DATA 514 Lecture 3

SQL Wrap-up
Relational Algebra
Announcements

• HW2 – deadline extended until tomorrow
• WQ3 is open, due on Tuesday

• Homework 3 will be posted tomorrow, due on Feb 3
  – We are using Microsoft Azure Cloud services!
  – Wait for instructions to be posted
Recap from last lectures

- Subqueries can occur in every clause:
  - SELECT
  - FROM
  - WHERE
- Monotone queries: SELECT-FROM-WHERE
  - Existential quantifier
- Non-monotone queries
  - Universal quantifier
  - Aggregation
Examples of Complex Queries

\[
\text{Likes(drinker, beer)} \\
\text{Frequents(drinker, bar)} \\
\text{Serves(bar, beer)}
\]

1. Find drinkers that frequent some bar that serves some beer they like.

2. Find drinkers that frequent some bar that serves only beers they don’t like.

3. Find drinkers that frequent only bars that serves some beer they like.
Example 1

Find drinkers that frequent some bar that serves some beer they like.
Example 1

Find drinkers that frequent some bar that serves some beer they like.

```
SELECT DISTINCT X.drinker
FROM Frequents X, Serves Y, Likes Z
WHERE X.bar = Y.bar
AND Y.beer = Z.beer
AND X.drinker = Z.drinker
```
Example 1

Find drinkers that frequent some bar that serves some beer they like.

```
SELECT DISTINCT X.drinker
FROM Frequents X, Serves Y, Likes Z
WHERE X.bar = Y.bar
AND Y.beer = Z.beer
AND X.drinker = Z.drinker
```

What happens if we didn’t write DISTINCT?
Example 2

Find drinkers that frequent some bar that serves only beers they don’t like.
Example 2

Find drinkers that frequent some bar that serves only beers they don’t like.

Let’s check if the drinker frequents one of the other bars.
Example 2

Find drinkers that frequent some bar that serves only beers they don’t like.

Let’s check if the drinker frequents one of the other bars.

Drinkers that frequent some bars that serves some beer they like.
Example 2

Find drinkers that frequent some bar that serves only beers they don’t like.

Let’s check if the drinker frequents one of the other bars.

Drinkers that frequent some bars that serves some beer they like.

That’s the previous query… but let’s write it with a subquery:

```
SELECT DISTINCT X.dinker
FROM Frequents X
WHERE EXISTS (
    SELECT *
    FROM Serves Y, Likes Z
    WHERE X.bar = Y.bar
    AND X.drinker = Z.drinker
    AND Y.beer = Z.beer)
```
Example 2

Find drinkers that frequent some bar that serves only beers they don’t like.

Let’s check if the drinker frequents one of the other bars

Drinkers that frequent some bars that serves some beer they like.

That’s the previous query… but let’s write it with a subquery:

Now negate!

```
SELECT DISTINCT X.drinker
FROM Frequents X
WHERE NOT EXISTS (SELECT *
FROM Serves Y, Likes Z
WHERE X.bar=Y.bar
AND X.drinker=Z.drinker
AND Y.beer = Z.beer)
```
Example 3

Find drinkers that frequent only bars that serves some beer they like.
Example 3

Find drinkers that frequent **only** bars that serves **some** beer they like.

Let’s find the other drinkers
Example 3

Find drinkers that frequent only bars that serves some beer they like.

Let’s find the other drinkers

Drinkers that frequent some bar that serves only beers they don’t like
Example 3

Find drinkers that frequent only bars that serves some beer they like.

Let’s find the other drinkers

Drinkers that frequent some bar that serves only beers they don’t like

That’s the previous query!

```
SELECT X.drinker
FROM Frequents X
WHERE NOT EXISTS (SELECT *
 FROM Serves Y, Likes Z
 WHERE X.bar=Y.bar
 AND X.drinker=Z.drinker
 AND Y.beer = Z.beer)
```
Example 3

Find drinkers that frequent only bars that serves some beer they like.

Let’s find the other drinkers

Drinkers that frequent some bar that serves only beers they don’t like

That’s the previous query!

Now find the other drinkers:

```
SELECT DISTINCT U.drinker
FROM Frequents U
WHERE U.drinker NOT IN
  (SELECT X.drinker
   FROM Frequents X
   WHERE NOT EXISTS (SELECT *
     FROM Serves Y, Likes Z
     WHERE X.bar=Y.bar
     AND X.drinker=Z.drinker
     AND Y.beer = Z.beer))
```
Unnesting Aggregates

Find the number of companies in each city
Find the number of companies in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
    FROM Company Y
    WHERE X.city = Y.city)
FROM Company X
```
Unnesting Aggregates

Find the number of companies in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
    FROM Company Y
    WHERE X.city = Y.city)
FROM Company X
```

```
SELECT city, count(*)
FROM Company
GROUP BY city
```
Unnesting Aggregates

Find the number of companies in each city

```
SELECT DISTINCT X.city, (SELECT count(*) FROM Company Y WHERE X.city = Y.city)
FROM Company X
```

```
SELECT city, count(*)
FROM Company
GROUP BY city
```

Equivalent queries
Note: no need for DISTINCT
(DISTINCT is the same as GROUP BY)
Unnesting Aggregates

Find the number of products made in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
    FROM Product Y, Company Z
    WHERE Z.cid=Y.cid
    AND Z.city = X.city)
FROM Company X
```
Unnesting Aggregates

Find the number of products made in each city

```
SELECT DISTINCT X.city, (SELECT count(*)
    FROM Product Y, Company Z
    WHERE Z.cid=Y.cid
    AND Z.city = X.city)
FROM Company X
```

```
SELECT X.city, count(*)
FROM Company X, Product Y
WHERE X.cid=Y.cid
GROUP BY X.city
```

NOT equivalent!
You should know why!
Unnesting Aggregates

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

Purchase(pid, product, quantity, price)
### Unnesting Aggregates

**SQL Query 1:**

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

**SQL Query 2:**

```sql
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
```
Unnesting Aggregates

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?
More Unnesting

Find authors who wrote ≥ 100 documents:

Attempt 1: with nested queries

```
SELECT x.login, x.name
FROM Author x,
     (SELECT login, count(*) as c
      FROM Wrote
      GROUP BY login) y
WHERE x.login = y.login and y.c > 100
```

This is SQL by a novice.
More Unnesting

Find authors who wrote ≥ 100 documents:

Attempt 1: with nested queries

Attempt 2: using GROUP BY and HAVING

```
SELECT Author.login, Author.name
FROM Author, Wrote
WHERE Author.login=Wrote.login
GROUP BY Author.login, Author.name
HAVING count(wrote.url) >= 100
```
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…

\[
\text{SELECT } x.\text{city}, \text{max}(y.\text{price}) \\
\text{FROM} \ \text{Company} \ x, \ \text{Product} \ y \\
\text{WHERE} \ x.\text{cid} = y.\text{cid} \\
\text{GROUP BY} \ x.\text{city};
\]

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

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

WITH MaxPrices 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, MaxPrices w
WHERE u.cid = v.cid
    and u.city = w.city
    and v.price = w.maxprice;

Or using the `WITH` clause:

Product (pname, price, cid)
Company(cid, cname, city)
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);
```
There is a more concise solution here:

```sql
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);
```
Summary of SQL

• What you learn from this class:
  – Write complex SQL queries (done)
  – Tune the database, create indices
  – Define constraints

• What you don’t learn in this class
  – The rest of the SQL ecosystem
  – Learn-as-you go (manual, google)
Relational Algebra
Where We Are

• Motivation for using a DBMS for managing data
• SQL:
  – Declaring the schema for our data (CREATE TABLE)
  – Inserting data one row at a time or in bulk (INSERT/.import)
  – Modifying the schema and updating the data (ALTER/UPDATE)
  – Querying the data (SELECT)

• **Next step: More knowledge of how DBMSs work**
  – Relational algebra and query execution
  – Client-server architecture
Query Evaluation Steps

1. **Parse & Check Query**
   - SQL query
   - Translate query string into internal representation
   - Check syntax, access control, table names, etc.

2. **Decide how best to answer query: query optimization**
   - Logical plan → physical plan

3. **Query Execution**

4. **Return Results**

**Notes:**
- Logical plan → physical plan
- Query Evaluation

DATA514 - Winter 2018

38
The WHAT and the HOW

• SQL = **WHAT** we want to get form the data

• Relational Algebra = **HOW** to get the data we want

• The passage from **WHAT** to **HOW** is called query optimization
  – SQL -> Relational Algebra -> Physical Plan
  – Relational Algebra = Logical Plan
Relational Algebra
Sets v.s. Bags

So far, we have said that relational algebra and SQL operate on relations that are sets of tuples.

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

Relational Algebra has two semantics:

- **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 $\tau$
Union and Difference

\[ R_1 \cup R_2 \]
\[ R_1 - R_2 \]

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 (will explain later)
  \[ R_1 \cap R_2 = R_1 \bowtie R_2 \]
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>

$\sigma_{\text{Salary} > 40000}$ (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:
  - \( \Pi_{\text{SSN, Name}} \text{(Employee)} \)
  - \( \text{Answer(SSN, Name)} \)

Different semantics over sets or bags! Why?
<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>

\[ \Pi_{Name,Salary} (Employee) \]

<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>
</tbody>
</table>

**Set semantics**

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

Which is more efficient?
### Composing RA Operators

**Patient**

<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>

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

<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>

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

<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>

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

<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>9999999999</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
</tr>
</tbody>
</table>

### Dependent

<table>
<thead>
<tr>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>7777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>

### Employee × Dependent

<table>
<thead>
<tr>
<th>Name</th>
<th>SSN</th>
<th>EmpSSN</th>
<th>DepName</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>9999999999</td>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>John</td>
<td>9999999999</td>
<td>7777777777</td>
<td>Joe</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
<td>9999999999</td>
<td>Emily</td>
</tr>
<tr>
<td>Tony</td>
<td>7777777777</td>
<td>7777777777</td>
<td>Joe</td>
</tr>
</tbody>
</table>
Renaming

• Changes the schema, not the instance

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

• Example:
  - \( R = \rho_{N,S}(\text{Employee}) \) makes \( R \) be a relation with attributes \( N, S \) and the same tuples as \( \text{Employee} \).

Not really used by systems, but needed on paper
Natural Join

\[ R_1 \Join R_2 \]

- **Meaning:** \[ R_1 \Join R_2 = \Pi_A(\sigma_\theta (R_1 \times R_2)) \]

- **Where:**
  - Selection \( \sigma \) checks equality of **all common attributes** (attributes with same names)
  - Projection eliminates duplicate **common attributes**
Natural Join Example

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

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

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

<table>
<thead>
<tr>
<th>R \bowtie S</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
</tr>
<tr>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>X</td>
<td>Z</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>Z</td>
<td>V</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>p1</td>
<td>54</td>
<td>98125</td>
</tr>
<tr>
<td>p2</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>p1</td>
</tr>
<tr>
<td>20</td>
<td>98120</td>
<td>flu</td>
<td>p2</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

- For our voters/patients example:

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

AnonPatient (age, zip, disease)
Voters (name, age, zip)
Equijoin

• A theta join where \( \theta \) is an equality predicate

• By far the most used variant of join in practice
**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>

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

• **Equijoin:** $R \bowtie_{\theta} S = \pi_A (\sigma_{\theta}(R \times S))$
  - Join condition $\theta$ consists only of equalities

• **Natural join:** $R \Join S = \pi_A (\sigma_{\theta}(R \times S))$
  - Equijoin
  - 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>disease</th>
<th>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>33</td>
<td>98120</td>
</tr>
</tbody>
</table>
Query Evaluation Steps

1. SQL query
2. Parse & Check Query
   - Translate query string into internal representation
   - Check syntax, access control, table names, etc.
3. Decide how best to answer query: query optimization
4. Query Execution
5. Return Results

Logical plan → physical plan

Translation: Query Evaluation Steps

- SQL query
- Parse & Check Query
  - Translate query string into internal representation
  - Check syntax, access control, table names, etc.
- Decide how best to answer query: query optimization
- Query Execution
- Return Results

DATA514 - Winter 2018
From SQL to RA

\[
\begin{align*}
\text{Product} & (\text{pid, name, price}) \\
\text{Purchase} & (\text{pid, cid, store}) \\
\text{Customer} & (\text{cid, name, city})
\end{align*}
\]

\[
\begin{align*}
\text{SELECT DISTINCT} & \quad \text{x.name, z.name} \\
\text{FROM} & \quad \text{Product} \ x, \ \text{Purchase} \ y, \ \text{Customer} \ z \\
\text{WHERE} & \quad \text{x.pid} = \text{y.pid} \ \text{and} \ \text{y.cid} = \text{y.cid} \ \text{and} \\
& \quad \text{x.price} > 100 \ \text{and} \\
& \quad \text{z.city} = \text{‘Seattle’}
\end{align*}
\]
From SQL to RA

SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = z.cid and
    x.price > 100 and
    z.city = 'Seattle'

Can you think of a “better” plan?
From SQL to RA

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid and y.cid = z.cid and
    x.price > 100 and
    z.city = 'Seattle'
```

Can you think of a “better” plan?

Push selections down the query plan!
Product(pkt, name, price)
Purchase(pkt, cid, store)
Customer(cid, name, city)

**From SQL to RA**

```
SELECT DISTINCT x.name, z.name
FROM Product x, Purchase y, Customer z
WHERE x.pid = y.pid AND y.cid = z.cid AND
  x.price > 100 AND
  z.city = 'Seattle'
```

Can you think of a “better” plan?

Push selections down the query plan!

Query optimization: find an equivalent optimal plan
Extended RA: Operators on Bags

- Duplicate elimination $\delta$
- Grouping $\gamma$
- Sorting $\tau$
**Logical Query Plan**

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

T1, T2, T3 = temporary tables

sales(product, city, price)

T1(city,p,c)

T2(city,p,c)

T3(city, c)

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

\[ \sigma_{p > 100} \]

\[ \Pi_{\text{city, c}} \]
Typical Plan for Block (1/2)

SELECT fields
FROM R, S, ...
WHERE condition

SELECT-PROJECT-JOIN Query

DATA514 - Winter 2018
Typical Plan For Block (2/2)

\[ \sigma \text{ having condition} \]
\[ \gamma \text{ fields, sum/count/min/max(fields)} \]
\[ \Pi \text{ fields} \]
\[ \sigma \text{ where condition} \]

join condition

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

DATA514 - Winter 2018
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)
```
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)

Correlation!
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)

**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?

Un-nesting

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

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?

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

Each of the logical operators may have one or more implementations = physical operators

Will discuss several basic physical operators, with a focus on join
Main Memory Algorithms

Logical operator:

Product(pid, name, price) $\bowtie_{\text{pid} = \text{pid}}$ Purchase(pid, cid, store)

Propose three physical operators for the join, assuming the tables are in main memory:

1.
2.
3.
Main Memory Algorithms

Logical operator:

\[ \text{Product}(\text{pid}, \text{name}, \text{price}) \Join_{\text{pid} = \text{pid}} \text{Purchase}(\text{pid}, \text{cid}, \text{store}) \]

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join \[ \mathcal{O}(??) \]
2. Merge join \[ \mathcal{O}(??) \]
3. Hash join \[ \mathcal{O}(??) \]
Main Memory Algorithms

Logical operator:

Product(pid, name, price) \Join_{pid=pid} \approx \text{Purchase(pid, cid, store)}

Propose three physical operators for the join, assuming the tables are in main memory:

1. Nested Loop Join \quad O(n^2)
2. Merge join \quad O(n \log n)
3. Hash join \quad O(n) \ldots O(n^2)
BRIEF Review of Hash Tables

Separate chaining:

A (naïve) hash function:

\[ h(x) = x \text{ mod } 10 \]

Operations:

find(103) = ??
insert(488) = ??

Duplicates OK
WHY ??
BRIEF Review of Hash Tables

- \text{insert}(k, v) = \text{inserts a key } k \text{ with value } v

- Many values for one key
  - Hence, duplicate k’s are OK

- \text{find}(k) = \text{return the list of all values } v \text{ associated to the key } k
Query Evaluation Steps Review

1. **SQL query**
2. **Parse & Check Query**
   - Translate query string into internal representation
   - Check syntax, access control, table names, etc.
3. **Logical Query Plan**
4. **Physical Query Plan**
5. **Query Execution**
6. **Return Results**

DATA514 - Winter 2018
Relational Algebra

SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
  and y.pno = 2
  and x.scity = ‘Seattle’
  and x.sstate = ‘WA’

Give a relational algebra expression for this query
SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'

\[ \Pi_{\text{sname}} (\sigma_{\text{scity}='Seattle' \land \text{sstate}='WA' \land pno=2}(\text{Supplier} \Join_{\text{sid} = \text{sid}} \text{Supply})) \]
Relational Algebra

SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'

Relational algebra expression is also called the "logical query plan"
A physical query plan is a logical query plan annotated with physical implementation details.

SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
  and y.pno = 2
  and x.scity = 'Seattle'
  and x.sstate = 'WA'
Physical Query Plan 2

\[
\begin{align*}
\text{(On the fly)} & \quad \Pi_{\text{sname}} \\
\text{(On the fly)} & \quad \sigma_{\text{scity}=\text{Seattle} \land \text{sstate}=\text{WA} \land pno=2} \\
\text{(Hash join)} & \quad \text{sid} = \text{sid}
\end{align*}
\]

\[
\begin{align*}
\text{Supplier (File scan)} & \quad \text{Supply (File scan)}
\end{align*}
\]

SELECT sname
FROM Supplier x, Supply y
WHERE x.sid = y.sid
    and y.pno = 2
    and x.scity = 'Seattle'
    and x.sstate = 'WA'

Same logical query plan
Different physical plan
Physical Query Plan 3

Different but equivalent logical query plan; different physical plan

\[
\text{SELECT } \text{snname} \\
\text{FROM } \text{Supplier} \text{ x, Supply y} \\
\text{WHERE } x.\text{sid} = y.\text{sid} \\
\text{and } y.\text{pno} = 2 \\
\text{and } x.\text{scity} = 'Seattle' \\
\text{and } x.\text{sstate} = 'WA'
\]
Query Optimization Problem

- For each SQL query… many logical plans
- For each logical plan… many physical plans
- How do find a fast physical plan?
  - Will discuss in a few lectures