Database Management Systems
CSEP 544

Lecture 3: SQL
Relational Algebra, and Datalog
Announcements

• HW2 due tonight (11:59pm)

• PA3 & HW3 released
HW3

• We will be using SQL Server in the cloud (Azure)
  – Same dataset
  – More complex queries 😊

• Logistics
  – You will receive an email from invites@microsoft.com to join the “Default Directory organization” --- accept it!
  – You are allocated $100 to use for this quarter
  – We will use Azure for two HW assignments
  – Use SQL Server Management Studio to access the DB
    • Installed on all CSE lab machines and VDI machines
# Scythe

## CSE 344 SQL Synthesizer

Synthesize queries from newly created I/O tables or provided examples!

**WARNING!**

The purpose of this webtool is to help you getting a better understanding of SQL queries, not to do your assignments for you! Multiple queries may output the same result for one particular I/O example, but they are not necessarily equivalent (due to lack of data or wrong specification). Please study the queries thoroughly and use it wisely.

---

**Input Table 1**

<table>
<thead>
<tr>
<th>c0</th>
<th>c1</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

**Output Table**

<table>
<thead>
<tr>
<th>c0</th>
<th>c1</th>
<th>c2</th>
</tr>
</thead>
<tbody>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
<tr>
<td>0</td>
<td>0</td>
<td>0</td>
</tr>
</tbody>
</table>

---

**Buttons**

- Create New Panel
- Load Example Panel
- Add Row
- Add Column
- Remove Column
- Synthesize

---

**Constant**

| None |

**Aggregators**

| (Optional) |

**Select Query**

No query to display yet.
Plan for Today

• Wrap up SQL

• Study two other languages for the relational data model
  – Relational algebra
  – Datalog
Reading Assignment 2

• Normal form

• Compositionality of relations and operators

\[ \text{foo}(\ldots) \cdot \text{bar}(\ldots) \]

\[ t = \text{foo}(\ldots) \]

\[ = t \cdot \text{bar}(\ldots) \]
Review

• SQL
  – Selection
  – Projection
  – Join
  – Ordering
  – Grouping
  – Aggregates
  – Subqueries

• Query Evaluation
Monotone Queries

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

<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>c004</td>
</tr>
<tr>
<td>Camera</td>
<td>c003</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>c004</td>
</tr>
<tr>
<td>Camera</td>
<td>c003</td>
</tr>
<tr>
<td>iPad</td>
<td>c001</td>
</tr>
</tbody>
</table>
SQL Idioms
Including Empty Groups

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

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

Count(*) is never 0
Including Empty Groups

\[
\text{SELECT } x.\text{manufacturer}, \text{count}(y.\text{pid}) \\
\text{FROM Product } x \text{ LEFT OUTER JOIN Purchase } y \\
\text{ON } x.\text{pname} = y.\text{product} \\
\text{GROUP BY } x.\text{manufacturer}
\]

Count(pid) is 0 when all pid’s in the group are NULL
GROUP BY vs. Nested Queries

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

SELECT DISTINCT x.product, (SELECT Sum(y.quantity)
    FROM Purchase y
    WHERE x.product = y.product
    AND y.price > 1)
    AS TotalSales
FROM Purchase x
WHERE x.price > 1

Why twice?
Author(login, name) 
Wrote(login, url)

More Unnesting

Find authors who wrote ≥ 10 documents:
Find authors who wrote $\geq 10$ documents:

Attempt 1: with nested queries

```
SELECT DISTINCT Author.name 
FROM Author 
WHERE (SELECT count(Wrote.url) 
    FROM Wrote 
    WHERE Author.login=Wrote.login) 
   $\geq 10$
```
Find authors who wrote $\geq 10$ documents:

Attempt 1: with nested queries

Attempt 2: using GROUP BY and HAVING

```
SELECT Author.name
FROM Author, Wrote
WHERE Author.login=Wrote.login
GROUP BY Author.name
HAVING count(wrote.url) $\geq$ 10
```

This is SQL by an expert
Product \((\text{pname, price, cid})\)
Company \((\text{cid, cname, city})\)

Finding Witnesses

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

For each city, find the most expensive product made in that city
Finding the maximum price is easy...

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

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

To find the witnesses, compute the maximum price in a subquery

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

Or we can use a subquery in where clause

```
SELECT u.city, v.pname, v.price
FROM Company u, Product v
WHERE u.cid = v.cid
  AND v.price >= ALL (SELECT y.price
                       FROM Company x, Product y
                       WHERE u.city=x.city
                       AND x.cid=y.cid);
```
Finding Witnesses

There is a more concise solution here:

```
SELECT u.city, v.pname, v.price
FROM Company u, Product v, Company x, Product y
WHERE u.cid = v.cid and u.city = x.city
      and x.cid = y.cid
GROUP BY u.city, v.pname, v.price
HAVING v.price = max(y.price)
```
SQL: Our first language for the relational model

- Projections
- Selections
- Joins (inner and outer)
- Inserts, updates, and deletes
- Aggregates
- Grouping
- Ordering
- Nested queries
Relational Algebra
Class overview

• Data models
  – Relational: SQL, RA, and Datalog
  – NoSQL: SQL++

• RDMBS internals
  – Query processing and optimization
  – Physical design

• Parallel query processing
  – Spark and Hadoop

• Conceptual design
  – E/R diagrams
  – Schema normalization

• Transactions
  – Locking and schedules
  – Writing DB applications
Next: Relational Algebra

• Our second language for the relational model
  – Developed before SQL
  – Simpler syntax than SQL
Why bother with another language?

• Used extensively by DBMS implementations
  – As we will see in 2 weeks

• RA influences the design SQL
Relational Algebra

• In SQL we say *what* we want
• In RA we can express *how* to get it
• Set-at-a-time algebra, which manipulates relations
• Every RDBMS implementations converts a SQL query to RA in order to execute it

• An RA expression is also called a *query plan*
Basics

• Relations and attributes

• Functions that are applied to relations
  – Return relations
  – Can be composed together
  – Often displayed using a tree rather than linearly
  – Use Greek symbols: σ, π, δ, etc
Sets v.s. Bags

- Sets: \{a,b,c\}, \{a,d,e,f\}, \{\}\ldots
- Bags: \{a, a, b, c\}, \{b, b, b, b, b\}, \ldots

Relational Algebra has two flavors:
- Set semantics = standard Relational Algebra
- Bag semantics = extended Relational Algebra

DB systems implement bag semantics (Why?)
Relational Algebra Operators

- Union $\cup$, intersection $\cap$, difference $-$
- Selection $\sigma$
- Projection $\pi$
- Cartesian product $\times$, join $\Join$
- (Rename $\rho$)
- Duplicate elimination $\delta$
- Grouping and aggregation $\gamma$
- Sorting $\tau$

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

R1 ∪ R2
R1 – R2

Only make sense if R1, R2 have the same schema

What do they mean over bags?
What about Intersection?

- Derived operator using minus

\[ R_1 \cap R_2 = R_1 - (R_1 - R_2) \]

- Derived using join

\[ R_1 \cap R_2 = R_1 \bowtie R_2 \]
Selection

• Returns all tuples which satisfy a condition

\[ \sigma_c(R) \]

• Examples
  - \( \sigma \text{Salary} > 40000 \) (Employee)
  - \( \sigma \text{name} = \text{"Smith"} \) (Employee)

• The condition c can be =, <, <=, >, >=, <> combined with AND, OR, NOT
### Employee

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>5423341</td>
<td>Smith</td>
<td>60000</td>
</tr>
<tr>
<td>4352342</td>
<td>Fred</td>
<td>50000</td>
</tr>
</tbody>
</table>

\[ \sigma_{\text{Salary} > 40000} (\text{Employee}) \]

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>5423341</td>
<td>Smith</td>
<td>60000</td>
</tr>
<tr>
<td>4352342</td>
<td>Fred</td>
<td>50000</td>
</tr>
</tbody>
</table>
Projection

- Eliminates columns

\[ \pi_{A_1,\ldots,A_n}(R) \]

- Example: project social-security number and names:

\[ \pi_{\text{SSN, Name}}(\text{Employee}) \rightarrow \text{Answer}(\text{SSN, Name}) \]

Different semantics over sets or bags! Why?
### Employee

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>5423341</td>
<td>John</td>
<td>60000</td>
</tr>
<tr>
<td>4352342</td>
<td>John</td>
<td>20000</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{Name},\text{Salary}} (\text{Employee}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>John</td>
<td>60000</td>
</tr>
<tr>
<td>John</td>
<td>20000</td>
</tr>
</tbody>
</table>

### Bag semantics

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>John</td>
<td>60000</td>
</tr>
<tr>
<td>John</td>
<td>20000</td>
</tr>
</tbody>
</table>

### Set semantics

<table>
<thead>
<tr>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>20000</td>
</tr>
<tr>
<td>John</td>
<td>60000</td>
</tr>
</tbody>
</table>

Which is more efficient?
**Functional Composition of RA Operators**

<table>
<thead>
<tr>
<th>no</th>
<th>name</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>p1</td>
<td>98125</td>
<td>flu</td>
</tr>
<tr>
<td>2</td>
<td>p2</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>3</td>
<td>p3</td>
<td>98120</td>
<td>lung</td>
</tr>
<tr>
<td>4</td>
<td>p4</td>
<td>98120</td>
<td>heart</td>
</tr>
</tbody>
</table>

\[\sigma_{\text{disease}='\text{heart'}(\text{Patient})\}

<table>
<thead>
<tr>
<th>no</th>
<th>name</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>p2</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>4</td>
<td>p4</td>
<td>98120</td>
<td>heart</td>
</tr>
</tbody>
</table>

\[\pi_{\text{zip,disease}}(\text{Patient})\}

<table>
<thead>
<tr>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>98125</td>
<td>flu</td>
</tr>
<tr>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>98120</td>
<td>lung</td>
</tr>
<tr>
<td>98120</td>
<td>heart</td>
</tr>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>98120</td>
<td>heart</td>
</tr>
</tbody>
</table>
Cartesian Product

- Each tuple in R1 with each tuple in R2

\[ R1 \times R2 \]

- Rare in practice; mainly used to express joins
### Cross-Product Example

#### Employee

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
</tr>
</tbody>
</table>

#### Dependent

<table>
<thead>
<tr>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

#### Employee X Dependent

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>999999999</td>
<td>999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>John</td>
<td>999999999</td>
<td>777777777</td>
<td>Joe</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
<td>999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>Tony</td>
<td>777777777</td>
<td>777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>
Renaming

• Changes the schema, not the instance

\[ \rho_{B_1, \ldots, B_n}(R) \]

• Example:
  – Given Employee(Name, SSN)
  – \( \rho_{N, S}(Employee) \) → Answer(N, S)
Natural Join

\[ R_1 \bowtie R_2 \]

- Meaning: \( R_1 \bowtie R_2 = \pi_A(\sigma_\theta(R_1 \times R_2)) \)

- Where:
  - Selection \( \sigma_\theta \) checks equality of all common attributes (i.e., attributes with same names)
  - Projection \( \pi_A \) eliminates duplicate common attributes
Natural Join Example

Let's consider two relations, $R$ and $S$, and perform a natural join on their common columns $B$. The natural join of $R$ and $S$, denoted as $R \bowtie S$, is defined as:

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

The tables for $R$ and $S$ are as follows:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>4</td>
<td>Z</td>
<td>V</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th></th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>2</td>
<td>V</td>
<td>W</td>
</tr>
<tr>
<td>3</td>
<td>Z</td>
<td>V</td>
</tr>
</tbody>
</table>

After performing the join, we get:

<table>
<thead>
<tr>
<th></th>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>X</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>2</td>
<td>X</td>
<td>Z</td>
<td>V</td>
</tr>
<tr>
<td>3</td>
<td>Y</td>
<td>Z</td>
<td>U</td>
</tr>
<tr>
<td>4</td>
<td>Y</td>
<td>Z</td>
<td>V</td>
</tr>
<tr>
<td>5</td>
<td>Z</td>
<td>V</td>
<td>W</td>
</tr>
</tbody>
</table>
Natural Join Example 2

AnonPatient $P$

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

Voters $V$

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>Alice</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>Bob</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

$P \bowtie V$

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>Alice</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>Bob</td>
</tr>
</tbody>
</table>
Natural Join

• Given schemas $R(A, B, C, D)$, $S(A, C, E)$, what is the schema of $R \bowtie S$ ?

• Given $R(A, B, C)$, $S(D, E)$, what is $R \bowtie S$?

• Given $R(A, B)$, $S(A, B)$, what is $R \bowtie S$?
Theta Join

- A join that involves a predicate

\[ R1 \bowtie_\theta R2 = \sigma_\theta (R1 \times R2) \]

- Here \( \theta \) can be any condition
- No projection in this case!
- For our voters/patients example:

\[ P \bowtie \quad P.zip = V.zip \text{ and } P.age \geq V.age - 1 \text{ and } P.age \leq V.age + 1 \]
Equijoin

• A theta join where $\theta$ is an equality predicate

\[
R_1 \join_\theta R_2 = \sigma_\theta (R_1 \times R_2)
\]

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

AnonPatient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
</tbody>
</table>

Voters V

<table>
<thead>
<tr>
<th>name</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

\[ P \bowtie_{P.\text{age}=V.\text{age}} V \]

<table>
<thead>
<tr>
<th>P.\text{age}</th>
<th>P.\text{zip}</th>
<th>P.\text{disease}</th>
<th>V.\text{name}</th>
<th>V.\text{age}</th>
<th>V.\text{zip}</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>
Join Summary

• **Theta-join**: \( R \bowtie_{\theta} S = \sigma_{\theta} (R \times S) \)
  
  – Join of \( R \) and \( S \) with a join condition \( \theta \)
  
  – Cross-product followed by selection \( \theta \)
  
  – No projection

• **Equijoin**: \( R \bowtie_{\theta} S = \sigma_{\theta} (R \times S) \)
  
  – Join condition \( \theta \) consists only of equalities
  
  – No projection

• **Natural join**: \( R \Join S = \pi_A (\sigma_{\theta} (R \times S)) \)
  
  – Equality on all fields with same name in \( R \) and in \( S \)
  
  – Projection \( \pi_A \) drops all redundant attributes
So Which Join Is It?

When we write $R \bowtie S$ we usually mean an equijoin, but we often omit the equality predicate when it is clear from the context.
More Joins

• **Outer join**
  – Include tuples with no matches in the output
  – Use NULL values for missing attributes
  – Does not eliminate duplicate columns

• **Variants**
  – Left outer join
  – Right outer join
  – Full outer join
Outer Join Example

AnonPatient P

<table>
<thead>
<tr>
<th>age</th>
<th>zip</th>
<th>disease</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
</tr>
<tr>
<td>33</td>
<td>98120</td>
<td>lung</td>
</tr>
</tbody>
</table>

AnonJob J

<table>
<thead>
<tr>
<th>job</th>
<th>age</th>
<th>zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>lawyer</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>cashier</td>
<td>20</td>
<td>98120</td>
</tr>
</tbody>
</table>

P ⋈ J

<table>
<thead>
<tr>
<th>P.age</th>
<th>P.zip</th>
<th>P.disease</th>
<th>J.job</th>
<th>J.age</th>
<th>J.zip</th>
</tr>
</thead>
<tbody>
<tr>
<td>54</td>
<td>98125</td>
<td>heart</td>
<td>lawyer</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>cashier</td>
<td>20</td>
<td>98120</td>
</tr>
<tr>
<td>33</td>
<td>98120</td>
<td>lung</td>
<td>null</td>
<td>null</td>
<td>null</td>
</tr>
</tbody>
</table>
Some Examples

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

Name of supplier of parts with size greater than 10
\[ \pi_{sname}(\text{Supplier} \join \text{Supply} \join (\sigma_{\text{psize}>10}(\text{Part}))) \]

Name of supplier of red parts or parts with size greater than 10
\[ \pi_{sname}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{\text{psize}>10}(\text{Part}) \cup \sigma_{\text{pcolor}='red'}(\text{Part})) ) \]
\[ \pi_{sname}(\text{Supplier} \bowtie \text{Supply} \bowtie (\sigma_{\text{psize}>10 \lor \text{pcolor}='red'}(\text{Part}))) \]

Can be represented as trees as well
Representing RA Queries as Trees

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

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

\[ \pi_{sname} \]

\[ \sigma_{\text{psize}>10} \]

Part

Supplier

Supply

Answer
Relational Algebra Operators

- **Union** \( \cup \), **intersection** \( \cap \), **difference** -
- **Selection** \( \sigma \)
- **Projection** \( \pi \)
- **Cartesian product** \( X \), **join** \( \Join \)
- **(Rename** \( \rho \))
- **Duplicate elimination** \( \delta \)
- **Grouping and aggregation** \( \gamma \)
- **Sorting** \( \tau \)

All operators take in 1 or more relations as inputs and return another relation.
Extended RA: Operators on Bags

- **Duplicate elimination** $\delta$
- **Grouping** $\gamma$
  - Takes in relation and a list of grouping operations (e.g., aggregates). Returns a new relation.
- **Sorting** $\tau$
  - Takes in a relation, a list of attributes to sort on, and an order. Returns a new relation.
Using Extended RA Operators

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

**Answer**

\[ \pi_{\text{city}, q} \left( T_2(\text{city}, q, c) \right) \]

\[ \sigma_{c > 100} \left( \text{city, sum(quantity)} \rightarrow q, \text{count(*)} \rightarrow c \right) \]

\[ \bigcup_{\text{city, sum(quantity)} \rightarrow q, \text{count(*)} \rightarrow c} \]

**T1, T2** = temporary tables

sales(product, city, quantity)
Typical Plan for a Query (1/2)

Answer

\[ \pi_{\text{fields}} \]

\[ \sigma_{\text{selection condition}} \]

SELECT-PROJECT-JOIN Query

\[ \text{SELECT fields FROM } R, S, \ldots \text{ WHERE condition} \]
SELECT fields
FROM R, S, ...
WHERE condition
GROUP BY fields
HAVING condition
How about Subqueries?

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

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

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

De-Correlation

```
SELECT Q.sno
FROM Supplier Q
WHERE Q.sstate = 'WA'
and Q.sno not in
(SELECT P.sno
FROM Supply P
WHERE P.price > 100)
```
How about Subqueries?

(\text{SELECT} \ Q.sno \\
\text{FROM} \ Supplier \ Q \\
\text{WHERE} \ Q.sstate = \text{‘WA’})

\text{EXCEPT}

(\text{SELECT} \ P.sno \\
\text{FROM} \ Supply \ P \\
\text{WHERE} \ P.price > 100)

\text{EXCEPT} = \text{set difference}

\text{SELECT} \ Q.sno \\
\text{FROM} \ Supplier \ Q \\
\text{WHERE} \ Q.sstate = \text{‘WA’} \text{ and } Q.sno \text{ not in}

(\text{SELECT} \ P.sno \\
\text{FROM} \ Supply \ P \\
\text{WHERE} \ P.price > 100)

\text{Un-nesting}
(SELECT Q.sno FROM Supplier Q WHERE Q.sstate = 'WA')
EXCEPT
(SELECT P.sno FROM Supply P WHERE P.price > 100)

Finally…

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

CSEP 544 - Fall 2017
62
Summary of RA and SQL

- SQL = a declarative language where we say *what* data we want to retrieve
- RA = an algebra where we say *how* we want to retrieve the data
- Both implements the relational data model
- **Theorem**: SQL and RA can express exactly the same class of queries
- RDBMS translate SQL → RA, then optimize RA
Summary of RA and SQL

• SQL (and RA) cannot express ALL queries that we could write in, say, Java

• Example:
  – Parent(p,c): find all descendants of ‘Alice’
  – No RA query can compute this!
  – This is called a recursive query

• Next: Datalog is an extension that can compute recursive queries
Summary of RA and SQL

• Translating from SQL to RA gives us a way to evaluate the input query

• Transforming one RA plan to another forms the basis of *query optimization*

• Will see more in 2 weeks
Datalog
What is Datalog?

• Another *declarative* query language for relational model
  – Designed in the 80’s
  – Minimal syntax
  – Simple, concise, elegant
  – Extends relational queries with *recursion*

• Today:
  – Adopted by some companies for data analytics, e.g., LogicBlox (HW4)
  – Usage beyond databases: e.g., network protocols, static program analysis
SQL Query vs Datalog
(which would you rather write?)
(any Java fans out there?)
HW4: Preview

```plaintext
Welcome to the LogicBlox playground!

8-14 /*
  addblock 'r(x,y) -> int(x), int(y).'
=>
Successfully added block 'block_1Z331BSE'
exec '+r(1,2). +r(2,1). +r(2,3). +r(1,4). +r(3,4). +r(4,5).'
8-14 /*
print r
=>

<p>| | | | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>1</td>
<td></td>
<td>4</td>
<td></td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td>1</td>
<td>3</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td></td>
<td></td>
<td>4</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td></td>
<td></td>
<td></td>
<td>5</td>
</tr>
<tr>
<td>4</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
```
Datalog: Facts and Rules

Facts = tuples in the database

Rules = queries

Schema

Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)
Datalog: Facts and Rules

**Facts** = tuples in the database

**Rules** = queries

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(29445, ‘Ave Maria’, 1940).
Datalog: Facts and Rules

**Facts** = tuples in the database

- Actor(344759, ‘Douglas’, ‘Fowley’).
- Casts(344759, 29851).
- Casts(355713, 29000).

**Rules** = queries

- Q1(y) :- Movie(x,y,z), z='1940'.
**Facts** = tuples in the database

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(29445, ‘Ave Maria’, 1940).

**Rules** = queries

Q1(y) :- Movie(x,y,z), z=’1940’.

**Find Movies made in 1940**
<table>
<thead>
<tr>
<th>Facts = tuples in the database</th>
</tr>
</thead>
<tbody>
<tr>
<td>Actor(344759, ‘Douglas’, ‘Fowley’).</td>
</tr>
<tr>
<td>Casts(344759, 29851).</td>
</tr>
<tr>
<td>Casts(355713, 29000).</td>
</tr>
<tr>
<td>Movie(29445, ‘Ave Maria’, 1940).</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Rules = queries</th>
</tr>
</thead>
<tbody>
<tr>
<td>Q1(y) :- Movie(x, y, z), z = ‘1940’.</td>
</tr>
<tr>
<td>Q2(f, l) :- Actor(z, f, l), Casts(z, x), Movie(x, y, ‘1940’).</td>
</tr>
</tbody>
</table>
Actor(id, fname, lname)
Casts(pid, mid)
Movie(id, name, year)

Datalog: Facts and Rules

Facts = tuples in the database

- Actor(344759, ‘Douglas’, ‘Fowley’).
- Casts(344759, 29851).
- Casts(355713, 29000).

Rules = queries

- Q1(y) :- Movie(x,y,z), z='1940'.
- Q2(f, l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,’1940’).

Find Actors who acted in Movies made in 1940
Datalog: Facts and Rules

Facts = tuples in the database

Actor(344759, ‘Douglas’, ‘Fowley’).
Casts(344759, 29851).
Casts(355713, 29000).
Movie(29445, ‘Ave Maria’, 1940).

Rules = queries

Q1(y) :- Movie(x,y,z), z=’1940’.
Q2(f,l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,’1940’).
Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910),
         Casts(z,x2), Movie(x2,y2,1940).
Datalog: Facts and Rules

**Facts** = tuples in the database

- Actor(344759, ‘Douglas’, ‘Fowley’).
-casts(344759, 29851).
- Casts(355713, 29000).

**Rules** = queries

- Q1(y) :- Movie(x,y,z), z=’1940’.
- Q2(f, l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,’1940’).
- Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910), Casts(z,x2), Movie(x2,y2,1940).

Find Actors who acted in a Movie in 1940 and in one in 1910.
Datalog: Facts and Rules

**Facts** = tuples in the database

- Actor(344759, ‘Douglas’, ‘Fowley’).
- Casts(344759, 29851).
- Casts(355713, 29000).

**Rules** = queries

- Q1(y) :- Movie(x,y,z), z='1940'.
- Q2(f, l) :- Actor(z,f,l), Casts(z,x), Movie(x,y,’1940’).
- Q3(f,l) :- Actor(z,f,l), Casts(z,x1), Movie(x1,y1,1910), Casts(z,x2), Movie(x2,y2,1940).

**Extensional Database Predicates** = EDB = Actor, Casts, Movie

**Intensional Database Predicates** = IDB = Q1, Q2, Q3
Datalog: Terminology

In this class we discuss datalog evaluated under set semantics.

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

- **f, l** = head variables
- **x,y,z** = existential variables
More Datalog Terminology

- $R_i(\text{args}_i)$ is called an atom, or a relational predicate
- $R_i(\text{args}_i)$ evaluates to true when relation $R_i$ contains the tuple described by $\text{args}_i$.
  - Example: $\text{Actor}(344759, 'Douglas', 'Fowley')$ is true
- In addition to relational predicates, we can also have arithmetic predicates
  - Example: $z > '1940'$.
- Note: Logicblox uses $\leq$ instead of $:$

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

Your book uses:
Q(args) :- R1(args) AND R2(args) AND ....
Semantics of a Single Rule

• Meaning of a datalog rule = a logical statement!

\[ Q1(y) : - \text{Movie}(x,y,z), z='1940'. \]

• For all values of \( x, y, z \):
  
  if \((x,y,z)\) is in the Movies relation, and that \( z = '1940' \)
  
  then \( y \) is in \( Q1 \) (i.e., it is part of the answer)

• Logically equivalent:

\[ \forall y. [ (\exists x. \exists z. \text{Movie}(x,y,z) \text{ and } z='1940') \Rightarrow Q1(y) ] \]

• That's why head variables are called "existential variables"

• We want the \textit{smallest} set \( Q1 \) with this property (why?)
Datalog program

- A datalog program consists of several rules
- Importantly, rules may be recursive!
- Usually there is one distinguished predicate that’s the output
- We will show an example first, then give the general semantics.
R encodes a graph

\[
\begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]
Example

R encodes a graph

\[ R = \]

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

What does it compute?

\[
T(x,y) \ :- \ R(x,y) \\
T(x,y) \ :- \ R(x,z), T(z,y)
\]
Example

R encodes a graph

![Graph Diagram]

R =

<p>| | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>

Initially:
T is empty.

What does it compute?

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

R encodes a graph

What does it compute?

First iteration:

First rule generates this

Second rule generates nothing (because T is empty)

Initially:
T is empty.

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

R =

<table>
<thead>
<tr>
<th>1</th>
<th>2</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>4</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
</tbody>
</table>
Example

R encodes a graph

\[ R(x,y) : - R(x,y), T(x,z), T(z,y) \]

Initially:
T is empty.

First iteration:
T =

\[
\begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]

Second iteration:
T =

\[
\begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
1 & 1 \\
2 & 2 \\
1 & 3 \\
2 & 4 \\
1 & 5 \\
3 & 5 \\
\end{array}
\]

What does it compute?

First rule generates this

Second rule generates this

New facts
Example

R encodes a graph

\[ R = \begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array} \]

Initially:
T is empty.

First iteration:

\[ T = \begin{array}{ccc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array} \]

Second iteration:

\[ T = \begin{array}{cccc}
1 & 2 & 5 \\
2 & 1 & 5 \\
2 & 3 & 5 \\
1 & 4 & 5 \\
3 & 4 & 5 \\
4 & 5 & 5 \\
\end{array} \]

Third iteration:

\[ T = \begin{array}{cc}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array} \]

What does it compute?

R encodes a graph

\[ T(x,y) :- R(x,y) \]
\[ T(x,y) :- R(x,z), T(z,y) \]

First rule

Second rule

Both rules

New fact

First iteration:

T =

Second iteration:

T =

Third iteration:
Example

R encodes a graph

\[
R = \begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}
\]

First iteration:

T = \begin{array}{c|c}
1 & 2 \\
2 & 1 \\
2 & 3 \\
1 & 4 \\
3 & 4 \\
4 & 5 \\
\end{array}

Second iteration:

Third iteration:

Fourth iteration

[Same as previous iteration]

This is called the **fixpoint semantics** of a datalog program.

\[
T(x,y) \text{ :- } R(x,y)
\]

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

What does it compute?

No new facts.

DONE