CS 5614: (Big) Data Management Systems

B. Aditya Prakash

Lecture #2: The Relational Model, and SQL/Relational Algebra
Data Model

- A Data Model is a notation for describing data or information.
  - **Structure of data** (e.g. arrays, structs)
    - Conceptual model: In databases, structures are at a higher level.
  - **Operations on data** (Modifications and Queries)
  - **Constraints on data** (what the data can be)

- Examples of data models
  - The Relational Model
  - The Semistructured-Data Model
    - XML and related standards
  - Object-Relational Model
The Relational Model

- **Structure**: Table (like an array of structs)
- **Operations**: Relational algebra (selection, projection, conditions, etc)
- **Constraints**: E.g., grades can be only \{A, B, C, F\}

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermione Grainger</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Draco Malfoy</td>
<td>Potions</td>
<td>B</td>
</tr>
<tr>
<td>Harry Potter</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Ron Weasley</td>
<td>Potions</td>
<td>C</td>
</tr>
</tbody>
</table>
The Semi-structured model

```xml
<CoursesTaken>
  <Student>Hermione Grainger</Student>
  <Course>Potions</Course>
  <Grade>A</Grade>
  <Student>Draco Malfoy</Student>
  <Course>Potions</Course>
  <Grade>B</Grade>
  ...
</CoursesTaken>
```

- **Structure**: Trees or graphs, tags define role played by different pieces of data.
- **Operations**: Follow paths in the implied tree from one element to another.
- **Constraints**: E.g., can express limitations on data types.
Comparing the two models

- Flexibility: XML can represent graphs
- Ease of use: SQL enables programmer to express wishes at high level.
The Relational Model

- Simple: Built around a single concept for modeling data: the relation or table.
  - A relational database is a collection of relations.
  - Each relation is a table with rows and columns.

- Supports high-level programming language (SQL).
  - Limited but very useful set of operations

- Has an elegant mathematical design theory.
- Most current DBMS are relational (Oracle, IBM DB2, MS SQL)
Relations

- A relation is a two-dimensional table:
  - Relation == table.
  - Attribute == column name.
  - Tuple == row (not the header row).

- Database == collection of relations.

- A relation has two parts:
  - **Schema** defines column heads of the table (attributes).
  - **Instance** contains the data rows (tuples, rows, or records) of the table.

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermione Grainger</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Draco Malfoy</td>
<td>Potions</td>
<td>B</td>
</tr>
<tr>
<td>Harry Potter</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Ron Weasley</td>
<td>Potions</td>
<td>C</td>
</tr>
</tbody>
</table>
The schema of a relation is the name of the relation followed by a parenthesized list of attributes.

CoursesTaken(Student, Course, Grade)

A design in a relational model consists of a set of schemas.

Such a set of schemas is called a relational database schema.
Relations: Equivalent Representations

CoursesTaken:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermione Grainger</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Draco Malfoy</td>
<td>Potions</td>
<td>B</td>
</tr>
<tr>
<td>Harry Potter</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Ron Weasley</td>
<td>Potions</td>
<td>C</td>
</tr>
</tbody>
</table>

CoursesTaken(Student, Course, Grade)

- Relation is a \textit{set} of tuples and not a list of tuples.
  - Order in which we present the tuples does not matter.
  - Very important!
- The attributes in a schema are also a \textit{set} (not a list).
  - Schema is the same irrespective of order of attributes.

CoursesTaken(Student, Grade, Course)

- We specify a “standard” order when we introduce a schema.

- How many equivalent representations are there for a relation with \( m \) attributes and \( n \) tuples? \( m! \cdot n! \)
Degree and Cardinality

CoursesTaken:

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermione Grainger</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Draco Malfoy</td>
<td>Potions</td>
<td>B</td>
</tr>
<tr>
<td>Harry Potter</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Ron Weasley</td>
<td>Potions</td>
<td>C</td>
</tr>
</tbody>
</table>

- **Degree/Arity** is the number of fields/attributes in schema (=3 in the table above)
- **Cardinality** is the number of tuples in relation (=4 in the table above)
Keys of Relations

- Keys are one form of integrity constraints (IC)
  - No pair of tuples should have identical keys
- What is the key for CoursesTaken?
  - *Student* if only one course in the relation
  - Pair *(Student, Course)* if multiple courses
  - What if student takes same course many times?

<table>
<thead>
<tr>
<th>Student</th>
<th>Course</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>Hermione Grainger</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Draco Malfoy</td>
<td>Potions</td>
<td>B</td>
</tr>
<tr>
<td>Harry Potter</td>
<td>Potions</td>
<td>A</td>
</tr>
<tr>
<td>Ron Weasley</td>
<td>Potions</td>
<td>C</td>
</tr>
</tbody>
</table>
Keys of Relations

- Keys help associate tuples in different relations

<table>
<thead>
<tr>
<th>SID</th>
<th>CID</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>15-401</td>
<td>A</td>
</tr>
<tr>
<td>111</td>
<td>15-401</td>
<td>B</td>
</tr>
<tr>
<td>123</td>
<td>14-501</td>
<td>B</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SID</th>
<th>Student</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>Hermione Grainger</td>
<td>3.9</td>
</tr>
<tr>
<td>111</td>
<td>Draco Malfoy</td>
<td>3.0</td>
</tr>
<tr>
<td>234</td>
<td>Harry Potter</td>
<td>3.7</td>
</tr>
<tr>
<td>456</td>
<td>Ron Weasley</td>
<td>3.1</td>
</tr>
</tbody>
</table>
Example

- Create a database for managing class enrollments in a single semester. You should keep track of all students (their names, Ids, and addresses) and professors (name, Id, department). Do not record the address of professors but keep track of their ages. Maintain records of courses also. Like what classroom is assigned to a course, what is the current enrollment, and which department offers it. At most one professor teaches each course. Each student evaluates the professor teaching the course. Note that all course offerings in the semester are unique, i.e. course names and numbers do not overlap. A course can have $\geq 0$ pre-requisites, excluding itself. A student enrolled in a course must have enrolled in all its pre-requisites. Each student receives a grade in each course. The departments are also unique, and can have at most one chairperson (or dept. head). A chairperson is not allowed to head two or more departments.
Create a database for managing class enrollments in a single semester. You should keep track of all students (their names, IDs, and addresses) and professors (name, ID, department). Do not record the address of professors but keep track of their ages. Maintain records of courses also. Like what classroom is assigned to a course, what is the current enrollment, and which department offers it. At most one professor teaches each course. Each student evaluates the professor teaching the course. Note that all course offerings in the semester are unique, i.e., course names and numbers do not overlap. A course can have \( \geq 0 \) pre-requisites, excluding itself. A student enrolled in a course must have enrolled in all its pre-requisites. Each student receives a grade in each course. The departments are also unique, and can have at most one chairperson (or dept. head). A chairperson is not allowed to head two or more departments.
Relational Design for the Example

- **Students** (PID: `string`, Name: `string`, Address: `string`)
- **Professors** (PID: `string`, Name: `string`, Office: `string`, Age: `integer`, DepartmentName: `string`)
- **Courses** (Number: `integer`, DeptName: `string`, CourseName: `string`, Classroom: `string`, Enrollment: `integer`)
- **Teach** (ProfessorPID: `string`, Number: `integer`, DeptName: `string`)
- **Take** (StudentPID: `string`, Number: `integer`, DeptName: `string`, Grade: `string`, ProfessorEvaluation: `integer`)
- **Departments** (Name: `string`, ChairmanPID: `string`)
- **PreReq** (Number: `integer`, DeptName: `string`, PreReqNumber: `integer`, PreReqDeptName: `string`)
Relational Design Example: Keys?

- Students (PID: \textit{string}, Name: \textit{string}, Address: \textit{string})

- Professors (PID: \textit{string}, Name: \textit{string}, Office: \textit{string}, Age: \textit{integer}, DepartmentName: \textit{string})

- Courses (Number: \textit{integer}, DeptName: \textit{string}, CourseName: \textit{string}, Classroom: \textit{string}, Enrollment: \textit{integer})

- Teach (ProfessorPID: \textit{string}, Number: \textit{integer}, DeptName: \textit{string})

- Take (StudentPID: \textit{string}, Number: \textit{integer}, DeptName: \textit{string}, Grade: \textit{string}, ProfessorEvaluation: \textit{integer})

- Departments (Name: \textit{string}, ChairmanPID: \textit{string})

- PreReq (Number: \textit{integer}, DeptName: \textit{string}, PreReqNumber: \textit{integer}, PreReqDeptName: \textit{string})
Relational Design: Keys?

- Students (PID: string, Name: string, Address: string)
- Professors (PID: string, Name: string, Office: string, Age: integer, DepartmentName: string)
- Courses (Number: integer, DeptName: string, CourseName: string, Classroom: string, Enrollment: integer)
- Teach (ProfessorPID: string, Number: integer, DeptName: string)
- Take (StudentPID: string, Number: integer, DeptName: string, Grade: string, ProfessorEvaluation: integer)
- Departments (Name: string, ChairmanPID: string)
- PreReq (Number: integer, DeptName: string, PreReqNumber: integer, PreReqDeptName: string)
Issues to Consider in the Design

- Can we merge Courses and Teach since each professor teaches at most one course?
- Do we need a separate relation to store evaluations?
- How can we handle pre-requisites that are “or”’s, e.g., you can take CS 4604 if you have taken either CS 3114 or CS 2606?
- How do we generalize this schema to handle data over more than one semester?
- What modifications does the schema need if more than one professor can teach a course?
SQL AND RELATIONAL ALGEBRA
Relational Algebra

- Relational algebra is a notation for specifying queries about the contents of relations

- Notation of relational algebra eases the task of reasoning about queries

- Operations in relational algebra have counterparts in SQL
What is SQL

- SQL = Structured Query Language (pronounced “sequel”).
- Language for defining as well as querying data in an RDBMS.
- Primary mechanism for querying and modifying the data in an RDBMS.
- SQL is declarative:
  - Say what you want to accomplish, without specifying how.
  - One of the main reasons for the commercial success of RDBMSs.
- SQL has many standards and implementations:
  - ANSI SQL
  - SQL-92/SQL2 (null operations, outerjoins)
  - SQL-99/SQL3 (recursion, triggers, objects)
  - Vendor-specific variations.
What is an Algebra?

- An algebra is a set of operators and operands
  - Arithmetic: operands are variables and constants, operators are +, -, ×, ÷, /, etc.
  - Set algebra: operands are sets and operators are ∩, U, -

- An algebra allows us to
  - **construct expressions** by combining operands and expression using operators
  - has **rules for reasoning** about expressions

\[
\begin{align*}
a^2 + 2 \times a \times b + 2b, & \quad (a + b)^2 \\
R - (R - S), & \quad R \cap S 
\end{align*}
\]
FUNDAMENTAL
Relational operators

- selection \( \sigma_{\text{condition}} (R) \)
- projection \( \pi_{\text{att-list}} (R) \)
- cartesian product \( R \times S \)
- set union \( R \cup S \)
- set difference \( R - S \)
Projection

- The projection operator produces from a relation R a new relation containing only some of R’s columns
  - “Delete” (i.e. not show) attributes not in projection list
  - Duplicates eliminated (sets vs multisets)

- To obtain a relation containing only the columns $A_1, A_2, \ldots, A_n$ of R

  \[
  RA: \quad \pi A_1, A_2, \ldots, A_n (R)
  \]

  SQL: \quad \text{SELECT DISTINCT } A_1, A_2, \ldots, A_n \text{ FROM } R;
Projection Example

\[ S_2 \]

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{surname}, \text{rating}}(S_2) \]

<table>
<thead>
<tr>
<th>sname</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>yuppy</td>
<td>9</td>
</tr>
<tr>
<td>lubber</td>
<td>8</td>
</tr>
<tr>
<td>guppy</td>
<td>5</td>
</tr>
<tr>
<td>rusty</td>
<td>10</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{age}}(S_2) \]

<table>
<thead>
<tr>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>35.0</td>
</tr>
<tr>
<td>55.5</td>
</tr>
</tbody>
</table>
Selection

- The selection operator applied to a relation R produces a new relation with a subset of R’s tuples.
- The tuples in the resulting relation satisfy some condition C that involves the attributes of R
  - with duplicate removal

\[ RA: \sigma_C(R) \]

\[ SQL: \text{SELECT * FROM R WHERE C;} \]

- The WHERE clause of a SQL command corresponds to \( \sigma( ) \)
Selection: Syntax of Conditional

- Syntax of conditional (C): similar to conditionals in programming languages.

- Values compared are constants and attributes of the relations mentioned in the FROM clause.

- We may apply usual arithmetic operators to numeric values before comparing them.

**RA**  Compare values using standard arithmetic operators.

**SQL** Compare values using =, <>, <, >, <=, >=.
Selection Example

\[ s_{2} \]

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[ \sigma_{\text{rating} > 8} (s_{2}) \]

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

\[ \pi_{\text{sname}, \text{rating}} (\sigma_{\text{rating} > 8} (s_{2})) \]

<table>
<thead>
<tr>
<th>sname</th>
<th>rating</th>
</tr>
</thead>
<tbody>
<tr>
<td>yuppy</td>
<td>9</td>
</tr>
<tr>
<td>rusty</td>
<td>10</td>
</tr>
</tbody>
</table>

Combining Operators
Set Operations: Union

- Standard definition: The union of two relations R and S is the set of tuples that are in R, or S or in both.

- When is it valid?
  - R and S must have identical sets of attributes and the types of the attributes must be the same.
  - The attributes of R and S must occur in the same order.
Set Operations: Union

- RA  R U S
- SQL (SELECT * FROM R)
  UNION
  (SELECT * FROM S);
Set Operations: Intersection

- The intersection of $R$ and $S$ is the set of tuples in both $R$ and $S$
- Same conditions hold on $R$ and $S$ as for the union operator
- $RA \quad R \cap S$
- $SQL \quad (SELECT * FROM R)
  \quad INTERSECT
  \quad (SELECT * FROM S)$;
Set Operations: Difference

- Set of tuples in R but NOT in S
- Same conditions on R and S as union
- \( R \cap S \)
- SQL
  
  \[
  \text{(SELECT * FROM R)}
  \]
  
  \[
  \text{EXCEPT}
  \]
  
  \[
  \text{(SELECT * FROM S)};
  \]

- \( R - (R - S) = R \cap S \)
### Difference

#### $S1$

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

#### $S2$

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>28</td>
<td>yuppy</td>
<td>9</td>
<td>35.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>44</td>
<td>guppy</td>
<td>5</td>
<td>35.0</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

#### $S1 - S2$

<table>
<thead>
<tr>
<th>sid</th>
<th>surname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
</tbody>
</table>
What about strings?

find student ssns who live on “main” (st or str or street - ie., “main st” or “main str” ... )
What about strings?

find student ssnss who live on “main” (st or str or street)

    select ssn
    from student
    where address like “main%”

%: variable-length don’t care
_: single-character don’t care
Relational operators

Are we done yet?

Q: Give a query we can not answer yet!
Relational operators

A: any query across two or more tables, eg., ‘find names of students in 4604’

Q: what extra operator do we need??

<table>
<thead>
<tr>
<th>STUDENT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ssn</td>
<td>Name</td>
<td>Address</td>
</tr>
<tr>
<td>123</td>
<td>smith</td>
<td>main str</td>
</tr>
<tr>
<td>234</td>
<td>jones</td>
<td>forbes ave</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4604</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>3114</td>
<td>B</td>
</tr>
</tbody>
</table>
Relational operators

A: any query across **two** or more tables, 
   eg., ‘find names of students in 4604’

Q: what extra operator do we need??

A: surprisingly, cartesian product is enough!

<table>
<thead>
<tr>
<th>STUDENT</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>Ssn</td>
<td>Name</td>
<td>Address</td>
</tr>
<tr>
<td>123</td>
<td>smith</td>
<td>main str</td>
</tr>
<tr>
<td>234</td>
<td>jones</td>
<td>forbes ave</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SSN</th>
<th>c-id</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>123</td>
<td>4604</td>
<td>A</td>
</tr>
<tr>
<td>234</td>
<td>3114</td>
<td>B</td>
</tr>
</tbody>
</table>
The Cartesian product (or cross-product or product) of two relations R and S is a the set of pairs that can be formed by pairing each tuple of R with each tuple of S.

- The result is a relation whose schema is the schema for R followed by the schema for S.

\[
\text{RA: } R \times S \\
\text{SQL: } \text{SELECT} * \text{ FROM} R, S;
\]
### Cartesian Product

#### \( S_1 \)

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

#### \( R_1 \)

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

#### \( S_1 \times R_1 \)

<table>
<thead>
<tr>
<th>(sid)</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
<th>(sid)</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>31</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

We **rename** attributes to avoid ambiguity or we **prefix attribute** with the name of the relation it belongs to.
REMINDEER: FUNDAMENTAL
Relational operators

- selection \( \sigma_{\text{condition}}(R) \)
- projection \( \pi_{\text{att-list}}(R) \)
- cartesian product \( R \times S \)
- set union \( R \cup S \)
- set difference \( R - S \)
Surprisingly, they are enough, to help us answer almost any query we want!

- set intersection  --- (We have seen this)
- join (theta join, equi-join, natural join)
- ‘rename’ operator  \( \rho_{R'}(R) \)
- division  \( R \div S \)
Theta-Join

- The theta-join of two relations R and S is the *set of tuples* in the Cartesian product of R and S that satisfy some condition C.

\[
\text{RA: } R \bowtie_C S
\]

\[
\text{SQL: } \text{SELECT } * \text{ FROM } R, S \text{ WHERE } C;
\]

- \( R \bowtie_C S = \sigma_C (R \times S) \)
Theta-Join

\[ S_1 \bowtie_{s_1.sid < R_1.sid} R_1 \]

<table>
<thead>
<tr>
<th>sid</th>
<th>sname</th>
<th>rating</th>
<th>age</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>dustin</td>
<td>7</td>
<td>45.0</td>
</tr>
<tr>
<td>31</td>
<td>lubber</td>
<td>8</td>
<td>55.5</td>
</tr>
<tr>
<td>58</td>
<td>rusty</td>
<td>10</td>
<td>35.0</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>bid</th>
<th>day</th>
</tr>
</thead>
<tbody>
<tr>
<td>22</td>
<td>101</td>
<td>10/10/96</td>
</tr>
<tr>
<td>58</td>
<td>103</td>
<td>11/12/96</td>
</tr>
</tbody>
</table>

\[ R \bowtie_{c} S = \sigma_{c} (R \times S) \]
Natural Join

- The natural join of two relations R and S is a set of pairs of tuples, one from R and one from S, that agree on whatever attributes are common to the schemas of R and S.
- The schema for the result contains the union of the attributes of R and S. (so duplicate cols. are dropped)
- Assume the schemas R(A,B, C) and S(B, C,D)

RA: \[ R \bowtie S \]

SQL: \[
SELECT R.A, R.B, R.C, S.D \\
FROM R, S \\
WHERE R.B = S.B AND R.C = S.C;
\]
Natural Join: Nit-picking

- What if R and S have not attributes in common?

  natural join → cartesian product

- Some (like Oracle) provide a special single NATURAL JOIN operator, but some (like IBM DB2) don’t.
  - So assume there is no special SQL natural join operator
Operators so far

- **Remove parts of single relations**
  - Projection: \( \pi_{(A,B)}(R) \) and SELECT A, B FROM R
  - Selection: \( \sigma_C(R) \) and SELECT * FROM R WHERE C
  - Combining Projection and Selection:
    - \( \pi_{(A,B)}(\sigma_C(R)) \)
    - SELECT A, B FROM R WHERE C
Operations so far

- **Set operations**
  - R and S must have the same attributes, same attribute types, and same order of attributes
  - Union: $R \cup S$ and $(R) \text{ UNION} (S)$
  - Intersection: $R \cap S$ and $(R) \text{ INTERSECT} (S)$
  - Difference: $R - S$ and $(R) \text{ EXCEPT} (S)$
Operations so far

- **Combine the tuples of two relations**
  - Cartesian Product: $R \times S$, .... FROM $R$, $S$ ..... 
  - Theta Join: $R \bowtie_C S$, .... FROM $R$, $S$ WHERE $C$
  - Natural Join: $R \bowtie S$
Ordering

- find student records, sorted in name order
  - select *
  - from student
  - order by name asc

- asc is the default
Ordering

- find student records, sorted in name order; break ties by reverse ssn
  - select *
  - from student
  - order by name, ssn desc
Rename op.

- Q: why? $\rho_{\text{AFTER}}(\text{BEFORE})$
- A: shorthand; self-joins; ...
- for example, find the grand-parents of ‘Tom’, given PC (parent-id, child-id)
Rename op.

- PC (parent-id, child-id)

\[ PC \supset\!
\supset PC \]

<table>
<thead>
<tr>
<th>PC</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p-id</td>
<td>c-id</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>Tom</td>
<td></td>
</tr>
<tr>
<td>Peter</td>
<td>Mary</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td>Tom</td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>PC</th>
<th></th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>p-id</td>
<td>c-id</td>
<td></td>
</tr>
<tr>
<td>Mary</td>
<td>Tom</td>
<td></td>
</tr>
<tr>
<td>Peter</td>
<td>Mary</td>
<td></td>
</tr>
<tr>
<td>John</td>
<td>Tom</td>
<td></td>
</tr>
</tbody>
</table>
Rename op.

- first, WRONG attempt:
  \[ PC \bowtie PC \]
  (why? how many columns?)
- Second WRONG attempt:
  \[ PC_{PC.c-id=PC.p-id} \bowtie PC \]
Rename op.

- we clearly need two different names for the same table - hence, the ‘rename’ op.

\[ \rho_{PC_1} (PC) \bowtie_{PC_1.c-id=PC.p-id} PC \]
Disambiguation and Renaming

**RA:** give R the name S; R has n attributes, which are \( \rho_R \left( A_1, A_2, \ldots, A_n \right) \) in S

**SQL:** Use the **AS** keyword in the **FROM** clause: Students AS Students1 renames Students to Students1.

**SQL:** Use the **AS** keyword in the **SELECT** clause to rename attributes.
Disambiguation and Renaming

- Name pairs of students who live at the same address: \textit{Students} (Name, Address)

\textbf{RA:} \[ \pi_{S1.Name,S2.Name} \left( \sigma_{S1.Address=S2.Address} \left( \rho_{S1}(Students) \times \rho_{S2}(Students) \right) \right) \]

\textbf{SQL:} SELECT S1.name, S2.name
FROM Students AS S1, Students AS S2
WHERE S1.address = S2.address
Disambiguation and Renaming

- Name pairs of students who live at the same address:

  SQL:  SELECT S1.name, S2.name
        FROM Students AS S1, Students AS S2
        WHERE S1.address = S2.address

- Are these correct?

- No !!! the result includes tuples where a student is paired with himself/herself

- Solution: Add the condition S1.name <> S2.name.
Division

- Rarely used, but powerful.
- Example: find suspicious suppliers, ie., suppliers that supplied all the parts in A_BOMB
Division

\[
\begin{array}{c|c}
\text{SHIPMENT} & \text{ABOMB} \\
\hline
s# & p# \\
s1 & p1 \\
s2 & p1 \\
s1 & p2 \\
s3 & p1 \\
s5 & p3 \\
\end{array}
\div
\begin{array}{c|c}
p# & \\
p1 & \\
p2 & \\
\end{array}
=\begin{array}{c|c}
\text{BAD_S} & \\
s# & \\
s1 & \\
\end{array}
\]
Division

- Observations: ~reverse of cartesian product
- It can be derived from the 5 fundamental operators (‼)
- How?
Division

- **Answer:**

\[
r \div s = \pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times s) - r]
\]

- **Observation:** find ‘good’ suppliers, and subtract! *(double negation)*
**Answer:**

\[ r \div s = \pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times s) - r] \]

- \( R \): attributes of \( r \)
- \( S \): attributes of \( s \)

**Observation:** find ‘good’ suppliers, and subtract! (double negation)
**Division**

- **Answer:**

\[ r \div s = \pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times s) - r] \]

### SHIPMENT

<table>
<thead>
<tr>
<th>s#</th>
<th>p#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
</tr>
<tr>
<td>s3</td>
<td>p1</td>
</tr>
<tr>
<td>s5</td>
<td>p3</td>
</tr>
</tbody>
</table>

### ABOMB

<table>
<thead>
<tr>
<th>p#</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

### BAD_S

<table>
<thead>
<tr>
<th>s#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
</tbody>
</table>

**All suppliers**

**All bad parts**
Division

Answer:

\[ r \div S = \pi_{(R-S)}(r) - \pi_{(R-S)}[(r \times S) - r] \]

\[ \begin{array}{c|c}
 s\# & p\# \\
 s1 & p1 \\
 s2 & p1 \\
 s1 & p2 \\
 s3 & p1 \\
 s5 & p3 \\
\end{array} \]

\[ \begin{array}{c|c}
 p\# & s\# \\
 p1 & s1 \\
 p2 & \text{BAD_S} \\
\end{array} \]

all possible suspicious shipments
Division

**Answer:**

\[
\pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times s) - r]
\]

- **SHIPMENT**
<table>
<thead>
<tr>
<th>s#</th>
<th>p#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
<td>p1</td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
</tr>
<tr>
<td>s3</td>
<td>p1</td>
</tr>
<tr>
<td>s5</td>
<td>p3</td>
</tr>
</tbody>
</table>

- **ABOMB**
<table>
<thead>
<tr>
<th>p#</th>
</tr>
</thead>
<tbody>
<tr>
<td>p1</td>
</tr>
<tr>
<td>p2</td>
</tr>
</tbody>
</table>

- **BAD_S**
<table>
<thead>
<tr>
<th>s#</th>
</tr>
</thead>
<tbody>
<tr>
<td>s1</td>
</tr>
</tbody>
</table>

all possible suspicious shipments that didn’t happen
Division

- Answer:

\[ r \div S = \pi_{(R-S)}(r) - \pi_{(R-S)}[(\pi_{(R-S)}(r) \times S) - r] \]

<table>
<thead>
<tr>
<th>SHIPMENT</th>
<th></th>
<th>ABOMB</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s#</td>
<td>p#</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>p1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s2</td>
<td>p1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td>p2</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s3</td>
<td>p1</td>
<td></td>
<td></td>
</tr>
<tr>
<td>s5</td>
<td>p3</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>BAD_S</th>
<th></th>
</tr>
</thead>
<tbody>
<tr>
<td>s#</td>
<td></td>
</tr>
<tr>
<td>s1</td>
<td></td>
</tr>
</tbody>
</table>

all suppliers who missed at least one suspicious shipment,

i.e.: ‘good’ suppliers
Quick Quiz: Independence of Operators

\[ R \cap S = R - (R - S) \]
\[ R \bowtie_C = \sigma_C (R \times S) \]
\[ R \bowtie S = ?? \]
Quick Quiz: Independence of Operators

Suppose R and S share the attributes A1,A2,..An

Let L be the list of attributes in R \ Union list of attributes in S (so no duplicate attributes)

Let C be the condition


\[ R \bowtie S = \pi_L(\sigma_C(R \times S)) \]
Linear Notation for Relational Algebra

- Relational algebra expressions can become very long.

- Use linear notation to store results of intermediate expressions.
  - A relation name and a parenthesized list of attributes for that relation
  - Use Answer as the conventional name for the final result
  - The assignment symbol :=
  - Any expression in relational algebra on the right
Example of Linear Notation

- Name pairs of students who live at the same address.
- Normal expression:

\[
\pi S_1.Name, S_2.Name

\sigma S_1.Address = S_2.Address

(\rho_{S_1}(Students) \times \rho_{S_2}(Students))
\]
Example of Linear Notation

- Normal expression:

\[
\pi_{S1.Name, S2.Name} \\
\sigma_{S1.Address = S2.Address} \\
(\rho_{S1}(Students) \times \rho_{S2}(Students))
\]

- Linear Notation:

\[
\text{Pairs}(P1, N1, A1, P2, N2, A2) := \rho_{S1}(Students) \times \rho_{S2}(Students) \\
\text{Matched}(P1, N1, A1, P2, N2, A2) := \sigma_{A1=A2}(\text{Pairs}(P1, N1, A1, P2, N2, A2)) \\
\text{Answer}(\text{Name1}, \text{Name2}) := \pi_{N1,N2}(\text{Matched}(P1, N1, A1, P2, N2, A2))
\]
Interpreting Queries Involving Multiple Relations

- SELECT A, B FROM R, S WHERE C;
- Nested loops:
  for each tuple t1 in R
    for each tuple t2 in S
      if the attributes in t1 and t2 satisfy C
        output the tuples involving attributes A and B
Interpreting Queries Involving Multiple Relations

- SELECT A, B FROM R, S WHERE C;
- Conversion to relational algebra:

\[
\pi_{A,B}(\sigma_C(R \times S))
\]

Compute \( R \times S \)
Apply selection operator \( \sigma() \) to \( R \times S \)
Project the result tuples to attributes A and B