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

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

**SQL Query**

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

**Selection**

$\sigma_{\text{category}='Gadgets'}$
A simple SQL query is shown on the left, which selects product names, prices, and manufacturers from the `Product` table where the price is greater than 100. The query is:

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

The table below shows the results of the query:

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

A separate table above the main query shows the full `Product` table for reference:

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
Eliminating duplicates

### Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>PowerGizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

**SELECT** category  
**FROM** Product

**SELECT** DISTINCT category  
**FROM** Product

Set vs. Bag semantics
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

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

**SQL Queries:**

1. `SELECT DISTINCT category FROM Product ORDER BY category`  
2. `SELECT category FROM Product ORDER BY pName`  
3. `SELECT DISTINCT category FROM Product ORDER BY pName`
### Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

#### SQL Queries

1. **Correct Query:**
   - ```
      SELECT DISTINCT category
      FROM Product
      ORDER BY category
   ```

2. **Incorrect Query:**
   - ```
      SELECT category
      FROM Product
      ORDER BY pName
   ```

3. **Incorrect Query:**
   - ```
      SELECT DISTINCT category
      FROM Product
      ORDER BY pName
   ```

   **Error:** Syntax error
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
```
SELECT pName, price
FROM Product, Company
WHERE manufacturer=cName and country='Japan' and price <= 200
Semantics are tricky…

What do these queries compute?

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

Returns $R \cap S$

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

If $S \neq \emptyset$ and $T \neq \emptyset$
then returns $R \cap (S \cup T)$
else returns $\emptyset$
Formal semantics of SQL queries

```
SELECT a_1, a_2, ..., a_k
FROM R_1 as x_1, R_2 as x_2, ..., R_n as x_n
WHERE Conditions
```

Conceptual evaluation strategy (nested for loops):

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

return Answer
```
Joins introduce duplicates

### Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

### Company

<table>
<thead>
<tr>
<th>CName</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

**SQL Query**

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

**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
```
Q: Find all companies that make some 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)

Q: Find all companies that make some 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)

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

Using ANY:

\[
\begin{align*}
&\text{SELECT DISTINCT } C.\text{cname} \\
&\text{FROM Company } C \\
&\text{WHERE } 100 > \text{ANY (SELECT price} \\
&\quad\text{FROM Product } P \\
&\quad\text{WHERE } P.\text{cid} = C.\text{cid})
\end{align*}
\]
3. Subqueries in WHERE

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

Q: Find all companies that make some 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)

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

same as:

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

Universal quantifiers are more complicated!
3. Subqueries in WHERE

1. Find the other companies: i.e. they have some product $\geq 100$!

```sql
SELECT DISTINCT C.cname
FROM Company C
WHERE C.cid IN (SELECT P.cid
    FROM Product P
    WHERE P.price $\geq$ 100)
```

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

```sql
SELECT DISTINCT C.cname
FROM Company C
WHERE C.cid NOT IN (SELECT P.cid
    FROM Product P
    WHERE P.price $\geq$ 100)
```
3. Subqueries in WHERE

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)

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

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.

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

Challenge: write these in SQL
### Aggregation

SQL supports several aggregation operations:

- **sum**, **count**, **min**, **max**, **avg**

Except **count**, all aggregations apply to a single attribute.
COUNT applies to duplicates, unless otherwise stated:

\[
\text{SELECT count (category) FROM Product WHERE year > 1995}
\]

We probably want:

\[
\text{SELECT count (DISTINCT category) FROM Product WHERE year > 1995}
\]
# Simple aggregation

## Purchase

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

\[
\begin{align*}
3 \times 20 &= 60 \\
2 \times 20 &= 40 \\
\text{sum: } 100
\end{align*}
\]

SQL creates attribute name (No column name)

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

100

SQL can use arithmetic expressions
Find total quantities for all sales over $1, by product.
From → Where → Group By → Select

### SQL Query

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

### Result Table

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>2</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>1</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>40</td>
</tr>
<tr>
<td>Banana</td>
<td>20</td>
</tr>
</tbody>
</table>

Select contains
- grouped attributes
- and aggregates
Another example

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

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
<th>MaxPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>40</td>
<td>3</td>
</tr>
<tr>
<td>Banana</td>
<td>70</td>
<td>4</td>
</tr>
</tbody>
</table>

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
```

<table>
<thead>
<tr>
<th>Product</th>
<th>TotalSales</th>
<th>MaxPrice</th>
</tr>
</thead>
<tbody>
<tr>
<td>Banana</td>
<td>70</td>
<td>4</td>
</tr>
</tbody>
</table>

Q: Similar to before, but only products with at least 50 sales.
**General form of grouping and aggregation**

```
SELECT S FROM R_1, ..., R_n WHERE C1 GROUP BY a_1, ..., a_k HAVING C2
```

**Evaluation**

1. Evaluate From → Where, apply condition C1
2. Group by the attributes a_1, ..., 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

```sql
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 exist
- 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 \( x = \text{NULL} \) then

- Arithmetic operations produce NULL. E.g. \( 4 \times (3-x)/7 \)
- Boolean conditions are also NULL. E.g. \( x = \text{‘Joe’} \)

In SQL there are three boolean values:

FALSE, TRUE, UNKNOWN

Reasoning:

- \( \text{FALSE} = 0 \)
- \( \text{TRUE} = 1 \)
- \( \text{UNKNOWN} = 0.5 \)
- \( x \text{ AND } y = \min(x,y) \)
- \( x \text{ OR } y = \max(x,y) \)
- \( \text{NOT } x = (1 - x) \)
The SQL query is:

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

Rule in SQL: include only tuples that yield TRUE

<table>
<thead>
<tr>
<th>Age</th>
<th>Height</th>
<th>Weight</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>NULL</td>
<td>200</td>
</tr>
<tr>
<td>NULL</td>
<td>6.5</td>
<td>170</td>
</tr>
</tbody>
</table>

Unexpected behavior

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

Test NULL explicitly

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

<table>
<thead>
<tr>
<th>Name</th>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

### Purchase

<table>
<thead>
<tr>
<th>ProdName</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

### Result

<table>
<thead>
<tr>
<th>Name</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
<tr>
<td>OneClick</td>
<td>NULL</td>
</tr>
</tbody>
</table>

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?
Compute, for each product, the total number of sales in ‘September’

```sql
SELECT Product.name, count(*)
FROM Product
LEFT OUTER JOIN Purchase
ON Product.name = Purchase.prodName
and Purchase.month = 'September'
GROUP BY Product.name
```
Compute, for each product, the total number of sales in ‘September’.

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

```sql
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.
- Today: in a couple of commercial products, e.g., LogicBlox, Datomic

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

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

f, l : head variables
x, y, z : existential variables
Datalog Program

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?

\[
T(x, y) \leftarrow R(x, y) \\
T(x, y) \leftarrow R(x, z), T(z, y)
\]

\[
\begin{array}{|c|c|}
\hline
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\hline
\end{array}
\]
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 =

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Simple datalog programs

What does this compute?

\[
T(x,y) :- R(x,y) \\
T(x,y) :- R(x,z) \ T(z,y)
\]

1st iteration

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

R=

1 2
2 1
2 3
1 4
3 4
4 5
Simple datalog programs

What does this compute?

\[
T(x,y) :- R(x,y)
\]
\[
T(x,y) :- R(x,z) \land T(z,y)
\]

\[
R = \\
1 \rightarrow 2 \\
2 \rightarrow 1 \\
2 \rightarrow 3 \\
1 \rightarrow 4 \\
3 \rightarrow 4 \\
4 \rightarrow 5 \\
\]

1\text{st iteration}

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

2\text{nd iteration}

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>1</td>
<td>1</td>
</tr>
<tr>
<td>1</td>
<td>3</td>
</tr>
<tr>
<td>2</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>4</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>3</td>
<td>5</td>
</tr>
</tbody>
</table>
Simple datalog programs

What does this compute?

\[ T(x,y) :- R(x,y) \]
\[ T(x,y) :- R(x,z) \quad T(z,y) \]

1\textsuperscript{st} iteration

\[
\begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]

2\textsuperscript{nd} iteration

\[
\begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
1 & 1 \\
1 & 3 \\
2 & 2 \\
2 & 4 \\
1 & 5 \\
3 & 5 \\
2 & 5 \\
\end{array}
\]

3\textsuperscript{rd} iteration

\[
\begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
1 & 1 \\
1 & 3 \\
2 & 2 \\
2 & 4 \\
1 & 5 \\
3 & 5 \\
2 & 5 \\
\end{array}
\]
**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)

\[
\begin{align*}
S(x) & : \neg R(x), \neg T(x) \\
T(x) & : \neg R(x), \neg S(x)
\end{align*}
\]

The fixpoint is unclear!
Unsafe Datalog Rules

What is unsafe about these rules?

\[ U1(x,y) :- \text{Movie}(x,z,'2007'), y > '2000' \]

\[ U2(x,u) :- \text{Movie}(x,z,'2007'), \text{not} \text{Casts}(u,x) \]

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