Principles of Database Systems
CSE 544

Lecture #1

Introduction and SQL

Staff

- Instructor: Dan Suciu
  - CSE 662, suciu@cs.washington.edu
  - Office hour: Tuesdays, 5:30-6:20, CSE 662

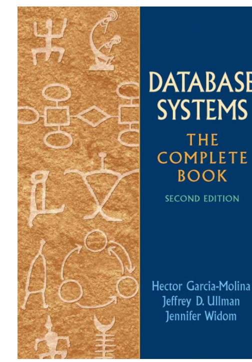
- TA:
  - Priya Rao Chagaleti, priyarao@cs.washington.edu
  - Office hours: Monday, 5-6pm, Office TBA

Class Format

- Lectures Tuesdays, 6:30-9:20 – video archived
- 6 Homework Assignments
- 9 Reading assignments
- An online, take-home final (two days)

Textbook and Papers

- Main Textbook:
  - Database Management Systems. **3rd Ed.**, by Ramakrishnan and Gehrke. McGraw-Hill.
  - Book available on the Kindle too
  - Use it to read background material
  - You may borrow it, no need to buy



- Optional Textbook
  - Database Systems: The Complete Book, by Garcia-Molina, Ullman, Widom
- Other Books
  - Foundations of Databases, by Abiteboul, Hull, Vianu

Textbook and Papers

- Nine papers to read and review
  - Three are short blogs (Stonebraker)
  - Five are real papers
  - All papers are available from the course Website with your CSE or UWID credentials
  - Most are also available online, and on Kindle

Resources

- Web page: http://courses.cs.washington.edu/courses/csep544/14wi/
  - Lectures
  - Homework assignments
  - Reading assignments
  - Information about the final
- Mailing list:
  - Announcements, group discussions
- Discussion board:
  - Feel free to post; the TA will check regularly

Content of the Class

- Relational Data Model
  - SQL, Data Models, Relational calculus
- Database internals
  - Storage, query execution/optimization, statistics,
- Parallel databases and MapReduce
- Transactions
  - Recovery (Aries), Concurrency control
- Advanced Topics
  - Datalog
  - NoSQL, ColumnStore

Evaluation

- Homework Assignments 50%:
  - Four light programming, two theory
- Paper reviews 20%:
  - Nine reviews, about ½ page each
- Final exam 30%:
  - Take home, online exam
  - Two days: Saturday/Sunday, March 15/16

Homework Assignments 50%

• HW1: SQL	programming
• HW2: RC/RA, DB Design	theory
• HW3: PigLatin on AWS	programming
• HW4: DB Application	programming
• HW5: Transactions	theory
• HW6: XML	programming

Assignments 50%

HW1: SQL – posted!

- Three Tasks:
  - Create tables
  - Create indexes
  - Compute 11 SQL Queries
- Dataset = a copy of IMDB from 2010
- Tools:
  - Install a DMBS on your machine: either Postgres or SQL Server.
  - SQL Azure: login=your email@washington.edu; password=[in class]

Extra credit = download a current copy of IMDB

Due: Monday, January 20

Paper Reviews

- Papers:
  - Three short blogs by Stonebraker
  - Six systems-research papers

- Reviews:
  - Due Tuesdays, before class
  - Review should be a brief (½ page) summary of the lessons you learned from the paper

Final

Format

- Take-home, online final
- Posted online on Saturday, March 15, at 12:00am
- Answers by Sunday, March 16, at 11:59pm
  - Note: remember to push “Submit”!
- No late days/hours/minutes/seconds

Goals of the Class

This is a graduate level class!

- Deep understanding of relational calculus:
  - Complex SQL queries, RC, RA
  - Full appreciation of the data independence principle
- Some discussion of database internals
- Parallel data processing:
  - Parallel query processing of relational operators
  - MapReduce
  - A deep understanding of “SQL is embarrassingly parallel”
- Transactions:
  - ARIES!
  - Pessimistic and optimistic concurrency control (MVCC)
- Advanced topics:
  - ColumnStores, NoSQL (NewSQL?)

Background

You should have heard about most of:

- E/R diagrams
- Normal forms (1^{st}, 3^{rd})
- SQL
- Relational Algebra
- Indexes, search trees
- Search in a binary tree
- Query optimization (e.g. join reordering)
- Transactions (e.g. ACID)
- Logic: $\wedge$, $\vee$, $\forall$, $\exists$, $\neg$, $\in$
- Reachability in a graph

We will cover these topics in class, but assume some background

Agenda for Today

- Brief overview of a traditional database systems

- SQL: Chapters 5.2 – 5.6 in the textbook

Databases

What is a database ?

Give examples of databases

Databases

What is a database ?

- A collection of files storing related data

Give examples of databases

- Accounts database; payroll database; UW's students database; Amazon's products database; airline reservation database

Database Management System

What is a DBMS ?

Give examples of DBMS

Database Management System

What is a DBMS ?

- A big C program written by someone else that allows us to manage efficiently a large database and allows it to persist over long periods of time

Give examples of DBMS

- DB2 (IBM), SQL Server (MS), Oracle, Sybase
- MySQL, Postgres, …

Market Shares

From 2006 Gartner report:

- IBM: 21% market with $3.2BN in sales
- Oracle: 47% market with $7.1BN in sales
- Microsoft: 17% market with $2.6BN in sales

An Example

The Internet Movie Database
http://www.imdb.com

- Entities:
  Actors (1.5M), Movies (1.8M), Directors

- Relationships:
  who played where, who directed what, …

Tables

Actor:

id	fName	lName	gender
195428	Tom	Hanks	M
645947	Amy	Hanks	F
. . .			

Casts:

pid	mid
195428	337166
. . .	

Movie:

id	Name	year
337166	Toy Story	1995
. . .	. . .	. ..

SQL

```
SELECT *
FROM  Actor
```

SQL

```
SELECT *
FROM  Actor
```

↓

```
SELECT *
FROM  Actor
LIMIT 50
```

SQL

```
SELECT *
FROM  Actor
```

```
SELECT *
FROM  Actor
LIMIT 50
```

```
SELECT count(*)
FROM  Actor
```

SQL

```
SELECT *
FROM  Actor
```

```
SELECT *
FROM  Actor
LIMIT 50
```

```
SELECT count(*)
FROM  Actor
```

```
SELECT *
FROM  Actor
WHERE lName = 'Hanks'
```

SQL

```
SELECT *
FROM  Actor x, Casts y, Movie z
WHERE x.lname='Hanks'
      and x.id = y.pid
      and y.mid=z.id
      and z.year=1995
```

This query has *selections* and *joins*

1.8M actors, 11M casts, 1.5M movies;
How can it be so fast ?

How Can We Evaluate the Query ?

Actor:

id	fName	lName	gender
. . .		Hanks	
. . .			

1.8M actors

Casts:

pid	mid
. . .	
. . .	

11M casts

Movie:

id	Name	year
. . .		1995
. . .		

1.5M movies

```
SELECT *
FROM  Actor x, Casts y, Movie z
WHERE x.lname='Hanks'
      and x.id = y.pid
      and y.mid=z.id
      and z.year=1995
```

How Can We Evaluate the Query ?

Actor:

id	fName	lName	gender
. . .		Hanks	
. . .			

1.8M actors

Casts:

pid	mid
. . .	
. . .	

11M casts

Movie:

id	Name	year
. . .		1995
. . .		

1.5M movies

```
SELECT *
FROM  Actor x, Casts y, Movie z
WHERE x.lname='Hanks'
      and x.id = y.pid
      and y.mid=z.id
      and z.year=1995
```

Plan 1: . . . . [in class]

Plan 2: . . . . [in class]

Evaluating Tom Hanks

Classical query execution

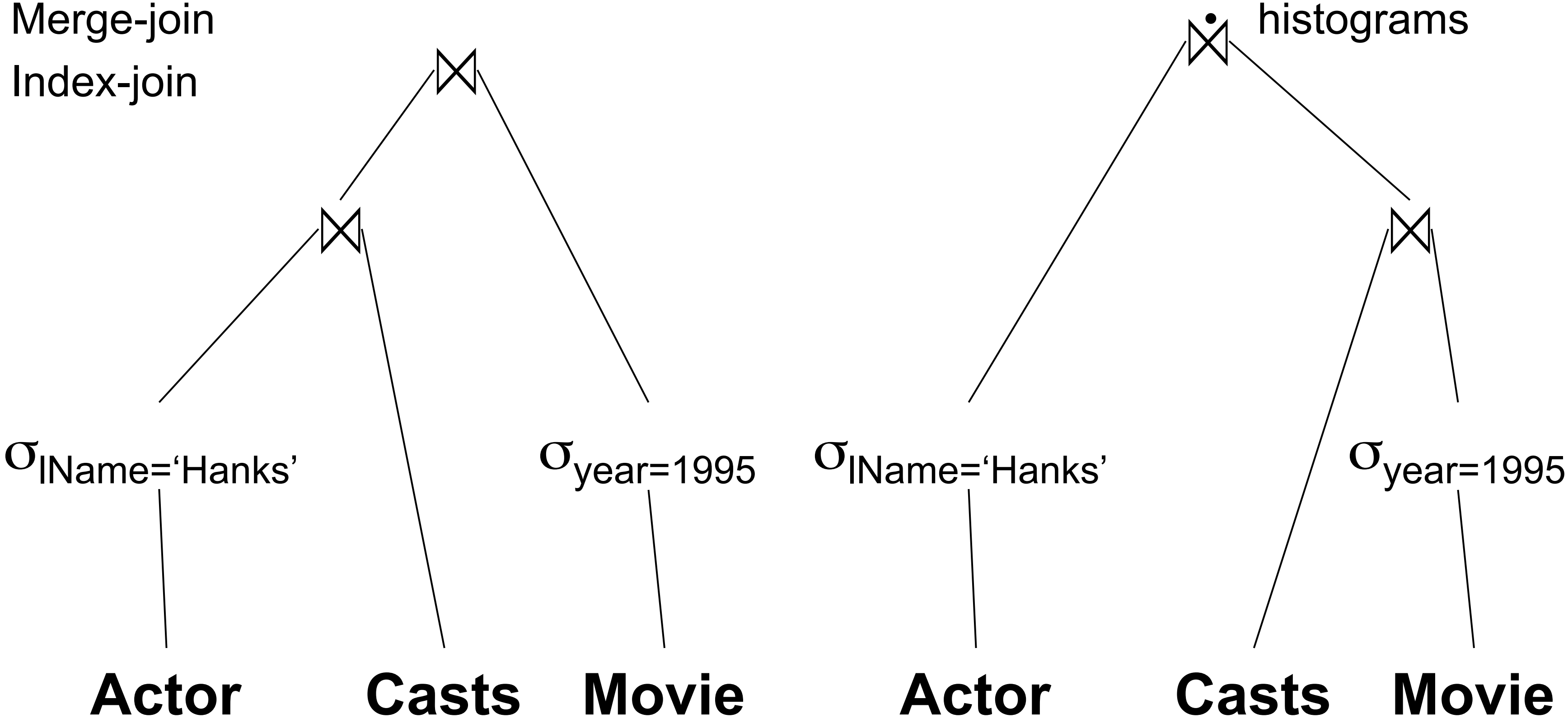
- Index-based selection
- Hash-join
- Merge-join
- Index-join

Classical query optimizations:

- Pushing selections down
- Join reorder

Classical statistics

- Table cardinalities
- # distinct values
- histograms



Terminology for Query Workloads

- OLTP (OnLine-Transaction-Processing)
  - Many updates: transactions are critical
  - Many “point queries”: access record by key
  - Commercial applications

- Decision-Support
  or OLAP (Online Analytical Processing)
  - Many aggregate/group-by queries.
  - Sometimes called *data warehouse*
  - Data analytics

Physical Data Independence

Physical data independence:

- Applications should be isolated from changes to the physical organization
- E.g. add/drop index

Physical Data Independence

Physical data independence:

- Applications should be isolated from changes to the physical organization
- E.g. add/drop index
- E.g. Different storage organization:

Physical Data Independence

Physical data independence:

- Applications should be isolated from changes to the physical organization
- E.g. add/drop index
- E.g. Different storage organization:

(Actor,Movie*)*

A1	M1	M2	M3	A2	M4	M5	A3	M6	M7	…

Physical Data Independence

Physical data independence:

- Applications should be isolated from changes to the physical organization
- E.g. add/drop index
- E.g. Different storage organization:

(Actor,Movie*)*

A1	M1	M2	M3	A2	M4	M5	A3	M6	M7	…

(Movie,Actor*)*

M1	A1	A2	M2	A3	A4	M3	A5	A6	A7	…

Physical Data Independence

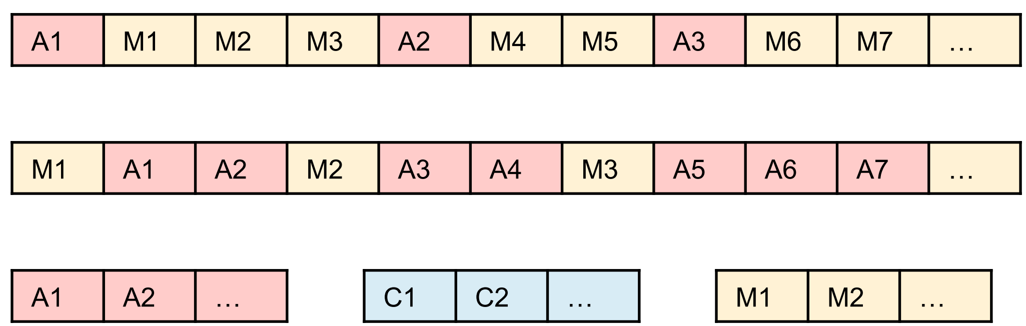
Physical data independence:

- Applications should be isolated from changes to the physical organization
- E.g. add/drop index
- E.g. Different storage organization:

(Actor,Movie*)*

(Movie,Actor*)*

(Movie*, Casts*, Actor*)



Physical Data Independence

Query optimizer = Translate WHAT to HOW:

- SQL = WHAT we want = declarative
- Relational algebra = HOW to get it = algorithm
- RDBMS are about translating WHAT to HOW

Transactions

- Recovery + Concurrency control
- ACID =
  - Atomicity (= recovery)
  - Consistency
  - Isolation (= concurrency control)
  - Durability

Client/Server Architecture

- One server: stores the database
  - called DBMS or RDBMS
  - Usually a beefed-up system:
    - Can be cluster of servers, or parallel DBMS
    - In 544 you will install the postgres server on your own computer
- Many clients: run apps and connect to DBMS
  - Interactive: psql (postgres), Management Studio (SQL Server)
  - Java/C++/C#/… applications
  - Connection protocol: ODBC/JDBC
- Exceptions exists; e.g. SQL Lite
- Three-tier architecture: add the app server

SQL

- Will cover SQL rather quickly today

- Resources for learning SQL:
  - The slides
  - The textbook
  - SQL Server help
  - Postgres help: type \h or \?

- Start working on HW1 !

SQL

- Data Manipulation Language (DML)
  - Querying: SELECT-FROM-WHERE
  - Modifying: INSERT/DELETE/UPDATE
- Data Definition Language (DDL)
  - CREATE/ALTER/DROP
  - Constraints: will discuss these in class

Tables in SQL

Table name

Attribute names

Key

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Tuples or rows

Creating Tables, Importing Data

```
CREATE TABLE Product (
   pname varchar(10) primary key,
   price float,
   category char(20),
   manufacturer text
);
```

```
INSERT INTO Product VALUES ('Gizmo', 19.99, 'Gadgets','GizmoWorks');
INSERT INTO Product VALUES ('Powergizmo',29.99,'Gadgets','GizmoWorks');
INSERT INTO Product VALUES ('SingleTouch',149.99,'Photography','Canon');
INSERT INTO Product VALUES ('MultiTouch', 203.99,'Household','Hitachi');
```

Better: bulk insert (but database specific!)

```
COPY Product FROM '/my/directory/datafile.txt'; -- postgres only!
```

Other Ways to Bulk Insert

```
CREATE TABLE Product (
   pname varchar(10) primary key,
   price float,
   category char(20),
   manufacturer text
);
```

```
INSERT into Product (
     SELECT …
     FROM …
     WHERE…
);
```

Quick method: create AND insert

```
CREATE TABLE Product AS
     SELECT …
     FROM …
     WHERE…
```

Data Types in SQL

- Atomic types:
  - Characters: CHAR(20), VARCHAR(50)
  - Numbers: INT, BIGINT, SMALLINT, FLOAT
  - Others: MONEY, DATETIME, …
  - Note: an attribute cannot be another table!
- Record (aka tuple)
  - Has atomic attributes
- Table (relation)
  - A set of tuples

No nested tables! (Discussion next…)

Normal Forms

- First Normal Form
  - All tables must be flat tables
  - Why?
- Boyce Codd Normal Form
  - The only functional dependencies are from a key
  - What is a “functional dependency”?
  - Why?
- Third Normal Form
  - The only functional dependencies are from keys, except … [boring technical condition here]
  - Why?

Normal Forms

- First Normal Form
  - All tables must be flat tables
  - Why? Physical data independence!
- Boyce Codd Normal Form
  - The only functional dependencies are from a key
  - What is a "functional dependency"?
  - Why? Avoid data anomalies (redundancy, update, delete)
- Third Normal Form
  - The only functional dependencies are from keys, except … [boring technical condition here]
  - Why? Because that's how we can recover all FD's.

Selections in SQL

```
SELECT   *
FROM     Product
WHERE    category='Gadgets'
```

Selections in SQL

```
SELECT   *
FROM     Product
WHERE    category='Gadgets'
```

```
SELECT   *
FROM     Product
WHERE    category > 'Gadgets'
```

Selections in SQL

```
SELECT   *
FROM     Product
WHERE    category='Gadgets'
```

```
SELECT   *
FROM     Product
WHERE    category LIKE 'Ga%'
```

```
SELECT   *
FROM     Product
WHERE    category > 'Gadgets'
```

Product (PName, price, category, manufacturer)

Selections in SQL

```
SELECT   *
FROM     Product
WHERE    category='Gadgets'
```

```
SELECT   *
FROM     Product
WHERE    category LIKE 'Ga%'
```

```
SELECT   *
FROM     Product
WHERE    category > 'Gadgets'
```

```
SELECT   *
FROM     Product
WHERE category LIKE '%dg%'
```

Projections (and Selections) in SQL

```
SELECT   pname
FROM     Product
WHERE    category='Gadgets'
```

Projections (and Selections) in SQL

```
SELECT   pname
FROM     Product
WHERE    category='Gadgets'
```

```
SELECT   category
FROM     Product
```

Projections (and Selections) in SQL

```
SELECT   pname
FROM     Product
WHERE    category='Gadgets'
```

```
SELECT   category
FROM     Product
```

Need DISTINCT (why?)

```
SELECT DISTINCT category
FROM     Product
```

“DISTINCT”, “ORDER BY”, “LIMIT”

```
SELECT   DISTINCT category
FROM     Product
```

```
SELECT   pname, price, manufacturer
FROM     Product
WHERE    category='gizmo' AND price > 50
ORDER BY price, pname
LIMIT 20
```

Postgres uses LIMIT k
SQL Server uses TOP k

Keys and Foreign Keys

Company

CName	StockPrice	Country
GizmoWorks	25	USA
Canon	65	Japan
Hitachi	15	Japan

Key

Product

PName	Price	Category	Manufacturer
Gizmo	$19.99	Gadgets	GizmoWorks
Powergizmo	$29.99	Gadgets	GizmoWorks
SingleTouch	$149.99	Photography	Canon
MultiTouch	$203.99	Household	Hitachi

Foreign key

Joins

Product (<u>PName</u>, Price, Category, Manufacturer)
Company (<u>CName</u>, stockPrice, Country)

Find all products under $200 manufactured in Japan;

```
SELECT   x.PName, x.Price
FROM     Product x, Company y
WHERE    x.Manufacturer=y.CName
     AND y.Country='Japan'
     AND x.Price <= 200
```

Semantics of SQL Queries

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

Semantics of SQL Queries

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

Subqueries

- A *subquery* or a *nested query* is another SQL query nested inside a larger query
- A subquery may occur in:

SELECT
FROM
} Examples at the end of the lecture

WHERE
} Examples on following slides

> Avoid writing nested queries when possible;
> keep in mind that sometimes it's impossible

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

Running Example

Run this in postgres, then try the examples on the following slides.

```
create table company(cname text primary key, city text);
create table product(pname text primary key, price int, company text references company);

insert into company values('abc', 'seattle');
insert into company values('cde', 'seattle');
insert into company values('fgh', 'portland');
insert into company values('klm', 'portland');

insert into product values('p1',  10, 'abc');
insert into product values('p2', 200, 'abc');
insert into product values('p3',  10, 'cde');
insert into product values('p4',  20, 'cde');

insert into product values('p5',  10, 'fgh');
insert into product values('p6', 200, 'fgh');
insert into product values('p7',  10, 'klm');
insert into product values('p8', 220, 'klm');
```

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

Existential Quantifiers

Find cities that have a company
that manufacture *some* product with price < 100

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

Existential Quantifiers

Find cities that have a company
that manufacture *some* product with price < 100

```
SELECT DISTINCT  c.city
FROM      Company c, Product p
WHERE   c.cname = p.company
    and  p.price < 100
```

Existential quantifiers are easy! ☺

Product (<u>pname</u>, price, company)
Company(<u>cname</u>, city)

Universal Quantifiers

Find cities that have a company
such that *all* its products have price < 100

Product (<u>pname</u>, price, company)
Company(<u>cname</u>, city)

Universal Quantifiers

Find cities that have a company
such that ***all*** its products have price < 100

Universal quantifiers are hard ! ☹

Product (<u>pname</u>, price, company)
Company(<u>cname</u>, city)

Universal Quantifiers

Find cities that have a company
such that *<u>all</u>* its products have price < 100

Relational Calculus (a.k.a. First Order Logic) – next week

$$q(y) = \exists x.\ Company(x,y) \land (\forall z. \forall p.\ Product(z,p,x) \rightarrow p < 100)$$

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

Universal Quantifiers

De Morgan's Laws:

$\neg(A \land B) = \neg A \lor \neg B$

$\neg(A \lor B) = \neg A \land \neg B$

$\neg \forall x.\ P(x) = \exists x.\ \neg P(x)$

$\neg \exists x.\ P(x) = \forall x.\ \neg P(x)$

$\neg(A \rightarrow B) = A \land \neg B$

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

Universal Quantifiers

De Morgan's Laws:

$\neg(A \wedge B) = \neg A \vee \neg B$

$\neg(A \vee B) = \neg A \wedge \neg B$

$\neg \forall x.\ P(x) = \exists x.\ \neg P(x)$

$\neg \exists x.\ P(x) = \forall x.\ \neg P(x)$

$\neg(A \rightarrow B) = A \wedge \neg B$

$q(y) = \exists x.\ Company(x,y) \wedge (\forall z. \forall p.\ Product(z,p,x) \rightarrow p < 100)$

=

Product (<u>pname</u>, price, company)
Company(<u>cname</u>, city)

Universal Quantifiers

De Morgan's Laws:

$\neg(A \wedge B) = \neg A \vee \neg B$

$\neg(A \vee B) = \neg A \wedge \neg B$

$\neg \forall x.\ P(x) = \exists x.\ \neg P(x)$

$\neg \exists x.\ P(x) = \forall x.\ \neg P(x)$

$\neg(A \rightarrow B) = A \wedge \neg B$

$q(y) = \exists x.\ Company(x,y) \wedge (\forall z. \forall p.\ Product(z,p,x) \rightarrow p < 100)$

=

$q(y) = \exists x.\ Company(x,y) \wedge \neg(\exists z \exists p.\ Product(z,p,x) \wedge p \geq 100)$

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

Universal Quantifiers

De Morgan's Laws:

$\neg(A \land B) = \neg A \lor \neg B$

$\neg(A \lor B) = \neg A \land \neg B$

$\neg \forall x.\ P(x) = \exists x.\ \neg P(x)$

$\neg \exists x.\ P(x) = \forall x.\ \neg P(x)$

$\neg(A \rightarrow B) = A \land \neg B$

$q(y) = \exists x.\ Company(x,y) \land (\forall z. \forall p.\ Product(z,p,x) \rightarrow p < 100)$

=

$q(y) = \exists x.\ Company(x,y) \land \neg(\exists z \exists p.\ Product(z,p,x) \land p \geq 100)$

=

$theOtherCompanies(x) = \exists z \exists p.\ Product(z,p,x) \land p \geq 100$

$q(y) = \exists x.\ Company(x,y) \land \neg\ theOtherCompanies(x)$

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

Universal Quantifiers: NOT IN

theOtherCompanies(x) = $\exists z \exists p.\ \text{Product}(z,p,x) \wedge p \geq 100$

q(y) = $\exists x.\ \text{Company}(x,y) \wedge \neg\ \text{theOtherCompanies}(x)$

```
SELECT DISTINCT  c.city
FROM     Company c
WHERE  c.cname NOT IN (SELECT p.company
                       FROM Product p
                       WHERE p.price >= 100)
```

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

Universal Quantifiers: NOT EXISTS

theOtherCompanies(x) = $\exists z \exists p.$ Product(z,p,x) $\wedge$ $p \geq 100$
q(y) = $\exists x.$ Company(x,y) $\wedge$ $\neg$ theOtherCompanies(x)

```
SELECT DISTINCT  c.city
FROM     Company c
WHERE  NOT EXISTS (SELECT *
                   FROM Product p
                   WHERE c.cname = p.company AND p.price >= 100)
```

Correlated subquery!

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

Universal Quantifiers: ALL

```
SELECT DISTINCT  c.city
FROM      Company c
WHERE 100 > ALL  (SELECT p.price
                  FROM Product p
                  WHERE p.company = c.cname)
```

Question for Database Fans and their Friends

- Can we unnest this query ?

Find cities that have a company
such that *all* its products have price < 100

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 existing 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 existing tuples

Product

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

Company

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

Q ⇨

A	B
149.99	Lodtz
19.99	Lyon

Is the mystery query monotone?

Product

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

Company

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

Q ⇨

A	B
149.99	Lyon
19.99	Lyon
19.99	Bonn
149.99	Bonn

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 existing tuples

Product

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

Company

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

Q ⇒

A	B
149.99	Lodtz
19.99	Lyon

Is the mystery query monotone?

NO!

Product

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

Company

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

Q ⇒

A	B
149.99	Lyon
19.99	Lyon
19.99	Bonn
149.99	Bonn

Monotone Queries

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

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

Monotone Queries

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

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

Proof. We use the nested loop semantics: if we insert a tuple in a relation R_i, then x_i will take all the old values, in addition to the new value.

```
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

This query is not monotone:

Find cities that have a company such that *all* its products have price < 100

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

Monotone Queries

This query is not monotone:

Find cities that have a company
such that *all* its products have price < 100

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

Product (<u>pname</u>, price, cid)

Company(<u>cid</u>, cname, city)

Monotone Queries

This query is not monotone:

Find cities that have a company such that *<u>all</u>* its products have price < 100

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

<u>Consequence</u>: we cannot write it as a SELECT-FROM-WHERE query without nested subqueries

NULLS in SQL

- Whenever we don't have a value, we can put a NULL

- Can mean many things:
  - Value does not exists
  - Value exists but is unknown
  - Value not applicable
  - Etc.

- The schema specifies for each attribute if can be null (*nullable* attribute) or not

Null Values

Person(<u>name</u>, age, height, weight)

height unknown

```
INSERT INTO Person VALUES('Joe',20,NULL,200)
```

Null Values

Person(<u>name</u>, age, height, weight)

height unknown

```
INSERT INTO Person VALUES('Joe',20,NULL,200)
```

Rules for computing with NULLs

- If x is NULL then x+7 is still NULL
- If x is 2 then x>5 is FALSE
- If x is NULL then x>5 is UNKNOWN
- If x is 10 then x>5 is TRUE

Null Values

height unknown

Person(<u>name</u>, age, height, weight)

INSERT INTO Person VALUES('Joe',20,NULL,200)

Rules for computing with NULLs

- If x is NULL then x+7 is still NULL
- If x is 2 then x>5 is FALSE
- If x is NULL then x>5 is UNKNOWN
- If x is 10 then x>5 is TRUE

FALSE	=	0
UNKNOWN	=	0.5
TRUE	=	1

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

Null Values

- C1 AND C2 = min(C1, C2)
- C1 OR C2 = max(C1, C2)
- NOT C1 = 1 – C1

```
SELECT *
FROM Person
WHERE  (age < 25) AND
              (height > 6 OR weight > 190)
```

E.g.
age=20
height=NULL
weight=200

Rule in SQL: result includes only tuples that yield TRUE

Null Values

Unexpected behavior:

```
SELECT *
FROM    Person
WHERE  age < 25  OR  age >= 25
```

Some Persons not included !

Null Values

Can test for NULL explicitly:

x IS NULL

x IS NOT NULL

```
SELECT *
FROM    Person
WHERE  age < 25  OR  age >= 25 OR age IS NULL
```

Now all Person are included

Detour into DB Research

Imielinski&Libski, *Incomplete Databases*, 1986

- Database = is in one of several states, or *possible worlds*
  - Number of possible worlds is exponential in size of db
- Query semantics = return the *certain answers*

Detour into DB Research

Imielinski&Libski, *Incomplete Databases*, 1986

- Database = is in one of several states, or *possible worlds*
  - Number of possible worlds is exponential in size of db
- Query semantics = return the *certain answers*

Very influential paper:

- Incomplete DBs used in probabilistic databases, *what-if* scenarios, data cleaning, data exchange

Detour into DB Research

Imielinski&Libski, *Incomplete Databases*, 1986

- Database = is in one of several states, or *possible worlds*
  - Number of possible worlds is exponential in size of db
- Query semantics = return the *certain answers*

Very influential paper:

- Incomplete DBs used in probabilistic databases, *what-if* scenarios, data cleaning, data exchange

In SQL, NULLs are the simplest form of incomplete database:

- Database: NULL takes independently any possible value
- Query semantics: not exactly certain answers (why?)

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

Outerjoins

An "inner join":

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

Same as:

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

But Products that never sold will be lost

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

Outerjoins

If we want the never-sold products, need a "left outer join":

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

Product(<u>name</u>, category)
Purchase(prodName, store)

Product

<u>name</u>	category
Gizmo	gadget
Camera	Photo
OneClick	Photo

Purchase

prodName	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz

name	store
Gizmo	Wiz
Camera	Ritz
Camera	Wiz
OneClick	NULL

Outer Joins

- Left outer join:
  - Include the left tuple even if there's no match
- Right outer join:
  - Include the right tuple even if there's no match
- Full outer join:
  - Include both left and right tuples even if there's no match

Aggregations

Five basic aggregate operations in SQL

- count
- sum
- avg
- max
- min

Purchase(product, price, quantity)

Counting Duplicates

COUNT applies to duplicates, unless otherwise stated:

```
SELECT  count(product)
FROM    Purchase
WHERE   price>3.99
```

Same as count(*)

Except if some product is NULL

We probably want:

```
SELECT  count(DISTINCT product)
FROM    Purchase
WHERE   price>3.99
```

Purchase(product, price, quantity)

Grouping and Aggregation

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

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

product	price	quantity
Bagel	3	20
Bagel	1.50	20
Banana	0.5	50
Banana	2	10
Banana	4	10

What is the answer?

Grouping and Aggregation

1. Compute the FROM and WHERE clauses.

2. Group by the attributes in the GROUP BY

3. Compute the SELECT clause: group attrs and aggregates.

1&2. FROM-WHERE-GROUPBY

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

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

3. SELECT: Each Group → One Answer

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
```

Ordering Results

```
SELECT product, sum(quantity) as TotalSales
FROM   purchase
GROUP BY product
ORDER BY TotalSales DESC
LIMIT 20
```

```
SELECT product, sum(quantity) as TotalSales
FROM   purchase
GROUP BY product
ORDER BY sum(quantity) DESC
LIMIT 20
```

Equivalent, but not all systems accept both syntax forms

HAVING Clause

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

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

HAVING clause contains conditions on aggregates.

WHERE vs HAVING

- WHERE condition: applied to individual rows
  - Determine which rows contributed to the aggregate
  - All attributes are allowed
  - No aggregates functions allowed
- HAVING condition: applied to the entire group
  - Entire group is returned, or not al all
  - Only group attributes allowed
  - Aggregate functions allowed

General form of Grouping and Aggregation

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

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

Why ?

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
```

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

Empty Groups Running Example

For the next slides, run this in postgres:

```
create table Purchase(pid int primary key, product text, price float, quantity int, month varchar(15));
create table Product (pid int primary key, pname text, manufacturer text);

insert into Purchase values(01,'bagel',1.99,20,'september');
insert into Purchase values(02,'bagel',2.50,12,'december');
insert into Purchase values(03,'banana',0.99,9,'september');
insert into Purchase values(04,'banana',1.59,9,'february');
insert into Purchase values(05,'gizmo',99.99,5,'february');
insert into Purchase values(06,'gizmo',99.99,3,'march');
insert into Purchase values(07,'gizmo',49.99,3,'april');
insert into Purchase values(08,'gadget',89.99,3,'january');
insert into Purchase values(09,'gadget',89.99,3,'february');
insert into Purchase values(10,'gadget',49.99,3,'march');
insert into Purchase values(11,'orange',null,5,'may');

insert into product values(1,'bagel','Sunshine Co.');
insert into product values(2,'banana','BusyHands');
insert into product values(3,'gizmo','GizmoWorks');
insert into product values(4,'gadget','BusyHands');
insert into product values(5,'powerGizmo','PowerWorks');
```

Purchase(product, price, quantity)
Product(pname, manufacturer)

Empty Group Problem

Query: for each manufacturer, compute the total number of purchases for its products

Problem: a group can never be empty! In particular, count(*) is never 0

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

Purchase(product, price, quantity)

Product(pname, manufacturer)

Solution 1: Outer Join

Query: for each manufacturer,
compute the total number of purchases
for its products

Use a LEFT OUTER JOIN.

Make sure you count an attribute that may be NULL

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

Purchase(product, price, quantity)
Product(pname, manufacturer)

Solution 2: Nested Query

Query: for each manufacturer,
compute the total number of purchases
for its products

Use a subquery in the SELECT clause

Notice second use of Product. Why?

```
SELECT DISTINCT x.manufacturer,
      (SELECT count(*)
        FROM Product z, Purchase y
        WHERE x.manufacturer = z.manufacturer
            and   z.pname = y.product)
FROM Product x
```

Purchase(product, price, quantity)

Product(pname, manufacturer)

Finding Witnesses

Query: for each manufacturer, find its most expensive product

Finding the maximum price is easy:

Purchase(product, price, quantity)

Product(<u>pname</u>, manufacturer)

Finding Witnesses

<u>Query</u>: for each manufacturer, find its most expensive product

Finding the maximum price is easy:

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

…but we need to find the product that sold at that price!

Purchase(product, price, quantity)

Product(pname, manufacturer)

Finding Witnesses

Query: for each manufacturer, find its most expensive product

Use a subquery in the FROM clause:

```
SELECT DISTINCT u.manufacturer, u.pname
FROM Product u, Purchase v,
        (SELECT x.manufacturer, max(y.price) as mprice
         FROM Product x, Purchase y
         WHERE x.pname = y.product
         GROUP BY x.manufacturer) z
WHERE u.pname = v.product
    and  u.manufacturer = z.manufacturer
    and  v.price = z.mprice
```

Purchase(product, price, quantity)

Product(pname, manufacturer)

Finding Witnesses

Query: for each manufacturer, find its most expensive product

Using WITH :

```
WITH Temp as (SELECT x.manufacturer, max(y.price) as mprice
              FROM Product x, Purchase y
              WHERE x.pname = y.product
              GROUP BY x.manufacturer)
SELECT DISTINCT u.manufacturer, u.pname
FROM Product u, Purchase v, Temp z
WHERE u.pname = v.product
   and  u.manufacturer = z.manufacturer
   and  v.price = z.mprice
```