Relational Algebra and SQL

CSE 305 – Principles of Database Systems
Paul Fodor
Stony Brook University
http://www.cs.stonybrook.edu/~cse305
Relational Query Languages

• Now that we know how to create a database, the next step is to learn how to query it to retrieve the information needed for some particular application.

• A database query language is a special-purpose programming language designed for retrieving information stored in a database.
Relational Query Languages

- Languages for describing queries on a relational databases:
  - **Structured Query Language (SQL)**
    - Predominant application-level query language
    - Declarative
  - **Relational Algebra**
    - Intermediate language used within DBMS
    - Procedural
    - The query optimizer converts the query algebraic expression into an equivalent faster query execution plan
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
Relational Algebra

- *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

- SQL Query
  - Parser
  - Parser
  - Query Optimizer
  - Code Generator
- Relational Algebra Expression
  - Query Execution Plan
  - Executable Code

(c) Pearson Education Inc. and Paul Fodor (CS Stony Brook)
Select Operator

- Produces a table containing subset of rows of argument table satisfying a condition

\[ \sigma_{\text{condition}} (\text{relation}) \]

- 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>

\[ \sigma_{\text{Hobby='stamps'}} (\text{Person}) \]

<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>9876</td>
<td>Bart</td>
<td>5 Pine St</td>
<td>stamps</td>
</tr>
</tbody>
</table>
Selection Condition

- Operators:  <, ≤, ≥, >, =, ≠
- Simple selection condition:
  - `<attribute>` operator `<constant>`
  - `<attribute>` operator `<attribute>`
- And Boolean expressions:
  - `<condition>` AND `<condition>`
  - `<condition>` OR `<condition>`
  - NOT `<condition>`
Selection Condition - Examples

- $\sigma \operatorname{Id}>3000 \ OR \ \operatorname{Hobby}=\text{`hiking'}$  \text{(Person)}
- $\sigma \operatorname{Id}>3000 \ \text{AND} \ \operatorname{Id}<3999$  \text{(Person)}
- $\sigma \ \text{NOT}(\operatorname{Hobby}=\text{`hiking'})$  \text{(Person)}
- $\sigma \ \operatorname{Hobby}\neq\text{`hiking'}$ \text{(Person)}
Project Operator

- Produces table containing subset of columns of argument table

\[ \pi_{\text{attribute list}}(\text{relation}) \]

- 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>

\[ \pi_{\text{Name},\text{Hobby}}(\text{Person}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Hobby</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>stamps</td>
</tr>
<tr>
<td>John</td>
<td>coins</td>
</tr>
<tr>
<td>Mary</td>
<td>hiking</td>
</tr>
<tr>
<td>Bart</td>
<td>stamps</td>
</tr>
</tbody>
</table>
Project Operator

- **Relational Algebra: No Duplicates!**

<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>

\[ \pi_{\text{Name}, \text{Address}}(\text{Person}) \]

<table>
<thead>
<tr>
<th>Name</th>
<th>Address</th>
</tr>
</thead>
<tbody>
<tr>
<td>John</td>
<td>123 Main</td>
</tr>
<tr>
<td>Mary</td>
<td>7 Lake Dr</td>
</tr>
<tr>
<td>Bart</td>
<td>5 Pine St</td>
</tr>
</tbody>
</table>

The result is a relation/table (no duplicates by definition), so the result can have fewer tuples than the original!
Relational Algebra Expressions

\[ \pi_{Id, Name} (\sigma_{Hobby='stamps' \text{ OR } Hobby='coins'} (\text{Person})) \]

<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>

Result

<table>
<thead>
<tr>
<th>Id</th>
<th>Name</th>
</tr>
</thead>
<tbody>
<tr>
<td>1123</td>
<td>John</td>
</tr>
<tr>
<td>9876</td>
<td>Bart</td>
</tr>
</tbody>
</table>
Set Operators

• A Relation is a **set** of tuples, so set operations should apply: \( \cap, \cup, \neg \) (set difference)

• The result of combining two relations with a set operator is also a relation \( \Rightarrow \) all its elements must be tuples having the 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*
Union 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.
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:
    - Quadratic in the number of rows

\[
\begin{array}{c c}
A & B \\
\hline
x1 & x2 \\
x3 & x4 \\
\end{array}
\quad
\begin{array}{c c}
C & D \\
\hline
y1 & y2 \\
y3 & y4 \\
\end{array}
\quad
\begin{array}{c c c c}
A & B & C & D \\
\hline
x1 & x2 & y1 & y2 \\
x1 & x2 & y3 & y4 \\
x3 & x4 & y1 & y2 \\
x3 & x4 & y3 & y4 \\
\end{array}
\]

$R \quad S \quad R \times S$
Renaming

- The result of expression evaluation is a relation
- The attributes of relation must have distinct names. This is not guaranteed with Cartesian product
  - e.g., suppose in previous example $a$ and $c$ have the same name
- 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

  $$expr [A_1, A_2, \ldots A_n]$$
Renaming Example

Transcript \((\text{StudId, CrsCode, Semester, Grade})\)
Teaching \((\text{ProfId, CrsCode, Semester})\)

\[
\pi_{\text{StudId}, \text{CrsCode}} (\text{Transcript})[\text{StudId}, \text{CrsCode}_1]
\times
\pi_{\text{ProfId}, \text{CrsCode}} (\text{Teaching}) [\text{ProfId}, \text{CrsCode}_2]
\]

This is a relation with 4 attributes:

\(\text{StudId, CrsCode}_1, \text{ProfId, CrsCode}_2\)
Derived Operation: 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:

\[ A_i \text{ operator } B_i \]

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

The meaning is:

\[ \sigma_{\text{join-condition}'} (R \times S) \]

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 $R \bowtie join\_condition \ R$

    In $R.A$, how do we know which $R$ is meant?

(c) Pearson Education Inc. and Paul Fodor (CS Stony Brook)
Theta Join – Example

Employee\((Name, Id, MngrId, Salary)\)
Manager\((Name, Id, Salary)\)

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

\[
\pi_{\text{Employee.Name}} \left( \text{Employee} \bowtie \text{Manager} \right)
\]

\(\bowtie\) \(MngrId=Id\) AND \(Salary>Salary\)

The join yields a table with attributes:

\(\text{Employee.Name, Employee.Id, Employee.Salary, MngrId}\)
\(\text{Manager.Name, Manager.Id, Manager.Salary}\)
**Equijoin Join - Example**

**Equijoin**: Join condition is a conjunction of *equalities*.

$$\pi_{Name, CrsCode} \left( \text{Student} \Join \sigma_{\text{Grade}=\text{A}}(\text{Transcript}) \right)$$

<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>

<table>
<thead>
<tr>
<th>Mary</th>
<th>CSE306</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bill</td>
<td>CSE304</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

\[
\text{Transcript} \bowtie \text{Teaching} = \pi_{\text{StudId}, \text{Transcript}.\text{CrsCode}, \text{Transcript}.\text{Sem}, \text{Grade}, \text{ProfId}} \\
( \text{Transcript} \bowtie \text{CrsCode}=\text{CrsCode} \text{ AND } \text{Sem}=\text{Sem} \text{ Teaching} ) \\
[\text{StudId, CrsCode, Sem, Grade, ProfId}]
\]
Natural Join

- More generally:

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

where

\[ \text{attr-list} = \text{attributes}(R) \cup \text{attributes}(S) \]

(duplicates are eliminated) and \( \text{join-cond} \) has the form:

\[ R.A_1 = S.A_1 \ \text{AND} \ \ldots \ \text{AND} \ R.A_n = S.A_n \]

where

\[ \{A_1 \ldots A_n\} = \text{attributes}(R) \cap \text{attributes}(S) \]
Natural Join Example

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

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

We don’t want to join on CrsCode, Sem, and Grade attributes, hence renaming!
Division (\(\,/,\div\))

- **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 \(<a>\) such that for every tuple \(<b>\) in \(s\), \(<a,b>\) is in \(r\)
- Can be expressed in terms of projection, set difference, and cross-product:
  
  \[
  \begin{align*}
  \text{let } t & := \pi_{A_1, \ldots, A_n}(r) \times s \\
  \text{let } u & := t - r \\
  \text{let } v & := \pi_{A_1, \ldots, A_n}(u) \\
  \text{let } r/s & = \pi_{A_1, \ldots, A_n}(r) - v
  \end{align*}
  \]
Division (/, ÷)

Relation \( r \)

Relation \( s \)

In \( r/s \)

Not in \( r/s \)
Division Example

- List the Ids of students who have passed all courses that were taught in Fall 2016
- **Numerator:**
  - `StudId` and `CrsCode` for every course passed by every student:
    \[ \pi_{StudId, CrsCode} (\sigma_{\text{Grade} \neq 'F'} (\text{Transcript})) \]
- **Denominator:**
  - `CrsCode` of all courses taught in Fall 2016
    \[ \pi_{CrsCode} (\sigma_{\text{Semester} = 'F2016'} (\text{Teaching})) \]
- Result is Numerator / Denominator
Remember the Schema for the 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)
Query Sublanguage of SQL

SELECT  C.CrsName  
FROM    Course C  
WHERE   C.DeptId = 'CSE' 

- Evaluation strategy:
- FROM clause produces Cartesian product of listed tables
  - Tuple variable (alias for the relation) C ranges over rows of Course.
- WHERE clause assigns rows to C in sequence and produces table containing only rows satisfying condition
- SELECT clause retains listed columns
- Equivalent to: \( \pi_{\text{CrsName}} \sigma_{\text{DeptId}=\text{'CSE'}}(\text{Course}) \)
Join Queries

SELECT C.CrsName
FROM Course C, Teaching T
WHERE C.CrsCode = T.CrsCode AND T.Semester = 'F2016'

• List courses taught in F2016
• Join condition “C.CrsCode = T.CrsCode”
  • relates facts to each other
• Selection condition “T.Semester = ‘F2016’
  • eliminates irrelevant rows
Correspondence Between SQL and Relational Algebra

SELECT  C.CrsName
FROM    Course C, Teaching T
WHERE   C.CrsCode = T.CrsCode AND T.Semester = 'F2016'

Equivalent relational algebra expressions:
\[ \pi_{\text{CrsName}}(\sigma_{\text{Sem} = 'F2016'}(\text{Course} \bowtie \text{Teaching})) \]
\[ \pi_{\text{CrsName}}(\text{Course} \bowtie \text{Teaching}) \]
Self-join Queries

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

\[
\text{SELECT T1.ProfId} \\
\text{FROM Teaching T1, Teaching T2} \\
\text{WHERE T1.ProfId = T2.ProfId} \\
\quad \text{AND T1.Semester = T2.Semester} \\
\quad \text{AND T1.CrsCode <> T2.CrsCode}
\]

Tuple variables are essential in this query!

Equivalent to:

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

```sql
SELECT DISTINCT ..... FROM ..... 
```
Use of Expressions

Equality and comparison operators apply to strings (based on lexical ordering)

WHERE S.Name < ‘P’

Concatenate operator applies to strings

WHERE S.Name || ‘--’ || S.Address = ....

Expressions can also be used in SELECT clause:

SELECT S.Name || ‘--’ || S.Address AS NmAdd
FROM Student S
Set Operators

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

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

List all courses that were not taught in F2016

```sql
SELECT C.CrsName
FROM Course C
WHERE C.CrsCode NOT IN
  (SELECT T.CrsCode --subquery
   FROM Teaching T
   WHERE T.Sem = 'F2016')
```

Evaluation strategy: subquery evaluated once to produces set of courses taught in F2016. Each row (as C) tested against this set.
Correlated Nested Queries

Output a row \( <prof, dept> \) if \( prof \) has taught a course in \( dept \).

\[
\text{SELECT } P.\text{Name}, D.\text{Name} \quad \text{--outer query} \\
\text{FROM Professor } P, \text{ Department } D \\
\text{WHERE } P.\text{Id} \text{ IN} \\
\quad \text{-- set of all ProfId’s who have taught a course in } D.\text{DeptId} \\
\left( \text{SELECT T.ProfId} \quad \text{--subquery} \\
\text{FROM Teaching } T, \text{ Course } C \\
\text{WHERE } T.\text{CrsCode}=C.\text{CrsCode} \quad \text{AND} \\
\quad C.\text{DeptId}=D.\text{DeptId} \quad \text{--correlation} \\
\right)
\]
Correlated Nested Queries

- 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 be re-evaluated for each distinct value of D.DeptId
- **Correlated queries can be expensive to evaluate!!!**
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?

\[ \pi_{\text{ProfId, DeptId}}(\text{Teaching} \bowtie \text{Course}) \div \pi_{\text{DeptId}}(\text{Department}) \]
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
Division in SQL

```sql
SELECT P.Id 
FROM Professor P 
WHERE NOT EXISTS 
  (SELECT D.DeptId 
   FROM Department D 
   EXCEPT 
   SELECT C.DeptId 
   FROM Teaching T, Course C 
   WHERE T.ProfId=P.Id 
     AND T.CrsCode=C.CrsCode) 
```
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

Count the number of courses taught in F2016:

```
SELECT COUNT (T.CrsCode)
FROM Teaching T
WHERE T.Semester = 'F2016'
```

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

```
SELECT COUNT (DISTINCT T.CrsCode)
FROM Teaching T
WHERE T.Semester = 'F2016'
```
Grouping

• But how do we compute the number of courses taught in F2016 per professor?

• Strategy 1: Fire off a separate query for each professor:

```sql
SELECT COUNT(T.CrsCode) FROM Teaching T WHERE T.Semester = 'F2016' AND T.ProfId = 123456789
```

• Cumbersome

• What if the number of professors changes? Add another query?

• Strategy 2: define a special grouping operator:

```sql
SELECT T.ProfId, COUNT(T.CrsCode) FROM Teaching T WHERE T.Semester = 'F2016' GROUP BY T.ProfId
```
GROUP BY

- Each row describes a group
- Attributes in the GROUP BY list
- Aggregates over rows in GROUP BY list

Groups

Attributes in the GROUP BY list

All rows in a group agree on all attributes in the GROUP BY list
GROUP BY – Example 2

Find the: student’s Id, avg grade and number of courses

SELECT T.StudId, AVG(T.Grade), COUNT(*)
FROM Transcript T
GROUP BY T.StudId

Transcript

<table>
<thead>
<tr>
<th>StudId</th>
<th>Grade</th>
<th>Courses</th>
</tr>
</thead>
<tbody>
<tr>
<td>1234</td>
<td>3.3</td>
<td>4</td>
</tr>
<tr>
<td>1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1234</td>
<td></td>
<td></td>
</tr>
<tr>
<td>1234</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
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
- Filter the previous example for students with GPA > 3.5

Find the: student’s Id, avg grade and number of courses

```sql
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
```
Order of Operations with GroupBy\&Having

1. Select from WHERE
2. Select from WHERE
3. Select from WHERE
4. Select from WHERE
5. Select from WHERE
6. Select from WHERE
7. Select from WHERE
8. Select from WHERE
9. Select from WHERE
10. Select from WHERE
11. Select from WHERE
12. Select from WHERE
13. Select from WHERE
14. Select from WHERE
15. Select from WHERE
16. Select from WHERE
17. Select from WHERE
18. Select from WHERE
19. Select from WHERE
20. Select from WHERE
21. Select from WHERE
22. Select from WHERE
23. Select from WHERE
24. Select from WHERE
25. Select from WHERE
26. Select from WHERE
27. Select from WHERE
28. Select from WHERE
29. Select from WHERE
30. Select from WHERE
31. Select from WHERE
32. Select from WHERE
33. Select from WHERE
34. Select from WHERE
35. Select from WHERE
36. Select from WHERE
37. Select from WHERE
38. Select from WHERE
39. Select from WHERE
40. Select from WHERE
41. Select from WHERE
42. Select from WHERE
43. Select from WHERE
44. Select from WHERE
45. Select from WHERE
46. Select from WHERE
47. Select from WHERE
48. Select from WHERE
49. Select from WHERE
50. Select from WHERE

(c) Pearson Education Inc. and Paul Fodor (CS Stony Brook)
Example

- Output the name and address of all seniors on the Dean’s List

```
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
```

Every attribute that occurs in SELECT clause must also occur in GROUP BY or it must be an aggregate. S.Name does not.
Aggregates: Proper and Improper Usage

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

SELECT COUNT (*), AVG (T.Grade)
   -- but this is OK since it is for the whole relation

SELECT ... FROM ...
WHERE T.Grade > COUNT (SELECT ....)
   -- aggregate cannot be applied to the result of a SELECT statement
```
**ORDER BY Clause**

- Causes rows to be output in a specified order

```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
Views

- Used as a relation, but rows are not physically stored.
  - The contents of a view is *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
CREATE VIEW CumGpa (StudId, Cum) AS
SELECT T.StudId, AVG (T.Grade)
FROM Transcript T
GROUP BY T.StudId

SELECT S.Name, C.Cum
FROM CumGpa C, Student S
WHERE C.StudId = S.StudId AND C.Cum > 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 = 'F2016' -- 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

(c) Pearson Education Inc. and Paul Fodor (CS Stony Brook)
View Benefits

• **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.
Nulls

- **Conditions**: $x \text{ op } y$ (where \text{ op } is $<$, $>$, $<$>, $=$, etc.) has value \textit{unknown} ($U$) when either $x$ or $y$ is null
  - WHERE \text{ T.cost} > \text{ T.price}

- **Arithmetic expression**: $x \text{ op } y$ (where \text{ op } is $+$, $-$, $\ast$, etc.) has value \text{ NULL} if $x$ or $y$ is \text{ NULL}
  - WHERE \text{ (T.price/T.cost)} > 2

- **Aggregates**: \text{ COUNT} counts \text{ NULL}s like any other value; other aggregates ignore \text{ NULL}s

```sql
SELECT COUNT (T.CrsCode), AVG (T.Grade)
FROM Transcript T
WHERE T.StudId = '1234'
```
Nulls

- **WHERE** clause uses a *three-valued logic* — $T, F, U$ (undefined) — 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 $U(nknown)$

Example: **WHERE** $T.CrsCode = 'CS305'$ AND $T.Grade > 2.5$
INNER JOIN keyword selects all rows from both tables as long as there is a match between the columns in both tables.

```
SELECT column_name(s) 
FROM table1 
INNER JOIN table2 
ON table1.column_name=table2.column_name;
```

or:

```
SELECT column_name(s) 
FROM table1 
JOIN table2 
ON table1.column_name=table2.column_name;
```

INNER JOIN is the same as JOIN
SQL LEFT JOIN Keyword

• INNER JOIN: if there is no match between the columns in both tables, then those rows are not returned.

- The LEFT JOIN keyword returns all rows from the left table (table1), with the matching rows in the right table (table2).
- The result is NULL in the right side when there is no match.
SQL LEFT JOIN Keyword

- SELECT column_name(s)
  FROM table1
  LEFT JOIN table2
  ON table1.column_name=table2.column_name;

- or:

  SELECT column_name(s)
  FROM table1
  LEFT OUTER JOIN table2
  ON table1.column_name=table2.column_name;

- INNER JOIN is the same as JOIN
The RIGHT JOIN keyword returns all rows from the right table (table2), with the matching rows in the left table (table1).

- The result is NULL in the left side when there is no match.

```sql
SELECT column_name(s)
FROM table1
RIGHT JOIN table2
ON table1.column_name=table2.column_name;
```

- or:

```sql
SELECT column_name(s)
FROM table1
RIGHT OUTER JOIN table2
ON table1.column_name=table2.column_name;
```
SQL FULL OUTER JOIN

- SQL FULL OUTER JOIN Keyword: combines the result of both LEFT and RIGHT joins.

```
SELECT column_name(s)
FROM table1
FULL OUTER JOIN table2
ON table1.column_name=table2.column_name;
```
The LIKE operator is used to search for a specified pattern in a column.

SELECT column_name(s)  
FROM table_name  
WHERE column_name LIKE pattern;

selects all customers with a City starting with the letter "s" AND a Country containing the pattern "land" AND the Country NOT LIKE '%green%':

SELECT * FROM Customers  
WHERE City LIKE '%s'  
AND Country LIKE '%land%'  
AND Country NOT LIKE '%green%';
A wildcard character can be used to substitute for any other character(s) in a string.

<table>
<thead>
<tr>
<th>Wildcard</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>%</td>
<td>A substitute for zero or more characters</td>
</tr>
<tr>
<td>_</td>
<td>A substitute for a single character</td>
</tr>
<tr>
<td>[charlist]</td>
<td>Sets and ranges of characters to match</td>
</tr>
<tr>
<td>[^charlist]</td>
<td>Matches only a character NOT specified within the brackets</td>
</tr>
</tbody>
</table>

SELECT * FROM Customers WHERE City LIKE 'L_n_on';
The BETWEEN operator is used to select values within a range.

```
SELECT column_name(s)
FROM table_name
WHERE column_name BETWEEN value1 AND value2;
```

```
SELECT * FROM Products
WHERE Price BETWEEN 10 AND 20;
```
SQL IN Operator

- The IN operator allows you to specify multiple values in a WHERE clause.

```
SELECT column_name(s)
FROM table_name
WHERE column_name IN (value1,value2,...);
```

```
SELECT * FROM Customers
WHERE City IN ('Paris','London');
```
MySQL Date Functions

- **INNER JOIN** keyword selects all rows from both tables as long as there is a match between the columns in both tables.

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>NOW()</td>
<td>Returns the current date and time</td>
</tr>
<tr>
<td>CURDATE()</td>
<td>Returns the current date</td>
</tr>
<tr>
<td>CURTIME()</td>
<td>Returns the current time</td>
</tr>
<tr>
<td>DATE()</td>
<td>Extracts the date part of a date or date/time expression</td>
</tr>
<tr>
<td>EXTRACT()</td>
<td>Returns a single part of a date/time</td>
</tr>
<tr>
<td>DATE_ADD()</td>
<td>Adds a specified time interval to a date</td>
</tr>
<tr>
<td>DATE_SUB()</td>
<td>Subtracts a specified time interval from a date</td>
</tr>
<tr>
<td>DATEDIFF()</td>
<td>Returns the number of days between two dates</td>
</tr>
<tr>
<td>DATE_FORMAT()</td>
<td>Displays date/time data in different formats</td>
</tr>
</tbody>
</table>
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 do not omit it)
  • NULL and DEFAULT values can be specified

```sql
INSERT INTO Transcript (StudId, CrsCode, Semester, Grade) VALUES (12345, 'CSE305', 'F2016', NULL)
```
Bulk Insertion

- Insert the rows output by a SELECT

```sql
CREATE TABLE DeansList (  
    StudId INTEGER,  
    Credits INTEGER,  
    CumGpa FLOAT,  
    PRIMARY KEY StudId)

INSERT INTO DeansList (StudId, Credits, CumGpa) 
SELECT T.StudId, 3 * COUNT (*), AVG(T.Grade) 
FROM Transcript T 
GROUP BY T.StudId 
HAVING AVG(T.Grade) > 3.5 AND COUNT(*) > 30
```
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 output

```sql
DELETE FROM Transcript T
WHERE T.Grade IS NULL AND T.Semester <> 'F2016'
```
Modifying Data - Update

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

```sql
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='F2016'
```
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

INSERT INTO CsReg (StudId, CrsCode, Semester) VALUES (1111, 'CSE305', 'F2016')
Problem 2

Problem: New tuple not in view

Solution: Allow insertion (assuming the WITH CHECK OPTION clause has not been appended to the CREATE VIEW statement)

```sql
INSERT INTO CsReg (StudId, CrsCode, Semester) VALUES (1111, 'ECO105', 'F2016')
```
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

• Tuple <Smith, CS> can be deleted from ProfDept by:
  • 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 (seems like a good idea, but how would the computer know?)
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
```