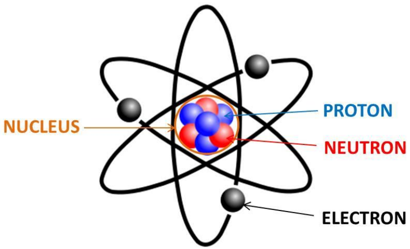
Lecture 3:
SQL Part II

This Week

- JOINs
- Set operators & nested queries
- Aggregation & GROUP BY

Project v1 rollout

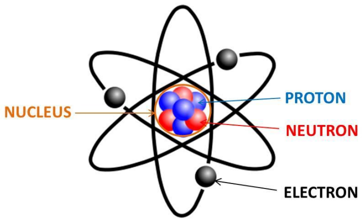
Details + Big picture



Focus on detailed, micro-examples

Take in big picture, flavor of issues, how pieces fit

Reminder on schemas



Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Students(sid*: string,* name: *string*, gpa: *float*)

Enrolled(student_id: *string,* cid: *string*, grade: *string*)

We'll use different Tables/tuples, for examples to build ideas

Data about local areas (for real-world examples)

SolarPanel(region_name*: string,* kw_total: *float*, carbon_offset_ton_metrics: *float, …*)

Census(zipcode: *string, population*: *int*, ...)

Pollution(zipcode: string, Particle_count: int…)

BikeShare(zipcode: string, trip_origin: float, trip_end: float, …)

...

Why Joins?

Option 1 (organized tables, with 10s-100s of columns)

Zipcode	Census
94305	
94040	
94041	

Zipcode	Solar
94305	
94040	
94041	

Zipcode	Bikeshare
94305	
94040	
94041	

Zipcode . . .

Option 2 ('universal table', with 1000s-millions of columns)

Zipcode	{ Census }	{ Solar }	{ BikeShare }	…….
94305				
94040				
94041				

Option 3 (One table per column, zipcode in each column)

Trade offs?

- Reads? Writes?
- 100s - thousands of applications reading/writing data

Joins

Product(PName, Price, Category, Manufacturer)

Company(CName, StockPrice, Country)

Note: we will often omit attribute types in schema definitions for brevity, but assume attributes are always atomic types

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

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
JOIN   Company
ON     Manufacturer = Cname
WHERE  Price <= 200
       AND Country='Japan'
```

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

A few more later on

Joins

Product(<u>PName</u>, Price, Category, Manufacturer)

Company(<u>CName</u>, StockPrice, Country)

Ex: 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
```

A <u>**join**</u> between tables returns all unique combinations of their tuples **which meet some specified join condition**

Joins

Product

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

Company

CName	Stock Price	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

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

PName	Price
SingleTouch	$149

Tuple Variable Ambiguity in Multi-Table

Person(<u>name</u>, address, worksfor)

Company(<u>name</u>, address)

```
1.  SELECT DISTINCT name, address
2.  FROM  Person, Company
3.  WHERE     worksfor = name
```

Which “address” does this refer to**?**

Which name”s??

Tuple Variable Ambiguity in Multi-Table

Person(<u>name</u>, address, worksfor)

Company(<u>name</u>, address)

Both equivalent ways to resolve variable ambiguity

```
SELECT DISTINCT Person.name, Person.address
FROM            Person, Company
WHERE           Person.worksfor = Company.name
```

```
SELECT DISTINCT p.name, p.address
FROM            Person p, Company c
WHERE           p.worksfor = c.name
```

Semantics of JOINs

SELECT $x_1.a_1$, $x_1.a_2$, …, $x_n.a_k$
FROM R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE Conditions(x_1,…, x_n)

Answer = {}
for x_1 **in** R_1 **do**
    for x_2 **in** R_2 **do**
    …..
        for x_n **in** R_n **do**
        if Conditions(x_1,…, x_n)
            then Answer = Answer $\cup$ {($x_1.a_1$, $x_1.a_2$, …, $x_n.a_k$)}
return Answer

Note:
This is a *multiset* union

Semantics of JOINs (2 tables)

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

1. Take **cross product**:

   $X = R \times S$

Recall: Cross product (A X B) is the set of all unique tuples in A,B
Ex: {a,b,c} X {1,2}
= {(a,1), (a,2), (b,1), (b,2), (c,1), (c,2)}

2. Apply **selections / conditions**:

   $Y = \{(r,s) \in X \mid r.A == r.B\}$

= Filtering!

3. Apply **projections** to get final output:

   $Z = (y.A,) \ for \ y \in Y$

= Returning only *some* attributes

Remembering this order is critical to understanding the output of certain queries (see later on…)

An example of SQL semantics

```
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

Output

A
3
3

Note: we say “semantics” not “execution order”

- The preceding slides show *what a join means*
- Not actually how the DBMS executes it under the covers

A Subtlety about Joins

Product(<u>PName</u>, Price, Category, Manufacturer)

Company(<u>CName</u>, StockPrice, Country)

Find all countries that manufacture some product in the 'Gadgets' category.

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

A Subtlety about Joins

Product

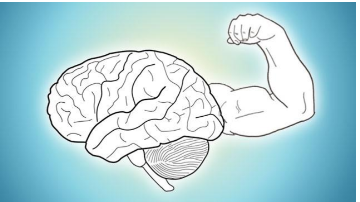
PName	Price	Category	Manuf
Gizmo	$19	Gadgets	GWorks
Powergizmo	$29	Gadgets	GWorks
SingleTouch	$149	Photography	Canon
MultiTouch	$203	Household	Hitachi

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 is the Solution?

Set Operators & Nested Queries

What you will learn about in this section

1. Multiset operators in SQL
2. Nested queries

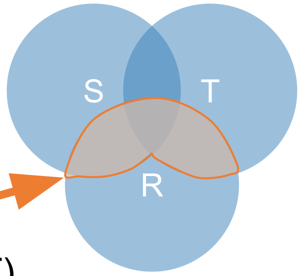
An Unintuitive Query

```
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
```



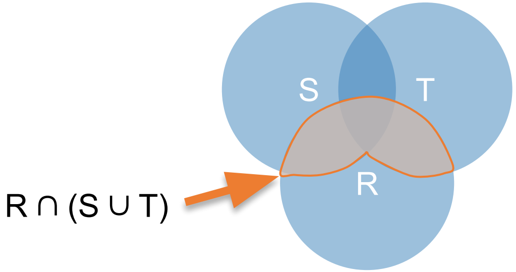
Computes R ∩ (S ∪ T)

But what if S = φ?

Go back to the semantics!

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 <u>cross-product</u>

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

  2. Apply <u>selections</u> / <u>conditions</u>

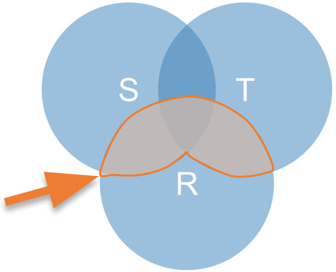
     If-then statements!

  3. Apply <u>projection</u>

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
```

R ∩ (S ∪ T)




```
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 = []?

Algebra (preview)

R ⋈ S	JOIN
+, -, … ∀, ∃	Union, intersect...
f (g (x))	Nesting
∑ #	SUM, Count, . . .

Multiset Operations

Recall Multisets

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

Multiset X

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

Equivalent Representations of a **Multiset**

Multiset X

Tuple	$\lambda(X)$
(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	$\lambda(X)$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cap$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Multiset Z

Tuple	$\lambda(Z)$
(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	$\lambda(X$
(1, a)	2
(1, b)	0
(2, c)	3
(1, d)	0

$\cup$

Multiset Y

Tuple	$\lambda(Y)$
(1, a)	5
(1, b)	1
(2, c)	2
(1, d)	2

$=$

Multiset Z

Tuple	$\lambda(Z)$
(1, a)	7
(1, b)	1
(2, c)	5
(1, d)	2

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

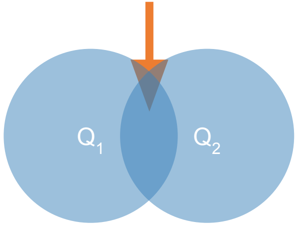
For sets,
this is **union**

Multiset Operations in SQL

Explicit Set Operators: INTERSECT

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

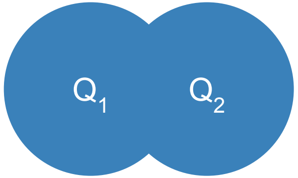
$$\{r.A \mid r.A = s.A\} \cap \{r.A \mid r.A = t.A\}$$



UNION

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

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



Why aren't there duplicates?

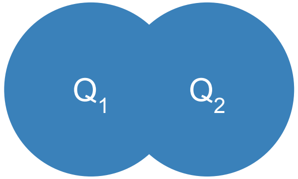
By default:
SQL retains set semantics for UNIONs, INTERSECTs!

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
```

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

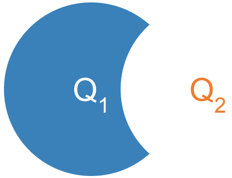


ALL indicates Multiset operations

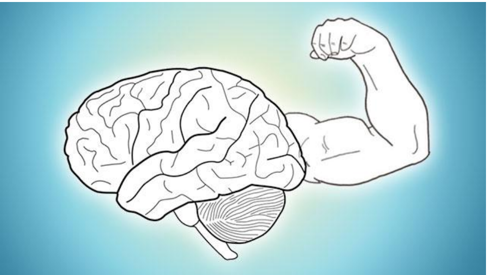
EXCEPT

```
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?



Algebra (preview)

R ⋈ S	JOIN
+, -, … ∀ , ∃	Union, intersect...
f (g (x))	Nesting
∑ #	SUM, Count, . . .

Nested queries: Sub-queries Return Relations

Another example:

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

```
SELECT pr.maker
FROM   Purchase p, Product pr
WHERE  p.product = pr.name
       AND p.buyer = ‘Mickey‘)
```

“

- Companies making products bought by Mickey”
- Location of companies?

”

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

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 = 'Mickey')
```

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

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 = 'Mickey'
```

```
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 = 'Mickey')
```

Now they are equivalent (both use set semantics)

Subqueries Return Relations

You can also use operations of the form:

- <u>s > ALL R</u>
- 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
- <u>EXISTS R</u>

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"

Nested queries as alternatives to INTERSECT and EXCEPT

INTERSECT and EXCEPT not in some DBMSs!

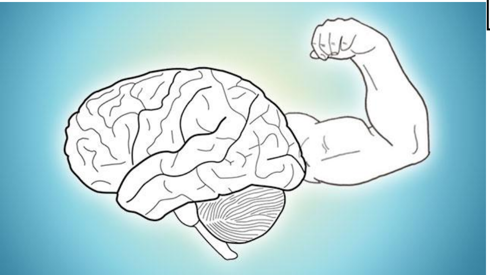
```
(SELECT R.A, R.B
 FROM   R)
INTERSECT
(SELECT S.A, S.B
 FROM   S)
```

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

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

```
(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)
```



Correlated Queries Using External Vars in Internal Subquery

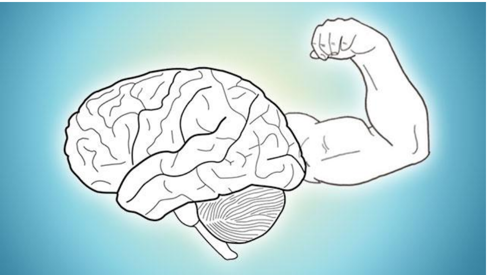
Movie(<u>title</u>, <u>year</u>, director, length)

```
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

Algebra (preview)

R * S	JOIN
+, -, … ∀ , ∃	Union, intersect...
f (g (x))	Nesting
∑ #	SUM, Count, . . .

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

Option 1: With Nested queries

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

```
        SELECT maker
        FROM   Product
        WHERE  factory_loc = ‘US’)

        SELECT maker
        FROM   Product
        WHERE  factory_loc = ‘China’)
```

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

Option 2 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!

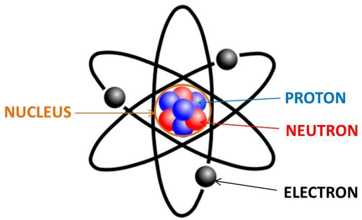
We did the INTERSECT on the wrong attributes!

Lecture 4
SQL Part II

Algebra (preview)

R * S	JOIN
+, -, … ∀ , ∃	Union, intersect...
f (g (x))	Nesting
∑ #	SUM, Count, . . .

Reminder on schemas



Product(<u>PName</u>, Price, Category, Manufacturer)

Company(<u>CName</u>, StockPrice, Country)

Students(<u>sid</u>*: string,* name: *string*, gpa: *float*)

Enrolled(<u>student_id</u>: *string,* <u>cid</u>: *string*, grade: *string*)

We'll use different Tables/tuples, for examples to build ideas

Data about local areas (for real-world examples)

SolarPanel(<u>region_name</u>*: string,* kw_total: *float*, carbon_offset_ton_metrics: *float, …*)

Census(<u>zipcode</u>: *string, population*: *int*, ...)

Pollution(<u>zipcode</u>: string, Particle_count: int…)

BikeShare(<u>zipcode</u>: string, trip_origin: float, trip_end: float, …)

...

What you will learn about in this section

1. Aggregation operators

2. GROUP BY

3. GROUP BY: with HAVING, semantics

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
```

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

What do these mean?

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

What GROUPings are possible?
- Type, Size, Color
- Number of holes
- Combination?

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

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

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

SELECT

Product	TotalSales
Bagel	50
Banana	15

GROUP BY v.s. Nested Queries

```
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      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

Why?

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

General form of Grouping and Aggregation

```
SELECT      S
FROM        R_1,…,R_n
WHERE       C_1
GROUP BY    a_1,…,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 need to compute aggregates)
4. Compute aggregates in S and return the result

Group-by v.s. Nested Query

Students(<u>sid</u>, name, gpa)
Enrolled(<u>student_id</u>, <u>cid</u>, grade)

- Find students enrolled in > 5 classes
- Attempt 1: with nested queries

```
SELECT DISTINCT Students.name
FROM    Students
WHERE  COUNT(
     SELECT cid
     FROM    Enrolled
     WHERE  Students.sid = Enrolled.student_id) > 5
```

This is SQL by a novice

Group-by v.s. Nested Query

- Attempt 2: SQL style (with GROUP BY)

```
SELECT   Students.name
FROM     Students, Enrolled
WHERE    Students.sid = Enrolled.student_id
GROUP BY Students.sid
HAVING   COUNT(Enrolled.cid) > 5
```

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 Enrolled relations?

- Attempt #2- *With group-by*: How about when written this way?

With GROUP BY can be **much** more efficient!

3. Advanced SQL-izing

What you will learn about in this section

1. Quantifiers
2. NULLs
3. Outer Joins

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 <u>some</u> products with price < 100

An **<u>existential quantifier</u>** is a logical quantifier (roughly) of the form "there exists"

Quantifiers

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

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

A **universal quantifier** is of the form “for all”

Find all companies with products all having price < 100

Equivalent

Find all companies that make only products with price < 100

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 = 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

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- 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 IS NULL
- 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
```

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Note: another equivalent way to write an INNER JOIN!

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

Reminder

SunRoof
Public dataset

Public Dataset: Solar_potential_by_postal_code.

Schema (+ sample records)

region_name	percent_covered	kw_total	carbon_offset_metric_tons
94043	97.79146031321109	215612.5	84929.0098507134 7
94041	99.05200433369447	56704.25	22189.3482386231 8

Example 1

Public dataset

Public Dataset: census_bureau_usa.population_by_zip_2010

Schema (+ sample records)

zipcode	population
99776	124
38305	49808
37086	31513
41667	720
67001	1676

Example 2

SunRoof

Public dataset On BigQuery

How many metric tons of carbon would we offset, if building in communities with 100% coverage all had solar roofs? [Run query]

Saved Query: CO2 offset in 100percent zips

```sql
#StandardSQL
SELECT
  ROUND(SUM(s.carbon_offset_metric_tons),2) total_carbon_offset_possible_metric_tons
FROM `bigquery-public-data.sunroof_solar.solar_potential_by_postal_code` s
JOIN `bigquery-public-data.census_bureau_usa.population_by_zip_2010` c
ON s.region_name = c.zipcode
WHERE
  percent_covered = 100.0
  AND c.population > 0
```

Standard SQL Dialect

Ctrl + Enter: run qu

RUN QUERY | Save Query | Save View | Format Query | Schedule Query | Show Options | Query com

Results | Details | Download as CSV | Download as JSON | Save as

Row	total_carbon_offset_possible_metric_tons
1	3689508.33

Summary

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

THANK YOU!