Introduction to Data Management
CSE 414

Unit 2: The Relational Data Model
SQL
Relational Algebra
Datalog

(9 lectures*)

*Slides may change: refresh each lecture
Introduction to Data Management
CSE 414

Lecture 2: Data Models
Class Overview

- Unit 1: Intro
- **Unit 2: Relational Data Models and Query Languages**
  - Data models, SQL RA, Datalog
- Unit 3: Non-relational data
- Unit 4: RDMBS internals and query optimization
- Unit 5: Parallel query processing
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
- Unit 8: Advanced topics (time permitting)
Reminders

• Sections tomorrow (bring your laptops)

• HW1 due on Friday

• Webquiz due on Saturday
Review

• What is a database?
  – A collection of files storing related data

• What is a DBMS?
  – An application program that allows us to manage efficiently the collection of data files
Data Models

• Recall our example: want to design a database of books:
  – author, title, publisher, pub date, price, etc
  – How should we describe this data?
• **Data model** = mathematical formalism (or conceptual way) for describing the data
Data Models

- Relational
  - Data represented as relations

- Semi-structured (JSON)
  - Data represented as trees

- Key-value pairs
  - Used by NoSQL systems

- Graph

- Object-oriented
Example: storing FB friends

As a graph

As a relation

We will learn the tradeoffs of different data models later this quarter
3 Elements of Data Models

• Instance
  – The actual data

• Schema
  – Describe what data is being stored

• Query language
  – How to retrieve and manipulate data
Relational Model

- Data is a collection of relations / tables:
  - columns / attributes / fields
  - rows / tuples / records

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
<td>20000</td>
<td>True</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>True</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>True</td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</td>
<td>500</td>
<td>False</td>
</tr>
</tbody>
</table>

- mathematically, relation is a set of tuples
  - each tuple appears 0 or 1 times in the table
  - order of the rows is unspecified
The Relational Data Model

• Each attribute has a type. E.g.
  – Strings: CHAR(20), VARCHAR(50), TEXT
  – Numbers: INT, SMALLINT, FLOAT
  – MONEY, DATETIME, …
  – Few more that are vendor specific

• Types statically and strictly enforced

• #Attributes= “degree” (arity) of a relation
Keys

• Key = one (or multiple) attributes that uniquely identify a record
Keys

• Key = one (or multiple) attributes that uniquely identify a record

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
<td>20000</td>
<td>True</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>True</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>True</td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</td>
<td>500</td>
<td>False</td>
</tr>
</tbody>
</table>
Keys

• Key = one (or multiple) attributes that uniquely identify a record

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
<td>20000</td>
<td>True</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>True</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>True</td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</td>
<td>500</td>
<td>False</td>
</tr>
</tbody>
</table>
Keys

• Key = one (or multiple) attributes that uniquely identify a record

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>country</td>
<td>no_employees</td>
<td>for_profit</td>
<td></td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
<td>20000</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>True</td>
<td></td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</td>
<td>500</td>
<td>False</td>
<td></td>
</tr>
</tbody>
</table>
Keys

- Key = one (or multiple) attributes that uniquely identify a record

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
<td>20000</td>
<td>True</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>True</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>True</td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</td>
<td>500</td>
<td>False</td>
</tr>
</tbody>
</table>
### Multi-attribute Key

Key = fName, lName

*(what does this mean?)*

<table>
<thead>
<tr>
<th>fName</th>
<th>lName</th>
<th>Income</th>
<th>Department</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Smith</td>
<td>20000</td>
<td>Testing</td>
</tr>
<tr>
<td>Alice</td>
<td>Thompson</td>
<td>50000</td>
<td>Testing</td>
</tr>
<tr>
<td>Bob</td>
<td>Thompson</td>
<td>30000</td>
<td>SW</td>
</tr>
<tr>
<td>Carol</td>
<td>Smith</td>
<td>50000</td>
<td>Testing</td>
</tr>
</tbody>
</table>
Multiple Keys

We can choose one key and designate it as \textit{primary key}.
E.g.: primary key = SSN
Foreign Key

Company(cname, country, no_employees, for_profit)
Country(name, population)

| Company | | | | | |
|---|---|---|---|---|
| cname | country | no_employees | for_profit |
| Canon | Japan | 50000 | Y |
| Hitachi | Japan | 30000 | Y |

| Country | | |
|---|---|
| name | population |
| USA | 320M |
| Japan | 127M |
Keys: Summary

• Key = columns that uniquely identify tuple
  – Usually we underline
  – A relation can have many keys, but only one can be chosen as primary key

• Foreign key:
  – Attribute(s) whose value is a key of a record in some other relation
  – Foreign keys are sometimes called semantic pointer
Query Language

• SQL
  – Structured Query Language
  – Developed by IBM in the 70s
  – Most widely used language to query relational data

• Other relational query languages
  – Datalog, relational algebra
Our First DBMS

- SQL Lite
- Will switch to SQL Server later in the quarter
Demo 1
Discussion

• Tables are NOT ordered
  – they are sets or multisets (bags)

• Tables are FLAT
  – No nested attributes

• Tables DO NOT prescribe how they are implemented / stored on disk
  – This is called physical data independence
Table Implementation

• How would you implement this?

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
<td>20000</td>
<td>True</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>True</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>True</td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</td>
<td>500</td>
<td>False</td>
</tr>
</tbody>
</table>
Table Implementation

• How would you implement this?

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
<td>20000</td>
<td>True</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>True</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>True</td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</td>
<td>500</td>
<td>False</td>
</tr>
</tbody>
</table>

Row major: as an array of objects

| GizmoWorks USA 20000 True | Canon Japan 50000 True | Hitachi Japan 30000 True | HappyCam Canada 500 False |
Table Implementation

• How would you implement this?

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
<td>20000</td>
<td>True</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>True</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>True</td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</td>
<td>500</td>
<td>False</td>
</tr>
</tbody>
</table>

Column major: as one array per attribute

<table>
<thead>
<tr>
<th>GizmoWorks</th>
<th>Canon</th>
<th>Hitachi</th>
<th>HappyCam</th>
</tr>
</thead>
<tbody>
<tr>
<td>USA</td>
<td>Japan</td>
<td>Japan</td>
<td>Canada</td>
</tr>
<tr>
<td>20000</td>
<td>50000</td>
<td>30000</td>
<td>500</td>
</tr>
<tr>
<td>True</td>
<td>True</td>
<td>True</td>
<td>False</td>
</tr>
</tbody>
</table>
Table Implementation

• How would you implement this?

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
<td>20000</td>
<td>True</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>True</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>True</td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</td>
<td>500</td>
<td>False</td>
</tr>
</tbody>
</table>

Physical data independence
The logical definition of the data remains unchanged, even when we make changes to the actual implementation
First Normal Form

- All relations must be flat: we say that the relation is in first normal form.

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>Y</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>Y</td>
</tr>
</tbody>
</table>
First Normal Form

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>Y</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>Y</td>
</tr>
</tbody>
</table>

- All relations must be flat: we say that the relation is in *first normal form*
- E.g. we want to add products manufactured by each company:
First Normal Form

- All relations must be flat: we say that the relation is in *first normal form*
- E.g. we want to add products manufactured by each company:

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
<th>products</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>Y</td>
<td></td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>Y</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>AC</td>
<td>300</td>
<td>Appliance</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>149.99</td>
<td>Photography</td>
</tr>
<tr>
<td>Gadget</td>
<td>200</td>
<td>Toy</td>
</tr>
</tbody>
</table>
First Normal Form

- All relations must be flat: we say that the relation is in *first normal form*
- E.g. we want to add products manufactured by each company:

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>Y</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>Y</td>
</tr>
</tbody>
</table>

```
cname | country | no_employees | for_profit |
-------|---------|--------------|------------|
Canon  | Japan   | 50000        | Y          |
Hitachi| Japan   | 30000        | Y          |
```

| products | |
|----------|------|-----------------|
|          |      | pname | price | category |
|          |      | SingleTouch | 149.99 | Photography |
|          |      | Gadget | 200 | Toy |
|          |      | AC | 300 | Appliance |

Non-1NF!
First Normal Form

Company

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
<th>no_employees</th>
<th>for_profit</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>Y</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>Y</td>
</tr>
</tbody>
</table>

Products

<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>AC</td>
<td>300</td>
<td>Appliance</td>
<td>Hitachi</td>
</tr>
<tr>
<td>Gadget</td>
<td>200</td>
<td>Toy</td>
<td>Canon</td>
</tr>
</tbody>
</table>
Data Models: Summary

• Schema + Instance + Query language
• Relational model:
  – Database = collection of tables
  – Each table is flat: “first normal form”
  – Key: may consists of multiple attributes
  – Foreign key: “semantic pointer”
  – Physical data independence
Introduction to Data Management
CSE 414

Lecture 3: SQL Basics
Review

• Relational data model
  – Schema+instance+query language

• Query language: SQL
  – Create tables
  – Retrieve records from tables
  – Declare keys and foreign keys
Review

• Tables are NOT ordered
  – they are sets or multisets (bags)
  – arity: # of attributes in a relation
  – cardinality: # of records in a relation

• Tables are FLAT
  – No nested attributes

• Tables DO NOT prescribe how they are implemented / stored on disk
  – This is called physical data independence
SQL

• Structured Query Language
• Most widely used language to query relational data
• One of the many languages for querying relational data

• A declarative programming language
Selections in SQL

```
SELECT *  
FROM Product 
WHERE price > 100.0 
```
Demo 2
Retrieve all Japanese products that cost < $150
Joins in SQL

Retrieve all Japanese products that cost < $150

```
SELECT  pname, price
FROM    Product, Company
WHERE   ...
```
Retrieve all Japanese products that cost < $150

```
SELECT  pname, price
FROM     Product, Company
WHERE    manufacturer=cname AND
         country='Japan' AND price < 150
```
Product (pname, price, category, manufacturer)
Company (cname, country)

Joins in SQL

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>category</th>
<th>manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiTouch</td>
<td>199.99</td>
<td>gadget</td>
<td>Canon</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>49.99</td>
<td>photography</td>
<td>Canon</td>
</tr>
<tr>
<td>Gizom</td>
<td>50</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SuperGizmo</td>
<td>250.00</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>

Retrieve all USA companies that manufacture “gadget” products
Joins in SQL

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>category</th>
<th>manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>MultiTouch</td>
<td>199.99</td>
<td>gadget</td>
<td>Canon</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>49.99</td>
<td>photography</td>
<td>Canon</td>
</tr>
<tr>
<td>Gizom</td>
<td>50</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SuperGizmo</td>
<td>250.00</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>

Retrieve all USA companies that manufacture “gadget” products

```
SELECT DISTINCT cname
FROM Product, Company
WHERE country = 'USA' AND category = 'gadget'
    AND manufacturer = cname
```

Why DISTINCT?
Joins in SQL

• The standard join in SQL is sometimes called an **inner join**
  – Each row in the result **must come from both tables in the join**

• Sometimes we want to include rows from only one of the two table: **outer join**
Inner Join

Employee(id, name)
Sales(employeeID, productID)

Retrieve employees and their sales
Inner Join

Employee(id, name)
Sales(employeeID, productID)

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>2</td>
<td>Jack</td>
</tr>
<tr>
<td>3</td>
<td>Jill</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>employeeID</th>
<th>productID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>344</td>
</tr>
<tr>
<td>1</td>
<td>355</td>
</tr>
<tr>
<td>2</td>
<td>544</td>
</tr>
</tbody>
</table>

Retrieve employees and their sales

```
SELECT *
FROM Employee E, Sales S
WHERE E.id = S.employeeID
```
Employee(id, name)
Sales(employeeID, productID)

Inner Join

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>2</td>
<td>Jack</td>
</tr>
<tr>
<td>3</td>
<td>Jill</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>employeeID</th>
<th>productID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>344</td>
</tr>
<tr>
<td>1</td>
<td>355</td>
</tr>
<tr>
<td>2</td>
<td>544</td>
</tr>
</tbody>
</table>

Retrieve employees and their sales

```
SELECT * 
FROM Employee E, Sales S 
WHERE E.id = S.employeeID
```
Inner Join

Employee(id, name)
Sales(employeeID, productID)

Retrieve employees and their sales

SELECT * 
FROM Employee E, Sales S 
WHERE E.id = S.employeeID

Jill is missing
Select all employees and their sales:

```sql
SELECT *
FROM Employee E
INNER JOIN Sales S
ON E.id = S.employeeID
```

---

**Employee**

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>2</td>
<td>Jack</td>
</tr>
<tr>
<td>3</td>
<td>Jill</td>
</tr>
</tbody>
</table>

**Sales**

<table>
<thead>
<tr>
<th>employeeID</th>
<th>productID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>344</td>
</tr>
<tr>
<td>1</td>
<td>355</td>
</tr>
<tr>
<td>2</td>
<td>544</td>
</tr>
</tbody>
</table>

Jill is missing.
### Employee Table

<table>
<thead>
<tr>
<th>id</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Joe</td>
</tr>
<tr>
<td>2</td>
<td>Jack</td>
</tr>
<tr>
<td>3</td>
<td>Jill</td>
</tr>
</tbody>
</table>

### Sales Table

<table>
<thead>
<tr>
<th>employeeID</th>
<th>productID</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>344</td>
</tr>
<tr>
<td>1</td>
<td>355</td>
</tr>
<tr>
<td>2</td>
<td>544</td>
</tr>
</tbody>
</table>

### SQL Query

```sql
SELECT *
FROM Employee E
LEFT OUTER JOIN Sales S
ON E.id = S.employeeID
```

- **Joe**: Employee ID 1, Sold products 344 and 355
- **Jack**: Employee ID 2, Sold product 544
- **Jill**: Employee ID 3, Present but no sales

Jill is present.
Introduction to Data Management
CSE 414

Lecture 4: Joins and Aggregates
Review: Our SQL Toolchest

• Selection
• Projection
• Ordering and distinct

• Inner Join
• Outer Join
(Inner) joins

Product(pname, price, category, manufacturer)
Company(cname, country)

manufacturer = foreign key to Company.cname

Return all companies in the ‘USA’ that manufacture some product in the ‘gadget’ category.
(Inner) joins

SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget'
AND manufacturer = cname
INNER joins

```
SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget'
AND manufacturer = cname
```
(Inner) joins

SELECT DISTINCT cname
FROM Product, Company
WHERE country = 'USA' AND category = 'gadget'
AND AND manufacturer = cname

Product

<table>
<thead>
<tr>
<th>pname</th>
<th>category</th>
<th>manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

Company

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>
(Inner) joins

```
SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget'
    AND manufacturer = cname
```

**Product**

<table>
<thead>
<tr>
<th>pname</th>
<th>category</th>
<th>manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

**Company**

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>
### SQL Query

```
SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget'
    AND manufacturer = cname
```

### Product Table

<table>
<thead>
<tr>
<th>pname</th>
<th>category</th>
<th>manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

### Company Table

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>

### Combined Table

<table>
<thead>
<tr>
<th>pname</th>
<th>category</th>
<th>manufacturer</th>
<th>cname</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>GizmoWorks</td>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
</tbody>
</table>
SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget'
AND manufacturer = cname
INNER joins

\[
\text{SELECT DISTINCT} \quad \text{cname} \\
\text{FROM} \quad \text{Product, Company} \\
\text{WHERE} \quad \text{country} = 'USA' \text{ AND category} = 'gadget' \text{ AND manufacturer} = \text{cname}
\]

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>pname</td>
<td>cname</td>
</tr>
<tr>
<td>category</td>
<td>country</td>
</tr>
<tr>
<td>price</td>
<td></td>
</tr>
<tr>
<td>category</td>
<td></td>
</tr>
<tr>
<td>manufacturer</td>
<td>country</td>
</tr>
<tr>
<td>Gizmo</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>
(Inner) joins

```
SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget'
AND AND manufacturer = cname
```
(Inner) joins

```
SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget' 
AND manufacturer = cname
```

### Product

<table>
<thead>
<tr>
<th>pname</th>
<th>category</th>
<th>manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

### Company

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>
(Inner) joins

SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget'
    AND manufacturer = cname

Product

<table>
<thead>
<tr>
<th>pname</th>
<th>category</th>
<th>manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

Company

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>
(Inner) joins

SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget'
AND manufacturer = cname

Product

<table>
<thead>
<tr>
<th>pname</th>
<th>category</th>
<th>manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

Company

<table>
<thead>
<tr>
<th>cname</th>
<th>country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>

And son on...
(Inner) joins

```
SELECT DISTINCT cname
FROM   Product, Company
WHERE  country='USA' AND category = 'gadget'
       AND manufacturer = cname
```

```
SELECT DISTINCT cname
FROM   Product
JOIN   Company
ON     country = 'USA'
       AND category = 'gadget'
       AND manufacturer = cname
```
(Inner) joins

```sql
SELECT DISTINCT cname
FROM Product, Company
WHERE country='USA' AND category = 'gadget'
    AND manufacturer = cname
```

```sql
SELECT DISTINCT cname
FROM Product
JOIN Company ON country = 'USA'
    AND category = 'gadget'
    AND manufacturer = cname
```

Same thing, different syntax
(Inner) Joins

```
SELECT  x1.a1, x2.a2, ... xm.am
FROM    R1 as x1, R2 as x2, ... Rm as xm
WHERE   Cond
```

for x1 in R1:
    for x2 in R2:
        ...
        for xm in Rm:
            if Cond(x1, x2...):
                output(x1.a1, x2.a2, ... xm.am)

This is called nested loop semantics since we are interpreting what a join means using a nested loop
Another example

Retrieve all USA companies that manufacture products in both ‘gadget’ and ‘photography’ categories
Another example

Retrieve all USA companies that manufacture products in both ‘gadget’ and ‘photography’ categories

```
SELECT DISTINCT z.cname
FROM Product x, Company z
WHERE z.country = 'USA'
    AND x.manufacturer = z.cname
    AND x.category = 'gadget'
    AND x.category = 'photography';
```
Another example

Retrieve all USA companies that manufacture products in both ‘gadget’ and ‘photography’ categories.

```
SELECT DISTINCT z.cname
FROM Product x, Company z
WHERE z.country = 'USA'
  AND x.manufacturer = z.cname
  AND x.category = 'gadget'
  AND x.category = 'photography';
```

Does this work?

Returns the empty set.
Another example

Retrieve all USA companies that manufacture products in both ‘gadget’ and ‘photography’ categories

What about this?

```
SELECT DISTINCT z.cname
FROM Product x, Company z
WHERE z.country = 'USA'
    AND x.manufacturer = z.cname
    AND (x.category = 'gadget'
        OR x.category = 'photography');
```
Another example

Retrieve all USA companies that manufacture products in both ‘gadget’ and ‘photography’ categories

\[
\text{SELECT DISTINCT z.cname} \\
\text{FROM Product x, Company z} \\
\text{WHERE z.country = 'USA'} \\
\text{AND x.manufacturer = z.cname} \\
\text{AND (x.category = 'gadget'} \\
\text{OR x.category = 'photography');}
\]
Another example

Retrieve all USA companies that manufacture products in both ‘gadget’ and ‘photography’ categories

```
SELECT DISTINCT z.cname
FROM Product x, Product y, Company z
WHERE z.country = 'USA'
    AND x.manufacturer = z.cname
    AND y.manufacturer = z.cname
    AND x.category = 'gadget'
    AND y.category = 'photography';
```

Need to include Product twice!
Self-Joins and Tuple Variables

- Find USA companies that manufacture both products in the ‘gadgets’ and ‘photo’ category.

- Joining Product with Company is insufficient: need to join Product, with Product, and with Company.

- When a relation occurs twice in the FROM clause we call it a self-join; in that case we must use tuple variables (why?)
SELECT DISTINCT z.cname
FROM Product x, Product y, Company z
WHERE z.country = 'USA'
    AND x.category = 'gadget'
    AND y.category = 'photo'
    AND x.manufacturer = z.cname
    AND y.manufacturer = z.cname;

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>pname</td>
<td>category</td>
</tr>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>photo</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>Photo</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>cname</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td></td>
</tr>
<tr>
<td>Hitachi</td>
<td></td>
</tr>
</tbody>
</table>
Self-joins

```
SELECT DISTINCT z.cname
FROM   Product x, Product y, Company z
WHERE  z.country = 'USA'
   AND x.category = 'gadget'
   AND y.category = 'photo'
   AND x.manufacturer = z.cname
   AND y.manufacturer = z.cname;
```
Self-joins

```
SELECT DISTINCT z.cname
FROM   Product x, Product y, Company z
WHERE  z.country = 'USA'
       AND x.category = 'gadget'
       AND y.category = 'photo'
       AND x.manufacturer = z.cname
       AND y.manufacturer = z.cname;
```
SELECT DISTINCT z.cname
FROM Product x, Product y, Company z
WHERE z.country = 'USA'
AND x.category = 'gadget'
AND y.category = 'photo'
AND x.manufacturer = z.cname
AND y.manufacturer = z.cname;
Self-joins

```
SELECT DISTINCT z.cname
FROM  Product x, Product y, Company z
WHERE z.country = 'USA'
  AND x.category = 'gadget'
  AND y.category = 'photo'
  AND x.manufacturer = z.cname
  AND y.manufacturer = z.cname;
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
</tr>
<tr>
<td>pname</td>
<td>category</td>
</tr>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>photo</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>Photo</td>
</tr>
<tr>
<td>z.cname</td>
<td>country</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>
SELECT DISTINCT z.cname
FROM Product x, Product y, Company z
WHERE z.country = 'USA'
    AND x.category = 'gadget'
    AND y.category = 'photo'
    AND x.manufacturer = z.cname
    AND y.manufacturer = z.cname;

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>x.pname</td>
<td>x.category</td>
</tr>
<tr>
<td></td>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td></td>
<td>SingleTouch</td>
<td>photo</td>
</tr>
<tr>
<td></td>
<td>MultiTouch</td>
<td>Photo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Company</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>z</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td>z.cname</td>
<td>z.country</td>
</tr>
<tr>
<td></td>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td></td>
<td>Hitachi</td>
<td>Japan</td>
</tr>
</tbody>
</table>
Self-joins

```
SELECT DISTINCT z.cname
FROM Product x, Product y, Company z
WHERE z.country = 'USA'
    AND x.category = 'gadget'
    AND y.category = 'photo'
    AND x.manufacturer = z.cname
    AND y.manufacturer = z.cname;
```
Self-joins

```
SELECT DISTINCT z.cname
FROM Product x, Product y, Company z
WHERE z.country = 'USA'
    AND x.category = 'gadget'
    AND y.category = 'photo'
    AND x.manufacturer = z.cname
    AND y.manufacturer = z.cname;
```
Self-joins

```sql
SELECT DISTINCT z.cname
FROM Product x, Product y, Company z
WHERE z.country = 'USA'
AND x.category = 'gadget'
AND y.category = 'photo'
AND x.manufacturer = z.cname
AND y.manufacturer = z.cname;
```
Self-joins

SELECT DISTINCT z.cname
FROM Product x, Product y, Company z
WHERE z.country = 'USA'
AND x.category = 'gadget'
AND y.category = 'photo'
AND x.manufacturer = z.cname
AND y.manufacturer = z.cname;
Outer joins

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

-- prodName is foreign key

Retrieve all product names and the stores where they were purchased. Include products that never sold
Outer joins

Retrieve all product names and the stores where they were purchased. Include products that never sold.

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

-- prodName is foreign key

SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName
Outer joins

Retrieve all product names and the stores where they were purchased.
Include products that never sold!

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

-- prodName is foreign key

SELECT Product.name, Purchase.store
FROM Product, Purchase
WHERE Product.name = Purchase.prodName

Does not include products that never sold! (why?)
Outer joins

Retrieve all product names and the stores where they were purchased.
Include products that never sold.

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

-- prodName is foreign key

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

Retrieve all product names and the stores where they were purchased. Include products that never sold

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

Now they show up!
### 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>

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

<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>

<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>

<table>
<thead>
<tr>
<th>Name</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
</tbody>
</table>
```sql
SELECT Product.name, Purchase.store
FROM Product JOIN Purchase ON Product.name = Purchase.prodName
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>ProdName</td>
</tr>
<tr>
<td>Gizmo</td>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
<td>Camera</td>
</tr>
<tr>
<td>OneClick</td>
<td>Camera</td>
</tr>
</tbody>
</table>

**Output**

<table>
<thead>
<tr>
<th>Name</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
</tbody>
</table>
```
SELECT Product.name, Purchase.store
FROM Product 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>

**Output**

<table>
<thead>
<tr>
<th>Name</th>
<th>Store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
</tbody>
</table>
SELECT Product.name, Purchase.store
FROM Product JOIN Purchase ON
    Product.name = Purchase.prodName

Output
### SQL Query

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

### Tables

**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>

### Output

<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>
</tbody>
</table>
```sql
SELECT Product.name, Purchase.store 
FROM Product 
JOIN Purchase ON Product.name = Purchase.prodName
```

<table>
<thead>
<tr>
<th>Product</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>

<table>
<thead>
<tr>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProdName</td>
</tr>
<tr>
<td>----------</td>
</tr>
<tr>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
</tr>
<tr>
<td>Camera</td>
</tr>
</tbody>
</table>

Output

<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>
</tbody>
</table>
```
SELECT Product.name, Purchase.store
FROM Product JOIN Purchase ON
Product.name = Purchase.prodName
```

<table>
<thead>
<tr>
<th>Product</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>

<table>
<thead>
<tr>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>ProdName</td>
</tr>
<tr>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
</tr>
<tr>
<td>Camera</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Output</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
</tr>
<tr>
<td>Gizmo</td>
</tr>
<tr>
<td>Camera</td>
</tr>
<tr>
<td>Camera</td>
</tr>
</tbody>
</table>
### SELECT Query

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

### Product Table

<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

<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>

### Output Table

<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>
</tbody>
</table>
### SQL Query

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

### Product Table

<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

<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>

### Output

<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>
### 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>
<tr>
<td>Phone</td>
<td>Foo</td>
</tr>
</tbody>
</table>

### Output

<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></td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
</tr>
<tr>
<td>NULL</td>
<td>Foo</td>
</tr>
</tbody>
</table>
Outer Joins

\[
\text{tableA (LEFT/RIGHT/FULL) OUTER JOIN tableB ON p}
\]

- Left outer join:
  - Include tuples from tableA even if no match
- Right outer join:
  - Include tuples from tableB even if no match
- Full outer join:
  - Include tuples from both even if no match

- In all cases:
  - Patch tuples without matches using NULL
Loading Data into SQLite

>sqlite3 lecture04

sqlite> create table Purchase
    (pid int primary key,
     product text,
     price float,
     quantity int,
     month varchar(15));

sqlite> -- download data.txt
sqlite> .import lec04-data.txt Purchase

Specify a filename where the database will be stored

Other DBMSs have other ways of importing data
Comment about SQLite

• Cannot load NULL values such that they are actually loaded as null values

• So we need to use two steps:
  – Load null values using some type of special value
  – Update the special values to actual null values

update Purchase
  set price = null
where price = ‘null’
Simple Aggregations

Five basic aggregate operations in SQL

- `select count(*) from Purchase`
- `select sum(quantity) from Purchase`
- `select avg(price) from Purchase`
- `select max(quantity) from Purchase`
- `select min(quantity) from Purchase`

Except count, all aggregations apply to a single attribute.
Aggregates and NULL Values

Null values are not used in aggregates

```
insert into Purchase
values(12, 'gadget', NULL, NULL, NULL, 'april')
```

Let’s try the following

```
select count(*) from Purchase
select count(quantity) from Purchase

select sum(quantity) from Purchase

select count(*)
from Purchase
where quantity is not null;
```
COUNT applies to duplicates, unless otherwise stated:

```sql
SELECT count(product) 
FROM Purchase 
WHERE price > 4.99
```

same as `count(*)` if no nulls

We probably want:

```sql
SELECT count(DISTINCT product) 
FROM Purchase 
WHERE price > 4.99
```
More Examples

SELECT Sum(price * quantity) FROM Purchase

SELECT Sum(price * quantity) FROM Purchase WHERE product = 'bagel'

What do they mean?
Introduction to Data Management
CSE 414

Lecture 5: Grouping and Query Evaluation
Announcements

• Welcome new TA: Esteban Posada!
• New section AG, starting next week
• Webquiz due tonight
• Homework 2 due on Monday
• No lecture on Monday!
  Makeup lecture Thursday, 4/18, 5:30pm
Grouping and Aggregation

Purchase(product, price, quantity)

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

```
SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```
SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
### SELECT product, Sum(quantity) AS TotalSales
### FROM Purchase
### WHERE price > 1
### GROUP BY product
Other Examples

Compare these two queries:

```
SELECT product, count(*)
FROM Purchase
GROUP BY product
```

```
SELECT month, count(*)
FROM Purchase
GROUP BY month
```
Other Examples

Compare these two queries:

```
SELECT product, count(*)
FROM Purchase
GROUP BY product

SELECT month, count(*)
FROM Purchase
GROUP BY month
```

One answer for each product. One answer for each month.
Other Examples

SELECT product, 
    sum(quantity) AS SumQuantity, 
    max(price) AS MaxPrice 
FROM Purchase 
GROUP BY product
Need to be Careful…

```
SELECT product, max(quantity)
FROM Purchase
GROUP BY product
```
Need to be Careful…

```sql
SELECT product, max(quantity)
FROM Purchase
GROUP BY product
```

<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>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</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>Max(quantity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>50</td>
</tr>
</tbody>
</table>
Need to be Careful…

```
SELECT product, 
    max(quantity) 
FROM Purchase 
GROUP BY product
```

```
SELECT product, quantity 
FROM Purchase 
GROUP BY product 
-- what does this mean?
```

<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>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</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>Max(quantity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>50</td>
</tr>
</tbody>
</table>
Need to be Careful…

```
SELECT product, max(quantity)
FROM Purchase
GROUP BY product
```

```
SELECT product, quantity
FROM Purchase
GROUP BY product
-- what does this mean?
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Max(quantity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>50</td>
</tr>
</tbody>
</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>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</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>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>??</td>
</tr>
</tbody>
</table>
Need to be Careful…

```
SELECT product, max(quantity)
FROM Purchase
GROUP BY product
```

```
SELECT product, quantity
FROM Purchase
GROUP BY product
-- what does this mean?
```

<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>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</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>Max(quantity)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>50</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>??</td>
</tr>
</tbody>
</table>

CSE 414 - 2019sp
Select product, max(quantity) from Purchase group by product

Select product, quantity from Purchase group by product

-- what does this mean?

<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>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</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>

Everything in SELECT must be either a GROUP-BY attribute, or an aggregate.
Number of Groups

Purchase(product, price, quantity)

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

SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
Number of Groups

Purchase(product, price, quantity)

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

```
SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```

Clearly, queries return different answers. What about # groups?

```
SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
GROUP BY product
```
Number of Groups

Purchase(product, price, quantity)

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

\[
\text{SELECT} \quad \text{product, } \text{Sum(quantity)} \text{ AS TotalSales} \\
\text{FROM} \quad \text{Purchase} \\
\text{WHERE} \quad \text{price} > 1 \\
\text{GROUP BY} \quad \text{product}
\]

Clearly, queries return different answers. What about # groups?

\[
\text{SELECT} \quad \text{product, } \text{Sum(quantity)} \text{ AS TotalSales} \\
\text{FROM} \quad \text{Purchase} \\
\text{GROUP BY} \quad \text{product}
\]

Empty groups are removed, hence first query may return fewer groups.
1. Compute the **FROM** and **WHERE** clauses.

2. Group by the attributes in the **GROUPBY**

3. Compute the **SELECT** clause:
   grouped attributes and aggregates.
SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
3.4. Grouping, Select

```
SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```

<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>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</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>
Ordering Results

```
SELECT product, sum(price*quantity) as rev
FROM Purchase
GROUP BY product
ORDER BY rev DESC
```

Note: some SQL engines want you to say
```
ORDER BY sum(price*quantity) DESC
```
HAVING Clause

Same query as before, except that we consider only products that had at least 30 sales.

```
SELECT    product, sum(price*quantity)
FROM      Purchase
WHERE     price > 1
GROUP BY  product
HAVING    sum(quantity) > 30
```

HAVING clause contains conditions on aggregates.
### General form of Grouping and Aggregation

<table>
<thead>
<tr>
<th>SELECT</th>
<th>S</th>
</tr>
</thead>
<tbody>
<tr>
<td>FROM</td>
<td>R₁,...,Rₙ</td>
</tr>
<tr>
<td>WHERE</td>
<td>C₁</td>
</tr>
<tr>
<td>GROUP BY</td>
<td>a₁,...,aₖ</td>
</tr>
<tr>
<td>HAVING</td>
<td>C₂</td>
</tr>
</tbody>
</table>

- S = may contain attributes $a_1, ..., a_k$ and/or any aggregates but NO OTHER ATTRIBUTES
- C₁ = is any condition on the attributes in $R₁,...,Rₙ$
- C₂ = is any condition on aggregate expressions and on attributes $a_1,...,a_k$

*Why?*
Semantics of SQL With Group-By

Evaluation steps:
1. Evaluate FROM-WHERE using Nested Loop Semantics
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
Exercise

Compute the total income per month
Show only months with less than 10 items sold
Order by quantity sold and display as “TotalSold”
Exercise

Compute the total income per month
Show only months with less than 10 items sold
Order by quantity sold and display as “TotalSold”

```
FROM Purchase
```
Exercise

Compute the total income per month
Show only months with less than 10 items sold
Order by quantity sold and display as “TotalSold”

```
FROM Purchase
GROUP BY month
```
Compute the total income per month
Show only months with less than 10 items sold
Order by quantity sold and display as “TotalSold”

```
FROM Purchase
GROUP BY month
HAVING sum(quantity) < 10
```
Exercise

Compute the total income per month
Show only months with less than 10 items sold
Order by quantity sold and display as “TotalSold”

```
SELECT      month, sum(price*quantity),
            sum(quantity) as TotalSold
FROM        Purchase
GROUP BY    month
HAVING      sum(quantity) < 10
```
Exercise

Compute the total income per month
Show only months with less than 10 items sold
Order by quantity sold and display as “TotalSold”

```
SELECT month, sum(price*quantity),
       sum(quantity) as TotalSold
FROM Purchase
GROUP BY month
HAVING sum(quantity) < 10
ORDER BY sum(quantity)
```
WHERE vs HAVING

• WHERE condition is applied to individual rows
  – The rows may or may not contribute to the aggregate
  – No aggregates allowed here
  – Occasionally, some groups become empty and are removed

• HAVING condition is applied to the entire group
  – Entire group is returned, or removed
  – May use aggregate functions on the group
What do they compute?

```
SELECT  month, sum(quantity), max(price)
FROM    Purchase
GROUP BY month
```

```
SELECT  month, sum(quantity)
FROM    Purchase
GROUP BY month
```

```
SELECT  month
FROM    Purchase
GROUP BY month
```
Mystery Query

What do they compute?

```
SELECT month, sum(quantity), max(price)
FROM Purchase
GROUP BY month
```

Lesson: DISTINCT is a special case of GROUP BY

```
SELECT month, sum(quantity)
FROM Purchase
GROUP BY month
```

```
SELECT month
FROM Purchase
GROUP BY month
```
Aggregate + Join

For each manufacturer, compute how many products with price > $100 they sold
Aggregate + Join

For each manufacturer, compute how many products with price > $100 they sold

Problem: manufacturer is in Product, price is in Purchase...
Aggregate + Join

For each manufacturer, compute how many products with price > $100 they sold

Problem: manufacturer is in Product, price is in Purchase...

-- step 1: think about their join
SELECT ...
FROM Product x, Purchase y
WHERE x.product_id = y.product_id and y.price > 100

<table>
<thead>
<tr>
<th>manufacturer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hitachi</td>
<td>150</td>
</tr>
<tr>
<td>Canon</td>
<td>300</td>
</tr>
<tr>
<td>Hitachi</td>
<td>180</td>
</tr>
</tbody>
</table>
Aggregate + Join

For each manufacturer, compute how many products with price > $100 they sold

Problem: manufacturer is in Product, price is in Purchase...

-- step 1: think about their join
SELECT ...
FROM Product x, Purchase y
WHERE x.product_id = y.product_id
    and y.price > 100

<table>
<thead>
<tr>
<th>manufacturer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hitachi</td>
<td>150</td>
</tr>
<tr>
<td>Canon</td>
<td>300</td>
</tr>
<tr>
<td>Hitachi</td>
<td>180</td>
</tr>
</tbody>
</table>

-- step 2: do the group-by on the join
SELECT x.manufacturer, count(*)
FROM Product x, Purchase y
WHERE x.product_id = y.product_id
    and y.price > 100
GROUP BY x.manufacturer

<table>
<thead>
<tr>
<th>manufacturer</th>
<th>count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hitachi</td>
<td>2</td>
</tr>
<tr>
<td>Canon</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Aggregate + Join

Variant:
For each manufacturer, compute how many products with price > $100 they sold in each month

```
SELECT x.manufacturer, y.month, count(*)
FROM Product x, Purchase y
WHERE x.product_id = y.product_id
  and y.price > 100
GROUP BY x.manufacturer, y.month
```
Including Empty Groups

- In the result of a group by query, there is one row per group in the result.

```sql
SELECT x.manufacturer, count(*)
FROM Product x, Purchase y
WHERE x.product_id = y.product_id
GROUP BY x.manufacturer
```

Count(*) is never 0
Including Empty Groups

```
SELECT x.manufacturer, count(*)
FROM Product x, Purchase y
WHERE x.product_id = y.product_id
GROUP BY x.manufacturer
```

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
<th>Final results</th>
</tr>
</thead>
<tbody>
<tr>
<td>pname</td>
<td>manufacturer</td>
<td>product</td>
</tr>
<tr>
<td>Gizmo</td>
<td>GizmoWorks</td>
<td>Camera</td>
</tr>
<tr>
<td>Camera</td>
<td>Canon</td>
<td>Camera</td>
</tr>
<tr>
<td>OneClick</td>
<td>Hitachi</td>
<td>OneClick</td>
</tr>
</tbody>
</table>

No GizmoWorks!
Including Empty Groups

```
SELECT x.manufacturer, count(y.pid)
FROM Product x LEFT OUTER JOIN Purchase y
ON x.product_id = y.product_id
GROUP BY x.manufacturer
```

Count(pid) is 0 when all pid’s in the group are NULL.
**Including Empty Groups**

```sql
SELECT x.manufacturer, count(y.pid)
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.product_id = y.product_id
GROUP BY x.manufacturer
```

<table>
<thead>
<tr>
<th>prod_id</th>
<th>manufacturer</th>
<th>price</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera</td>
<td>Canon</td>
<td>150</td>
</tr>
<tr>
<td>Camera</td>
<td>Canon</td>
<td>300</td>
</tr>
<tr>
<td>OneClick</td>
<td>Hitachi</td>
<td>180</td>
</tr>
</tbody>
</table>

**Product**
- **Gizmo**
  - GizmoWorks
- **Camera**
  - Canon
- **OneClick**
  - Hitachi

**Purchase**
- **Camera**
  - 150
- **Camera**
  - 300
- **OneClick**
  - 180

**Final results**

<table>
<thead>
<tr>
<th>manufacturer</th>
<th>Count(y.pid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon</td>
<td>2</td>
</tr>
<tr>
<td>Hitachi</td>
<td>1</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>0</td>
</tr>
</tbody>
</table>

**Why 0 for GizmoWorks?**

- **Left Outer Join**
  - **Product**
    - Gizmo
      - GizmoWorks
    - Camera
      - Canon
    - OneClick
      - Hitachi
  - **Purchase**
    - Camera
      - 150
    - Camera
      - 300
    - OneClick
      - 180

**GizmoWorks is paired with NULLs**
Including Empty Groups

```sql
SELECT x.manufacturer, count(*)
FROM Product x
LEFT OUTER JOIN Purchase y
ON x.product_id = y.product_id
GROUP BY x.manufacturer
```

### Product

<table>
<thead>
<tr>
<th>prod_id</th>
<th>manufacturer</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>GizmoWorks</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Canon</td>
<td></td>
</tr>
<tr>
<td>OneClick</td>
<td>Hitachi</td>
<td></td>
</tr>
</tbody>
</table>

### Purchase

<table>
<thead>
<tr>
<th>product</th>
<th>price</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>OneClick</td>
<td>180</td>
<td></td>
</tr>
</tbody>
</table>

---

**Final results**

<table>
<thead>
<tr>
<th>manufacturer</th>
<th>Count(*)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon</td>
<td>2</td>
</tr>
<tr>
<td>Hitachi</td>
<td>1</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>1</td>
</tr>
</tbody>
</table>

**Left Outer Join(Product, Purchase)**

<table>
<thead>
<tr>
<th>prod_id</th>
<th>manufacturer</th>
<th>...</th>
<th>product</th>
<th>price</th>
<th>...</th>
</tr>
</thead>
<tbody>
<tr>
<td>Camera</td>
<td>Canon</td>
<td></td>
<td>Camera</td>
<td>150</td>
<td></td>
</tr>
<tr>
<td>Camera</td>
<td>Canon</td>
<td></td>
<td>Camera</td>
<td>300</td>
<td></td>
</tr>
<tr>
<td>OneClick</td>
<td>Hitachi</td>
<td></td>
<td>OneClick</td>
<td>180</td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>GizmoWorks</td>
<td></td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

**Probably not what we want!**
Introduction to Data Management
CSE 414

Lecture 6: Nested Queries in SQL
Announcements

• No lecture on Monday, 4/15
• Makeup lecture on Thursday, 4/18, 5:30-6:20, in G20
• Webquiz tomorrow
• Homework 2 due on Monday
What have we learned so far

• Data models
• Relational data model
  – Instance: relations
  – Schema: table with attribute names
  – Language: SQL
What have we learned so far

SQL features
• Projections
• Selections
• Joins (inner and outer)
• Aggregates
• Group by
• Inserts, updates, and deletes

Make sure you read the textbook!
Lecture Goals

• Today we will learn how to write (even) more powerful SQL queries

• Reading: Ch. 6.3
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 nested queries when possible
  – But sometimes it’s impossible, as we will see
Subqueries…

- Can return a single value to be included in a SELECT clause
- Can return a relation to be included in the FROM clause, aliased using a tuple variable
- Can return a single value to be compared with another value in a WHERE clause
- Can return a relation to be used in the WHERE or HAVING clause under an existential quantifier
1. Subqueries in SELECT

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

For each product return the city where it is manufactured

```
SELECT X.pname, (SELECT Y.city
FROM Company Y
WHERE Y.cid=X.cid) as City
FROM Product X
```

What happens if the subquery returns more than one city?
We get a runtime error
(and SQLite simply ignores the extra values…)
1. Subqueries in SELECT

Whenever possible, don’t use a nested queries:

```
SELECT X.pname, (SELECT Y.city
             FROM Company Y
             WHERE Y.cid=X.cid) as City
FROM   Product X
```

We have “unnested” the query
1. Subqueries in SELECT

Compute the number of products made by each company

Product (pname, price, cid)
Company (cid, cname, city)
1. Subqueries in SELECT

Compute the number of products made by each company

```
SELECT C.cid, C.cname, (SELECT count(*)
    FROM Product P
    WHERE P.cid=C.cid)
FROM Company C
```
1. Subqueries in SELECT

Compute the number of products made by each company

```
SELECT C.cid, C.cname, (SELECT count(*)
FROM Product P
WHERE P.cid=C.cid)
FROM Company C
```

Better: we can unnest using a GROUP BY

```
SELECT C.cid, C.cname, count(*)
FROM Company C, Product P
WHERE C.cid=P.cid
GROUP BY C.cid, C.cname
```
1. Subqueries in SELECT

But are these really equivalent?

```sql
SELECT C.cid, C.cname, (SELECT count(*)
FROM Product P
WHERE P.cid=C.cid)
FROM Company C
```

```sql
SELECT C.cid, C.cname, count(*)
FROM Company C, Product P
WHERE C.cid=P.cid
GROUP BY C.cid, C.cname
```
1. Subqueries in SELECT

But are these really equivalent?

```
SELECT C.cid, C.cname, (SELECT count(*)
    FROM Product P
    WHERE P.cid=C.cid)
FROM Company C
```

```
SELECT C.cid, C.cname, count(*)
FROM Company C, Product P
WHERE C.cid=P.cid
GROUP BY C.cid, C.cname
```

```
SELECT C.cid, C.cname, count(pname)
FROM Company C LEFT OUTER JOIN Product P
ON C.cid=P.cid
GROUP BY C.cid, C.cname
```

Recall: count of an empty table is 0

No! Different results if a company has no products
2. Subqueries in FROM

Find all products whose prices is > 20 and < 500
2. Subqueries in FROM

Find all products whose prices is > 20 and < 500

```sql
SELECT X.pname
FROM (SELECT *
      FROM Product AS Y
      WHERE price > 20) as X
WHERE X.price < 500
```
2. Subqueries in FROM

Find all products whose prices is > 20 and < 500

\[
\text{SELECT X.pname}
\text{FROM (SELECT *}
\text{    FROM Product AS Y}
\text{    WHERE price > 20) as X}
\text{WHERE X.price < 500}
\]

Try unnest this query!
2. Subqueries in FROM

Find all products whose prices is > 20 and < 500

```
SELECT X.pname
FROM (SELECT *
      FROM Product AS Y
      WHERE price > 20) as X
WHERE X.price < 500
```

Side note: This is not a correlated subquery. (why?)

Try unnest this query!
2. Subqueries in FROM

Sometimes we need to compute an intermediate table only to use it later in a SELECT-FROM-WHERE

• Option 1: use a subquery in the FROM clause
• Option 2: use the WITH clause
2. Subqueries in FROM

```
SELECT X.pname
FROM (SELECT *
    FROM Product AS Y
    WHERE price > 20) as X
WHERE X.price < 500
```

WITH myTable AS (SELECT * FROM Product AS Y WHERE price > 20)
SELECT X.pname
FROM myTable as X
WHERE X.price < 500

A subquery whose result we called myTable
3. Subqueries in WHERE

Find all companies that make **some** products with price < 200
Product (pname, price, cid)
Company (cid, cname, city)

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Existential quantifiers
3. Subqueries in WHERE

Find all companies that make some products with price < 200

Using EXISTS:

```
SELECT C.cid, C.cname
FROM Company C
WHERE EXISTS (SELECT *
               FROM Product P
               WHERE C.cid = P.cid and P.price < 200)
```
Product (pname, price, cid)
Company (cid, cname, city)

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Using IN

SELECT C.cid, C.cname
FROM Company C
WHERE C.cid IN (SELECT P.cid
    FROM Product P
    WHERE P.price < 200)
3. Subqueries in WHERE

Find all companies that make some products with price < 200

Using **ANY**:

```
SELECT C.cid, C.cname
FROM Company C
WHERE 200 > ANY (SELECT price
                    FROM Product P
                    WHERE P.cid = C.cid)
```
3. Subqueries in WHERE

Find all companies that make some products with price < 200

Using ANY:

```
SELECT C.cid, C.cname
FROM Company C
WHERE 200 > ANY (SELECT price
                  FROM Product P
                  WHERE P.cid = C.cid)
```

Existential quantifiers

Not supported in sqlite
3. Subqueries in WHERE

Find all companies that make some products with price < 200

```
SELECT DISTINCT C.cid, C.cname
FROM Company C, Product P
WHERE C.cid = P.cid and P.price < 200
```
3. Subqueries in WHERE

Find all companies that make **some** products with price < 200

Now let’s unnest it:

```
SELECT DISTINCT C.cid, C.cname
FROM Company C, Product P
WHERE C.cid = P.cid and P.price < 200
```

Existential quantifiers are easy! 😊
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

same as:

Find all companies that make only products with price < 200
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

same as:

Find all companies that make only products with price < 200

Product (pname, price, cid)
Company (cid, cname, city)
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

same as:

Find all companies that make only products with price < 200

Universal quantifiers are hard! 😞
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

1. Find *the other* companies ... ...which ones?
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

1. Find the other companies that make some product ≥ 200
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

1. Find the other companies that make some product ≥ 200

```
SELECT C.cid, C.cname
FROM Company C
WHERE C.cid IN (SELECT P.cid
                FROM Product P
                WHERE P.price >= 200)
```
3. Subqueries in WHERE

Find all companies s.t. **all** their products have price < 200

1. Find *the other* companies that make *some* product ≥ 200

   ```sql
   SELECT C.cid, C.cname
   FROM Company C
   WHERE C.cid IN (SELECT P.cid
                   FROM Product P
                   WHERE P.price >= 200)
   ```

2. Find all companies s.t. **all** their products have price < 200

   ```sql
   SELECT C.cid, C.cname
   FROM Company C
   WHERE C.cid NOT IN (SELECT P.cid
                        FROM Product P
                        WHERE P.price >= 200)
   ```
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

Using **EXISTS**:

```sql
SELECT C.cid, C.cname 
FROM Company C 
WHERE NOT EXISTS (SELECT * 
    FROM Product P 
    WHERE P.cid = C.cid AND P.price >= 200)
```
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

Using \textbf{ALL}:

\begin{verbatim}
SELECT C.cid, C.cname
FROM Company C
WHERE 200 >= ALL (SELECT price
                   FROM Product P
                   WHERE P.cid = C.cid)
\end{verbatim}
3. Subqueries in WHERE

Find all companies s.t. all their products have price < 200

Using **ALL**:  

```
SELECT C.cid, C.cname
FROM Company C
WHERE 200 >= ALL (SELECT price
                    FROM Product P
                    WHERE P.cid = C.cid)
```

**Not supported in sqlite**
Question for Database Theory

Fans and their Friends

• Can we unnest the *universal quantifier* query?

• We need to first discuss the concept of *monotonicity*
Monotone Queries

• Definition A query $Q$ is monotone if:
  – Whenever we add tuples to one or more input tables, the answer to the query will not lose any output tuple
Monotone Queries

• Definition A query Q is monotone if:
  – Whenever we add tuples to one or more input tables, the answer to the query will not lose any output tuple

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

- Definition: A query $Q$ is **monotone** if:
  - Whenever we add tuples to one or more input tables, the answer to the query will not lose any output tuple.
Monotone Queries

- **Definition** A query $Q$ is **monotone** if:
  - Whenever we add tuples to one or more input tables, the answer to the query will not lose any output tuple.
Monotone Queries

• **Theorem:** If Q is a SELECT-FROM-WHERE query that does not have subqueries, and no aggregates, then it is monotone.
Monotone Queries

- **Theorem**: If Q is a SELECT-FROM-WHERE query that does not have subqueries, and no aggregates, then it is monotone.

- **Proof.** We use the nested loop semantics: if we insert a tuple in a relation $R_i$, this will not remove any tuples from the answer.
Monotone Queries

- **Theorem**: If Q is a SELECT-FROM-WHERE query that does not have subqueries, and no aggregates, then it is monotone.

- **Proof.** We use the nested loop semantics: if we insert a tuple in a relation $R_i$, this will not remove any tuples from the answer.

```sql
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
```

```
for x_1 in R_1 do
  for x_2 in R_2 do
    ...
    for x_n in R_n do
      if Conditions
        output (a_1,...,a_k)
```
Monotone Queries

• **Theorem**: If Q is a SELECT-FROM-WHERE query that does not have subqueries, and no aggregates, then it is monotone.

• **Proof.** We use the nested loop semantics: if we insert a tuple in a relation $R_i$, this will not remove any tuples from the answer.

```
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
```

```python
for x_1 in R_1 do
    for x_2 in R_2 do
        ...
        for x_n in R_n do
            if Conditions
                output (a_1, ..., a_k)

...can’t lose anything here.
```
Monotone Queries

• The query:

Find all companies s.t. all their products have price < 200 is not monotone
Monotone Queries

- The query:

Find all companies s.t. all their products have price < 200 is not monotone
Monotone Queries

• The query:

Find all companies s.t. **all** their products have price < 200
is not monotone

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>c001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>c001</td>
<td>Sunworks</td>
<td>Bonn</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>c001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>c001</td>
<td>Sunworks</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunworks</td>
</tr>
</tbody>
</table>

**Consequence:** If a query is not monotone, then we cannot write it as a SELECT-FROM-WHERE query without nested subqueries
Queries that must be nested

• Queries with universal quantifiers or with negation
Queries that must be nested

- Queries with universal quantifiers or with negation

- Queries with aggregates are usually not monotone
  - \text{sum(\ldots)} and \text{count(\ast)} are NOT monotone, because they do not satisfy set containment
  - select \text{count(\ast)} from \text{R} is not monotone!
Introduction to Data Management
CSE 414

Lecture 7-8: SQL Wrap-up
Relational Algebra
Announcements

• Webquiz tonight

• Makeup lecture tomorrow, 5:30pm, this room
GROUP BY v.s. Nested Queries

SELECT product, Sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product

SELECT DISTINCT x.product, (SELECT Sum(y.quantity)
FROM Purchase y
WHERE x.product = y.product
AND y.price > 1)
AS TotalSales
FROM Purchase x
WHERE x.price > 1

Why twice?
More Unnesting

Find authors who wrote ≥ 10 documents:
More Unnesting

Find authors who wrote ≥ 10 documents:

Attempt 1: with nested queries

```sql
SELECT DISTINCT Author.name
FROM Author
WHERE (SELECT count(Wrote.url)
        FROM Wrote
        WHERE Author.login=Wrote.login)
     >= 10
```

This is SQL by a novice
More Unnesting

Find authors who wrote ≥ 10 documents:

Attempt 1: with nested queries

Attempt 2: using GROUP BY and HAVING

```sql
SELECT Author.name
FROM Author, Wrote
WHERE Author.login=Wrote.login
GROUP BY Author.name
HAVING count(wrote.url) >= 10
```

This is SQL by an expert
Finding Witnesses

For each city, find the most expensive product made in that city
Finding Witnesses

For each city, find the most expensive product made in that city

Finding the maximum price is easy...

\[
\text{SELECT x.city, max(y.price)} \\
\text{FROM Company x, Product y} \\
\text{WHERE x.cid = y.cid} \\
\text{GROUP BY x.city;} \\
\]

But we need the witnesses, i.e., the products with max price
To find the witnesses, compute the maximum price in a subquery (in FROM or in WITH)

```sql
WITH CityMax AS
    (SELECT x.city, max(y.price) as maxprice
     FROM Company x, Product y
     WHERE x.cid = y.cid
     GROUP BY x.city)
```
Finding Witnesses

To find the witnesses, compute the maximum price in a subquery (in FROM or in WITH)

```sql
WITH CityMax AS
  (SELECT x.city, max(y.price) as maxprice
   FROM Company x, Product y
   WHERE x.cid = y.cid
   GROUP BY x.city)
SELECT DISTINCT u.city, v.pname, v.price
FROM Company u, Product v, CityMax w
WHERE u.cid = v.cid
  AND u.city = w.city
  AND v.price = w.maxprice;
```
Finding Witnesses

To find the witnesses, compute the maximum price in a subquery (in FROM or in WITH)

```
SELECT DISTINCT u.city, v.pname, v.price
FROM Company u, Product v,
     (SELECT x.city, max(y.price) as maxprice
      FROM Company x, Product y
      WHERE x.cid = y.cid
      GROUP BY x.city) w
WHERE u.cid = v.cid
   and u.city = w.city
   and v.price = w.maxprice;
```
Finding Witnesses

Or we can use a subquery in where clause

```
SELECT u.city, v.pname, v.price
FROM Company u, Product v
WHERE u.cid = v.cid
  AND v.price >= ALL (SELECT y.price
                       FROM Company x, Product y
                       WHERE u.city = x.city
                       AND x.cid = y.cid);
```
There is a more concise solution here:

```
SELECT u.city, v.pname, v.price
FROM Company u, Product v, Company x, Product y
WHERE u.cid = v.cid
    and u.city = x.city
    and x.cid = y.cid
GROUP BY u.city, v.pname, v.price
HAVING v.price = max(y.price)
```
SQL: Our first language for the relational model

- Projections
- Selections
- Joins (inner and outer)
- Inserts, updates, and deletes
- Aggregates
- Grouping
- Ordering
- Nested queries
Relational Algebra
Relational Algebra

• In SQL we say *what* we want
• In RA we can express *how* to get it
• RA = set-at-a-time algebra for relations

• Every DBMS implementations converts a SQL query to RA in order to execute it
• An RA expression is called a *query plan*
Basics

• Inputs: Relations (with attributes)
• RA: defines a function on relations
  – Returns a relation
  – Can be composed together
  – Often displayed using a tree rather than linearly
  – Use Greek symbols: $\sigma$, $\pi$, $\delta$, etc
Sets v.s. Bags

- Sets: {a,b,c}, {a,d,e,f}, { }, . . .
- Bags: {a, a, b, c}, {b, b, b, b, b}, . . .

Relational Algebra has two flavors:
- Set semantics = standard Relational Algebra
- Bag semantics = extended Relational Algebra

DB systems implement bag semantics (Why?)
Relational Algebra Operators

- Union $\cup$, intersection $\cap$, difference $-$
- Selection $\sigma$
- Projection $\pi$
- Cartesian product $\times$, join $\Join$
- (Rename $\rho$)
- Duplicate elimination $\delta$
- Grouping and aggregation $\gamma$
- Sorting $\tau$

All operators take in 1 or 2 relations as inputs and return another relation.
Union and Difference

$R1 \cup R2$

$R1 - R2$

Only make sense if $R1$, $R2$ have the same schema

What do they mean over bags?
What about Intersection?

• Derived operator using minus

\[ R_1 \cap R_2 = R_1 - (R_1 - R_2) \]

• Derived using join

\[ R_1 \cap R_2 = R_1 \bowtie R_2 \]
Selection

• Returns all tuples which satisfy a condition

\[ \sigma_c(R) \]

• Examples
  - \[ \sigma_{\text{Salary} > 40000} (\text{Employee}) \]
  - \[ \sigma_{\text{name} = \text{“Smith”}} (\text{Employee}) \]

• The condition c can be =, <, <=, >, >=, <> combined with AND, OR, NOT
### Employee

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>5423341</td>
<td>Smith</td>
<td>60000</td>
</tr>
<tr>
<td>4352342</td>
<td>Fred</td>
<td>50000</td>
</tr>
</tbody>
</table>

\( \sigma_{\text{Salary} > 40000} \) (Employee)

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>5423341</td>
<td>Smith</td>
<td>60000</td>
</tr>
<tr>
<td>4352342</td>
<td>Fred</td>
<td>50000</td>
</tr>
</tbody>
</table>
Projection

• Eliminates columns

\[ \pi_{A_1, \ldots, A_n}(R) \]

• Example: project social-security number and names:

\[ \pi_{\text{SSN}, \text{Name}}(\text{Employee}) \rightarrow \text{Answer(SSN, Name)} \]

Different semantics over sets or bags! Why?
<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>5423341</td>
<td>John</td>
<td>60000</td>
</tr>
<tr>
<td>4352342</td>
<td>John</td>
<td>20000</td>
</tr>
</tbody>
</table>

\[ \pi_{Name, Salary}(Employee) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>John</td>
<td>60000</td>
</tr>
<tr>
<td>John</td>
<td>20000</td>
</tr>
</tbody>
</table>

Bag semantics

Set semantics

Which is more efficient?
## Composing RA Operators

### Patient Table

<table>
<thead>
<tr>
<th>no</th>
<th>name</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p1</td>
<td>98125</td>
<td>flu</td>
</tr>
<tr>
<td>2</td>
<td>p2</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>3</td>
<td>p3</td>
<td>98120</td>
<td>lung</td>
</tr>
<tr>
<td>4</td>
<td>p4</td>
<td>98120</td>
<td>heart</td>
</tr>
</tbody>
</table>

### \( \sigma_{\text{disease='heart'}}(\text{Patient}) \)

<table>
<thead>
<tr>
<th>no</th>
<th>name</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>p2</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>4</td>
<td>p4</td>
<td>98120</td>
<td>heart</td>
</tr>
</tbody>
</table>

### \( \pi_{\text{zip,disease}}(\text{Patient}) \)

<table>
<thead>
<tr>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>98125</td>
<td>flu</td>
</tr>
<tr>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>98120</td>
<td>lung</td>
</tr>
<tr>
<td>98120</td>
<td>heart</td>
</tr>
</tbody>
</table>

### \( \pi_{\text{zip,disease}}(\sigma_{\text{disease='heart'}}(\text{Patient})) \)

<table>
<thead>
<tr>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>98120</td>
<td>heart</td>
</tr>
</tbody>
</table>
Cartesian Product

- Each tuple in R1 with each tuple in R2

  \[ R1 \times R2 \]

- Rare in practice; mainly used to express joins
Cross-Product Example

<table>
<thead>
<tr>
<th>Employee</th>
<th>Dependent</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>SSN</td>
</tr>
<tr>
<td>John</td>
<td>9999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
</tr>
<tr>
<td>EmpSSN</td>
<td>DepName</td>
</tr>
<tr>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>7777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

Employee × Dependent

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>9999999999</td>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>John</td>
<td>9999999999</td>
<td>7777777777</td>
<td>Joe</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
<td>7777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>
Renaming

• Changes the schema, not the instance

\[ \rho_{B_1, \ldots, B_n}(R) \]

• Example:
  – Given Employee(Name, SSN)
  – \( \rho_{N, S}(Employee) \) \( \rightarrow \) Answer(N, S)
Natural Join

\[ R1 \bowtie R2 \]

- Meaning: \( R1 \bowtie R2 = \Pi_A (\sigma_\theta (R1 \times R2)) \)

- Where:
  - Selection \( \sigma_\theta \) checks equality of all common attributes (i.e., attributes with same names)
  - Projection \( \Pi_A \) eliminates duplicate common attributes
### Natural Join Example

**R**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>Z</td>
<td>V</td>
</tr>
</tbody>
</table>

**S**

<table>
<thead>
<tr>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>Z</td>
<td>V</td>
</tr>
</tbody>
</table>

\[ R \bowtie S = \prod_{ABC}(\sigma_{R.B=S.B}(R \times S)) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>X</td>
<td>Z</td>
<td>V</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
<td>V</td>
</tr>
<tr>
<td>Z</td>
<td>V</td>
<td>W</td>
</tr>
</tbody>
</table>
### Natural Join Example 2

#### AnonPatient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

#### Voters V

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>Bob</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

#### P \Join V

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>Alice</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>Bob</td>
</tr>
</tbody>
</table>
Natural Join

• Given schemas $R(A, B, C, D), S(A, C, E)$, what is the schema of $R \Join S$?

• Given $R(A, B, C), S(D, E)$, what is $R \Join S$?

• Given $R(A, B), S(A, B)$, what is $R \Join S$?
Theta Join

- A join that involves a predicate
  \[ R1 \bowtie_\theta R2 = \sigma_\theta (R1 \times R2) \]
- Here \( \theta \) can be any condition
- No projection in this case!
- For our voters/patients example:
  \[ P \bowtie P.zip = V.zip \text{ and } P.age \geq V.age - 1 \text{ and } P.age \leq V.age + 1 \]

AnonPatient (age, zip, disease)
Voters (name, age, zip)
Equijoin

• A theta join where \( \theta \) is an equality predicate

\[
R_1 \bowtie_\theta R_2 = \sigma_\theta (R_1 \times R_2)
\]

• By far the most used variant of join in practice
• What is the relationship with natural join?
Equijoin Example

AnonPatient $P$

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

Voters $V$

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

$P \bowtie_{P\text{.age}=V\text{.age}} V$

<table>
<thead>
<tr>
<th>P\text{.age}</th>
<th>P\text{.zip}</th>
<th>P\text{.disease}</th>
<th>V\text{.name}</th>
<th>V\text{.age}</th>
<th>V\text{.zip}</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>
Natural Join Example

AnonPatient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

Voters V

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

P \bowtie V

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
<th>name</th>
<th>V.age</th>
<th>V.zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>
Join Summary

- **Theta-join**: $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
  - Join of $R$ and $S$ with a join condition $\theta$
  - Cross-product followed by selection $\theta$
  - No projection

- **Equijoin**: $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$
  - Join condition $\theta$ consists only of equalities
  - No projection

- **Natural join**: $R \bowtie S = \pi_A(\sigma_{\theta}(R \times S))$
  - Equality on **all** fields with same name in $R$ and in $S$
  - Projection $\pi_A$ drops all redundant attributes
So Which Join Is It?

When we write $R \bowtie S$ we usually mean an equijoin, but we often omit the equality predicate when it is clear from the context.
More Joins

• Outer join
  – Include tuples with no matches in the output
  – Use NULL values for missing attributes
  – Does not eliminate duplicate columns

• Variants
  – Left outer join
  – Right outer join
  – Full outer join
Outer Join Example

AnonPatient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
<tr>
<td>33</td>
<td>98120</td>
<td>lung</td>
</tr>
</tbody>
</table>

AnnonJob J

<table>
<thead>
<tr>
<th>job</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>lawyer</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>cashier</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

P \Join\ J

<table>
<thead>
<tr>
<th>P.age</th>
<th>P.zip</th>
<th>P.disease</th>
<th>J.job</th>
<th>J.age</th>
<th>J.zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>lawyer</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>cashier</td>
<td>20</td>
<td>98120</td>
</tr>
<tr>
<td>33</td>
<td>98120</td>
<td>lung</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
Some Examples

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,qty,price)

Name of supplier of parts with size greater than 10
\[
\pi_{sname}(\text{Supplier} \bowtie (\text{Supply} \bowtie (\sigma_{psize>10} \text{Part})))
\]

Name of supplier of red parts or parts with size greater than 10
\[
\pi_{sname}(\text{Supplier} \bowtie (\text{Supply} \bowtie (\sigma_{psize>10} \text{Part}) \cup \sigma_{pcolor='red'} \text{Part})))
\]
\[
\pi_{sname}(\text{Supplier} \bowtie (\text{Supply} \bowtie (\sigma_{psize>10} \lor pcolor='red' \text{Part})))
\]

Can be represented as trees as well

CSE 414 - 2019sp 248
Some Examples

Supplier(sno, sname, scity, sstate)
Part(pno, pname, psize, pcolor)
Supply(sno, pno, qty, price)

Name of supplier of parts with size greater than 10
Project[sname](Supplier Join[sno=sno]
  (Supply Join[pno=pno] (Select[psize>10](Part))))

Name of supplier of red parts or parts with size greater than 10
Project[sname](Supplier Join[sno=sno]
  (Supply Join[pno=pno]
   ((Select[psize>10](Part)) Union
    (Select[pcolor='red'](Part))))

Project[sname](Supplier Join[sno=sno] (Supply Join[pno=pno]
(Select[psize>10 OR pcolor='red'](Part))))

Can be represented as trees as well
Representing RA Queries as Trees

\[
\text{SELECT } z.\text{sname} \\
\text{FROM Part } x, \text{ Supply } y, \text{ Supplier } z \\
\text{WHERE } x.\text{psize} > 10 \\
\quad \text{and } x.\text{pno} = y.\text{pno} \\
\quad \text{and } y.\text{sno} = z.\text{sno}
\]
Relational Algebra Operators

- Union $\cup$
- Intersection $\cap$
- Difference $-$
- Selection $\sigma$
- Projection $\pi$
- Cartesian product $\times$
- Join $\Join$
- Rename $\rho$
- Duplicate elimination $\delta$
- Grouping and aggregation $\gamma$
- Sorting $\tau$

All operators take in 1 or 2 relations as inputs and return another relation
Extended RA: Operators on Bags

• Duplicate elimination $\delta$

• Grouping $\gamma$
  – Takes in relation and a list of grouping operations (e.g., aggregates). Returns a new relation.

• Sorting $\tau$
  – Takes in a relation, a list of attributes to sort on, and an order. Returns a new relation.
Using Extended RA Operators

SELECT city, sum(quantity) 
FROM sales 
GROUP BY city 
HAVING count(*) > 100

T1, T2 = temporary tables

Answer

\[\Pi_{\text{city, q}} (\gamma_{\text{city, sum(quantity)} \rightarrow q, \text{count(*)} \rightarrow c} (T1(city, q, c)))\]

\[\sigma_{c > 100} (T2(city, q, c))\]
Typical Plan for a Query (1/2)

```
SELECT fields
FROM R, S, ...
WHERE condition
```

SELECT-PROJECT-JOIN Query

Answer

\[ \pi_{\text{fields}} \]

\[ \sigma_{\text{selection condition}} \]

\[ \text{join condition} \]

\[ \text{join condition} \]

R

S

CSE 414 - 2019sp
Typical Plan for a Query (1/2)

\[
\begin{align*}
\sigma_{\text{having condition}} & \quad | \quad \gamma_{\text{fields, sum/count/min/max(fields)}} \\
\pi_{\text{fields}} & \quad | \quad \sigma_{\text{where condition}} \\
\text{join condition} & \quad | \quad \text{...} \quad | \quad \text{...}
\end{align*}
\]

\[
\text{SELECT fields} \\
\text{FROM R, S, ...} \\
\text{WHERE condition} \\
\text{GROUP BY fields} \\
\text{HAVING condition}
\]
How about Subqueries?

```
SELECT Q.sno
FROM Supplier Q
WHERE Q.sstate = 'WA'
  and not exists
  (SELECT *
   FROM Supply P
   WHERE P.sno = Q.sno
   and P.price > 100)
```
**How about Subqueries?**

```
SELECT  Q.sno 
FROM    Supplier Q 
WHERE   Q.sstate = 'WA'  
        and not exists 
        (SELECT *  
         FROM  Supply P  
         WHERE P.sno = Q.sno  
         and P.price > 100)
```

**Correlation!**

*Supplier*(sno, sname, scity, sstate)
*Part*(pno, pname, psize, pcolor)
*Supply*(sno, pno, price)
How about Subqueries?

```sql
SELECT Q.sno
FROM Supplier Q
WHERE Q.sstate = 'WA'
    and not exists
    (SELECT *
     FROM Supply P
     WHERE P.sno = Q.sno
     and P.price > 100)
```

De-Correlation

```sql
SELECT Q.sno
FROM Supplier Q
WHERE Q.sstate = 'WA'
    and Q.sno not in
    (SELECT P.sno
     FROM Supply P
     WHERE P.price > 100)
```
How about Subqueries?

\[
\begin{align*}
&\text{(SELECT } Q\text{.sno} \\
&\quad \text{FROM Supplier } Q \\
&\quad \text{WHERE } Q\text{.sstate} = \text{‘WA’}) \\
&\quad \text{EXCEPT} \\
&\quad \text{(SELECT } P\text{.sno} \\
&\quad \text{FROM Supply } P \\
&\quad \text{WHERE } P\text{.price} > 100) \\
&\quad \text{EXCEPT} = \text{set difference}
\end{align*}
\]

\[
\begin{align*}
&\text{SELECT } Q\text{.sno} \\
&\quad \text{FROM Supplier } Q \\
&\quad \text{WHERE } Q\text{.sstate} = \text{‘WA’} \quad \text{and } Q\text{.sno} \text{ not in} \\
&\quad \text{(SELECT } P\text{.sno} \\
&\quad \text{FROM Supply } P \\
&\quad \text{WHERE } P\text{.price} > 100)
\end{align*}
\]
How about Subqueries?

\[
\begin{align*}
&\left(\text{SELECT } Q.sno \\
&\text{FROM Supplier Q} \\
&\text{WHERE } Q.sstate = 'WA'\right) \\
\text{EXCEPT} \\
&\left(\text{SELECT } P.sno \\
&\text{FROM Supply P} \\
&\text{WHERE } P.price > 100\right)
\end{align*}
\]
Summary of RA and SQL

- SQL = a declarative language where we say *what* data we want to retrieve
- RA = an algebra where we say *how* we want to retrieve the data
- **Theorem**: SQL and RA can express exactly the same class of queries

RDBMS translates SQL → RA, then optimize RA
Summary of RA and SQL

- SQL (and RA) cannot express ALL queries that we could write in, say, Java
- Example:
  - Parent(p,c): find all descendants of ‘Alice’
  - No RA query can compute this!
  - This is called a recursive query
- Next lecture: Datalog is an extension that can compute recursive queries
Introduction to Data Management

CSE 414

Lectures 9-10: Datalog
Class Overview

- Unit 1: Intro
- Unit 2: Relational Data Models and Query Languages
  - Data models, SQL, Datalog, Relational Algebra
- Unit 3: Non-relational data
- Unit 4: RDMBS internals and query optimization
- Unit 5: Parallel query processing
- Unit 6: DBMS usability, conceptual design
- Unit 7: Transactions
What is Datalog?

- Another query language for relational model
  - Designed in the 80’s
  - Simple, concise, elegant
  - Extends relational queries with recursion

- Today is a hot topic:
  - Souffle (we will use in HW4)
  - Eve http://witheve.com/
  - Differential datalog
    https://github.com/frankmcsherry/differential-dataflow
  - Beyond databases in many research projects: network protocols, static program analysis
• Open-source implementation of Datalog DBMS
• Under active development
• Commercial implementations are available
  – More difficult to set up and use
• “sqlite” of Datalog
  – Set-based rather than bag-based

• Install in your VM
  – Run `sudo yum install souffle` in terminal
  – More details in upcoming HW4
Why bother with *yet* another relational query language?
Example: storing FB friends

As a graph

Peter
  Mary
    Phil
  John

Or

<table>
<thead>
<tr>
<th>Person1</th>
<th>Person2</th>
<th>is_friend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>John</td>
<td>1</td>
</tr>
<tr>
<td>John</td>
<td>Mary</td>
<td>0</td>
</tr>
<tr>
<td>Mary</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>Phil</td>
<td>Peter</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

As a relation

We will learn the tradeoffs of different data models later this quarter
Compute your friends graph

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
<th>isFriend</th>
</tr>
</thead>
<tbody>
<tr>
<td>Peter</td>
<td>John</td>
<td>1</td>
</tr>
<tr>
<td>John</td>
<td>Mary</td>
<td>0</td>
</tr>
<tr>
<td>Mary</td>
<td>Phil</td>
<td>1</td>
</tr>
<tr>
<td>Phil</td>
<td>Peter</td>
<td>1</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Friends(p1, p2, isFriend)

Datalog allows us to write *recursive queries* easily

My own friends

```
SELECT f1.p2
FROM Friends as f1,
(SELECT f.p2
 FROM Friends as f
 WHERE f.p1 = 'me' AND f.isFriend = 1) as f2
WHERE f1.p1 = f2.p2 AND f1.isFriend = 1
```

My FoF

```
SELECT f1.p2
FROM Friends as f1,
(SELECT f.p2
 FROM Friends as f
 WHERE f.p1 = 'me' AND f.isFriend = 1) as f2
WHERE f1.p1 = f2.p2 AND f1.isFriend = 1
```

My FoFoF...

When does it end???
**Datalog: Facts and Rules**

**Facts** = tuples in the database

**Rules** = queries

**Schema**

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)
Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759, 'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Rules = queries

Table declaration

Types in Souffle:
- number
- symbol (aka varchar)

Insert data
Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(29445, ‘Ave Maria’, 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z=1940.

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)
Datalog: Facts and Rules

**Facts** = tuples in the database

- Actor(344759, ‘Douglas’, ‘Fowley’).
- Casts(344759, 29851).
- Casts(355713, 29000).

**Rules** = queries

Q1(y) :- Movie(x, y, z), z = 1940.

Find Movies made in 1940
Datalog: Facts and Rules

**Facts** = tuples in the database

- `Actor(344759,'Douglas', 'Fowley')`.
- `Casts(344759, 29851)`.
- `Casts(355713, 29000)`.
- `Movie(7909, 'A Night in Armour', 1910)`.
- `Movie(29000, 'Arizona', 1940)`.
- `Movie(29445, 'Ave Maria', 1940)`.

**Rules** = queries

- `Q1(y) :- Movie(x,y,z), z=1940`.

**SQL**

```
SELECT name
FROM Movie
WHERE year = 1940
```

Find Movies made in 1940
Datalog: Facts and Rules

**Facts** = tuples in the database

- Actor(344759, ‘Douglas’, ‘Fowley’).
- Casts(344759, 29851).
- Casts(355713, 29000).

**Rules** = queries

Q1(y) :- Movie(x, y, z), z=1940.

Order of variable matters!

Find Movies made in 1940
Datalog: Facts and Rules

**Facts** = tuples in the database

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(29445, ‘Ave Maria’, 1940).

**Rules** = queries

Q1(y) :- Movie(iDontCare, y, z), z=1940.

Find Movies made in 1940
Datalog: Facts and Rules

**Facts** = tuples in the database

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(29445, ‘Ave Maria’, 1940).

**Rules** = queries

Q1(y) :- Movie(_, y, z), z=1940.

_ = “don’t care” variables

Find Movies made in 1940
**Datalog: Facts and Rules**

**Facts** = tuples in the database

```
Actor(344759, 'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).
```

**Rules** = queries

```
Q1(y) :- Movie(x, y, z), z=1940.
Q2(f, l) :- Actor(z, f, l), Casts(z, x), Movie(x, y, 1940).
```
Datalog: Facts and Rules

**Facts** = tuples in the database

- Actor(344759, ‘Douglas’, ‘Fowley’).
- Casts(344759, 29851).
- Casts(355713, 29000).

**Rules** = queries

- Q1(y) :- Movie(x,y,z), z=1940.
- Q2(f,l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,1940).
Datalog: Facts and Rules

Facts = tuples in the database

- Actor(344759,'Douglas', 'Fowley')
- Casts(344759, 29851)
- Casts(355713, 29000)
- Movie(7909, 'A Night in Armour', 1910)
- Movie(29000, 'Arizona', 1940)
- Movie(29445, 'Ave Maria', 1940)

Rules = queries

- Q1(y) :- Movie(x,y,z), z=1940.
- Q2(f,l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,1940).

Find Actors who acted in Movies made in 1940
Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Actor(344759, 'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
Movie(7909, 'A Night in Armour', 1910).
Movie(29000, 'Arizona', 1940).
Movie(29445, 'Ave Maria', 1940).

Q1(y) :- Movie(x,y,z), z=1940.

Q2(f,l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,1940).

Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
           Casts(z,x2), Movie(x2,y2,1940).
Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Actors who acted in a Movie in 1940 and in one in 1910

Actors: Douglas Fowley
Movies: A Night in Armour, Arizona, Ave Maria

Q1(y) :- Movie(x,y,z), z=1940.
Q2(f,l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,1940).
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910), Casts(z,x2), Movie(x2,y2,1940).
Datalog: Facts and Rules

**Facts** = tuples in the database

- `Actor(344759, 'Douglas', 'Fowley').`  
- `Casts(344759, 29851).`  
- `Casts(355713, 29000).`  
- `Movie(7909, 'A Night in Armour', 1910).`  
- `Movie(29000, 'Arizona', 1940).`  
- `Movie(29445, 'Ave Maria', 1940).`  

**Rules** = queries

- `Q1(y) :- Movie(x,y,z), z=1940.`
- `Q2(f,l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,1940).`  
- `Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910), Casts(z,x2), Movie(x2,y2,1940).`  

**Extensional Database Predicates** = EDB = Actor, Casts, Movie

**Intensional Database Predicates** = IDB = Q1, Q2, Q3
Datalog: Terminology

Q2(f, l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,1940).

f, l = head variables
x, y, z = existential variables
More Datalog Terminology

Q(args) :- R1(args), R2(args), ...

• $R_i(\text{args}_i)$ called an **atom**, or a **relational predicate**
• $R_i(\text{args}_i)$ evaluates to true when relation $R_i$ contains the tuple described by $\text{args}_i$.
  - Example: $\text{Actor}(344759, \text{‘Douglas’}, \text{‘Fowley’})$ is true
• In addition we can also have arithmetic predicates
  - Example: $z > 1940$.
• Book uses **AND** instead of ,

Q(args) :- R1(args) **AND** R2(args) ...
Datalog program

• A Datalog program consists of several rules
• Importantly, rules may be recursive!
  – Recall CSE 143!
• Usually there is one distinguished predicate that’s the output
• We will show an example first, then give the general semantics.
Announcements (1/30/2019)

• Webquiz 5 (datalog): deadline is now on Thursday, 1/31

• Next week:
  – Guest lecturer (Jonathan) on M+W
  – Lecture on Friday canceled; replaced with Midterm Review on Saturday (2/9), 2pm.

• Check the calendar!
R encodes a graph
e.g., connected cities

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

R =

<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 encodes a graph e.g., connected cities

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

Multiple rules for the same IDB means OR

What does it compute?
R encodes a graph e.g., connected cities

\[ R = \]

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

Initially: 
T is empty.

\[ T(x,y) :- R(x,y). \]
\[ T(x,y) :- R(x,z), T(z,y). \]
R encodes a graph e.g., connected cities

Example

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

Initially: T is empty.

First iteration:

T =

First rule generates this

Second rule generates nothing (because T is empty)

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

What does it compute?

R encodes a graph e.g., connected cities.
Example

$R = \{(x,y)\}$

Initially:

$T$ is empty.

$T(x,y) \leftarrow R(x,y)$.
$T(x,y) \leftarrow R(x,z), T(z,y)$.

First iteration:

Second iteration:

What does it compute?

New facts
R encodes a graph e.g., connected cities

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>

Initially: T is empty.

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

First iteration: T =

Second iteration: T =

Third iteration: T =

What does it compute?

Example

New fact

Both rules

First rule

Second rule

First iteration:

Second iteration:

Third iteration:
Example

R encodes a graph e.g., connected cities

\[ R = \]

<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>

Initially: T is empty.

First iteration:

\[
T = \]

<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>
</tbody>
</table>

Second iteration:

\[
T = \]

<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>

Third iteration:

\[
T = \]

<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>

Fourth iteration:

\[
T = \]

<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>

No new facts.

DONE

What does it compute?

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

R encodes a graph e.g., connected cities
Datalog Semantics

Fixpoint semantics

• Start:
  \[ \text{IDB}_0 = \text{empty relations} \]
  \[ t = 0 \]

Repeat:
  \[ \text{IDB}_{t+1} = \text{Compute Rules(EDB, IDB}_t) \]
  \[ t = t + 1 \]

Until \( \text{IDB}_t = \text{IDB}_{t-1} \)

• Remark: since rules are monotone:
  \( \emptyset = \text{IDB}_0 \subseteq \text{IDB}_1 \subseteq \text{IDB}_2 \subseteq \ldots \)

• It follows that a datalog program w/o functions (+, *, ...) always terminates. (Why?)
Three Equivalent Programs

R encodes a graph
e.g., connected cities

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

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

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

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

Right linear
Left linear
Non-linear

Question: which terminates in fewest iterations?
More Features

• Aggregates
• Grouping
• Negation
Aggregates

[aggregate name] <var> : { [relation to compute aggregate on] }

\[
\text{min } x : \{ \text{Actor}(x, y, _), y = \text{‘John’} \}
\]

\[
Q(\text{minId}) :- \text{minId} = \text{min } x : \{ \text{Actor}(x, y, _), y = \text{‘John’} \}
\]

Assign variable to the value of the aggregate

Meaning (in SQL)

\[
\text{SELECT min(id) as minId}
\text{FROM Actor as a}
\text{WHERE a.name = ‘John’}
\]
Counting

Q(c) :- c = count : { Actor(_, y, _), y = ‘John’ }

No variable here!

Meaning (in SQL, assuming no NULLs)

```sql
SELECT count(*) as c
FROM Actor as a
WHERE a.name = ‘John’
```
GROUPING

Q(y,c) :- Movie(_,_,y), c = count : { Movie(_,_,y) }

Meaning (in SQL)

```
SELECT m.year, count(*)
FROM Movie as m
GROUP BY m.year
```
Examples

A genealogy database (parent/child)

ParentChild

<table>
<thead>
<tr>
<th>p</th>
<th>c</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Carol</td>
</tr>
<tr>
<td>Bob</td>
<td>Carol</td>
</tr>
<tr>
<td>Bob</td>
<td>David</td>
</tr>
<tr>
<td>Carol</td>
<td>Eve</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Count Descendants

For each person, count his/her descendants

ParentChild(p,c)
Count Descendants

For each person, count his/her descendants

Answer

<table>
<thead>
<tr>
<th>p</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
</tr>
<tr>
<td>Bob</td>
<td>5</td>
</tr>
<tr>
<td>Carol</td>
<td>3</td>
</tr>
<tr>
<td>David</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>1</td>
</tr>
</tbody>
</table>
Count Descendants

For each person, count his/her descendants

Answer

<table>
<thead>
<tr>
<th>p</th>
<th>cnt</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>4</td>
</tr>
<tr>
<td>Bob</td>
<td>5</td>
</tr>
<tr>
<td>Carol</td>
<td>3</td>
</tr>
<tr>
<td>David</td>
<td>2</td>
</tr>
<tr>
<td>Fred</td>
<td>1</td>
</tr>
</tbody>
</table>

Note: Eve and George do not appear in the answer (why?)
Count Descendants

For each person, compute the total number of descendants

// for each person, compute his/her descendants
Count Descendants

For each person, compute the total number of descendants

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
Count Descendants

For each person, compute the total number of descendants

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
Count Descendants

For each person, compute the total number of descendants

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
Count Descendants

For each person, compute the total number of descendants

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,y) }.
Count Descendants

How many descendants does Alice have?

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = \text{count} : \{ D(p,y) \}.
Count Descendants

How many descendants does Alice have?

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,y) }.

// Find the number of descendants of Alice
Count Descendants

How many descendants does Alice have?

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// For each person, count the number of descendants
T(p,c) :- D(p,_), c = count : { D(p,y) }.

// Find the number of descendants of Alice
Q(d) :- T(p,d), p = “Alice”.
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice

Alice

Bob

Carol

Eve

Fred

David

George

Answer

| x | David |
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).
Negation: use “!”

Find all descendants of Bob that are not descendants of Alice

// for each person, compute his/her descendants
D(x,y) :- ParentChild(x,y).
D(x,z) :- D(x,y), ParentChild(y,z).

// Compute the answer: notice the negation
Q(x) :- D(“Bob”,x), !D(“Alice”,x).
Same Generation

Two people are in the *same generation* if they are descendants at the same generation of some common ancestor.

<table>
<thead>
<tr>
<th>p1</th>
<th>p2</th>
</tr>
</thead>
<tbody>
<tr>
<td>Carol</td>
<td>David</td>
</tr>
<tr>
<td>Eve</td>
<td>George</td>
</tr>
<tr>
<td>Fred</td>
<td>George</td>
</tr>
<tr>
<td>Fred</td>
<td>Eve</td>
</tr>
</tbody>
</table>
Same Generation

Compute pairs of people at the same generation

// common parent
Same Generation

Compute pairs of people at the same generation

// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)
Same Generation

Compute pairs of people at the same generation

// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)

// parents at the same generation
Same Generation

Compute pairs of people at the same generation

// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q)
Same Generation

Compute pairs of people at the same generation

// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q)

Problem: this includes answers like SG(Carol, Carol)
And also SG(Eve, George), SG(George, Eve)

How to fix?
Compute pairs of people at the same generation

// common parent
SG(x,y) :- ParentChild(p,x), ParentChild(p,y), x < y

// parents at the same generation
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q), x < y
Here are unsafe datalog rules. What’s “unsafe” about them?

\[
\begin{align*}
U1(x,y) & \leftarrow \text{ParentChild(“Alice”,} x\text{)}, \ y \neq \text{“Bob”} \\
U2(x) & \leftarrow \text{ParentChild(“Alice”,} x\text{)}, \ !\text{ParentChild}(x,y) \\
U3(\text{minId}, \ y) & \leftarrow \text{minId} = \text{min } x : \{ \text{Actor}(x, \ y, \ _) \}
\end{align*}
\]
Here are *unsafe* datalog rules. What’s “unsafe” about them?

U1(x,y) :- ParentChild("Alice",x), y != "Bob"

U2(x) :- ParentChild("Alice",x), !ParentChild(x,y)

U3(minId, y) :- minId = \text{min } x : \{ \text{Actor}(x, y, _) \}
Safe Datalog Rules

Here are unsafe datalog rules. What’s “unsafe” about them?

\[ U_1(x, y) : \neg \text{ParentChild("Alice",x), y \neq "Bob"} \]

\[ U_2(x) : \neg \text{ParentChild("Alice",x), !ParentChild(x,y)} \]

Want Alice’s childless children, but we get all children x (because there exists some y that x is not parent of y)

\[ U_3(\text{minId}, y) : \text{minId} = \min x : \{ \text{Actor}(x, y, _) \} \]
Here are **unsafe** datalog rules. What’s “unsafe” about them?

\[ U1(x,y) :\text{-} \text{ParentChild(“Alice”,}x), \ y \neq \text{“Bob”)} \]

\[ U2(x) :\text{-} \text{ParentChild(“Alice”},x), \ !\text{ParentChild}(x,y) \]

Want Alice’s childless children, but we get all children \( x \) (because there exists some \( y \) that \( x \) is not parent of \( y \))

\[ U3(\text{minId,} y) :\text{-} \text{minId} = \text{min } x : \{ \text{Actor}(x, y, \_)} \}

Unclear what \( y \) is
Here are *unsafe* datalog rules. What’s “unsafe” about them?

\[ U1(x,y) :- \text{ParentChild(“Alice”,x)}, y \neq “Bob” \]

\[ U2(x) :- \text{ParentChild(“Alice”,x)}, !\text{ParentChild(x,y)} \]

A datalog rule is *safe* if every variable appears in some positive, non-aggregated relational atom.

\[ U3(\text{minId}, y) :- \text{minId} = \min x : \{ \text{Actor(x, y, _)} \} \]
Stratified Datalog

• Recursion does not cope well with aggregates or negation
• Example: what does this mean?

\[
\begin{align*}
A() &: - !B(). \\
B() &: - !A().
\end{align*}
\]

• A datalog program is **stratified** if it can be partitioned into **strata**
  – Only IDB predicates defined in strata 1, 2, ..., n may appear under ! or agg in stratum n+1.

• Many Datalog DBMSs (including souffle) accepts only stratified Datalog.
Stratified Datalog

Stratum 1

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(x,y) :- ParentChild(x,y).</td>
</tr>
<tr>
<td>D(x,z) :- D(x,y), ParentChild(y,z).</td>
</tr>
</tbody>
</table>

Stratum 2

<table>
<thead>
<tr>
<th>Rule</th>
</tr>
</thead>
<tbody>
<tr>
<td>T(p,c) :- D(p,_) , c = count : { D(p,y) }.</td>
</tr>
<tr>
<td>Q(d) :- T(p,d), p = “Alice”.</td>
</tr>
</tbody>
</table>

May use D in an agg since it was defined in previous stratum
## Stratified Datalog

<table>
<thead>
<tr>
<th>Stratum 1</th>
<th>Stratum 2</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(x,y) :- ParentChild(x,y).</td>
<td>T(p,c) :- D(p,_), c = count : { D(p,y) }.</td>
</tr>
<tr>
<td>D(x,z) :- D(x,y), ParentChild(y,z).</td>
<td>Q(d) :- T(p,d), p = “Alice”.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Stratum 1</th>
</tr>
</thead>
<tbody>
<tr>
<td>D(x,y) :- ParentChild(x,y).</td>
</tr>
<tr>
<td>D(x,z) :- D(x,y), ParentChild(y,z).</td>
</tr>
<tr>
<td>Q(x) :- D(“Alice”,x), !D(“Bob”,x).</td>
</tr>
</tbody>
</table>

### Non-stratified

<p>| |</p>
<table>
<thead>
<tr>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A() :- !B().</td>
</tr>
<tr>
<td>B() :- !A().</td>
</tr>
</tbody>
</table>

May use D in an agg since it was defined in previous stratum

May use !D

Cannot use !A
Stratified Datalog

- If we don’t use aggregates or negation, then the Datalog program is already stratified

- If we do use aggregates or negation, it is usually quite natural to write the program in a stratified way