Principles of Database Systems
CSE 544

Lecture 01
Introduction and SQL
Staff

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

• TA:
  – Laurel Orr ljorr1@cs.washington.edu
  – Office hours: Mondays, 5:30-6:20, room TBA
About Me

• Joined CSE in 2000
  – Previously at Bell Labs / AT&T Labs

• Research: where math can bring big gains to data management:
  – Probabilistic databases
  – Novel query evaluation algorithms for big data
  – Pricing and enforcing data use agreements
  – Discovering causal relationships in data
Class Format

• Lectures Mondays, 6:30-9:20
  – Two lecture rooms: CSE and Microsoft
  – Video streamed + archived

• 5 Homework Assignments

• Several reading assignments

• An online, take-home final (two days), Dec. 9-10
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

Several papers to read and review

- Some are short blogs (Stonebraker)
- Most 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: courses.cs.washington.edu/courses/csep544/15au/
  – Lectures + video
  – Homework assignments
  – Reading assignments
  – Information about the final

• Mailing list:
  – Announcements, group discussions
  – Low traffic, must read

• Discussion board:
  – Feel free to post, discuss
  – 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
  – ColumnStore (maybe NoSQL)
Evaluation

• **Homework Assignments 50%:**
  – Three light programming, two theory

• **Paper reviews 20%:**
  – About ½ page each

• **Final exam 30%:**
  – Take home, online exam
  – Two days: Wednesday-Thursday, Dec. 9-10
Homework Assignments 50%

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

**Late days policy:**
- 4 late-days in 24-hour chunks, no questions asked
- at most 2 late-days per assignment
- Absolutely no additional extensions granted
Assignments 50%

**HW1: SQL – posted!**

- Three Tasks:
  - Create tables
  - Create indexes
  - Compute 11 SQL Queries

- Dataset = a copy of IMDB from 2010
- Install a DMBS on your machine: either Postgres or SQL Server.
- Watch mailing list for a possible update (+Azure)

**Due:** Tuesday, October 20
Paper Reviews

• Papers:
  – A few short blogs by Stonebraker
  – Several systems-research papers

• Reviews:
  – Due Mondays, by 3pm
  – Brief (½ page) summary of the lessons you learned from the paper
  – Website has some suggested questions

• Next Monday:
  – Blog on Big Data (what is that??)
  – Paper on data models + perils of inventing new
Final

Format
• Take-home, online final

• Opens: Wednesday, Dec. 9 at 8am

• Closes: Thursday, Dec. 10 at 8pm

• No late days/hours/minutes/seconds
Goals of the Class

This is a graduate level class!

• Deep understanding of traditional material

• Novel material
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
• 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, …
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>
SELECT * FROM Actor
SQL – Preview

SELECT * FROM Actor

SELECT * FROM Actor
LIMIT 50
SELECT * FROM Actor

SELECT count(*) FROM Actor

SELECT * FROM Actor

LIMIT 50
SQL – Preview

```
SELECT * FROM Actor
```

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

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

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

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

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?

<table>
<thead>
<tr>
<th>Actor:</th>
<th>Casts:</th>
<th>Movie:</th>
</tr>
</thead>
<tbody>
<tr>
<td>id</td>
<td>fName</td>
<td>lName</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>Hanks</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

1.8M actors

11M casts

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 ]
Query Plans – Preview

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
Physical Data Independence

Physical data independence:

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

See Goes around… paper, due next week
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:

See Goes around… paper, due next week
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*)
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*)

See Goes around… paper, due next week.
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**
Client/Server Architecture

- **Server**: stores the database
  - Single server or cluster of servers
  - Postgres: runs server on your own computer
- **Clients**: run apps and connect to DBMS
- **Connection Protocol**: ODBC/JDBC
- **Others**:
  - Three-tier architecture: add the app server
  - Embedded in app (e.g. SQLite): no server

Why is client/server preferable to embedded in app?
SQL

• Will cover SQL rather quickly today

• Will not finish discussing all slides in class: please read the rest on your own!

• Other resources for learning SQL:
  – Textbook
  – Office hours
  – Postgres help: type \h or \?
  – SQL Server help
  – Discussion board

• Start working on HW1!
# Tables in SQL

<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>
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
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), TEXT`
  – 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?

• 1NF is an exception. Other NF’s refer to splitting a wide table into smaller tables:
  – Boyce Codd Normal Form (BCNF)
  – Third Normal Form (3NF)

• We will discuss BCNF later in class
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

- \[
\text{SELECT} \quad * \\
\text{FROM} \quad \text{Product} \\
\text{WHERE} \quad \text{category} = \text{‘Gadgets’}
\]

- \[
\text{SELECT} \quad * \\
\text{FROM} \quad \text{Product} \\
\text{WHERE} \quad \text{category} > \text{‘Gadgets’}
\]

- \[
\text{SELECT} \quad * \\
\text{FROM} \quad \text{Product} \\
\text{WHERE} \quad \text{category} \text{ LIKE} \text{ ‘Ga%’}
\]

- \[
\text{SELECT} \quad * \\
\text{FROM} \quad \text{Product} \\
\text{WHERE} \quad \text{category} \text{ LIKE} \text{ ‘%dg%’}
\]

Product (PName, price, category, manufacturer)
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
```

```
SELECT  DISTINCT category
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

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

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, ..., 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`  
  - `WHERE`

Examples at the end of the lecture.

Avoid writing nested queries when possible; keep in mind that sometimes it’s impossible.
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('fgl', '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, 'fgl');
insert into product values('p6', 200, 'fgl');
insert into product values('p7', 10, 'klm');
insert into product values('p8', 220, 'klm');
Existential Quantifiers

Find cities that have a company that manufacture *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

Existential quantifiers are easy! 😊
Universal Quantifiers

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

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

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

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

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

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

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)
Universal Quantifiers: NOT IN

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

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

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

\text{Correlated subquery!}
Universal Quantifiers: **ALL**

```sql
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

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pname</strong></td>
<td><strong>cid</strong></td>
</tr>
<tr>
<td>Gizmo</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>c003</td>
</tr>
<tr>
<td>Camera</td>
<td>c001</td>
</tr>
</tbody>
</table>

<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>
</tbody>
</table>
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.
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

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pname</strong></td>
<td><strong>cid</strong></td>
</tr>
<tr>
<td>Gizmo</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>c003</td>
</tr>
<tr>
<td>Camera</td>
<td>c001</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Product</th>
<th>Company</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>pname</strong></td>
<td><strong>cid</strong></td>
</tr>
<tr>
<td>Gizmo</td>
<td>c001</td>
</tr>
<tr>
<td>Gadget</td>
<td>c002</td>
</tr>
<tr>
<td>Camera</td>
<td>c003</td>
</tr>
</tbody>
</table>

Is the mystery query monotone?

- Yes
- No
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)**

<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 (cid, cname, city)**

<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>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>

**Q**

<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>
</tbody>
</table>

**Q**

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

**NO!**
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 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
```

**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.

```plaintext
for x_1 in R_1 do
  for x_2 in R_2 do
    ....
  for x_n in R_n do
    if Conditions
      output (a_1, ..., a_k)
```
Monotone Queries

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

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
pname & price & cid \\
\hline
Gizmo & 19.99 & c001 \\
\hline
Gadget & 999.99 & c001 \\
\hline
\end{tabular}
\end{table}

\begin{table}[h]
\centering
\begin{tabular}{|c|c|c|}
\hline
cid & cname & city \\
\hline
c001 & Sunworks & Bonn \\
\hline
\end{tabular}
\end{table}
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)  

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)

```sql
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

<table>
<thead>
<tr>
<th>Value</th>
<th>Code</th>
</tr>
</thead>
<tbody>
<tr>
<td>FALSE</td>
<td>0</td>
</tr>
<tr>
<td>UNKNOWN</td>
<td>0.5</td>
</tr>
<tr>
<td>TRUE</td>
<td>1</td>
</tr>
</tbody>
</table>
Null Values

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

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

\[
\text{SELECT } * \\
\text{FROM } \text{Person} \\
\text{WHERE } (\text{age} < 25) \text{ AND } \\
(\text{height} > 6 \text{ OR } \text{weight} > 190)
\]

Rule in SQL: result includes only tuples that yield TRUE

E.g. age=20 height=NULL weight=200
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”:

\[
\begin{align*}
& \text{SELECT } x.\text{name}, y.\text{store} \\
& \text{FROM} \quad \text{Product } x, \text{ Purchase } y \\
& \text{WHERE} \quad x.\text{name} = y.\text{prodName}
\end{align*}
\]

Same as:

\[
\begin{align*}
& \text{SELECT } x.\text{name}, y.\text{store} \\
& \text{FROM} \quad \text{Product } x \ \text{JOIN} \ \text{Purchase } y \ \text{ON} \\
& \quad x.\text{name} = y.\text{prodName}
\end{align*}
\]

But Products that never sold will be lost
SELECT x.name, y.store
FROM Product x LEFT OUTER JOIN Purchase y ON
x.name = y.prodName

If we want the never-sold products, need a “left outer join”: 
<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
COUNT applies to duplicates, unless otherwise stated:

\[
\text{SELECT count(product) FROM Purchase WHERE price>3.99}
\]

Same as \text{count(*)}

Except if some product is NULL

We probably want:

\[
\text{SELECT count(DISTINCT product) FROM Purchase WHERE price>3.99}
\]
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
```

<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.
### 1&2. FROM-WHERE-GROUPBY

```
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>
3. SELECT: Each Group $\rightarrow$ 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**

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

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

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

C2 = is any condition on aggregate expressions and on attributes \(a_1, ..., a_k\)
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
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, 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

```sql
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:
Query: 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!
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
```
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