Database Design and Implementation

CS 645

SQL and Datalog

What you need

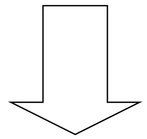
- Refresh your SQL:
  - http://sqlzoo.net
- Practice!
  - You probably already have sqlite.
  - Instructions to install Postgres on the assignments page on the website.
- Homework assignment 1!

Simple SQL query

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

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



$\sigma_{category='Gadgets'}$

Selection

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

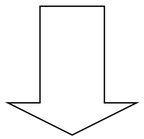
Simple SQL query

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

$\pi_{pname,price,manufacturer}$

```
SELECT  pName, price, manufacturer
FROM    Product
WHERE   price > 100
```



$\sigma_{price>100}$

Selection
& Projection

PName	Price	Manufacturer
SingleTouch	$149.99	Canon
MultiTouch	$203.99	Hitachi

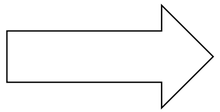
Eliminating duplicates

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

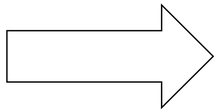
Set vs. Bag semantics

```
SELECT  category
FROM    Product
```



Category
Gadgets
Gadgets
Photography
Household

```
SELECT  DISTINCT category
FROM    Product
```



Category
Gadgets
Photography
Household

Ordering the results

```
SELECT      pName, price, manufacturer
FROM        Product
WHERE       category='Gadgets'
            and price > 10
ORDER BY    price, pName
```

- Ties in *price* attribute broken by *pname* attribute

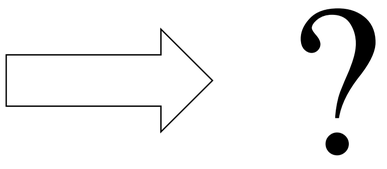
- Ordering is ascending by default. Descending:

```
... ORDER BY price, pname DESC
```

Product

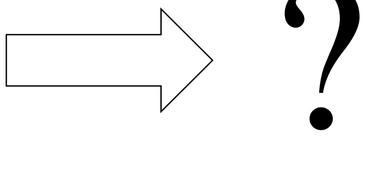
PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   DISTINCT category
FROM     Product
ORDER BY category
```



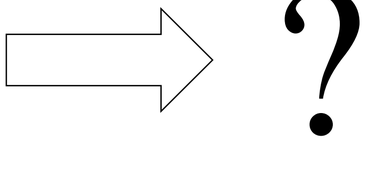
?

```
SELECT   category
FROM     Product
ORDER BY pName
```



?

```
SELECT   DISTINCT category
FROM     Product
ORDER BY pName
```

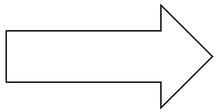


?

Product

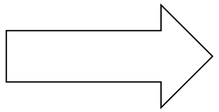
PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

```
SELECT   DISTINCT category
FROM     Product
ORDER BY category
```



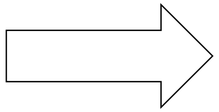
Category
Gadgets
Household
Photography

```
SELECT   category
FROM     Product
ORDER BY pName
```



Category
Gadgets
Household
Gadgets
Photography

```
SELECT   DISTINCT category
FROM     Product
ORDER BY pName
```



Syntax error*

Joins

Product (pName, price, category, manufacturer)
Company (cName, stockPrice, country)

Q: 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
```

Join between Product and Company

Joins

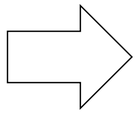
Product

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

Company

CName	StockPrice	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
```



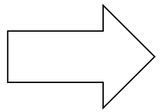
PName	Price
SingleTouch	$149.99

Semantics are tricky…

What do these queries compute?

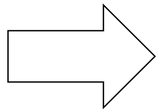
R(a), S(a), T(a)

```
SELECT  DISTINCT R.a
FROM    R, S
WHERE   R.a=S.a
```



Returns R ∩ S

```
SELECT  DISTINCT R.a
FROM    R, S, T
WHERE   R.a=S.a
        or R.a=T.a
```



If S ≠ ∅ and T ≠ ∅
then returns R ∩ (S ∪ T)
else returns ∅

Formal semantics of SQL queries

```
SELECT  a1, a2, …, ak
FROM    R1 as x1, R2 as x2, …, Rn as xn
WHERE   Conditions
```

Conceptual evaluation strategy (nested for loops):

```
Answer = {}
for x1 in R1 do
    for x2 in R2 do
        …..
            for xn in Rn do
                if Conditions
                    then Answer = Answer ∪ {(a1,…,ak)}
return Answer
```

Joins introduce duplicates

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

Company

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

```
SELECT  country
FROM    Product, Company
WHERE   manufacturer = cName
        and category = 'Gadgets'
```

Country
USA
USA

Q: Find all countries that manufacture some product in the 'Gadgets' category!

Remember to use DISTINCT

Subqueries

- A subquery is a SQL query nested inside a larger query
- Such inner-outer queries are called nested queries
- A subquery may occur in:
  - A SELECT clause
  - A FROM clause
  - A WHERE clause
- Rule of thumb: avoid writing nested queries when possible; keep in mind that sometimes it's impossible

1. Subqueries in SELECT

Product (pname, price, cid)
Company (cid, cname, city)

Q: For each product return the city where it is manufactured!

```
SELECT  P.pname, (SELECT  C.city
                  FROM    Company C
                  WHERE   C.cid = P.cid)
FROM    Product P
```

What happens if the subquery returns more than one city ?

Runtime error

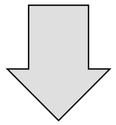
1. Subqueries in SELECT

Product (pname, price, cid)
Company (cid, cname, city)

Q: For each product return the city where it is manufactured!

```
SELECT  P.pname, (SELECT  C.city
                  FROM    Company C
                  WHERE   C.cid = P.cid)
FROM    Product P
```

"unnesting the query"



Whenever possible,
don't use nested queries

```
SELECT  P.pname, C.city
FROM    Product P, Company C
WHERE   C.cid = P.cid
```

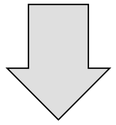
2. Subqueries in FROM

Product (pname, price, cid)
Company (cid, cname, city)

Q: Find all products whose prices are > 20 and < 30!

```
SELECT  X.pname
FROM    (SELECT *
         FROM   Product as P
         WHERE  price >20 ) as X
WHERE   X.price < 30
```

unnesting



```
SELECT  pname
FROM    Product
WHERE   price > 20 and price < 30
```

3. Subqueries in WHERE

Product (pname, price, cid)
Company (cid, cname, city)

Existential quantifiers ∃

Q: Find all companies that make <u>some</u> products with price < 100!

Using EXISTS:

```
SELECT  DISTINCT C.cname
FROM    Company C
WHERE   EXISTS  (SELECT    *
                 FROM      Product P
                 WHERE     C.cid = P.cid
                           and P.price < 100)
```

3. Subqueries in WHERE

Product (pname, price, cid)
Company (cid, cname, city)

Existential quantifiers ∃

Q: Find all companies that make <u>some</u> products with price < 100!

Using IN:

```
SELECT  DISTINCT C.cname
FROM    Company C
WHERE   C.cid IN   (SELECT P.cid
                    FROM   Product P
                    WHERE  P.price < 100)
```

3. Subqueries in WHERE

Product (pname, price, cid)
Company (cid, cname, city)

Existential quantifiers ∃

Q: Find all companies that make <u>some</u> products with price < 100!

Using ANY:

```
SELECT  DISTINCT C.cname
FROM    Company C
WHERE   100 > ANY (SELECT price
                   FROM   Product P
                   WHERE  P.cid = C.cid)
```

3. Subqueries in WHERE

Product (pname, price, cid)
Company (cid, cname, city)

Existential quantifiers ∃

Q: Find all companies that make <u>some</u> products with price < 100!

Now, let's unnest:

```
SELECT  DISTINCT C.cname
FROM    Company C, Product P
WHERE   C.cid = P.cid
        and P.price < 100
```

Existential quantifiers are easy ! ☺

3. Subqueries in WHERE

Product (pname, price, cid)
Company (cid, cname, city)

Universal quantifiers ∀

Q: Find all companies that make <u>only</u> products with price < 100!

same as:

Q: Find all companies for which <u>all</u> products have price < 100!

Universal quantifiers are more complicated ! ☹

3. Subqueries in WHERE

1. Find the other companies: i.e. they have some product ≥ 100!

```
SELECT  DISTINCT C.cname
FROM    Company C
WHERE   C.cid IN (SELECT P.cid
                  FROM         Product P
                  WHERE        P.price >= 100)
```

2. Find all companies s.t. all their products have price < 100!

```
SELECT  DISTINCT C.cname
FROM    Company C
WHERE   C.cid NOT IN (SELECT P.cid
                      FROM   Product P
                      WHERE  P.price >= 100)
```

3. Subqueries in WHERE

Product (pname, price, cid)
Company (cid, cname, city)

Universal quantifiers ∀

Q: Find all companies that make only products with price < 100!

Using NOT EXISTS:

```
SELECT  DISTINCT C.cname
FROM    Company C
WHERE   NOT EXISTS (SELECT *
                    FROM   Product P
                    WHERE  C.cid = P.cid
                           and P.price >= 100)
```

3. Subqueries in WHERE

Product (pname, price, cid)
Company (cid, cname, city)

Universal quantifiers ∀

Q: Find all companies that make only products with price < 100!

Using ALL:

```
SELECT  DISTINCT C.cname
FROM    Company C
WHERE   100 > ALL (SELECT price
                   FROM   Product P
                   WHERE  P.cid = C.cid)
```

Challenging question

- How can we unnest a universal quantifier query?

Queries that must be nested

- A query Q is monotone if:
  - Adding tuples to the input cannot remove tuples from the output
- Fact: all unnested queries are monotone
  - Proof: using the "nested for loops" semantics
- Fact: Query with universal quantifier is not monotone
  - Add one tuple violating the condition. Then not "all"...
- Consequence: we cannot unnest a query with a universal quantifier

The drinkers-bars-beers example

Likes(drinker, beer)
Frequents(drinker, bar)
Serves(bar, beer)

Challenge: write these in SQL

Find drinkers that frequent some bar that serves some beer they like.

Find drinkers that frequent only bars that serve some beer they like.

Find drinkers that frequent some bar that serves only beers they like.

Find drinkers that frequent only bars that serve only beer they like.

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 distinct

COUNT applies to duplicates, unless otherwise stated:

```
SELECT  count (category)
FROM    Product
WHERE   year > 1995
```

We probably want:

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

Simple aggregation

Purchase

Product	Price	Quantity
Bagel	3	20
Bagel	2	20
Banana	1	50
Banana	2	10
Banana	4	10

3 * 20 = 60

2 * 20 = 40

sum: 100

SQL creates attribute name

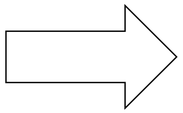
can use arithmetic expressions

```
SELECT  sum (price * quantity)
FROM    Purchase
WHERE   product = 'Bagel'
```

(No column name)
100

Grouping and Aggregation

Product	Price	Quantity
Bagel	3	20
Bagel	2	20
Banana	1	50
Banana	2	10
Banana	4	10

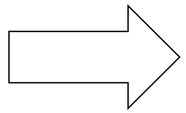


Product	TotalSales
Bagel	40
Banana	20

Find total quantities for all sales over $1, by product.

From → Where → Group By → Select

Product	Price	Quantity
Bagel	3	20
Bagel	2	20
Banana	1	50
Banana	2	10
Banana	4	10



Product	TotalSales
Bagel	40
Banana	20

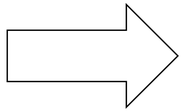
Select contains

- grouped attributes
- and aggregates

```
4 SELECT     product, sum(quantity) as TotalSales
1 FROM       Purchase
2 WHERE      price > 1
3 GROUP BY   product
```

Another example

```
SELECT    product,
          sum(quantity) as TotalSales,
          max(price) as MaxPrice
FROM      Purchase
GROUP BY  product
```

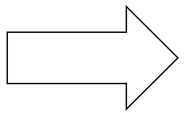


Product	TotalSales	MaxPrice
Bagel	40	3
Banana	70	4

Next, focus only on products with at least 50 sales

HAVING clause

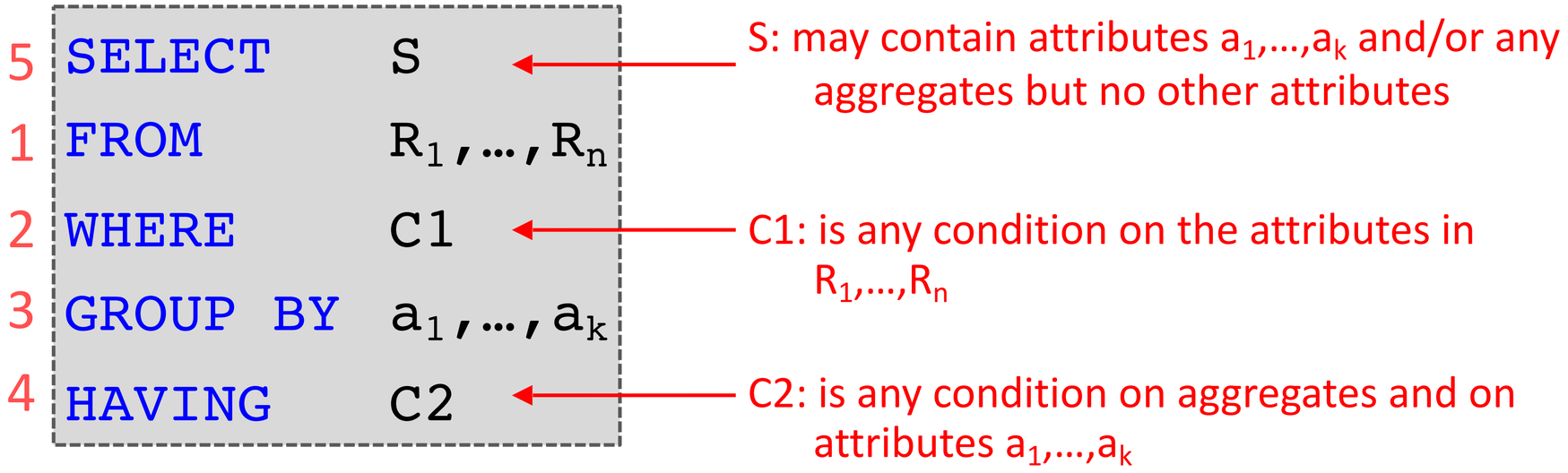
```
SELECT   product,
         sum(quantity) as TotalSales,
         max(price) as MaxPrice
FROM     Purchase
GROUP BY product
HAVING   sum(quantity) >= 50
```



Product	TotalSales	MaxPrice
Banana	70	4

Q: Similar to before, but only products with at least 50 sales.

General form of grouping and aggregation



Evaluation

1. Evaluate From → Where, apply condition C1
2. Group by the attributes $a_1,\ldots,a_k$
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

Finding witnesses

Store(sid, sname)
Product(pid, pname, price, sid)

Q: For each store, find its most expensive products

Finding the maximum price is easy…

```
SELECT    Store.sid, max(Product.price)
FROM      Store, Product
WHERE     Store.sid = Product.sid
GROUP BY Store.sid
```

But we want the “witnesses”, i.e., the products with max price

Finding witnesses

- Compute max price in a subquery
- Compare it with each product price

```
SELECT Store.sname, Product.pname
FROM   Store, Product,
       (SELECT   Store.sid as sid,
                 max(Product.price) as p
        FROM     Store, Product
        WHERE    Store.sid = Product.sid
        GROUP BY Store.sid) X
WHERE  Store.sid = Product.sid
       and Store.sid = X.sid
       and Product.price = X.p
```

Finding witnesses

There is a more concise solution here:

```
SELECT Store.sname, x.pname
FROM   Store, Product x
WHERE  Store.sid = x.sid
       and  x.price >=
            ALL (SELECT y.price
                 FROM Product y
                 WHERE Store.sid = y.sid)
```

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 it can be NULL or not
- How does SQL cope with tables that have NULLs ?

NULL values

- If x= NULL then
  - Arithmetic operations produce NULL. E.g: 4*(3-x)/7
  - Boolean conditions are also NULL. E.g: x='Joe'

- In SQL there are three boolean values:
  FALSE, TRUE, UNKNOWN

- Reasoning:
  FALSE = 0
  TRUE = 1
  UNKNOWN = 0.5

  x AND y = min(x,y)
  x OR y = max(x,y)
  NOT x = (1 – x)

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

Age	Height	Weight
20	NULL	200
~~NULL~~	~~6.5~~	~~170~~

Rule in SQL:
include only tuples that yield TRUE

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

Unexpected behavior

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

Test NULL explicitly

Outer joins

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

If we want the never-sold products, we need an "outerjoin":

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

Product

Name	Category
Gizmo	Gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Result

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

Inner join does not produce this tuple

Example

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

- Compute, for each product, the total number of sales in 'September'

```
SELECT    Product.name, count(*)
FROM      Product, Purchase
WHERE     Product.name = Purchase.prodName
          and Purchase.month = 'September'
GROUP BY Product.name
```

What's wrong?

Example

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

- Compute, for each product, the total number of sales in 'September'

```
SELECT    Product.name, count(*)
FROM      Product LEFT OUTER JOIN Purchase ON
          Product.name = Purchase.prodName
      and Purchase.month = 'September'
GROUP BY Product.name
```

What's wrong?

Example

Product(<u>name</u>, category)
Purchase(prodName, month, store)

- Compute, for each product, the total number of sales in 'September'

We need to use the attribute to get the correct 0 count.

```
SELECT    Product.name, count(month)
FROM      Product LEFT OUTER JOIN Purchase ON
          Product.name = Purchase.prodName
      and Purchase.month = 'September'
GROUP BY Product.name
```

Datalog

Datalog

- Friendly notation for queries
- Designed for recursive queries in the 80s.
- In a few commercial products:

- Today: recursion-free datalog with negation

Datalog: Facts and Rules

Facts = tuples in the database

```
Actor(34524, 'Johnny', 'Depp')
Casts(34524, 28756)
Casts(67725, 28756)
Movie(28756, 'Sweeney Todd', 2007)
Movie(28757, 'The Da Vinci Code', 2006)
```

Rules = queries

```
Q1(y) :- Movie(x,y,z), z='2007'
```

Find movies made in 2007

```
Q2(f,l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,'2007')
```

Find actors who acted in a movie in 2007

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y,'2007'),
           Casts(z,x2), Movie(x2,y2,'2006')
```

Find actors who acted in a movie in 2007 and in 2006

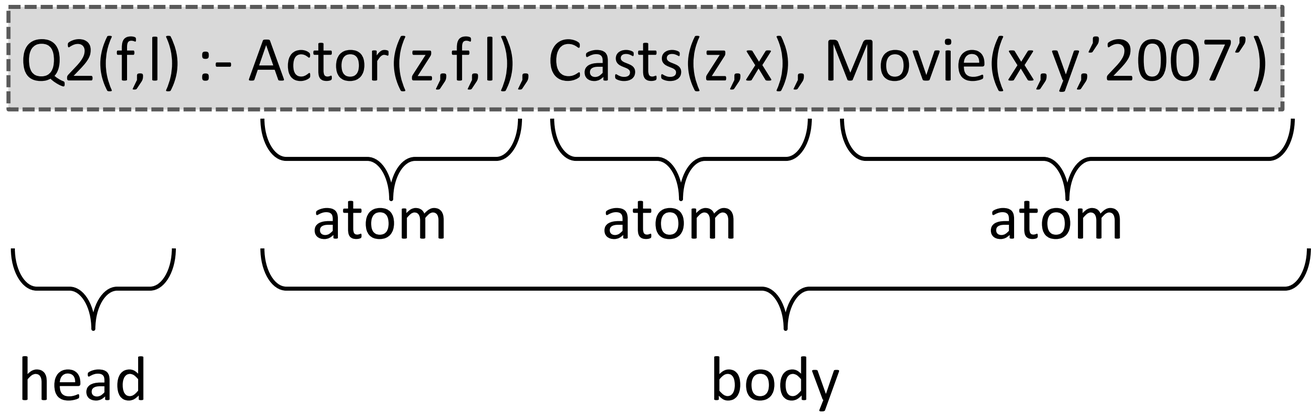
EDB and IDB

Q2(f,l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,'2007')

- Extensional Database Predicates: EDB
  - Actor, casts, movie

- Intentional Database Predicates: IDB
  - Q1, Q2, Q3

Terminology



f, l : head variables

x, y, z : existential variables

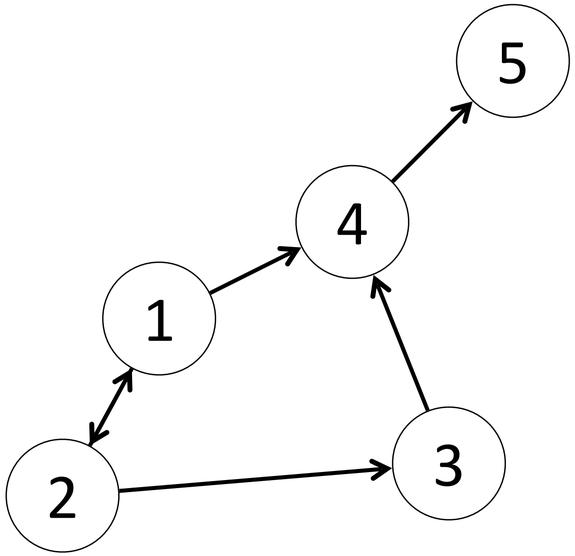
Datalog Program

union

```
B0(x) :- Actor(x,'Kevin','Bacon')
B1(x) :- Actor(x,f,l), Casts(x,z),Casts(y,z),B0(y)
B2(x) :- Actor(x,f,l), Casts(x,z),Casts(y,z),B1(y)
Q4(x) :- B1(x)
Q4(x) :- B2(x)
```

Find actors with Bacon number ≤ 2

Simple datalog programs



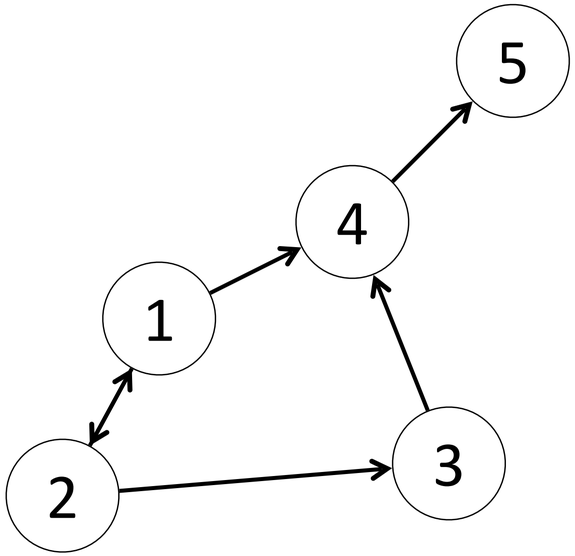
What does this compute?

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z) T(z,y)
```

R=

1	2
2	1
2	3
1	4
3	4
4	5

Simple datalog programs



What does this compute?

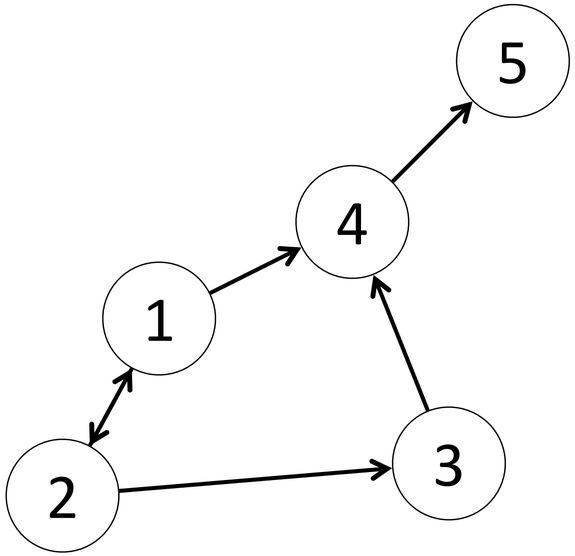
```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z) T(z,y)
```

T is initially empty

R=

1	2
2	1
2	3
1	4
3	4
4	5

Simple datalog programs



What does this compute?

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z) T(z,y)
```

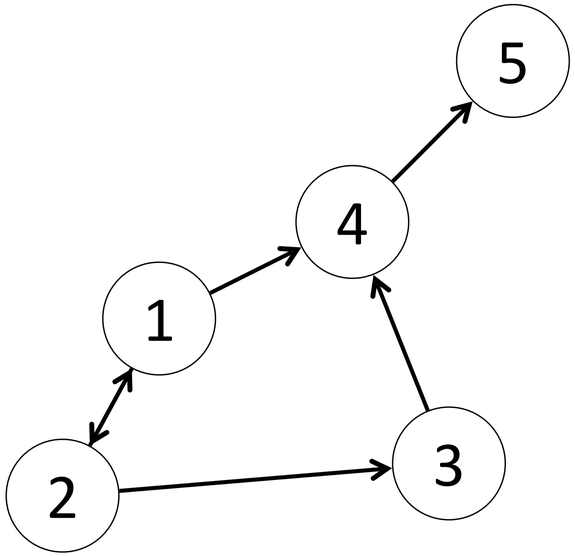
1st iteration

1	2
2	1
2	3
1	4
3	4
4	5

R=

1	2
2	1
2	3
1	4
3	4
4	5

Simple datalog programs



What does this compute?

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z) T(z,y)
```

R=

1	2
2	1
2	3
1	4
3	4
4	5

1st iteration

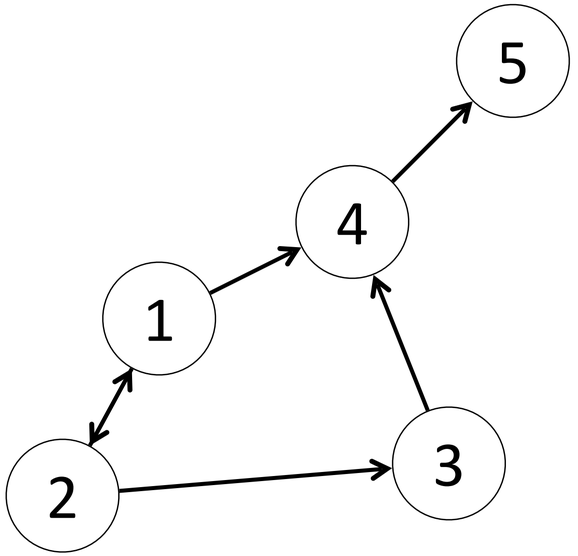
1	2
2	1
2	3
1	4
3	4
4	5

2nd iteration

1	2
2	1
2	3
1	4
3	4
4	5

1	1
1	3
2	2
2	4
1	5
3	5

Simple datalog programs



What does this compute?

```
T(x,y) :- R(x,y)
T(x,y) :- R(x,z) T(z,y)
```

R=

1	2
2	1
2	3
1	4
3	4
4	5

1st iteration

1	2
2	1
2	3
1	4
3	4
4	5

2nd iteration

1	2
2	1
2	3
1	4
3	4
4	5

1	1
1	3
2	2
2	4
1	5
3	5

3rd iteration

1	2
2	1
2	3
1	4
3	4
4	5

1	1
1	3
2	2
2	4
1	5
3	5

2	5

Datalog with Negation

```
B0(x) :- Actor(x,'Kevin','Bacon')
B1(x) :- Actor(x,f,l), Casts(x,z),Casts(y,z),B0(y)
Q5(x) :- Actor(x,f,l), not B1(x), not B0(x)
```

Find actors with Bacon number ≥ 2

Recursion and negation: ☹

EDB: R(a)

```
S(x) :- R(x), not T(x)
T(x) :- R(x), not S(x)
```

The fixpoint in unclear!

Unsafe Datalog Rules

What is unsafe about these rules?

U1(x,y) :- Movie(x,z,'2007'), y > '2000'

U2(x,u) :- Movie(x,z,'2007'), not Casts(u,x)

A rule is safe if every variable appears in some positive relational atom