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

- HW2 due tonight
- HW3 released
  - Accept your Azure credits! Email from invites@microsoft.com
  - TAs will walk through Azure set up in section
Recap - Nested Queries

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

SELECT P.Name, (SELECT AVG(P1.Salary) FROM Payroll AS P1 WHERE P.Job = P1.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

```sql
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

```sql
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
  - (NOT) EXISTS, (NOT) IN, 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)
```
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.
Bonuses: Set Operations

- SQL mimics set theory in many ways
  - Bag = duplicates allowed
  - **UNION (ALL)** ▷ set union (bag union)
  - **INTERSECT (ALL)** ▷ set intersection (bag intersection)
  - **EXCEPT (ALL)** ▷ set difference (bag difference)

- SQL Server Management Studio 2017
  - INTERSECT ALL not supported
  - EXCEPT ALL not supported
Set Operations

- SQL set-like operators basically slap two queries together (not really a subquery...)

\[
\text{(SELECT * FROM T1)} \quad \text{UNION} \quad \text{(SELECT * FROM T2)}
\]

IS THIS A SUBQUERY?
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

Information Retrieval

A Relational Model of Data for Large Shared Data Banks

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

Physical data independence!

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

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.
What’s the Point of RA?

- 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
These are all the operators you will see in this class

- We’ll profile these one at a time

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

- Extended RA
RA Operators

- For the curious...

- Right Outer Join
- Left Outer Join
- Full Outer Join
- Rename

October 9, 2019
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';
```
Database Internals

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

```
SELECT P.Name, P.UserID
FROM Payroll AS P
WHERE P.Job = 'TA';
```

Parser

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

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

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

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

For-each semantics
- 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)
Database Internals

- 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
RA Operators

- Get ready for some examples...
RA Operators

\[ \pi \]  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>
RA Operators

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

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

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)

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

<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

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

Cartesian Product

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

Rare in practice - this is mainly used to express joins. Think our nested loop semantics.
RA Equivalencies

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;
```

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

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

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

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

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

**Join**
Combine tuples on the provided predicate

\[
\Pi_{P.Name, R.Car}
\bowtie_{P.UserID = R.UserID}
\]

*Payroll P  Regist R*
RA Equivalencies

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

\[
\Pi_{P.Name, R.Car} \sigma_{P.UserID = R.UserID} (P \times R)
\]
RA Equivalencies

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

Cross Product
Same intuition from set theory
Select $\sigma$, Project $\pi$, Join $\bowtie$ are the most common operators.

We also have set operators.
RA Operators

- Binary operators
- Same semantics as in set theory (but over bags)
- Used in SQL “UNION” and “INTERSECTION”
  - Also useful when rewriting SQL to RA

\[ T(A, B) \cup S(A, B) \rightarrow R(A, B) \]
RA Operators

**Difference**

- Binary operator (but direction matters)
- Reads as \((\text{left input}) \, \ominus \, (\text{right input})\)
- Used in SQL “DIFFERENCE”
  - Also useful when rewriting SQL to RA

\[
T(A, B) \, \ominus \, S(A, B) \rightarrow R(A, B)
\]

<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>
</tbody>
</table>

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>3</td>
<td>4</td>
</tr>
</tbody>
</table>
RA Operators

### Difference

- Binary operator (but direction matters)
- Reads as \((\text{left input}) - (\text{right input})\)
- Used in SQL "DIFFERENCE"
  - Also useful when rewriting SQL to RA

\[
T(A, B) - S(A, B) \rightarrow R(A, B)
\]

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
</tr>
</thead>
<tbody>
<tr>
<td>?</td>
<td>?</td>
</tr>
</tbody>
</table>
RA Operators

- **Difference**
  - Binary operator (but direction matters)
  - Reads as (left input) – (right input)
  - Used in SQL “DIFFERENCE”
  - Also useful when rewriting SQL to RA

\[ T(A, B) \setminus S(A, B) \rightarrow R(A, B) \]
Onto extended RA

Original relational algebra only worked with sets

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

\[ \gamma \] Grouping & Aggregation

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

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

<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>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

Grouping & Aggregation

- 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>

attribute to group by
aggregate to compute
only these attributes will be passed “up” the tree
RA Operators

τ  Sort

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

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

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
</tr>
</thead>
<tbody>
<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>

<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>
RA Operators

\[ \delta \text{ Duplicate Elimination} \]

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

$\delta$
$\Pi$
$\tau$
$\sigma$
$\gamma$

$\sigma \Join \times \cdots$

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

Selection Join Cartesian Product

σ △ □ ⋯ Tables
SQL and RA Vocab Summary

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

Tables

Aggregation

$\sigma \times \times \times \cdots$

Aggregation
SQL and RA Vocab Summary

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

\[ \sigma \forall \times \cdots \]

Tables

Selection

\[ \delta \]

\[ \Pi \]

\[ \tau \]

\[ \sigma \]

\[ \gamma \]
SQL and RA Vocab Summary

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

δ
Π
τ
σ
γ
σ × × × ...

Tables

Sorting

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

Tables

δ
Π
τ
σ
γ

Projection Deduplication
FWGHOS™

SELECT ...  
FROM  ... 
WHERE ... 
GROUP BY ... 
HAVING ... 
ORDER BY ...
Basic SQL to RA Conversion

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

\[ \pi \]
\[ \tau \]
\[ \sigma \]
\[ \gamma \]

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

Tables

- SELECT
- ORDER BY
- HAVING
- GROUP BY & aggregates
- FROM & WHERE
How is aggregation processed internally?

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

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

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

<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

```sql
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>

\[ \forall Job, \max(P \cdot Salary) \rightarrow \text{maxSal}, \min(P \cdot Salary) \rightarrow \text{minSal} \]

<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>
<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
```

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

<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

```
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>
<tr>
<td>TA</td>
<td>60000</td>
<td>50000</td>
</tr>
<tr>
<td>Prof</td>
<td>100000</td>
<td>90000</td>
</tr>
</tbody>
</table>

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

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

<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

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

<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

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

\[ \prod_{Job, maxSal} \]

\[ \sigma_{minSal>80000} \]

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

<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>
<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
Aggregation RA

\[ \Pi_{\text{Job}, \text{maxSal}} \sigma_{\text{minSal} > 80000} \]

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

Payroll \( P \)

**Selection**

HAVING uses the same symbol and operation used by WHERE clause.
Takeaways

▪ Relational Algebra has operators that can express everything we can express in SQL
▪ We can convert SQL to equivalent RA trees