Database design and implementation
CMPSCI 645

Lecture 04: SQL and Datalog
What you need

- Install Postgres.
  - Instructions are on the assignments page on the website.
  - Use it to practice

- Refresh your SQL:
  - http://sqlzoo.net
### 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:**

<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>
</tbody>
</table>
Simple SQL query

SELECT pName, price, manufacturer
FROM Product
WHERE price > 100
Eliminating duplicates

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

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

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

---

Set vs. Bag semantics

<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Photography</td>
</tr>
<tr>
<td>Household</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gadgets</td>
</tr>
<tr>
<td>Photography</td>
</tr>
<tr>
<td>Household</td>
</tr>
</tbody>
</table>
Ordering the results

```sql
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:

```sql
... 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 Query 1:**

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

**SQL Query 2:**

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

**SQL Query 3:**

```sql
SELECT DISTINCT category
FROM Product
ORDER BY pName
```
The image contains a table of products with columns for PName, Price, Category, and Manufacturer. The SQL queries and results are also shown:

1. `SELECT DISTINCT category FROM Product ORDER BY category` results in a category list: Gadgets, Household, Photography.
2. `SELECT category FROM Product ORDER BY pName` results in a category list: Gadgets, Household, Photography.
3. `SELECT DISTINCT category FROM Product ORDER BY pName` results in a category list: Gadgets, Household, Photography.

A syntax error is highlighted in red.
**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
<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
```
SELECT pName, price
FROM Product, Company
WHERE manufacturer=cName
    AND country='Japan'
    AND price <= 200
```
Semantics are tricky...

*What do these queries compute?*

- **First Query:**
  ```sql
  SELECT DISTINCT R.a
  FROM R, S
  WHERE R.a=S.a
  ```
  Returns \( R \cap S \)

- **Second Query:**
  ```sql
  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

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>PName</td>
<td>CName</td>
</tr>
<tr>
<td>Gizmo</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

<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>

**Q:** Find all countries that manufacture some product in the ‘Gadgets’ category!

Remember to use DISTINCT

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

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

\textit{Q: Find all companies that make some products with price < 100!}

Using \textbf{EXISTS}:

\begin{verbatim}
SELECT DISTINCT C.cname
FROM Company C
WHERE EXISTS (SELECT *
                FROM Product P
                WHERE C.cid = P.cid
                and P.price < 100)
\end{verbatim}
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)

Existential quantifiers $\exists$

Q: Find all companies that make some 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)

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

Universal quantifiers ∀

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

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

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)

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

Using **ALL**:

```sql
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 the *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

```sql
SELECT avg(price) 
FROM Product 
WHERE maker='Toyota'
```

```sql
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.
COUNT applies to duplicates, unless otherwise stated:

<table>
<thead>
<tr>
<th>SELECT</th>
<th>count (category)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Product</td>
</tr>
<tr>
<td>WHERE</td>
<td>year &gt; 1995</td>
</tr>
</tbody>
</table>

We probably want:

<table>
<thead>
<tr>
<th>SELECT</th>
<th>count (DISTINCT category)</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>Product</td>
</tr>
<tr>
<td>WHERE</td>
<td>year &gt; 1995</td>
</tr>
</tbody>
</table>
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

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

(No column name)

100

can use arithmetic expressions
**Grouping and Aggregation**

<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>

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

<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>
From → Where → Group By → Select

<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>

Select contains
- grouped attributes
- and aggregates

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

```
SELECT product,
       sum(quantity) as SumQuantity,
       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 SumQuantity,
      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

\[
\begin{align*}
5) \text{SELECT} & \quad S \\
1) \text{FROM} & \quad R_1, \ldots, R_n \\
2) \text{WHERE} & \quad C_1 \\
3) \text{GROUP BY} & \quad a_1, \ldots, a_k \\
4) \text{HAVING} & \quad C_2 \\
\end{align*}
\]

S: may contain attributes \(a_1, \ldots, a_k\) and/or any aggregates but no other attributes

C1: is any condition on the attributes in \(R_1, \ldots, R_n\)

C2: is any condition on aggregates and on attributes \(a_1, \ldots, a_k\)

Evaluation

1. Evaluate From \(\rightarrow\) Where, apply condition \(C_1\)
2. Group by the attributes \(a_1, \ldots, a_k\)
3. Apply condition \(C_2\) 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...

```sql
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 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:
  - FALSE = 0
  - TRUE = 1
  - UNKNOWN = 0.5
  - \( x \text{ AND } y = \min(x,y) \)
  - \( x \text{ OR } y = \max(x,y) \)
  - \( \text{NOT } x = (1 - x) \)
SELECT * FROM Person WHERE (age < 25) and (height > 6 or weight > 190)

<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>

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

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
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Category</td>
<td>ProdName</td>
</tr>
<tr>
<td>Gizmo</td>
<td>Gadget</td>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Camera</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Camera</td>
</tr>
<tr>
<td></td>
<td></td>
<td>OneClick</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’

```sql
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
WHERE 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’
  
  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 
WHERE 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

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

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

\[
Q2(f,l) : \text{:- Actor}(z,f,l), \text{Casts}(z,x), \text{Movie}(x,y,'2007')
\]
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

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?

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

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

<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>

R =
Simple datalog programs

What does this compute?

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

1\textsuperscript{st} iteration

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

What does this compute?

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

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

R=

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

1st iteration

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

2nd iteration

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

What does this compute?

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

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

<table>
<thead>
<tr>
<th>1st iteration</th>
<th>2nd iteration</th>
<th>3rd iteration</th>
</tr>
</thead>
<tbody>
<tr>
<td>1 2</td>
<td>1 2</td>
<td>1 2</td>
</tr>
<tr>
<td>2 1</td>
<td>2 1</td>
<td>2 1</td>
</tr>
<tr>
<td>2 3</td>
<td>2 3</td>
<td>2 3</td>
</tr>
<tr>
<td>1 4</td>
<td>1 4</td>
<td>1 4</td>
</tr>
<tr>
<td>3 4</td>
<td>3 4</td>
<td>3 4</td>
</tr>
<tr>
<td>4 5</td>
<td>4 5</td>
<td>4 5</td>
</tr>
</tbody>
</table>

R =

1 2
2 1
2 3
1 4
3 4
4 5
Datalog with Negation

\[
\begin{align*}
B0(x) & : \text{Actor}(x, 'Kevin', 'Bacon') \\
B1(x) & : \text{Actor}(x,f,l), \text{Casts}(x,z), \text{Casts}(y,z), B0(y) \\
Q5(x) & : \text{Actor}(x,f,l), \text{not } B1(x), \text{not } B0(x)
\end{align*}
\]

Find actors with Bacon number $\geq 2$
Recursion and negation: 😞

EDB: R(a)

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

The fixpoint is unclear!
Unsafe Datalog Rules

What is unsafe about these rules?

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

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

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