Today’s Lecture

1. The Relational Model & Relational Algebra

2. Relational Algebra Pt. II
1. The Relational Model & Relational Algebra

What you will learn about in this section

1. The Relational Model
2. Relational Algebra: Basic Operators
3. Execution
Motivation

The Relational model is **precise**, **implementable**, and we can operate on it (query/update, etc.)

Database maps internally into this *procedural language*.

A Little History

- Relational model due to Edgar “Ted” Codd, a mathematician at IBM in 1970

- IBM didn’t want to use relational model
  - *Apparently used in the moon landing*...
  - *Google for “IMS and the Apollo program”*
The Relational Model: Schemata

- Relational Schema:

  Students(sid: string, name: string, gpa: float)

  *Relation name*

  *Attributes*

  String, float, int, etc. are the **domains** of the attributes

The Relational Model: Data

- An **attribute** (or **column**) is a typed data entry present in each tuple in the relation

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Bob</td>
<td>3.2</td>
</tr>
<tr>
<td>002</td>
<td>Joe</td>
<td>2.8</td>
</tr>
<tr>
<td>003</td>
<td>Mary</td>
<td>3.8</td>
</tr>
<tr>
<td>004</td>
<td>Alice</td>
<td>3.5</td>
</tr>
</tbody>
</table>

*The number of attributes is the **arity** of the relation*
The Relational Model: Data

**Student**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>Bob</td>
<td>3.2</td>
</tr>
<tr>
<td>002</td>
<td>Joe</td>
<td>2.8</td>
</tr>
<tr>
<td>003</td>
<td>Mary</td>
<td>3.8</td>
</tr>
<tr>
<td>004</td>
<td>Alice</td>
<td>3.5</td>
</tr>
</tbody>
</table>

A **tuple** or **row** (or **record**) is a single entry in the table having the attributes specified by the schema.

The number of tuples is the **cardinality** of the relation.

**Recall:** In practice, DBMSs relax the set requirement, and use multisets.

A **relational instance** is a set of tuples all conforming to the same **schema**.
To Reiterate

- A relational schema describes the data that is contained in a relational instance

Let $R(f_1: \text{Dom}_1, \ldots, f_m: \text{Dom}_m)$ be a relational schema then, an instance of $R$ is a subset of $\text{Dom}_1 \times \text{Dom}_2 \times \ldots \times \text{Dom}_n$

In this way, a relational schema $R$ is a total function from attribute names to types

One More Time

- A relational schema describes the data that is contained in a relational instance

A relation $R$ of arity $t$ is a function: $R : \text{Dom}_1 \times \ldots \times \text{Dom}_t \rightarrow \{0,1\}$ i.e. returns whether or not a tuple of matching types is a member of it

Then, the schema is simply the signature of the function

Note here that order matters, attribute name doesn’t... We’ll (mostly) work with the other model (last slide) in which attribute name matters, order doesn’t!
A relational database

• A *relational database schema* is a set of relational schemata, one for each relation

• A *relational database instance* is a set of relational instances, one for each relation

Two conventions:
1. We call relational database instances as simply *databases*
2. We assume all instances are valid, i.e., satisfy the *domain constraints*

Remember the CMS

• *Relation DB Schema*
  - Students(sid: string, name: string, gpa: float)
  - Courses(cid: string, cname: string, credits: int)
  - Enrolled(sid: string, cid: string, grade: string)

<table>
<thead>
<tr>
<th>Sid</th>
<th>Name</th>
<th>Gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>101</td>
<td>Bob</td>
<td>3.2</td>
</tr>
<tr>
<td>123</td>
<td>Mary</td>
<td>3.8</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cid</th>
<th>cname</th>
<th>credits</th>
</tr>
</thead>
<tbody>
<tr>
<td>564</td>
<td>564-2</td>
<td>4</td>
</tr>
<tr>
<td>308</td>
<td>417</td>
<td>2</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>564</td>
<td>A</td>
</tr>
</tbody>
</table>

Note that the schemas impose effective domain/type constraints, i.e. Gpa can’t be “Apple”
2\textsuperscript{nd} Part of the Model: Querying

```
SELECT S.name
FROM Students S
WHERE S.gpa > 3.5;
```

“We don’t tell the system how or where to get the data—just what we want, i.e., Querying is declarative.”

To make this happen, we need to translate the declarative query into a series of operators... we’ll see this next!

Actually, I showed how to do this translation for a much richer language!

Virtues of the model

- Physical independence (logical too), Declarative
- Simple, elegant, clean: Everything is a relation
- Why did it take multiple years?
  - Doubted it could be done efficiently.
RDBMS Architecture

How does an SQL engine work?

1. **SQL Query**
2. **Relational Algebra (RA) Plan**
3. **Optimized RA Plan**
4. **Execution**

- Declarative query (from user)
- Translate to relational algebra expression
- Find logically equivalent but more efficient RA expression
- Execute each operator of the optimized plan!

Relational Algebra allows us to translate declarative (SQL) queries into precise and optimizable expressions!
Relational Algebra (RA)

• **Five basic operators:**
  - Selection: $\sigma$
  - Projection: $\Pi$
  - Cartesian Product: $\times$
  - Union: $\cup$
  - Difference: $-$

• **Derived or auxiliary operators:**
  - Intersection, complement
  - Joins (natural, equi-join, theta join, semi-join)
  - Renaming: $\rho$
  - Division

Keep in mind: RA operates on sets!

• RDBMSs use *multisets*, however in relational algebra formalism we will consider *sets!*

• Also: we will consider the *named perspective*, where every attribute must have a unique name
  - $\rightarrow$ attribute order does not matter...

Now on to the basic RA operators...
Selection ($\sigma$)

- Returns all tuples which satisfy a condition
- Notation: $\sigma_c(R)$
- Examples
  - $\sigma_{\text{Salary} > 40000}$ (Employee)
  - $\sigma_{\text{name} = \text{"Smith"}}$ (Employee)
- The condition $c$ can be
  - $=, \leq, >, \geq, <$

SQL:

```
SELECT * FROM Students WHERE gpa > 3.5;
```

RA:

$$\sigma_{gpa > 3.5}(\text{Students})$$

Another example:

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>John</td>
<td>200000</td>
</tr>
<tr>
<td>5423341</td>
<td>Smith</td>
<td>600000</td>
</tr>
<tr>
<td>4352342</td>
<td>Fred</td>
<td>500000</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>600000</td>
</tr>
<tr>
<td>4352342</td>
<td>Fred</td>
<td>500000</td>
</tr>
</tbody>
</table>
Projection (\(\Pi\))

- Eliminates columns, then removes duplicates
- Notation: \(\Pi_{A_1,...,A_n}(R)\)
- Example: project social-security number and names:
  - \(\Pi_{SSN,Name}(Employee)\)
  - Output schema: \(Answer (SSN, Name)\)

SQL:

```
SELECT DISTINCT 
sname, gpa
FROM Students;
```

RA:

\[\Pi_{sname,gpa}(Students)\]

Another example:

<table>
<thead>
<tr>
<th>SSN</th>
<th>Name</th>
<th>Salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>John</td>
<td>200000</td>
</tr>
<tr>
<td>5423341</td>
<td>John</td>
<td>600000</td>
</tr>
<tr>
<td>4352342</td>
<td>John</td>
<td>200000</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>200000</td>
</tr>
<tr>
<td>John</td>
<td>600000</td>
</tr>
</tbody>
</table>
Note that RA Operators are Compositional!

```
Students(sid,sname,gpa)
```

```sql
SELECT DISTINCT sname, gpa FROM Students WHERE gpa > 3.5;
```

How do we represent this query in RA?

```
\( \Pi_{sname,gpa}(\sigma_{gpa>3.5}(Students)) \)
```

```
\( \sigma_{gpa>3.5}(\Pi_{sname,gpa}(Students)) \)
```

Are these logically equivalent?

Cross-Product (\( \times \))

- Each tuple in R1 with each tuple in R2
- Notation: R1 \( \times \) R2
- Example:
  - Employee \( \times \) Departments
  - Mainly used to express joins

```
Students(sid,sname,gpa)
People(ssn,pname,address)
```

```
SQL:
SELECT *
FROM Students, People;
```

```
RA:
Students \( \times \) People
```
Another example:

<table>
<thead>
<tr>
<th>People</th>
<th>Students</th>
</tr>
</thead>
<tbody>
<tr>
<td>ssn</td>
<td>sid</td>
</tr>
<tr>
<td>1234545</td>
<td>001</td>
</tr>
<tr>
<td>5423341</td>
<td>002</td>
</tr>
<tr>
<td>pname</td>
<td>snam e</td>
</tr>
<tr>
<td>John</td>
<td>John</td>
</tr>
<tr>
<td>216 Rosse</td>
<td>3.4</td>
</tr>
<tr>
<td>address</td>
<td>gpa</td>
</tr>
<tr>
<td>217 Rosse</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Students $\times$ People

<table>
<thead>
<tr>
<th>ssn</th>
<th>sid</th>
<th>sname</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>001</td>
<td>John</td>
<td>3.4</td>
</tr>
<tr>
<td>5423341</td>
<td>002</td>
<td>Bob</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Renaming ($\rho$ – Rho)

- Changes the schema, not the instance
- A ‘special’ operator- neither basic nor derived
- Notation: $\rho_{B_1, \ldots, B_n}(R)$

**Note:** this is shorthand for the proper form (since names, not order matters!):

- $\rho_{A_1 \rightarrow B_1, \ldots, A_n \rightarrow B_n}(R)$

**SQL:**

```sql
SELECT sid AS studId, sname AS name, gpa AS gradePtAvg
FROM Students;
```

**RA:**

$\rho_{studId, name, gradePtAvg}(Students)$

We care about this operator because we are working in a named perspective
Another example:

\[ \rho_{\text{studId}, \text{name}, \text{gradePtAvg}}(\text{Students}) \]

<table>
<thead>
<tr>
<th>studId</th>
<th>name</th>
<th>gradePtAvg</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>John</td>
<td>3.4</td>
</tr>
<tr>
<td>002</td>
<td>Bob</td>
<td>1.3</td>
</tr>
</tbody>
</table>

Natural Join (\(\Join\))

- Notation: \(R_1 \Join R_2\)

- Joins \(R_1\) and \(R_2\) on equality of all shared attributes
  - If \(R_1\) has attribute set \(A\), and \(R_2\) has attribute set \(B\), and they share attributes \(A \cap B = C\), can also be written: \(R_1 \Join C \Join R_2\)

- Our first example of a derived RA operator:
  - Meaning: \(R_1 \Join R_2 = \Pi_{A \cup B}(\sigma_{C=D}(\rho_{C\rightarrow D}(R_1) \times R_2))\)
  - Where:
    - The rename \(\rho_{C\rightarrow D}\) renames the shared attributes in one of the relations
    - The selection \(\sigma_{C=D}\) checks equality of the shared attributes
    - The projection \(\Pi_{A \cup B}\) eliminates the duplicate common attributes

SQL:

```
SELECT DISTINCT ssid, S.name, gpa, ssn, address
FROM Students S, People P
WHERE S.name = P.name;
```
Another example:

<table>
<thead>
<tr>
<th>Students S</th>
<th>S.name</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>John</td>
<td>3.4</td>
</tr>
<tr>
<td>002</td>
<td>Bob</td>
<td>1.3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>People P</th>
<th>ssn</th>
<th>P.name</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234545</td>
<td>John</td>
<td>216 Rosse</td>
<td></td>
</tr>
<tr>
<td>5423341</td>
<td>Bob</td>
<td>217 Rosse</td>
<td></td>
</tr>
</tbody>
</table>

**Students ⋈ People**

<table>
<thead>
<tr>
<th>sid</th>
<th>S.name</th>
<th>gpa</th>
<th>ssn</th>
<th>address</th>
</tr>
</thead>
<tbody>
<tr>
<td>001</td>
<td>John</td>
<td>3.4</td>
<td>1234545</td>
<td>216 Rosse</td>
</tr>
<tr>
<td>002</td>
<td>Bob</td>
<td>1.3</td>
<td>5423341</td>
<td>217 Rosse</td>
</tr>
</tbody>
</table>

**Natural Join**

- Given schemas $R(A, B, C, D)$, $S(A, C, E)$, what is the schema of $R ⋈ S$ ?

- Given $R(A, B, C)$, $S(D, E)$, what is $R ⋈ S$ ?

- Given $R(A, B)$, $S(A, B)$, what is $R ⋈ S$ ?
Example: Converting SFW Query -> RA

\[
\text{SELECT DISTINCT gpa, address FROM Students S, People P WHERE gpa > 3.5 AND S.sname = P.sname;}
\]

\[\Pi_{\text{gpa}, \text{address}}(\sigma_{\text{gpa}>3.5}(S \bowtie P))\]

How do we represent this query in RA?

Division

- Notation: \( r ÷ s \)
- It has nothing to do with arithmetic division.
- Let \( r \) and \( s \) be relations on schemas \( R \) and \( S \) respectively where
  - \( R = (A_1, \ldots, A_m, B_1, \ldots, B_n) \)
  - \( S = (B_1, \ldots, B_n) \)

The result of \( r ÷ s \) is a relation on schema \( R - S = (A_1, \ldots, A_m) \)

\[
\frac{r}{s} = \left\{ t \mid t \in \Pi_{R-S}(r) \land \forall u \in s (tu \in r) \right\}
\]

Where \( tu \) means the concatenation of tuples \( t \) and \( u \) to produce a single tuple.
Division Operation - Example

Relations $r, s$

\[
\begin{array}{c|c}
A & B \\
\hline
a & 1 \\
\alpha & 2 \\
\beta & 3 \\
\gamma & 4 \\
\delta & 5 \\
\epsilon & 6 \\
\beta & 2 \\
\end{array}
\quad
\begin{array}{c}
B \\
1 \\
2 \\
\hline
s
\end{array}
\]

$\quad r \div s$

\[
\begin{array}{c|c}
A \\
\hline
\alpha \\
\beta \\
\end{array}
\quad
\begin{array}{c|c}
D & E \\
\hline
a & 1 \\
\alpha & 1 \\
\gamma & 1 \\
\beta & 3 \\
\gamma & 1 \\
\gamma & 1 \\
\beta & 1 \\
\end{array}
\]

$\quad r \times s$

\[
\begin{array}{c|c|c}
A & B & C \\
\hline
\alpha & a & \alpha \\
\alpha & a & \gamma \\
\beta & a & \gamma \\
\beta & a & \gamma \\
\gamma & a & \gamma \\
\gamma & a & \gamma \\
\gamma & a & \beta \\
\end{array}
\]

Database System Concepts, Silberschatz, Korth and Sudarshan
Division Operation - Example

Department (dID, dName)
Project (pID, pName, dID)
Supplier (suppID, sName, sAddress)
Supply (sCode, sName, amountAvailable)
Production (sCode, dID, amount)
Consumption (sCode, pID, amount)
Purchased (sCode, suppId, amount)

- Find projects that consume all different kinds of purchased supplies.
  \[ \Pi_{pID,sCode} \text{ (Consumption)} \div \text{Purchased} \]

- Find departments that produce all kinds of supplies.
  \[ \Pi_{dID,sCode} \text{ (Production)} \div \text{Supply} \]

Logical Equivalence of RA Plans

- Given relations R(A,B):
  - Here, projection & selection commute:
    \[ \sigma_{A=5}(\Pi_A(R)) = \Pi_A(\sigma_{A=5}(R)) \]
  - What about here?
    \[ \sigma_{A=5}(\Pi_B(R)) = \Pi_B(\sigma_{A=5}(R)) \quad ??? \]
RDBMS Architecture

How does a SQL engine work?

SQL Query → Relational Algebra (RA) Plan → Optimized RA Plan → Execution

We saw how we can transform declarative SQL queries into precise, compositional RA plans.

RDBMS Architecture

How does a SQL engine work?

SQL Query → Relational Algebra (RA) Plan → Optimized RA Plan → Execution

We’ll look at how to then optimize these plans later!
RDBMS Architecture

How is the RA “plan” executed?

SQL Query → Relational Algebra (RA) Plan → Optimized RA Plan → Execution

We will see later how to execute all the basic operators!

2. Advanced Relational Algebra
What you will learn about in this section

1. Set Operations in RA
2. Fancier RA
3. Extensions & Limitations

Relational Algebra (RA)

- **Five basic operators:**
  - Selection: \( \sigma \)
  - Projection: \( \Pi \)
  - Cartesian Product: \( \times \)
  - Union: \( \cup \)
  - Difference: \( - \)

- **Derived or auxiliary operators:**
  - Intersection, complement
  - Joins (natural, equi-join, theta join, semi-join)
  - Renaming: \( \rho \)
  - Division
Union ($\cup$) and Difference ($-$)

- $R_1 \cup R_2$
- Example:
  - $\text{ActiveEmployees} \cup \text{RetiredEmployees}$

- $R_1 - R_2$
- Example:
  - $\text{AllEmployees} - \text{RetiredEmployees}$

What about Intersection ($\cap$)?

- It is a derived operator
- $R_1 \cap R_2 = R_1 - (R_1 - R_2)$
- Also expressed as a join!
- Example
  - $\text{UnionizedEmployees} \cap \text{RetiredEmployees}$
Theta Join ($\bowtie_\theta$)

- A join that involves a predicate
- $R_1 \bowtie_\theta R_2 = \sigma_\theta (R_1 \times R_2)$
- Here $\theta$ can be any condition

Note that natural join is a theta join + a projection.

SQL:

```sql
SELECT * FROM Students, People WHERE \theta;
```

RA:

$Students \bowtie_\theta People$

Equi-join ($\bowtie_{A=B}$)

- A theta join where $\theta$ is an equality
- $R_1 \bowtie_{A=B} R_2 = \sigma_{A=B} (R_1 \times R_2)$
- Example:
  - Employee $\bowtie_{SSN=SSN}$ Dependents

Most common join in practice!
Semijoin (⋈)

- \( R \bowtie S = \Pi_{A_1, \ldots, A_n} (R \bowtie S) \)
- Where \( A_1, \ldots, A_n \) are the attributes in \( R \)
- Example:
  - Employee \( \bowtie \) Dependents

### SQL:

```
SELECT DISTINCT sid, sname, gpa
FROM Students, People
WHERE sname = pname;
```

### RA:

Students \( \bowtie \) People

Semijoins in Distributed Databases

- Semijoins are often used to compute natural joins in distributed databases

![Diagram](image)

Employee \( \bowtie_{\text{ssn}=\text{ssn}} (\sigma_{\text{age} > 71}(\text{Dependents})) \)

Send less data to reduce network bandwidth!
RA Expressions Can Get Complex!

Operations on Multisets

All RA operations need to be defined carefully on bags

- $\sigma_c(R)$: preserve the number of occurrences
- $\Pi_A(R)$: no duplicate elimination
- Cross-product, join: no duplicate elimination

This is important: relational engines work on multisets, not sets!
RA has Limitations!

- Cannot compute “transitive closure”

<table>
<thead>
<tr>
<th>Name1</th>
<th>Name2</th>
<th>Relationship</th>
</tr>
</thead>
<tbody>
<tr>
<td>Fred</td>
<td>Mary</td>
<td>Father</td>
</tr>
<tr>
<td>Mary</td>
<td>Joe</td>
<td>Cousin</td>
</tr>
<tr>
<td>Mary</td>
<td>Bill</td>
<td>Spouse</td>
</tr>
<tr>
<td>Nancy</td>
<td>Lou</td>
<td>Sister</td>
</tr>
</tbody>
</table>

- Find all direct and indirect relatives of Fred
- Cannot express in RA !!!
  - Need to write C program, use a graph engine, or modern SQL...

Example: Banking Database

**branch** (branch_name, branch_city, assets)
**customer** (customer_name, customer_street, customer_city)
**account** (account_number, branch_name, balance)
**loan** (loan_number, branch_name, amount)
**depositor** (customer_name, account_number)
**borrower** (customer_name, loan_number)
Example Queries

• Find all loans of over $1200

\[ \sigma_{amount > 1200} (loan) \]

• Find the loan number for each loan of an amount greater than $1200

\[ \Pi_{loan\_number} (\sigma_{amount > 1200} (loan)) \]
Example Queries

• Find the names of all customers who have a loan, an account, or both, from the bank.

\[ \Pi_{\text{customer\_name}}(\text{borrower}) \cup \Pi_{\text{customer\_name}}(\text{depositor}) \]

• Find the names of all customers who have a loan and an account at bank.

\[ \Pi_{\text{customer\_name}}(\text{borrower}) \cap \Pi_{\text{customer\_name}}(\text{depositor}) \]

Example Queries

• Find the names of all customers who have a loan at the Perryridge branch.

\[ \Pi_{\text{customer\_name}}(\sigma_{\text{branch\_name}=“Perryridge”} (\sigma_{\text{borrower.loan\_number} = \text{loan.loan\_number}(\text{borrower \times loan}))}} \]

• Find the names of all customers who have a loan at the Perryridge branch but do not have an account at any branch of the bank.

\[ \Pi_{\text{customer\_name}}(\sigma_{\text{branch\_name}=“Perryridge”} (\sigma_{\text{borrower.loan\_number} = \text{loan.loan\_number}(\text{borrower \times loan}))} – \Pi_{\text{customer\_name}}(\text{depositor}) \]
Example Queries

• Find the names of all customers who have a loan at the Perryridge branch.

  ● Query 1
  \[ \Pi_{\text{customer\_name}} (\sigma_{\text{branch\_name} = "Perryridge"} (\sigma_{\text{borrower\_loan\_number} = \text{loan\_loan\_number}} (\text{borrower} \times \text{loan}))) \]

  ● Query 2
  \[ \Pi_{\text{customer\_name}} (\sigma_{\text{loan\_loan\_number} = \text{borrower\_loan\_number}} (\sigma_{\text{branch\_name} = "Perryridge"} (\text{loan} \times \text{borrower}))) \]

Example Queries

• Find all customers who have an account at all branches located in Brooklyn city.

\[ \Pi_{\text{customer\_name, branch\_name}} (\sigma_{\text{branch\_city} = "Brooklyn"} (\text{branch})) \quad \div \quad \Pi_{\text{branch\_name}} \]

Acknowledgements

The course material used for this lecture is mostly taken and/or adopted from the course materials of the CS145 Introduction to Databases lecture given by Christopher Ré at Stanford University (http://web.stanford.edu/class/cs145/).