WISCONSIN
UNIVERSITY OF WISCONSIN–MADISON

CS639:
Data Management for Data Science

Lecture 4: SQL for Data Science

Theodoros Rekatsinas

Announcements

- Assignment 1 is due tomorrow (end of day)
  - Any questions?

- PA2 is out. It is due on the 19th
  - Start early ☺
  - Ask questions on Piazza
  - Go over activities and reading before attempting

- Out of town for the next two lectures.
  - We will resume on Feb 13th.

Today’s Lecture

1. Finish Relational Algebra (slides in previous lecture)
2. Introduction to SQL
3. Single-table queries
4. Multi-table queries
5. Advanced SQL

1. Introduction to SQL

SQL Motivation

- But why use SQL?
  - The relational model of data is the most widely used model today
    - Main Concept: the *relation*- essentially, a table

Remember: The reason for using the relational model is data independence!

Logical data independence:
protection from changes in the *logical structure of the data*

SQL is a logical, declarative query language. We use SQL because we happen to use the relational model.

Basic SQL

SQL Introduction

- SQL is a standard language for querying and manipulating data

- SQL is a **very high-level** programming language
  - This works because it is optimized well!

SQL stands for
Structured Query Language

- Many standards out there:
  - ANSI SQL, SQL92 (a.k.a. SQL2), SQL99 (a.k.a. SQL3), ….
  - Vendors support various subsets

Probably the world's most successful **parallel** programming language (multicore?)

SQL is a…

- Data Definition Language (DDL)
  - Define relational *schemata*
  - Create/alter/delete tables and their attributes

- Data Manipulation Language (DML)
  - Insert/delete/modify tuples in tables
  - Query one or more tables – discussed next!

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

A **relation** or **table** is a multiset of tuples having the attributes specified by the schema

Let’s break this definition down

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

A **multiset** is an unordered list (or: a set with multiple duplicate instances allowed)

List: [1, 1, 2, 3]
Set: {1, 2, 3}
Multiset: {1, 1, 2, 3}

i.e. no *next()*, etc. methods!

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

An **attribute** (or **column**) is a typed data entry present in each tuple in the relation

Attributes must have an ***atomic*** *type in standard SQL, i.e. not a list, set, etc.*

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

A **tuple** or **row** is a single entry in the table having the attributes specified by the schema

*Also referred to sometimes as a **record***

Tables in SQL

Product

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

The number of tuples is the **cardinality** of the relation

The number of attributes is the **arity** of the relation

Data Types in SQL

- Atomic types:
  - Characters: CHAR(20), VARCHAR(50)
  - Numbers: INT, BIGINT, SMALLINT, FLOAT
  - Others: MONEY, DATETIME, …

- Every attribute must have an atomic type
  - Hence tables are flat

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)
```

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

Product(<u>Pname</u>: *string*, Price: *float*, Category: *string*, <u>Manufacturer</u>: *string*)

Key constraints

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

- A key is an implicit constraint on which tuples can be in the relation
  - i.e. if two tuples agree on the values of the key, then they must be the same tuple!

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

1. Which would you select as a key?
2. Is a key always guaranteed to exist?
3. Can we have more than one key?

NULL and NOT NULL

- To say "don't know the value" we use NULL
  - NULL has (sometimes painful) semantics, more details later

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

sid	name	gpa
123	Bob	3.9
143	Jim	NULL

Say, Jim just enrolled in his first class.

In SQL, we may constrain a column to be NOT NULL, e.g., "name" in this table

General Constraints

- We can actually specify arbitrary assertions
  - E.g. *"There cannot be 25 people in the DB class"*

- In practice, we don't specify many such constraints. Why?
  - <u>Performance!</u>

Whenever we do something ugly (or avoid doing something convenient) it's for the sake of performance

Go over Activity 2-1

2. Single-table queries

SQL Query

- 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.

Simple SQL Query: Selection

<u>Selection</u> is the operation of filtering a relation's tuples on some condition

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT *
FROM   Product
WHERE  Category = 'Gadgets'
```

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks

Simple SQL Query: Projection

<u>Projection</u> is the operation of producing an output table with tuples that have a subset of their prior attributes

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```

PName	Price	Manufacturer
Gizmo	$19.99	GizmoWorks
Powergizmo	$29.99	GizmoWorks

Notation

Input schema

Product(PName, Price, Category, Manfacturer)

```
SELECT Pname, Price, Manufacturer
FROM   Product
WHERE  Category = 'Gadgets'
```

Output schema

Answer(PName, Price, Manfacturer)

A Few Details

- SQL **commands** are case insensitive:
  - Same: SELECT, Select, select
  - Same: Product,   product

- **Values** are **not:**
  - Different: 'Seattle', 'seattle'

- Use single quotes for constants:
  - 'abc' - yes
  - "abc" - no

LIKE: Simple String Pattern Matching

```
SELECT *
FROM   Products
WHERE  PName LIKE '%gizmo%'
```

- s **LIKE** p: pattern matching on strings
- p may contain two special symbols:
  - % = any sequence of characters
  - _ = any single character

DISTINCT: Eliminating Duplicates

```
SELECT DISTINCT Category
FROM   Product
```

Category
Gadgets
Photography
Household

Versus

```
SELECT Category
FROM   Product
```

Category
Gadgets
Gadgets
Photography
Household

ORDER BY: Sorting the Results

```
SELECT    PName, Price, Manufacturer
FROM      Product
WHERE     Category='gizmo' AND Price > 50
ORDER BY  Price, PName
```

Ties are broken by the second attribute on the ORDER BY list, etc.

Ordering is ascending, unless you specify the DESC keyword.

Go over Activity 2-2

3. Multi-table queries

Foreign Key constraints

- Suppose we have the following schema:

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

- And we want to impose the following constraint:
  - 'Only bona fide students may enroll in courses' i.e. a student must appear in the Students table to enroll in a class

student_id alone is not a key- what is?

Students

sid	name	gpa
101	Bob	3.2
123	Mary	3.8

Enrolled

student_id	cid	grade
123	564	A
123	537	A+

We say that student_id is a **foreign key** that refers to Students

Declaring Foreign Keys

```
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)
)
```

Foreign Keys and update operations

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

- What if we insert a tuple into Enrolled, but no corresponding student?
  - INSERT is rejected (foreign keys are constraints)!

- What if we delete a student?

  DBA chooses (syntax in the book)

  1. Disallow the delete
  2. Remove all of the courses for that student
  3. *SQL allows a third via NULL (not yet covered)*

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?

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

Joins

```
Product(PName, Price, Category, Manufacturer)

Company(CName, 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
```

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

Joins

```
Product(PName, Price, Category, Manufacturer)

Company(CName, 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)

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
```

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

Tuple Variable Ambiguity in Multi-Table

```
Person(name, address, worksfor)

Company(name, address)
```

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

Which "address" does this refer to?

Which "name"s??

Tuple Variable Ambiguity in Multi-Table

```
Person(name, address, worksfor)

Company(name, 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
```

Meaning (Semantics) of SQL Queries

```
SELECT x1.a1, x1.a2, …, xn.ak
FROM   R1 AS x1, R2 AS x2, …, Rn AS xn
WHERE  Conditions(x1,…, xn)
```

Almost never the *fastest* way to compute it!

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

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 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

Go over Activity 2-3

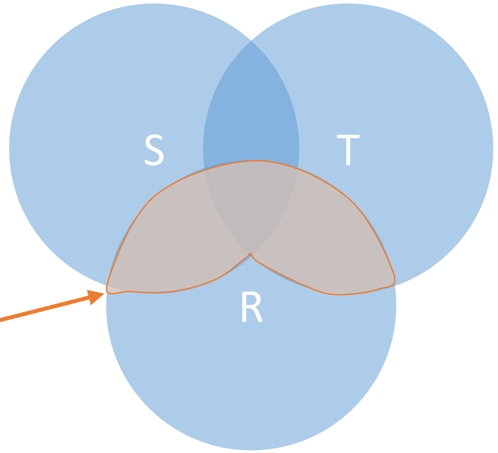
4. Advanced SQL

Set Operators and 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?



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 = {}!

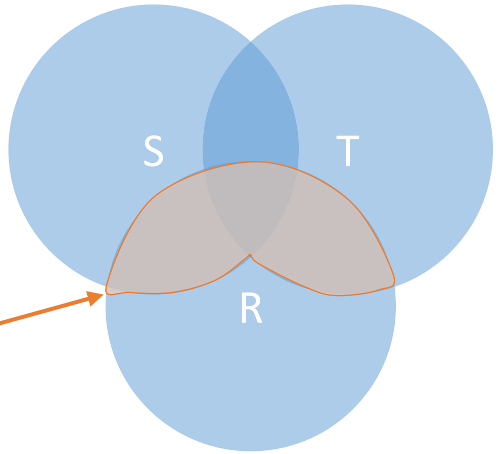
Must consider semantics here.

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
```

$R \cap (S \cup T)$



- 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>

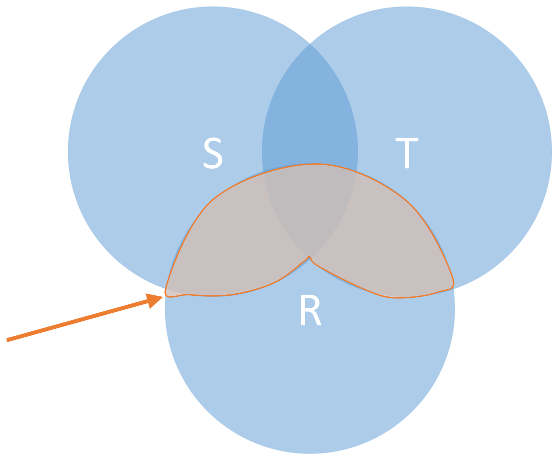
     If-then statements!

  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
```

$R \cap (S \cup T)$




```
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 = []?

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 <u>Multiset</u>

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

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

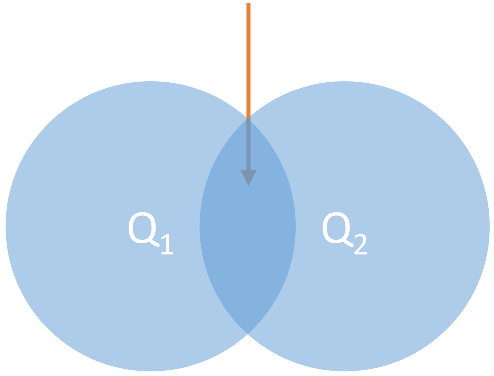
For sets,
this is **union**

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
```

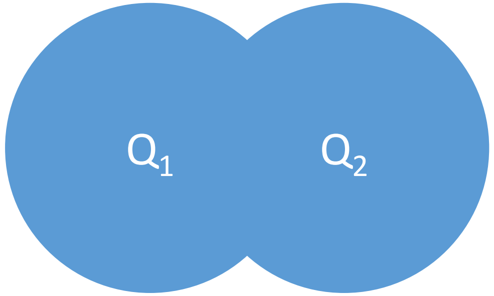
$$\{r.A \mid r.A = s.A\} \cap \{r.A \mid r.A = t.A\}$$



UNION

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

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



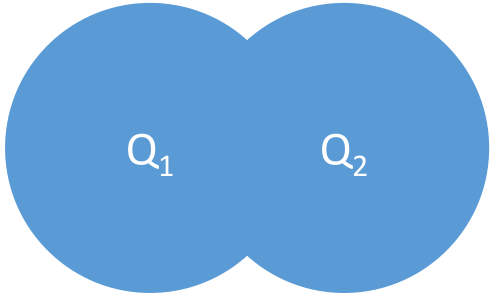
Why aren't there duplicates?

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

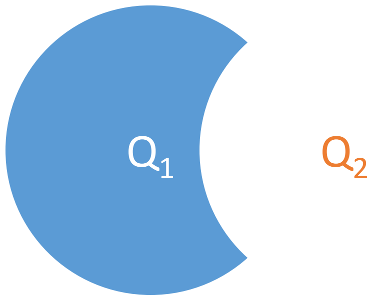
$$\lambda(Z) = \lambda(X) + \lambda(Y)$$

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
```

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



What is the multiset version?

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

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 <u>extremely</u> 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
- <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)
```

```
(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

```
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)

Go over Activity 3-1

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.