CSE544
Data Management

Lectures 1-3:
Introduction, SQL

Outline

- Introduction, class overview
- Database management systems (DBMS)
- The relational model
- SQL (continued on Wed.)

Course Staff

- Instructor: Dan Suciu
  - Office hours: Tuesdays, 5:30-6:20
- TAs (Office hours TBD)
  - Maureen Daum
  - Brandon Ko
  - Kyle Yan

Goals of the Class

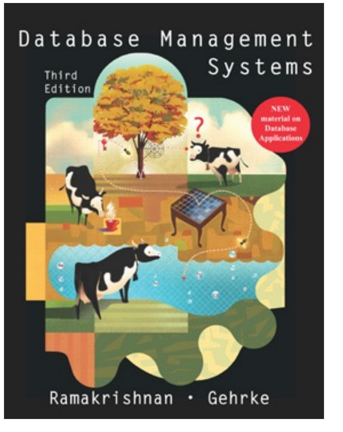
- Relational Data Model
  - Data models, data independence, declarative query language.
- Relational Database Systems
  - Storage, query execution and optimization
  - Parallel data processing, column-oriented db etc.
- Transactions
  - Optimistic/pessimistic concurrency control
  - [ARIES recovery system – will likely run out of time]

Readings

- Paper reviews
  - Mix of old seminal papers and new papers
  - Papers are available on class website

- Lecture notes (the slides)
  - Posted on class website after each lecture

- Background from:
  - Database Management Systems. **Third Ed**. Ramakrishnan and Gehrke. McGraw-Hill.



Class Resources

Website: lectures, assignments

- http://www.cs.washington.edu/csep544

Canvas: zoom, videos

Ed: discussion board

Evaluation

- Assignments 50%
- Reviews 20%
- Mini-Project 20%
- Intangibles 10%

Assignments – 50%

- **HW1**: Data analysis in postgres
- **HW2**: Data analysis in Snowflake
- **HW3**: Query Execution and SimpleDB
- **HW4**: Datalog
- **HW5**: Spark

Paper reviews – 20%

- Recommended length: ½ page – 1 page
  - Summary of main points
  - Critical discussion
- Grading: credit/partial-credit/no-credit
- Submit review *before* the lecture

- First review due on Wednesday!

MiniProject – 20%

Topic of your own choosing, open ended

- Suggestion 1: based on a paper
  - Repeat 1-2 experiments
  - Try variations
  - Compare with another system
  - Something else
- Suggestion 2: based on your work
  - Evaluate a technology that you need at work

Intangibles 10%

- Class participation
- Exceptionally good reviews, or homework, or project
- Etc, etc

How to Turn In

- Homeworks: gitlab
- Project: gitlab
- Reviews: google forms

Now onward to the world of databases!

Data Management

- **Entities**: employees, positions (ceo, manager, cashier), stores, products, sells, customers.

- **Relationships**: employee positions, staff of each store, inventory of each store.

Database Management System

- A DBMS is a software system designed to provide data management services

- Examples of DBMS
  - Oracle, DB2 (IBM), SQL Server (Microsoft),
  - PostgreSQL, MySQL,…
  - Snowflake, Redshift, SQL Azure, BigQuery

DBMS Functionality

- Create & persistently store large datasets
- Efficiently query & update
- Change structure (e.g., add attributes)
- Concurrency control: enable simultaneous updates
- Crash recovery
- Access control, security, integrity

Single Client

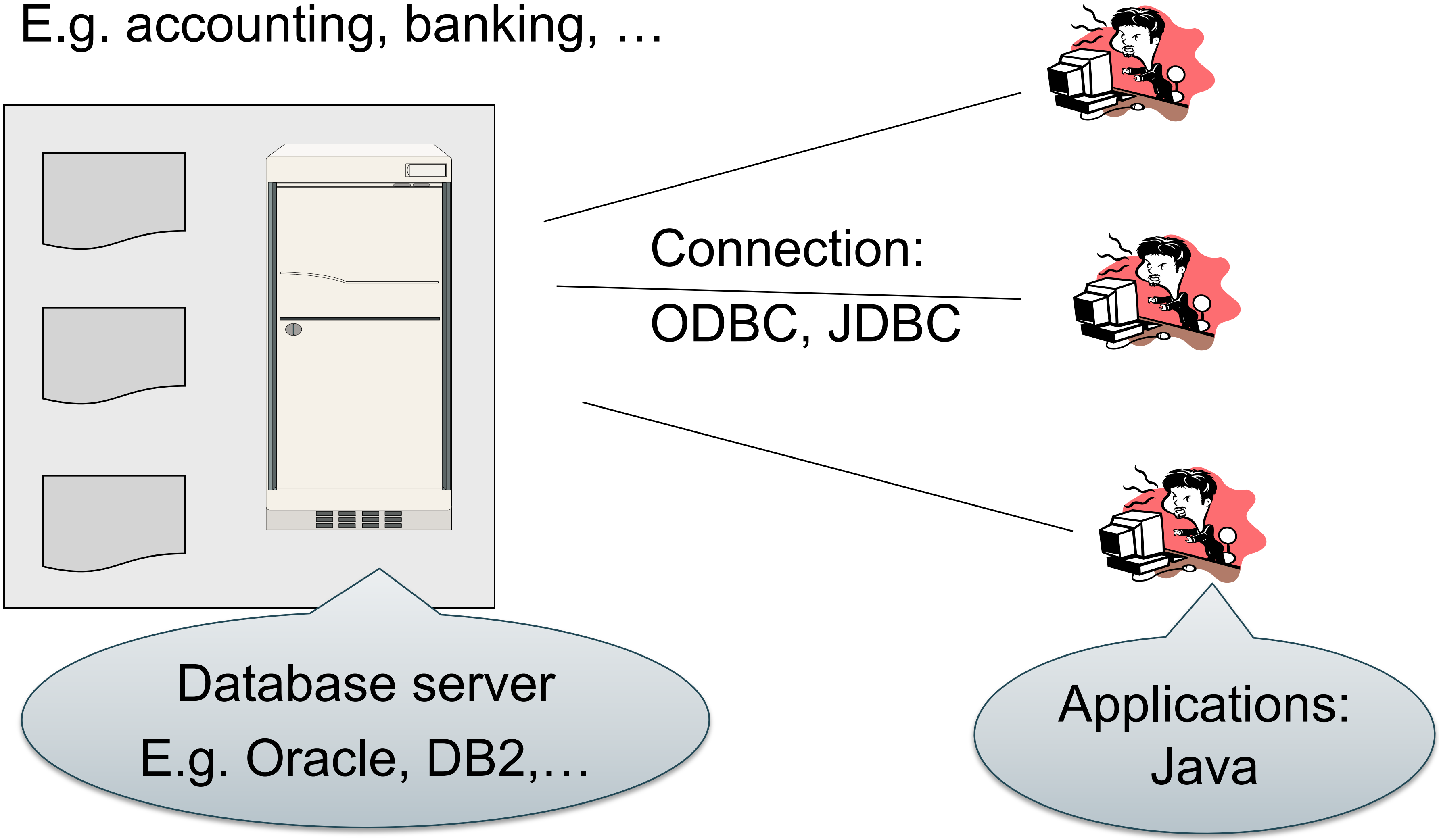
E.g. data analytics

Application and database
on the same computer

E.g. sqlite, postgres

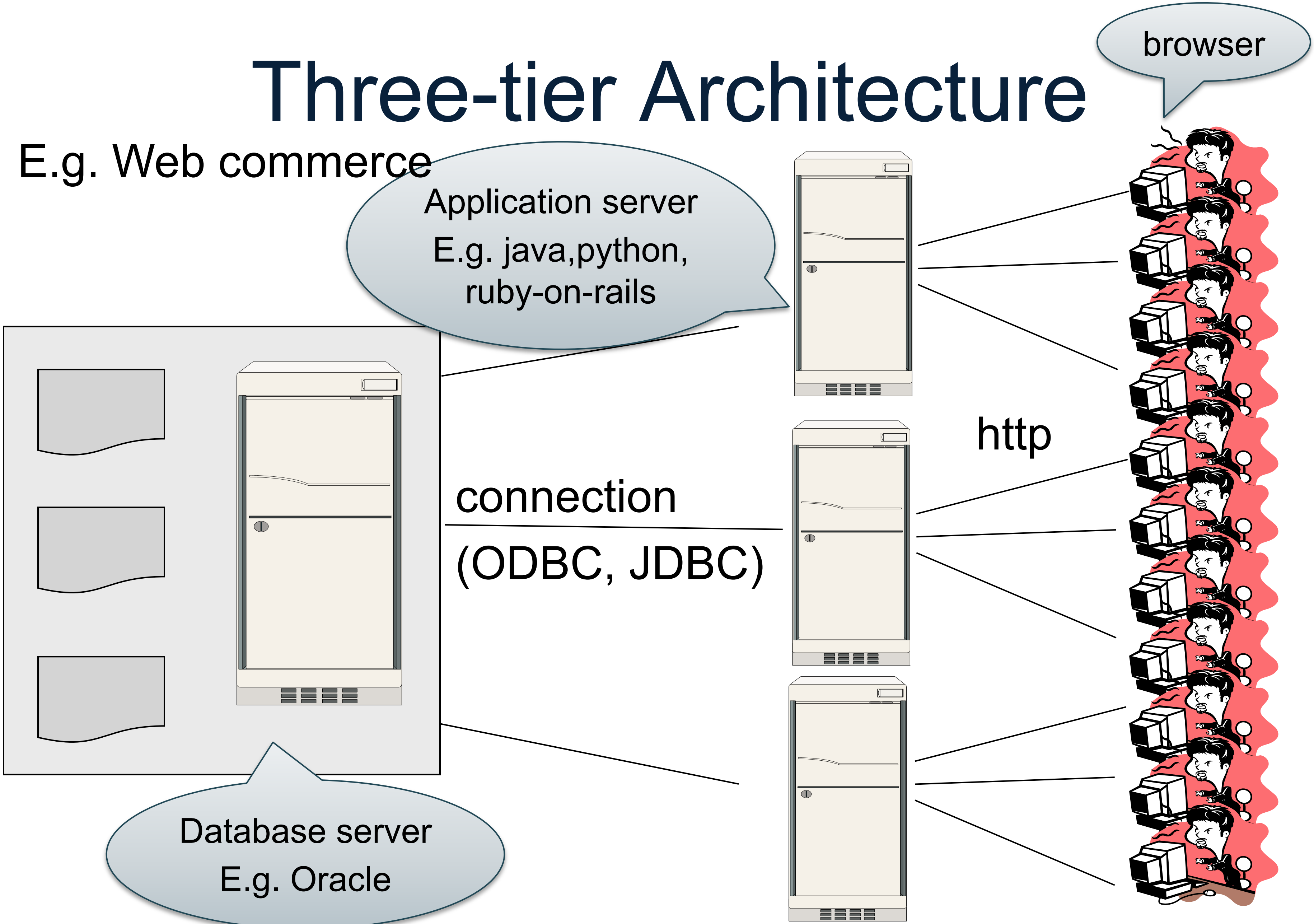
Two-tier Architecture
Client-Server

E.g. accounting, banking, …

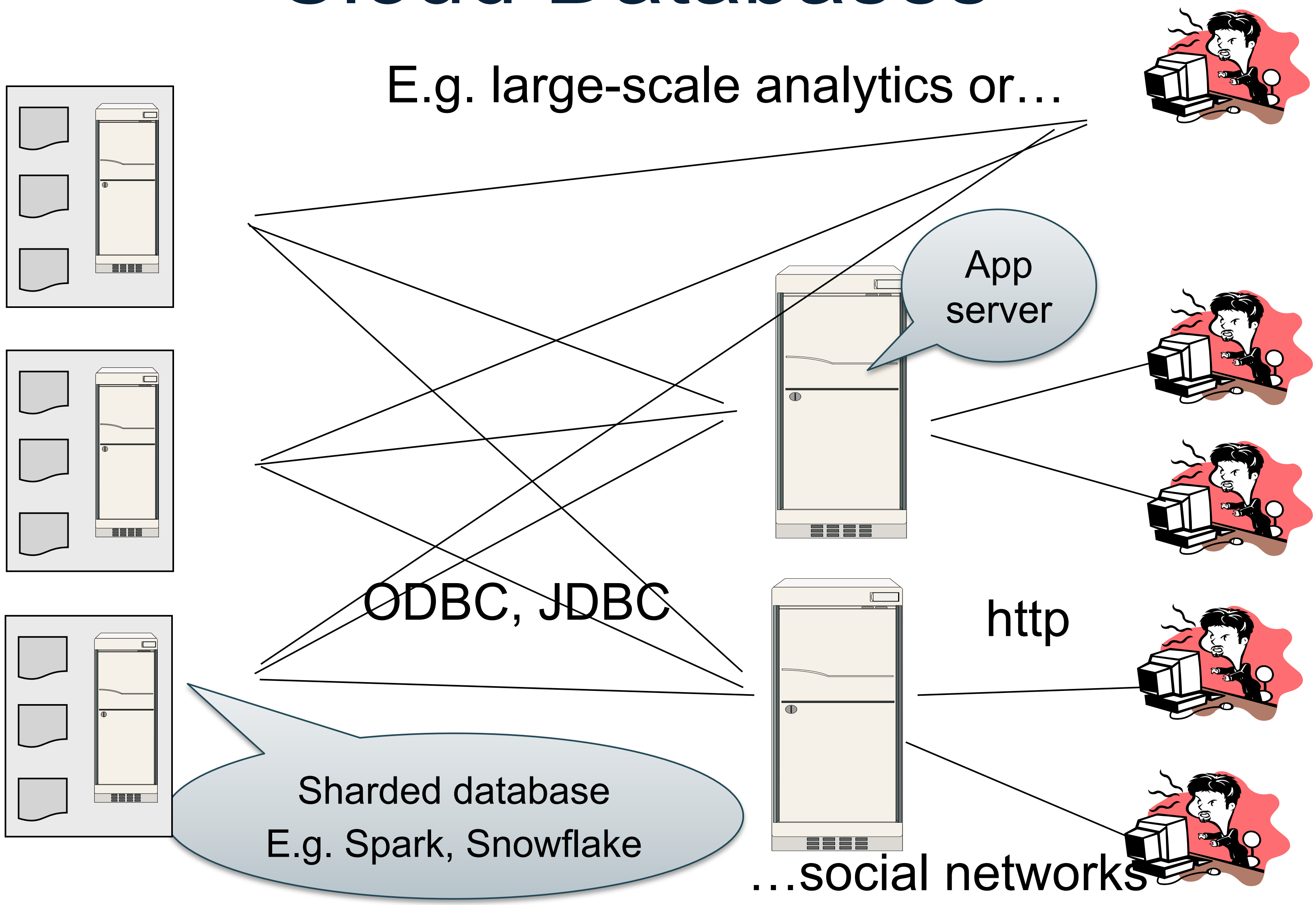


Three-tier Architecture

E.g. Web commerce



Cloud Databases



Workloads

- OLTP – online transaction processing
- OLAP – online analytics processing, a.k.a. Decision Support

Most of
this course

Relational Data Model

Relational Data Model

- A Database is a collection of relations
- A Relation is a set of tuples
  - Also called Table
- A Tuple t is an element of $\mathbf{Dom_1 \times Dom_2 \times \ldots \times Dom_n}$
  - $\mathbf{Dom_i}$ is the domain of attribute **i**
  - **n** is number of attributes of the relation
  - Also called Row or Record

Discussion

- Rows in a relation:
  - Ordering immaterial (a relation is a set)
  - All rows are distinct – set semantics
  - Query answers may have duplicates – bag semantics

Data independence!

- Columns in a tuple:
  - Ordering is immaterial
  - Applications refer to columns by their names

Or is it?

- Domain of each column is a primitive type

Schema

- Relation schema: describes column heads
  - Relation name
  - Name of each field (or column, or attribute)
  - Domain of each field
  - The *arity* of the relation = # attributes

- Database schema: set of all relation schemas

Instance

- Relation instance: concrete table content
  - Set of records matching the schema
  - The *cardinality* or *size* of the relation = # tuples

- Database instance: set of relation instances

What is the schema?
What is the instance?

Supplier

sno	**sname**	**scity**	**sstate**
1005	ACME	Seattle	WA
1006	Freddie	Austin	TX
1007	Joe’s	Seattle	WA
1008	ACME	Austin	TX

What is the schema?
What is the instance?

Relation schema

Supplier(<u>sno: integer</u>, sname: string, scity: string, sstate: string)

Supplier

sno	sname	scity	sstate
1005	ACME	Seattle	WA
1006	Freddie	Austin	TX
1007	Joe's	Seattle	WA
1008	ACME	Austin	TX

} instance

What is the schema?
What is the instance?

Relation schema

Supplier(<u>sno: integer</u>, sname: string, scity: string, sstate: string)

Supplier

sno	**sname**	**scity**	**sstate**
1005	ACME	Seattle	WA
1006	Freddie	Austin	TX
1007	Joe's	Seattle	WA
1008	ACME	Austin	TX

} instance

In class: discuss keys, foreign keys, FD

Relational Query Language

- Set-at-a-time:
  - Query inputs and outputs are relations
- Two variants of the query language:
  - SQL: declarative
  - Relational algebra: specifies order of operations

SQL

SQL

- Standard query language
- Introduced late 70's, now it ballooned
- We briefly review "core SQL" (whatever that means); study more on you own!
- Read by Wed: *<u>A case against SQL</u>*

Structured Query Language: SQL

- Data definition language: DDL
  - Statements to create, modify tables and views
  - CREATE TABLE …,
    CREATE VIEW …,
    ALTER TABLE…
- Data manipulation language: DML
  - Statements to issue queries, insert, delete data
  - SELECT-FROM-WHERE…, (Our focus)
    INSERT…,
    UPDATE…,
    DELETE…

SQL Query

Basic form: (plus many many more bells and whistles)

```
SELECT  <attributes>
FROM    <one or more relations>
WHERE   <conditions>
```

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

Quick Review of SQL

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

Quick Review of SQL

```
SELECT DISTINCT  z.pno, z.pname, x.scity
FROM      Supplier x, Supply y, Part z
WHERE     x.sno = y.sno
          and y.pno = z.pno
          and x.sstate = 'WA'
          and y.price < 100
```

What does this query compute?

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

Terminology

- Selection/filter: return a subset of the rows:
  - SELECT * FROM Supplier WHERE scity = 'Seattle'

Filtering is called *selection* in RA

- Projection: return subset of the columns:
  - SELECT DISTINCT scity FROM Supplier;
- Join: refers to combining two or more tables
  - SELECT * FROM Supplier, Supply, Part …

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

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

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

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT  y.pno
FROM      Supplier x, Supply y
WHERE     x.scity = 'Seattle'
          and x.scity = 'Portland'
          and x.sno = y.sno
```

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

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT  y.pno
FROM      Supplier x, Supply y
WHERE     x.scity = 'Seattle'
      and x.scity = 'Portland'
      and x.sno = y.sno
```

This doesn't work… Why?

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

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT  y.pno
FROM      Supplier x, Supply y
WHERE     (x.scity = 'Seattle'
          or x.scity = 'Portland')
       and x.sno = y.sno
```

Does this work?

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

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT  y.pno
FROM        Supplier x, Supply y
WHERE      (x.scity = 'Seattle'
         or x.scity = 'Portland')
       and x.sno = y.sno
```

Does this work?

Nope!

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

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

Need TWO Suppliers and TWO Supplies

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE     x1.scity = 'Seattle'
          and x1.sno = y1.sno
          and x2.scity = 'Portland'
          and x2.sno = y2.sno
          and y1.pno = y2.pno
```

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

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

Need TWO Suppliers and TWO Supplies

```
SELECT DISTINCT  y1.pno
FROM     Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE    x1.scity = 'Seattle'
         and x1.sno = y1.sno
         and x2.scity = 'Portland'
         and x2.sno = y2.sno
         and y1.pno = y2.pno
```

one in Seattle the other in Portland

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

Self-Joins

Find the Parts numbers available both from suppliers in Seattle, and suppliers in Portland

```
SELECT DISTINCT  y1.pno
FROM      Supplier x1, Supplier x2, Supply y1, Supply y2
WHERE     x1.scity = 'Seattle'
          and x1.sno = y1.sno
          and x2.scity = 'Portland'
          and x2.sno = y2.sno
          and y1.pno = y2.pno
```

Need TWO Suppliers and TWO Supplies

one in Seattle the other in Portland

the SAME part

Nested-Loop Semantics of SQL

SELECT $a_1, a_2, \ldots, a_k$
FROM R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE Conditions

Nested-Loop Semantics of SQL

```
SELECT a_1, a_2, …, a_k
FROM   R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE  Conditions
```

SELECT $a_1, a_2, \ldots, a_k$
FROM R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE Conditions

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

Nested-Loop Semantics of SQL

SELECT $a_1, a_2, \ldots, a_k$
FROM R_1 AS x_1, R_2 AS x_2, …, R_n AS x_n
WHERE Conditions

This SEMANTICS!
It is NOT how the engine computes the query!

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

NULLs in SQL

- A NULL value means missing, or unknown, or undefined, or inapplicable

NULLs in WHERE Clause

Boolean predicate:

- Atomic: Expr1 op Expr2
- AND / OR / NOT

Example:

price < 100 and (pcolor='red' or psize=2)

How do we compute the predicate when values are NULL?

`Part(pno,pname,price,psize,pcolor)`

Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

Part(pno,pname,price,psize,pcolor)

Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

```
select *
from Part
where price < 100
 and (psize=2 or pcolor='red')
```

Part(pno,pname,price,psize,pcolor)

Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

```
select *
from Part
where price < 100
 and (psize=2 or pcolor='red')
```

pno	pname	price	psize	pcolor
1	iPad	500	13	blue
2	Scooter	99	NULL	NULL
3	Charger	NULL	NULL	red
1	iPad	50	2	NULL

Part(pno,pname,price,psize,pcolor)

Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

```
select *
from Part
where price < 100
 and (psize=2 or pcolor='red')
```

pno	pname	price	psize	pcolor
1	iPad	500	13	blue
2	Scooter	99	NULL	NULL
3	Charger	NULL	NULL	red
1	iPad	50	2	NULL

`Part(pno,pname,price,psize,pcolor)`

Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

```
select *
from Part
where price < 100
 and (psize=2 or pcolor='red')
```

pno	pname	price	psize	pcolor
1	iPad	500	13	blue
2	Scooter	99	NULL	NULL
3	Charger	NULL	NULL	red
1	iPad	50	2	NULL

`Part(pno,pname,price,psize,pcolor)`

Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

```
select *
from Part
where price < 100
 and (psize=2 or pcolor='red')
```

pno	pname	price	psize	pcolor
1	iPad	500	13	blue
2	Scooter	99	NULL	NULL
3	Charger	NULL	NULL	red
1	iPad	50	2	NULL

Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

```
-- problem: (A or not(A)) ≠ true
-- does NOT return all Products
select *
from Product
where (price <= 100) or (price > 100)
```

Three-Valued Logic

- False=0, Unknown=0.5, True=1
- A op B is
  - **False** or **True** when both A, B are not null
  - **Unknown** otherwise
- AND, OR, NOT are **min**, **max**.
- Return only tuples whose condition is **True**

```
-- problem: (A or not(A)) ≠ true
-- does NOT return all Products
select *
from Product
where (price <= 100) or (price > 100)
```

```
-- returns ALL Products
select *
from Product
where (price <= 100) or (price > 100)
         or isNull(price)
```

Likbkin’s Critique Of SQL

- Libkin’s slides: *A Case Against SQL*
- In class: discuss some of the main inconsistencies in SQL

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

More SQL: Aggregates

```
SELECT count(*)
FROM   Part
```

What do these queries compute?

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

More SQL: Aggregates

```
SELECT count(*)
FROM   Part
```

What do these queries compute?

```
SELECT x.scity, avg(psize)
FROM   Supplier x, Supply y, Part z
WHERE  x.sno = y.sno  and y.pno = z.pno
GROUP BY x.scity
```

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

More SQL: Aggregates

```
SELECT count(*)
FROM   Part
```

What do these queries compute?

```
SELECT x.scity, avg(psize)
FROM   Supplier x, Supply y, Part z
WHERE  x.sno = y.sno  and y.pno = z.pno
GROUP BY x.scity
```

```
SELECT x.scity, avg(psize)
FROM   Supplier x, Supply y, Part z
WHERE  x.sno = y.sno  and y.pno = z.pno
GROUP BY x.scity
HAVING count(*) > 200
```

Discussion

- SQL Aggregates = simple data analytics
- Semantics:
  1. FROM-WHERE (nested-loop semantics)
  2. Group answers by GROUP BY attrs
  3. Apply HAVING predicates on groups
  4. Apply SELECT aggregates on groups
- Aggregate functions:
  - count, sum, min, max, avg
- DISTINCT same as GROUP BY

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

prodName is foreign Key

Outer joins

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

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

prodName is foreign Key

Outer joins

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

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

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

prodName is foreign Key

Outer joins

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

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

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

prodName is foreign Key

Outer joins

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

```sql
SELECT x.name, x.category, y.store
FROM   Product x, Purchase y
WHERE  x.name = y.prodName
```

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

missing

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Category	Store
Gizmo	gadget	Wiz
Camera	Photo	Ritz
Camera	Photo	Wiz

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

prodName is foreign Key

Outer joins

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

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

Product

Name	Category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

ProdName	Store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

Output

Name	Category	Store
Gizmo	gadget	Wiz
Camera	Photo	Ritz
Camera	Photo	Wiz
OneClick	Photo	NULL

Now it's present

Left Outer Join (Details)

from R left outer join S on C1 where C2

1. Compute cross product R×S
2. Filter on C1
3. Add all R records without a match
4. Filter on C2

Left Outer Join (Details)

```
select ...
from    R left outer join S on C1
where   C2
```

```
Tmp = {}
for x in R do                         // left outer join using C1
    for y in S do
        if C1 then Tmp = Tmp ∪ {(x,y)}
for x in R do
    if not (x in Tmp) then Tmp = Tmp ∪ {(x,NULL)}

Answer = {}                           // apply condition C2
for (x,y) in Tmp if C2 then Answer = Answer ∪ {(x,y)}
return Answer
```

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

prodName is foreign Key

ON v.s. WHERE

- Outer join condition in the ON clause
- Different from the WHERE clause
- Compare:

```
SELECT x.name, y.store
FROM   Product x
LEFT OUTER JOIN Purchase y
ON     x.name = y.prodName
   AND y.price < 10
```

```
SELECT x.name, y.store
FROM   Product x
LEFT OUTER JOIN Purchase y
ON     x.name = y.prodName
WHERE y.price < 10
```

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

prodName is foreign Key

ON v.s. WHERE

- Outer join condition in the ON clause
- Different from the WHERE clause
- Compare:

```
SELECT x.name, y.store
FROM   Product x
LEFT OUTER JOIN Purchase y
ON     x.name = y.prodName
   AND y.price < 10
```

Includes products that were never purchased with price < 10

```
SELECT x.name, y.store
FROM   Product x
LEFT OUTER JOIN Purchase y
ON     x.name = y.prodName
WHERE y.price < 10
```

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

prodName is foreign Key

ON v.s. WHERE

- Outer join condition in the ON clause
- Different from the WHERE clause
- Compare:

```
SELECT x.name, y.store
FROM   Product x
LEFT OUTER JOIN Purchase y
ON     x.name = y.prodName
   AND y.price < 10
```

Includes products that were never purchased with price < 10

```
SELECT x.name, y.store
FROM   Product x
LEFT OUTER JOIN Purchase y
ON     x.name = y.prodName
WHERE y.price < 10
```

Includes products that were never purchased, *then* checks price <10

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

prodName is foreign Key

ON v.s. WHERE

- Outer join condition in the ON clause
- Different from the WHERE clause
- Compare:

```
SELECT x.name, y.store
FROM   Product x
LEFT OUTER JOIN Purchase y
ON     x.name = y.prodName
   AND y.price < 10
```

Includes products that were never purchased with price < 10

```
SELECT x.name, y.store
FROM   Product x
LEFT OUTER JOIN Purchase y
ON     x.name = y.prodName
WHERE y.price < 10
```

Includes products that were never purchased, *then* checks price <10

Same as inner join!

Joins

- **Inner join** = includes only matching tuples (i.e. regular join)
- **Left outer join** = includes everything from the left
- **Right outer join** = includes everything from the right
- **Full outer join** = includes everything

Other use of Relational Data

- Sparse vectors, matrics
- Graph databases

Sparse Matrix

$$A = \begin{bmatrix} 5 & 0 & -2 \\ 0 & 0 & -1 \\ 0 & 7 & 0 \end{bmatrix}$$

How can we represent it as a relation?

Sparse Matrix

$$A = \begin{bmatrix} 5 & 0 & -2 \\ 0 & 0 & -1 \\ 0 & 7 & 0 \end{bmatrix}$$

Row	Col	Val
1	1	5
1	3	-2
2	3	-1
3	2	7

Matrix Multiplication in SQL

$C = A \cdot B$

Matrix Multiplication in SQL

$$C = A \cdot B \qquad\qquad C_{ik} = \sum_j A_{ij} \cdot B_{jk}$$

Matrix Multiplication in SQL

$$C = A \cdot B$$

$$C_{ik} = \sum_j A_{ij} \cdot B_{jk}$$

```
SELECT A.row, B.col, sum(A.val*B.val)
FROM A, B
WHERE A.col = B.row
GROUP BY A.row, B.col;
```

Discussion

- Matrix multiplication = join + group-by
- Many operations can be written in SQL
- E.g. try at home: write in SQL
$$Tr(A \cdot B \cdot C)$$
where the trace is defined as:
$$Tr(X) = \sum_i X_{ii}$$
- Surprisingly, $A + B$ is a bit harder…

Matrix Addition in SQL

$$C = A + B$$

Matrix Addition in SQL

$$C = A + B$$

```
SELECT A.row, A.col, A.val + B.val as val
FROM   A, B
WHERE  A.row = B.row and A.col = B.col
```

Matrix Addition in SQL

$$C = A + B$$

```
SELECT A.row, A.col, A.val + B.val as val
FROM   A, B
WHERE  A.row = B.row and A.col = B.col
```

Why is this wrong?

Solution 1: Outer Joins

$$C = A + B$$

```
SELECT



FROM A full outer join B ON A.row = B.row and A.col = B.col;
```

Solution 1: Outer Joins

$$C = A + B$$

```
SELECT


 (CASE WHEN A.val is null THEN 0 ELSE A.val END) +
 (CASE WHEN B.val is null THEN 0 ELSE B.val END)  as val
FROM A full outer join B ON A.row = B.row and A.col = B.col;
```

Solution 1: Outer Joins

$$C = A + B$$

```
SELECT
 (CASE WHEN A.row is null THEN B.row ELSE A.row END) as row,

 (CASE WHEN A.val is null THEN 0 ELSE A.val END) +
 (CASE WHEN B.val is null THEN 0 ELSE B.val END)  as val
FROM A full outer join B ON A.row = B.row and A.col = B.col;
```

Solution 1: Outer Joins

$$C = A + B$$

```
SELECT
 (CASE WHEN A.row is null THEN B.row ELSE A.row END) as row,
 (CASE WHEN A.col is null THEN B.col ELSE A.col END) as col,
 (CASE WHEN A.val is null THEN 0 ELSE A.val END) +
 (CASE WHEN B.val is null THEN 0 ELSE B.val END)  as val
FROM A full outer join B ON A.row = B.row and A.col = B.col;
```

Solution 2: Group By

$$C = A + B$$

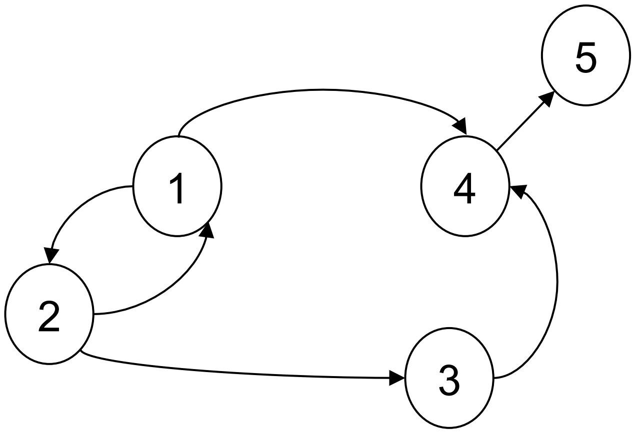
```
SELECT m.row, m.col, sum(m.val)
FROM (SELECT * FROM A
          UNION ALL
        SELECT * FROM B) as m
GROUP BY m.row, m.col;
```

Graph Databases

- Graph databases systems are a niche category of products specialized for processing large graphs
- E.g. Neo4J, TigerGraph
- A graph is a special case of a relation, and can be processed using SQL

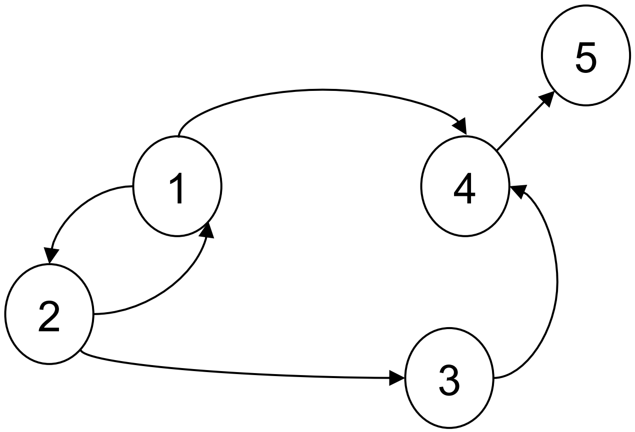
Graph Databases

A graph:



Graph Databases

A graph:

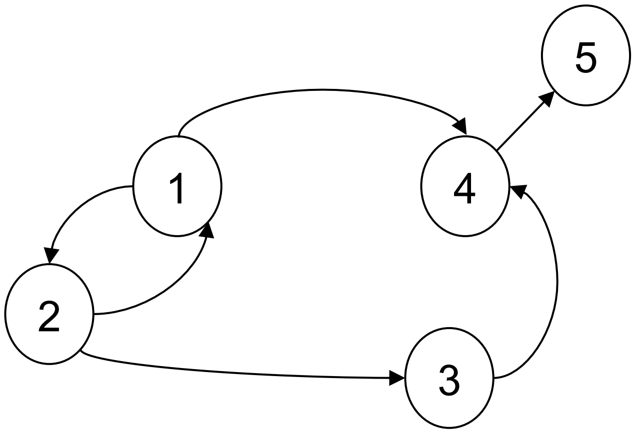


A relation: Edge

src	dst
1	2
2	1
2	3
1	4
3	4
4	5

Graph Databases

A graph:



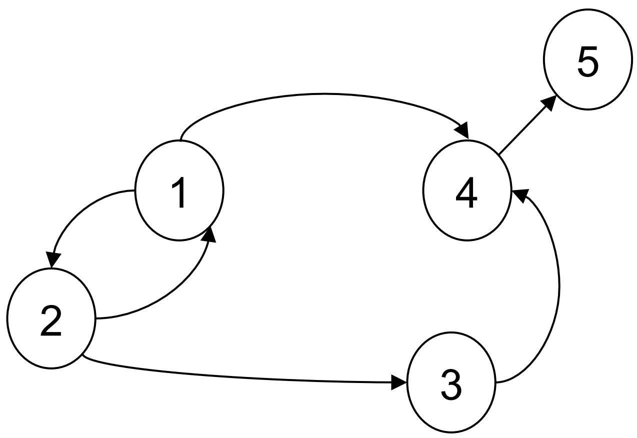
A relation: Edge

src	dst
1	2
2	1
2	3
1	4
3	4
4	5

Find nodes at distance 2: $\{(x,z) | \exists y\ Edge(x,y) \wedge Edge(y,z)\}$

Graph Databases

A graph:



A relation: Edge

src	dst
1	2
2	1
2	3
1	4
3	4
4	5

Find nodes at distance 2: $\{(x, z) | \exists y\ Edge(x, y) \wedge Edge(y, z)\}$

```
SELECT DISTINCT e1.src as X, e2.dst as Z
FROM Edge e1, Edge e2
WHERE e1.dst = e2.src;
```

Crash Course in Formal Logic

- The Relational Data Model is ***founded*** on first order logic (”What goes around”)
- SQL was designed as a more friendly language than FO
- Complex SQL queries are sometimes best understood in the framework of FO

Crash Course in Formal Logic

Atomic predicates:

- Likes(x,y)
- Product(x,y,z)
  -- pid, name, color
- Product(x,y,'red')

Connectives: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$

Crash Course in Formal Logic

Atomic predicates:

- Likes(x,y)
- Product(x,y,z)
  -- pid, name, color
- Product(x,y,'red')

Connectives: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$

- $\exists x\ P(x)$:
  there exists x s.t. P(x) is true
- $\forall x\ P(x)$:
  for every x, P(x) is true

Crash Course in Formal Logic

Atomic predicates:

- Likes(x,y)
- Product(x,y,z)
  -- pid, name, color
- Product(x,y,'red')

Connectives: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$

- $\exists x\ P(x)$:
  there exists x s.t. P(x) is true
- $\forall x\ P(x)$:
  for every x, P(x) is true

What do these sentences say?

$\exists x(\text{Likes}(\text{'Alice'},x) \wedge \text{Likes}(\text{'Bob'},x))$

Crash Course in Formal Logic

Atomic predicates:

- Likes(x,y)
- Product(x,y,z)
  -- pid, name, color
- Product(x,y,'red')

Connectives: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$

- $\exists$x P(x):
  there exists x s.t. P(x) is true
- $\forall$x P(x):
  for every x, P(x) is true

What do these sentences say?

$\exists x(\text{Likes}(\text{'Alice'},x) \wedge \text{Likes}(\text{'Bob'},x))$

There is somebody liked by both Alice and Bob

Crash Course in Formal Logic

Atomic predicates:

- Likes(x,y)
- Product(x,y,z)
  -- pid, name, color
- Product(x,y,'red')

Connectives: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$

- $\exists x\ P(x)$:
  there exists x s.t. P(x) is true
- $\forall x\ P(x)$:
  for every x, P(x) is true

What do these sentences say?

$\exists x(\text{Likes}(\text{'Alice'},x) \wedge \text{Likes}(\text{'Bob'},x))$

There is somebody liked
by both Alice and Bob

$\forall x\ (\text{Likes}(\text{'Alice'},x) \Rightarrow \text{Likes}(\text{'Bob'},x))$

Crash Course in Formal Logic

Atomic predicates:

- Likes(x,y)
- Product(x,y,z)
  -- pid, name, color
- Product(x,y,'red')

Connectives: ∧, ∨, ¬, ⇒, ∃, ∀

- ∃x P(x):
  there exists x s.t. P(x) is true
- ∀x P(x):
  for every x, P(x) is true

What do these sentences say?

∃x(Likes('Alice',x)∧Likes('Bob',x))

There is somebody liked by both Alice and Bob

∀x (Likes('Alice',x) ⇒Likes('Bob',x))

Everybody liked by Alice, is also liked by Bob

Crash Course in Formal Logic

Atomic predicates:

- Likes(x,y)
- Product(x,y,z)
  -- pid, name, color
- Product(x,y,'red')

Connectives: $\wedge$, $\vee$, $\neg$, $\Rightarrow$, $\exists$, $\forall$

- $\exists x\, P(x)$:
  there exists x s.t. P(x) is true
- $\forall x\, P(x)$:
  for every x, P(x) is true

What do these sentences say?

$\exists x(\text{Likes}(\text{'Alice'},x) \wedge \text{Likes}(\text{'Bob'},x))$

There is somebody liked by both Alice and Bob

$\forall x\, (\text{Likes}(\text{'Alice'},x) \Rightarrow \text{Likes}(\text{'Bob'},x))$

Everybody liked by Alice, is also liked by Bob

$\forall x\, (\exists y\, \text{Likes}(x,y) \Rightarrow \text{Likes}(x,\text{'Alice'}))$

Crash Course in Formal Logic

Atomic predicates:

- Likes(x,y)
- Product(x,y,z)
  -- pid, name, color
- Product(x,y,’red’)

Connectives: ∧, ∨, ¬, ⇒, ∃, ∀

- ∃x P(x):
  there exists x s.t. P(x) is true
- ∀x P(x):
  for every x, P(x) is true

What do these sentences say?

∃x(Likes(‘Alice’,x)∧Likes(‘Bob’,x))

There is somebody liked by both Alice and Bob

∀x (Likes(‘Alice’,x) ⇒Likes(‘Bob’,x))

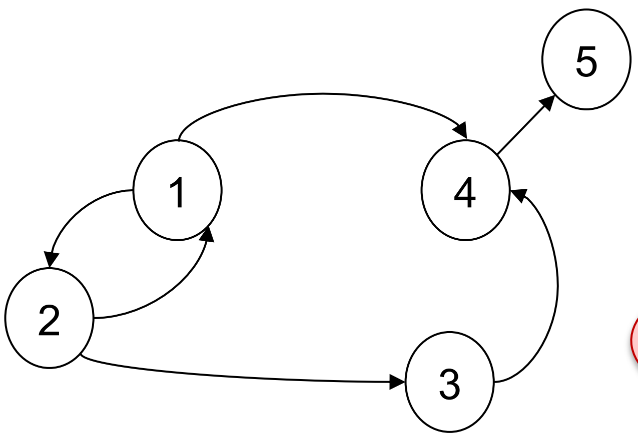
Everybody liked by Alice, is also liked by Bob

∀x (∃y Likes(x,y) ⇒ Likes(x,‘Alice’))

Everybody who likes somebody also likes Alice

Graph Databases

A graph:



A relation: Edge

src	dst
1	2
2	1
2	3
1	4
3	4
4	5

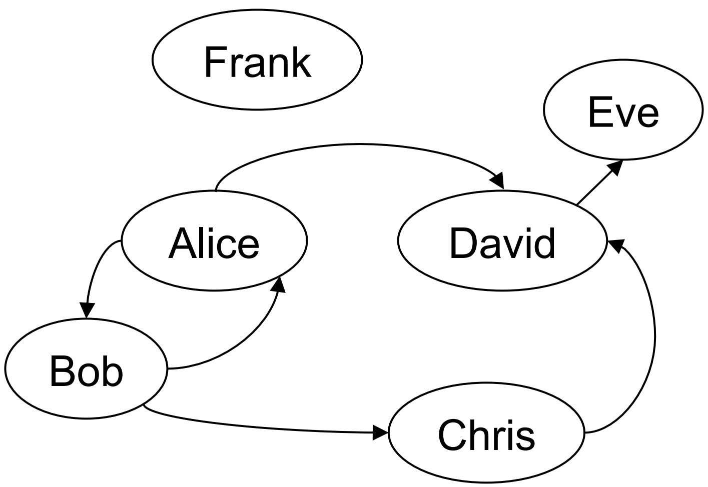
Now this should be clear

Find nodes at distance 2: $\{(x,z) | \exists y \; Edge(x,y) \wedge Edge(y,z)\}$

```
SELECT DISTINCT e1.src as X, e2.dst as Z
FROM Edge e1, Edge e2
WHERE e1.dst = e2.src;
```

Other Representation

Representing nodes separately;
needed for “isolated nodes” e.g. Frank



Node

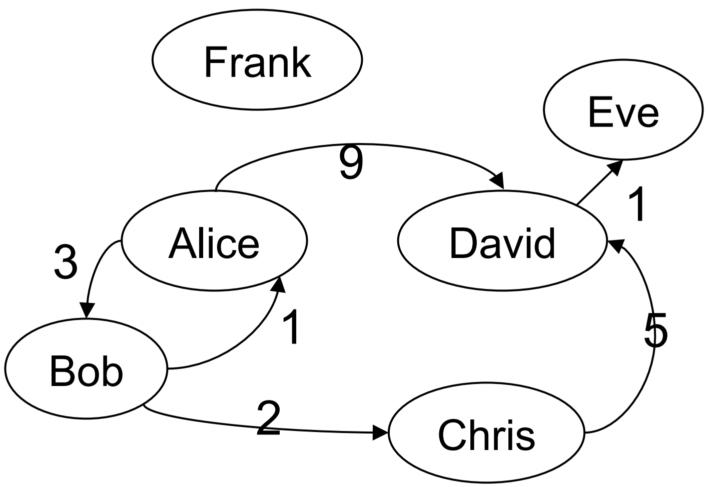
src
Alice
Bob
Chris
David
Eve
Frank

Edge

src	dst
Alice	Bob
Bob	Alice
Bob	Chris
Alice	David
Chris	David
David	Eve

Other Representation

Adding edge labels
Adding node labels…



Node

src
Alice
Bob
Chris
David
Eve
Frank

Edge

src	dst	weight
Alice	Bob	3
Bob	Alice	1
Bob	Chris	2
Alice	David	9
Chris	David	5
David	Eve	1

Limitations of SQL

- No recursion! Examples requiring recursion:
  - Gradient descent
  - Connected components in a graph
- Advanced systems *do* support recursion
- Practical solution: use some external driver, e.g. pyton

Example: Logistic Regression

Tom Mitchell: Machine Learning

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	0
3	6	3	0
5	5	9	1
9	3	3	1
…	…	…	
…	…	…	

Example: Logistic Regression

Tom Mitchell: [Machine Learning]

Switched
(following Mitchell)

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	0
3	6	3	0
5	5	9	1
9	3	3	1
…	…	…	
…	…	…	

$$P(Y = 0|X) = \frac{1}{1 + exp\left(w_0 + \sum_{i=1,3} w_i X_i\right)}$$

$$P(Y = 1|X) = \frac{exp\left(w_0 + \sum_{i=1,3} w_i X_i\right)}{1 + exp\left(w_0 + \sum_{i=1,3} w_i X_i\right)}$$

Example: Logistic Regression

Tom Mitchell: Machine Learning

Switched (following Mitchell)

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	0
3	6	3	0
5	5	9	1
9	3	3	1
…	…	…	
…	…	…	

$$P(Y=0|X) = \frac{1}{1+exp\left(w_0 + \sum_{i=1,3} w_i X_i\right)}$$

$$P(Y=1|X) = \frac{exp\left(w_0 + \sum_{i=1,3} w_i X_i\right)}{1+exp\left(w_0 + \sum_{i=1,3} w_i X_i\right)}$$

Train weights w_0, w_1, w_2, w_3 to minimize loss:

$$L(w_0, \dots, w_3) = \sum_{\ell=1,N} \left(Y^\ell \cdot \ln P\left(Y=1|X^\ell\right) + (1-Y^\ell) \cdot \ln P\left(Y=0|X^\ell\right)\right)$$

Example: Logistic Regression

Tom Mitchell: Machine Learning

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	0
3	6	3	0
5	5	9	1
9	3	3	1
…	…	…	
…	…	…	

Gradient Descent:

$$w_i \leftarrow w_i + \eta \sum_{\ell=1,N} X_i^\ell \left(Y^\ell - P(Y = 1 | X^\ell)\right)$$

Example: Logistic Regression

Tom Mitchell: Machine Learning

Gradient Descent:

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	0
3	6	3	0
5	5	9	1
9	3	3	1
…	…	…	
…	…	…	

$$w_i \leftarrow w_i + \eta \sum_{\ell=1,N} X_i^{\ell}\left(Y^{\ell} - P(Y = 1|X^{\ell})\right)$$

```
CREATE TABLE W (k int primary key, w0 real, w1 real, w2 real, w3 real);
INSERT INTO W VALUES (1, 0, 0, 0, 0);
```

Example: Logistic Regression

Tom Mitchell: Machine Learning

Gradient Descent:

$$w_i \leftarrow w_i + \eta \sum_{\ell=1,N} X_i^{\ell}\left(Y^{\ell} - P(Y = 1 | X^{\ell})\right)$$

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	[illegible]
3	6	3	[illegible]

```
CREATE TABLE W (k int primary key, w0 real, w1 real, w2 real, w3 real);
INSERT INTO W VALUES (1, 0, 0, 0, 0);
```

```
FROM data d, W
WHERE W.k=1
```

Example: Logistic Regression

Tom Mitchell: Machine Learning

Gradient Descent:

$$w_i \leftarrow w_i + \eta \sum_{\ell=1,N} X_i^\ell \left(Y^\ell - P(Y=1|X^\ell)\right)$$

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	[illegible]
3	6	3	[illegible]

```
CREATE TABLE W (k int primary key, w0 real, w1 real, w2 real, w3 real);
INSERT INTO W VALUES (1, 0, 0, 0, 0);
```

```
SELECT
  W.w0+0.01*sum(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3))) as w0,




FROM data d, W
WHERE W.k=1
```

Example: Logistic Regression

Tom Mitchell: Machine Learning

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	
3	6	3	

Gradient Descent:

$$w_i \leftarrow w_i + \eta \sum_{\ell=1,N} X_i^\ell \left(Y^\ell - P(Y=1|X^\ell)\right)$$

```
CREATE TABLE W (k int primary key, w0 real, w1 real, w2 real, w3 real);
INSERT INTO W VALUES (1, 0, 0, 0, 0);
```

```
SELECT
  W.w0+0.01*sum(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3))) as w0,
  W.w1+0.01*sum(d.X1*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w1,



FROM data d, W
WHERE W.k=1
```

Example: Logistic Regression

Tom Mitchell: Machine Learning

Gradient Descent:

$$w_i \leftarrow w_i + \eta \sum_{\ell=1,N} X_i^\ell \left(Y^\ell - P(Y=1|X^\ell)\right)$$

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	[illegible]
3	6	3	[illegible]

```
CREATE TABLE W (k int primary key, w0 real, w1 real, w2 real, w3 real);
INSERT INTO W VALUES (1, 0, 0, 0, 0);
```

```
SELECT
  W.w0+0.01*sum(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3))) as w0,
  W.w1+0.01*sum(d.X1*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w1,
  W.w2+0.01*sum(d.X2*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w2,
  W.w3+0.01*sum(d.X3*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w3
FROM data d, W
WHERE W.k=1
```

Example: Logistic Regression

Tom Mitchell: Machine Learning

Gradient Descent:

$$w_i \leftarrow w_i + \eta \sum_{\ell=1,N} X_i^\ell \left(Y^\ell - P(Y=1|X^\ell)\right)$$

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	[illegible]
3	6	3	[illegible]

```
CREATE TABLE W (k int primary key, w0 real, w1 real, w2 real, w3 real);
INSERT INTO W VALUES (1, 0, 0, 0, 0);
```

```
SELECT
  W.w0+0.01*sum(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3))) as w0,
  W.w1+0.01*sum(d.X1*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w1,
  W.w2+0.01*sum(d.X2*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w2,
  W.w3+0.01*sum(d.X3*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w3
FROM data d, W
WHERE W.k=1
GROUP BY W.k, W.w0, W.w1, W.w2, W.w3;
```

Example: Logistic Regression

Tom Mitchell: Machine Learning

Data

X1	X2	X3	Y
3	9	3	0
3	5	7	1
6	2	2	0
3	6	3	0

Gradient Descent:

$$w_i \leftarrow w_i + \eta \sum_{\ell=1,N} X_i^\ell \left(Y^\ell - P(Y=1|X^\ell)\right)$$

```
CREATE TABLE W (k int primary key, w0 real, w1 real, w2 real, w3 real);
INSERT INTO W VALUES (1, 0, 0, 0, 0);
```

```
SELECT
  W.w0+0.01*sum(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3))) as w0,
  W.w1+0.01*sum(d.X1*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w1,
  W.w2+0.01*sum(d.X2*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w2,
  W.w3+0.01*sum(d.X3*(d.Y - 1 + 1/(1+exp(W.w0+W.w1*d.X1+W.w2*d.X2+W.w3*d.X3)))) as w3
FROM data d, W
WHERE W.k=1
GROUP BY W.k, W.w0, W.w1, W.w2, W.w3;
```

Update W, then repeat this
e.g. using python

Discussion

SQL in Data Science:

- Used primarily to prepare the data
  - ETL – Extract/Transform/Load
  - Join tables, process columns, filter rows
- Can also be used in training
  - Much less convenient than ML packages
  - But can be the best option if data is huge

More To Know About SQL

- create table
- help
- create view
- create index
- explain
- insert into, delete from, update set

Create Table

```
CREATE TABLE  User (
        uid int PRIMARY KEY,
        firstName text,
        lastName text NOT NULL,
        age int CHECK (age > 12 and age < 120),
        email text,
        phone text,
        FOREIGN KEY (email, phone) REFERENCES Accnt
)
```

Primary key constraint

Attribute-level constraint

Composite foreign key constraint

Create Table

Hints for HW1:

- Constraints are *good*:
  - they keep the data clean
  - But they make uploads SOOO slow
- Hint: use this order
  - Create table
  - Upload data (COPY…)
  - ALTER TALBE … (add constraints)
  - If error, use SQL to debug!

Help

Postgres

- \help
- \help ALTER TABLE
- \?

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

Create View

- Need to write same SQL expression repeatedly? Create a view, then use it:

```
create view SeattleSupplierRed as
  select distinct x.*
  from Supplier x, Supply y, Part z
  where x.sno=y.sno and y.pno=z.pno
    and x.scity='Seatte'
    and z.pcolor='red'
```

```
select y.pno, y.price
from SeattleSupplierRed x
        Supply y
where x.sno=y.sno
```

View Variants

- CREATE TEMPORARY VIEW name…
- Not stored in the catalog

- WITH name AS (SELECT…)
  SELECT … FROM … WHERE…
- Used only within one query

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

Create Index

- Index = auxiliary file that helps speedup some queries
- create index

```
create index Supplier_scity
         on Supplier(scity);
```

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

Create Index

- Index = auxiliary file that helps speedup some queries
- create index

```
create index Supplier_scity
        on Supplier(scity);
create index Supplier_sstate_sname
        on Supplier(sstate,sname);
create index Supply_sno
        on Supply(sno);
```

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

Create Index

- Index = auxiliary file that helps speedup some queries
- create index

```
create index Supplier_scity
        on Supplier(scity);
create index Supplier_sstate_sname
        on Supplier(sstate,sname);
create index Supply_sno
        on Supply(sno);
cluster Supply using Supply_sno;
```

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

Create Index

- Index = auxiliary file that helps speedup some queries
- create index

```
create index Supplier_scity
        on Supplier(scity);
create index Supplier_sstate_sname
        on Supplier(sstate,sname);
create index Supply_sno
        on Supply(sno);
cluster Supply using Supply_sno;
```

```
select * from Supplier
where scity='Seattle'
```

Big speedup from Supplier_city

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

Create Index

- Index = auxiliary file that helps speedup some queries
- create index

```
create index Supplier_scity
        on Supplier(scity);
create index Supplier_sstate_sname
        on Supplier(sstate,sname);
create index Supply_sno
        on Supply(sno);
cluster Supply using Supply_sno;
```

```
select * from Supplier
where scity='Seattle'
```

Big speedup from Supplier_city

```
select *
from Supplier x, Supply y
where x.sno = y.sno
   and sname = 'iPad'
```

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

Create Index

- Index = auxiliary file that helps speedup some queries
- create index

```
create index Supplier_scity
        on Supplier(scity);
create index Supplier_sstate_sname
        on Supplier(sstate,sname);
create index Supply_sno
        on Supply(sno);
cluster Supply using Supply_sno;
```

```
select * from Supplier
where scity='Seattle'
```

Big speedup from Supplier_city

```
select *
from Supplier x, Supply y
where x.sno = y.sno
   and sname = 'iPad'
```

Big speedup from Supply_sno

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

Create Index

- Index = auxiliary file that helps speedup some queries
- create index

```
create index Supplier_scity
        on Supplier(scity);
create index Supplier_sstate_sname
        on Supplier(sstate,sname);
create index Supply_sno
        on Supply(sno);
cluster Supply using Supply_sno;
```

```
select * from Supplier
where scity='Seattle'
```

Big speedup from Supplier_city

```
select *
from Supplier x, Supply y
where x.sno = y.sno
   and sname = 'iPad'
```

Big speedup from Supply_sno

```
select *
from Supplier x, Supply y
where x.sno = y.sno
```

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

Create Index

- Index = auxiliary file that helps speedup some queries
- create index

```
create index Supplier_scity
        on Supplier(scity);
create index Supplier_sstate_sname
        on Supplier(sstate,sname);
create index Supply_sno
        on Supply(sno);
cluster Supply using Supply_sno;
```

```
select * from Supplier
where scity='Seattle'
```

Big speedup from Supplier_city

```
select *
from Supplier x, Supply y
where x.sno = y.sno
  and sname = 'iPad'
```

Big speedup from Supply_sno

Unlikely benefit (discuss clustered)

```
select *
from Supplier x, Supply y
where x.sno = y.sno
```

Create Index

Hints for HW1

- Indexes are *great* for speeding up queries
- But they make uploads SOOO slow!
- Hint: upload first, create index later

Explain

Postgres:

- explain select * from Supplier where scity=‘Seattle’
- Checkout: \h explain
- Other systems have similar commands:
  use it frequently to understand the query plan

Update Commands

- insert into Product values (33,’iPad’,…);
- insert into NewTable (select * from…);
- delete from Product where price > 100;

Update Commands

- insert into Product values (33,’iPad’,…);
- insert into NewTable (select * from…);
- delete from Product where price > 100;
- delete from Product; -- don’t do this!

Update Commands

- insert into Product values (33,'iPad',…);
- insert into NewTable (select * from…);
- delete from Product where price > 100;
- delete from Product; -- don't do this!
- update Product
  set price = 99
  where price > 100

SQL – Summary

- Very complex: >1000 pages,
  - No vendor supports full standard; (in practice, people use postgres as *de facto* standard)
  - Much more than DML
- It is a *declarative* language:
  - we say what we want
  - we don't say how to get it
- Relational algebra says how to get it

Relational Algebra

- Queries specified in an operational manner
  - A query gives a step-by-step procedure
- Relational operators
  - Take one or two relation instances as input
  - Return one relation instance as result
  - Easy to compose into relational algebra expressions

Five Basic Relational Operators

- Selection: $\sigma_{condition}(S)$
  - Condition is Boolean combination ($\wedge$,$\vee$) of atomic predicates (<, <=, =, $\neq$, >=, >)
- Projection: $\pi_{list\text{-}of\text{-}attributes}(S)$
- Union ($\cup$)
- Set difference (–),
- Cross-product/cartesian product ($\times$), Join: $R \bowtie_{\theta} S = \sigma_{\theta}(R \times S)$

Other operators: anti-semijoin, renaming

Extended Operators

- Duplicate elimination (δ)
  - Since commercial DBMSs operate on multisets not sets
- Group-by/aggregate (γ)
  - Min, max, sum, average, count
  - Partitions tuples of a relation into “groups”
  - Aggregates can then be applied to groups
- Sort operator (τ)

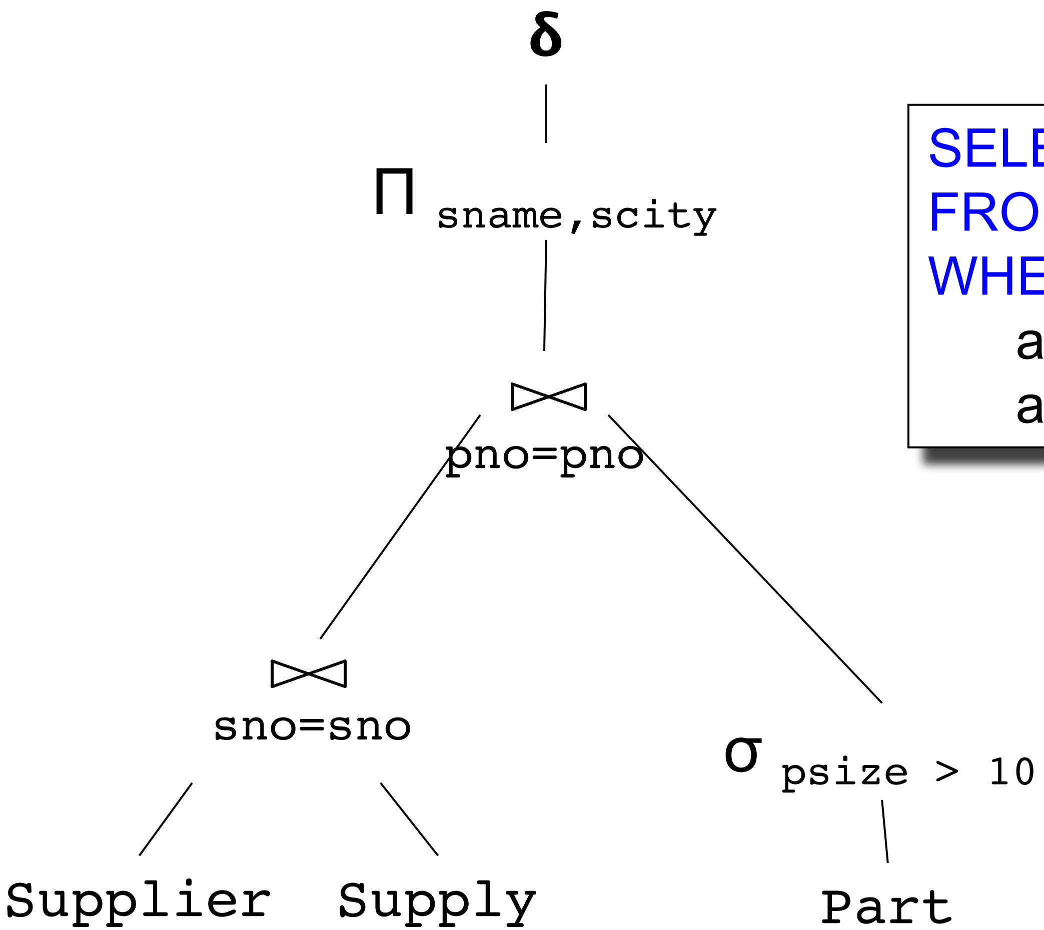
```
Supplier(sno,sname,scity,sstate)
Supply(sno,pno,qty,price)
Part(pno,pname,psize,pcolor)
```

Logical Query Plans

```
SELECT DISTINCT x.sname, x.scity
FROM Supplier x, Supply y, Part z
WHERE x.sno=y.sno
    and  y.pno=z.pno
    and z.psize > 10;
```

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

Logical Query Plans




```
SELECT DISTINCT x.sname, x.scity
FROM Supplier x, Supply y, Part z
WHERE x.sno=y.sno
     and  y.pno=z.pno
     and z.psize > 10;
```

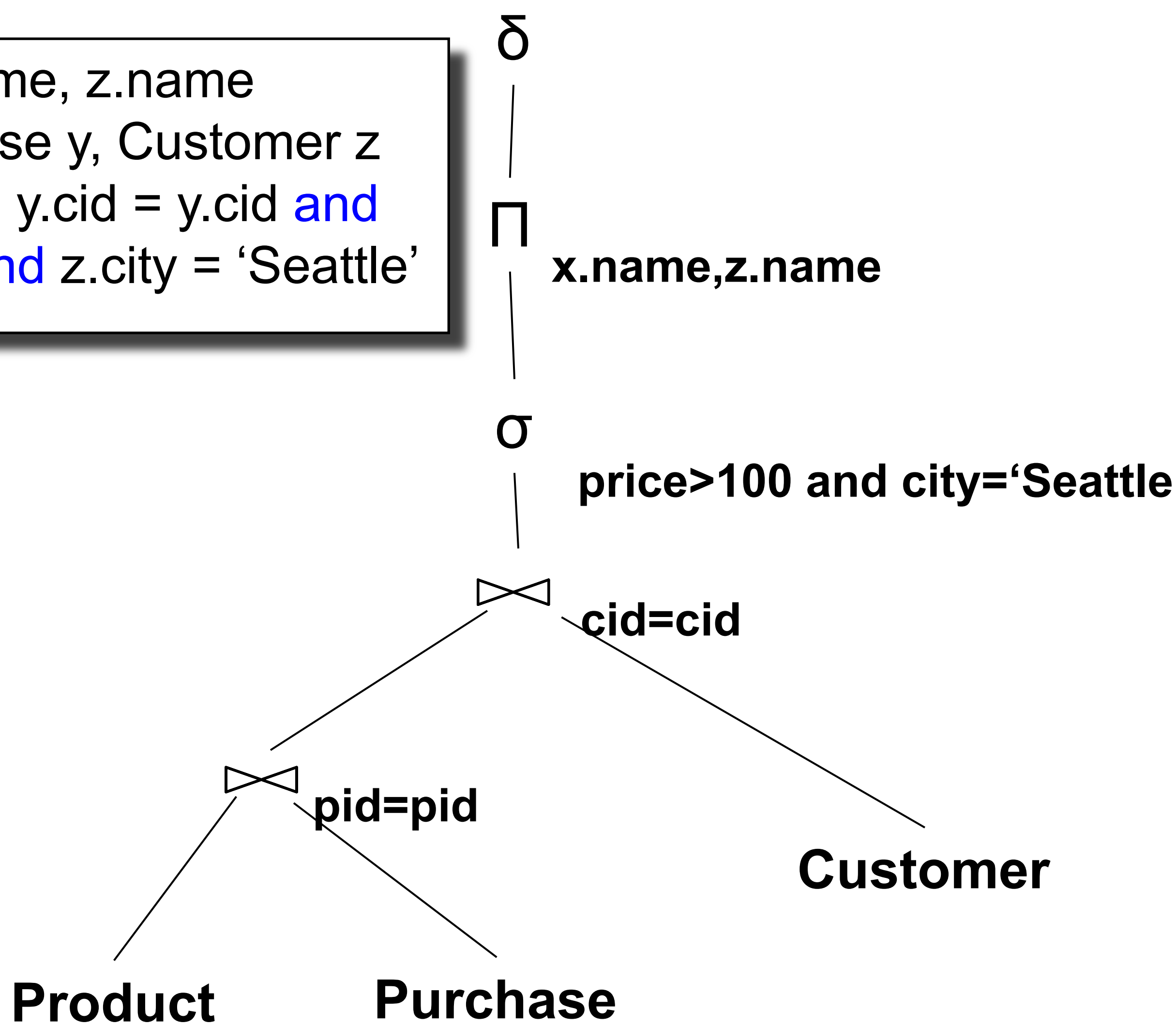
Query Optimizer

- Rewrite one relational algebra expression to a better one
- Very brief review now, more details next lectures

Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

Optimization

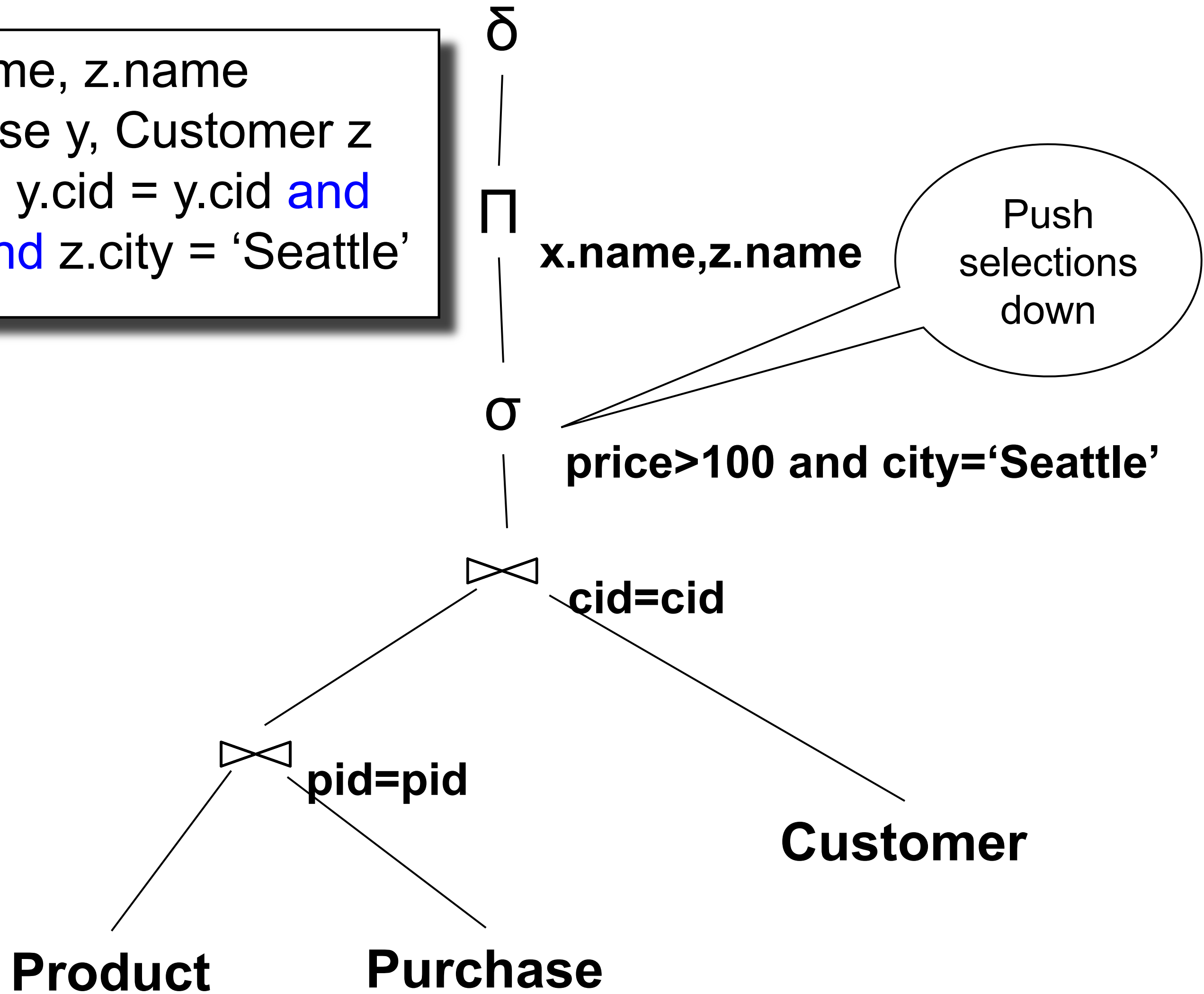
```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
      x.price > 100 and z.city = ‘Seattle’
```



Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

Optimization

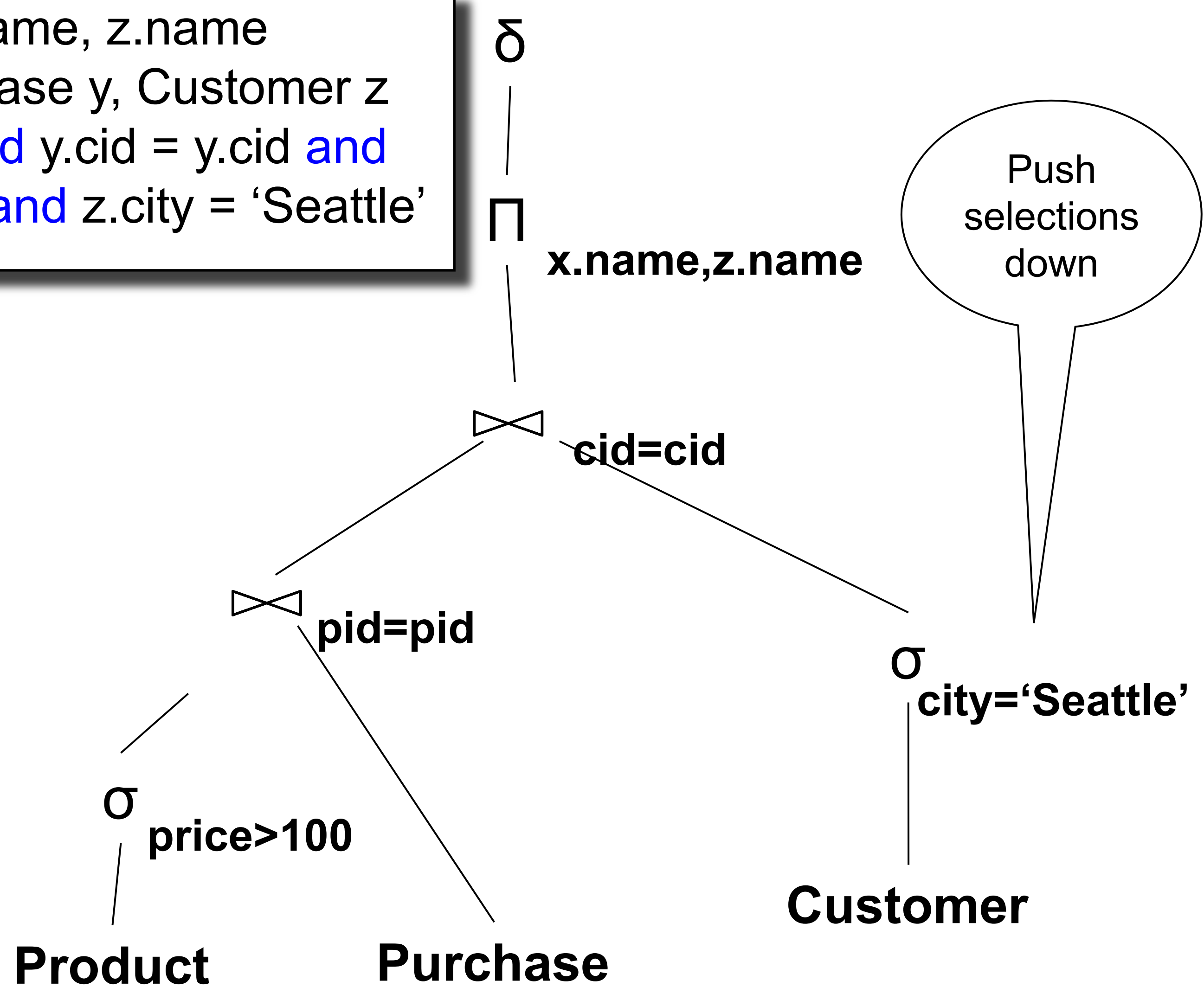
```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
        x.price > 100 and z.city = 'Seattle'
```



Product(pid, name, price)
Purchase(pid, cid, store)
Customer(cid, name, city)

Optimization

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = y.cid and
         x.price > 100 and z.city = 'Seattle'
```



More about this in future lectures

Benefits of Relational Model

- Physical data independence
  - Can change how data is organized on disk without affecting applications
- Logical data independence
  - Can change the logical schema without affecting applications (not 100%... consider updates)

Physical Data Independence

Supplier

sno	sname	scity	sstate
1	s1	city 1	WA
2	s2	city 1	WA
3	s3	city 2	MA
4	s4	city 2	MA

```
SELECT DISTINCT sname
FROM Supplier
WHERE scity = 'Seattle'
```

How is the data stored on disk?
(e.g. row-wise, column-wise)

The SQL query works the same, regardless of the answers to these questions

Is there an index on scity?
(e.g. no index, unclustered index, clustered index)

Lecture on Wednesday

- Data model – what's so hard about it?
- Review "What goes around…