Introduction to Database Systems
CSE 414

Lecture 7: SQL Wrap-up and Relational Algebra
Announcements

• Additional Office Hours and room changes
  – Website calendar is up-to-date

• Check email for Microsoft Azure invite
  “Action required: Accept your Azure lab assignment”
Subqueries

• A subquery is a SQL query nested inside a larger query
• Such inner-outer queries are called nested queries
• A subquery may occur in:
  – A SELECT clause
    • Must return single value
  – A FROM clause
    • Can return multi-valued relation
  – A WHERE clause

• Rule of thumb: avoid nested queries when possible
  – But sometimes it’s impossible, as we will see
Subqueries in FROM

Sometimes we need to compute an intermediate table only to use it later in a SELECT-FROM-WHERE

- Option 1: use a subquery in the FROM clause
- Option 2: use the WITH clause
  – See textbook for details
Product (pname, price, cid)
Company (cid, cname, city)

2. Subqueries in FROM

```sql
WITH myTable AS (SELECT * FROM Product AS Y WHERE price > 20)
SELECT X.pname
FROM myTable as X
WHERE X.price < 500
```

A subquery whose result we called myTable
Subqueries in WHERE

- SELECT .......... WHERE EXISTS (sub);
- SELECT .......... WHERE NOT EXISTS (sub);
- SELECT .......... WHERE attribute IN (sub);
- SELECT .......... WHERE attribute NOT IN (sub);
- SELECT .......... WHERE attribute > ANY (sub);
- SELECT .......... WHERE attribute > ALL (sub);
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.
Monotone Queries

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

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

• **Proof.** We use the nested loop semantics: if we insert a tuple in a relation $R_i$, this will not remove any tuples from the answer.

```sql
SELECT a_1, a_2, ..., a_k
FROM R_1 AS x_1, R_2 AS x_2, ..., R_n AS x_n
WHERE Conditions
```
Monotone Queries

- The query:

Find all companies s.t. all their products have price < 200
is not monotone
Monotone Queries

- The query:

Find all companies s.t. all their products have price $< 200$
is not monotone

<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>
</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>

<table>
<thead>
<tr>
<th>cname</th>
</tr>
</thead>
<tbody>
<tr>
<td>Sunworks</td>
</tr>
</tbody>
</table>
Monotone Queries

• The query:

Find all companies s.t. all their products have price < 200 is not monotone

• Consequence: If a query is not monotonic, then we cannot write it as a SELECT-FROM-WHERE query without nested subqueries
Queries that must be nested

- Queries with universal quantifiers or with negation
Queries that must be nested

• Queries with universal quantifiers or with negation

• Queries that use aggregates in certain ways
  – sum(..) and count(*) are NOT monotone, because they do not satisfy set containment
  – select count(*) from R is not monotone!
SQL Idioms
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 (in FROM or in WITH)

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

To find the witnesses, compute the maximum price in a subquery (in FROM or in WITH)

```
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

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
Relational Algebra

• Set-at-a-time algebra, which manipulates relations
• In SQL we say *what* we want
• In RA we can express *how* to get it
• Every DBMS implementation converts a SQL query to RA in order to execute it
• An RA expression is called a *query plan*
Why study another relational query language?

• RA is how SQL is implemented in DBMS
  – We will see more of this in a few weeks

• RA opens up opportunities for query optimization
Basics

- Relations and attributes
- Functions that are applied to relations
  - Return relations
    \[ R_2 = \sigma (R_1) \]
  - Can be composed together
    \[ R_3 = \pi (\sigma (R_1)) \]
  - Often displayed using a tree rather than linearly
  - Use Greek symbols: \( \sigma, \pi, \delta \), etc
Sets v.s. Bags

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

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 $\bowtie$
- 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 \cup 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 $=$, $<$, $\leq$, $>$, $\geq$, $\neq$ 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:
  $$\pi_{\text{SSN}, \text{Name}}(\text{Employee}) \rightarrow \text{Answer(SSN, Name)}$$

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

<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 with 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>

### Table with 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?
Composing 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>

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

<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>

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

<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)
  – \( \rho_{N, S}(Employee) \) \( \rightarrow \) 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

\[ R \Join S = \Pi_{ABC}(\sigma_{R.B=S.B}(R \times S)) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
</tr>
<tr>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>Z</td>
<td>V</td>
</tr>
</tbody>
</table>

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

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

<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>

AnonPatient P

<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>

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 \Join 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

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

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

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

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

$$R1 \bowtie_{\theta} R2 = \sigma_{\theta} (R1 \times R2)$$

- 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>

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

<table>
<thead>
<tr>
<th>P.age</th>
<th>P.zip</th>
<th>P.disease</th>
<th>V.name</th>
<th>V.age</th>
<th>V.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**: $R \bowtie S = \pi_A (\sigma_\theta (R \times S))$
  - 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
Project[sname](Supplier Join[sno=sno]
  (Supply Join[pno=pno] (Select[psize>10](Part))))

Using symbols:
πsname(Supplier ⨝ (Supply ⨝ (σpsize>10 (Part))))

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

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

Answer

\[
\pi_{\text{sname}}
\]

Supplier

Supply

Part

σ_{\text{psize}>10}

CSE 414 - Autumn 2018
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
Project[sname](Supplier Join[sno=sno]
  (Supply Join[pno=pno] (Select[psize>10](Part))))

Name of supplier of red parts or parts with size greater than 10
Project[sname](Supplier Join[sno=sno]
  (Supply Join[pno=pno]
   ((Select[psize>10](Part)) Union
    (Select[pcolor='red'](Part))))

Can be represented as trees as well
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
\[ \pi_{sname}(\text{Supplier} \bowtie (\text{Supply} \bowtie (\sigma_{\text{psize}>10} (\text{Part})))) \]

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

Can be represented as trees as well
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
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

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

\[
\sigma_{c > 100}
\]

\[
\Pi_{\text{city, } q}
\]

\[
\text{Sales(product, city, quantity)}
\]
Typical Plan for a Query (1/2)

Answer

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

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

join condition

\[
\text{SELECT fields} \\
\text{FROM } R, S, \ldots \\
\text{WHERE } \text{condition}
\]

SELECT-PROJECT-JOIN

Query
Typical Plan for a Query (1/2)

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

\[ \gamma_{\text{fields, sum/count/min/max(fields)}} \]

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

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

join condition

\[ \ldots \]

\[ \ldots \]

SELECT fields FROM R, S, ... WHERE condition GROUP BY fields HAVING condition
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?

Option 1: create nested plans

```
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) 
```

```
π_{sno} 
σ_{sstate='WA'}(Supplier) 
```

```
σ_{price>100}(Supplier) 
```
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?

Suppliers

\[ \text{Supplier(sno, sname, scity, sstate)} \]
\[ \text{Part(pno, pname, psize, pcolor)} \]
\[ \text{Supply(sno, pno, price)} \]
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)
```

**De-Correlation**

```sql
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?

(SELECT Q.sno
 FROM Supplier Q
 WHERE Q.sstate = 'WA')
 EXCEPT
 (SELECT P.sno
  FROM Supply P
  WHERE P.price > 100)
 EXCEPT = set difference
How about Subqueries?

\[
\begin{align*}
\text{(SELECT} & \quad Q.\text{sno} \\
\text{FROM} & \quad \text{Supplier } Q \\
\text{WHERE} & \quad Q.\text{sstate} = 'WA') \\
\text{EXCEPT} & \quad (\text{SELECT} \quad P.\text{sno} \\
\text{FROM} & \quad \text{Supply } P \\
\text{WHERE} & \quad P.\text{price} > 100) \end{align*}
\]

Finally…

\[
\begin{align*}
\Pi_{\text{sno}} & \quad \eta \\
\sigma_{\text{sstate}=\text{'WA'}} & \quad \Pi_{\text{sno}} \\
\sigma_{\text{Price} > 100} & \quad \text{Supplier} \\
& \quad \text{Supply}
\end{align*}
\]

Supplier(sno,sname,scity,sstate)
Part(pno,pname,psize,pcolor)
Supply(sno,pno,price)
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
• **Theorem**: SQL and RA can express exactly the same class of queries

RDBMS translate SQL $\rightarrow$ 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 lecture: Datalog is an extension that can compute recursive queries