First assignment will be available today
- Due next Thursday, October 5, 2:00 AM

TAs will be decided soon
- Should start having office hours on Sunday or Monday
Query Languages

- A query language specifies how to access the data in the database.

- Different kinds of query languages:
  - Declarative languages specify what data to retrieve, but not how to retrieve it.
  - Procedural languages specify what to retrieve, as well as the process for retrieving it.

- Query languages often include updating and deleting data as well.

- Also called data manipulation language (DML).
The Relational Algebra

- A procedural query language
- Comprised of relational algebra operations
- Relational operations:
  - Take one or two relations as input
  - Produce a relation as output
- Relational operations can be composed together
  - Each operation produces a relation
  - A query is simply a relational algebra expression
- Six “fundamental” relational operations
- Other useful operations can be composed from these fundamental operations
“Why is this useful?”

- SQL is only loosely based on relational algebra
- SQL is much more on the “declarative” end of the spectrum
- Many relational database implementations use relational algebra operations as a basis for representing execution plans
  - Simple, clean, effective abstraction for representing how results will be generated
  - Relatively easy to manipulate for query optimization
Fundamental Relational Algebra Operations

- Six fundamental operations:
  - \( \sigma \) select operation
  - \( \Pi \) project operation
  - \( \cup \) set-union operation
  - \( - \) set-difference operation
  - \( \times \) Cartesian product operation
  - \( \rho \) rename operation

- Each operation takes one or two relations as input
- Produces another relation as output

- Important details:
  - What tuples are included in the result relation?
  - Any constraints on input schemas? What is schema of result?
Select Operation

- **Written as:** $\sigma_P(r)$
- **P** is the predicate for selection
  - **P** can refer to attributes in $r$ (but no other relation!), as well as literal values
  - Can use comparison operators: $=, \neq, <, \leq, >, \geq$
  - Can combine multiple predicates using: $\land$ (and), $\lor$ (or), $\neg$ (not)
- **r** is the input relation
- Result relation contains all tuples in $r$ for which P is true
- Result schema is identical to schema for $r$
Select Examples

Using the account relation:

“Retrieve all tuples for accounts in the Los Angeles branch.”

\[ \sigma_{\text{branch name}= \text{“Los Angeles”}}(\text{account}) \]

“Retrieve all tuples for accounts in the Los Angeles branch, with a balance under $300.”

\[ \sigma_{\text{branch name}= \text{“Los Angeles”}/ \text{balance}<300}(\text{account}) \]
Project Operation

- Written as: $\Pi_{a,b,...}(r)$

- Result relation contains only specified attributes of $r$
  - Specified attributes must actually be in schema of $r$
  - Result’s schema only contains the specified attributes
  - Domains are same as source attributes’ domains

- Important note:
  - Result relation may have fewer rows than input relation!
  - Why?
    - Relations are sets of tuples, not multisets
Project Example

Using the account relation:

"Retrieve all branch names that have at least one account."

\[ \Pi_{\text{branch\_name}}(\text{account}) \]

- Result only has three tuples, even though input has five
- Result schema is just \( \text{branch\_name} \)
Composing Operations

- Input can also be an expression that evaluates to a relation, instead of just a relation

- \( \Pi_{\text{acct\_id}}(\sigma_{\text{balance} \geq 300}(\text{account})) \)
  - Selects the account IDs of all accounts with a balance of $300 or more
  - Input relation’s schema is:
    \( \text{Account\_schema} = (\text{acct\_id}, \text{branch\_name}, \text{balance}) \)
  - Final result relation’s schema?
    - Just one attribute: \( (\text{acct\_id}) \)

- Distinguish between base and derived relations
  - \text{account} is a base relation
  - \( \sigma_{\text{balance} \geq 300}(\text{account}) \) is a derived relation
Set-Union Operation

- Written as: \( r \cup s \)
- Result contains all tuples from \( r \) and \( s \)
  - Each tuple is unique, even if it’s in both \( r \) and \( s \)
- Constraints on schemas for \( r \) and \( s \)?
- \( r \) and \( s \) must have compatible schemas:
  - \( r \) and \( s \) must have same arity
    - (same number of attributes)
  - For each attribute \( i \) in \( r \) and \( s \), \( r[i] \) must have the same domain as \( s[i] \)
  - (Our examples also generally have same attribute names, but not required! Arity and domains are what matter.)
Set-Union Example

More complicated schema: accounts and loans

<table>
<thead>
<tr>
<th>acct_id</th>
<th>branch_name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-301</td>
<td>New York</td>
<td>350</td>
</tr>
<tr>
<td>A-307</td>
<td>Seattle</td>
<td>275</td>
</tr>
<tr>
<td>A-318</td>
<td>Los Angeles</td>
<td>550</td>
</tr>
<tr>
<td>A-319</td>
<td>New York</td>
<td>80</td>
</tr>
<tr>
<td>A-322</td>
<td>Los Angeles</td>
<td>275</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loan_id</th>
<th>branch_name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-421</td>
<td>San Francisco</td>
<td>7500</td>
</tr>
<tr>
<td>L-445</td>
<td>Los Angeles</td>
<td>2000</td>
</tr>
<tr>
<td>L-437</td>
<td>Las Vegas</td>
<td>4300</td>
</tr>
<tr>
<td>L-419</td>
<td>Seattle</td>
<td>2900</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cust_name</th>
<th>acct_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson</td>
<td>A-318</td>
</tr>
<tr>
<td>Smith</td>
<td>A-322</td>
</tr>
<tr>
<td>Reynolds</td>
<td>A-319</td>
</tr>
<tr>
<td>Lewis</td>
<td>A-307</td>
</tr>
<tr>
<td>Reynolds</td>
<td>A-301</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cust_name</th>
<th>loan_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>L-437</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-419</td>
</tr>
<tr>
<td>Lewis</td>
<td>L-421</td>
</tr>
<tr>
<td>Smith</td>
<td>L-445</td>
</tr>
</tbody>
</table>
Find names of all customers that have either a bank account or a loan at the bank.

<table>
<thead>
<tr>
<th>acct_id</th>
<th>branch_name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-301</td>
<td>New York</td>
<td>350</td>
</tr>
<tr>
<td>A-307</td>
<td>Seattle</td>
<td>275</td>
</tr>
<tr>
<td>A-318</td>
<td>Los Angeles</td>
<td>550</td>
</tr>
<tr>
<td>A-319</td>
<td>New York</td>
<td>80</td>
</tr>
<tr>
<td>A-322</td>
<td>Los Angeles</td>
<td>275</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cust_name</th>
<th>acct_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Johnson</td>
<td>A-318</td>
</tr>
<tr>
<td>Smith</td>
<td>A-322</td>
</tr>
<tr>
<td>Reynolds</td>
<td>A-319</td>
</tr>
<tr>
<td>Lewis</td>
<td>A-307</td>
</tr>
<tr>
<td>Reynolds</td>
<td>A-301</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loan_id</th>
<th>branch_name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-421</td>
<td>San Francisco</td>
<td>7500</td>
</tr>
<tr>
<td>L-445</td>
<td>Los Angeles</td>
<td>2000</td>
</tr>
<tr>
<td>L-437</td>
<td>Las Vegas</td>
<td>4300</td>
</tr>
<tr>
<td>L-419</td>
<td>Seattle</td>
<td>2900</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>cust_name</th>
<th>loan_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>L-437</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-419</td>
</tr>
<tr>
<td>Lewis</td>
<td>L-421</td>
</tr>
<tr>
<td>Smith</td>
<td>L-445</td>
</tr>
</tbody>
</table>
Set-Union Example (3)

- Find names of all customers that have either a bank account or a loan at the bank
  - Easy to find the customers with an account:
    \[ \Pi_{\text{cust\_name}}(\text{depositor}) \]
  - Also easy to find customers with a loan:
    \[ \Pi_{\text{cust\_name}}(\text{borrower}) \]

- Result is set-union of these expressions:
  \[ \Pi_{\text{cust\_name}}(\text{depositor}) \cup \Pi_{\text{cust\_name}}(\text{borrower}) \]

- Note that inputs have 8 tuples, but result has 6 tuples.
Set-Difference Operation

- Written as: \( r - s \)
- Result contains tuples that are only in \( r \), but not in \( s \)
  - Tuples in both \( r \) and \( s \) are excluded
  - Tuples only in \( s \) are also excluded
- Constraints on schemas of \( r \) and \( s \)?
  - Schemas must be compatible
  - (Exactly like set-union.)
Set-Difference Example

“Find all customers that have an account but not a loan.”
Set-Difference Example (2)

- Again, each component is easy
  - All customers that have an account:
    \[ \Pi_{\text{cust\_name}}(\text{depositor}) \]
  - All customers that have a loan:
    \[ \Pi_{\text{cust\_name}}(\text{borrower}) \]

- Result is set-difference of these expressions
  \[ \Pi_{\text{cust\_name}}(\text{depositor}) - \Pi_{\text{cust\_name}}(\text{borrower}) \]
Cartesian Product Operation

- **Written as:** $r \times s$
  - Read as “$r$ cross $s$”
- **No** constraints on schemas of $r$ and $s$
- **Schema of result is** concatenation **of schemas for** $r$ and $s$
- **If** $r$ and $s$ have overlapping attribute names:
  - **All** overlapping attributes are included; none are eliminated
  - Distinguish overlapping attribute names by prepending the source relation’s name
- **Example:**
  - Input relations: $r(a, b)$ and $s(b, c)$
  - Schema of $r \times s$ is $(a, r.b, s.b, c)$
Cartesian Product Operation (2)

- Result of $r \times s$
  - Contains every tuple in $r$, combined with every tuple in $s$
  - If $r$ contains $N_r$ tuples, and $s$ contains $N_s$ tuples, result contains $N_r \times N_s$ tuples

- Allows two relations to be compared and/or combined
  - If we want to correlate tuples in relation $r$ with tuples in relation $s$...
  - Compute $r \times s$, then select out desired results with an appropriate predicate
Cartesian Product Example

- Compute result of borrower $\times$ loan

<table>
<thead>
<tr>
<th>cust_name</th>
<th>loan_id</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>L-437</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-419</td>
</tr>
<tr>
<td>Lewis</td>
<td>L-421</td>
</tr>
<tr>
<td>Smith</td>
<td>L-445</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>loan_id</th>
<th>branch_name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-421</td>
<td>San Francisco</td>
<td>7500</td>
</tr>
<tr>
<td>L-445</td>
<td>Los Angeles</td>
<td>2000</td>
</tr>
<tr>
<td>L-437</td>
<td>Las Vegas</td>
<td>4300</td>
</tr>
<tr>
<td>L-419</td>
<td>Seattle</td>
<td>2900</td>
</tr>
</tbody>
</table>

- Result will contain $4 \times 4 = 16$ tuples
Cartesian Product Example (2)

- Schema for borrower is:
  \[ \text{Borrower\_schema} = (\text{cust\_name}, \text{loan\_id}) \]

- Schema for loan is:
  \[ \text{Loan\_schema} = (\text{loan\_id}, \text{branch\_name}, \text{amount}) \]

- Schema for result of borrower $\times$ loan is:
  \[ (\text{cust\_name}, \text{borrower.loan\_id}, \text{loan.loan\_id}, \text{branch\_name}, \text{amount}) \]

  Overlapping attribute names are distinguished by including name of source relation
Cartesian Product Example (3)

Result:

<table>
<thead>
<tr>
<th>cust_name</th>
<th>loan_id</th>
<th>loan_id</th>
<th>branch_name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>Anderson</td>
<td>L-437</td>
<td>L-421</td>
<td>San Francisco</td>
<td>7500</td>
</tr>
<tr>
<td>Anderson</td>
<td>L-437</td>
<td>L-445</td>
<td>Los Angeles</td>
<td>2000</td>
</tr>
<tr>
<td>Anderson</td>
<td>L-437</td>
<td>L-437</td>
<td>Las Vegas</td>
<td>4300</td>
</tr>
<tr>
<td>Anderson</td>
<td>L-437</td>
<td>L-419</td>
<td>Seattle</td>
<td>2900</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-419</td>
<td>L-421</td>
<td>San Francisco</td>
<td>7500</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-419</td>
<td>L-445</td>
<td>Los Angeles</td>
<td>2000</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-419</td>
<td>L-437</td>
<td>Las Vegas</td>
<td>4300</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-419</td>
<td>L-419</td>
<td>Seattle</td>
<td>2900</td>
</tr>
<tr>
<td>Lewis</td>
<td>L-421</td>
<td>L-421</td>
<td>San Francisco</td>
<td>7500</td>
</tr>
<tr>
<td>Lewis</td>
<td>L-421</td>
<td>L-445</td>
<td>Los Angeles</td>
<td>2000</td>
</tr>
<tr>
<td>Lewis</td>
<td>L-421</td>
<td>L-437</td>
<td>Las Vegas</td>
<td>4300</td>
</tr>
<tr>
<td>Lewis</td>
<td>L-421</td>
<td>L-419</td>
<td>Seattle</td>
<td>2900</td>
</tr>
<tr>
<td>Smith</td>
<td>L-445</td>
<td>L-421</td>
<td>San Francisco</td>
<td>7500</td>
</tr>
<tr>
<td>Smith</td>
<td>L-445</td>
<td>L-437</td>
<td>Las Vegas</td>
<td>4300</td>
</tr>
<tr>
<td>Smith</td>
<td>L-445</td>
<td>L-419</td>
<td>Seattle</td>
<td>2900</td>
</tr>
</tbody>
</table>
Can use Cartesian product to associate related rows between two tables

...but, a lot of extra rows are included!

<table>
<thead>
<tr>
<th>cust_name</th>
<th>borrower.loan_id</th>
<th>loan.loan_id</th>
<th>branch_name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-419</td>
<td>L-437</td>
<td>Las Vegas</td>
<td>4300</td>
</tr>
<tr>
<td>Jackson</td>
<td>L-419</td>
<td>L-419</td>
<td>Seattle</td>
<td>2900</td>
</tr>
<tr>
<td>Lewis</td>
<td>L-421</td>
<td>L-421</td>
<td>San Francisco</td>
<td>7500</td>
</tr>
<tr>
<td>Lewis</td>
<td>L-421</td>
<td>L-445</td>
<td>Los Angeles</td>
<td>2000</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

Combine Cartesian product with a select operation

\[ \sigma_{\text{borrower.loan}_id = \text{loan.loan}_id}(\text{borrower} \times \text{loan}) \]
Cartesian Product Example (5)

- “Retrieve the names of all customers with loans at the Seattle branch.”

- Need both borrower and loan relations
- Correlate tuples in the relations using loan_id
- Then, computing result is easy.
Associate customer names with loan details, using Cartesian product and a select:

$$\sigma_{\text{borrower} \cdot \text{loan}_\text{id} = \text{loan} \cdot \text{loan}_\text{id}} (\text{borrower} \times \text{loan})$$

Select out loans at Seattle branch:

$$\sigma_{\text{branch}_\text{name} = \text{“Seattle”}} (\sigma_{\text{borrower} \cdot \text{loan}_\text{id} = \text{loan} \cdot \text{loan}_\text{id}} (\text{borrower} \times \text{loan}))$$

Simplify:

$$\sigma_{\text{borrower} \cdot \text{loan}_\text{id} = \text{loan} \cdot \text{loan}_\text{id} \land \text{branch}_\text{name} = \text{“Seattle”}} (\text{borrower} \times \text{loan})$$

Project results down to customer name:

$$\Pi_{\text{cust}_\text{name}} (\sigma_{\text{borrower} \cdot \text{loan}_\text{id} = \text{loan} \cdot \text{loan}_\text{id} \land \text{branch}_\text{name} = \text{“Seattle”}} (\text{borrower} \times \text{loan}))$$

Final result:

<table>
<thead>
<tr>
<th>cust_name</th>
</tr>
</thead>
<tbody>
<tr>
<td>Jackson</td>
</tr>
</tbody>
</table>
Rename Operation

- Results of relational operations are unnamed
  - Result has a schema, but the relation itself is unnamed
- Can give result a name using the rename operator
- Written as: $\rho_x(E)$ (Greek rho, not lowercase “P”)
  - $E$ is an expression that produces a relation
  - $E$ can also be a named relation or a relation-variable
  - $x$ is new name of relation
- More general form is: $\rho_x(A_1, A_2, ..., A_n)(E)$
  - Allows renaming of relation’s attributes
  - Requirement: $E$ has arity $n$
Scope of Renamed Relations

- Rename operation $\rho$ only applies within a specific relational algebra expression
  - This does not create a new relation-variable!
  - The new name is only visible to enclosing relational-algebra expressions

- Rename operator is used for two main purposes:
  - Allow a derived relation and its attributes to be referred to by enclosing relational-algebra operations
  - Allow a base relation to be used multiple ways in one query
    - $r \times \rho_s(r)$

- In other words, rename operation $\rho$ is used to resolve ambiguities within a specific relational algebra expression
“Find the ID of the loan with the largest amount.”

- Hard to find the loan with the largest amount!
  - (At least, with the tools we have so far…)
- Much easier to find all loans that have an amount smaller than some other loan
- Then, use set-difference to find the largest loan

<table>
<thead>
<tr>
<th>loan_id</th>
<th>branch_name</th>
<th>amount</th>
</tr>
</thead>
<tbody>
<tr>
<td>L-421</td>
<td>San Francisco</td>
<td>7500</td>
</tr>
<tr>
<td>L-445</td>
<td>Los Angeles</td>
<td>2000</td>
</tr>
<tr>
<td>L-437</td>
<td>Las Vegas</td>
<td>4300</td>
</tr>
<tr>
<td>L-419</td>
<td>Seattle</td>
<td>2900</td>
</tr>
</tbody>
</table>
How to find all loans with an amount smaller than some other loan?

- Use Cartesian Product of loan with itself:
  \[ \text{loan} \times \text{loan} \]
- Compare each loan’s amount to all other loans

Problem: Can’t distinguish between attributes of left and right loan relations!

Solution: Use rename operation

\[ \text{loan} \times \rho_{\text{test}}(\text{loan}) \]
- Now, right relation is named test
Find IDs of all loans with an amount smaller than some other loan:

\[ \Pi_{\text{loan.loan\_id}}(\sigma_{\text{loan.amount}<\text{test.amount}}(\text{loan} \times \rho_{\text{test}}(\text{loan}))) \]

Finally, we can get our result:

\[ \Pi_{\text{loan\_id}}(\text{loan}) - \Pi_{\text{loan.loan\_id}}(\sigma_{\text{loan.amount}<\text{test.amount}}(\text{loan} \times \rho_{\text{test}}(\text{loan}))) \]

What if multiple loans have max value?
- All loans with max value appear in result.
Additional Relational Operations

- The fundamental operations are sufficient to query a relational database...
- Can produce some large expressions for common operations!
- Several additional operations, defined in terms of fundamental operations:
  - \( \cap \) set-intersection
  - \( \bowtie \) natural join
  - \( \div \) division
  - \( \leftarrow \) assignment
Set-Intersection Operation

- Written as:  \( r \cap s \)

- \( r \cap s = r - (r - s) \)
  
  \( r - s = \) the rows in \( r \), but not in \( s \)
  
  \( r - (r - s) = \) the rows in both \( r \) and \( s \)

- Relations must have compatible schemas

- Example: find all customers with both a loan and a bank account

  \[ \Pi_{\text{cust\_name}}(\text{borrower}) \cap \Pi_{\text{cust\_name}}(\text{depositor}) \]
Most common use of Cartesian product is to correlate tuples with the same key-values
- Called a join operation

The natural join is a shorthand for this operation

Written as: \( r \bowtie s \)
- \( r \) and \( s \) must have common attributes
- The common attributes are usually a key for \( r \) and/or \( s \), but certainly don’t have to be
Natural Join Definition

- For two relations \( r(R) \) and \( s(S) \)
- Attributes used to perform natural join:
  \( R \cap S = \{A_1, A_2, \ldots, A_n\} \)
- Formal definition:
  \[ r \bowtie s = \Pi_{R \cup S} (\sigma_{r.A_1 = s.A_1 \land r.A_2 = s.A_2 \land \ldots \land r.A_n = s.A_n} (r \times s) ) \]
  - \( r \) and \( s \) are joined using an equality condition based on their common attributes
  - Result is projected so that common attributes only appear once
Natural Join Example

- **Simple example:**
  
  “Find the names of all customers with loans.”

- **Result:**
  
  \[ \Pi_{\text{cust\_name}}(\sigma_{\text{borrower.loan\_id}=\text{loan.loan\_id}}(\text{borrower } \times \text{ loan})) \]

- **Rewritten with natural join:**
  
  \[ \Pi_{\text{cust\_name}}(\text{borrower } \bowtie \text{ loan}) \]
Natural Join Characteristics

- Very common to compute joins across multiple tables
- Example: $\text{customer} \bowtie \text{borrower} \bowtie \text{loan}$
- Natural join operation is associative:
  
  $(\text{customer} \bowtie \text{borrower}) \bowtie \text{loan}$ is equivalent to $\text{customer} \bowtie (\text{borrower} \bowtie \text{loan})$

- Note:
  
  Even though these expressions are equivalent, order of join operations can dramatically affect query cost!
  
  (Keep this in mind for later…)
Division Operation

- Binary operator: \( r \div s \)
- Implements a “for each” type of query
  - “Find all rows in \( r \) that have one row corresponding to each row in \( s \).”
  - Relation \( r \) divided by relation \( s \)
- Easiest to illustrate with an example:
- Puzzle Database
  - \textit{puzzle\_list} (puzzle\_name)
    - Simple list of puzzles by name
  - \textit{completed} (person\_name, puzzle\_name)
    - Records which puzzles have been completed by each person
“Who has solved every puzzle?”

- Need to find every person in completed that has an entry for every puzzle in puzzle_list
- Divide completed by puzzle_list to get answer:
  \[
  \text{completed} \div \text{puzzle_list} = \\
  \begin{array}{|c|}
  \hline
  \text{person_name} \\
  \hline
  \text{Alex} \\
  \text{Carl} \\
  \hline
  \end{array}
  \]

- Only Alex and Carl have completed every puzzle in puzzle_list.
“Who has solved every puzzle?”

\[
\text{completed} \div \text{puzzle\_list} = \begin{array}{|c|}
\hline
\text{person\_name} \\
\hline
\text{Alex} \\
\text{Carl} \\
\hline
\end{array}
\]

- Very reminiscent of integer division
  - Result relation contains tuples from \text{completed} that are evenly divided by \text{puzzle\_name}
- Several other kinds of relational division operators
  - e.g. some can compute “remainder” of the division operation
Division Operation

For $r(R) \div s(S)$

- **Required:** $S \subseteq R$
  - All attributes in $S$ must also be in $R$

- **Result has schema** $R - S$
  - Result has attributes that are in $R$ but not also in $S$
  - (This is why we don’t allow $S = R$)

- **Every tuple $t$ in result satisfies these conditions:**
  
  $t \in \Pi_{R-S}(r)$
  
  $\langle \forall t_s \in s : \exists t_r \in r : t_r[S] = t_s[S] \land t_r[R-S] = t \rangle$

  - Every tuple in the result has a row in $r$ corresponding to every row in $s$
Puzzle Database

For completed ÷ puzzle_list

- Schemas are compatible
- Result has schema (person_name)
  - Attributes in completed schema, but not also in puzzle_list schema

Every tuple \( t \) in result satisfies these conditions:

\[
t \in \Pi_{R-S}(r) \\
\langle \forall t_s \in s : \exists t_r \in r : t_r[S] = t_s[S] \land t_r[R-S] = t \rangle
\]
Division Operation

- Not provided natively in most SQL databases
  - Rarely needed!
  - Easy enough to implement in SQL, if needed

- Will see it in the homework assignments, and on the midterm… 😊
  - Often a very nice shortcut for more involved queries
Recall: relation variables refer to a specific relation
- A specific set of tuples, with a particular schema

Example: account relation

<table>
<thead>
<tr>
<th>acct_id</th>
<th>branch_name</th>
<th>balance</th>
</tr>
</thead>
<tbody>
<tr>
<td>A-301</td>
<td>New York</td>
<td>350</td>
</tr>
<tr>
<td>A-307</td>
<td>Seattle</td>
<td>275</td>
</tr>
<tr>
<td>A-318</td>
<td>Los Angeles</td>
<td>550</td>
</tr>
<tr>
<td>A-319</td>
<td>New York</td>
<td>80</td>
</tr>
<tr>
<td>A-322</td>
<td>Los Angeles</td>
<td>275</td>
</tr>
</tbody>
</table>

account is actually technically a relation variable, as are all our named relations so far
Assignment Operation

- Can assign a relation-value to a relation-variable
  - Written as: \( \text{relvar} \leftarrow E \)
    - \( E \) is an expression that evaluates to a relation
- Unlike \( \rho \), the name \( \text{relvar} \) persists in the database
- Often used for temporary relation-variables:
  
  \[
  \begin{align*}
  \text{temp1} & \leftarrow \Pi_{R\leftarrow S}(r) \\
  \text{temp2} & \leftarrow \Pi_{R\leftarrow S}((\text{temp1} \times s) - \Pi_{R\leftarrow S,S}(r)) \\
  \text{result} & \leftarrow \text{temp1} - \text{temp2}
  \end{align*}
  \]
  - Query evaluation becomes a sequence of steps
  - (This is an implementation of the \( \div \) operator)
- Can also use assignment operation to modify data
  - More about updates next time…