Relational Algebra and SQL

Chapter 5

Topics

• Relational Algebra
  – Introduction
  – Basic Operators
  – Derived Operators
• SQL

Relational Query Languages

• Languages for describing queries on a relational database
• Structured Query Language (SQL)
  – Predominant application-level query language
  – Declarative
• Relational Algebra
  – Intermediate language used within DBMS
  – Procedural

What is an Algebra?

• A language based on operators and a domain of values
• Operators map values taken from the domain into other domain values
• Hence, an expression involving operators and arguments produces a value in the domain
• When the domain is a set of all relations (and the operators are as described later), we get the relational algebra
• We refer to the expression as a query and the value produced as the query result
Relational Algebra

- **Domain**: set of relations
- **Basic operators**: select, project, union, set difference, Cartesian product
- **Derived operators**: set intersection, division, join
- **Procedural**: Relational expression specifies query by describing an algorithm (the sequence in which operators are applied) for determining the result of an expression

The Role of Relational Algebra in a DBMS

Topics

- Relational Algebra
  - Introduction
  - Basic Operators
  - Derived Operators
- SQL

Select Operator

- Produce table containing subset of rows of argument table satisfying condition
  \[ \sigma_{condition}(relation) \]
- Example:

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>
Selection Condition

- Operators: $<, \leq, \geq, >, =, \neq$

1. Simple selection condition:
   - $<\text{attribute}>\operatorname{operator}<\text{constant}>$
   - $<\text{attribute}_1>\operatorname{operator}<\text{attribute}_2>$
2. $<\text{condition}>\operatorname{AND}<\text{condition}>$
3. $<\text{condition}>\operatorname{OR}<\text{condition}>$
4. $\neg<\text{condition}>$

Selection Condition - Examples

- $\sigma\text{Id}>3000\text{ OR }\text{Hobby}='\text{hiking'}$(Person)
- $\sigma\text{Id}>3000\text{ AND }\text{Id}<3999$(Person)
- $\sigma\neg(\text{Hobby}='\text{hiking'})(\text{Person})$
- $\sigma\text{Hobby}='\text{hiking'}$(Person)
- Conditions of the form $\text{expression}_1\operatorname{oper}\text{expression}_2$ are also allowed
  - Salary > (MgrSalary*2)
  - In the above example, Salary & MgrSalary are attributes of the same relation

Project Operator

- Produces table containing subset of columns of argument table
  \[ \pi_{\text{list}}(\text{relation}) \]
- Example:
  \[
  \begin{array}{ccc}
  \text{Id} & \text{Name} & \text{Address} & \text{Hobby} \\
  1123 & \text{John} & 123 \text{ Main} & \text{stamps} \\
  1123 & \text{John} & 123 \text{ Main} & \text{coins} \\
  5556 & \text{Mary} & 7 \text{ Lake Dr} & \text{hiking} \\
  9876 & \text{Bart} & 5 \text{ Pine St} & \text{stamps} \\
  \end{array}
  \]
  \[
  \begin{array}{ll}
  \pi_{\text{Name, Hobby}}(\text{Person}) \\
  \text{Name} & \text{Hobby} \\
  \text{John} & \text{stamps} \\
  \text{John} & \text{coins} \\
  \text{Mary} & \text{hiking} \\
  \text{Bart} & \text{stamps} \\
  \end{array}
  \]

Project Operator

- Example:
  \[
  \begin{array}{cccc}
  \text{Id} & \text{Name} & \text{Address} & \text{Hobby} \\
  1123 & \text{John} & 123 \text{ Main} & \text{stamps} \\
  1123 & \text{John} & 123 \text{ Main} & \text{coins} \\
  5556 & \text{Mary} & 7 \text{ Lake Dr} & \text{hiking} \\
  9876 & \text{Bart} & 5 \text{ Pine St} & \text{stamps} \\
  \end{array}
  \]
  \[
  \begin{array}{ll}
  \pi_{\text{Name, Address}}(\text{Person}) \\
  \text{Name} & \text{Address} \\
  \text{John} & 123 \text{ Main} \\
  \text{Mary} & 7 \text{ Lake Dr} \\
  \text{Bart} & 5 \text{ Pine St} \\
  \end{array}
  \]

- Result is a table (no duplicates)
  - can have fewer tuples than the original
Relational Expressions
We can create Relational Expressions by composing relational operators
\[ \pi_{\text{Id, Name}} (\sigma_{\text{Hobby}=\text{stamps} \text{ OR Hobby}=\text{coins}} (\text{Person})) \]

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Address</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>stamps</td>
</tr>
<tr>
<td>1123</td>
<td>John</td>
<td>123 Main</td>
<td>coins</td>
</tr>
<tr>
<td>5556</td>
<td>Mary</td>
<td>7 Lake Dr</td>
<td>hiking</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>

Person

Set Operators
- Relation is a set of tuples, so set operations should apply: \( \cap, \cup, - \) (set difference)
- Result of combining two relations with a set operator is a relation => all its elements must be tuples having same structure
- Hence, scope of set operations limited to union compatible relations

Union Compatible Relations
- Two relations are union compatible if
  - Both have same number of columns
  - Names of attributes are the same in both
  - Attributes with the same name in both relations have the same domain
- Union compatible relations can be combined using union, intersection, and set difference

Example
Tables:
- Person (SSN, Name, Address, Hobby)
- Professor (Id, Name, Office, Phone)
are not union compatible
But
\[ \pi_{\text{Name}} (\text{Person}) \text{ and } \pi_{\text{Name}} (\text{Professor}) \]
are union compatible so
\[ \pi_{\text{Name}} (\text{Person}) - \pi_{\text{Name}} (\text{Professor}) \]
makes sense
Schema for Student Registration System

Student (Id, Name, Addr, Status)
Professor (Id, Name, DeptId)
Course (DeptId, CrsCode, CrsName, Descr)
Transcript (StudId, CrsCode, Semester, Grade)
Teaching (ProfId, CrsCode, Semester)
Department (DeptId, Name)

Example 2

<table>
<thead>
<tr>
<th>TEACHING</th>
<th>ProfId</th>
<th>CrsCode</th>
<th>Semester</th>
</tr>
</thead>
<tbody>
<tr>
<td>0059065321</td>
<td>MAT123</td>
<td>F1994</td>
<td></td>
</tr>
<tr>
<td>121232943</td>
<td>EE101</td>
<td>S1991</td>
<td></td>
</tr>
<tr>
<td>6655666777</td>
<td>CS305</td>
<td>F1995</td>
<td></td>
</tr>
<tr>
<td>064297531</td>
<td>MAT123</td>
<td>F1994</td>
<td></td>
</tr>
<tr>
<td>101202303</td>
<td>CS215</td>
<td>S1997</td>
<td></td>
</tr>
<tr>
<td>900120450</td>
<td>MAT123</td>
<td>S1998</td>
<td></td>
</tr>
<tr>
<td>121232943</td>
<td>EE101</td>
<td>F1995</td>
<td></td>
</tr>
<tr>
<td>101202303</td>
<td>CS305</td>
<td>S1996</td>
<td></td>
</tr>
<tr>
<td>900120450</td>
<td>MAT123</td>
<td>F1997</td>
<td></td>
</tr>
<tr>
<td>783452188</td>
<td>MAT123</td>
<td>F1997</td>
<td></td>
</tr>
<tr>
<td>0059065321</td>
<td>MAT123</td>
<td>F1997</td>
<td></td>
</tr>
</tbody>
</table>

Example 2 (cont’d)

<table>
<thead>
<tr>
<th>TRANSSCRIPT</th>
<th>StudId</th>
<th>CrsCode</th>
<th>Semester</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>0666666066</td>
<td>MAT123</td>
<td>F1994</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>0666666066</td>
<td>EE101</td>
<td>S1996</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>0666666066</td>
<td>MAT123</td>
<td>F1997</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>907545321</td>
<td>CS305</td>
<td>F1995</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>007545321</td>
<td>MAT123</td>
<td>F1994</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>123454321</td>
<td>CS215</td>
<td>S1997</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>123454321</td>
<td>CS305</td>
<td>S1996</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>123454321</td>
<td>MAT123</td>
<td>S1996</td>
<td>C</td>
<td></td>
</tr>
<tr>
<td>023456789</td>
<td>EE101</td>
<td>F1996</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>023456789</td>
<td>CS305</td>
<td>S1996</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1111111111</td>
<td>EE101</td>
<td>F1997</td>
<td>A</td>
<td></td>
</tr>
<tr>
<td>1111111111</td>
<td>MAT123</td>
<td>F1997</td>
<td>B</td>
<td></td>
</tr>
<tr>
<td>1111111111</td>
<td>MAT123</td>
<td>F1997</td>
<td>B</td>
<td></td>
</tr>
</tbody>
</table>
**Cartesian Product**

- If \( R \) and \( S \) are two relations, \( R \times S \) is the set of all concatenated tuples \(<x,y>\), where \( x \) is a tuple in \( R \) and \( y \) is a tuple in \( S \)
  - \( R \) and \( S \) need not be union compatible
- \( R \times S \) is expensive to compute:
  - Factor of two in the size of each row
  - Quadratic in the number of rows

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>x2</td>
<td>y1</td>
<td>y2</td>
</tr>
<tr>
<td>x3</td>
<td>x4</td>
<td>y3</td>
<td>y4</td>
</tr>
</tbody>
</table>

**Example**

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>11223546</td>
<td>Smith, Gary</td>
</tr>
<tr>
<td>033456789</td>
<td>Simpson, Homer</td>
</tr>
<tr>
<td>007541201</td>
<td>Simpson, Barz</td>
</tr>
</tbody>
</table>

A subset of \( \pi_{\text{Name}}(\text{STUDENT}) \)

<table>
<thead>
<tr>
<th>Id</th>
<th>Dept/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>500064477</td>
<td>CS</td>
</tr>
<tr>
<td>100201200</td>
<td>CS</td>
</tr>
</tbody>
</table>

A subset of \( \pi_{\text{Dept/C}}(\text{PROFESSOR}) \)

<table>
<thead>
<tr>
<th>STUDENT</th>
<th>Id</th>
<th>Name</th>
<th>DEPT/C</th>
</tr>
</thead>
<tbody>
<tr>
<td>11223546</td>
<td>Smith, Gary</td>
<td>585656777</td>
<td>CS</td>
</tr>
<tr>
<td>11223546</td>
<td>Smith, Gary</td>
<td>101232000</td>
<td>CS</td>
</tr>
<tr>
<td>023456789</td>
<td>Simpson, Homer</td>
<td>589969777</td>
<td>CS</td>
</tr>
<tr>
<td>033456789</td>
<td>Simpson, Homer</td>
<td>101232000</td>
<td>CS</td>
</tr>
<tr>
<td>090541201</td>
<td>Simpson, Barz</td>
<td>585656777</td>
<td>CS</td>
</tr>
<tr>
<td>090754120</td>
<td>Simpson, Barz</td>
<td>101232000</td>
<td>CS</td>
</tr>
</tbody>
</table>

Their Cartesian product

**Renaming**

- Result of expression evaluation is a relation
- Attributes of relation must have distinct names. This is not guaranteed with Cartesian product
  - e.g., suppose in previous example there is an \( Id \) attribute in both \( \text{Student} \) and \( \text{Professor} \)
- Renaming operator tidies this up. To assign the names \( A_1, A_2, \ldots, A_n \) to the attributes of the \( n \) column relation produced by expression expr use \( expr[A_1, A_2, \ldots, A_n] \)

<table>
<thead>
<tr>
<th>A</th>
<th>B</th>
<th>C</th>
<th>D</th>
</tr>
</thead>
<tbody>
<tr>
<td>x1</td>
<td>x2</td>
<td>y1</td>
<td>y2</td>
</tr>
<tr>
<td>x3</td>
<td>x4</td>
<td>y3</td>
<td>y4</td>
</tr>
</tbody>
</table>

\( R \times S \)
Topics

• Relational Algebra
  – Introduction
  – Basic Operators
  – Derived Operators
• SQL

Join

A (general or theta) join of $R$ and $S$ is the expression

$$R \bowtie_{\text{join-condition}} S$$

where join-condition is a conjunction of terms:

$$A_i \text{ oper } B_i$$

in which $A_i$ is an attribute of $R$; $B_i$ is an attribute of $S$; and oper is one of $=, <, >, \geq, \neq, \leq$.

The meaning is:

$$\sigma_{\text{join-condition} \; \prime} (R \times S)$$

where join-condition and join-condition' are the same, except for possible renamings of attributes (next).

Join and Renaming

• Problem: $R$ and $S$ might have attributes with the same name – in which case the Cartesian product is not defined

• Solutions:
  1. Rename attributes prior to forming the product and use new names in join-condition’.
  2. Qualify common attribute names with relation names (thereby disambiguating the names). For instance: Transcript.CrsCode or Teaching.CrsCode
     - This solution is nice, but doesn’t always work: consider
       $$R \bowtie_{\text{join-condition}} R$$
     In $R.A$, how do we know which $R$ is meant?

Theta Join – Example

Emp(Name,Id,MngrId,Salary)
Mngr(Name,Id,Salary)

Output the names of all employees that earn more than their managers

$$\pi_{\text{Emp.Name}} (\text{Emp} \bowtie_{\text{MngrId=Mngr.Id AND Emp.Salary>Mngr.Salary}} \text{Mngr.Name, Mgr.Id, Mgr.Salary})$$

The join yields a table with attributes:

Emp.Name, Emp.Id, Emp.Salary, Emp.MngrId
Mngr.Name, Mgr.Id, Mgr.Salary
Equijoin Join - Example

Equijoin: Join condition is a conjunction of equalities.

\[ \pi_{\text{Name}, \text{CrsCode}}(\text{Student}) \bowtie_{\text{Id} = \text{StudId}} \sigma_{\text{Grade} = 'A'}(\text{Transcript}) \]

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
<th>Addr</th>
<th>Status</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>John</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>222</td>
<td>Mary</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>333</td>
<td>Bill</td>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>444</td>
<td>Joe</td>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>StudId</th>
<th>CrsCode</th>
<th>Sem</th>
<th>Grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>CSE305</td>
<td>S00</td>
<td>B</td>
</tr>
<tr>
<td>222</td>
<td>CSE306</td>
<td>S99</td>
<td>A</td>
</tr>
<tr>
<td>333</td>
<td>CSE304</td>
<td>F99</td>
<td>A</td>
</tr>
</tbody>
</table>

The equijoin is used very frequently since it combines related data in different relations.

Natural Join

- Special case of equijoin:
  - join condition equates all and only those attributes with the same name (condition doesn’t have to be explicitly stated)
  - duplicate columns eliminated from the result

\[ \pi_{\text{StudId, CrsCode, Sem, Grade}}(\text{Transcript}) \bowtie \pi_{\text{ProfId, CrsCode, Sem}}(\text{Teaching}) \]

\[ \pi_{\text{StudId, CrsCode, Sem, Grade, ProfId}}(\text{Transcript} \bowtie \text{Teaching}) \]

Natural Join (cont’d)

- More generally:

\[ R \bowtie S = \pi_{\text{attr-list}}(\sigma_{\text{join-cond}}(R \times S)) \]

where

- attr-list = attributes (R) \cup attributes (S) (duplicates are eliminated) and join-cond has the form:
  \[ A_1 = A_1 \text{ AND ... AND } A_n = A_n \]

- \{A_1 ... A_n\} = attributes(R) \cap attributes(S)

Natural Join Example

- List all Ids of students who took at least two different courses:

\[ \pi_{\text{StudId}}(\sigma_{\text{CrsCode} \neq \text{CrsCode2}}(\text{Transcript} \bowtie \text{Transcript} [\text{StudId, CrsCode2, Sem2, Grade2}])) \]

We don’t want to join on CrsCode, Sem, and Grade attributes, hence renaming!
Natural Join for Union Compatible Relations

- If there are 2 relations that are union compatible, the result would be the same as obtained using a well known set operator. Which one?
  \[ R \bowtie S = \pi_{\text{attr-list}} ( \sigma_{\text{join-cond}} (R \times S) ) \]
- \( R \) & \( S \) have same attributes \( A_1, \ldots, A_n \)
- All attributes are included in the project attribute list
- Join condition also includes all the attributes:
  \[ A_1 = A_1 \text{ AND } \ldots \text{ AND } A_n = A_n \]
- Only tuples that are both in \( R \) & \( S \) will be in the result
- In other words, same as intersection

Outer Join

- When two relations are joined, tuples that do not match are eliminated from the result
- E.g.,
  \[ R \bowtie S = \pi_{\text{attr-list}} ( \sigma_{\text{join-cond}} (R \times S) ) \]
- A join between \( \text{Dept} & \text{Professor} \) to list the name of each professor and his/her dept would not include professors currently not assigned to a dept
- Outer join, left outer join, right outer join
  - Operators for cases where tuples that do not match are to be included in the result

Definition of Outer Join

- (Full) outer join between relations \( r \) and \( s \) contains tuples that:
  1. Appear in regular join of \( r \) and \( s \)
  2. Tuple of \( r \) that don’t join any tuple in \( s \). These tuples have NULL for attributes belonging to \( s \)
  3. Tuples of \( s \) that don’t join any tuple in \( r \). Again NULL for attributes belonging to \( r \)
- Left outer join does not include tuples in category 3
- Right outer join does not include tuples in category 2

Example: Outer Joins
Example: Left & Right Outer Joins

<table>
<thead>
<tr>
<th>Supplier</th>
<th>PartNumber</th>
<th>PartNumber2</th>
<th>PartName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ace Inc.</td>
<td>P120</td>
<td></td>
<td>10-32 screw</td>
</tr>
<tr>
<td></td>
<td>M30</td>
<td>P120</td>
<td>10-32 screw</td>
</tr>
<tr>
<td>Electronics 2000</td>
<td>NULL</td>
<td></td>
<td>NULL</td>
</tr>
</tbody>
</table>

Left outer join SUPPLIER on PartNumber = PARTS

<table>
<thead>
<tr>
<th>Supplier</th>
<th>PartNumber</th>
<th>PartNumber2</th>
<th>PartName</th>
</tr>
</thead>
<tbody>
<tr>
<td>Ace Inc.</td>
<td>P120</td>
<td></td>
<td>10-32 screw</td>
</tr>
<tr>
<td></td>
<td>M30</td>
<td></td>
<td>10-32 screw</td>
</tr>
<tr>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
<td>NULL</td>
</tr>
</tbody>
</table>

Right outer join SUPPLIER on PartNumber = PARTS

Division

- Goal: Produce the tuples in one relation, r, that match all tuples in another relation, s
  - \( r(A_1, \ldots A_n, B_1, \ldots B_m) \)
  - \( s(B_1, \ldots B_m) \)
  - \( r/s \), with attributes \( A_1, \ldots A_n \) is the set of all tuples \( \langle a \rangle \) such that for every tuple \( \langle b \rangle \) in s, \( \langle a, b \rangle \) is in r
- Can be expressed in terms of projection, set difference, and cross-product

Division (cont’d)

- What are the courses taught by every professor in computer science?
Division - Example

- List the IDs of students who have passed all courses that were taught in spring 2004
- **Numerator:**
  - StudId and CrsCode for every course passed by every student:
    \[ \pi_{\text{StudId}, \text{CrsCode}} (\sigma_{\text{Grade} \neq 'F'} (\text{Transcript})) \]
- **Denominator:**
  - CrsCode of all courses taught in spring 2004
    \[ \pi_{\text{CrsCode}} (\sigma_{\text{Semester}='S2004'} (\text{Teaching})) \]
- Result is numerator/denominator

Query Sublanguage of SQL

```
SELECT C.\text{CrsName}
FROM Course C
WHERE C.\text{DeptId} = 'CS'
```

- Tuple variable (table alias) C ranges over rows of Course.
- Evaluation strategy:
  - FROM clause produces Cartesian product of listed tables
  - WHERE clause assigns rows to C in sequence and produces table containing only rows satisfying condition
  - SELECT clause retains listed columns
- (Approx.) equivalent to:
  \[ \pi_{\text{CrsName}} (\sigma_{\text{DeptId}='CS'} (\text{Course})) \]

Join Queries

```
SELECT C.\text{CrsName}
FROM Course C, Teaching T
WHERE C.\text{CrsCode} = T.\text{CrsCode} AND T.\text{Semester} = 'S2004'
```

- List CS courses taught in S2004
- Tuple variables clarify meaning
- Join condition “C.\text{CrsCode} = T.\text{CrsCode}”
  - relates facts to each other
- Selection condition “T.\text{Semester} = ‘S2004’”
  - eliminates irrelevant rows
- Equivalent (using natural join) to:
  \[ \pi_{\text{CrsName}} \; (\sigma_{\text{Sem='S2004'}} (\text{Course} \bowtie \text{Teaching})) \]
  \[ \pi_{\text{CrsName}} (\sigma_{\text{Sem='S2004'}} (\text{Course} \bowtie \text{Teaching})) \]
Correspondence Between SQL and Relational Algebra

SELECT C.CrsName
FROM Course C, Teaching T
WHERE C.CrsCode = T.CrsCode AND T.Semester = 'S2004'

Also equivalent to:
\[ \pi_{\text{CrsName}} \sigma_{C.\text{CrsCode}=T.\text{CrsCode} \text{ AND} \text{ Semester} = 'S2004'} \]
(Course [C.\text{CrsCode}, DeptId, CrsName, Desc] \times Teaching [ProfId, T.\text{CrsCode}, Semester])

• This is the simplest evaluation algorithm for SELECT.
• Relational algebra expressions are procedural.
  ➢ Which of the two equivalent expressions is more easily evaluated?

Self-join Queries

Find Ids of all professors who taught at least two courses in the same semester:

SELECT T1.ProfId
FROM Teaching T1, Teaching T2
WHERE T1.ProfId = T2.ProfId
AND T1.Semester = T2.Semester
AND T1.CrsCode <> T2.CrsCode

Tuple variables are essential in this query!

Equivalent to:
\[ \pi_{\text{ProfId}} (\sigma_{T1.\text{CrsCode} \neq T2.\text{CrsCode}} (\text{Teaching}[\text{ProfId}, T1.\text{CrsCode}, \text{Semester}]) \times \text{Teaching}[\text{ProfId}, T2.\text{CrsCode}, \text{Semester}])) \]

Duplicates

• Duplicate rows not allowed in a relation
• However, duplicate elimination from query result is costly and not done by default; must be explicitly requested:

SELECT DISTINCT ..... FROM .....
### Set Operators

- SQL provides **UNION**, **EXCEPT** (set difference), and **INTERSECT** for union compatible tables.
- E.g.: Find all professors in the CS Department and all professors that have taught CS courses

```sql
(SELECT P.Name
 FROM Professor P, Teaching T
 WHERE P.Id=T.ProfId AND T.CrsCode LIKE 'CS%')
UNION
(SELECT P.Name
 FROM Professor P
 WHERE P.DeptId = 'CS')
```

### Comments on UNION

- In Oracle, Union compatibility does not require that the names of the attributes be the same in the two relations
  - Number of attributes and the corresponding data types must match
- Question: does name incompatibility open up a theoretical problem?
  - Yes. If attribute is named A in one relation and B in another relation, what attribute name should be used in the result of the UNION?
  - In Oracle, the attribute name in the 1st relation in the UNION is used in the result

### Other Operators

- Oracle provides a **MINUS** operator in place of **EXCEPT** operator
- To match a string pattern SQL provides the **LIKE** predicate
  - “%” symbol matches one or more characters

### Topics

- **Relational Algebra**
- **SQL**
  - Simple & Set Queries
  - Nested queries
- **Aggregation**
- Views, NULL Values
- Modifying relational data
Nested Queries

- List all courses that were not taught in S2004
  
  SELECT C.CrsName
  FROM Course C
  WHERE C.CrsCode NOT IN
    (SELECT T.CrsCode --subquery
     FROM Teaching T
     WHERE T.Sem = 'S2004')

- Evaluation strategy: subquery evaluated once to produces set of courses taught in S2004. Each row (as C) tested against this set

Correlated Nested Queries

- Output a row <prof, dept> if prof has taught a course in dept
  
  SELECT P.Name, D.Name --outer query
  FROM Professor P, Department D
  WHERE P.Id IN
    -- set of all ProfId's who have taught a course in D.DeptId
    (SELECT T.ProfId --subquery
     FROM Teaching T, Course C
     WHERE T.CrsCode = C.CrsCode AND
     C.DeptId = D.DeptId --correlation
    )

Correlated Nested Queries (con’t)

- Tuple variables T and C are local to subquery
- Tuple variables P and D are global to subquery
- Correlation: subquery uses a global variable, D
- The value of D.DeptId parameterizes an evaluation of the subquery
- Subquery must (at least) be re-evaluated for each distinct value of D.DeptId

- Correlated queries can be expensive to evaluate
- In general, DBMS query optimizers aren’t that good at optimizing nested queries. Avoid if a simpler non-nested query can be written

EXISTS Operator

- Sometimes it is necessary to check if a nested subquery returns no answers
- EXISTS operator: returns TRUE if a set is non-empty
- e.g., Find all students who never took a CS course
  
  SELECT S.Id
  FROM Student S
  WHERE NOT EXISTS
    -- All CS courses taken by S.Id
    (SELECT T.CrsCode
     FROM Transcript T
     WHERE T.CrsCode LIKE 'CS%' AND
     T.StudentId = S.Id --correlation
    )

- Subquery must (at least) be re-evaluated for each distinct value of D.DeptId
Division in SQL

- **Query type**: Find the subset of items in one set that are related to all items in another set
- **Example**: Find professors who taught courses in all departments
  
  - Why does this involve division?

\[ \pi_{\text{ProfId}, \text{DeptId}}(\text{Teaching} \bowtie \text{Course}) / \pi_{\text{DeptId}}(\text{Department}) \]

Division in SQL

- **Strategy for implementing division in SQL**:
  
  - Find set, A, of all departments in which a particular professor, \( p \), has taught a course
  
  - Find set, B, of all departments
  
  - Output \( p \) if \( A \supseteq B \), or, equivalently, if \( B - A \) is empty

Other Set Related Operators

- Set comparison operators
  
  - \( > \text{ALL} \)
  
  - \( >\text{ANY} \)

- Quantified predicates
  
  - FOR ALL
  
  - FOR SOME

- Read book for examples
Topics

- Relational Algebra
- SQL
  - Simple & Set Queries
  - Nested queries
  - Aggregation
  - Views, NULL Values
  - Modifying relational data

Aggregates

- Functions that operate on sets:
  - COUNT, SUM, AVG, MAX, MIN
- Produce numbers (not tables)
- Not part of relational algebra (but not hard to add)

SELECT COUNT(*)
FROM Professor P

SELECT MAX(Salary)
FROM Employee E

Aggregates (cont’d)

- Count the number of courses taught in S2004

SELECT COUNT(T.CrsCode)
FROM Teaching T
WHERE T.Semester = ‘S2004’

- But if multiple sections of same course are taught, use:

SELECT COUNT(DISTINCT T.CrsCode)
FROM Teaching T
WHERE T.Semester = ‘S2004’

Grouping

- But how do we compute the number of courses taught in S2004 per professor?

  - Strategy 1: Fire off a separate query for each professor:
    SELECT COUNT(T.CrsCode)
    FROM Teaching T
    WHERE T.Semester = ‘S2004’ AND T.ProfId = 123456789
    - Cumbersome
    - What if the number of professors changes? Add another query?

  - Strategy 2: Define a special grouping operator:
    SELECT T.ProfId, COUNT(T.CrsCode)
    FROM Teaching T
    WHERE T.Semester = ‘S2004’
    GROUP BY T.ProfId
GROUP BY

Attributes:
- student's Id
- avg grade
- number of courses

HAVING Clause

- Eliminates unwanted groups (analogous to WHERE clause, but works on groups instead of individual tuples)
- HAVING condition is constructed from attributes of GROUP BY list and aggregates on attributes not in that list

SELECT T.StudId, AVG(T.Grade) AS CumGpa, COUNT(*) AS NumCrs
FROM Transcript T
WHERE T.CrsCode LIKE 'CS%'
GROUP BY T.StudId
HAVING AVG(T.Grade) > 3.5

Evaluation of Group By with Having
Example

- Output the name of all seniors on with GPA > 3.5 and more than 90 credits

```sql
SELECT S.Id, S.Name
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND S.Status = 'senior'
GROUP BY S.Id
HAVING AVG(T.Grade) > 3.5 AND SUM(T.Credit) > 90
```

Aggregates: Proper and Improper Usage

**Ex1:** SELECT COUNT(T.CrsCode), T.ProfId
-- makes no sense (in the absence of GROUP BY clause)

**Ex2:** SELECT COUNT(*), AVG(T.Grade)
-- OK even without GROUP BY

**Ex3:** WHERE T.Grade > COUNT(SELECT ....)
-- aggregate cannot be applied to result of SELECT statement

ORDER BY Clause

- Causes rows to be output in a specified order
  - Note: ORDER BY can be used without GROUP BY

```sql
SELECT T.StudId, COUNT(*) AS NumCrs,
       AVG(T.Grade) AS CumGpa
FROM Transcript T
WHERE T.CrsCode LIKE 'CS%
GROUP BY T.StudId
HAVING AVG(T.Grade) > 3.5
ORDER BY DESC CumGpa, ASC StudId
```

Query Evaluation with GROUP BY, HAVING, ORDER BY

1. Evaluate FROM: produces Cartesian product, A, of tables in FROM list
2. Evaluate WHERE: produces table, B, consisting of rows of A that satisfy WHERE condition
3. Evaluate GROUP BY: partitions B into groups that agree on attribute values in GROUP BY list
4. Evaluate HAVING: eliminates groups in B that do not satisfy HAVING condition
5. Evaluate SELECT: produces table C containing a row for each group. Attributes in SELECT list limited to those in GROUP BY list and aggregates over group
6. Evaluate ORDER BY: orders rows of C
Topics

- Relational Algebra
- SQL
  - Simple & Set Queries
  - Nested queries
  - Aggregation
  - Views, NULL Values
  - Modifying relational data

Views

- Used as a relation, but rows are not physically stored.
  - The contents of a view are *computed* when it is used within an SQL statement
- View is the result of a SELECT statement over other views and *base relations*
- When used in an SQL statement, the view definition is substituted for the view name in the statement
  - As SELECT statement nested in FROM clause

View - Example

CREATE VIEW Gpa (StudentId, Gpa) AS
SELECT T.StudentId, AVG(T.Grade) FROM Transcript T
GROUP BY T.StudentId

SELECT S.Name, C.Gpa FROM Gpa C, Student S WHERE C.StudentId = S.StudentId AND C.Gpa > 3.5

View Benefits

- *Access Control*: Users not granted access to base tables. Instead they are granted access to the view of the database appropriate to their needs.
  - *External schema* is composed of views.
- View allows owner to provide SELECT access to a subset of columns (analogous to providing UPDATE and INSERT access to a subset of columns)
Views – Limiting Visibility

CREATE VIEW PartOfTranscript (StudId, CrsCode, Semester) AS
SELECT T.StudId, T.CrsCode, T.Semester -- limit columns
FROM Transcript T
WHERE T.Semester = 'S2004' -- limit rows

Give permissions to access data through view:
GRANT SELECT ON PartOfTranscript TO joe

This would have been analogous to:
GRANT SELECT (StudId, CrsCode, Semester)
ON Transcript TO joe

on regular tables, if SQL allowed attribute lists in GRANT SELECT

View Benefits (cont’d)

• Customization:
  – Users need not see full complexity of database
  – View creates the illusion of a simpler database
    customized to the needs of a particular category
    of users
  – A view is similar in many ways to a subroutine in standard programming
  – Can be reused in multiple queries

Dropping Views

• There is a DROP VIEW command just as there is a DROP TABLE command
• What happens if you drop a table on which a view is defined, or drop a view on which another view is defined?
• SQL standard provides RESTRICT and CASCADE options with DROP TABLE/VIEW commands
  – With RESTRICT, drop fails if some view depends on the table or view being dropped
  – With CASCADE, drop causes drop of dependent views
  – Oracle does not support CASCADE or RESTRICT options. The dependent objects become invalid

Materialized Views

• If a query is very popular and it is important to have good response time for the query, the results of the query can be cached by creating a Materialized View (MV)
• Unlike (non-materialized) views, a MV physically stores the results of a query
  – Query response time improved
  – Updates to base tables will make the result inconsistent
• Primary users are OLAP applications
  – Data warehouse: an (infrequently updated) database of data stored in a separate production OLTP database
    • Data warehouse typically consists of complex MVs
MV Example
CREATE MATERIALIZED VIEW LowerClassCount
BUILD IMMEDIATE
REFRESH ON COMMIT
ENABLE QUERY REWRITE
AS
SELECT Status, COUNT(*)
FROM Student
WHERE Status = 'F' OR Status = 'S'
GROUP BY Status
• The table Student is not used in answering the query:
  – SELECT Status, COUNT(*) FROM LowerClassCount
  – Instead the MV LowerClassCount is used

Maintaining Materialized Views
• Maintaining a MV is non-trivial
  – Example:
    SELECT Status, COUNT(*)
    FROM Student
    WHERE Status = 'F' OR Status = 'S'
    GROUP BY Status
  – If tuples get added from one or more base tables, the MV may or may not need to be updated
  – Same is true if tuples get deleted or updated

Maintenance Options
• Build method
  – How to build the MV the first time?
  – IMMEDIATE or DEFERRED
• Refresh mode
  – When to refresh the MV when the base tables get updated?
  – ON COMMIT or ON DEMAND or periodically
• Refresh method
  – INCREMENTAL or COMPLETE
• Query rewriting
  – Should queries that use the base tables be evaluated against the MV?

Example: Query Rewriting
• Consider the query:
  SELECT Status, COUNT(*)
  FROM Student
  WHERE Status = 'F'
• This query can be answered by using the MV LowerClassCount
• If Query Rewriting is enabled, then an attempt will be made to find a MV that answers the query
Null Values

- NULL a placeholder where an attribute does not have a value in some tuples of the table
- Question: what is the truth value when a comparison is made with a NULL?
- Example:
  - In table Transcript the value of column Grade can be null
  - If a particular tuple has value of Grade as NULL, then what is the truth value of the following two predicates:
    • T.Grade = 'A'
    • T.Grade <> 'A'
  - In 2-valued logic, one of the predicates must evaluate to true and the other must evaluate to false

3-Valued Logic

- In SQL the truth value of a predicate can be:
  - True (T), False (F) or Unknown (U)
- Conditions:
  - \( x \text{ op } y \) (where \( \text{op} \) is <, >, <=, =, etc.) has value unknown (U) when either \( x \) or \( y \) is null
  - Example:
    • WHERE T.cost > T.price
    • Would evaluate to U if either both of T.cost or T.price are NULL
- SQL provides a special operator IS NULL
  - E.g., WHERE T.grade IS NULL
  - Evaluates to T if the value is NULL, otherwise evaluates to F
  - Predicates with IS NULL operator are the only 2-valued predicates in SQL

Null Values: Arithmetic Exp & Aggregates

- Arithmetic expression:
  - \( x \text{ op } y \) (where \( \text{op} \) is +, -, *, etc.) has value NULL if \( x \) or \( y \) is NULL
  - E.g., WHERE (T.price/T.cost) > 2
- Aggregates:
  - COUNT counts NULLs like any other value
  - other aggregates ignore NULLs

Nulls: WHERE clause

- WHERE clause uses the three-valued logic – T, F, U – to filter rows. Portion of truth table:

<table>
<thead>
<tr>
<th>C1</th>
<th>C2</th>
<th>C1 AND C2</th>
<th>C1 OR C2</th>
</tr>
</thead>
<tbody>
<tr>
<td>T</td>
<td>U</td>
<td>U</td>
<td>T</td>
</tr>
<tr>
<td>F</td>
<td>U</td>
<td>F</td>
<td>U</td>
</tr>
<tr>
<td>U</td>
<td>U</td>
<td>U</td>
<td>U</td>
</tr>
</tbody>
</table>

- Rows are discarded if WHERE condition is F(alse) or U(known)
- Ex: WHERE T.CrsCode = ‘CS305’ AND T.Grade > 2.5
Topics

- Relational Algebra
- SQL
  - Simple & Set Queries
  - Nested queries
  - Aggregation
  - Views, NULL Values
  - Modifying relational data

Modifying Tables – Insert

- Inserting a single row into a table
  - Attribute list can be omitted if it is the same as in CREATE TABLE
  - but not omitting it is less error prone
  - NULL and DEFAULT values can be specified

```sql
INSERT INTO Transcript
(StudId, CrsCode, Semester, Grade)
VALUES (12345, 'CSE305', 'S2004', NULL)
```

Modifying Tables – Delete

- Similar to SELECT except:
  - No project list in DELETE clause
  - No Cartesian product in FROM clause (only 1 table name)
  - Rows satisfying WHERE clause (general form, including subqueries, allowed) are deleted instead of being output

```sql
DELETE FROM Transcript
WHERE T.Grade IS NULL AND T.Semester <> 'S2004'
```

Bulk Insertion

- Insert the rows output by a SELECT

```sql
CREATE TABLE DeansList (StudentId INTEGER,
Credits INTEGER,
CumulativeGpa FLOAT,
PRIMARY KEY StudentId)

INSERT INTO DeansList (StudentId, Credits, CumulativeGpa)
SELECT T.StudentId, 3 * COUNT(*), AVG(T.Grade)
FROM Transcript T
GROUP BY T.StudentId
HAVING AVG(T.Grade) > 3.5 AND COUNT(*) > 30
```
Modifying Data - Update

- Updates rows in a single table
- All rows satisfying WHERE clause (general form, including subqueries, allowed) are updated

```
UPDATE Employee E
SET E.Salary = E.Salary * 1.05
WHERE E.Department = 'R&D'
```

Updating Views

- Question: Since views look like tables to users, can they be updated?
- Answer: Yes – a view update changes the underlying base table to produce the requested change to the view

```
CREATE VIEW CsReg (StudId, CrsCode, Semester) AS
SELECT T.StudId, T.CrsCode, T.Semester
FROM Transcript T
WHERE T.CrsCode LIKE 'CS%' AND T.Semester='S2004'
```

Updating Views - Problem 1

```
INSERT INTO CsReg (StudId, CrsCode, Semester)
VALUES (1111, 'CSE305', 'S2004')
```

- Question: What value should be placed in attributes of underlying table that have been projected out (e.g., Grade)?
- Answer: NULL (assuming null allowed in the missing attribute) or DEFAULT

Updating Views - Problem 2

```
INSERT INTO CsReg (StudId, CrsCode, Semester)
VALUES (1111, 'ECO105', 'S2004')
```

- Problem: New tuple not in view
- Solution:
  - i) Allow insertion
  - ii) Reject insertion if CREATE VIEW statement had WITH CHECK OPTION clause
Updating Views - Problem 3

• Update to a view might not uniquely specify the change to the base table(s) that results in the desired modification of the view (ambiguity)

CREATE VIEW ProfDept (PrName, DeName) AS
SELECT P.Name, D.Name
FROM Professor P, Department D
WHERE P.DeptId = D.DeptId

Updating Views - Problem 3 (cont’d)

• Tuple <Smith, CS> can be deleted from ProfDept by any of the following ops:
  – Deleting row for Smith from Professor (but this is inappropriate if he is still at the University)
  – Deleting row for CS from Department (not what is intended)
  – Updating row for Smith in Professor by setting DeptId to null

Updating Views - Restrictions

• Updatable views are restricted to those in which
  – No Cartesian product in FROM clause
  – no aggregates, GROUP BY, HAVING
  – ...

For example, if we allowed:
CREATE VIEW AvgSalary (DeptId, Avg_Sal) AS
SELECT E.DeptId, AVG(E.Salary)
FROM Employee E
GROUP BY E.DeptId
then how do we handle:
UPDATE AvgSalary
SET Avg_Sal = 1.1 * Avg_Sal

Topics

• Relational Algebra
• SQL

→ Summary
Summary

- In this chapter we looked at 2 languages for describing queries on a relational database
  - Structured Query Language (SQL)
  - Relational Algebra
- We also looked at how a query can be expressed in both these languages
- SQL supports aggregate operators, nested queries, etc.
  - These types of queries can be expensive
  - Primarily used in OLAP applications
- In addition to views, DBMS also support Materialized Views
  - Primarily used in OLAP applications

Summary (cont’d)

- Other relational query languages also exist
- At the theoretical level two widely studied (declarative) languages are:
  - Tuple relational calculus
  - Domain relational calculus
- At the application level
  - QBE (Query By Example) is supported by some DBMS