Unit 2: The Relational Data Model
SQL
Relational Algebra
Datalog

(9 lectures*)

*Slides may change: refresh each lecture
Introduction to Data Management
CSE 344

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)
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
3 Elements of Data Models

• Instance
  – The actual data

• Schema
  – Describe what data is being stored

• Query language
  – How to retrieve and manipulate data
Turing Awards in Data Management

Charles Bachman, 1973
*IDS and CODASYL*

Ted Codd, 1981
*Relational model*

Jim Gray, 1998
*Transaction processing*

Michael Stonebraker, 2014
*INGRES and Postgres*
Relational Model

- Data is a collection of relations / tables:
  - Mathematically, a relation is a set of tuples:
    - Each tuple appears 0 or 1 times in the table.
    - Order of the rows is unspecified.

<table>
<thead>
<tr>
<th>columns / attributes / fields</th>
<th>rows / tuples / records</th>
</tr>
</thead>
<tbody>
<tr>
<td>cname</td>
<td>country</td>
</tr>
<tr>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
</tr>
<tr>
<td>HappyCam</td>
<td>Canada</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

• Degree (arity) of a relation = #attributes
• Each attribute has a type.
  – Examples types:
    • Strings: CHAR(20), VARCHAR(50), TEXT
    • Numbers: INT, SMALLINT, FLOAT
    • MONEY, DATETIME, …
    • Few more that are vendor specific
  – Statically and strictly enforced
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

<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>
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 *primary key*

E.g.: primary key = SSN
Foreign Key

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

<table>
<thead>
<tr>
<th>Company</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>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>Country</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>name</td>
<td>population</td>
</tr>
<tr>
<td>USA</td>
<td>320M</td>
</tr>
<tr>
<td>Japan</td>
<td>127M</td>
</tr>
</tbody>
</table>
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>
</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>

<table>
<thead>
<tr>
<th>products</th>
<th></th>
<th>product</th>
</tr>
</thead>
<tbody>
<tr>
<td>pname</td>
<td>price</td>
<td>category</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>
<tr>
<td>AC</td>
<td>300</td>
<td>Appliance</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>
<th>products</th>
</tr>
</thead>
<tbody>
<tr>
<td>Canon</td>
<td>Japan</td>
<td>50000</td>
<td>Y</td>
<td>SingleTouch</td>
</tr>
<tr>
<td>Hitachi</td>
<td>Japan</td>
<td>30000</td>
<td>Y</td>
<td>Gadget</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>category</th>
</tr>
</thead>
<tbody>
<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>

<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>
</tbody>
</table>

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>
Demo 1 (cont’d)
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 344

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
### Joins in SQL

**Retrieve all Japanese products that cost < $150**

<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>
Joins in SQL

---

Retrieve all Japanese products that cost < $150

```sql
SELECT  pname, price  
FROM    Product, Company  
WHERE   ...
```

---

<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 Japanese products that cost < $150

```
SELECT pname, price
FROM Product, Company
WHERE manufacturer=cname AND country='Japan' AND price < 150
```
Joins in SQL

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

Retrieve all USA companies that manufacture "gadget" products

<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>

Retrieved 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>
Joins in SQL

Product\((p\text{name}, \text{price, category, manufacturer})\)
Company\((c\text{name}, \text{country})\)

<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?
Demo 2 – cont’d
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

<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
Inner Join

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>

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

```sql
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
```
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 
INNER JOIN Sales S 
ON E.id = S.employeeID
```
Outer Join

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

Retrieve employees and their sales

```
SELECT *
FROM Employee E
LEFT OUTER JOIN Sales S
ON E.id = S.employeeID
```
Introduction to Data Management
CSE 344

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 is foreign key to Company

SELECT DISTINCT cname
FROM Product, Company
WHERE 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
```
(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

<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>

<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>

Product

Company
(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 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
```
for \( x_1 \) in R1:
  for \( x_2 \) in R2:
    ...
    for \( x_m \) in R_m:
      if Cond(\( x_1, x_2 \ldots \)):
        output(\( x_1.a_1, x_2.a_2, \ldots x_m.a_m \))

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

\[
\text{Product}(\text{pname}, \text{price}, \text{category}, \text{manufacturer})
\]
\[
\text{Company}(\text{cname}, \text{country})
\]

-- manufacturer is foreign key to Company

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

Product(pname, price, category, manufacturer)  
Company(cname, country)  
-- manufacturer is foreign key to Company

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?
Another example

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

-- manufacturer is foreign key to Company

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'
         OR x.category = 'photography');

What about this?
Another example

Product(\textit{pname}, \textit{price}, \textit{category}, \textit{manufacturer})
Company(\textit{cname}, \textit{country})

-- manufacturer is foreign key to Company

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

\texttt{SELECT DISTINCT z.cname}
\texttt{FROM Product x, Product y, Company z}
\texttt{WHERE z.country = 'USA'}
\hspace{1cm} \texttt{AND x.manufacturer = z.cname}
\hspace{1cm} \texttt{AND y.manufacturer = z.cname}
\hspace{1cm} \texttt{AND x.category = 'gadget'}
\hspace{1cm} \texttt{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;
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**

```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

```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;
```

**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>SingleTouch</td>
<td>photo</td>
<td>Hitachi</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>Photo</td>
<td>GizmoWorks</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>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;
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;

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>x</strong></td>
<td><strong>z</strong></td>
</tr>
<tr>
<td><strong>pname</strong></td>
<td><strong>cname</strong></td>
</tr>
<tr>
<td><strong>category</strong></td>
<td><strong>country</strong></td>
</tr>
<tr>
<td><strong>manufacturer</strong></td>
<td></td>
</tr>
<tr>
<td>Gizmo</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>gadget</td>
<td>USA</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>Hitachi</td>
</tr>
<tr>
<td>Photo</td>
<td>Japan</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>GizmoWorks</td>
</tr>
</tbody>
</table>
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;
```

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td>x</td>
<td></td>
<td>z</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>y</td>
<td></td>
<td></td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

- **Product**
  - **pname**: Gizmo, SingleTouch, MultiTouch
  - **category**: gadget, photo, Photo
  - **manufacturer**: GizmoWorks, Hitachi

- **Company**
  - **cname**: GizmoWorks, Hitachi
  - **country**: USA, Japan
### 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;
```

#### Product Table

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

#### Company Table

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

#### Table Join

<table>
<thead>
<tr>
<th>x.pname</th>
<th>x.category</th>
<th>x.manufacturer</th>
<th>y.pname</th>
<th>y.category</th>
<th>y.manufacturer</th>
<th>z.cname</th>
<th>z.country</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
<td>GizmoWorks</td>
<td>MultiTouch</td>
<td>Photo</td>
<td>GizmoWorks</td>
<td>GizmoWorks</td>
<td>USA</td>
</tr>
</tbody>
</table>

The query selects the distinct company names (`z.cname`) where the country is 'USA' and the product category is 'gadget' for `x`, and 'photo' for `y`, with matching manufacturers for `x` and `y`. The relationships are visualized with a table and a self-join diagram.
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

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

We want to include products that are never sold, but some are not listed! Why?
Outer joins

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

<table>
<thead>
<tr>
<th>Product</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Category</td>
</tr>
<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>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>ProdName</td>
<td>Store</td>
</tr>
<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

### 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 
```
**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

<table>
<thead>
<tr>
<th>Product</th>
<th>Purchase</th>
</tr>
</thead>
<tbody>
<tr>
<td>Name</td>
<td>Category</td>
</tr>
<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>
```
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
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>
<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

---

**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>
<tr>
<td>Camera</td>
<td>Wiz</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>
</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>

**SQL Query**

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

**Product**

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

**Purchase**

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

**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>
</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>
<tr>
<td>NULL</td>
<td>Foo</td>
</tr>
</tbody>
</table>

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

```sql
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

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

Let's try the following

```sql
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:

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

same as `count(*)` if no nulls

We probably want:

```
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 344

Lecture 5: Grouping and Query Evaluation
Announcement

• The Webquiz is due tonight!
Grouping and Aggregation

Purchase(product, price, quantity)

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

```sql
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>
Other Examples

Compare these two queries:

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

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

```sql
SELECT product,
    sum(quantity) AS SumQuantity,
    max(price) AS MaxPrice
FROM Purchase
GROUP BY product
```

What does it return?
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>
Need to be Careful…

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

```sql
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>
Need to be Careful…

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

---

```sql
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>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>
Need to be Careful…

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

```sql
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 344 - 2017au
Everything in SELECT must be either a GROUP-BY attribute, or an aggregate.

**Need to be Careful...**

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

```
SELECT product, quantity 
FROM Purchase 
GROUP BY product
```

---

**Product** | **Max(quantity)** | **Price** | **Quantity**
---|---|---|---
Bagel | 20 | 3 | 20
Banana | 50 | 0.5 | 50

---

**Product** | **Quantity**
---|---
Bagel | 20
Banana | ??
Grouping and Aggregation

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
```

How is this query processed?
Grouping and Aggregation

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
```

Do these queries return the same number of rows? Why?

```
SELECT   product, Sum(quantity) AS TotalSales
FROM     Purchase
GROUP BY product
```
Grouping and Aggregation

Purchase(product, price, quantity)

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

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

Do these queries return the same number of rows? Why?

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

Empty groups are removed, hence first query may return fewer groups
Grouping and Aggregation

1. Compute the **FROM** and **WHERE** clauses.

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

3. Compute the **SELECT** clause: grouped attributes and aggregates.
### 1,2: From, Where

**SQL Query:**
```
SELECT product, SUM(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
```

**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>
### 3.4. Grouping, Select

#### SQL Query

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

#### Tabular Data

<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

```sql
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

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

S = may contain attributes \(a_1, \ldots, a_k\) and/or any aggregates but NO OTHER ATTRIBUTES

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

C2 = is any condition on aggregate expressions and on attributes \(a_1, \ldots, a_k\)
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
```
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
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
Mystery Query

What do they compute?

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

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

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

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
```

Lesson: DISTINCT is a special case of GROUP BY
Product(pid,pname,manufacturer)
Purchase(id,product_id,price,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 Purchase, price is in Product...
Aggregate + Join

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

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

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

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

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

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

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

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

<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.pid = 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

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

Count(*) is never 0
Including Empty Groups

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

Count(pid) is 0 when all pid’s in the group are NULL
Introduction to Data Management
CSE 344

Lecture 6: Nested Queries in SQL
Announcements

• HW2 is due tomorrow (Tuesday)

• HW3: soon you will receive an email from invites@microsoft.com: accept it

• Webquiz 2 due on Friday
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…)

“correlated subquery”
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
Compute the number of products made by each company

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

But are these really equivalent?

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

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

But are these really equivalent?

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

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

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

No! Different results if a company has no products
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
```
Product (pname, price, cid)
Company (cid, cname, city)

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
```

Try unnest this query!
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
```

Try unnest this query!

Side note: This is not a correlated subquery. (why?)
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
Product (\textit{pname}, \textit{price}, \textit{cid})
Company (\textit{cid}, \textit{cname}, \textit{city})

2. Subqueries in FROM

\begin{verbatim}
SELECT X.pname
FROM (SELECT *
    FROM Product AS Y
    WHERE price > 20) as X
WHERE X.price < 500
\end{verbatim}

A subquery whose result we called myTable

\begin{verbatim}
WITH myTable AS (SELECT * FROM Product AS Y WHERE price > 20)
SELECT X.pname
FROM myTable as X
WHERE X.price < 500
\end{verbatim}
3. Subqueries in WHERE

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

Find all companies that make some products with price < 200

Using `EXISTS`:

```
SELECT DISTINCT C.cname
FROM Company C
WHERE EXISTS (SELECT *
               FROM 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

Using **IN**

```
SELECT DISTINCT  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**:

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

*Existential quantifiers*
3. Subqueries in WHERE

Find all companies that make some products with price < 200

Now let’s unnest it:

```
SELECT DISTINCT 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

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

Existential quantifiers are easy! 😊
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
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

Product (pname, price, cid)
Company (cid, cname, city)
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 DISTINCT 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 DISTINCT 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 DISTINCT 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 \textit{EXISTS}:

\begin{verbatim}
SELECT DISTINCT C.cname
FROM Company C
WHERE NOT EXISTS (SELECT *
                   FROM Product P
                   WHERE P.cid = C.cid and P.price >= 200)
\end{verbatim}
3. Subqueries in WHERE

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

Using \textbf{ALL}:

\begin{verbatim}
SELECT DISTINCT   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**: 

```sql
SELECT DISTINCT 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 of the tuples

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 of the tuples.

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pname</strong></td>
<td><strong>cname</strong></td>
</tr>
<tr>
<td>Gizmo</td>
<td>Sunworks</td>
</tr>
<tr>
<td>Gadget</td>
<td>DB Inc.</td>
</tr>
<tr>
<td>Camera</td>
<td>Builder</td>
</tr>
</tbody>
</table>

| Company | |
|---------| |
| c001 | DB Inc. | Lyon |
| c002 | Sunworks | Bonn |
| c003 | Builder | Lodtz |
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 of the tuples

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

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>pname</td>
<td>cid</td>
</tr>
<tr>
<td>price</td>
<td>cname</td>
</tr>
<tr>
<td>cid</td>
<td>city</td>
</tr>
<tr>
<td>Gizmo</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>c004</td>
</tr>
<tr>
<td>Camera</td>
<td>c003</td>
</tr>
</tbody>
</table>

So far it looks monotone...
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 of the tuples.

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pname</strong></td>
<td><strong>cid</strong></td>
</tr>
<tr>
<td>Gizmo</td>
<td>19.99</td>
</tr>
<tr>
<td>Gadget</td>
<td>999.99</td>
</tr>
<tr>
<td>Camera</td>
<td>149.99</td>
</tr>
</tbody>
</table>

Q is not monotone!

- **Product**
  - Gizmo: 19.99, c001
  - Gadget: 999.99, c004
  - Camera: 149.99, c003
  - iPad: 499.99, c001

- **Company**
  - c002: Sunworks, Bonn
  - c001: DB Inc., Lyon
  - c003: Builder, Lodtz

Q is not monotone!

- **Product**
  - Gizmo: 19.99, c001
  - Gadget: 999.99, c004
  - Camera: 149.99, c003
  - iPad: 499.99, c001

- **Company**
  - Gizmo: Lyon
  - Camera: Lodtz
  - iPad: Lyon
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.

```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
```

```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)
```
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>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>999.99</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>

• **Consequence**: If a query is not monotonic, 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 that use aggregates in certain ways
  - $\text{sum(..)}$ and $\text{count(*)}$ are NOT monotone, because they do not satisfy set containment
  - $\text{select count(*) from R}$ is not monotone!
Introduction to Data Management

CSE 344

Lecture 7-8: SQL Wrap-up

Relational Algebra
Announcements

• You received invitation email to @cs
• You will be prompted to choose passwd
  – Problems with existing account?
  – In the worst case we will ask you to create a new @outlook account just for this class
• If OK, create the database server
  – Choose cheapest pricing tier!
• Remember: WQ2 due on Friday
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?
Author(login, name)
Wrote(login, url)

More Unnesting

Find authors who wrote ≥ 10 documents:
Find authors who wrote ≥ 10 documents:

Attempt 1: with nested queries

```
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 \( \geq 10 \) documents:

Attempt 1: with nested queries

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

Attempt 2: using GROUP BY and HAVING

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

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

But we need the witnesses, i.e., the products with max price
Finding Witnesses

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

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;
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);
```
Finding Witnesses

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

• Set-at-a-time algebra, which manipulates relations
• In SQL we say *what* we want
• In RA we can express *how* to get it
• Every DBMS implementations converts a SQL query to RA in order to execute it
• An RA expression is called a *query plan*
Basics

• Relations and attributes
• Functions that are applied to relations
  – Return relations
  – Can be composed together
  – Often displayed using a tree rather than linearly
  – Use Greek symbols: σ, π, δ, etc
Sets v.s. Bags

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

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 $\bigcup$, intersection $\bigcap$, difference $-$
- Selection $\sigma$
- Projection $\pi$
- Cartesian product $\times$, join $\bowtie$
- (Rename $\rho$)
- Duplicate elimination $\delta$
- Grouping and aggregation $\gamma$
- Sorting $\tau$

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

R1 ∪ 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 \( =, <, \leq, >, \geq, <> \) 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>

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

<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:

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

Different semantics over sets or bags! Why?
### 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>John</td>
<td>60000</td>
</tr>
<tr>
<td>4352342</td>
<td>John</td>
<td>20000</td>
</tr>
</tbody>
</table>

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

<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>
<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>

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

<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>

\[ \pi_{\text{zip}, \text{disease}}(\text{Patient}) \]

<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>

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

<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

\[ R_1 \times R_2 \]

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

### Employee

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>9999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
</tr>
</tbody>
</table>

### Dependent

<table>
<thead>
<tr>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>7777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

### Employee X 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)
  – \( \rho_{N, S}(\text{Employee}) \rightarrow \text{Answer}(N, S) \)
Natural Join

\[ R1 \Join R2 = \Pi_A(\sigma_\theta (R1 \times R2)) \]

- 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

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

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

\[
R \Join S =
\Pi_{ABC}(\sigma_{R.B=S.B}(R \times S))
\]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<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>

$P \bowtie V$

<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 $R \bowtie S$?

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

• Given $R(A, B), S(A, B)$, what is $R \bowtie S$?
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:

\[
P \bowtie P.zip = V.zip \text{ and } P.age \geq V.age - 1 \text{ and } P.age \leq V.age + 1
\]
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>

\[
P \bowtie_{P.age = V.age} V
\]

<table>
<thead>
<tr>
<th>P.age</th>
<th>P.zip</th>
<th>P.disease</th>
<th>V.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>

#### AnonJob 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>

#### Join Result

<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
\[ \pi_{\text{sname}}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{\text{psize}>10} \text{Part})) \]

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

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

\[\pi_{\text{sname}}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{\text{psize}>10}\ (\text{Part})))]\]
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 more 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

Answer

\[ T1, T2 = \text{temporary tables} \]

\[ \text{sales(product, city, quantity)} \]

\[ \begin{align*}
\text{SELECT} & \quad \text{city, sum(quantity)} \\
\text{FROM} & \quad \text{sales} \\
\text{GROUP BY} & \quad \text{city} \\
\text{HAVING} & \quad \text{count(*)} > 100 \\
\end{align*} \]
Typical Plan for a Query (1/2)

Answer

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

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

\[ \text{join condition} \]

\[ \text{R} \]

\[ \text{S} \]

SELECT-PROJECT-JOIN

Query

\[ \text{SELECT fields} \]

\[ \text{FROM R, S, ...} \]

\[ \text{WHERE condition} \]
Typical Plan for a Query (1/2)

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

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

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

join condition

\[ \gamma_{\text{fields, sum/count/min/max(fields)}} \]

\[ \text{SELECT fields FROM R, S, ... WHERE condition GROUP BY fields HAVING condition} \]
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)
```
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?

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

```
(SELECT Q.sno
 FROM Supplier Q
 WHERE Q.sstate = 'WA')
EXCEPT
(SELECT P.sno
 FROM Supply P
 WHERE P.price > 100)
```

```
Finally...
```

```
Suppliers
  └── P
        └── Price > 100
          └── Supply
```

CSE 344 - 2017au
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 translate SQL $\rightarrow$ 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 344

Lectures 9-10: Datalog
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:
  – LogicBlox (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
SQL Query vs Datalog
(which would you rather write?)
(any Java fans out there?)
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

Rules = queries

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

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

Find Movies made in 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'.
- 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(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

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)

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

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

\[ Q1(y) :\text{-} Movie(x,y,z), \; z='1940'. \]
\[ Q2(f, l) :\text{-} Actor(z,f,l), \; Casts(z,x), \; Movie(x,y,'1940'). \]
\[ Q3(f,l) :\text{-} Actor(z,f,l), \; Casts(z,x1), \; Movie(x1,y1,1910), \; Casts(z,x2), \; Movie(x2,y2,1940) \]

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

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

\[
Q(\text{args}) :- R1(\text{args}), R2(\text{args}), ....
\]

\[
Q(\text{args}) \leftarrow R1(\text{args}), R2(\text{args}), ....
\]

\[
Q(\text{args}) :- R1(\text{args}) \text{ AND } R2(\text{args}) ....
\]
Semantics of a Single Rule

• Meaning of a datalog rule = a logical statement!

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

• For all x, y, z: if (x,y,z) ∈ Movies and z = ‘1940’ then y is in Q1 (i.e. is part of the answer)

• ∀x ∀y ∀z [(Movie(x,y,z) and z='1940') ⇒ Q1(y)]

• Logically equivalent:

∀y [( ∃x ∃z Movie(x,y,z) and z='1940') ⇒ Q1(y)]

• Thus, head variables are called "existential variables"

• We want the **smallest** set Q1 with this property (why?)
Datalog program

- A datalog program consists of several rules
- Importantly, rules may be recursive!
- Usually there is one distinguished predicate that’s the output
- We will show an example first, then give the general semantics.
R encodes a graph

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

\[
\begin{align*}
R &= \\
1 & \rightarrow 2 \\
2 & \rightarrow 3 \\
2 & \rightarrow 1 \\
2 & \rightarrow 4 \\
1 & \rightarrow 4 \\
3 & \rightarrow 4 \\
4 & \rightarrow 5 \\
5 & \rightarrow 1 \\
\end{align*}
\]

Example

\[
\begin{align*}
T(x, y) & \leftarrow R(x, y) \\
T(x, y) & \leftarrow R(x, z), T(z, y)
\end{align*}
\]

What does it compute?
Example

R encodes a graph

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></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.

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

What does it compute?
Example

R encodes a graph

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>

First iteration:
T =

First rule generates this

Second rule generates nothing (because T is empty)

Initially:
T is empty.

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

What does it compute?
Example

R encodes a graph

\[ \begin{align*}
R &= \begin{pmatrix}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5
\end{pmatrix}
\]

Initially: T is empty.

First iteration: T =

Second iteration:

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

What does it compute?

- First rule generates this
- Second rule generates this

New facts

First rule generates this

Second rule generates this
Example

R encodes a graph

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

First iteration:
\[ T = \begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array} \]

Second iteration:
\[ T = \begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
1 & 1 \\
2 & 2 \\
1 & 3 \\
2 & 4 \\
1 & 5 \\
3 & 5 \\
\end{array} \]

Third iteration:
\[ T = \begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
1 & 1 \\
2 & 2 \\
1 & 3 \\
2 & 4 \\
1 & 5 \\
3 & 5 \\
\end{array} \]

What does it compute?

R encodes a graph

Initially:
\[ T = \begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array} \]

New fact

Both rules

First rule

Second rule
Example

R encodes a graph

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

Initially: T is empty.

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

First iteration:

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

Second iteration:

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

Third iteration:

Third iteration:

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

Fourth iteration:

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

No new facts.

DONE

What does it compute?

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

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

Fixpoint semantics

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

Repeat:

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

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? In what time?)
Datalog Semantics

Minimal model semantics:

• Return the IDB that
  1) For every rule,
    \( \forall \text{vars} \ [\text{Body}(\text{EDB}, \text{IDB}) \Rightarrow \text{Head}(\text{IDB})]\)
  2) Is the smallest IDB satisfying (1)

• Theorem: there exists a smallest IDB satisfying (1)
Datalog Semantics: Example

Fixpoint semantics:
• Start: $T_0 = \emptyset$; $t = 0$
  Repeat:
    $T_{t+1}(x,y) = R(x,y) \cup \Pi_{xy}(R(x,z) \bowtie T_t(z,y))$
    $t = t+1$
  Until $T_t = T_{t-1}$

Minimal model semantics: smallest $T$ s.t.
• $\forall x \forall y \left[ (R(x,y) \Rightarrow T(x,y)) \land \right.$
  $\forall x \forall y \forall z \left[ (R(x,z) \land T(z,y)) \Rightarrow T(x,y) \right]$
Datalog Semantics

• The fixpoint semantics tells us how to compute a datalog query

• The minimal model semantics is more declarative: only says what we get

• The two semantics are equivalent meaning: you get the same thing
Three Equivalent Programs

R encodes a graph

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

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)

Question: which terminates in fewest iterations?
Extensions

• Functional data model (LogicBlox)

• Aggregates, negation

• Stratified datalog
Functional Data Model

- Relational data model:
  Person(Alice, Smith) = true
  Person(Bob, Peters) = false

- Functional data model:
  Person[Alice, Smith] = can be a value v

- This is just a syntactic sugar for keyed relations (next slide)
Functional Data Model

- **Person**(fName,lName,friends)
  (note the key)

- Functional model:
  
<table>
<thead>
<tr>
<th>fName</th>
<th>lName</th>
<th>friends</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>Smith</td>
<td>22</td>
</tr>
<tr>
<td>Bob</td>
<td>Toth</td>
<td>5</td>
</tr>
<tr>
<td>Carol</td>
<td>Unger</td>
<td>9</td>
</tr>
</tbody>
</table>

Person[Alice,Smith]=22
Person[Bob,Toth]=5
Person[Carol,Unger]=9

CSE 344 - 2017au 267
Aggregates: use agg<<...>>>

General syntax in Logicblox:

\[ Q[\text{headVars}] \gets R1(\text{args1}), R2(\text{args2}), ... \]

Meaning (in SQL)

```sql
select headVars
from R1, R2, ...
where ...
```
Aggregates: use agg<<...>>>

General syntax in Logicblox:

```
Q[headVars] <- agg<<v = sum(w)>> R1(args1), R2(args2), ...
```

Meaning (in SQL)

```
select headVars, sum(w) as v
from R1, R2, ...
where ...
group by headVars
```
Aggregates: use agg<<...>>

General syntax in Logicblox:

Q[headVars] = v <- agg<<v = sum(w)>> R1(args1), R2(args2), ...

Meaning (in SQL)

```sql
select headVars, sum(w) as v
from R1, R2, ...
where ...
group by headVars
```
Example

For each person, compute the total number of descendants

/* We use Logicblox syntax (as in the homework) */
/* for each person, compute his/her descendants */
Example

For each person, compute the total number of descendants

/* We use Logicblox syntax (as in the homework) */
/* for each person, compute his/her descendants */
D(x,y) <- ParentChild(x,y).
Example

For each person, compute the total number of descendants

/* We use Logicblox syntax (as in the homework) */
/* for each person, compute his/her descendants */
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
Example

For each person, compute the total number of descendants

/* We use Logicblox syntax (as in the homework) */
/* 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 */
Example

For each person, compute the total number of descendants

/* We use Logicblox syntax (as in the homework) */
/* 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 */
N[x] = m  <- agg<<m = count()>> D(x,y).
Example

For each person, compute the total number of descendants

/* We use Logicblox syntax (as in the homework) */
/* 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 */
N[x] = m <- agg<<m = count()>> D(x,y).
/* Find the number of descendants of Alice */
Example

For each person, compute the total number of descendants

/* We use Logicblox syntax (as in the homework) */
/* 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 */
N[x] = m <- agg<<m = count()>> D(x,y).
/* Find the number of descendants of Alice */
Q(d) <- N[“Alice”]=d.
Negation: use “!”

Find all descendants of Alice, who are not descendants of Bob

/* 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("Alice",x) !D("Bob",x).
Safe Datalog Rules

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

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

\[ U2(x) :- \text{ParentChild(“Alice”,x), } \neg \text{ParentChild(x,y)} \]
Safe Datalog Rules

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)

Holds for every y other than “Bob”
U1 = infinite!
Here are \textit{unsafe} datalog rules. What’s “unsafe” about them?

\begin{align*}
U1(x,y) & :\text{ ParentChild(“Alice”,} x\text{), } y \neq “Bob” \\
U2(x) & :\text{ ParentChild(“Alice”,} x\text{), parentChild}(x,y)
\end{align*}

Holds for every $y$ other than “Bob”

U1 = infinite!

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

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

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

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

A datalog rule is *safe* if every variable appears in some positive relational atom.
Stratified Datalog

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

\[
A() \leftarrow \neg B(). \\
B() \leftarrow \neg A().
\]

- A datalog program is **stratified** if it can be partitioned into strata s.t., for all n, only IDB predicates defined in strata 1, 2, ..., n may appear under \( \neg \) or agg in stratum n+1.
- LogicBlox (and others) accepts only stratified datalog.
Stratified Datalog

Stratum 1

D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).

Stratum 2

N[x] = m <- agg<<m = count()> D(x,y).
Q(d) <- N[“Alice”]=d.

May use D in an agg because was defined in previous stratum
Stratified Datalog

\[
\begin{align*}
D(x,y) & \leftarrow \text{ParentChild}(x,y). \\
D(x,z) & \leftarrow D(x,y), \text{ParentChild}(y,z).
\end{align*}
\]

\[
\begin{align*}
N[x] = m & \leftarrow \text{agg} \{ m = \text{count()} \} \triangleright D(x,y).
Q(d) & \leftarrow N["Alice"] = d.
\end{align*}
\]

- May use \( D \) in an \text{agg} because it was defined in the previous stratum.
- May use \( \neg D \).
Stratified Datalog

Stratum 1

D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).

Stratum 2

N[x] = m <- agg<<m = count()>> D(x,y).
Q(d) <- N[“Alice”]=d.

Stratum 1

D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).

Stratum 2

Q(x) <- D(“Alice”,x), !D(“Bob”,x).

Non-stratified

A() <- !B().
B() <- !A().

May use D in an agg because 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
Datalog v.s. RA (and SQL)

• “Pure” datalog has recursion, but no negation, aggregates: all queries are monotone; impractical

• Datalog without recursion, plus negation and aggregates expresses the same queries as RA: next slides
RA to Datalog by Examples

Union:
R(A,B,C) ∪ S(D,E,F)

U(x,y,z) :- R(x,y,z)
U(x,y,z) :- S(x,y,z)
RA to Datalog by Examples

Intersection:
\[ R(A, B, C) \cap S(D, E, F) \]

\[ I(x, y, z) :- R(x, y, z), S(x, y, z) \]
RA to Datalog by Examples

Selection: \( \sigma_{x>100 \text{ and } y='foo'} (R) \)
L(x,y,z) :- R(x,y,z), x > 100, y='foo'

Selection: \( \sigma_{x>100 \text{ or } y='foo'} (R) \)
L(x,y,z) :- R(x,y,z), x > 100
L(x,y,z) :- R(x,y,z), y='foo'
RA to Datalog by Examples

Equi-join: \( R \bowtie_{R.A=S.D \text{ and } R.B=S.E} S \)

\[ J(x,y,z,q) :- R(x,y,z), S(x,y,q) \]
RA to Datalog by Examples

Projection: $\Pi_A(R)$

$P(x) \leftarrow R(x,y,z)$
To express difference, we add negation

\[ \text{R} - \text{S} \]

\[ \text{D}(x,y,z) \leftarrow \text{R}(x,y,z), \text{NOT S}(x,y,z) \]
Examples

Translate: $\Pi_A(\sigma_{B=3} (R) )$

$A(a) :- R(a,3,\_)$

Underscore used to denote an "anonymous variable"
Each such variable is unique
Examples

Translate: \[ \Pi_A (\sigma_{B=3} (R) \bowtie_{R.A=S.D} \sigma_{E=5} (S) ) \]

\[
A(a) :- R(a,3,\_), S(a,5,\_)
\]

These are different "\_"s
More Examples w/o Recursion

Find Joe's friends, and Joe's friends of friends.

\[
A(x) \leftarrow \text{Friend('Joe', x)} \\
A(x) \leftarrow \text{Friend('Joe', z), Friend(z, x)}
\]
More Examples w/o Recursion

Find all of Joe's friends who do not have any friends except for Joe:

```
JoeFriends(x) :- Friend('Joe',x)
NonAns(x) :- JoeFriends(x), Friend(x,y), y != 'Joe'
A(x) :- JoeFriends(x), NOT NonAns(x)
```
More Examples w/o Recursion

Find all people such that all their enemies' enemies are their friends

• Q: if someone doesn't have any enemies nor friends, do we want them in the answer?
• A: Yes!

\[
\text{Everyone}(x) :- \text{Friend}(x,y) \\
\text{Everyone}(x) :- \text{Friend}(y,x) \\
\text{Everyone}(x) :- \text{Enemy}(x,y) \\
\text{Everyone}(x) :- \text{Enemy}(y,x) \\
\text{NonAns}(x) :- \text{Enemy}(x,y), \text{Enemy}(y,z), \text{NOT} \ \text{Friend}(x,z) \\
\text{A}(x) :- \text{Everyone}(x), \text{NOT} \ \text{NonAns}(x)
\]
More Examples w/o Recursion

Find all persons x that have a friend all of whose enemies are x's enemies.

Everyone(x) :- Friend(x,y)
NonAns(x) :- Friend(x,y) Enemy(y,z), NOT Enemy(x,z)
A(x) :- Everyone(x), NOT NonAns(x)

Friend(name1, name2)
Enemy(name1, name2)
More Examples w/ Recursion

- Two people are in the same generation if they are siblings, or if they have parents in the same generation.

- Find all persons in the same generation with Alice.
More Examples w/ Recursion

- Find all persons in the same generation with Alice
- Let’s compute $SG(x,y) = \text{“}x,y \text{ are in the same generation}$$\text{“}$

```
SG(x,y) :- ParentChild(p,x), ParentChild(p,y)
SG(x,y) :- ParentChild(p,x), ParentChild(q,y), SG(p,q)
Answer(x) :- SG("Alice", x)
```
Datalog Summary

• EDB (base relations) and IDB (derived relations)
• Datalog program = set of rules
• Datalog is recursive

• Some reminders about semantics:
  – Multiple atoms in a rule mean join (or intersection)
  – Variables with the same name are join variables
  – Multiple rules with same head mean union
Datalog and SQL

• Stratified data (w/ recursion, w/o +,*,...,): expresses precisely* queries in PTIME
  – Cannot find a Hamiltonian cycle (why?)
• SQL has also been extended to express recursive queries:
  – Use a recursive “with” clause, also CTE (Common Table Expression)
  – Often with bizarre restrictions...
  – ... Just use datalog

* need to use the < predicate