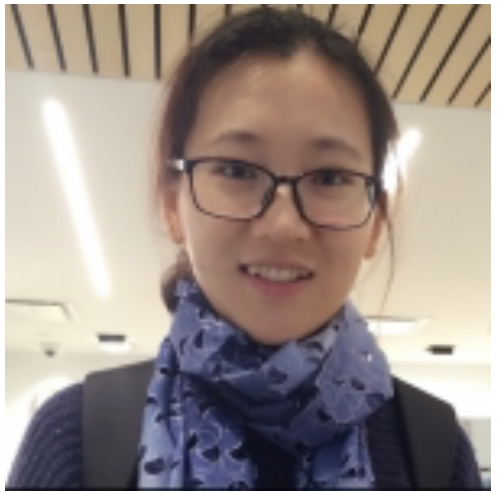
Lecture 2 (cont'd) & Lecture 3: Advanced SQL – Part I

Announcements!

1. You should be Jupyter notebook Ninjas!

2. Welcome Ting!
   - New TA-Office hours on website (room to be announced)



3. Project groups finalized!
   - If you do not have a group talk with us ASAP!

4. Problem Set #1 released

Lecture 2 (cont'd) & Lecture 3: Advanced SQL – Part I

Today's Lecture

1. Recap from Lecture 2 & Multi-table queries
   - ACTIVITY: Multi-table queries

2. Set operators & nested queries
   - ACTIVITY: Set operator subtleties

Lecture 2 (cont'd): Introduction to SQL

3. Multi-table queries

What you will learn about in this section

1. Primary keys and Foreign keys recap
2. Joins: SQL semantics
3. ACTIVITY: Multi-table queries

Keys and Foreign Keys

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

What is a foreign key vs. a key here?

A key is a **minimal subset of attributes** that acts as a unique identifier for tuples in a relation

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

If two tuples agree on the values of the key, then they must be the **same** tuple!

Keys and Foreign Keys

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

What is a foreign key vs. a key here?

A **foreign key** is an attribute (or collection of attributes) in one table that uniquely identifies a row of another table.

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

The foreign key is **defined** in a second table, but it refers to the primary key in the first table.

Declaring Foreign Keys

```
Company(CName: string, StockPrice: float, Country: string)
Product(PName: string, Price: float, Category: string, Manufacturer: string)

CREATE TABLE Product(
      pname             VARCHAR(100),
      price             FLOAT,
      category          VARCHAR(100),
      manufacturer      VARCHAR(100),
      PRIMARY KEY (pname, manufacturer),
      FOREIGN KEY (manufacturer) REFERENCES Company(cname)
)
```

Declaring Foreign Keys

```
CREATE TABLE Company(
       cname          VARCHAR(100),
       stockprice     FLOAT,
       country              VARCHAR(100),
       PRIMARY KEY (cname),
       FOREIGN KEY (cname) REFERENCES Product(pname, manufacturer)
)

CREATE TABLE Product(
       pname          VARCHAR(100),
       price          FLOAT,
       category             VARCHAR(100),
       manufacturer VARCHAR(100),
       PRIMARY KEY (pname, manufacturer)
)
```

Can we do this? What would be the problem?

Declaring Foreign Keys

```
CREATE TABLE Company(
       cname          VARCHAR(100),
       stockprice     FLOAT,
       country                VARCHAR(100),
       PRIMARY KEY (cname),
       FOREIGN KEY (cname) REFERENCES Product(pname, manufacturer)
)

CREATE TABLE Product(
       pname          VARCHAR(100),
       price          FLOAT,
       category               VARCHAR(100),
       manufacturer VARCHAR(100),
       PRIMARY KEY (pname, manufacturer)
)
```

Can we do this? What would be the problem?

We can have products without a registered company! Bad design! We'll see more next week.

Declaring Foreign Keys

```
CREATE TABLE Company(
        cname           VARCHAR(100),
        stockprice      FLOAT,
        country                 VARCHAR(100),
        PRIMARY KEY (cname),
        FOREIGN KEY (cname) REFERENCES Product(pname, manufacturer)
)

CREATE TABLE Product(
        pname           VARCHAR(100),
        price           FLOAT,
        category                VARCHAR(100),
        manufacturer VARCHAR(100),
        PRIMARY KEY (pname, manufacturer)
)
```

If the **primary key** is a set of columns (a **composite key**), then the **foreign key** also must be a set of columns that corresponds to the **composite key**.

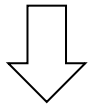
Joins

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

PName	Price
SingleTouch	$149.99

An example of SQL semantics

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

Output

A
3
3

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

Note the ***semantics*** of a join

```
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…)

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(PName, Price, Category, Manufacturer)
Company(CName, StockPrice, Country)
```

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

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

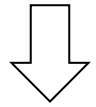
A subtlety about Joins

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's the solution ?

ACTIVITY: Lecture-2-3.ipynb

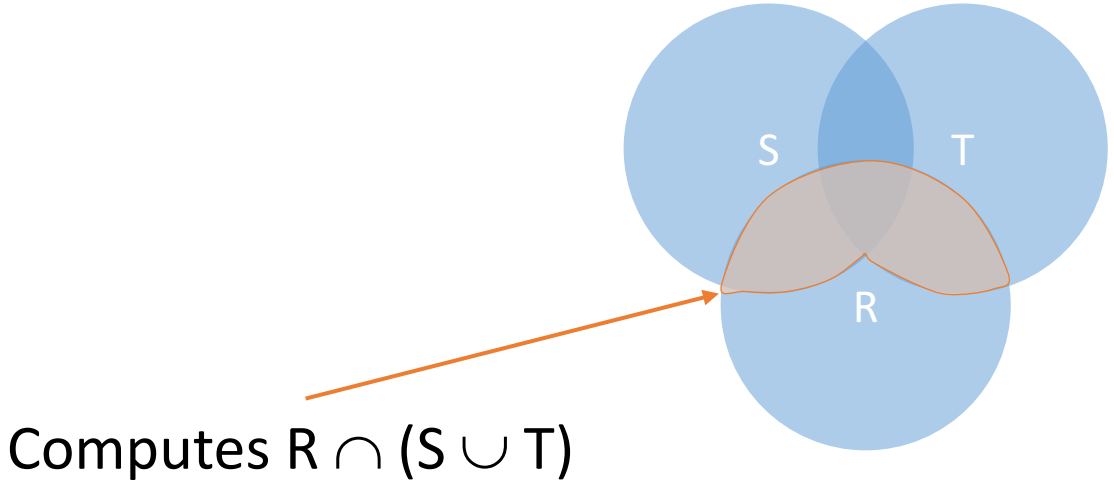
An Unintuitive Query

```
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!

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?

Lecture 3:
Advanced SQL – Part I

1. Set Operators & Nested Queries

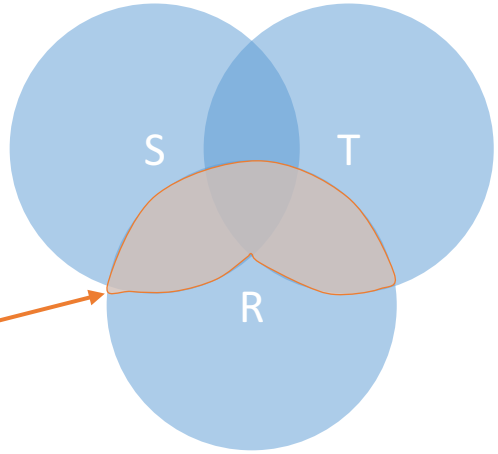
What you will learn about in this section

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?



Computes $R \cap (S \cup T)$

But what if $S = \phi$?

Go back to the semantics!

An Unintuitive Query

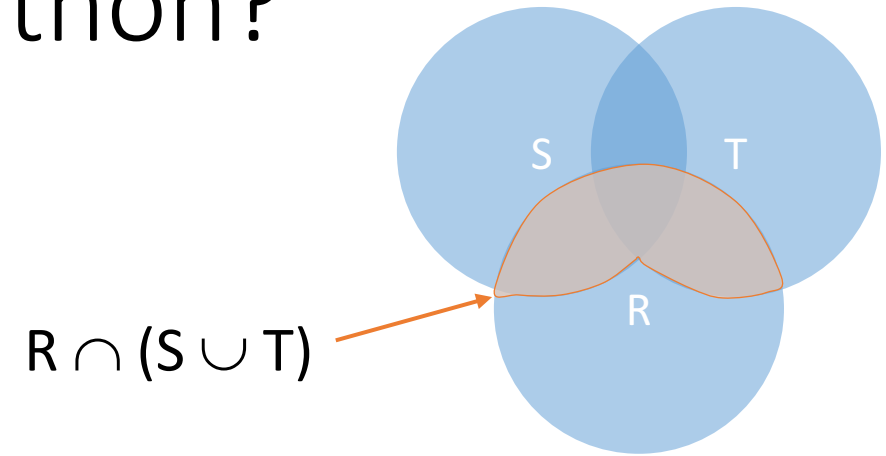
```
SELECT DISTINCT R.A
FROM   R, S, T
WHERE  R.A=S.A OR R.A=T.A
```

- 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
```



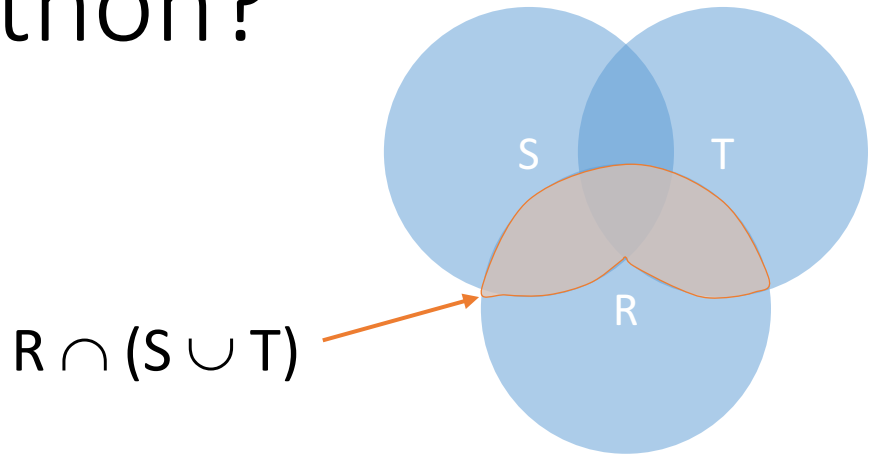
- Semantics:
  1. Take cross-product
  2. Apply selections / conditions
  3. Apply projection

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**

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

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)	5
(1, b)	1
(2, c)	3
(1, d)	2

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

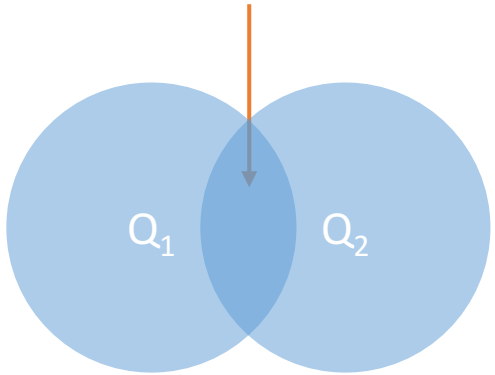
For sets,
this is **union**

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
```

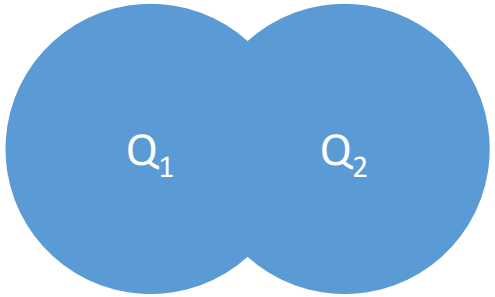
$$\{r.A \mid r.A = s.A\} \cap \{r.A \mid r.A = t.A\}$$



UNION

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

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



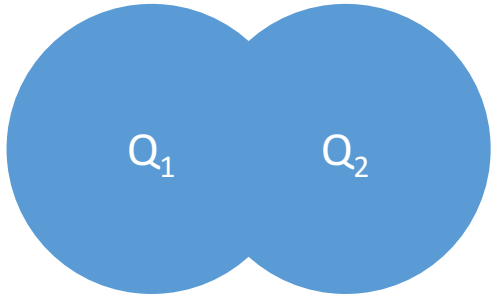
Why aren't there duplicates?

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 the Multiset disjoint union operation

Generalizing Set Operations to Multiset Operations

Multiset X

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

$\sqcup$

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

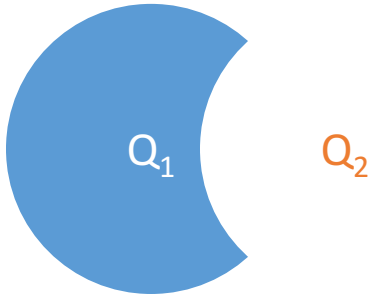
$$\lambda(Z) = \lambda(X) + \lambda(Y)$$

For sets, this is **disjoint union**

EXCEPT

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

$$\{r.A \mid r.A = s.A\} \backslash \{r.A | r.A = t.A\}$$



What is the multiset version?

$$\boldsymbol{\lambda(Z) = \lambda(X) - \lambda(Y)}$$

For elements that are in X

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 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)
```

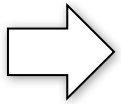
<> means !=

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!

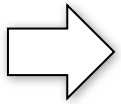
```
(SELECT R.A, R.B
 FROM   R)
INTERSECT
(SELECT S.A, S.B
 FROM   S)
```



```
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

```
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)

Activity-3-1.ipynb

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.