Introduction to Data Management

Relational Algebra

Alyssa Pittman
Based on slides by Jonathan Leang, Dan Suciu, et al

Paul G. Allen School of Computer Science and Engineering
University of Washington, Seattle
Announcements

- Holiday Monday, no class/OH/minimal Piazza
- HW2 due Tuesday
- HW3 will be released then
  - Will announce on Piazza
  - Accept your Azure credits! Email from invites@microsoft.com
  - TAs will walk through Azure set up next section
A subquery is a SQL query nested inside a larger query

A subquery may occur in:
- A SELECT clause
- A FROM clause
- A WHERE or HAVING clause

Rule of thumb:
Avoid nested queries when possible...
...but sometimes it’s impossible
Recap - Subqueries in SELECT

- Must return a single value
- Uses:
  - Compute an associated value

\[
\text{SELECT } P\text{.Name, (SELECT } \text{AVG(P1}\text{.Salary)} \text{ FROM Payroll AS P1}
\text{ WHERE P\text{.Job } = P1\text{.Job)} FROM Payroll AS P}
\]

**Correlated subquery!**
Semantics are that the entire subquery is recomputed for each tuple
Recap - Unnesting

For each person find the number of cars they drive

```
SELECT P.Name, (SELECT COUNT(R.Car) FROM Regist AS R WHERE P.UserID = R.UserID)
FROM Payroll AS P
```

Still possible to decorrelate and unnest

```
SELECT P.Name, COUNT(R.Car)
FROM Payroll AS P  LEFT OUTER JOIN Regist AS R ON P.UserID = R.UserID
GROUP BY P.Name
```
Recap – Subqueries in FROM

- **Uses:**
  - Solve subproblems that can be later joined/evaluated

```sql
SELECT P.Name, P.Salary
FROM Payroll AS P,
(SELECT P1.Job AS Job,
  MAX(P1.Salary) AS Salary
FROM Payroll AS P1
GROUP BY P1.Job) AS Pmax
WHERE P.Job = Pmax.Job AND
  P.Salary = Pmax.Salary
```
Recap – Subqueries in WHERE/HAVING

- Can return a relation
- Uses:
  - Use with an existential or universal quantifier
  - \((\text{NOT}) \text{ EXISTS}, (\text{NOT}) \text{ IN}, \text{ANY, ALL}\)

Ex: Find all people who drive some car made before 2017.

```sql
SELECT P.Name
FROM Payroll AS P
WHERE EXISTS (SELECT *
               FROM Regist R
               WHERE R.UserID = P.UserID
               AND R.Year < 2017)
```
Recap - Monotonicity

**Monotone**

A **Monotonic** query is one that obeys the following rule where $I$ and $J$ are data instances and $q$ is a query:

$$I \subseteq J \rightarrow q(I) \subseteq q(J)$$

That is for any superset of $I$, the query over that superset must contain at least the query results of $I$. 
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

Monotone queries can be similar to monotonically increasing functions when considering cardinalities of results.
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

```
SELECT P.Name, P.Car
FROM Payroll AS P, Regist AS R
WHERE P.UserID = R.UserID
```

Is this query monotone?
A Monotonic query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

**SELECT**  P.Name, P.Car  
**FROM**  Payroll AS P, Regist AS R  
**WHERE**  P.UserID = R.UserID  

Is this query monotone? Yes!
A **Monotonic** query is one that obeys the following rule where $I$ and $J$ are data instances and $q$ is a query:

$$I \subseteq J \rightarrow q(I) \subseteq q(J)$$

That is, for any superset of $I$, the query over that superset must contain at least the query results of $I$.

**SELECT**  
\[ \text{P.Name, P.Car} \]

**FROM**  
\[ \text{Payroll AS P, Regist AS R} \]

**WHERE**  
\[ \text{P.UserID = R.UserID} \]

Is this query monotone? **Yes!**
Monotonicity

Monotone

A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

$$I \subseteq J \rightarrow q(I) \subseteq q(J)$$

That is for any superset of I, the query over that superset must contain at least the query results of I.

```sql
SELECT P.Name
FROM Payroll AS P
WHERE P.Salary >= ALL (SELECT Salary FROM Payroll)
```

Is this query monotone?
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

**SELECT** P.Name  
**FROM** Payroll AS P  
**WHERE** P.Salary >= ALL (**SELECT** Salary  
**FROM** Payroll)

Is this query monotone? **No!**
Monotonicity

Monotone

A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

```sql
SELECT P.Name
FROM Payroll AS P
WHERE P.Salary >= ALL (SELECT Salary
                        FROM Payroll)
```

Is this query monotone? No!
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

$$I \subseteq J \rightarrow q(I) \subseteq q(J)$$

That is for any superset of I, the query over that superset must contain at least the query results of I.

**SELECT**  
**P.Job, COUNT(*)**  
**FROM**  
Payroll AS P  
**GROUP BY**  
P.Job

Is this query monotone?
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is for any superset of I, the query over that superset must contain at least the query results of I.

```sql
SELECT P.Job, COUNT(*)
FROM Payroll AS P
GROUP BY P.Job
```

Is this query monotone? **No!**
A **Monotonic** query is one that obeys the following rule where I and J are data instances and q is a query:

\[ I \subseteq J \rightarrow q(I) \subseteq q(J) \]

That is, for any superset of I, the query over that superset must contain at least the query results of I.

Aggregates generally are sensitive to any new tuples since the aggregate value will change.

**Monotone**

```sql
SELECT P.Job, COUNT(*)
FROM Payroll AS P
GROUP BY P.Job
```

Is this query monotone? **No!**
Theorem:
If Q is a SELECT-FROM-WHERE query that does not have subqueries or aggregates, then it is monotone.
Monotonicity

Theorem:
If Q is a SELECT–FROM–WHERE query that does not have subqueries or aggregates, then it is monotone.

Proof:
We use nested loop semantics. If we insert a tuple in relation R, this will not remove any tuples from the answer.

```
SELECT a1, a2, ..., ak
FROM R1 AS x1, R2 AS x2, ..., Rn AS xn
WHERE Conditions
```

for x1 in R1 do
  for x2 in R2 do
    ...
    for xn in Rn do
      if Conditions
        output (a1,...,ak)
Theorem:
The query “Find all people who drive only cars older than 2017” is not monotone.

Proof:
We use example. For user 123 who previously only drove a car made in 2009, we add another car made in 2018. Now user 123 does not appear in the results. Thus, the query is not monotone.
Monotonicity

Theorem:
The query “Find all people who drive only cars older than 2017” is not monotone.

Proof:
We use example. For user 123 who previously only drove a car made in 2009, we add another car made in 2018. Now user 123 does not appear in the results. Thus, the query is not monotone.

If a query is not monotonic, then we can’t write it as a SELECT-FROM-WHERE query without subqueries.
Queries That Cannot Be S-F-W

- Queries with universal quantifiers or negation

SELECT P.Name
FROM Payroll AS P
WHERE P.UserID NOT IN (SELECT R.UserID
FROM Regist AS R
WHERE R.Year < 2017)

SELECT P.Name
FROM Payroll AS P
WHERE P.Salary >= ALL (SELECT Salary
FROM Payroll)
Goals for Today

- We’ve completed SQL! Now we know how to write a query for any question the relational model can answer.
- Next we’ll dive into another language for working with relational data – Relational Algebra.
Outline

- What is Relational Algebra (RA)?
- Introduce RA operators
- See equivalent SQL and RA queries
What’s the Point of RA?

- SQL is a **Declarative Language**
  - “What to get” rather than “how to get it”
  - Easier to write a SQL query than write a whole Java program that will probably perform worse

- But computers are imperative/procedural
  - Computers only understand the “how”
History of RA

Formalized and published by Ted Codd of IBM

Initially IBM didn’t use his approach... 10 years later he won the Turing Award

Information Retrieval

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.
History of RA

Formalized and published by Ted Codd of IBM

Initially IBM didn’t use his approach... 10 years later he won the Turing Award

Information Retrieval

A Relational Model of Data for Large Shared Data Banks

E. F. Codd
IBM Research Laboratory, San Jose, California

Physical data independence!

Future users of large data banks must be protected from having to know how the data is organized in the machine (the internal representation). A prompting service which supplies such information is not a satisfactory solution. Activities of users at terminals and most application programs should remain unaffected when the internal representation of data is changed and even when some aspects of the external representation are changed. Changes in data representation will often be needed as a result of changes in query, update, and report traffic and natural growth in the types of stored information.
What’s the Point of RA?

- We need a language that reads **more like instructions** but still captures the fundamental operations of a query.

![Diagram showing SQL to RDBMS with binary code conversion](#)
Relational Algebra (RA) does the job

- When processing your query, the **RDBMS will actually store an RA tree** (like a bunch of labeled nodes and pointers)
- After some optimizations, the **RA tree is converted into instructions** (like a bunch of functions linked together)
RA Operators

- Read RA tree from bottom to top
  - Bottom → Data sources
  - Top → Query output

- Semantics
  - Every operator takes 1 or 2 relations as inputs
  - Every operator outputs a relation as an output
RA Operators

- These are all the operators you will see in this class
  - We’ll profile these one at a time

- Join
- Cartesian Product
- Selection
- Projection
- Union
- Intersection
- Difference
- Grouping & Aggregation
- Sort
- Duplicate Elimination
RA Operators

- These are all the operators you will see in this class
  - We’ll profile these one at a time

- ![Join](Join_icon.png)
- ![Cartesian Product](CartesianProduct_icon.png)
- ![Selection](Selection_icon.png)
- ![Projection](Projection_icon.png)
- ![Union](Union_icon.png)
- ![Intersection](Intersection_icon.png)
- ![Difference](Difference_icon.png)
- ![Grouping & Aggregation](GroupingAggregation_icon.png)
- ![Sort](Sort_icon.png)
- ![Duplicate Elimination](DuplicateElimination_icon.png)

RA

Extended RA
RA Operators

- For the curious...

- Right Outer Join
- Left Outer Join
- Full Outer Join
- Rename
How does a computer understand abstract SQL text?
- Code has to boil down to instructions at some point
- Relational Database Management Systems (RDBMSs) use **Relational Algebra (RA)**

```sql
SELECT P.Name, P.UserID
FROM Payroll AS P
WHERE P.Job = 'TA';
```
• Code has to boil down to instructions at some point
• Relational Database Management Systems (RDBMSs) use Relational Algebra (RA)

Parser

\[ \pi_{P.Name, P.UserID} \]

\[ \sigma_{P.Job='TA'} \]

\[ \text{Payroll P} \]

\[ \text{SELECT } P.Name, P.UserID \text{ FROM Payroll AS P WHERE P.Job = 'TA';} \]
- Code has to boil down to instructions at some point
- Relational Database Management Systems (RDBMSs) use **Relational Algebra** (RA).

\[
\pi_{P.Name, P.UserID} \\
\sigma_{P.Job='TA'} \\
Payroll P
\]
Database Internals

- Code has to boil down to instructions at some point
- Relational Database Management Systems (RDBMSs) use **Relational Algebra** (RA).

```
\[ \pi_{P.\text{Name},P.\text{UserID}} \]
\[ \sigma_{P.\text{Job}='TA'} \]
```

\( \text{Payroll } P \)

For-each semantics

for each row in P:

if (row.Job == 'TA'):
    output (row.Name, row.UserID)
- Code has to boil down to instructions at some point
- Relational Database Management Systems (RDBMSs) use **Relational Algebra** (RA).

\[
\pi_{P.Name, P.UserID} \\
\sigma_{P.Job='TA'}
\]

Tuples “flow” up the RA tree getting filtered and modified
Get ready for some examples...
### RA Operators

**Projection**

- **Unary operator**
- **Projection removes unspecified columns**
- **Happens in the SQL “SELECT” clause**

\[
\pi_{A,B}(T(A, B, C)) \rightarrow S(A, B)
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
**Unary operator**

- **Selection** returns tuples from the input which satisfy the condition (filtering)
- Happens in the SQL “WHERE” or “HAVING” clauses

\[ \sigma_{T.A < 6}(T(A, B, C)) \rightarrow S(A, B, C) \]
RA Operators

\[ \sigma \]

Selection

- Unary operator
- Selection returns tuples from the input which satisfy the condition (filtering)
- Happens in the SQL “WHERE” or “HAVING” clauses

\[ \sigma_{T.A < 6} \left( T(A, B, C) \right) \rightarrow S(A, B, C) \]

Can use =, <, <=, >, >=, <>
Combine with AND, OR, NOT

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
RA Operators

Join

- Binary operator
- Joins inputs relations on the specified condition
- Happens in the SQL “JOIN” clause (or implicit joins using WHERE)

\[ T(A, B) \Join_{T.B=S.C} S(C, D) \rightarrow R(A, B, C, D) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
</tr>
<tr>
<td>3</td>
<td>4</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>5</td>
<td>6</td>
<td>6</td>
<td>7</td>
</tr>
</tbody>
</table>
RA Operators

Cartesian Product

- Binary operator
- Same semantics as in set theory
- Indiscriminate join of input relations

\[ T(A, B) \times S(C, D) \rightarrow R(A, B, C, D) \]
RA Operators

- **Cartesian Product**
  - Binary operator
  - Same semantics as in set theory
  - Indiscriminate join of input relations

\[
T(A, B) \times R(A, B, C, D)
\]

- Rare in practice - this is mainly used to express joins. Think our nested loop semantics.
So far we haven’t discussed equivalent RA trees. But all joins can be parsed directly into a “join tree”
RA Equivalencies

**SELECT**  P.Name, R.Car  
**FROM**  Payroll AS P, Regist AS R  
**WHERE**  P.UserID = R.UserID;  

\[ \Pi_{P.Name, R.Car} ( \bowtie_{P.UserID = R.UserID} (\text{Payroll } P \times \text{Regist } R )) \]
RA Equivalencies

\[ \text{SELECT} \quad P.\text{Name, R.Car} \]
\[ \text{FROM} \quad \text{Payroll AS P, Regist AS R} \]
\[ \text{WHERE} \quad \text{P.UserID} = \text{R.UserID}; \]

\[ \Pi_{P.\text{Name, R.Car}}(\bowtie_{\text{P.UserID} = \text{R.UserID}}(\text{Payroll P, Regist R})) \]

\[ \Pi_{P.\text{Name, R.Car}} \]
\[ \bowtie_{\text{P.UserID} = \text{R.UserID}} \]
\[ \text{Payroll P, Regist R} \]
RA Equivalencies

\[
\begin{align*}
\text{SELECT} & \quad P.\text{Name}, R.\text{Car} \\
\text{FROM} & \quad \text{Payroll AS P, Regist AS R} \\
\text{WHERE} & \quad P.\text{UserID} = R.\text{UserID};
\end{align*}
\]
RA Equivalencies

\[ \text{SELECT} \ P.\text{Name}, \ R.\text{Car} \]
\[ \text{FROM} \ \text{Payroll} \ AS \ P, \ \text{Regist} \ AS \ R \]
\[ \text{WHERE} \ P.\text{UserID} = R.\text{UserID}; \]

Join
Combine tuples on the provided predicate
SELECT P.Name, R.Car
FROM Payroll AS P, Regist AS R
WHERE P.UserID = R.UserID;

\[
\pi_{P\text{.Name}, R\text{.Car}} \\
\sigma_{P\text{.UserID} = R\text{.UserID}}
\]

\[
\times
\]

Payroll P  Regist R
SELECT P.Name, R.Car
FROM Payroll AS P, Regist AS R
WHERE P.UserID = R.UserID;

Cross Product
Same intuition from set theory
CREATE TABLE Person (  
    pid INT PRIMARY KEY, -- person ID  
    name VARCHAR(100));  -- person name

CREATE TABLE Email (  
    eid INT PRIMARY KEY, -- email ID  
    pidFrom INT REFERENCES Person, -- email sender  
    length INT);  -- email char length

CREATE TABLE EmailTo (  
    eid INT REFERENCES Email, -- email ID  
    pidTo INT REFERENCES Person, -- email recipient  
    PRIMARY KEY (eid, pidTo));
Your Turn!

- A warm up
- Find the length of all emails Alice sent.

```sql
SELECT E.length
FROM Person P, Email E
WHERE P.pid = E.pidFrom AND
  P.name = 'Alice';
```
Your Turn!

- A warm up
- Find the length of all emails Alice sent.

```
SELECT E.length
FROM Person P, Email E
WHERE P.pid = E.pidFrom AND
P.name = 'Alice';
```
Your Turn!

- A warm up
- Find the length of all emails Alice sent.

```
SELECT E.length
FROM Person P, Email E
WHERE P.pid = E.pidFrom AND
  P.name = 'Alice';
```
A warm up

Find the length of all emails Alice sent.

```
SELECT E.length
FROM Person P, Email E
WHERE P.pid = E.pidFrom AND
  P.name = 'Alice';
```
Onto extended RA

Original relational algebra only worked with sets

We clearly need operators working with real-life relations: bags, ordering, grouping...
RA Operators

- Unary operator
- Specifies grouped attributes and then aggregates
- ONLY operation that can compute aggregates

\[ \gamma_{T.A, \text{max}(T.B) \rightarrow mB} \left( T(A, B, C) \right) \rightarrow R(A, mB) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>mB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
RA Operators

\[ \gamma \text{ Grouping & Aggregation} \]

- Unary operator
- Specifies grouped attributes and then aggregates
- ONLY operation that can compute aggregates

\[ \gamma_{T.A, \max(T.B) \rightarrow mB}(T(A, B, C)) \rightarrow R(A, mB) \]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>mB</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>5</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
</tr>
</tbody>
</table>
**RA Operators**

\( \tau \)  Sort

- Unary operator
- Orders the input by any of the columns
- Happens in SQL “ORDER BY” clause
- Assume default ascending order like in SQL

\[ \tau_{T.A,T.B}(T(A,B,C)) \rightarrow R(A,B,C) \]

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
</tbody>
</table>

<p>| | | |</p>
<table>
<thead>
<tr>
<th></th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>A</td>
<td>B</td>
<td>C</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>5</td>
<td>6</td>
</tr>
<tr>
<td>7</td>
<td>8</td>
<td>9</td>
</tr>
</tbody>
</table>
RA Operators

**Duplicate Elimination**

- Unary operator
- Deduplicates tuples
- Happens with SQL “DISTINCT” keyword
- Technically useless because it’s the same as grouping on all attributes

\[
\delta(T(A, B, C)) \rightarrow R(A, B, C)
\]

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>2</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>5</td>
<td>6</td>
</tr>
</tbody>
</table>
SQL and RA Vocab Summary

SELECT ...
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...

σ \times \times \cdots

<table>
<thead>
<tr>
<th>Tables</th>
</tr>
</thead>
</table>

δ
Π
τ
σ
γ
SELECT ... 
FROM ... 
WHERE ... 
GROUP BY ... 
HAVING ... 
ORDER BY ...

\[ \sigma \bowtie \times \cdots \]

Tables

Selection
Join
Cartesian
Product

\( \delta \)
\( \Pi \)
\( \tau \)
\( \sigma \)
\( \gamma \)
SQL and RA Vocab Summary

\[
\delta \\
\Pi \\
\sigma \\
\gamma \\
\sigma \bowtie \times \ldots
\]

SELECT ... 
FROM ...
WHERE ...
GROUP BY ...
HAVING ...
ORDER BY ...

Aggregation
SQL and RA Vocab Summary

SELECT ...  
FROM ...  
WHERE ...  
GROUP BY ...  
HAVING ...  
ORDER BY ...

Selection

Tables

$\delta$

$\Pi$

$\tau$

$\sigma$

$\gamma$

$\sigma \Join \times \cdots$
SQL and RA Vocab Summary

SELECT ... 
FROM ... 
WHERE ... 
GROUP BY ... 
HAVING ... 
ORDER BY ...

Tables

\( \sigma \Join \times \cdots \)

\( \gamma \)

\( \sigma \)

\( \tau \)

\( \Pi \)

\( \delta \)

Sorting
SQL and RA Vocab Summary

SELECT ... 
FROM ... 
WHERE ... 
GROUP BY ... 
HAVING ... 
ORDER BY ...

Projection Deduplication
FWGHOS™

- SELECT ...
- FROM ...
- WHERE ...
- GROUP BY ...
- HAVING ...
- ORDER BY ...

\[\sigma \Join \times \times \cdots\]

Tables
Basic SQL to RA Conversion

- The general plan structure for a “flat” SQL query

- \( \pi \)
- \( \tau \)
- \( \sigma \)
- \( \gamma \)

- SELECT
- ORDER BY
- HAVING
- GROUP BY & aggregates
- FROM & WHERE

Table
How is aggregation processed internally?

```sql
SELECT Job, MAX(Salary)
FROM Payroll
GROUP BY Job
HAVING MIN(Salary) > 80000
```
```sql
SELECT Job, MAX(Salary)
FROM Payroll
GROUP BY Job
HAVING MIN(Salary) > 80000
```

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
SELECT Job, MAX(Salary) 
FROM Payroll 
GROUP BY Job 
HAVING MIN(Salary) > 80000
Aggregation RA

SELECT Job, MAX(Salary)
FROM Payroll
GROUP BY Job
HAVING MIN(Salary) > 80000

<table>
<thead>
<tr>
<th>Job</th>
<th>maxSal</th>
<th>minSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA</td>
<td>60000</td>
<td>50000</td>
</tr>
<tr>
<td>Prof</td>
<td>100000</td>
<td>90000</td>
</tr>
</tbody>
</table>

\[ \forall \text{Job}, \text{MAX(P.Salary)} \rightarrow \text{maxSal}, \text{MIN(P.Salary)} \rightarrow \text{minSal} \]

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Aggregation RA

\[
\begin{align*}
\text{SELECT} & \quad \text{Job, MAX(Salary)} \\
\text{FROM} & \quad \text{Payroll} \\
\text{GROUP BY} & \quad \text{Job} \\
\text{HAVING} & \quad \text{MIN(Salary)} > 80000
\end{align*}
\]

\[\sigma_{\text{minSal} > 80000}\]

\[\forall \text{Job}, \text{MAX(P.Salary)} \rightarrow \text{maxSal}, \text{MIN(P.Salary)} \rightarrow \text{minSal}\]

<table>
<thead>
<tr>
<th>Job</th>
<th>maxSal</th>
<th>minSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA</td>
<td>60000</td>
<td>50000</td>
</tr>
<tr>
<td>Prof</td>
<td>100000</td>
<td>90000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
```
SELECT Job, MAX(Salary)
FROM Payroll
GROUP BY Job
HAVING MIN(Salary) > 80000
```

<table>
<thead>
<tr>
<th>Job</th>
<th>maxSal</th>
<th>minSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prof</td>
<td>100000</td>
<td>90000</td>
</tr>
</tbody>
</table>

\[\sigma_{\text{minSal} > 80000}\]

<table>
<thead>
<tr>
<th>Job</th>
<th>maxSal</th>
<th>minSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA</td>
<td>60000</td>
<td>50000</td>
</tr>
<tr>
<td>Prof</td>
<td>100000</td>
<td>90000</td>
</tr>
</tbody>
</table>

\[\forall \text{Job}, \: \text{MAX(P.Salary)} \rightarrow \text{maxSal}, \: \text{MIN(P.Salary)} \rightarrow \text{minSal}\]
Aggregation RA

\[
\Pi_{\text{Job}, \text{maxSal}}
\]

<table>
<thead>
<tr>
<th>Job</th>
<th>maxSal</th>
<th>minSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prof</td>
<td>100000</td>
<td>90000</td>
</tr>
</tbody>
</table>

\[
\sigma_{\text{minSal}>80000}
\]

<table>
<thead>
<tr>
<th>Job</th>
<th>maxSal</th>
<th>minSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA</td>
<td>60000</td>
<td>50000</td>
</tr>
<tr>
<td>Prof</td>
<td>100000</td>
<td>90000</td>
</tr>
</tbody>
</table>

\[
\forall_{\text{Job}, \text{MAX(P.Salary)}} \rightarrow \text{maxSal}, \text{MIN(P.Salary)} \rightarrow \text{minSal}
\]

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Aggregation RA

\[
\text{SELECT } \text{Job, MAX(Salary)} \\
\text{FROM Payroll} \\
\text{GROUP BY Job} \\
\text{HAVING MIN(Salary) > 80000}
\]

\[
\Pi_{\text{Job}, \text{maxSal}}
\]

<table>
<thead>
<tr>
<th>Job</th>
<th>maxSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>Prof</td>
<td>100000</td>
</tr>
</tbody>
</table>

\[
\sigma_{\text{minSal}>80000}
\]

<table>
<thead>
<tr>
<th>Job</th>
<th>maxSal</th>
<th>minSal</th>
</tr>
</thead>
<tbody>
<tr>
<td>TA</td>
<td>60000</td>
<td>50000</td>
</tr>
<tr>
<td>Prof</td>
<td>100000</td>
<td>90000</td>
</tr>
</tbody>
</table>

\[
\forall_{\text{Job}, \text{MAX(P.Salary)\rightarrow maxSal, MIN(P.Salary)\rightarrow minSal}}
\]

<table>
<thead>
<tr>
<th>UserID</th>
<th>Name</th>
<th>Job</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
Aggregation RA

```
SELECT Job, MAX(Salary)
FROM Payroll
GROUP BY Job
HAVING MIN(Salary) > 80000
```
Aggregation RA

```
SELECT Job, MAX(Salary)
FROM Payroll
GROUP BY Job
HAVING MIN(Salary) > 80000
```

\[ \pi_{\text{Job}, \text{maxSal}} \]

Selection

```
\sigma_{\text{minSal} > 80000}
```

\[ \nu_{\text{Job}, \text{MAX(P.Salary)} \rightarrow \text{maxSal}, \text{MIN(P.Salary)} \rightarrow \text{minSal}} \]

Payroll P
Your Turn!

- An extended problem
- Find the number of emails that each person has received.

```sql
SELECT P.name, COUNT(*)
    FROM Person P, EmailTo T
WHERE P.pid = T.pidTo
GROUP BY P.pid, P.name;
```
An extended problem

Find the number of emails that each person has received.

```sql
SELECT P.name, COUNT(*)
FROM Person P, EmailTo T
WHERE P.pid = T.pidTo
GROUP BY P.pid, P.name;
```
Your Turn!

- An extended problem
- Find the number of emails that each person has received.

```sql
SELECT P.name, COUNT(*)
FROM Person P, EmailTo T
WHERE P.pid = T.pidTo
GROUP BY P.pid, P.name;
```
Your Turn!

- An extended problem
- Find the number of emails that each person has received.

```
SELECT P.name, COUNT(*)
FROM Person P, EmailTo T
WHERE P.pid = T.pidTo
GROUP BY P.pid, P.name;
```
Relational Algebra has operators that can express everything we can express in SQL
We can convert SQL to equivalent RA trees