CS 2451
Database Systems: Relational Algebra & Relational Calculus

http://www.seas.gwu.edu/~bhagiweb/cs2541
Spring 2018
Instructor: Dr. Bhagi Narahari

Relational Model Definitions
- A relation is a table with columns and rows.
- An attribute is a named column of a relation.
- A tuple is a row of a relation.
- A domain is a set of allowable values for one or more attributes.
- The degree of a relation is the number of attributes it contains.
- The cardinality of a relation is the number of tuples it contains.
- A relational database is a collection of normalized relations with distinct relation names.

These notes include examples using two different schemas – 'mini-banner' and 'company' database.

Codd’s Relational Algebra
- A set of mathematical operators that compose, modify, and combine tuples within different relations

- Relational algebra operations operate on relations and produce relations ("closure")
  
  f: Relation → Relation
  
  f: Relation x Relation → Relation
Preliminaries

- A query is applied to relation instances, and the result of a query is also a relation instance.
  - Schemas of input relations for a query are fixed (but query will run regardless of instance!)
  - The schema for the result of a given query is also fixed!
    Determined by definition of query language constructs.
- Positional (R[0]) vs. named-field notation (R.name):
  - Positional notation easier for formal definitions, named-field notation more readable.
    We will use named field notation
  - Both used in SQL

Relational Algebra

- A query language is used to update and retrieve data that is stored in a data model.
- Relational algebra is a set of relational operations for retrieving data.
  - Just like algebra with numbers, relational algebra consists of operands (which are relations) and a set of operators.
  - Every relational operator takes as input one or more relations and produces a relation as output.
    - Closure property - input is relations, output is relations
    - Unary operations - operate on one relation
    - Binary operations - have two relations as input
  - A sequence of relational algebra operators is called a relational algebra expression.

Relational Algebra Operators

- Basic operations:
  - Selection (σ) Selects a subset of rows from relation.
  - Projection (π) Deletes unwanted columns from relation.
  - Cross-product (×) Allows us to combine two relations.
  - Set-difference (−) Tuples in relation 1, but not in relation 2.
  - Union (∪) Tuples in relation 1 or in relation 2.
- Additional operations:
  - Intersection, join, assignment, division, renaming: Not essential, but (very!) useful.
- Since each operation returns a relation, operations can be composed! (Algebra is “closed”.)

Relational Algebra Expression: Syntax

- RA operators operate on relations and produce relations – closed algebra
  - Defined recursively
- (B) basic expression consists of a relation in the schema or a constant relation
  - What is a constant relation?
- (R) Let E₁ and E₂ be RA expressions, then
RA expressions...contd..

- \((E_1 \cup E_2)\) is a RA expression
- \((E_1 - E_2)\) is a RA expression
- \((E_1 \times E_2)\) is a RA expression
- \(\sigma_P (E_1)\) is a RA expression
- \(\pi_S (E_1)\) is a RA expression
- \(\rho_R (E_1)\) is a RA expression

Operations Can be composed
- If \(R_1, R_2\) are relations (sets), then \(R_1 <\text{op}> R_2\) is also a relation (set) -- <\text{op}> is any of the relational algebra operators
  - Closed algebra – how is closure defined ?
- Operations are defined as Set operations
  - Input is a set, output is a set
  - SQL allows duplicated, RA does not

Operator Precedence

- Just like mathematical operators, the relational operators have precedence.
  - The precedence of operators from highest to lowest is:
    - unary operators - \(\sigma, \Pi, \rho\)
    - Cartesian product and joins - \(\times, \bowtie\), division
    - intersection
    - union and set difference
  - Parentheses can be used to changed the order of operations.
  - It is a good idea to always use parentheses around the argument for both unary and binary operators.

Data Instance for Mini-Banner Example

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>Takes</th>
<th>COURSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
<td>sid</td>
</tr>
<tr>
<td>1</td>
<td>Jill</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Jack</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROFESSOR</th>
<th>Teaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>fid</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Narahari</td>
</tr>
<tr>
<td>2</td>
<td>Youssef</td>
</tr>
<tr>
<td>8</td>
<td>Choi</td>
</tr>
</tbody>
</table>

Projection Operation

- The projection operation \(\pi (\pi_i)\) is a unary operation that takes in a relation as input and returns a new relation as output that contains a subset of the attributes of the input relation and all non-duplicate tuples.
  - The output relation has the same number of tuples as the input relation unless removing the attributes caused duplicates to be present.
  - Question: When are we guaranteed to never have duplicates when performing a projection operation?
- Besides the relation, the projection operation takes as input the names of the attributes that are to be in the output relation.
  - Given a list of column names \(\alpha\) and a relation \(R, \pi_\alpha (R)\) extracts the columns in \(\alpha\) from the relation.
**Projection Operation Formal Definition**

- The projection operation on relation $R$ with output attributes $A_1, \ldots, A_m$ is denoted by $\Pi_{A_1, \ldots, A_m}(R)$.

$$\Pi_{A_1, \ldots, A_m}(R) = \{ [t[A_1, \ldots, A_m] | t \in R] \}$$

where

- $R$ is a relation, $t$ is a tuple variable
- $\{A_1, \ldots, A_m\}$ is a subset of the attributes of $R$ over which the projection will be performed.
- Order of $A_1, \ldots, A_m$ is significant in the result.
- Cardinality of $\Pi_{A_1, \ldots, A_m}(R)$ is not necessarily the same as $R$ because of duplicate removal.

---

**Projection, $\Pi_\alpha$**

- Example: find sid and grade from enrollment table

<table>
<thead>
<tr>
<th>sid</th>
<th>exp-grade</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>550-0103</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>500-0103</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

**Note:** duplicate elimination. In contrast, SQL returns by default a multiset and duplicates must be explicitly removed.

---

**Selection Operation**

- The *selection operation* $\sigma$ (sigma) is a unary operation that takes in a relation as input and returns a new relation as output that contains a subset of the tuples of the input relation.
  - That is, the output relation has the same number of columns as the input relation, but may have less rows.

  - To determine which tuples are in the output, the selection operation has a specified condition, called a *predicate*, that tuples must satisfy to be in the output.
  - The predicate is similar to a condition in an *if* statement.

  - Selection $\sigma_\theta R$ takes a relation $R$ and extracts those rows from it that satisfy the condition $\theta$
Selection Operation Formal Definition

- The selection operation on relation \( R \) with predicate \( F \) is denoted by \( \sigma_F(R) \).

\[
\sigma_F(R) = \{ t \mid t \in R \text{ and } F(t) \text{ is true} \}
\]

where

- \( R \) is a relation, \( t \) is a tuple variable
- \( F \) is a formula (predicate) consisting of
  - operands that are constants or attributes
  - comparison operators: \( <, >, =, \neq, \leq, \geq \)
  - logical operators: \( \text{AND}, \text{OR}, \text{NOT} \)

Example Selection, \( \sigma_9 \)

- Example: find tuples where \( \text{sid}=3 \) has exp grade of A

<table>
<thead>
<tr>
<th>sid</th>
<th>exp-grade</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>A</td>
<td>550-0103</td>
</tr>
<tr>
<td>1</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>C</td>
<td>500-0103</td>
</tr>
<tr>
<td>4</td>
<td>C</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

Can the result have duplicates?

Complex Predicate Conditions

- Conditions are built up from boolean-valued operations on the field names.
  - \( \text{exp-grade}>"A", \text{name}="Jill", \text{STUDENT.sid}=\text{Takes.sid} \)
  - RA allows comparison predicate on attributes
    - \( =, \neq, >, <, \geq, \leq \)
    - Larger predicates can be formed using logical connectives – \( \text{or} (\lor) \) and \( \text{and} (\land) \) and \( \text{not} (\neg) \)
    - Selection predicate can include comparison between attributes
    - We don’t lose any expressive power if we don’t have complex predicates in the language, but they are convenient and useful in practice.

Selection Example

<table>
<thead>
<tr>
<th>eno</th>
<th>ename</th>
<th>title</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>J. Doe</td>
<td>EE</td>
<td>30000</td>
</tr>
<tr>
<td>E2</td>
<td>M. Smith</td>
<td>SA</td>
<td>50000</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
<td>ME</td>
<td>40000</td>
</tr>
<tr>
<td>E4</td>
<td>J. Miller</td>
<td>PR</td>
<td>20000</td>
</tr>
<tr>
<td>E5</td>
<td>B. Casey</td>
<td>SA</td>
<td>50000</td>
</tr>
<tr>
<td>E6</td>
<td>L. Chu</td>
<td>EE</td>
<td>30000</td>
</tr>
<tr>
<td>E7</td>
<td>R. Davis</td>
<td>ME</td>
<td>40000</td>
</tr>
<tr>
<td>E8</td>
<td>J. Jones</td>
<td>SA</td>
<td>50000</td>
</tr>
</tbody>
</table>

\( \sigma_{\text{title}=\text{EE}}(\text{Emp}) \)

\( \sigma_{\text{salary}>35000 \text{ OR title}=\text{PR}}(\text{Emp}) \)

<table>
<thead>
<tr>
<th>eno</th>
<th>ename</th>
<th>title</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>J. Doe</td>
<td>EE</td>
<td>30000</td>
</tr>
<tr>
<td>E6</td>
<td>L. Chu</td>
<td>EE</td>
<td>30000</td>
</tr>
</tbody>
</table>

\( \sigma_{\text{salary}>35000 \text{ OR title}=\text{PR}}(\text{Emp}) \)

<table>
<thead>
<tr>
<th>eno</th>
<th>ename</th>
<th>title</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E2</td>
<td>M. Smith</td>
<td>SA</td>
<td>50000</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
<td>ME</td>
<td>40000</td>
</tr>
<tr>
<td>E4</td>
<td>J. Miller</td>
<td>PR</td>
<td>20000</td>
</tr>
<tr>
<td>E5</td>
<td>B. Casey</td>
<td>SA</td>
<td>50000</td>
</tr>
<tr>
<td>E7</td>
<td>R. Davis</td>
<td>ME</td>
<td>40000</td>
</tr>
<tr>
<td>E8</td>
<td>J. Jones</td>
<td>SA</td>
<td>50000</td>
</tr>
</tbody>
</table>
Union

- Union $\cup$ is a binary operation that takes two relations $R$ and $S$ as input and produces an output relation that includes all tuples that are either in $R$ or in $S$ or in both $R$ and $S$. Duplicate tuples are eliminated.

- General form:
  \[ R \cup S = \{ t \mid t \in R \text{ or } t \in S \} \]
  where $R$, $S$ are relations, $t$ is a tuple variable. $R$ and $S$ must be union-compatible. To be union-compatible means:
  1) Both relations have same number of attributes.
  2) Each attribute pair, $R_i$ and $S_i$, have compatible data types for all attribute indexes $i$.
  - Note that attributes do not need to have the same name.
  - Result has attribute names of first relation.

Example Union $\cup$

- Find persons who are faculty or students

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Darby</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
</tr>
<tr>
<td>3</td>
<td>Dan</td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>fid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>12</td>
<td>Youssef</td>
</tr>
<tr>
<td>18</td>
<td>Choi</td>
</tr>
</tbody>
</table>

Union Example

<table>
<thead>
<tr>
<th>emp</th>
<th>ename</th>
<th>title</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>J. Doe</td>
<td>EE</td>
<td>30000</td>
</tr>
<tr>
<td>E2</td>
<td>M. Smith</td>
<td>SA</td>
<td>20000</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
<td>ME</td>
<td>40000</td>
</tr>
<tr>
<td>E4</td>
<td>J. Miller</td>
<td>PF</td>
<td>20000</td>
</tr>
<tr>
<td>E5</td>
<td>B. Casey</td>
<td>SA</td>
<td>30000</td>
</tr>
<tr>
<td>E6</td>
<td>J. Chu</td>
<td>EE</td>
<td>30000</td>
</tr>
<tr>
<td>E7</td>
<td>R. Davis</td>
<td>ME</td>
<td>40000</td>
</tr>
<tr>
<td>E8</td>
<td>J. Jones</td>
<td>SA</td>
<td>50000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>emp</th>
<th>ename</th>
<th>title</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>J. Doe</td>
<td>EE</td>
<td>30000</td>
</tr>
<tr>
<td>E2</td>
<td>M. Smith</td>
<td>SA</td>
<td>20000</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
<td>ME</td>
<td>40000</td>
</tr>
<tr>
<td>E4</td>
<td>J. Miller</td>
<td>PF</td>
<td>20000</td>
</tr>
<tr>
<td>E5</td>
<td>B. Casey</td>
<td>SA</td>
<td>30000</td>
</tr>
<tr>
<td>E6</td>
<td>J. Chu</td>
<td>EE</td>
<td>30000</td>
</tr>
<tr>
<td>E7</td>
<td>R. Davis</td>
<td>ME</td>
<td>40000</td>
</tr>
<tr>
<td>E8</td>
<td>J. Jones</td>
<td>SA</td>
<td>50000</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>eno</th>
<th>pno</th>
<th>resp</th>
<th>dur</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>P1</td>
<td>Manager</td>
<td>12</td>
</tr>
<tr>
<td>E2</td>
<td>P1</td>
<td>Analyst</td>
<td>24</td>
</tr>
<tr>
<td>E3</td>
<td>P2</td>
<td>Analyst</td>
<td>24</td>
</tr>
<tr>
<td>E4</td>
<td>P4</td>
<td>Engineer</td>
<td>48</td>
</tr>
<tr>
<td>E5</td>
<td>P2</td>
<td>Manager</td>
<td>24</td>
</tr>
<tr>
<td>E6</td>
<td>P4</td>
<td>Manager</td>
<td>48</td>
</tr>
<tr>
<td>E7</td>
<td>P5</td>
<td>Engineer</td>
<td>23</td>
</tr>
</tbody>
</table>

Type Matching for Set operations

- Same number of attributes
- Same type of attributes
  - Each position must match domain
    - Real systems sometimes allow sub-types: CHAR(2) and CHAR(20)
Set Difference

- **Set difference** is a binary operation that takes two relations $R$ and $S$ as input and produces an output relation that contains all the tuples of $R$ that are not in $S$.

- General form:
  - $R - S = \{ t \mid t \in R \text{ and } t \notin S \}$
  - where $R$ and $S$ are relations, $t$ is a tuple variable.

- Note that:
  - $R - S \neq S - R$
  - $R$ and $S$ must be union compatible.

Set Difference Example

<table>
<thead>
<tr>
<th>Emp Relation</th>
<th>WorksOn Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>eno</td>
<td>ename</td>
</tr>
<tr>
<td>E1</td>
<td>J. Doe</td>
</tr>
<tr>
<td>E2</td>
<td>M. Smith</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
</tr>
<tr>
<td>E4</td>
<td>J. Miller</td>
</tr>
<tr>
<td>E5</td>
<td>B. Casey</td>
</tr>
<tr>
<td>E6</td>
<td>L. Chu</td>
</tr>
<tr>
<td>E7</td>
<td>R. Davis</td>
</tr>
<tr>
<td>E8</td>
<td>J. Jones</td>
</tr>
<tr>
<td>eno</td>
<td>pno</td>
</tr>
<tr>
<td>E1</td>
<td>P1</td>
</tr>
<tr>
<td>E2</td>
<td>P1</td>
</tr>
<tr>
<td>E2</td>
<td>P2</td>
</tr>
<tr>
<td>E3</td>
<td>P4</td>
</tr>
<tr>
<td>E5</td>
<td>P2</td>
</tr>
<tr>
<td>E6</td>
<td>P4</td>
</tr>
<tr>
<td>E7</td>
<td>P3</td>
</tr>
<tr>
<td>E7</td>
<td>P5</td>
</tr>
</tbody>
</table>

$$\Pi_{\text{eno}}(\text{Emp}) - \Pi_{\text{eno}}(\text{WorksOn})$$

**Question 1:** What is the meaning of this query?

**Question 2:** What is $\Pi_{\text{eno}}(\text{WorksOn}) - \Pi_{\text{eno}}(\text{Emp})$?

Difference –

- Another set operator. Example: Find persons who are students but not a faculty

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>FACULTY</th>
<th>STUDENT – FACULTY</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
<td></td>
</tr>
<tr>
<td>1</td>
<td>Darby</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td></td>
</tr>
<tr>
<td>2</td>
<td>Youssel</td>
<td></td>
</tr>
<tr>
<td>3</td>
<td>Dan</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Choi</td>
<td></td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
<td></td>
</tr>
</tbody>
</table>

Intersection??

- People who are both Students and Faculty??

- Do we need an Intersection operator?
Deriving Intersection

Intersection: as with set operations, derivable from difference

\[ A \cap B \equiv (A - (A-B)) = (B - (B-A)) \]

Aside: Division Operator

- There is also a division operator used when you want to determine if all combinations of a relationship are present.
  - E.g. Return the list of employees who work on all the projects that 'John Smith' works on.

- The division operator is not a base operator and is not frequently used, so we will not spend any time on it.
  - Note that \( R \div S = \Pi_{R, \delta}(R) - \Pi_{S, \delta}(\Pi_{R, \delta}(R) \times S) - R \).

Operators that ‘combine’ relations

- Thus far, only operated on a single relation
- How to connect two relations?
  - To find name of students taking a specific course with cid, we need to look at both students and enrolled tables

- Operator(s) that produce a relation (set of tuples) after combining two different relations

- Set theory provides us with the cartesian product operator (between two sets: but can be applied to product of any number of sets – to get a k-tuple)

Cartesian Product

- The Cartesian product of two relations \( R \) (of degree \( k_1 \)) and \( S \) (of degree \( k_2 \)) is:

\[ R \times S = \{ t \mid t[A_1, \ldots, A_{k_1}] \in R \text{ and } t[A_{k_1+1}, \ldots, A_{k_1+k_2}] \in S \} \]

- The result of \( R \times S \) is a relation of degree \( (k_1 + k_2) \) and consists of all \((k_1 + k_2)\)-tuples where each tuple is a concatenation of one tuple of \( R \) with one tuple of \( S \).

- The cardinality of \( R \times S \) is \(|R| \times |S|\).

- The Cartesian product is also known as cross product.
Basic Product/Join Operation

- The product \( \times \) operation allows combination of info from two tables – it is the set product
- \( R_1 \times R_2 \) is collection of tuples from cross product of the two relations
- If \( R_1 \) has \( k \) columns and \( R_2 \) has \( n \) columns then \( R_1 \times R_2 \) has \( k \cdot n \) columns
  - Resulting schema is concatenation of two schemas
  - Refer to attribute \( B_i \) of relation \( R_i \) as \( R_i.B_i \)

Product X Example

- “Join” is a generic term for a variety of operations that connect two relations. The basic operation is the cartesian product, \( R \times S \), which concatenates every tuple in \( R \) with every tuple in \( S \). Example:

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>school</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jill</td>
<td>UPenn</td>
</tr>
<tr>
<td>2</td>
<td>Matt</td>
<td>GWU</td>
</tr>
</tbody>
</table>

What if the attribute of SCHOOL was called “name”?

Cartesian Product Example

<table>
<thead>
<tr>
<th>Emp Relation</th>
<th>Proj Relation</th>
</tr>
</thead>
<tbody>
<tr>
<td>eno</td>
<td>ename</td>
</tr>
<tr>
<td>1</td>
<td>J. Doe</td>
</tr>
<tr>
<td>E2</td>
<td>M. Smith</td>
</tr>
<tr>
<td>E3</td>
<td>A. Lee</td>
</tr>
<tr>
<td>E4</td>
<td>J. Miller</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>pno</th>
<th>pname</th>
<th>budget</th>
</tr>
</thead>
<tbody>
<tr>
<td>P1</td>
<td>Instruments</td>
<td>150000</td>
</tr>
<tr>
<td>P2</td>
<td>DB Develop</td>
<td>135000</td>
</tr>
<tr>
<td>P3</td>
<td>CAD/CAM</td>
<td>250000</td>
</tr>
</tbody>
</table>

Product/Join

- Tuple in \( R_1 \times R_2 \) constructed by associating a tuple from \( R_1 \) with every tuple in \( R_2 \)
- If \( R_1 \) has \( n_1 \) tuples and \( R_2 \) has \( n_2 \) tuples how many does \( R_1 \times R_2 \) have?
- Same attribute can appear in both tables?
  - “link” between the two tables
\( \Theta \) -Join

- Theta (\( \Theta \)) join is a derivative of the Cartesian product. Instead of taking all combinations of tuples from \( R \) and \( S \), we only take a subset of those tuples that match a given condition \( F \).

\[ R \bowtie_F S = \{ t \mid t[A_1, \ldots, A_k] \in R \text{ and } t[A_{k+1}, \ldots, A_{k+2}] \in S \text{ and } F(t) \text{ is true} \} \]

- Where
  - \( R, S \) are relations, \( t \) is a tuple variable
  - \( F(t) \) is a formula defined as that of selection.

- Note that \( R \bowtie_F S = \sigma_F(R \times S) \).

(Theta) Join, \( \bowtie_\theta \): A Combination of Product and Selection

- Products are hardly ever used alone; they are typically used in conjunction with a selection.
- Example: Find students (id and name) and courses they took with grades and cid

\[
\sigma_{\text{STUDENT.sid=Takes.sid}} (\text{STUDENT} \bowtie \text{STUDENT.sid=Takes.sid} \text{Takes})
\]

<table>
<thead>
<tr>
<th>sid:1</th>
<th>name</th>
<th>sid:2</th>
<th>exp-grade</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jill</td>
<td>1</td>
<td>A</td>
<td>550-0103</td>
</tr>
<tr>
<td>1</td>
<td>Jill</td>
<td>1</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Alex</td>
<td>3</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Alex</td>
<td>3</td>
<td>C</td>
<td>500-0103</td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
<td>4</td>
<td>C</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

Joins

- Example: Find students (id and name) and courses they took with grades and cid

\[
\sigma_{\text{STUDENT.sid=Takes.sid}} (\text{STUDENT} \bowtie \text{STUDENT.sid=Takes.sid} \text{Takes})
\]

<table>
<thead>
<tr>
<th>sid:1</th>
<th>name</th>
<th>sid:2</th>
<th>exp-grade</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jill</td>
<td>1</td>
<td>A</td>
<td>550-0103</td>
</tr>
<tr>
<td>1</td>
<td>Jill</td>
<td>1</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Alex</td>
<td>3</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Alex</td>
<td>3</td>
<td>C</td>
<td>500-0103</td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
<td>4</td>
<td>C</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

Join condition

\( sid:1 \) and \( sid:2 \) are duplicate information.... Do we need two columns? Why not project only one of them?
“Natural” Join, ⊙

- The most common join to do is an equality join of two relations on commonly named fields, and to leave one copy of those fields in the resulting relation. Example:
  
  \[
  \text{STUDENT} \bowtie \text{Takes} = \rho_{\text{sid}:1 \rightarrow \text{sid}}(\pi_{\text{sid}, \text{name}, \text{exp-grade}, \text{cid}}(\text{STUDENT} \bowtie \text{STUDENT}.\text{sid} = \text{Takes}.\text{sid}))
  \]

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>exp-grade</th>
<th>cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Jill</td>
<td>A</td>
<td>550-0103</td>
</tr>
<tr>
<td>1</td>
<td>Jill</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Nick</td>
<td>A</td>
<td>700-1003</td>
</tr>
<tr>
<td>3</td>
<td>Nick</td>
<td>C</td>
<td>500-0103</td>
</tr>
<tr>
<td>4</td>
<td>Sina</td>
<td>F</td>
<td>500-0103</td>
</tr>
</tbody>
</table>

What if all the field names are the same in the two relations? What if the field names are all disjoint?

Types of Joins

- The \( \bowtie \)-Join is a general join in that it allows any expression in the condition \( F \). However, there are more specialized joins that are frequently used.

  - A **equijoin** only contains the equality operator (=) in formula \( F \).
    - e.g. WorksOn \( \bowtie \) WorksOn.\text{pno} = Proj.\text{pno} Proj

  - A **natural join** over two relations \( R \) and \( S \) denoted by \( R \bowtie S \) is the equijoin of \( R \) and \( S \) over a set of attributes common to both \( R \) and \( S \).
    - It removes the “extra copies” of the join attributes.
    - The attributes must have the same name in both relations.

Equijoin Example

<table>
<thead>
<tr>
<th>\text{eno}</th>
<th>\text{pno}</th>
<th>\text{resp}</th>
<th>\text{dur}</th>
<th>\text{proj}</th>
<th>\text{pno}</th>
<th>\text{pmame}</th>
<th>\text{budget}</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>P1</td>
<td>Manager</td>
<td>12</td>
<td></td>
<td>P1</td>
<td>Instruments</td>
<td>150000</td>
</tr>
<tr>
<td>E2</td>
<td>P1</td>
<td>Analyst</td>
<td>24</td>
<td></td>
<td>P1</td>
<td>Instruments</td>
<td>150000</td>
</tr>
<tr>
<td>E3</td>
<td>P1</td>
<td>Analyst</td>
<td>24</td>
<td></td>
<td>P1</td>
<td>Instruments</td>
<td>150000</td>
</tr>
<tr>
<td>E4</td>
<td>P2</td>
<td>Engineer</td>
<td>48</td>
<td></td>
<td>P3</td>
<td>CAD/CAM</td>
<td>250000</td>
</tr>
<tr>
<td>E5</td>
<td>P3</td>
<td>Engineer</td>
<td>36</td>
<td></td>
<td>P4</td>
<td>Maintenance</td>
<td>310000</td>
</tr>
<tr>
<td>E6</td>
<td>P3</td>
<td>Engineer</td>
<td>36</td>
<td></td>
<td>P4</td>
<td>Maintenance</td>
<td>310000</td>
</tr>
<tr>
<td>E7</td>
<td>P4</td>
<td>Engineer</td>
<td>23</td>
<td></td>
<td>P3</td>
<td>CAD/CAM</td>
<td>250000</td>
</tr>
</tbody>
</table>

What is the meaning of this join?

Natural join Example

<table>
<thead>
<tr>
<th>\text{eno}</th>
<th>\text{pno}</th>
<th>\text{resp}</th>
<th>\text{dur}</th>
<th>\text{pmame}</th>
<th>\text{budget}</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>P1</td>
<td>Manager</td>
<td>12</td>
<td>Instruments</td>
<td>150000</td>
</tr>
<tr>
<td>E2</td>
<td>P1</td>
<td>Analyst</td>
<td>24</td>
<td>DB Develop</td>
<td>135000</td>
</tr>
<tr>
<td>E3</td>
<td>P2</td>
<td>Engineer</td>
<td>36</td>
<td>Maintenance</td>
<td>310000</td>
</tr>
<tr>
<td>E4</td>
<td>P3</td>
<td>Engineer</td>
<td>36</td>
<td>CAD/CAM</td>
<td>250000</td>
</tr>
<tr>
<td>E5</td>
<td>P4</td>
<td>Engineer</td>
<td>23</td>
<td>Maintenance</td>
<td>310000</td>
</tr>
</tbody>
</table>

What is the meaning of this join?

Natural join is performed by comparing \text{pno} in both relations.
Outer Joins

- Outer joins are used in cases where performing a join "loses" some tuples of the relations. These are called dangling tuples.

- There are three types of outer joins:
  - 1) **Left outer join** - \( R \bowtie S \) - The output contains all tuples of \( R \) that match with tuples of \( S \). If there is a tuple in \( R \) that matches with no tuple in \( S \), the tuple is included in the final result and is padded with nulls for the attributes of \( S \).
  - 2) **Right outer join** - \( R \bowtie S \) - The output contains all tuples of \( S \) that match with tuples of \( R \). If there is a tuple in \( S \) that matches with no tuple in \( R \), the tuple is included in the final result and is padded with nulls for the attributes of \( R \).
  - 3) **Full outer join** - \( R \bowtie S \) - All tuples of \( R \) and \( S \) are included in the result whether or not they have a matching tuple in the other relation.

Semi-Join and Anti-Join

- A **semi-join** between tables returns rows from the first table where one or more matches are found in the second table.
  - Semi-joins are used in EXISTS and IN constructs in SQL.

- An **anti-join** between two tables returns rows from the first table where no matches are found in the second table.
  - Anti-joins are used with NOT EXISTS, NOT IN, and FOR ALL.
  - Anti-join is the complement of semi-join: \( R \bowtie S = R - R \bowtie S \)
Anti-Join Example

<table>
<thead>
<tr>
<th>eno</th>
<th>pno</th>
<th>resp</th>
<th>dur</th>
</tr>
</thead>
<tbody>
<tr>
<td>E1</td>
<td>P1</td>
<td>Manager</td>
<td>12</td>
</tr>
<tr>
<td>E2</td>
<td>P1</td>
<td>Analyst</td>
<td>24</td>
</tr>
<tr>
<td>E3</td>
<td>P4</td>
<td>Engineer</td>
<td>6</td>
</tr>
<tr>
<td>E4</td>
<td>P2</td>
<td>Analyst</td>
<td>24</td>
</tr>
<tr>
<td>E5</td>
<td>P4</td>
<td>Manager</td>
<td>48</td>
</tr>
<tr>
<td>E6</td>
<td>P3</td>
<td>Engineer</td>
<td>36</td>
</tr>
<tr>
<td>E7</td>
<td>P4</td>
<td>Engineer</td>
<td>23</td>
</tr>
</tbody>
</table>

Combining Operations

- Relational algebra operations can be combined in one expression by nesting them:
  \[ \Pi_{eno,pno,dur} (\sigma_{ename='J. Doe'} (Emp) \bowtie \sigma_{dur>16} (WorksOn)) \]
  - Return the eno, pno, and duration for employee ‘J. Doe’ when he has worked on a project for more than 16 months.
- Operations also can be combined by using temporary relation variables to hold intermediate results.
  - We will use the assignment operator \( \leftarrow \) for indicating that the result of an operation is assigned to a temporary relation.

\[
\begin{align*}
\text{empdoc} &\leftarrow \sigma_{ename='J. Doe'} (Emp) \\
\text{wodur} &\leftarrow \sigma_{dur>16} (WorksOn) \\
\text{empwo} &\leftarrow \text{empdoc} \bowtie \text{wodur} \\
\text{result} &\leftarrow \Pi_{eno,pno,dur} (\text{empwo})
\end{align*}
\]

Rename, \( \rho_{\alpha} (R) \)

- The rename operator can be expressed several ways:
  - The book has a very odd definition that’s not algebraic BUT is more readable!
  - An alternate definition:
    \[ \rho_{\alpha}(x) \]
    Takes the relation x and returns a copy of the relation with the name \( \alpha \).
    General Def: can rename only attribute list with new names \( \beta \)
  - Rename isn’t all that useful, except if you join a relation with itself
  - \( \rho_{Person} (STUDENT) = \) copy of STUDENT with table name Person
  - Find pairs of student IDs who have the same name:
    \[ \Pi_{\text{STUDENT.sid, Person.sid}} (\text{STUDENT} \bowtie \text{STUDENT.name=}Person.name \ \ \ (\rho_{Person} (\text{STUDENT})) \]

Rename Operator

- Variations allow renaming of specific attributes
  - \( \rho_{X(C,D)} (R (A,B)) \)
    Relation R renamed to X
    Fields A,B in R are now renamed to C,D in X
  - \( \rho_{Person(id, who)} (\text{STUDENT (sid, name)}) \)
  - Find pairs of student IDs who have the same name:?
**Rename Operation**

- Renaming can be applied when assigning a result:

\[
\text{result(EmployeeNum, ProjectNum, Duration)} \leftarrow \Pi_{\text{emp}, \text{pno}, \text{dur}}(\text{empwo})
\]

- Or by using the rename operator \( \rho \) (rho):

\[
\rho \text{result(EmployeeName, ProjectNum, Duration)}(\text{empwo})
\]

**Complete Set of Relational Algebra Operators**

- It has been shown that the relational operators \{\( \sigma \), \( \Pi \), \( \times \), \( \cup \), \( \setminus \)\} form a complete set of operators.
- That is, any of the other operators can be derived from a combination of these 5 basic operators.

- Examples:
  - Intersection - \( R \cap S = R \cup S \setminus ((R \setminus S) \cup (S \setminus R)) \)
  - We have also seen how a join is a combination of a Cartesian product followed by a selection.

**Other Relational Algebra Operators**

- There are other relational algebra operators that we will not discuss. Most notably, we often need **aggregate operations** that compute functions on the data.

- For example, given the current operators, we cannot answer the query:
  - What is the total amount of deposits at the Kelowna branch?
  - What are the total number of employees in department 5?

- We will see how to answer these queries when we study SQL.

**How to write a RA query?**

- Find out which tables you need to access
  - Compute \( \times \) of these tables
- What are the conditions/predicates you need to apply?
  - Determines what select \( \sigma \) operators you need to apply
- What attributes/columns are needed in result
  - Determines what project \( \Pi \) operators you need

\[
\text{Project ( Select ( Product))}
\]
More examples…

- This completes the basic operations of the relational algebra.
  - These form a complete set of operations
  - Additional operators provided for better writability
- Try writing queries for these:
  - The IDs of students named “Bob”
  - The names of students expecting an “A”
  - The names of students in Youssef’s class
  - The sids and names of students not enrolled in any class

Examples

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>Takes</th>
<th>COURSE</th>
</tr>
</thead>
<tbody>
<tr>
<td>sid</td>
<td>name</td>
<td>sid</td>
</tr>
<tr>
<td>1</td>
<td>Jill</td>
<td>1</td>
</tr>
<tr>
<td>2</td>
<td>Bob</td>
<td>1</td>
</tr>
<tr>
<td>3</td>
<td>Sina</td>
<td>3</td>
</tr>
<tr>
<td>4</td>
<td>Maury</td>
<td>3</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PROFESSOR</th>
<th>Teaches</th>
</tr>
</thead>
<tbody>
<tr>
<td>fid</td>
<td>name</td>
</tr>
<tr>
<td>1</td>
<td>Narahari</td>
</tr>
<tr>
<td>2</td>
<td>Youssef</td>
</tr>
<tr>
<td>8</td>
<td>Stanton</td>
</tr>
</tbody>
</table>

Modifying the Database

- Need to insert, delete, update tuples in the database
- What is insert?
  - Add a new tuple to existing set = Union
- What is delete?
  - Remove a tuple from existing set = Set difference
- How to update attribute to new value?
  - Need new operator: δ

Another Example: Bank Database

- Simplified Bank database
  - Bank has a number of branches at different locations
  - Bank handles customer accounts and loans
- Four relations/tables
  - Customer: stores info about customer
  - Deposit: stores accounts at the branch/bank
  - Loan: stores loans given by bank/branch
  - Branch: stores info about the branch
Schema of Bank DB

- **Customer** (CustID, Name, street, city, zip)
  - Customer ID, Name, and Address info: street, city, zip

- **Deposit** (CustID, Acct-num, balance, Branch-name)
  - Customer ID, Account number, Balance in account, name of branch where account is held

- **Loan** (CustID, Loan-num, Amount, Branch-name)
  - Customer ID, loan number, amount of loan…

- **Branch** (Branch-name, assets, Branch-city)

Modifying Database

- Delete all accounts of Customer with CustID=3333
  - Deposit ← Deposit – (tuples of CustID 3333)

- Insert tuple (4444, Downtown, 1000, 1234)
  - Deposit ← Deposit  (4444, Downtown, 1000, 1234)

- Update: δ_{A \leftarrow E}(R)
  - Update attribute A to E for tuples in relation R
  - δ_{balance \leftarrow 1.05*balance}(Deposit) : updates balances
  - Can also specify selection condition on Deposit
    Update balances only for customers with CustID=1234

Views: Important Concept

- Relational Model, using SQL, allows definition of a view
  - View is a virtual relation
  - Executed each time it is referenced
  - More when we get to SQL

Relational Calculus
Next . . .

- Quick look at Relational Calculus
  - Tuple calculus
  - Domain calculus

- Non-procedural language

- Relational algebra gave you a procedure:
  - Which relations to access
  - What selection/predicate conditions to apply
  - Etc…

Relational Calculus

- Comes in two flavors: Tuple relational calculus (TRC) and Domain relational calculus (DRC).
- Calculus has variables, constants, comparison ops, logical connectives and quantifiers.
  - TRC: Variables range over (i.e., get bound to) tuples.
  - DRC: Variables range over domain elements (= field values).
- Both TRC and DRC are simple subsets of first-order logic.
- Expressions in the calculus are called formulas. An answer tuple is essentially an assignment of values to variables that make the formula evaluate to true.

Relational Calculus: An Equivalent, But Very Different, Formalism

- Codd invented a relational calculus that he proved was equivalent in expressiveness
  - Based on a subset of first-order logic — declarative, without an implicit order of evaluation
    Tuple relational calculus
    Domain relational calculus
  - More convenient for describing certain things, and for certain kinds of manipulations
- The database uses the relational algebra internally, but query languages (e.g., SQL) use concepts from the relational calculus

Tuple Relational Calculus (RC)

- A tuple variable is a variable whose values can be tuples of a relational schema

- Formula/Query in RC is expressed as:
  \( \{ t \mid P(t) \} \)
  - \( t \) is a tuple variable
  - \( P(t) \) is property of tuple \( t \); it is a predicate formula that describes properties of the tuple variable
  - Thereby defining the possible values of \( t \)
  - Result is set of all tuples for which predicate \( P \) is true
- Find all loans at the Downtown branch
- "syntax" is set theory notations

\{ t | t \in \text{Loan} \land t[\text{branch-name}] = "Downtown" \}

Tuple Relational Calculus (RC) – alternate syntax
- A tuple variable is a variable whose values can be tuples of a relational schema
- Formula/Query in RC is expressed as:
  \{ \langle t[\text{att1}], t[\text{att2}], ... \rangle | R(t) \land P(t) \}
  - t is a tuple variable in relation R
  - t[\text{att1}] is value of attribute 1 in tuple t
  - P(t) is property of tuple t; it is a predicate formula that describes properties of the tuple variable
    - Therefore defining the possible values of t
  - Result is set of all tuples for which predicate P is true

Domain Relational Calculus (DRC)
- Queries have form:
  \{ \langle x_1, x_2, ..., x_n \rangle | p \} 
  - Predicate: boolean expression over \( x_1, x_2, ..., x_n \)
  - Answer includes all tuples that make the formula true.
  - The variables come from the domain of the attributes in the relation schema
    - In contrast to the tuple calculus where variables are tuples

Note: We will use set theory notations (to continue with what you learnt in CS1311!)
Domain Relational Calculus
- Define domain of each attribute in result set and the type
- Find branch-name, loan number, amount, custID for loans over $1200

\{ <b,l,c,a> | <b,l,c,a> \in \text{loan} \land b='Downtown' \}

RC Formulas
- Atomic formula:
  \( t \in \text{Rname} \), or \( X \ \text{op} \ Y \), or \( X \ \text{op} \ \text{constant} \)
  \( \text{op} \) is one of \( <,>,=,\geq,\leq,\neq \)

- Formula: an atomic formula, or
  \( \neg p, p \land q, p \lor q \), where \( p \) and \( q \) are formulas, or
  \( \exists X(p(X)) \), where variable \( X \) is free in \( p(X) \), or
  \( \forall X(p(X)) \) where variable \( X \) is free in \( p(X) \)

- The use of quantifiers \( \exists X \) and \( \forall X \) is said to bind \( X \).
  - A variable that is not bound is free.

Expressions and Formulas in RC
- Truth value of an atomic formula (atom) evaluates to either TRUE or FALSE
- Formula is made up of one or more atoms connected via logical operations AND, OR, NOT...

Existential and Universal Quantifiers
- Two special symbols can appear in formulas:
  - \( \forall t : \) universal quantifier
  - \( \exists t : \) existential quantifier
- Informally: a tuple is bound if it is quantified- it appears in an universal or existential clause, otherwise it is free
- If \( F \) is a formula, then so are \( (\exists t)(F) \) and \( (\forall t)(F) \)
  - The formula \( (\exists t)(F) \) is true if the formula \( F \) evaluates to true for some (at least one) tuple assigned to free occurences of \( t \) in \( F \); otherwise it is false.
  - The formula \( (\forall t)(F) \) is true if the formula \( F \) evaluates to true for every tuple (in the universe) assigned to free occurrences of \( t \) in \( F \); otherwise it is false.
  - \( \forall \) is called universal “for all” quantifier because every tuple in the universe of tuples must make \( F \) true to make the formula true
  - \( \exists \) is called existential or “there exists” quantifier because any tuple that exists may make \( F \) true to make formula true
More Complex Predicates in Relational Calculus

Starting with these atomic predicates, build up new predicates by the following rules:

- Logical connectives: If \( p \) and \( q \) are predicates, then so are
  \[ p \land q, \quad p \lor q, \quad \neg p, \quad \text{and} \quad p \Rightarrow q \]
  - \((x>2) \land (x<4) = ?\) (True or false)
  - \((x>2) \land \neg(x<0) = ?\)

- Existential quantification: If \( p \) is a predicate, then so is \( \exists x. p \)
  \[ \exists x. (x>2) \land (x<4) = ? \]

- Universal quantification: If \( p \) is a predicate, then so is \( \forall x. p \)
  \[ \forall x. (x>2) = ? \quad \forall x. \exists y. (y>x) = ? \]

Logical Equivalences

- Recall from discrete math cs1311
- There are two logical equivalences that are heavily used:
  - \( p \Rightarrow q \equiv \neg p \lor q \)
    (Whenever \( p \) is true, \( q \) must also be true.)
  - \( \forall x. p(x) \equiv \neg \exists x. \neg p(x) \)
    (\( p \) is true for all \( x \))
- The second can be a lot easier to check!
- Example:
  - The highest course number offered

Free and Bound Variables

- The use of quantifiers \( \forall \) or \( \exists \) in a formula is said to bind the variables
- A variable \( v \) is bound in a predicate \( p \) when \( p \) is of the form \( \forall v \ldots \) or \( \exists v \ldots \)
- A variable occurs free in \( p \) if it occurs in a position where it is not bound by an enclosing \( \forall \) or \( \exists \)
- Examples:
  - \( x \) is free in \( x > 2 \)
  - \( y \) is free and \( x \) is bound in \( \exists x. x > y \)
- Important restriction: the variables that appear to the left of \( "|" \) must be the only free variables in the formula \( p(...). \)

Can Rename Bound Variables Only

- When a variable is bound one can replace it with some other variable without altering the meaning of the expression, providing there are no name clashes
  - Example: \( \exists x. x > 2 \) is equivalent to \( \exists y. y > 2 \)
- Otherwise, the variable is defined outside our “scope”…
Free Variables

- A variable $v$ is bound in a predicate $p$ when $p$ is of the form $\forall v \ldots$ or $\exists v \ldots$
- Important restriction: the variables that appear to the left of `|` must be the only free variables in the formula $p(...)$.
- Implication: the values that the free variables can legally take on are the results of the query!

Safety of Operators

- Query of the form $\exists t \in R \ (Q(t))$
  - There exists tuple $t$ in set/relation $R$ such that predicate $Q$ is true
- Safety of Expressions
  - What about $\{ t \mid \exists t (t \in \text{loan}) \}$
    - Infinitely many tuples outside loan relation

Safety of Expressions

- A query is safe if no matter how we instantiate the relations, it always produces a finite answer
  - Domain independent: answer is the same regardless of the domain in which it is evaluated
  - Unfortunately, both this definition of safety and domain independence are semantic conditions, and are undecidable
- There are syntactic conditions that are used to guarantee “safe” formulas
  - One solution: For each tuple relational formula $P$, define domain $\text{Dom}(P)$ which is set of all values referenced by $P$
  - The formulas that are expressible in real query languages based on relational calculus are all “safe”
- Many DB languages include additional features, like recursion, that must be restricted in certain ways to guarantee termination and consistent answers

Safety and Termination Guarantees

- There are syntactic conditions that are used to guarantee “safe” formulas
  - The definition is complicated, and we won’t discuss it; you can find it in Ullman’s Principles of Database and Knowledge-Base Systems
  - The formulas that are expressible in real query languages based on relational calculus are all “safe”
Examples: Relational Calculus
- Find branch name, ID, loan number and amount for loans over $1200
  - What is the "type" of the elements in the result, i.e., where do they come from?
  - What is the property of the elements?
- \{ t | (t ∈ Loan) ∧ (t[amount]>1200) \}
  - Type of tuple t is Loan (since it is an element of Loan)
  - Property is that value of amount attribute in the tuple must be greater than 1200

Domain Relational Calculus
- Define domain of each attribute in result set and the type
- Find branch-name, loan number, amount, custID for loans over $1200
  - \{ <b,l,c,a> | <b,l,c,a> ∈ loan ∧ a>1200 \}
  - Domain of each attr in result is defined by <b,l,c,a> is an element in loan

Relational calculus – projections
- Two approaches depending on use of set notation or not...
- What if the type has to be inferred?
- Find only Customer ID attribute in the previous example
  - This type has to be inferred by the query
  - Tuples on ID, for which there is a tuple in Loan with same CustID and amount > 1200.
- \{ t | ∃ s ∈ Loan ( s[custID]=t[CustID] ∧ s[amount]>1200) \}
  - 'schema' of t can be deduced, from query, as containing an attribute CustID
    - No other attribute is defined for t
    - Therefore the 'type' of t is [CustID] (a single attribute)
  - Note use of existential quantifier...
    - s is bound variable
    - t is free variable….result of query is values that free variable can take to make the predicate true

Using named field notation…easier
\{ t.CustID | ∃ s ∈ Loan ( s[custID]=t[CustID] ∧ s[amount]>1200) \}
**Cross Products in TRC**
- Find Customer names of customers who have loans greater than 1200
  - for Cust ID with loans we had
    - \( \{ t \mid \exists s \in \text{Loan} \ (s[\text{custID}]=t[\text{CustID}] \land s[\text{amount}]>1200) \} \)
- How about name? It exists in Customer relation?
  - For tuple \( c \in \text{Customer} \) what property does \( c \) have?
    - The customer ID in tuple \( c \) Customer relation is same as customer ID in tuple \( s \) in Loan relation
- Free variable?
  - Result tuples \( t \) must have only name
  - So this is the only attribute for which \( t \) is defined in the predicate condition

**Summary: Relational Model, Formal Query languages**
- **Relational Completeness:** Query language (e.g., SQL) can express every query that is expressible in relational algebra/calculus.
- Relational calculus is non-operational
  - users define queries in terms of what they want, not in terms of how to compute it.
- Algebra and safe calculus have same expressive power, leading to the notion of relational completeness.

**Recap: How to write a RA query?**
- Find out which tables you need to access
  - Compute \( \times \) of these tables
- What are the conditions/predicates you need to apply?
  - Determines what select \( \sigma \) operators you need to apply
- What attributes/columns are needed in result
  - Determines what project \( \pi \) operators you need
Limitations of the Relational Algebra / Calculus
Can’t do:
• Aggregate operations
• Recursive queries
• Complex (non-tabular) structures

• Most of these are expressible in SQL, XQuery – using other special operators
• Sometimes we even need the power of a Turing-complete programming language

Why Formal languages?
Example: Optimization Is Based on Algebraic Equivalences

- Relational algebra has laws of commutativity, associativity, etc. that imply certain expressions are equivalent in semantics
- They may be different in cost of evaluation!
  • \( \sigma_{P_1 \land P_2}(R) = \sigma_{P_1}(\sigma_{P_2}(R)) \)
  • \( (R_1 \bowtie R_2) = (R_2 \bowtie R_1) \)
  • \( (R_1 \bowtie R_2) \bowtie R_3 = R_1 \bowtie (R_2 \bowtie R_3) \)

- Query optimization finds the most efficient representation to evaluate (or one that’s not bad)

The Big Picture: SQL to Algebra to Query Plan to Web Page

```
SELECT *
FROM STUDENT, Takes, COURSE
WHERE STUDENT.sid = Takes.sid
AND Takes.cid = cid
```