

3. SQL Advanced (Part II)

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

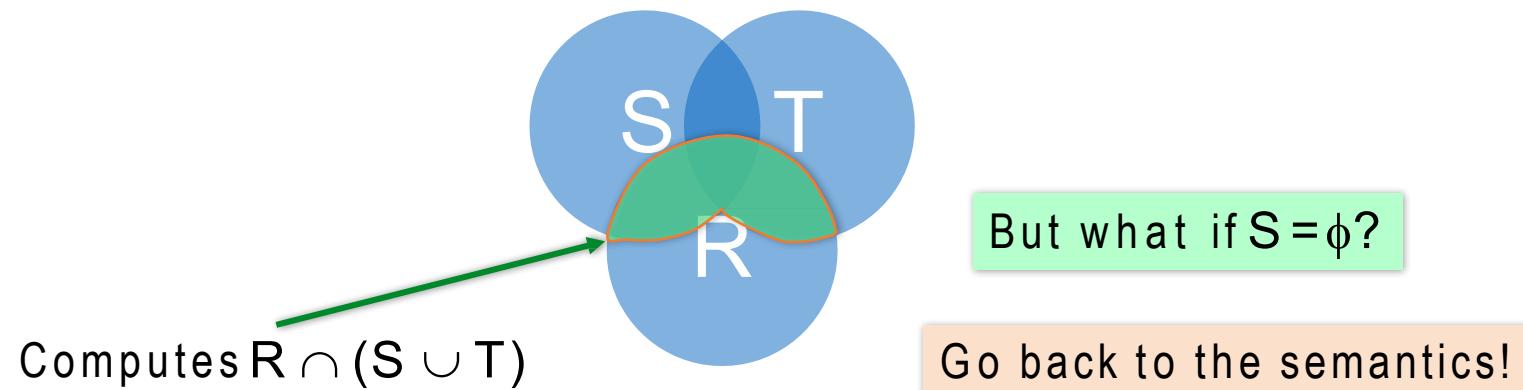
An Unintuitive Query

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

What does it compute?

An Unintuitive Query

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



An Unintuitive Query

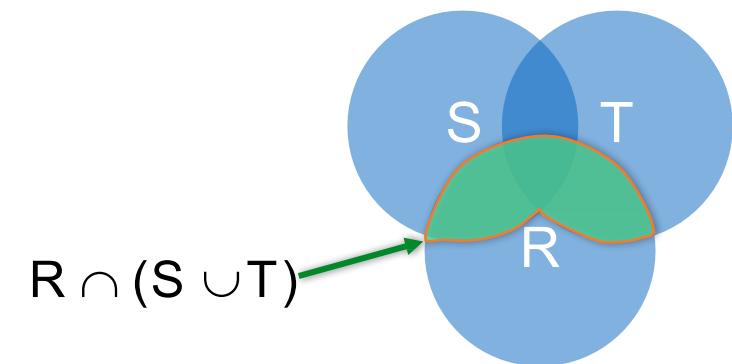
```
SELECT DISTINCT R.A  
FROM   R, S, T  
WHERE  R.A=S.A OR R.A=T.A
```

- Recall the semantics!
 1. Take cross-product
 2. Apply selections / conditions
 3. Apply projection
- If $S = \{\}$, then the cross product of $R, S, T = \{\}$, and the query result $= \{\}!$

Must consider semantics here.
Are there more explicit way to do set operations like this?

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
```



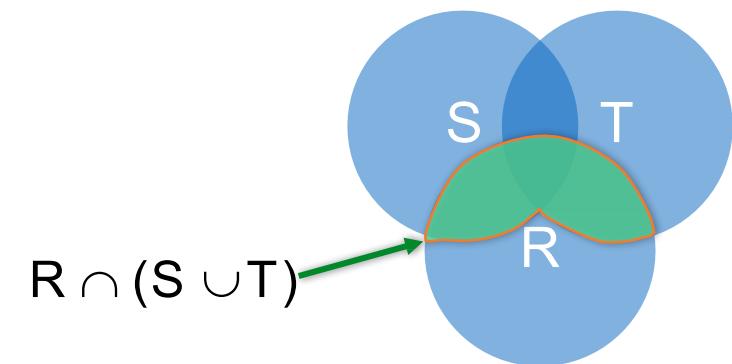
- Semantics:
 1. Take cross-product
 2. Apply selections / conditions
 3. Apply projection

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

If-then statements!

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

$\lambda X \ni$ “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

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

 \cup **Multiset Y**

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

 $=$ **Multiset Z**

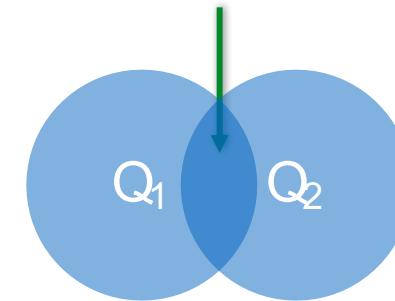
Tuple	
(1, a)	7
(1, b)	1
(2, c)	5
(1, d)	2

For sets,
this is **union**

Multiset Operations in SQL

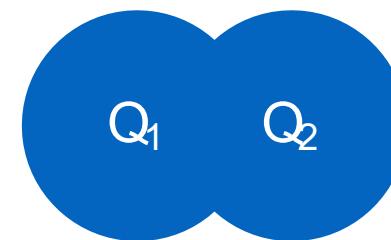
Explicit Set Operators: INTERSECT

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



UNION

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

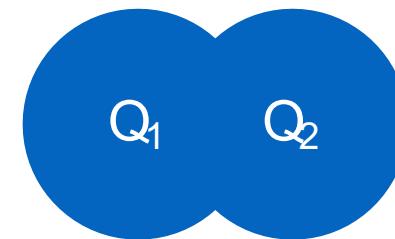


Why aren't there duplicates?

What if we want duplicates?

UNION ALL

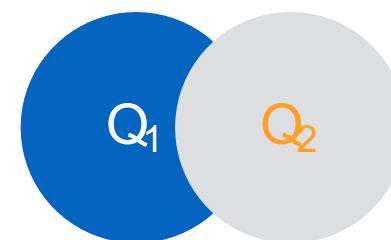
```
SELECT R.A  
FROM R, S  
WHERE R.A=S.A  
UNION ALL  
SELECT R.A  
FROM R, T  
WHERE R.A=T.A
```



*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
```



*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 factor 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 Returning 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

Is this query equivalent?

```
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

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

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

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

<> means !=

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?)

A question for Database Fans & Friends

Can we express the previous nested queries as single SFW queries?

Hint: show that all SFW queries are **monotone** (roughly: more tuples, more answers).

- A query with **ALL** is often not monotone

Correlated Queries

```
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...

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.

Pr-3-1.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?

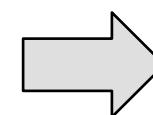
```
SELECT SUM(price * quantity)
FROM Purchase
WHERE product = 'bagel'
```

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

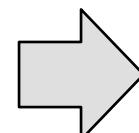
Semantics of the query:

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



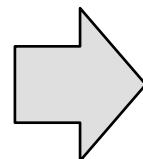
Product	Date	Price	Quantity
Bagel	10/21	1	20
Bagel	10/25	1.50	20
Banana	10/3	0.5	10
Banana	10/10	1	10

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
```

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**...

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!

Pr-3-2.ipynb

3. Advanced SQL-izing

What you will learn about in this section

1. Quantifiers
2. NULLs
3. Outer Joins
4. ACTIVITY: Fancy SQL Pt. II

Quantifiers

```
Product(name, price, company)  
Company(name, city)
```

```
SELECT DISTINCT Company.cname  
FROM Company, Product  
WHERE Company.name = Product.company  
AND Product.price < 100
```

Find all companies
that make some
products with price
< 100

An existential quantifier is a
logical quantifier (roughly) of
the form “there exists”

Existential: easy ! J

Quantifiers

```
Product (name, price, company)
Company (name, city)
```

Find all companies
with products all
having price < 100

```
SELECT DISTINCT Company.cname
FROM Company
WHERE Company.name NOT IN (
    SELECT Product.company
    FROM Product WHERE price >= 100)
```



Find all companies
that make only
products with price
< 100

A universal quantifier is of
the form “for all”

Universal: hard ! L

NULLS in SQL

- Whenever we don't have a value, we can put a NULL
- Can mean many things:
 - Value does not exists
 - Value exists but is unknown
 - Value not applicable
 - Etc.
- The schema specifies for each attribute if can be null (*nullable* attribute) or not
- How does SQL cope with tables that have NULLs?

Null Values

For numerical operations, NULL -> NULL:

- If $x = \text{NULL}$ 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 $x = \text{NULL}$ then $x = \text{"Joe"}$ is UNKNOWN

Null Values

- $C1 \text{ AND } C2 = \min(C1, C2)$
- $C1 \text{ OR } C2 = \max(C1, C2)$
- $\text{NOT } C1 = 1 - C1$

```
SELECT *
FROM Person
WHERE (age < 25)
    AND (height > 6 AND weight > 190)
```

Won't return e.g.
(age=20
height=NULL
weight=200)!

Rule in SQL: include only tuples that yield TRUE (1.0)

Null Values

Unexpected behavior:

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

Some Persons are not included !

Null Values

Can test for NULL explicitly:

- `x ISNULL`
- `x IS NOT NULL`

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

Now it includes all Persons!

RECAP: Inner Joins

By default, joins in SQL are “**inner joins**”:

```
Product(name, category)  
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store  
FROM Product  
JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store  
FROM Product, Purchase  
WHERE Product.name = Purchase.prodName
```

Both equivalent:
Both INNER JOINS!

Inner Joins + NULLS = Lost data?

By default, joins in SQL are “**inner joins**”:

```
Product(name, category)
Purchase(prodName, store)
```

```
SELECT Product.name, Purchase.store
FROM   Product
       JOIN Purchase ON Product.name = Purchase.prodName
```

```
SELECT Product.name, Purchase.store
FROM   Product, Purchase
WHERE  Product.name = Purchase.prodName
```

However: Products that never sold (with no Purchase tuple) will be lost!

Outer Joins

An **outer join** returns tuples from the joined relations that don't have a corresponding tuple in the other relations

- I.e. If we join relations A and B on $a.X = b.X$, and there is an entry in A with $X=5$, but none in B with $X=5$...
 - A LEFT OUTER JOIN will return a tuple (a, NULL) !

Left outer joins in SQL:

```
SELECT Product.name, Purchase.store
FROM   Product
LEFT OUTER JOIN Purchase ON
Product.name = Purchase.prodName
```

Now we'll get products even if they didn't sell

INNER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM Product
INNER JOIN Purchase
ON Product.name = Purchase.prodName
```

Note: another equivalent way to write
an INNER JOIN!



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

LEFT OUTER JOIN:

Product

name	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

```
SELECT Product.name, Purchase.store
FROM Product
LEFT OUTER JOIN Purchase
ON Product.name = Purchase.prodName
```



name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

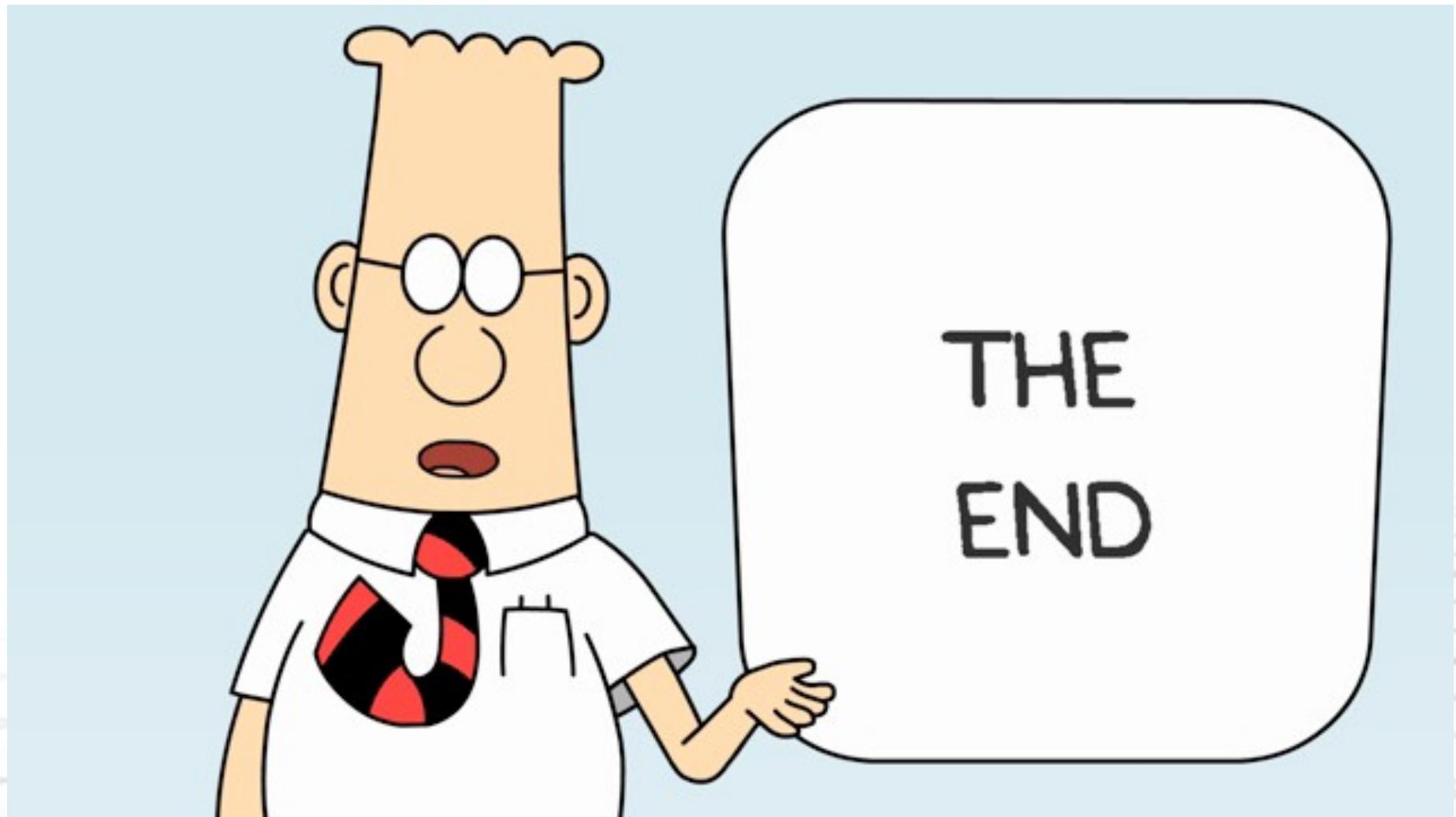
Other Outer Joins

- Left outer join:
 - Include the left tuple even if there's no match
- Right outer join:
 - Include the right tuple even if there's no match
- Full outer join:
 - Include the both left and right tuples even if there's no match

Pr-3-3.ipynb

Summary

SQL is a rich programming language that
handles the way data is processed
declaratively



출처: metachannels.com

Thank you!