Chapter 5

Relational Algebra and
Relational Calculus
Overview

◆ The previous chapter covers the relational model, which provides a formal description of the **structure** of a database

◆ This chapter covers the relational algebra and calculus, which provides a formal basis for the **query language** for the database
Chapter 5 – Topics We Will Cover

◆ Meaning of the term relational completeness.

◆ How to form queries in relational algebra.

◆ How to form queries in tuple relational calculus.
Introduction

◆ Relational algebra and relational calculus are formal languages associated with the relational model.

◆ Informally, relational algebra is a (high-level) procedural language and relational calculus a declarative, non-procedural language.

◆ However, formally both are equivalent to one another.

◆ A language that produces a relation that can be derived using relational calculus is relationally complete.
Relational Algebra

- Relational algebra operations work on one or more relations to define another relation without changing the original relations.

- Both operands and results are relations, so output from one operation can become input to another operation.

- Allows expressions to be nested, just as in arithmetic. This property is called **closure**.
Relational Algebra

- Five basic operations in relational algebra:
  1. Selection: selects rows from a relation
  2. Projection: selects columns from a relation
  3. Cartesian product
  4. Union
  5. Set Difference.

- Also have Join, Intersection, and Division operations, which can be expressed in terms of 5 basic operations.
Relational Algebra Operations

(a) Selection
(b) Projection
(c) Cartesian product

(d) Union
(e) Intersection
(f) Set difference

\[ P \cup Q \]

\[ P \cap Q \]

\[ P \setminus Q \]
Relational Algebra Operations

\[
\begin{array}{c|c}
T & U \\
\hline
A & B \\
\hline
a & 1 \\
b & 2 \\
\hline
B & C \\
\hline
1 & x \\
1 & y \\
3 & z \\
\end{array}
\quad
\begin{array}{c|c}
T \bowtie U & T \bowtie B U \\
\hline
A & B & C \\
\hline
a & 1 & x \\
\hline
a & 1 & y \\
\end{array}
\quad
\begin{array}{c|c|c}
T \bowtie C U & A & B & C \\
\hline
a & 1 & x \\
\hline
a & 1 & y \\
b & 2 \\
\end{array}
\]

- (g) Natural join
- (h) Semijoin
- (i) Left Outer join

\[
\begin{array}{c|c}
R & S \\
\hline
\end{array}
\quad
\begin{array}{c|c}
R + S & V \\
\hline
\end{array}
\quad
\begin{array}{c|c}
V & W \\
\hline
A & B \\
\hline
a & 1 \\
a & 2 \\
b & 1 \\
c & 2 \\
\hline
B & 1 \\
\hline
1 & 2 \\
\end{array}
\quad
\begin{array}{c|c}
V \div W & A \\
\hline
a & b \\
\end{array}
\]

- (j) Divis on (shaded area)
- Example of division

Pearson Education © 2009
Selection (or Restriction)

- $\sigma_{\text{predicate}} (R)$
  - Works on a single relation $R$ and defines a relation that contains only those tuples (rows) of $R$ that satisfy the specified condition (predicate).
Example - Selection (or Restriction)

- List all staff with a salary greater than £10,000.

\[ \sigma_{\text{salary} > 10000} \ (\text{Staff}) \]

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>position</th>
<th>sex</th>
<th>DOB</th>
<th>salary</th>
<th>branchNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>Manager</td>
<td>M</td>
<td>1-Oct-45</td>
<td>30000</td>
<td>B005</td>
</tr>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>Assistant</td>
<td>F</td>
<td>10-Nov-60</td>
<td>12000</td>
<td>B003</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>Supervisor</td>
<td>M</td>
<td>24-Mar-58</td>
<td>18000</td>
<td>B003</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>Manager</td>
<td>F</td>
<td>3-Jun-40</td>
<td>24000</td>
<td>B003</td>
</tr>
</tbody>
</table>

Pearson Education © 2009
Projection

\[ \Pi_{\text{col}_1, \ldots, \text{col}_n}(R) \]

- Works on a single relation \( R \) and defines a relation that contains a vertical subset of \( R \), extracting the values of specified attributes and eliminating duplicates.
Example - Projection

- Produce a list of salaries for all staff, showing only staffNo, fName, lName, and salary details.

\[ \Pi_{\text{staffNo, fName, lName, salary}}(\text{Staff}) \]

<table>
<thead>
<tr>
<th>staffNo</th>
<th>fName</th>
<th>lName</th>
<th>salary</th>
</tr>
</thead>
<tbody>
<tr>
<td>SL21</td>
<td>John</td>
<td>White</td>
<td>30000</td>
</tr>
<tr>
<td>SG37</td>
<td>Ann</td>
<td>Beech</td>
<td>12000</td>
</tr>
<tr>
<td>SG14</td>
<td>David</td>
<td>Ford</td>
<td>18000</td>
</tr>
<tr>
<td>SA9</td>
<td>Mary</td>
<td>Howe</td>
<td>9000</td>
</tr>
<tr>
<td>SG5</td>
<td>Susan</td>
<td>Brand</td>
<td>24000</td>
</tr>
<tr>
<td>SL41</td>
<td>Julie</td>
<td>Lee</td>
<td>9000</td>
</tr>
</tbody>
</table>
Union \((R \cup S)\)

- defines a relation that contains all the tuples of \(R\), or \(S\), or both \(R\) and \(S\)
  - duplicate tuples are eliminated.
  - \(R\) and \(S\) must be “union compatible”, which means they must have the same set of named attributes (the attributes may appear in two different orders as long as the names can be matched).

- If \(R\) and \(S\) have \(I\) and \(J\) tuples, respectively, union is obtained by concatenating them into one relation with a maximum of \((I + J)\) tuples.
Example - Union

- List all cities where there is either a branch office or a property for rent.

\[ \Pi_{\text{city}}(\text{Branch}) \cup \Pi_{\text{city}}(\text{PropertyForRent}) \]

<table>
<thead>
<tr>
<th>Id</th>
<th>City</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Knoxville</td>
<td>5485 Alcoa Hwy</td>
</tr>
<tr>
<td>2</td>
<td>Columbus</td>
<td>1678 Cardiff Rd.</td>
</tr>
<tr>
<td>3</td>
<td>Pittsburgh</td>
<td>100 Rockwood Ave</td>
</tr>
<tr>
<td>4</td>
<td>Columbus</td>
<td>585 Tremont Rd.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PropId</th>
<th>City</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nashville</td>
<td>110 Parthenon Way</td>
</tr>
<tr>
<td>2</td>
<td>Nashville</td>
<td>4868 Vanderbilt Dr</td>
</tr>
<tr>
<td>3</td>
<td>Knoxville</td>
<td>2408 Trillium Ln</td>
</tr>
<tr>
<td>4</td>
<td>Atlanta</td>
<td>5868 Peachtree Rd</td>
</tr>
<tr>
<td>5</td>
<td>Pittsburgh</td>
<td>386 Cedar Ln.</td>
</tr>
</tbody>
</table>

Note that we made the two relations “union-compatible” by projecting them down to a common column.
Set Difference

- **R – S**
  - Defines a relation consisting of the tuples that are in relation R, but not in S.
  - R and S must be union-compatible.
Example - Set Difference

- List all cities where there is a branch office but no properties for rent.

$$\Pi_{\text{city}}(\text{Branch}) - \Pi_{\text{city}}(\text{PropertyForRent})$$

<table>
<thead>
<tr>
<th>Id</th>
<th>City</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Knoxville</td>
<td>5485 Alcoa Hwy</td>
</tr>
<tr>
<td>2</td>
<td>Columbus</td>
<td>1678 Cardiff Rd.</td>
</tr>
<tr>
<td>3</td>
<td>Pittsburgh</td>
<td>100 Rockwood Ave</td>
</tr>
<tr>
<td>4</td>
<td>Columbus</td>
<td>585 Tremont Rd.</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PropId</th>
<th>City</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Nashville</td>
<td>110 Parthenon Way</td>
</tr>
<tr>
<td>2</td>
<td>Nashville</td>
<td>4868 Vanderbilt Dr</td>
</tr>
<tr>
<td>3</td>
<td>Knoxville</td>
<td>2408 Trillium Ln</td>
</tr>
<tr>
<td>4</td>
<td>Atlanta</td>
<td>5868 Peachtree Rd</td>
</tr>
<tr>
<td>5</td>
<td>Pittsburgh</td>
<td>386 Cedar Ln.</td>
</tr>
</tbody>
</table>

Note that we made the two relations “union-compatible” by projecting them down to a common column.
Intersection

- **R ∩ S**
  - Defines a relation consisting of the set of all tuples that are in both R and S.
  - R and S must be union-compatible.

- Expressed using basic operations:

  \[ R ∩ S = R - (R - S) \]

  \[ R ∩ S = R - (R - S) \]
Example - Intersection

◆ List all cities where there is both a branch office and at least one property for rent.

\[
\Pi_{\text{city}}(\text{Branch}) \cap \Pi_{\text{city}}(\text{PropertyForRent})
\]

<table>
<thead>
<tr>
<th>City</th>
<th>Address</th>
<th>City</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>Knoxville</td>
<td>5485 Alcoa Hwy</td>
<td>Nashville</td>
<td>110 Parthenon Way</td>
</tr>
<tr>
<td>Columbus</td>
<td>1678 Cardiff Rd.</td>
<td>Nashville</td>
<td>4868 Vanderbilt Dr</td>
</tr>
<tr>
<td>Pittsburgh</td>
<td>100 Rockwood Ave</td>
<td>Knoxville</td>
<td>2408 Trillium Ln</td>
</tr>
<tr>
<td>Columbus</td>
<td>585 Tremont Rd.</td>
<td>Atlanta</td>
<td>5868 Peachtree Rd</td>
</tr>
<tr>
<td>Pittsburgh</td>
<td>386 Cedar Ln.</td>
<td>Pittsburgh</td>
<td></td>
</tr>
</tbody>
</table>

Note that we made the two relations “union-compatible” by projecting them down to a common column
Cartesian product

- $R \times S$
  - Defines a relation that is the concatenation of every tuple of relation $R$ with every tuple of relation $S$. 
Example - Cartesian product

(Student) $\times$ (Courses)

<table>
<thead>
<tr>
<th>Id</th>
<th>Gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.5</td>
</tr>
<tr>
<td>2</td>
<td>2.6</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>CourseId</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CS140</td>
</tr>
<tr>
<td>1</td>
<td>ECE255</td>
</tr>
<tr>
<td>2</td>
<td>CS302</td>
</tr>
</tbody>
</table>

$= $

<table>
<thead>
<tr>
<th>Id</th>
<th>Gpa</th>
<th>Id</th>
<th>CourseId</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>3.5</td>
<td>1</td>
<td>CS140</td>
</tr>
<tr>
<td>1</td>
<td>3.5</td>
<td>1</td>
<td>ECE255</td>
</tr>
<tr>
<td>1</td>
<td>3.5</td>
<td>2</td>
<td>CS302</td>
</tr>
<tr>
<td>2</td>
<td>2.6</td>
<td>1</td>
<td>CS140</td>
</tr>
<tr>
<td>2</td>
<td>2.6</td>
<td>1</td>
<td>ECE255</td>
</tr>
<tr>
<td>2</td>
<td>2.6</td>
<td>2</td>
<td>CS302</td>
</tr>
</tbody>
</table>
Example - Cartesian product and Selection

- Use selection operation to extract those tuples where Student.Id = Courses.Id.

\[ \Pi_{\text{student.Id, courseId}}(\sigma_{\text{Student.Id} = \text{Courses.Id}}(\text{Student X Courses})) \]

- Cartesian product and Selection can be reduced to a single operation called a Join.
Join Operations

Join is a derivative of Cartesian product.

Equivalent to performing a Selection, using join predicate as selection formula, over Cartesian product of the two operand relations.

One of the most difficult operations to implement efficiently in an RDBMS and one reason why RDBMSs have intrinsic performance problems.
Join Operations

- Various forms of join operation
  - Theta join
  - Equijoin (a particular type of Theta join)
  - Natural join
  - Outer join
  - Semijoin
Theta join ($\theta$-join)

- $R \bowtie^F_S$
  - Defines a relation that contains tuples satisfying the predicate $F$ from the Cartesian product of $R$ and $S$.
  - The predicate $F$ is of the form $R.a_i \theta S.b_i$ where $\theta$ may be one of the comparison operators ($<$, $\leq$, $>$, $\geq$, $=$, $\neq$).
Theta join (\(\theta\)-join)

- Can rewrite Theta join using basic Selection and Cartesian product operations.

\[ R \bowtie_F S = \sigma_F(R \times S) \]

- Degree of a Theta join is sum of degrees of the operand relations \(R\) and \(S\). If predicate \(F\) contains only equality (\(=\)), the term *Equijoin* is used.
Example - Equijoin

- List the names and comments of all students who have taken a course.

\[
(\Pi_{Id, \text{Name}(\text{Student})})(\text{Student.Id} = \text{Courses.Id}) \bowtie (\Pi_{Id, \text{CourseId}, \text{Comment}(\text{Courses})})
\]

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Id</th>
<th>CourseId</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smiley</td>
<td>1</td>
<td>CS140</td>
<td>“Great course”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>ECE255</td>
<td>“Amazing”</td>
</tr>
<tr>
<td>2</td>
<td>Pooh</td>
<td>2</td>
<td>CS302</td>
<td>“Crushed my gpa”</td>
</tr>
<tr>
<td>3</td>
<td>Nels</td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

= 

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Id</th>
<th>CourseId</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smiley</td>
<td>1</td>
<td>CS140</td>
<td>“Great course”</td>
</tr>
<tr>
<td></td>
<td></td>
<td>1</td>
<td>ECE255</td>
<td>“Amazing”</td>
</tr>
<tr>
<td>2</td>
<td>Pooh</td>
<td>2</td>
<td>CS302</td>
<td>“Crushed my gpa”</td>
</tr>
</tbody>
</table>
Natural join

- **R⋈S**
  - An Equijoin of the two relations R and S over all common attributes \( x \). One occurrence of each common attribute is eliminated from the result.
Example – Natural Join

◆ List the names and comments of all students who have taken a course.

$$(\Pi_{\text{Id, Name(\text{Student})}}) \bowtie (\Pi_{\text{Id, CourseId, Comment(\text{Courses})}})$$
Outer join

- To display rows in the result that do not have matching values in the join column, use Outer join.

- $R \bowtie S$
  - (Left) outer join is join in which tuples from $R$ that do not have matching values in common columns of $S$ are also included in result relation.
Example - Left Outer join

 Produce a report that shows all students and the courses they are taking, even if the student is not taking a course.

\[(\Pi_{Id, Name(Student)}) \bowtie_{Student.Id= Courses.id} (\Pi_{Id, CourseId, Comment(Courses)})\]
Semijoin

- $R \bowtie_F S$
  - Defines a relation that contains the tuples of $R$ that participate in the join of $R$ with $S$.

- Can rewrite Semijoin using Projection and Join:

  $R \bowtie_F S = \Pi_A(R \bowtie_F S)$
Example - Semijoin

- List complete details of all students who are taking CS140.

\[ \text{Student} \triangleright \text{Student.Id=Courses.Id} (\sigma_{\text{CourseId}=\text{CS140}} (\text{Courses})) \]

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>Smiley</td>
</tr>
<tr>
<td>2</td>
<td>Pooh</td>
</tr>
<tr>
<td>3</td>
<td>Nels</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>CourseId</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>1</td>
<td>CS140</td>
<td>“Great course”</td>
</tr>
<tr>
<td>1</td>
<td>ECE255</td>
<td>“Amazing”</td>
</tr>
<tr>
<td>2</td>
<td>CS140</td>
<td>“Crushed my gpa”</td>
</tr>
</tbody>
</table>

1 | Smiley |
2 | Pooh   |
Division (R ÷ S)

- Example Queries
  - List all clients who have viewed all properties with three rooms
  - List all guests who have stayed in all rooms of the Grosvenor Hotel
  
  (the relations below were obtained from the booking and room relations by selecting only those bookings and rooms associated with the Grosvenor Hotel and by projecting away the dateFrom, bookingNo and dateTo fields in the booking relation)

<table>
<thead>
<tr>
<th>guestNo</th>
<th>roomNo</th>
</tr>
</thead>
<tbody>
<tr>
<td>20</td>
<td>100</td>
</tr>
<tr>
<td>30</td>
<td>200</td>
</tr>
<tr>
<td>30</td>
<td>300</td>
</tr>
<tr>
<td>10</td>
<td>100</td>
</tr>
<tr>
<td>30</td>
<td>100</td>
</tr>
<tr>
<td>20</td>
<td>300</td>
</tr>
</tbody>
</table>

\[
\begin{array}{c|c}
\text{roomNo} & \\
\hline
100 & \\
200 & \\
300 & \\
\end{array}
\]

\[
\begin{array}{c|c}
\text{guestNo} & \\
\hline
30 & \\
\end{array}
\]
Division \((R \div S)\)

- **Preconditions**
  - \(R\) is defined over the attribute set \(A\)
  - \(S\) is defined over the attribute set \(B\), such that \(B \subseteq A\)
  - Let \(C = A - B\) (i.e., \(C\) is the set of attributes of \(R\) that are not attributes of \(S\))

- Defines a relation over the attributes \(C\) that consists of set of tuples from \(R\) that match combination of *every* tuple in \(S\).

- A more “severe” form of a semi-join, because instead of a match applying to a single row in \(R\) and a single row in \(S\), it requires a match between multiple rows of \(R\) and \(S\).
Easy Way to Compute Division

1. Cross out all rows in R whose B attributes do not match any row in S

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ralph</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>Joe</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>Ralph</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>Mary</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>Mary</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>Fran</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>Ralph</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>Mary</td>
<td>d</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>7</td>
</tr>
</tbody>
</table>

Easy Way to Compute Division

2. Start with the first unprocessed row and find the other rows in R with the same values for the attributes in C. If these rows match what is in S, add a row with those values for the C attributes to the result relation.

<table>
<thead>
<tr>
<th>R</th>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>Ralph</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td></td>
<td>Joe</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td></td>
<td>Ralph</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Mary</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Mary</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>Fran</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Ralph</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td></td>
<td>Mary</td>
<td>d</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td></td>
<td>c</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
</tr>
<tr>
<td>Ralph</td>
</tr>
</tbody>
</table>
Easy Way to Compute Division

2. Start with the first unprocessed row and find the other rows in R with the same values for the attributes in C. If these rows match what is in S, add a row with those values for the C attributes to the result relation.

<table>
<thead>
<tr>
<th>X</th>
<th>Y</th>
<th>Z</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ralph</td>
<td>a</td>
<td>1</td>
</tr>
<tr>
<td>Joe</td>
<td>b</td>
<td>2</td>
</tr>
<tr>
<td>Ralph</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>Mary</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>Mary</td>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>Fran</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>Ralph</td>
<td>c</td>
<td>7</td>
</tr>
<tr>
<td>Mary</td>
<td>d</td>
<td>10</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>S</th>
<th>Result</th>
</tr>
</thead>
<tbody>
<tr>
<td>Y</td>
<td>Z</td>
</tr>
<tr>
<td>a</td>
<td>6</td>
</tr>
<tr>
<td>c</td>
<td>7</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>X</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ralph</td>
</tr>
<tr>
<td>Mary</td>
</tr>
</tbody>
</table>
Division \((R \div S)\): Formal Derivation

- Expressed using basic operations:
  \[ T_1 \leftarrow \Pi_C(R) \]  
  \hspace{1cm} \text{// Restrict } T_1 \text{ to the attributes in } R \text{ that are not in } S

  \[ T_2 \leftarrow \Pi_C((T_1 \times S) - R) \]  
  \hspace{1cm} \text{// Remove from } R \text{ the multi-rows that match } S

  \[ T \leftarrow T_1 - T_2 \]  
  \hspace{1cm} \text{// Remove from } R \text{ the rows that do not fully match the multi-rows in } S

<table>
<thead>
<tr>
<th>(R)</th>
<th>(S)</th>
<th>(T_1)</th>
<th>(T_1 \times S)</th>
<th>(T_2)</th>
<th>(T)</th>
</tr>
</thead>
<tbody>
<tr>
<td>X</td>
<td>Y</td>
<td>X</td>
<td>X</td>
<td>X</td>
<td>X</td>
</tr>
<tr>
<td>a</td>
<td>1</td>
<td>1</td>
<td>a</td>
<td>c</td>
<td>a</td>
</tr>
<tr>
<td>a</td>
<td>2</td>
<td>2</td>
<td>b</td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>b</td>
<td>1</td>
<td></td>
<td>c</td>
<td></td>
<td></td>
</tr>
<tr>
<td>b</td>
<td>2</td>
<td></td>
<td></td>
<td></td>
<td>b</td>
</tr>
<tr>
<td>c</td>
<td>1</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
Aggregate Operations

\[ \rho_R(\text{colnames}) \mathcal{F}_{AL}(R) \]

- Applies aggregate function list, \( AL \), to \( R \) to define a relation over the aggregate list.
- \( AL \) contains one or more \( (<\text{aggregate\_function}>, <\text{attribute}>) \) pairs.
- \( \rho_R(\text{colnames}) \) names the columns being created by the aggregate functions: same as a projection.

- Main aggregate functions are: \( \text{COUNT, SUM, AVG, MIN, and MAX} \).
Example – Aggregate Operations

- How many properties cost more than £350 per month to rent?

\[ \rho_R(\text{myCount}) \mathcal{F}_{\text{COUNT}} \text{propertyNo} (\sigma_{\text{rent} > 350} (\text{PropertyForRent})) \]
Grouping Operation

\[ \text{GA} \Join_{\text{AL}} (\text{R}) \]

- Groups tuples of R by grouping attributes, GA, and then applies aggregate function list, AL, to define a new relation.

- AL contains one or more \(<\text{aggregate	extunderscore function}>, \text{<attribute}>\) pairs.

- Resulting relation contains the grouping attributes, GA, along with results of each of the aggregate functions.
Example – Grouping Operation

- Find the number of staff working in each branch and the sum of their salaries.

\( \rho_R(\text{branchNo}, \text{myCount}, \text{mySum}) \)

\( \text{branchNo} \Join \text{COUNT staffNo, SUM salary (Staff)} \)

<table>
<thead>
<tr>
<th>branchNo</th>
<th>myCount</th>
<th>mySum</th>
</tr>
</thead>
<tbody>
<tr>
<td>B003</td>
<td>3</td>
<td>54000</td>
</tr>
<tr>
<td>B005</td>
<td>2</td>
<td>39000</td>
</tr>
<tr>
<td>B007</td>
<td>1</td>
<td>9000</td>
</tr>
</tbody>
</table>
Relational Calculus

- Relational calculus query specifies what is to be retrieved rather than how to retrieve it.
  - No description of how to evaluate a query.

- In first-order logic (or predicate calculus), predicate is a truth-valued function with arguments.

- When we substitute values for the arguments, function yields an expression, called a proposition, which can be either true or false.
Relational Calculus

◆ If predicate contains a variable (e.g. ‘$x$ is a member of staff’), there must be a range for $x$.

◆ When we substitute some values of this range for $x$, proposition may be true; for other values, it may be false.

◆ When applied to databases, relational calculus has forms: *tuple* and *domain*.
  
  – In this class only consider tuple relational calculus
Tuple Relational Calculus

- Interested in finding tuples for which a predicate is true. Based on use of tuple variables.

- Tuple variable is a variable that ‘ranges over’ a named relation: i.e., variable whose only permitted values are tuples of the relation.

- Specify range of a tuple variable $S$ as the Staff relation using the notation:
  
  $\text{Staff}(S)$

- To find set of all tuples $S$ such that $P(S)$ is true:
  
  $\{S \mid P(S)\}$
Tuple Relational Calculus - Example

- To find details of all staff earning more than £10,000:
  \[ \{ S \mid \text{Staff}(S) \land S.\text{salary} > 10000 \} \]

- To find a particular attribute, such as salary, write:
  \[ \{ S.\text{salary} \mid \text{Staff}(S) \land S.\text{salary} > 10000 \} \]
If you have trouble writing a relational calculus query, write an SQL query and translate it:

To find the salary details of all staff earning more than £10,000:

SQL: SELECT S.staffNo, S.salary
    FROM Staff S WHERE S.salary > 10000

Relational Calculus:

\{S.staffNo, S.salary \mid \text{Staff}(S) \land S.salary > 10000\}

Translation:

» Select attributes go on left side of | character
» FROM relations are mapped to tuple variables and combined using AND
» WHERE is appended to tuple variables with an AND
Tuple Relational Calculus

- Can use two quantifiers to tell how many instances the predicate applies to:
  - Existential quantifier \( \exists \) (‘there exists’)
  - Universal quantifier \( \forall \) (‘for all’)

- Tuple variables qualified by \( \forall \) or \( \exists \) are called bound variables, otherwise called free variables.
  - Only variables to the left of the bar (\( | \) ) may be free variables
Example: List all tuples in Staff such that the staff member works in a London Branch:

- Relational calculus query

\[ \{S\. fName, S\. lName \mid \text{Staff}(S) \land (\exists B)(\text{Branch}(B) \land (B\. branchNo = S\. branchNo) \land B\. city = \text{`London'})} \} \]
Tuple Relational Calculus

- Universal quantifier is used in statements about every instance, such as:
  
  $$(\forall B) \ (\text{Branch}(B) \land B\text{.city} \neq \text{‘Paris’})$$

1. The book’s interpretation of $\forall$ is confusing

2. Its construct iterates over all possible rows in the relation

- Means ‘For all Branch tuples, the address is not in Paris’.

- Can also use $\neg(\exists B) \ (\text{Branch}(B) \land B\text{.city} = \text{‘Paris’})$ which means ‘There are no branches with an address in Paris’.
Tuple Relational Calculus

◆ Formulae should be unambiguous and make sense.
◆ A (well-formed) formula is made out of atoms:
  » \( R(S_i) \), where \( S_i \) is a tuple variable and \( R \) is a relation
  » \( S_i.a_1 \ \theta \ S_j.a_2 \) where \( \theta \) is a relational or Boolean operator
  » \( S_i.a_1 \ \theta \ c \)
◆ Can recursively build up formulae from atoms:
  » An atom is a formula
  » If \( F_1 \) and \( F_2 \) are formulae, so are their conjunction, \( F_1 \land F_2 \); disjunction, \( F_1 \lor F_2 \); and negation, \( \neg F_1 \)
  » If \( F \) is a formula with free variable \( X \), then \( (\exists X)(F) \) and \( (\forall X)(F) \) are also formulae.
Tuple Relational Calculus

- Boolean operators are typically used for SELECT queries
- $(\exists X)(F)$ is typically used for semi-joins
- $\neg(\exists X)(F)$ and $(\forall X)(F)$ are typically used for integrity constraints
  - Example: All staff members must make less than $100,000$
  - Example: There does not exist a staff member who manages more than 100 properties
- $\exists$ and $\forall$ form subqueries
Tuple Relational Calculus: Set Operations

Union: List all cities where there is either a branch office or a property for rent:

\[ B.\text{city} \cup P.\text{city} = \{T.\text{city} \mid (\exists B)(\text{Branch}(B) \land (B.\text{city} = T.\text{city})) \lor (\exists P)(\text{PropertyForRent}(P) \land (P.\text{city} = T.\text{city}))\} \]

Difference: List all cities where there is a branch office but no property for rent:

\[ B.\text{city} - P.\text{city} = \{B \mid \text{Branch}(B) \land \neg (\exists P)(\text{PropertyForRent}(P) \land B.\text{city} = P.\text{city})\} \]

Intersection: List all cities where there is both a branch office and a property for rent:

\[ B.\text{city} \cap P.\text{city} = \{B.\text{city} \mid \text{Branch}(B) \land (\exists P)(\text{PropertyForRent}(P) \land B.\text{city} = P.\text{city})\} \]
Tuple Relational Calculus: Set Division

◆ Suppose you have the following relations:
   – Tourist(tname, addr): tourist information
   – Visit(tname, pname): parks visited by tourists
   – Park(pname, state): park information
◆ List the tourists who have visited all parks in TN

\{ T \mid \text{Tourist}(T) \land (\forall P)(\text{Park}(P) \\
\land ((P.\text{state}! = 'TN') \lor (\exists V)(\text{Visit}(V) \land T.tname = V.tname \\
\land V.pname = P.pname))) )\}
Example - Tuple Relational Calculus

- List the names of all managers who earn more than £25,000.

\{ S.fName, S.lName \mid Staff(S) \land S.position = 'Manager' \land S.salary > 25000 \}
Example Relational Calculus Queries

- List the staff who manage properties for rent in Glasgow.

\[ \{ S \mid \text{Staff}(S) \land (\exists P) (\text{PropertyForRent}(P) \land (P.\text{staffNo} = S.\text{staffNo}) \land P.\text{city} = '\text{Glasgow}')\} \]

SQL: 
\[
\text{SELECT S.* FROM Staff S}
\]
\[
\text{WHERE EXISTS(SELECT * FROM PropertyForRent P}
\]
\[
\text{WHERE P.\text{staffNo} = S.\text{staffNo}}
\]
\[
\text{AND P.\text{city} = 'Glasgow'}\);
\]
Example Relational Calculus Queries

- **List the staff who manage properties for rent in Glasgow.**
  - Seemingly comparable Query that you might try which is wrong

\[
\{S \mid \text{Staff}(S) \land \text{PropertyForRent}(P) \land (P.\text{staffNo} = S.\text{staffNo}) \land P.\text{city} = \text{‘Glasgow’})\}
\]

- PropertyForRent(P) makes P a free variable, even though P does not appear on the left side of |
Example Relational Calculus Queries

◆ List each property that rents for more than $450 and list the name of the staff member who manages the property.

\{P, S.name \mid \text{Staff}(S) \land \text{PropertyForRent}(P) \land (P.\text{staffNo} = S.\text{staffNo}) \land P.\text{rent} > 450\}\}

SQL: SELECT P.*, S.name FROM Staff S Natural Join PropertyForRent P
WHERE P.rent > 450;
Example - Tuple Relational Calculus

- List the names of staff who currently do not manage any properties.

\[ \{ S.f\text{Name}, S.l\text{Name} \mid \text{Staff}(S) \land (\neg (\exists P) (\text{PropertyForRent}(P) \land (S.\text{staffNo} = P.\text{staffNo}))) \} \]

Or

\[ \{ S.f\text{Name}, S.l\text{Name} \mid \text{Staff}(S) \land ((\forall P) (\text{PropertyForRent}(P) \land (S.\text{staffNo} \neq P.\text{staffNo}))) \} \]
Example - Tuple Relational Calculus

- List the names of clients who have viewed a property for rent in Glasgow.

\[
\{ \text{C.firstName, C.lastName} \mid \text{Client(C) } \land ((\exists V)(\exists P) \\
(\text{Viewing(V) } \land \text{PropertyForRent(P) } \land \\
(\text{C.clientNo=V.clientNo}) \land \\
(V.propertyNo=P.propertyNo) \land \\
P\text{.city =‘Glasgow’}) ) \}
\]
Expressions can generate an infinite set. For example:
\{S \mid \neg\text{Staff}(S)\}

To avoid this, add restriction that all values in result must be values in the domain of the expression.