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:**
  – Book available on the Kindle too
  – Use it to read background material
  – You may borrow it, no need to buy

• **Optional Textbook**

• **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
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\textsuperscript{st}, 3\textsuperscript{rd})
- SQL
- Relational Algebra
- Indexes, search trees
- Search in a binary tree

- Query optimization (e.g. join reordering)
- Transactions (e.g. ACID)
- Logic: $\land, \lor, \forall, \exists, \neg, \in$
- 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:

<table>
<thead>
<tr>
<th>id</th>
<th>fName</th>
<th>lName</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>195428</td>
<td>Tom</td>
<td>Hanks</td>
<td>M</td>
</tr>
<tr>
<td>645947</td>
<td>Amy</td>
<td>Hanks</td>
<td>F</td>
</tr>
<tr>
<td>. .</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

#### Casts:

<table>
<thead>
<tr>
<th>pid</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>195428</td>
<td>337166</td>
</tr>
<tr>
<td>. .</td>
<td></td>
</tr>
</tbody>
</table>

#### Movie:

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>337166</td>
<td>Toy Story</td>
<td>1995</td>
</tr>
<tr>
<td>. .</td>
<td>. .</td>
<td>. .</td>
</tr>
<tr>
<td>. .</td>
<td>. .</td>
<td>. .</td>
</tr>
</tbody>
</table>
SQL

SELECT * FROM Actor
SELECT * FROM Actor

SELECT * FROM Actor
LIMIT 50
SQL

SELECT * FROM Actor

SELECT count(*) FROM Actor

SELECT * FROM Actor LIMIT 50
SQL

SELECT * FROM Actor

SELECT count(*) FROM Actor

SELECT * FROM Actor LIMIT 50

SELECT * FROM Actor WHERE IName = 'Hanks'
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?

### SELECT Statement

```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
```

### Tables

<table>
<thead>
<tr>
<th>Actor</th>
<th>Casts</th>
<th>Movie</th>
</tr>
</thead>
<tbody>
<tr>
<td><img src="image" alt="Actor Table" /></td>
<td><img src="image" alt="Casts Table" /></td>
<td><img src="image" alt="Movie Table" /></td>
</tr>
</tbody>
</table>

- **Actor**: 1.8M actors
- **Casts**: 11M casts
- **Movie**: 1.5M movies
How Can We Evaluate the Query?

### Actor:

<table>
<thead>
<tr>
<th>id</th>
<th>fName</th>
<th>lName</th>
<th>gender</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>Hanks</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.8M actors

### Casts:

<table>
<thead>
<tr>
<th>pid</th>
<th>mid</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
</tr>
<tr>
<td>...</td>
<td></td>
</tr>
</tbody>
</table>

11M casts

### Movie:

<table>
<thead>
<tr>
<th>id</th>
<th>Name</th>
<th>year</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td></td>
<td>1995</td>
</tr>
<tr>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

1.5M movies

```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
```
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

σ_{lName='Hanks'}
σ_{year=1995}
σ_{lName='Hanks'}
σ_{year=1995}

Actor  Casts  Movie  Actor  Casts  Movie
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*)*

*Table data not shown*
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*)*
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

#### Attribute names
- PName
- Price
- Category
- Manufacturer

#### Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>

#### Key

Table name

Tuples or rows

CSEP544 - Winter, 2014
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

Product (PName, price, category, manufacturer)

- SELECT * FROM Product WHERE category='Gadgets'
- SELECT * FROM Product WHERE category LIKE 'Ga%'
- SELECT * FROM Product WHERE category > 'Gadgets'
## Selections in SQL

<table>
<thead>
<tr>
<th>SQL Query</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>SELECT * FROM Product WHERE category='Gadgets'</code></td>
<td>Select all columns from the <code>Product</code> table where category is 'Gadgets'</td>
</tr>
<tr>
<td><code>SELECT * FROM Product WHERE category LIKE 'Ga%'</code></td>
<td>Select all columns from the <code>Product</code> table where category starts with 'Ga'</td>
</tr>
<tr>
<td><code>SELECT * FROM Product WHERE category &gt; 'Gadgets'</code></td>
<td>Select all columns from the <code>Product</code> table where category is greater than 'Gadgets'</td>
</tr>
<tr>
<td><code>SELECT * FROM Product WHERE category LIKE '%dg%'</code></td>
<td>Select all columns from the <code>Product</code> table where category contains 'dg'</td>
</tr>
</tbody>
</table>

Product (PName, price, category, manufacturer)
Projections (and Selections) in SQL

```sql
SELECT pname
FROM Product
WHERE category='Gadgets'
```
Projections (and Selections) in SQL

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

```sql
SELECT category
FROM Product
```
Projections (and Selections) in SQL

\[
\text{SELECT \ } \text{pname} \\
\text{FROM \ \ Product} \\
\text{WHERE \ category='Gadgets'}
\]

\[
\text{SELECT \ \ category} \\
\text{FROM \ \ Product}
\]

\[
\text{SELECT DISTINCT \ category} \\
\text{FROM \ \ Product}
\]

Need DISTINCT (why?)
“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

<table>
<thead>
<tr>
<th>CName</th>
<th>StockPrice</th>
<th>Country</th>
</tr>
</thead>
<tbody>
<tr>
<td>GizmoWorks</td>
<td>25</td>
<td>USA</td>
</tr>
<tr>
<td>Canon</td>
<td>65</td>
<td>Japan</td>
</tr>
<tr>
<td>Hitachi</td>
<td>15</td>
<td>Japan</td>
</tr>
</tbody>
</table>

Product

<table>
<thead>
<tr>
<th>PName</th>
<th>Price</th>
<th>Category</th>
<th>Manufacturer</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>$19.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>Powergizmo</td>
<td>$29.99</td>
<td>Gadgets</td>
<td>GizmoWorks</td>
</tr>
<tr>
<td>SingleTouch</td>
<td>$149.99</td>
<td>Photography</td>
<td>Canon</td>
</tr>
<tr>
<td>MultiTouch</td>
<td>$203.99</td>
<td>Household</td>
<td>Hitachi</td>
</tr>
</tbody>
</table>
Joins

Product (PName, Price, Category, Manufacturer)
Company (CName, 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 a_1, a_2, ..., a_k
FROM  R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE Conditions
```
Semantics of SQL Queries

\[\text{SELECT } a_1, a_2, \ldots, a_k \]
\[\text{FROM } \ R_1 \text{ AS } x_1, \quad R_2 \text{ AS } x_2, \quad \ldots, \quad R_n \text{ AS } x_n \]
\[\text{WHERE } \ \text{Conditions} \]

\[\text{Answer} = \{\} \]
\[\text{for } x_1 \text{ in } R_1 \text{ do} \]
\[\quad \text{for } x_2 \text{ in } R_2 \text{ do} \]
\[\quad \quad \quad \quad \ldots \]
\[\quad \quad \quad \quad \quad \text{for } x_n \text{ in } R_n \text{ do} \]
\[\quad \quad \quad \quad \quad \quad \quad \text{if } \text{Conditions} \]
\[\quad \quad \quad \quad \quad \quad \quad \quad \text{then } \text{Answer} = \text{Answer} \cup \{(a_1,\ldots,a_k)\} \]
\[\text{return } \text{Answer} \]
Subqueries

• A subquery or a nested query is another SQL query nested inside a larger query

• A subquery may occur in:
  SELECT
  FROM
  WHERE

Examples at the end of the lecture

Examples on following slides

Avoid writing nested queries when possible; keep in mind that sometimes it’s impossible
Run this in postgres, then try the examples on the following slides.

```sql
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');
```
Existential Quantifiers

Find cities that have a company that manufacture \textit{some} product with price < 100
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
```
Universal Quantifiers

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

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

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

Universal quantifiers are hard! 😞
Universal Quantifiers

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

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

q(y)= \exists x. \text{Company}(x,y) \land (\forall z. \forall p. \text{Product}(z,p,x) \rightarrow p < 100)
Universal Quantifiers

De Morgan’s Laws:

\[ \neg (A \land B) = \neg A \lor \neg B \]
\[ \neg (A \lor B) = \neg A \land \neg B \]
\[ \neg \forall x. P(x) = \exists x. \neg P(x) \]
\[ \neg \exists x. P(x) = \forall x. \neg P(x) \]

\[ \neg (A \rightarrow B) = A \land \neg B \]
Universal Quantifiers

De Morgan’s Laws:

\[ \neg (A \land B) = \neg A \lor \neg B \]
\[ \neg (A \lor B) = \neg A \land \neg B \]
\[ \neg \forall x. P(x) = \exists x. \neg P(x) \]
\[ \neg \exists x. P(x) = \forall x. \neg P(x) \]

\[ q(y) = \exists x. \text{Company}(x,y) \land (\forall z. \forall p. \text{Product}(z,p,x) \rightarrow p < 100) \]

= Product (pname, price, company)
  Company (cname, city)
Universal Quantifiers

De Morgan’s Laws:

\[ \neg (A \land B) = \neg A \lor \neg B \]
\[ \neg (A \lor B) = \neg A \land \neg B \]
\[ \neg \forall x. P(x) = \exists x. \neg P(x) \]
\[ \neg \exists x. P(x) = \forall x. \neg P(x) \]

\[ q(y) = \exists x. \text{Company}(x,y) \land (\forall z. \forall p. \text{Product}(z,p,x) \rightarrow p < 100) \]

\[ = \]

\[ q(y) = \exists x. \text{Company}(x,y) \land \neg(\exists z \exists p. \text{Product}(z,p,x) \land p \geq 100) \]
Universal Quantifiers

De Morgan’s Laws:

\[ \neg(A \land B) = \neg A \lor \neg B \]
\[ \neg(A \lor B) = \neg A \land \neg B \]
\[ \neg \forall x. P(x) = \exists x. \neg P(x) \]
\[ \neg \exists x. P(x) = \forall x. \neg P(x) \]

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)
Universal Quantifiers: \textbf{NOT IN}

\begin{align*}
\text{theOtherCompanies}(x) &= \exists z \exists p. \ \text{Product}(z,p,x) \land p \geq 100 \\
q(y) &= \exists x. \ \text{Company}(x,y) \land \neg \text{theOtherCompanies}(x)
\end{align*}

\text{SELECT DISTINCT} \quad \text{c.city} \\
\text{FROM} \quad \text{Company c} \\
\text{WHERE} \quad \text{c.cname} \ \text{NOT IN} \quad (\text{SELECT} \ p.\text{company} \ \\
\quad \text{FROM} \ \text{Product p} \ \\
\quad \text{WHERE} \ p.\text{price} \geq 100)$
Universal Quantifiers: \textbf{NOT EXISTS}

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

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

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

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>999.99</td>
<td>c003</td>
</tr>
<tr>
<td>Camera</td>
<td>149.99</td>
<td>c001</td>
</tr>
</tbody>
</table>

**Company**

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>city</th>
</tr>
</thead>
<tbody>
<tr>
<td>c001</td>
<td>Sunworks</td>
<td>Bonn</td>
</tr>
<tr>
<td>c002</td>
<td>DB Inc.</td>
<td>Lyon</td>
</tr>
<tr>
<td>c003</td>
<td>Builder</td>
<td>Lodtz</td>
</tr>
</tbody>
</table>

Is the mystery query monotone?

**Product**

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>999.99</td>
<td>c003</td>
</tr>
<tr>
<td>Camera</td>
<td>149.99</td>
<td>c001</td>
</tr>
<tr>
<td>iPad</td>
<td>499.99</td>
<td>c001</td>
</tr>
</tbody>
</table>

**Company**

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>149.99</td>
<td>Lodtz</td>
</tr>
<tr>
<td>19.99</td>
<td>Lyon</td>
</tr>
<tr>
<td>149.99</td>
<td>Bonn</td>
</tr>
</tbody>
</table>

Is the mystery query monotone?
Monotone Queries

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

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

```
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
```
Monotone Queries

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

\begin{verbatim}
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
\end{verbatim}

**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.

\begin{verbatim}
for x_1 in R_1 do
  for x_2 in R_2 do
    ....
    for x_n in R_n do
      if Conditions
        output (a_1, ..., a_k)
\end{verbatim}
This query is not monotone:

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

This query is not monotone:

Find cities that have a company such that \textit{all} its products have price < 100

<table>
<thead>
<tr>
<th>pname</th>
<th>price</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>19.99</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>999.99</td>
<td>c001</td>
</tr>
</tbody>
</table>

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

Product \((\texttt{pname}, \texttt{price}, \texttt{cid})\)
Company\((\texttt{cid}, \texttt{cname}, \texttt{city})\)
Monotone Queries

This query is not monotone:

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

Consequence: 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(name, age, height, weight)

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

height unknown
Null Values

Person(name, 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
Null Values

Person(name, 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

- $C1 \text{ AND } C2 = \min(C1, C2)$
- $C1 \text{ OR } C2 = \max(C1, C2)$
- $\text{NOT } C1 = 1 - C1$
Null Values

- \( C_1 \text{ AND } C_2 = \min(C_1, C_2) \)
- \( C_1 \text{ OR } C_2 = \max(C_1, C_2) \)
- \( \text{NOT } C_1 = 1 - C_1 \)

```
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:
- IS NULL
- 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?)
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
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
```
<table>
<thead>
<tr>
<th>name</th>
<th>category</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>gadget</td>
</tr>
<tr>
<td>Camera</td>
<td>Photo</td>
</tr>
<tr>
<td>OneClick</td>
<td>Photo</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>prodName</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>name</th>
<th>store</th>
</tr>
</thead>
<tbody>
<tr>
<td>Gizmo</td>
<td>Wiz</td>
</tr>
<tr>
<td>Camera</td>
<td>Ritz</td>
</tr>
<tr>
<td>Camera</td>
<td>Wiz</td>
</tr>
<tr>
<td>OneClick</td>
<td>NULL</td>
</tr>
</tbody>
</table>
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
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
Grouping and Aggregation

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

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

<table>
<thead>
<tr>
<th>product</th>
<th>price</th>
<th>quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

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.
SELECT product, sum(quantity) AS TotalSales
FROM Purchase
WHERE price > 1
GROUP BY product
3. SELECT: Each Group → One Answer

<table>
<thead>
<tr>
<th>Product</th>
<th>Price</th>
<th>Quantity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bagel</td>
<td>3</td>
<td>20</td>
</tr>
<tr>
<td>Bagel</td>
<td>1.50</td>
<td>20</td>
</tr>
<tr>
<td>Banana</td>
<td>0.5</td>
<td>50</td>
</tr>
<tr>
<td>Banana</td>
<td>2</td>
<td>10</td>
</tr>
<tr>
<td>Banana</td>
<td>4</td>
<td>10</td>
</tr>
</tbody>
</table>

```
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 at all
  – Only group attributes allowed
  – Aggregate functions allowed
General form of Grouping and Aggregation

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

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

Evaluation steps:
1. Evaluate FROM-WHERE using Nested Loop Semantics
2. Group by the attributes $a_1, \ldots, a_k$
3. Apply condition $C_2$ to each group (may have aggregates)
4. Compute aggregates in $S$ and return the result

```sql
SELECT S
FROM R1, \ldots, Rn
WHERE C1
GROUP BY a1, \ldots, ak
HAVING C2
```
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');
```
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, \( \text{count}(*) \) is never 0

```
SELECT x.manufacturer, count(*)
FROM Product x, Purchase y
WHERE x.pname = y.product
GROUP BY x.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
```
Solution 2: Nested Query

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

Use a subquery in the SELECT clause

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

Notice second use of Product. Why?
Finding Witnesses

**Query**: for each manufacturer, find its most expensive product

Finding the maximum price is easy:
Finding Witnesses

**Query**: for each manufacturer, find its most expensive product

Finding the maximum price is easy:

```sql
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!
Finding Witnesses

**Query**: for each manufacturer, find its most expensive product

Use a subquery in the **FROM** clause:

```sql
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
```
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
```