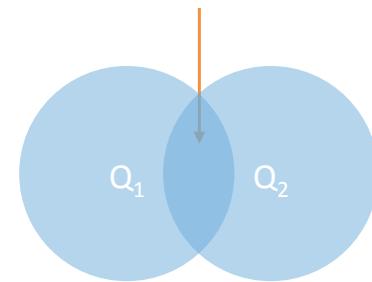
For sets,
this is union

Multiset Operations in SQL

Explicit Set Operators: INTERSECT

```
SELECT R.A  
FROM   R, S  
WHERE  R.A=S.A  
INTERSECT  
SELECT R.A  
FROM   R, T  
WHERE  R.A=T.A
```

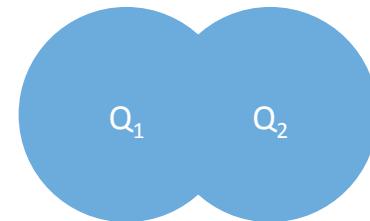
$$\{r.A \mid r.A = s.A\} \cap \{r.A \mid r.A = t.A\}$$



UNION

$$\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}$$

```
SELECT R.A  
FROM R, S  
WHERE R.A=S.A  
UNION  
SELECT R.A  
FROM R, T  
WHERE R.A=T.A
```



Why aren't there duplicates?

By default:
SQL uses set semantics!

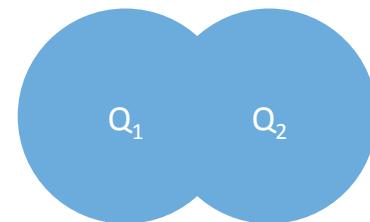
13

What if we want duplicates?

UNION ALL

```
SELECT R.A  
FROM R, S  
WHERE R.A=S.A  
UNION ALL  
SELECT R.A  
FROM R, T  
WHERE R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \cup \{r.A \mid r.A = t.A\}$$



*ALL indicates
Multiset
operations*

EXCEPT

```
SELECT R.A  
FROM   R, S  
WHERE  R.A=S.A  
EXCEPT  
SELECT R.A  
FROM   R, T  
WHERE  R.A=T.A
```

$$\{r.A \mid r.A = s.A\} \setminus \{r.A \mid r.A = t.A\}$$


What is the
multiset version?

INTERSECT: Still some subtle problems...

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

```
SELECT hq_city
FROM Company, Product
WHERE maker = name
    AND factory_loc = 'US'
INTERSECT
SELECT hq_city
FROM Company, Product
WHERE maker = name
    AND factory_loc = 'China'
```

"Headquarters of companies which make gizmos in US AND China"

What if two companies have HQ in US: BUT one has factory in China (but not US) and vice versa? What goes wrong?

INTERSECT: Remember the semantics!

```
Company(name, hq_city) AS C  
Product(pname, maker,  
factory_loc) AS P
```

```
SELECT hq_city  
FROM Company, Product  
WHERE maker = name  
AND factory_loc='US'
```

```
INTERSECT  
SELECT hq_city
```

```
FROM Company, Product  
WHERE maker = name  
AND factory_loc='China'
```

Example: C JOIN P on maker = name

C.name	C.hq_city	P.pname	P.maker	P.factory_loc
X Co.	Seattle	X	X Co.	U.S.
Y Inc.	Seattle	X	Y Inc.	China

INTERSECT: Remember the semantics!

```
Company(name, hq_city) AS C  
Product(pname, maker,  
factory_loc) AS P
```

```
SELECT hq_city  
FROM Company, Product  
WHERE maker = name  
AND factory_loc='US'
```

```
INTERSECT  
SELECT hq_city
```

```
FROM Company, Product  
WHERE maker = name  
AND factory_loc='China'
```

Example: C JOIN P on maker = name

C.name	C.hq_city	P.pname	P.maker	P.factory_loc
X Co.	Seattle	X	X Co.	U.S.
Y Inc.	Seattle	X	Y Inc.	China

X Co has a factory in the US (but not China)
Y Inc. has a factory in China (but not US)

But Seattle is returned by the query!

We did the INTERSECT
on the wrong attributes!

One Solution: Nested Queries

```
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?

High-level note on nested queries

- We can do nested queries because SQL is ***compositional***:
 - Everything (inputs / outputs) is represented as multisets- the output of one query can thus be used as the input to another (nesting)!
- This is extremely powerful!

Nested queries: Sub-queries Return Relations

Another example:

```
Company(name, city)
Product(name, maker)
Purchase(id, product, buyer)
```

```
SELECT c.city
FROM Company c
WHERE c.name IN (
    SELECT pr.maker
    FROM Purchase p, Product pr
    WHERE p.product = pr.name
    AND p.buyer = 'Joe Blow')
```

“Cities where one can find companies that manufacture products bought by Joe Blow”

Nested Queries

Are these queries equivalent?

```
SELECT c.city  
FROM Company c  
WHERE c.name IN (  
    SELECT pr.maker  
    FROM Purchase p, Product pr  
    WHERE p.name = pr.product  
    AND p.buyer = 'Joe Blow')
```

```
SELECT c.city  
FROM Company c,  
Product pr,  
Purchase p  
WHERE c.name = pr.maker  
AND pr.name = p.product  
AND p.buyer = 'Joe Blow'
```

Nested Queries

Are these queries equivalent?

```
SELECT c.city  
FROM Company c  
WHERE c.name IN (  
    SELECT pr.maker  
    FROM Purchase p, Product pr  
    WHERE p.name = pr.product  
    AND p.buyer = 'Joe Blow')
```

```
SELECT c.city  
FROM Company c,  
Product pr,  
Purchase p  
WHERE c.name = pr.maker  
AND pr.name = p.product  
AND p.buyer = 'Joe Blow'
```

Beware of duplicates!

Nested Queries

```
SELECT DISTINCT c.city
FROM Company c,
Product pr,
Purchase p
WHERE c.name = pr.maker
AND pr.name = p.product
AND p.buyer = 'Joe Blow'
```

```
SELECT DISTINCT c.city
FROM Company c
WHERE c.name IN (
SELECT pr.maker
FROM Purchase p, Product pr
WHERE p.product = pr.name
AND p.buyer = 'Joe Blow')
```

Now they are equivalent (both use set semantics)

Subqueries Return Relations

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”

Subqueries Returning Relations

You can also use operations of the form:

- $s > \text{ALL } R$
- $s < \text{ANY } R$
- $\text{EXISTS } R$

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 \neq

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

Nested queries as alternatives to INTERSECT and EXCEPT

INTERSECT and EXCEPT not in some DBMSs!

```
(SELECT R.A, R.B  
FROM R)  
INTERSECT  
(SELECT S.A, S.B  
FROM S)
```



```
SELECT R.A, R.B  
FROM R  
WHERE EXISTS(  
    SELECT *  
    FROM S  
    WHERE R.A=S.A AND R.B=S.B)
```

```
(SELECT R.A, R.B  
FROM R)  
EXCEPT  
(SELECT S.A, S.B  
FROM S)
```



```
SELECT R.A, R.B  
FROM R  
WHERE NOT EXISTS(  
    SELECT *  
    FROM S  
    WHERE R.A=S.A AND R.B=S.B)
```

If R, S have no duplicates, then can write without sub-queries (HOW?)

Correlated Queries Using External Vars in Internal Subquery

Movie(title, year, director, length)

```
SELECT DISTINCT title
FROM Movie AS m
WHERE year <> ANY(
    SELECT year
    FROM Movie
    WHERE title = m.title)
```

Find movies whose title appears more than once.

Note the scoping of the variables!

Note also: this can still be expressed as single SFW query...

```
SELECT DISTINCT title
FROM Movie AS m1, Movie AS m2
WHERE m1.title = m2.title and
m1.year <> m2.year
```

Movie(title, year, director, length)

Internal Subquery

```
SELECT DISTINCT title
FROM Movie AS m
WHERE year <> ANY(
    SELECT year
    FROM Movie
    WHERE title = m.title)
```

Find movies whose title appears more than once.

Note the scoping of the variables!

Note also: this can still be expressed as single SFW query...

Complex Correlated Query

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

```
SELECT DISTINCT x.name, x.maker
FROM Product AS x
WHERE x.price > ALL(
    SELECT y.price
    FROM Product AS y
    WHERE x.maker = y.maker
    AND y.year < 1972)
```

Find products (and their manufacturers) that are more expensive than all products made by the same manufacturer before 1972

Can be very powerful (also much harder to optimize)

Basic SQL Summary

- SQL provides a high-level declarative language for manipulating data (DML)
- The workhorse is the SFW block
- Set operators are powerful but have some subtleties
- Powerful, nested queries also allowed.

Activity-4-1.ipynb

2. Aggregation & GROUP BY

What you will learn about in this section

1. Aggregation operators
2. GROUP BY
3. GROUP BY: with HAVING, semantics
4. ACTIVITY: Fancy SQL Pt. I

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?*

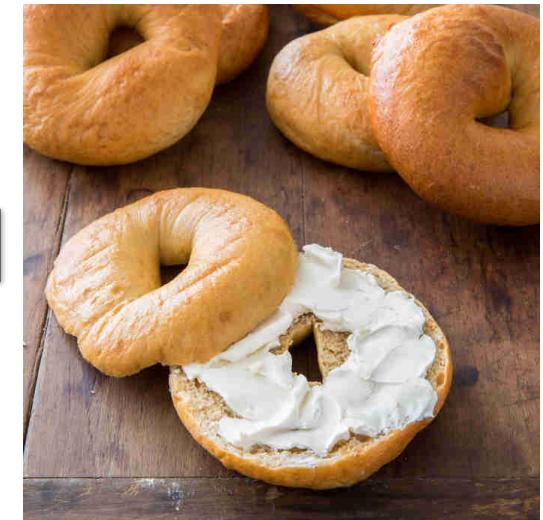
We probably want:

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

More Examples

```
Purchase(product, date, price, quantity)
```

```
SELECT SUM(price * quantity)
FROM Purchase
```



What do these mean?

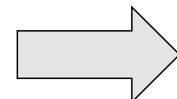
```
SELECT SUM(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```

Simple Aggregations

Purchase

Product	Date	Price	Quantity
bagel	10/21	1	20
banana	10/3	0.5	10
banana	10/10	1	10
bagel	10/25	1.50	20

```
SELECT SUM(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```



50 (= 1*20 + 1.50*20)

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...

Grouping and Aggregation

Semantics of the query:

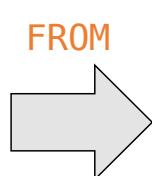
1. Compute the **FROM** and **WHERE** clauses

2. Group by the attributes in the **GROUP BY**

3. Compute the **SELECT** clause: grouped attributes and aggregates

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


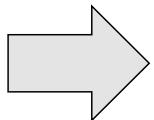
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

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

GROUP BY

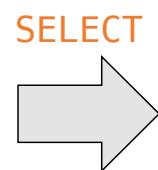


Product	Date	Price	Quantity
Bagel	10/21	1	20
	10/25	1.50	20
Banana	10/3	0.5	10
	10/10	1	10

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



Product	TotalSales
Bagel	50
Banana	15

GROUP BY v.s. Nested Quereis

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

```
SELECT DISTINCT x.product,  
              (SELECT Sum(y.price*y.quantity)  
               FROM Purchase y  
              WHERE x.product = y.product  
                AND y.date > '10/1/2005') AS TotalSales  
FROM        Purchase x  
WHERE       x.date > '10/1/2005'
```

HAVING Clause

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

HAVING clauses contains conditions on aggregates

Whereas WHERE clauses condition on individual tuples...

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

General form of Grouping and Aggregation

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

Why?

- S = Can ONLY contain attributes a_1, \dots, a_k and/or aggregates over other attributes
- C_1 = is any condition on the attributes in R_1, \dots, R_n
- C_2 = is any condition on the aggregate expressions

General form of Grouping and Aggregation

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

Evaluation steps:

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

Group-by v.s. Nested Query

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

- Find authors who wrote ≥ 10 documents:
- Attempt 1: with nested queries

```
SELECT DISTINCT Author.name  
FROM Author  
WHERE COUNT(  
    SELECT Wrote.url  
    FROM Wrote  
    WHERE Author.login = Wrote.login) > 10
```

This is
SQL by
a novice

Group-by v.s. Nested Query

- Find all authors who wrote at least 10 documents:
- Attempt 2: SQL style (with GROUP BY)

```
SELECT Author.name  
FROM Author, Wrote  
WHERE Author.login = Wrote.login  
GROUP BY Author.name  
HAVING COUNT(Wrote.url) > 10
```

This is
SQL by
an expert

No need for **DISTINCT**: automatically from **GROUP BY**

Group-by vs. Nested Query

Which way is more efficient?

- Attempt #1- *With nested*: How many times do we do a SFW query over all of the Wrote relations?
- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be much more efficient!

Activity-4-2.ipynb