Introduction to Data Management
CSE 344

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

Unit 2

Unit 3

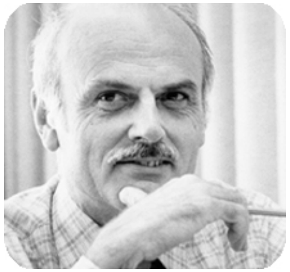
3 Elements of Data Models

- 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

columns / attributes / fields

- Data is a collection of relations / tables:

rows / tuples / records

cname	country	no_employees	for_profit
GizmoWorks	USA	20000	True
Canon	Japan	50000	True
Hitachi	Japan	30000	True
HappyCam	Canada	500	False

- 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

Key

<u>cname</u>	country	no_employees	for_profit
GizmoWorks	USA	20000	True
Canon	Japan	50000	True
Hitachi	Japan	30000	True
HappyCam	Canada	500	False

Keys

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

Key

Not a key

<u>cname</u>	country	no_employees	for_profit
GizmoWorks	USA	20000	True
Canon	Japan	50000	True
Hitachi	Japan	30000	True
HappyCam	Canada	500	False

Keys

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

Key

Not a key

Is this a key?

cname	country	no_employees	for_profit
GizmoWorks	USA	20000	True
Canon	Japan	50000	True
Hitachi	Japan	30000	True
HappyCam	Canada	500	False

Keys

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

Key

Not a key

Is this a key?

No: future updates to the database may create duplicate no_employees

<u>cname</u>	country	no_employees	for_profit
GizmoWorks	USA	20000	True
Canon	Japan	50000	True
Hitachi	Japan	30000	True
HappyCam	Canada	500	False

Multi-attribute Key

Key = fName,lName
(what does this mean?)

fName	lName	Income	Department
Alice	Smith	20000	Testing
Alice	Thompson	50000	Testing
Bob	Thompson	30000	SW
Carol	Smith	50000	Testing

Multiple Keys

Key

Another key

<u>SSN</u>	fName	lName	Income	Department
111-22-3333	Alice	Smith	20000	Testing
222-33-4444	Alice	Thompson	50000	Testing
333-44-5555	Bob	Thompson	30000	SW
444-55-6666	Carol	Smith	50000	Testing

We can choose one key and designate it as <u>*primary key*</u>

E.g.: primary key = SSN

Foreign Key

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

Foreign key to Country.name

Company

<u>cname</u>	country	no_employees	for_profit
Canon	Japan	50000	Y
Hitachi	Japan	30000	Y

Country

<u>name</u>	population
USA	320M
Japan	127M

Keys: Summary

- Key = columns that uniquely identify tuple
  - Usually we underline
  - A relation can have many keys, but only one can be chosen as *primary key*
- Foreign key:
  - Attribute(s) whose value is a key of a record in some other relation
  - Foreign keys are sometimes called *semantic pointer*

Query Language

- SQL
  - **S**tructured **Q**uery **L**anguage
  - 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?

cname	country	no_employees	for_profit
GizmoWorks	USA	20000	True
Canon	Japan	50000	True
Hitachi	Japan	30000	True
HappyCam	Canada	500	False

Table Implementation

- How would you implement this?

cname	country	no_employees	for_profit
GizmoWorks	USA	20000	True
Canon	Japan	50000	True
Hitachi	Japan	30000	True
HappyCam	Canada	500	False

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?

cname	country	no_employees	for_profit
GizmoWorks	USA	20000	True
Canon	Japan	50000	True
Hitachi	Japan	30000	True
HappyCam	Canada	500	False

Column major: as one array per attribute

GizmoWorks	Canon	Hitachi	HappyCam

USA	Japan	Japan	Canada

20000	50000	30000	500

True	True	True	False

Table Implementation

- How would you implement this?

cname	country	no_employees	for_profit
GizmoWorks	USA	20000	True
Canon	Japan	50000	True
Hitachi	Japan	30000	True
HappyCam	Canada	500	False

Physical data independence

The logical definition of the data remains unchanged, even when we make changes to the actual implementation

First Normal Form

cname	country	no_employees	for_profit
Canon	Japan	50000	Y
Hitachi	Japan	30000	Y

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

First Normal Form

cname	country	no_employees	for_profit
Canon	Japan	50000	Y
Hitachi	Japan	30000	Y

- 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

cname	country	no_employees	for_profit
Canon	Japan	50000	Y
Hitachi	Japan	30000	Y

- 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>
<tr><th>cname</th><th>country</th><th>no_employees</th><th>for_profit</th><th>products</th></tr>
<tr><td>Canon</td><td>Japan</td><td>50000</td><td>Y</td><td><table><tr><th>pname</th><th>price</th><th>category</th></tr><tr><td>SingleTouch</td><td>149.99</td><td>Photography</td></tr><tr><td>Gadget</td><td>200</td><td>Toy</td></tr></table></td></tr>
<tr><td>Hitachi</td><td>Japan</td><td>30000</td><td>Y</td><td><table><tr><th>pname</th><th>price</th><th>category</th></tr><tr><td>AC</td><td>300</td><td>Appliance</td></tr></table></td></tr>
</table>

First Normal Form

cname	country	no_employees	for_profit
Canon	Japan	50000	Y
Hitachi	Japan	30000	Y

- 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:

Non-1NF!

<table>
<tr><th>cname</th><th>country</th><th>no_employees</th><th>for_profit</th><th>products</th></tr>
<tr><td>Canon</td><td>Japan</td><td>50000</td><td>Y</td><td><table><tr><th>pname</th><th>price</th><th>category</th></tr><tr><td>SingleTouch</td><td>149.99</td><td>Photography</td></tr><tr><td>Gadget</td><td>200</td><td>Toy</td></tr></table></td></tr>
<tr><td>Hitachi</td><td>Japan</td><td>30000</td><td>Y</td><td><table><tr><th>pname</th><th>price</th><th>category</th></tr><tr><td>AC</td><td>300</td><td>Appliance</td></tr></table></td></tr>
</table>

First Normal Form

Now it's in 1NF

Company

cname	country	no_employees	for_profit
Canon	Japan	50000	Y
Hitachi	Japan	30000	Y

Products

pname	price	category	manufacturer
SingleTouch	149.99	Photography	Canon
AC	300	Appliance	Hitachi
Gadget	200	Toy	Canon

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

- **S**tructured **Q**uery **L**anguage
- 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

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

Joins in SQL

pname	price	category	manufacturer
MultiTouch	199.99	gadget	Canon
SingleTouch	49.99	photography	Canon
Gizom	50	gadget	GizmoWorks
SuperGizmo	250.00	gadget	GizmoWorks

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

Retrieve all Japanese products that cost < $150

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

Joins in SQL

pname	price	category	manufacturer
MultiTouch	199.99	gadget	Canon
SingleTouch	49.99	photography	Canon
Gizom	50	gadget	GizmoWorks
SuperGizmo	250.00	gadget	GizmoWorks

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

Retrieve all Japanese products that cost < $150

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

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

Joins in SQL

pname	price	category	manufacturer
MultiTouch	199.99	gadget	Canon
SingleTouch	49.99	photography	Canon
Gizom	50	gadget	GizmoWorks
SuperGizmo	250.00	gadget	GizmoWorks

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

Retrieve all Japanese products that cost < $150

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

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

Joins in SQL

pname	price	category	manufacturer
MultiTouch	199.99	gadget	Canon
SingleTouch	49.99	photography	Canon
Gizom	50	gadget	GizmoWorks
SuperGizmo	250.00	gadget	GizmoWorks

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

Retrieve all USA companies
that manufacture "gadget" products

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

Joins in SQL

pname	price	category	manufacturer
MultiTouch	199.99	gadget	Canon
SingleTouch	49.99	photography	Canon
Gizom	50	gadget	GizmoWorks
SuperGizmo	250.00	gadget	GizmoWorks

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

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

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

Inner Join

Employee

id	name
1	Joe
2	Jack
3	Jill

Sales

employeeID	productID
1	344
1	355
2	544

Retrieve employees and their sales

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

Inner Join

Employee

id	name
1	Joe
2	Jack
3	Jill

Sales

employeeID	productID
1	344
1	355
2	544

Retrieve employees and their sales

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

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

Inner Join

Employee

id	name
1	Joe
2	Jack
3	Jill

Sales

employeeID	productID
1	344
1	355
2	544

Retrieve employees and their sales

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

id	name	empolyeeID	productID
1	Joe	1	344
1	Joe	1	355
2	Jack	2	544

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

Inner Join

Employee

id	name
1	Joe
2	Jack
3	Jill

Sales

employeeID	productID
1	344
1	355
2	544

Retrieve employees and their sales

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

Jill is missing

id	name	empolyeeID	productID
1	Joe	1	344
1	Joe	1	355
2	Jack	2	544

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

Inner Join

Employee

id	name
1	Joe
2	Jack
3	Jill

Sales

employeeID	productID
1	344
1	355
2	544

Retrieve employees and their sales

Alternative syntax

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

Jill is missing

id	name	empolyeeID	productID
1	Joe	1	344
1	Joe	1	355
2	Jack	2	544

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

Outer Join

Employee

id	name
1	Joe
2	Jack
3	Jill

Sales

employeeID	productID
1	344
1	355
2	544

Retrieve employees and their sales

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

Jill is present

id	name	empolyeeID	productID
1	Joe	1	344
1	Joe	1	355
2	Jack	2	544
3	Jill	NULL	NULL

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

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

Product

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
Camera	Photo	Hitachi
OneClick	Photo	Hitachi

Company

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

(Inner) joins

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

Product

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
Camera	Photo	Hitachi
OneClick	Photo	Hitachi

Company

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

(Inner) joins

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

Product

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
Camera	Photo	Hitachi
OneClick	Photo	Hitachi

Company

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

(Inner) joins

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

Product

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
Camera	Photo	Hitachi
OneClick	Photo	Hitachi

Company

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

pname	category	manufacturer	cname	country
Gizmo	gadget	GizmoWorks	GizmoWorks	USA

(Inner) joins

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

Product

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
Camera	Photo	Hitachi
OneClick	Photo	Hitachi

Company

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

(Inner) joins

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

Product

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
Camera	Photo	Hitachi
OneClick	Photo	Hitachi

Company

cname	country
GizmoWorks	USA
Canon	Japan
Hitachi	Japan

(Inner) joins

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

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

(Inner) Joins

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

```
for x1 in R1:
   for x2 in R2:

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

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

Another example

```
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

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

Need to include Product twice!

Self-Joins and Tuple Variables

- Find USA companies that manufacture both products in the ‘gadgets’ and ‘photo’ category
- Joining Product with Company is insufficient: need to join Product, with Product, and with Company

- When a relation occurs twice in the FROM clause we call it a self-join; in that case we must use tuple variables (why?)

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;
```

Product

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
SingleTouch	photo	Hitachi
MultiTouch	Photo	GizmoWorks

Company

cname	country
GizmoWorks	USA
Hitachi	Japan

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;
```

Product

x

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
SingleTouch	photo	Hitachi
MultiTouch	Photo	GizmoWorks

Company

cname	country
GizmoWorks	USA
Hitachi	Japan

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;
```

Product

x

y

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
SingleTouch	photo	Hitachi
MultiTouch	Photo	GizmoWorks

Company

cname	country
GizmoWorks	USA
Hitachi	Japan

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;
```

Product

x

y

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
SingleTouch	photo	Hitachi
MultiTouch	Photo	GizmoWorks

Company

z

cname	country
GizmoWorks	USA
Hitachi	Japan

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;
```

Product

x

y

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
SingleTouch	photo	Hitachi
MultiTouch	Photo	GizmoWorks

Company

z

cname	country
GizmoWorks	USA
Hitachi	Japan

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;
```

Product

	pname	category	manufacturer
x	Gizmo	gadget	GizmoWorks
y	SingleTouch	photo	Hitachi
	MultiTouch	Photo	GizmoWorks

Company

cname	country	
GizmoWorks	USA	z
Hitachi	Japan	

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;
```

Product

	pname	category	manufacturer
x	Gizmo	gadget	GizmoWorks
y	SingleTouch	photo	Hitachi
	MultiTouch	Photo	GizmoWorks

Company

cname	country
GizmoWorks	USA
Hitachi	Japan

z

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;
```

Product

	pname	category	manufacturer
x	Gizmo	gadget	GizmoWorks
	SingleTouch	photo	Hitachi
y	MultiTouch	Photo	GizmoWorks

Company z

cname	country
GizmoWorks	USA
Hitachi	Japan

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;
```

Product

	pname	category	manufacturer
x	Gizmo	gadget	GizmoWorks
	SingleTouch	photo	Hitachi
y	MultiTouch	Photo	GizmoWorks

Company (z)

cname	country
GizmoWorks	USA
Hitachi	Japan

x.pname	x.category	x.manufacturer	y.pname	y.category	y.manufacturer	z.cname	z.country
Gizmo	gadget	GizmoWorks	MultiTouch	Photo	GizmoWorks	GizmoWorks	USA

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;
```

Product

x

pname	category	manufacturer
Gizmo	gadget	GizmoWorks
SingleTouch	photo	Hitachi
MultiTouch	Photo	GizmoWorks

y

Company

z

cname	country
GizmoWorks	USA
Hitachi	Japan

x.pname	x.category	x.manufacturer	y.pname	y.category	y.manufacturer	z.cname	z.country
Gizmo	gadget	GizmoWorks	MultiTouch	Photo	GizmoWorks	GizmoWorks	USA

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
```

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz
Camera	Ritz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
Phone	Foo

Output

Name	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL
NULL	Foo

Outer Joins

```
tableA (LEFT/RIGHT/FULL) OUTER JOIN tableB ON p
```

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

- In all cases:
  - Patch tuples without matches using `NULL`

Loading Data into SQLite

```
>sqlite3 lecture04

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

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

Specify a filename where the database will be stored

Other DBMSs have other ways of importing data

Comment about SQLite

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

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

```
update Purchase
  set price = null
  where price = 'null'
```

Simple Aggregations

Five basic aggregate operations in SQL

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

Except count, all aggregations apply to a single attribute

Aggregates and NULL Values

Null values are not used in aggregates

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

Let's try the following

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

select sum(quantity) from Purchase

select count(*)
from Purchase
where quantity is not null;
```

Counting Duplicates

`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

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

→

Product	TotalSales
Bagel	40
Banana	20

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

Other Examples

Compare these two queries:

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

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

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

What does it return?

Need to be Careful…

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

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

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?
```

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

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?
```

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

Product	Max(quantity)
Bagel	20
Banana	50

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?
```

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

Product	Max(quantity)
Bagel	20
Banana	50

Product	Quantity
Bagel	20
Banana	??

Need to be Careful…

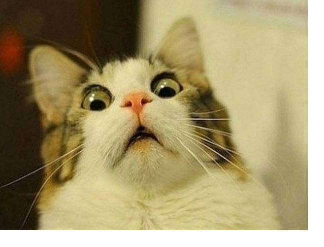
```sql
SELECT product,
       max(quantity)
FROM   Purchase
GROUP BY product
```

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

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

Product	Max(quantity)
Bagel	20
Banana	50

Product	Quantity
Bagel	20
Banana	??



Everything in SELECT must be either a GROUP-BY attribute, or an aggregate

Need to be Careful…

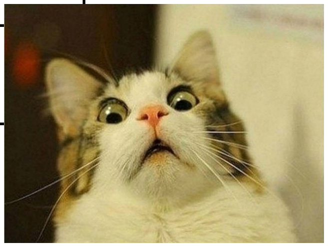
```
SELECT product,
       max(quantity)
FROM   Purchase
GROUP BY product
```

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

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

Product	Max(quantity)
Bagel	20
Banana	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.

```
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
```

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.

FWGS ™

1,2: From, Where

FWGS

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	~~0.5~~	~~50~~
Banana	2	10
Banana	4	10

WHERE price > 1

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

3,4. Grouping, Select

FWGS

Product	Price	Quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

Product	TotalSales
Bagel	40
Banana	20

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

Purchase(pid, product, price, quantity, month)

Ordering Results

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

FWGOS TM

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

```
Purchase(pid, product, price, quantity, month)
```

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

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

Why ?

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

```
SELECT    S
FROM      R1,…,Rn
WHERE     C1
GROUP BY  a1,…,ak
HAVING    C2
```

FWGHOS

Evaluation steps:

1. Evaluate FROM-WHERE using Nested Loop Semantics
2. Group by the attributes $a_1,\ldots,a_k$
3. Apply condition C2 to each group (may have aggregates)
4. Compute aggregates in S and return the result

`Purchase(pid, product, price, quantity, 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”

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
```

`Purchase(pid, product, price, quantity, 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"

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

Purchase(pid, product, price, quantity, 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”

```
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

```
Purchase(pid, product, price, quantity, 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
```

```
Purchase(pid, product, price, quantity, 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

```
Product(pid,pname,manufacturer)
Purchase(id,product_id,price,month)
```

Aggregate + Join

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

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

```
Product(pid,pname,manufacturer)
Purchase(id,product_id,price,month)
```

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
```

manu facturer	...	price	...
Hitachi		150	
Canon		300	
Hitachi		180	

```
Product(pid,pname,manufacturer)
Purchase(id,product_id,price,month)
```

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
```

manu facturer	...	price	...
Hitachi		150	
Canon		300	
Hitachi		180	

```
-- 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
```

manu facturer	count(*)
Hitachi	2
Canon	1
...	

```
Product(pid,pname,manufacturer)
Purchase(id,product_id,price,month)
```

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
```

manu facturer	month	count(*)
Hitachi	Jan	2
Hitachi	Feb	1
Canon	Jan	3
...		

FWGHOS

Including Empty Groups

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

Count(*) is never 0

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

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
```

"correlated subquery"

What happens if the subquery returns more than one city?

We get a runtime error

(and SQLite simply ignores the extra values…)

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

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
```

=

```
SELECT X.pname, Y.city
FROM   Product X, Company Y
WHERE  X.cid=Y.cid
```

We have "unnested" the query

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

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
```

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

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
```

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

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
```

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

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
```

No! Different results if a company has no products

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

```
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
```

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 !

```
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
```

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

Try unnest this query !

2. Subqueries in FROM

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

- Option 1: use a subquery in the FROM clause
- Option 2: use the WITH clause

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

2. Subqueries in FROM

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

=

A subquery whose result we called myTable

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

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

3. Subqueries in WHERE

Find all companies that make some products with price < 200

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

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Existential quantifiers

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

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Existential quantifiers

Using EXISTS:

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

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

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Existential quantifiers

Using IN

```
SELECT DISTINCT  C.cname
FROM  Company C
WHERE C.cid IN (SELECT P.cid
               FROM Product P
               WHERE P.price < 200)
```

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

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Existential quantifiers

Using ANY:

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

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

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Existential quantifiers

Using ANY:

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

Not supported in sqlite

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

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Existential quantifiers

Now let's unnest it:

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

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

3. Subqueries in WHERE

Find all companies that make some products with price < 200

Existential quantifiers

Now let's unnest it:

```
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

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

3. Subqueries in WHERE

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

same as:

Find all companies that make only products with price < 200

Universal quantifiers

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

3. Subqueries in WHERE

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

same as:

Find all companies that make only products with price < 200

Universal quantifiers

Universal quantifiers are hard! ☹

```
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

```
SELECT DISTINCT  C.cname
FROM   Company C
WHERE  C.cid IN (SELECT P.cid
                 FROM Product P
                 WHERE P.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

```
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

```
SELECT DISTINCT  C.cname
FROM   Company C
WHERE  C.cid NOT IN (SELECT P.cid
                     FROM Product P
                     WHERE P.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

Universal quantifiers

Using EXISTS:

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

Universal quantifiers

Using ALL:

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

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

3. Subqueries in WHERE

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

Universal quantifiers

Using ALL:

```
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*

```
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

```
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

Product

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c004
Camera	149.99	c003

Company

cid	cname	city
c002	Sunworks	Bonn
c001	DB Inc.	Lyon
c003	Builder	Lodtz

Q ⇒

pname	city
Gizmo	Lyon
Camera	Lodtz

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

Product

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c004
Camera	149.99	c003

Company

cid	cname	city
c002	Sunworks	Bonn
c001	DB Inc.	Lyon
c003	Builder	Lodtz

Q

pname	city
Gizmo	Lyon
Camera	Lodtz

Product

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c004
Camera	149.99	c003
iPad	499.99	c001

Company

cid	cname	city
c002	Sunworks	Bonn
c001	DB Inc.	Lyon
c003	Builder	Lodtz

Q

pname	city
Gizmo	Lyon
Camera	Lodtz
iPad	Lyon

So far it looks monotone...

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

Product

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c004
Camera	149.99	c003

Company

cid	cname	city
c002	Sunworks	Bonn
c001	DB Inc.	Lyon
c003	Builder	Lodtz

Q ⇒

pname	city
Gizmo	Lyon
Camera	Lodtz

Product

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c004
Camera	149.99	c003
iPad	499.99	c001

Company

cid	cname	city
c002	Sunworks	Bonn
c001	DB Inc.	Lyon
c003	Builder	Lodtz
c004	Crafter	Lodtz

Q ⇒

Q is not monotone!

pname	city
Gizmo	Lodtz
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

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

```
for x1 in R1 do
  for x2 in R2 do
    …
      for xn in Rn do
        if Conditions
          output (a1,…,ak)
```

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

Monotone Queries

- The query:

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

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

Monotone Queries

- The query:

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

pname	price	cid
Gizmo	19.99	c001

cid	cname	city
c001	Sunworks	Bonn

cname
Sunworks

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

Monotone Queries

- The query:

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

pname	price	cid
Gizmo	19.99	c001

cid	cname	city
c001	Sunworks	Bonn

cname
Sunworks

pname	price	cid
Gizmo	19.99	c001
Gadget	999.99	c001

cid	cname	city
c001	Sunworks	Bonn

cname

- 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
  - `sum(..)` and `count(*)` are NOT monotone, because they do not satisfy set containment
  - `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:

```
Author(login,name)
Wrote(login,url)
```

More Unnesting

Find authors who wrote ≥ 10 documents:

Attempt 1: with nested queries

This is SQL by a novice

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

Author(login,name)
Wrote(login,url)

More Unnesting

Find authors who wrote ≥ 10 documents:

Attempt 1: with nested queries

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
```

This is SQL by an expert

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

Finding Witnesses

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

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

Finding Witnesses

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

Finding the maximum price is easy…

```
SELECT x.city, max(y.price)
FROM   Company x, Product y
WHERE  x.cid = y.cid
GROUP BY x.city;
```

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

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

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;
```

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

Finding Witnesses

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

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

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

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);
```

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

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}, { }, . . .
- Bags: {a, a, b, c}, {b, b, b, b, b}, . . .

Relational Algebra has two flavors:

- Set semantics = standard Relational Algebra
- Bag semantics = extended Relational Algebra

DB systems implement bag semantics (Why?)

Relational Algebra Operators

- Union ∪, ~~intersection ∩~~, difference -
- Selection σ
- Projection π
- Cartesian product X, join ⋈
- (Rename ρ)
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ

RA

Extended RA

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

$$R1 \cap R2 = R1 - (R1 - R2)$$

- Derived using join

$$R1 \cap R2 = R1 \bowtie R2$$

Selection

- Returns all tuples which satisfy a condition

$$\sigma_c(R)$$

- Examples
  - $\sigma_{\text{Salary > 40000}}$ (Employee)
  - $\sigma_{\text{name = “Smith”}}$ (Employee)
- The condition c can be =, <, <=, >, >=, <> combined with AND, OR, NOT

Employee

SSN	Name	Salary
1234545	John	20000
5423341	Smith	60000
4352342	Fred	50000

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

SSN	Name	Salary
5423341	Smith	60000
4352342	Fred	50000

Projection

- Eliminates columns

$$\pi_{A1,\ldots,An}(R)$$

- Example: project social-security number and names:
  - $\pi_{SSN,\ Name}$ (Employee) → Answer(SSN, Name)

Different semantics over sets or bags! Why?

Employee

SSN	Name	Salary
1234545	John	20000
5423341	John	60000
4352342	John	20000

$\pi_{Name,Salary}$ (Employee)

Name	Salary
John	20000
John	60000
John	20000

Bag semantics

Name	Salary
John	20000
John	60000

Set semantics

Which is more efficient?

Composing RA Operators

Patient

no	name	zip	disease
1	p1	98125	flu
2	p2	98125	heart
3	p3	98120	lung
4	p4	98120	heart

$\pi_{zip,disease}$(Patient)

zip	disease
98125	flu
98125	heart
98120	lung
98120	heart

$\sigma_{disease='heart'}$(Patient)

no	name	zip	disease
2	p2	98125	heart
4	p4	98120	heart

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

zip	disease
98125	heart
98120	heart

Cartesian Product

- Each tuple in R1 with each tuple in R2

$$R1 \times R2$$

- Rare in practice; mainly used to express joins

Cross-Product Example

Employee

Name	SSN
John	999999999
Tony	777777777

Dependent

EmpSSN	DepName
999999999	Emily
777777777	Joe

Employee X Dependent

Name	SSN	EmpSSN	DepName
John	999999999	999999999	Emily
John	999999999	777777777	Joe
Tony	777777777	999999999	Emily
Tony	777777777	777777777	Joe

Renaming

- Changes the schema, not the instance

$$\rho_{B1,\ldots,Bn}(R)$$

- Example:
  - Given Employee(Name, SSN)
  - $\rho_{N,S}$(Employee) → Answer(N, S)

Natural Join

$R1 \bowtie R2$

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

- Where:
  - Selection σ_θ checks equality of all common attributes (i.e., attributes with same names)
  - Projection Π_A eliminates duplicate common attributes

Natural Join Example

R

A	B
X	Y
X	Z
Y	Z
Z	V

S

B	C
Z	U
V	W
Z	V

R ⋈ S =

$\Pi_{ABC}(\sigma_{R.B=S.B}(R \times S))$

A	B	C
X	Z	U
X	Z	V
Y	Z	U
Y	Z	V
Z	V	W

Natural Join Example 2

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
Alice	54	98125
Bob	20	98120

$P \bowtie V$

age	zip	disease	name
54	98125	heart	Alice
20	98120	flu	Bob

Natural Join

- Given schemas R(A, B, C, D), S(A, C, E), what is the schema of R ⋈ S ?

- Given R(A, B, C), S(D, E), what is R ⋈ S?

- Given R(A, B), S(A, B), what is R ⋈ S?

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

Theta Join

- A join that involves a predicate

$$R1 \bowtie_{\theta} R2 \quad = \quad \sigma_{\theta}\,(R1 \times R2)$$

- Here θ can be any condition
- No projection in this case!
- For our voters/patients example:

$$P \bowtie_{P.zip\ =\ V.zip\ \text{and}\ P.age\ >=\ V.age\ -1\ \text{and}\ P.age\ <=\ V.age\ +1} V$$

Equijoin

- A theta join where θ is an equality predicate

$$R1 \bowtie_{\theta} R2 \quad = \sigma_{\theta} (R1 \times R2)$$

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

Equijoin Example

AnonPatient P

age	zip	disease
54	98125	heart
20	98120	flu

Voters V

name	age	zip
p1	54	98125
p2	20	98120

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

P.age	P.zip	P.disease	V.name	V.age	V.zip
54	98125	heart	p1	54	98125
20	98120	flu	p2	20	98120

Join Summary

- **Theta-join**: $R \bowtie_\theta S = \sigma_\theta (R \times S)$
  - Join of R and S with a join condition θ
  - Cross-product followed by selection θ
  - No projection
- **Equijoin**: $R \bowtie_\theta S = \sigma_\theta (R \times S)$
  - Join condition θ 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 π_A drops all redundant attributes

So Which Join Is It ?

When we write R ⋈ 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

age	zip	disease
54	98125	heart
20	98120	flu
33	98120	lung

AnnonJob J

job	age	zip
lawyer	54	98125
cashier	20	98120

P ⟕ J

P.age	P.zip	P.disease	J.job	J.age	J.zip
54	98125	heart	lawyer	54	98125
20	98120	flu	cashier	20	98120
33	98120	lung	null	null	null

Some Examples

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

Name of supplier of parts with size greater than 10

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

Name of supplier of red parts or parts with size greater than 10

$\pi_{sname}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{psize>10} (\text{Part}) \cup \sigma_{pcolor='red'} (\text{Part})))$

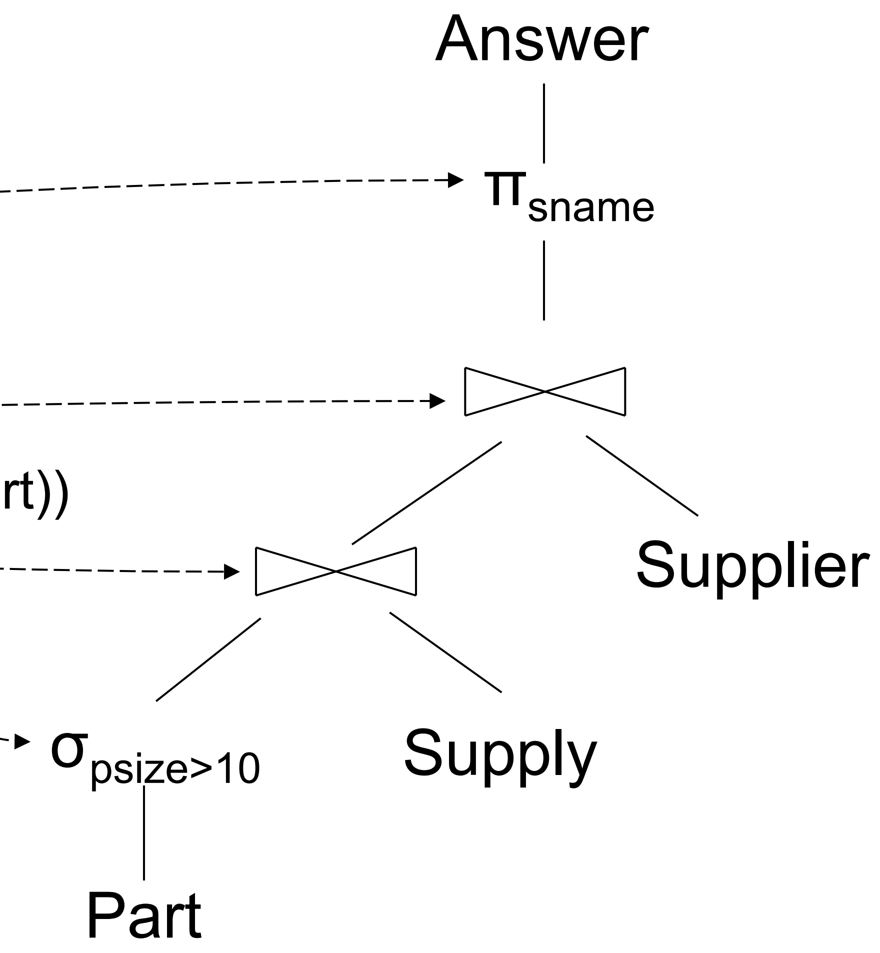
$\pi_{sname}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{psize>10 \lor pcolor='red'} (\text{Part})))$

Can be represented as trees as well

Representing RA Queries as Trees

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

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



Relational Algebra Operators

- Union ∪, ~~intersection ∩,~~ difference -
- Selection σ
- Projection π
- Cartesian product X, join ⋈
- (Rename ρ)
- Duplicate elimination δ
- Grouping and aggregation γ
- Sorting τ

RA

Extended RA

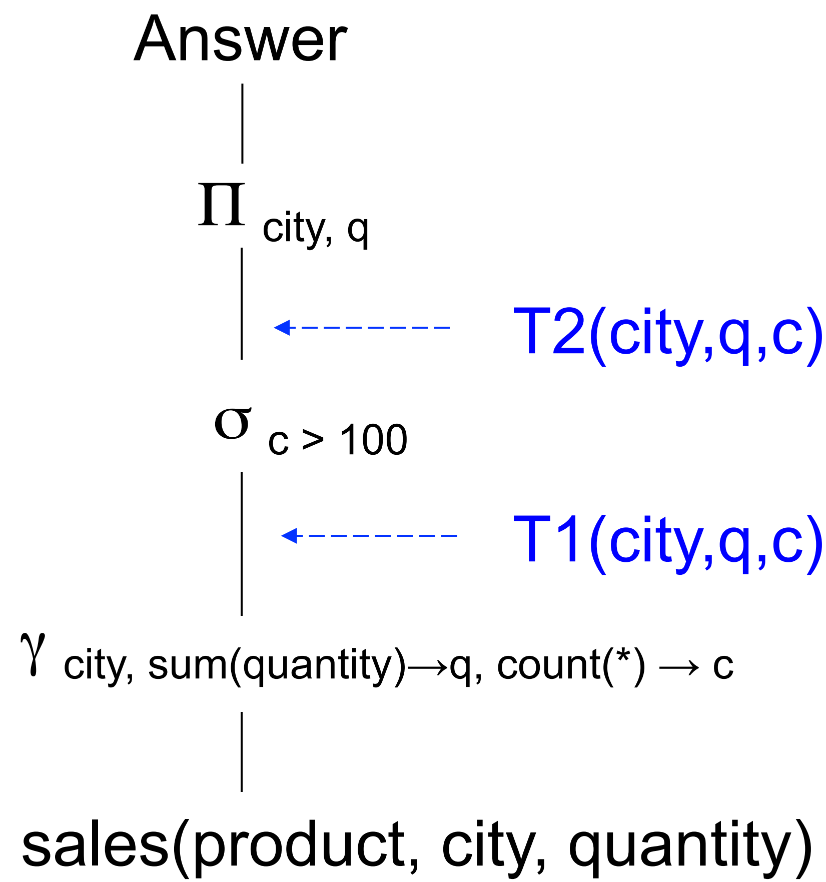
All operators take in 1 or more relations as inputs and return another relation

Extended RA: Operators on Bags

- Duplicate elimination δ
- Grouping γ
  - Takes in relation and a list of grouping operations (e.g., aggregates). Returns a new relation.
- Sorting τ
  - Takes in a relation, a list of attributes to sort on, and an order. Returns a new relation.

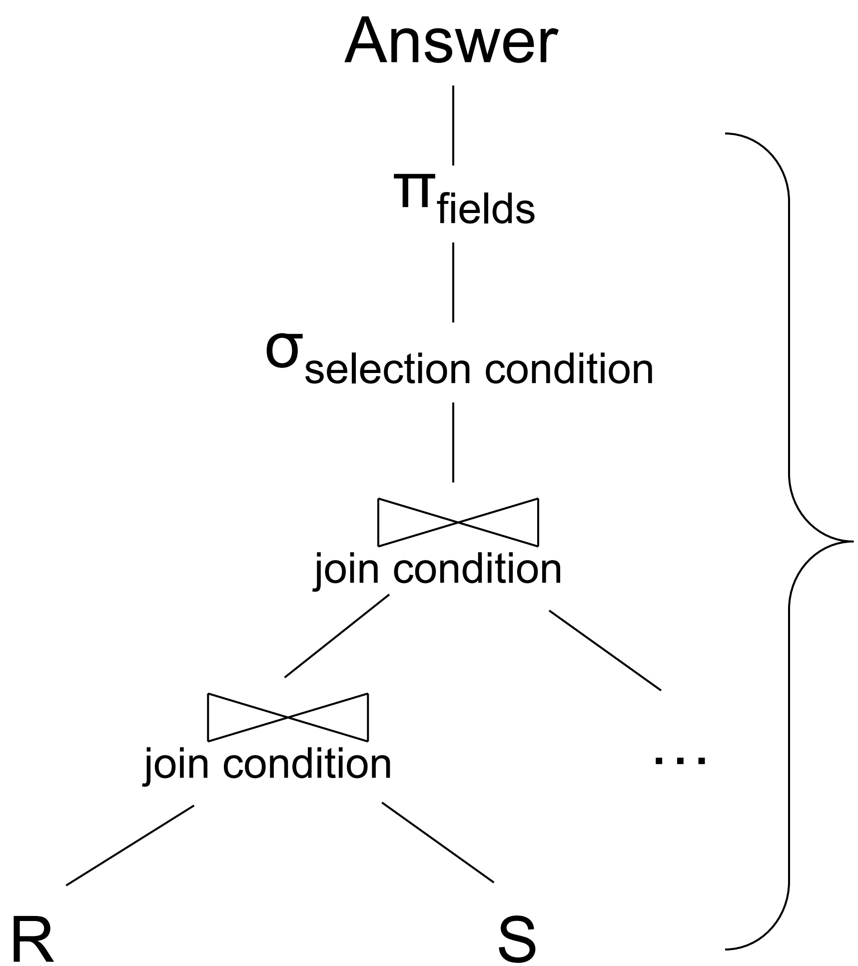
Using Extended RA Operators

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



T1, T2 = temporary tables

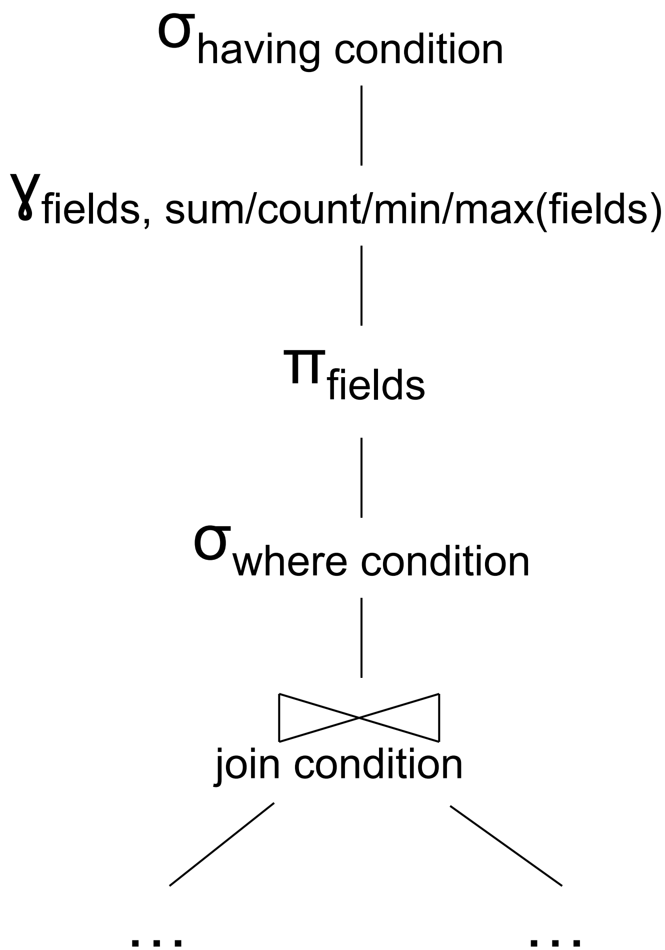
Typical Plan for a Query (1/2)




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

SELECT-PROJECT-JOIN
Query

Typical Plan for a Query (1/2)




```
SELECT fields
FROM R, S, …
WHERE condition
GROUP BY fields
HAVING condition
```

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

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)
```

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

How about Subqueries?

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

Correlation !

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

How about Subqueries?

```
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

```
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)
```

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

How about Subqueries?

Un-nesting

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

```
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)
```

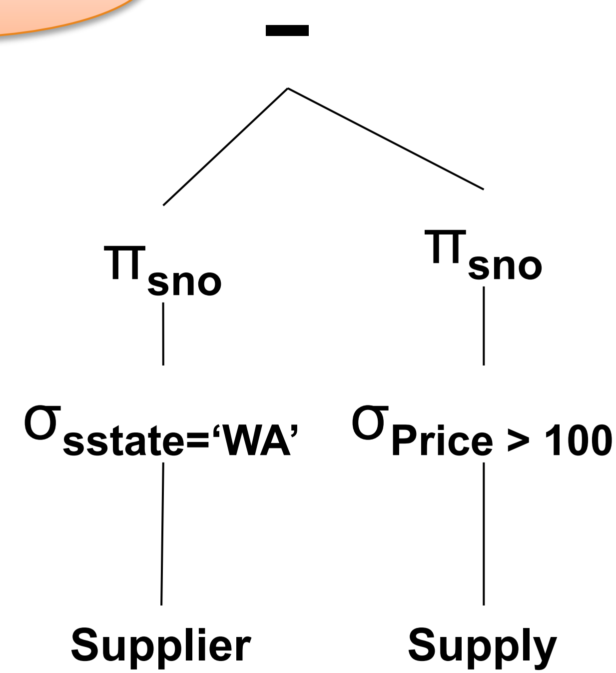
EXCEPT = set difference

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

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…



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 → 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
USE AdventureWorks2008R2;
GO
WITH DirectReports (ManagerID, EmployeeID, Title, DeptID, Level)
AS
(
-- Anchor member definition
    SELECT e.ManagerID, e.EmployeeID, e.Title, edh.DepartmentID,
        0 AS Level
    FROM dbo.MyEmployees AS e
    INNER JOIN HumanResources.EmployeeDepartmentHistory AS edh
        ON e.EmployeeID = edh.BusinessEntityID AND edh.EndDate IS NULL
    WHERE ManagerID IS NULL
    UNION ALL
-- Recursive member definition
    SELECT e.ManagerID, e.EmployeeID, e.Title, edh.DepartmentID,
        Level + 1
    FROM dbo.MyEmployees AS e
    INNER JOIN HumanResources.EmployeeDepartmentHistory AS edh
        ON e.EmployeeID = edh.BusinessEntityID AND edh.EndDate IS NULL
    INNER JOIN DirectReports AS d
        ON e.ManagerID = d.EmployeeID
)
-- Statement that executes the CTE
SELECT ManagerID, EmployeeID, Title, DeptID, Level
FROM DirectReports
INNER JOIN HumanResources.Department AS dp
    ON DirectReports.DeptID = dp.DepartmentID
WHERE dp.GroupName = N'Sales and Marketing' OR Level = 0;
GO
```

```
Manager(eid) :- Manages(_, eid)

DirectReports(eid, 0) :-
        Employee(eid),
        not Manager(eid)

DirectReports(eid, level+1) :-
        DirectReports(mid, level),
        Manages(mid, eid)
```

SQL Query vs Datalog

(which would you rather write?)

(any Java fans out there?)

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

Schema

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

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

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

```
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).
```

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

Datalog: Facts and Rules

Facts = tuples in the database

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

Rules = queries

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

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

Datalog: Facts and Rules

Facts = tuples in the database

```
Actor(344759,'Douglas', 'Fowley').
Casts(344759, 29851).
Casts(355713, 29000).
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).
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').
```

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

Datalog: Facts and Rules

Facts = tuples in the database

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

Rules = queries

```
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

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

Datalog: Facts and Rules

Facts = tuples in the database

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

Rules = queries

```
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)

Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

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

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

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

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
                    Casts(z,x2), Movie(x2,y2,1940)
```

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

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

Datalog: Facts and Rules

Facts = tuples in the database

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

Rules = queries

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

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

```
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
                         Casts(z,x2), Movie(x2,y2,1940)
```

Extensional Database Predicates = EDB = Actor, Casts, Movie

Intensional Database Predicates = IDB = Q1, Q2, Q3

Datalog: Terminology

head — body

atom — atom — atom (aka subgoal)

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

f, l = head variables

x,y,z = existential variables

More Datalog Terminology

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

- $R_i(args_i)$ called an *atom*, or a *relational predicate*
- $R_i(args_i)$ evaluates to true when relation R_i contains the tuple described by $args_i$.
  - Example: Actor(344759, 'Douglas', 'Fowley') is true
- In addition we can also have arithmetic predicates
  - Example: z > '1940'.
- Logicblox uses <- instead of :-
- Book uses AND instead of ,

```
Q(args) <-  R1(args), R2(args), ....
```

```
Q(args) :- R1(args) AND R2(args) ....
```

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

Semantics of a Single Rule

- Meaning of a datalog rule = a logical statement !

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

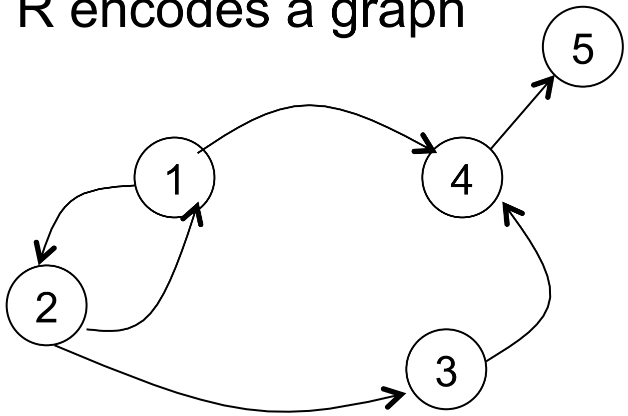
- For all x, y, z: if (x,y,z) ∈ Movies and z = '1940' then y is in Q1 (i.e. is part of the answer)
- $\forall x \forall y \forall z$ [(Movie(x,y,z) and z='1940') ⇒ Q1(y)]
- Logically equivalent:
  $\forall y$ [($\exists x \exists z$ Movie(x,y,z) and z='1940') ⇒ Q1(y)]
- Thus, head variables are called "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.

Example

R encodes a graph

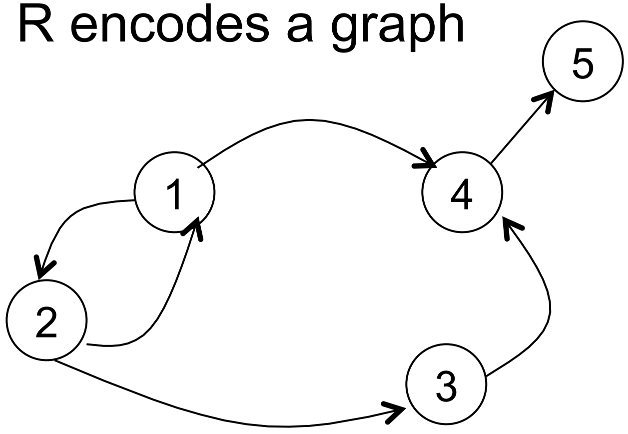


R=

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

Example

R encodes a graph



R=

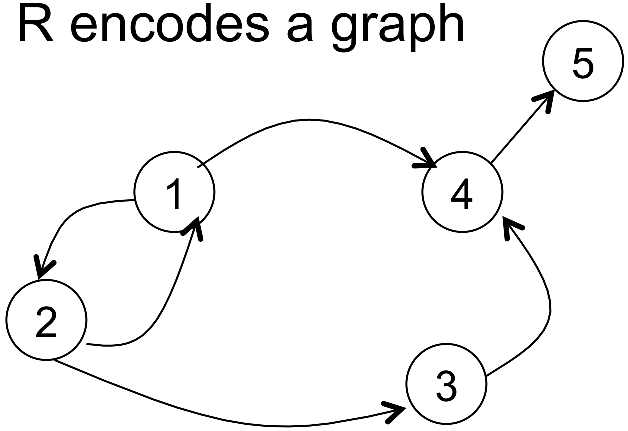
1	2
2	1
2	3
1	4
3	4
4	5

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

What does it compute?

Example

R encodes a graph



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

What does it compute?

R=

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

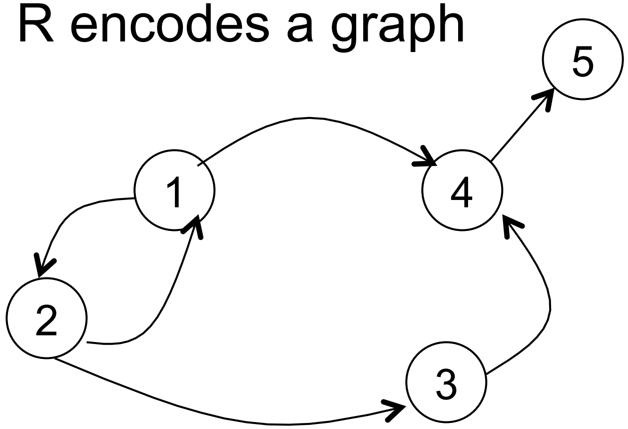
Initially:
T is empty.

Example

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

What does it compute?

R encodes a graph



R=

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

Initially:

T is empty.

First iteration:

T =

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

First rule generates this

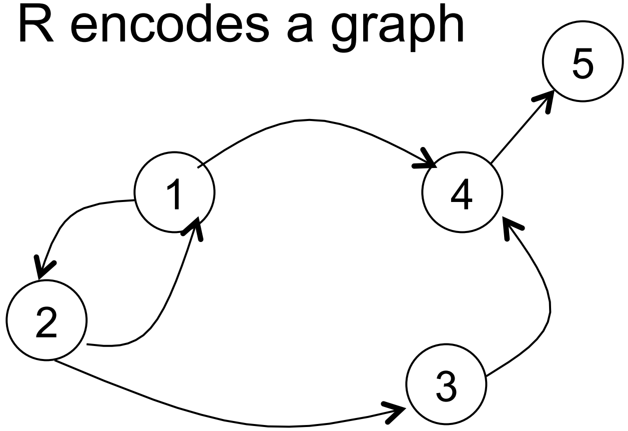
Second rule
generates nothing
(because T is empty)

Example

$$T(x,y) \text{ :- } R(x,y)$$
$$T(x,y) \text{ :- } R(x,z),\ T(z,y)$$

What does it compute?

R encodes a graph



R=

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

Initially:

T is empty.

First iteration:

T =

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

Second iteration:

T =

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

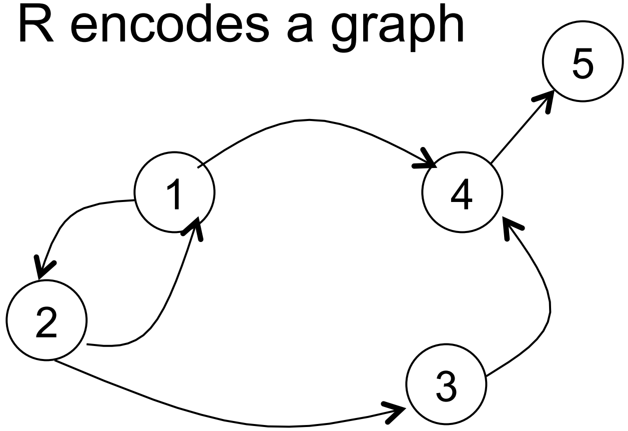
First rule generates this (rows 1–6)

Second rule generates this (rows 7–12)

New facts

Example

R encodes a graph




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

What does it compute?

R=

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

Initially:
T is empty.

First iteration:
T =

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

Second iteration:
T =

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

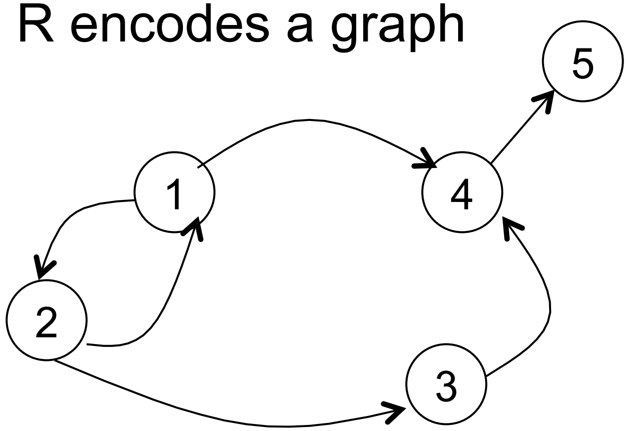
Third iteration:
T =

1	2	Both rules
2	1	Both rules
2	3	First rule
1	4	First rule
3	4	First rule
4	5	First rule
1	1	
2	2	Second rule
1	3	Second rule
2	4	Second rule
1	5	Second rule
3	5	Second rule
2	5	Second rule (New fact)

New fact

Example

R encodes a graph



$$T(x,y) \text{ :- } R(x,y)$$
$$T(x,y) \text{ :- } R(x,z), T(z,y)$$

What does it compute?

R=

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

Initially:
T is empty.

First iteration:
T =

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

Second iteration:
T =

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

Third iteration:
T =

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

Fourth iteration
T =
(same)

No new facts. DONE

Datalog Semantics

Fixpoint semantics

- Start:

  IDB_0 = empty relations

  $t = 0$

  Repeat:

  IDB_{t+1} = Compute Rules(EDB, IDB_t)

  $t = t+1$

  Until $IDB_t = IDB_{t-1}$

- Remark: since rules are monotone:

  $\varnothing = IDB_0 \subseteq IDB_1 \subseteq IDB_2 \subseteq \ldots$

- 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,
     $\forall$ vars [(Body(EDB,IDB) $\Rightarrow$ Head(IDB)]
  2) Is the smallest IDB satisfying (1)

- Theorem: there exists a smallest IDB satisfying (1)

Datalog Semantics: Example

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

Fixpoint semantics:

- Start: $T_0 = \varnothing$; $t = 0$
  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.

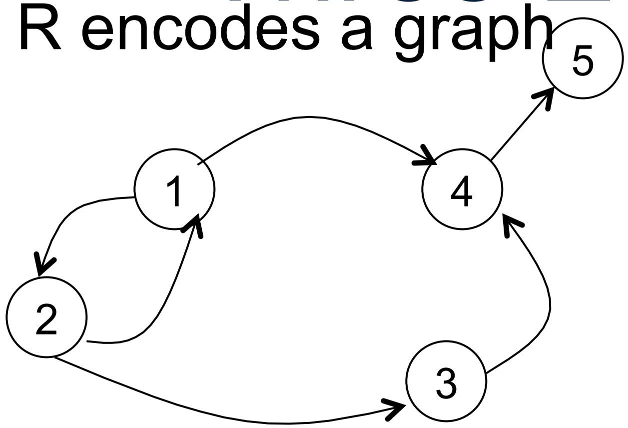
- $\forall x \forall y\ [(R(x,y) \Rightarrow T(x,y)] \wedge$
  $\forall x \forall y \forall z\ [(R(x,z) \wedge T(z,y)) \Rightarrow T(x,y)]$

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



R=

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

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

Right linear

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

Left linear

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

Non-linear

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

fName	lName
Alice	Smith
Bob	Toth
Carol	Unger

- 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)

fName	lName	friends
Alice	Smith	22
Bob	Toth	5
Carol	Unger	9

- Functional model:
Person[Alice,Smith]=22
Person[Bob,Toth]=5
Person[Carol,Unger]=9

fName	lName	
Alice	Smith	=22
Bob	Toth	=5
Carol	Unger	=9

Aggregates: use agg<<...>>

General syntax in Logicblox:

<- instead of :-

```
Q[headVars]     <-                    R1(args1),R2(args2),...
```

Meaning (in SQL)

```
select headVars
from R1, R2, ...
where ...
```

Aggregates: use agg<<...>>

General syntax in Logicblox:

<- instead of :-

```
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:

<- instead of :-

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

Meaning (in 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).
```

ParentChild(p,c)

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 */
```

ParentChild(p,c)

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

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

Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

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

Holds for
every y other than "Bob"
U1 = infinite!

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

Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

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

Holds for every y other than "Bob"
U1 = infinite!

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

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

Safe Datalog Rules

Here are *unsafe* datalog rules. What's "unsafe" about them ?

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

Holds for every y other than "Bob"
U1 = infinite!

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

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

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() <- !B().
B() <- !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 ! or agg in stratum n+1.
- LogicBlox (and others) accepts only stratified datalog.

Stratified Datalog

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

Stratum 1

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

Stratum 2

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

Stratified Datalog

```
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
-----------------------------------
N[x] = m  <-  agg<<m = count()>> D(x,y).
Q(d)  <-  N["Alice"]=d.
```

Stratum 1

Stratum 2

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

```
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
-----------------------------------
Q(x)  <-  D("Alice",x), !D("Bob",x).
```

Stratum 1

Stratum 2

May use !D

Stratified Datalog

```
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
-----------------------------------
N[x] = m  <-  agg<<m = count()>> D(x,y).
Q(d)  <-  N["Alice"]=d.
```

Stratum 1

Stratum 2

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

```
D(x,y) <- ParentChild(x,y).
D(x,z) <- D(x,y), ParentChild(y,z).
-----------------------------------
Q(x)  <-  D("Alice",x), !D("Bob",x).
```

Stratum 1

Stratum 2

May use !D

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

Non-stratified

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

R(A,B,C)
S(D,E,F)
T(G,H)

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)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Intersection:

R(A,B,C) ∩ S(D,E,F)

I(x,y,z) :- R(x,y,z), S(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Selection: $\sigma_{x>100 \text{ and } y=\text{'foo'}}$ (R)

L(x,y,z) :- R(x,y,z), x > 100, y='foo'

Selection: $\sigma_{x>100 \textbf{ or } y=\text{'foo'}}$ (R)

L(x,y,z) :- R(x,y,z), x > 100

L(x,y,z) :- R(x,y,z), y='foo'

R(A,B,C)
S(D,E,F)
T(G,H)

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)

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

Projection: $\Pi_A(R)$

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

R(A,B,C)
S(D,E,F)
T(G,H)

RA to Datalog by Examples

To express difference, we add negation

R – S

D(x,y,z) :- R(x,y,z), NOT S(x,y,z)

R(A,B,C)
S(D,E,F)
T(G,H)

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

R(A,B,C)
S(D,E,F)
T(G,H)

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

Friend(name1, name2)
Enemy(name1, name2)

More Examples w/o Recursion

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

```
A(x) :- Friend('Joe', x)
A(x) :- Friend('Joe', z), Friend(z, x)
```

Friend(name1, name2)
Enemy(name1, name2)

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)
```

Friend(name1, name2)
Enemy(name1, name2)

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!

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

Friend(name1, name2)

Enemy(name1, name2)

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)
```

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) = "x,y are in the same generation"

```
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