Relational Algebra and the SQL Query Language

Paul Fodor
CSE316: Fundamentals of Software Development
Stony Brook University

http://www.cs.stonybrook.edu/~cse316
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:
  - **Relational Algebra**
    - Intermediate language used within the DBMS
    - Procedural
      - the *query optimizer* converts the query algebraic expression into an equivalent faster *query execution plan*
  - **Structured Query Language (SQL)**
    - The predominant application-level query language
    - Declarative
      - translated internally by the DBMS into relational algebra expressions (or other calculi, such as tuple relational calculus)
The Role of Relational Algebra in a DBMS

SQL Query

<table>
<thead>
<tr>
<th>Relational Algebra Expression</th>
</tr>
</thead>
<tbody>
<tr>
<td>Query Optimizer</td>
</tr>
<tr>
<td>Code Generator</td>
</tr>
<tr>
<td>Query Execution Plan</td>
</tr>
<tr>
<td>Executable Code</td>
</tr>
</tbody>
</table>

(c) Pearson Education Inc. and Paul Fodor (CS Stony Brook)
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 that we will 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**: a set of relations

- **Basic operators**:
  - `select`
  - `project`
  - `union`
  - `set difference`
  - `Cartesian product`

- **Derived operators**:
  - `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.
Select Operator

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

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

- Simple selection condition:
  - `<attribute> operator <constant / attribute>`

<table>
<thead>
<tr>
<th>Operator</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>=</td>
<td>Equal</td>
</tr>
<tr>
<td>&gt;</td>
<td>Greater Than</td>
</tr>
<tr>
<td>&lt;</td>
<td>Less Than</td>
</tr>
<tr>
<td>&gt;=</td>
<td>Greater or Equal</td>
</tr>
<tr>
<td>&lt;=</td>
<td>Less or Equal</td>
</tr>
<tr>
<td>&lt;&gt;</td>
<td>Not Equal</td>
</tr>
</tbody>
</table>

- And Boolean expressions:
  - `<condition> AND <condition>`
  - `<condition> OR <condition>`
  - NOT `<condition>`
Selection Condition - Examples

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

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

- Produces the relation/table containing subset of columns of the argument table
- Example:

\begin{align*}
\text{Person} & \\
\begin{array}{|c|c|c|c|}
\hline
\text{Id} & \text{Name} & \text{Address} & \text{Hobby} \\
\hline
1123 & John & 123 Main & stamps \\
1123 & John & 123 Main & coins \\
5556 & Mary & 7 Lake Dr & hiking \\
9876 & Bart & 5 Pine St & stamps \\
\hline
\end{array}
\end{align*}

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

\begin{align*}
\text{Name} & \text{ Hobby} \\
John & stamps \\
John & coins \\
Mary & hiking \\
Bart & stamps \\
\end{align*}
Project Operator

• Relational Algebra has a set semantics: 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>

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

John 123 Main
Mary 7 Lake Dr
Bart 5 Pine St

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

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

<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: \( \cap, \cup, - \) (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, the scope of set operations is limited to *union compatible relations*
Union Compatible Relations

- Two relations are *union compatible* if:
  - Both have the 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 $\langle x, y \rangle$, where $x$ is a tuple in $R$ and $y$ is a tuple in $S$.

- $R \times S$ is **expensive to compute**:
  - Quadratic in the number of rows

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

\[
R \quad S \quad R \times S
\]
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 \((StudId, CrsCode, Semester, Grade)\)
Teaching \((ProfId, CrsCode, Semester)\)

\[
\pi_{StudId, CrsCode} (\text{Transcript})[StudId, CrsCode1]
\times
\pi_{ProfId, CrsCode} (\text{Teaching}) [ProfId, CrsCode2]
\]

This is a relation with 4 attributes:

\(StudId, CrsCode1, ProfId, CrsCode2\)
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`. OR
  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 $R \bowtie \text{join\_condition} R$

    how do we know which $R$ is meant?
Theta Join – Example

\( \text{Employee}(Name, Id, \text{MngrId}, \text{Salary}) \)
\( \text{Manager}(Name, Id, \text{Salary}) \)

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

\[ \pi_{\text{Employee}.Name} (\text{Employee} \bowtie \text{Manager} \text{MngrId}=Id \text{ AND Salary}>\text{Salary}) \]

The inner join yields a table with attributes:

\( \text{Employee}.Name, \text{Employee}.Id, \text{Employee}.Salary, \text{MngrId} \)
\( \text{Manager}.Name, \text{Manager}.Id, \text{Manager}.Salary \)

Then we project only on the \text{Employee}.Name.
### 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>

Mary CSE306
Bill CSE304

The equijoin is used very frequently since it combines related data in different relations on foreign keys.
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 (i.e., only 1 copy is kept)

Transcript \((StudId, CrsCode, Sem, Grade)\)
Teaching \((ProfId, CrsCode, Sem)\)

\[
\text{Transcript} \Join \text{Teaching} = \\
\pi_{StudId, \text{Transcript.}CrsCode, \text{Transcript.}Sem, \text{Grade, ProfId}} \\
( \text{Transcript} \Join \text{Teaching} \text{ CrsCode}=\text{CrsCode AND Sem}=\text{Sem} ) \\
[StudId, CrsCode, Sem, Grade, ProfId ]
\]
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 \text{CrsCode}, \text{Sem}, and \text{Grade} attributes, hence renaming!
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 \(<a>\) 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:
  
  Let \( t := \pi_{A_1,\ldots,A_n}(r) \times s \) has the same attributes with \( r \)
  
  Let \( u := t - r \)
  
  Let \( v := \pi_{A_1,\ldots,A_n}(u) \)
  
  \( r/s = \pi_{A_1,\ldots,A_n}(r) - v \)
Division (\(\div\))

Relation \(r\)

Relation \(s\)

\(\text{In } r/s\)

\(\text{Not in } r/s\)
**Division Example**

- List the Ids of students who have passed *all* courses that were taught in Fall 2020
  - **Numerator:**
    - *StudId* and *CrsCode* for every course passed by every student:
      \[
      \pi_{\text{StudId, CrsCode}} (\sigma_{\text{Grade} \neq 'F'} (\text{Transcript}) )
      \]
  - **Denominator:**
    - *CrsCode* of all courses taught in Fall 2020
      \[
      \pi_{\text{CrsCode}} (\sigma_{\text{Semester}='F2020'} (\text{Teaching}) )
      \]
  - Result is **Numerator/Denominator**
The Query Sublanguage of SQL

• SQL – Structured Query Language

• Syntax:

```
SELECT <column_list> FROM <table_name_list>
[WHERE condition]
[GROUP BY <column_name>]
[HAVING <condition>]
[ORDER BY <column_name> [ASC|DESC]]
```
Remember our 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)
The Query Sublanguage of SQL

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

• Evaluation strategy:
  • FROM clause produces Cartesian product of listed tables
    • C is a tuple variable (alias for the relation) and it ranges over rows of Course.
  • WHERE filters only rows satisfying the condition
  • SELECT clause retains/projects only the listed columns
• Equivalent to: $\pi_{CrsName} \sigma_{DeptId='CSE'}(Course)$
Join Queries

• List the names of all the courses taught in F2020

```
SELECT  C.CrsName
FROM    Course C, Teaching T
WHERE   C.CrsCode=T.CrsCode AND T.Semester='F2020'
```

• Join condition “C.CrsCode=T.CrsCode”
  • relates facts to each other
• Selection condition “T.Semester='F2020'
  • eliminates irrelevant rows
• Followed by projection on CrsName
  • Note SELECT "*" means all the attributes
Correspondence Between SQL and Relational Algebra

```
SELECT  C.CrsName
FROM    Course C, Teaching T
WHERE   C.CrsCode = T.CrsCode AND T.Semester = 'F2020'
```

Equivalent relational algebra expressions:

\[ \pi_{\text{CrsName}} (\text{Course} \bowtie \sigma_{\text{Sem}='F2020'} (\text{Teaching}) ) \]

\[ \pi_{\text{CrsName}} (\sigma_{\text{Sem}='F2020'} (\text{Course} \bowtie \text{Teaching}) ) \]

\[ \pi_{\text{CrsName}} \sigma_{\text{C_CrsCode}=\text{T_CrsCode AND Semester='F2020'}} (\text{Course} [\text{C_CrsCode}, \text{DeptId}, \text{CrsName}, \text{Desc}] \times \text{Teaching} [\text{ProfId}, \text{T_CrsCode}, \text{Semester}]) \]

• Relational algebra expressions are procedural
  ➢ Which of the equivalent expressions is most efficient? First one.
Self-join Queries

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

\[
\begin{align*}
\text{SELECT} & \quad T1.ProfId \\
\text{FROM} & \quad \text{Teaching} \ T1, \ \text{Teaching} \ T2 \\
\text{WHERE} & \quad T1.ProfId = T2.ProfId \\
& \quad \text{AND} \quad T1.Semester = T2.Semester \\
& \quad \text{AND} \quad T1.CrsCode \neq T2.CrsCode
\end{align*}
\]

Tuple variables are essential in this query!

Equivalent to:

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

```
SELECT DISTINCT ..... FROM ..... 
```
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 CSE Department and all professors that have taught CSE 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 F2020:

```
SELECT C.CrsName
FROM Course C
WHERE C.CrsCode NOT IN
  (SELECT T.CrsCode  -- subquery
   FROM Teaching T
   WHERE T.Sem = 'F2020')
```

Evaluation strategy: subquery evaluated once to produces set of courses taught in F2020. Each row (as C) tested against this set.
Correlated Nested Queries

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

```sql
SELECT P.Name, D.Name --outer query
FROM Professor P, Department D
WHERE P.Id IN
  (SELECT T.ProfId --subquery
   FROM Teaching T, Course C
   WHERE T.CrsCode=C.CrsCode AND 
   C.DeptId=D.DeptId --correlation
  )
```

- Tuple variables \(T\) and \(C\) are *local* to subquery, while \(P\) 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

Contains row \(<p,d>\) if professor \(p\) taught a course in department \(d\)

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

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 -- global variable
        AND T.CrsCode=C.CrsCode)

-- set B of all dept Ids
-- set A of dept Ids of depts in
-- which P taught a course
Aggregates

• Functions that operate on sets:
  • COUNT, SUM, AVG, MAX, MIN
• Produce numbers (not tables)

SELECT COUNT(*) FROM Professor P
SELECT MAX (Salary) FROM Employee E
Aggregates

Count the number of courses taught in F2020:

```sql
SELECT COUNT (T.CrsCode) FROM Teaching T WHERE T.Semester = 'F2020'
```

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

```sql
SELECT COUNT (DISTINCT T.CrsCode) FROM Teaching T WHERE T.Semester = 'F2020'
```
But how do we compute the number of courses taught in F2020 per professor?

- **Strategy 1:** Fire off a separate query for each professor:
  
  ```sql
  SELECT COUNT(T.CrsCode) 
  FROM Teaching T 
  WHERE T.Semester = 'F2020' AND T.ProfId = 123456789 
  ```

- Cumbersome: we have to query prof. by prof.

- **Strategy 2:** Define a special grouping operator:
  
  ```sql
  SELECT T.ProfId, COUNT(DISTINCT T.CrsCode) 
  FROM Teaching T 
  WHERE T.Semester = 'F2020' 
  GROUP BY T.ProfId 
  ```

In the SELECT part of a SQL query that contains GROUP BY, you can only include attributes of GROUP BY list and aggregates on attributes not in that list.
GROUP BY

Each row describes a group

Attributes in the GROUP BY list

Aggregates over rows in GROUP BY list

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

(c) Pearson Education Inc. and Paul Fodor (CS Stony Brook)
GROUP BY – multiple aggregates in one query

*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
```
HAVING Clause

- Eliminate 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
```
Example

• Output the name and address of all seniors on the Dean's List

```sql
SELECT S.Id, S.Name
FROM Student S, Transcript T
WHERE S.Id = T.StudId AND S.Status = 'senior'
GROUP BY S.Id, S.Name -- right
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

Note: never use an aggregate in WHERE
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
Order of Operations with GroupBy & Having

- DB
  - SELECT
  - FROM
  - WHERE
  - Attrs, Relations
  - Condition
  - Result

- SELECT
  - FROM
  - WHERE
  - GROUP BY
  - Attrs
  - Relations
  - Condition
  - Group Attr List
  - Result

- SELECT
  - FROM
  - WHERE
  - GROUP BY
  - Attrs
  - Relations
  - Condition
  - Group Attr List
  - Group Condition
  - Result

- SELECT
  - FROM
  - WHERE
  - GROUP BY
  - Attrs
  - Relations
  - Condition
  - Group Attr List
  - Group Condition
  - Result

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

• Used in other SELECT statements:

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)


CREATE VIEW PartOfTranscript (StudId, CrsCode, Semester) AS
SELECT T.StudId, T.CrsCode, T.Semester -- limit columns
FROM Transcript T
WHERE T.Semester = 'F2020' -- 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 and undefined

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

- Conditions: $x \, op \, y$ (where $op$ is $<$, $>$, $<=$, $<=$, $=$, etc.) has value undefined/unknown ($U$) when either $x$ or $y$ is null
  - WHERE $T.\, cost > T.\, price$
Nulls and **undefined**

- **WHERE** clause uses a *three-valued logic* – T, F, U(defined) – 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(ndefined)
Nulls

- **Aggregates**: COUNT counts NULLs like any other value; other aggregates ignore NULLs

```sql
SELECT COUNT (T.CrsCode), AVG (T.Grade)
FROM Transcript T
WHERE T.StudId = '1234'
```
INNER JOIN / JOIN keywords selects all rows from both tables as long as there is a match between the columns in both tables.

\[
\text{SELECT column\_name(s)} \\
\text{FROM table1} \\
\text{INNER JOIN table2} \\
\text{ON table1.column\_name=table2.column\_name;}
\]

or:

\[
\text{SELECT column\_name(s)} \\
\text{FROM table1} \\
\text{JOIN table2} \\
\text{ON table1.column\_name=table2.column\_name;}
\]

INNER JOIN is the same as JOIN
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`;

- LEFT JOIN is the same as LEFT OUTER 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;
Comma join vs. JOIN

SELECT pers.FirstName, pers.LastName, p.Name, p.ListPrice
FROM Sales.Customer AS c, Sales.SalesOrderHeader AS soh,
    Sales.SalesOrderDetail AS sod, Production.Product AS p,
    person.Person AS pers

VS

SELECT pers.FirstName, pers.LastName, p.Name, p.ListPrice
FROM Sales.Customer AS c
INNER JOIN Sales.SalesOrderHeader AS soh ON c.CustomerID = soh.CustomerID
INNER JOIN Sales.SalesOrderDetail AS sod ON sod.SalesOrderID = soh.SalesOrderID
INNER JOIN person.Person AS pers ON pers.BusinessEntityID = c.PersonID;
JOIN as comma join

- LEFT OUTER JOINs in the comma form: by changing the equal sign to an asterisk + equal sign (\*=) or a RIGHT OUTER JOIN by changing the equal sign to an equal sign + asterisk (\=*).
- The JOIN syntax makes it easy to add or eliminate the tables for trouble-shooting by commenting out the line.
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%';
```
SQL Wildcard Characters

- 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] or ![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';
SQL BETWEEN Operator

- 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

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>NOW()</strong></td>
<td>Returns the current date and time</td>
</tr>
<tr>
<td><strong>CURDATE()</strong></td>
<td>Returns the current date</td>
</tr>
<tr>
<td><strong>CURTIME()</strong></td>
<td>Returns the current time</td>
</tr>
<tr>
<td><strong>DATE()</strong></td>
<td>Extracts the date part of a date or date/time expression</td>
</tr>
<tr>
<td><strong>EXTRACT()</strong></td>
<td>Returns a single part of a date/time</td>
</tr>
<tr>
<td><strong>DATE_ADD()</strong></td>
<td>Adds a specified time interval to a date</td>
</tr>
<tr>
<td><strong>DATE_SUB()</strong></td>
<td>Subtracts a specified time interval from a date</td>
</tr>
<tr>
<td><strong>DATEDIFF()</strong></td>
<td>Returns the number of days between two dates</td>
</tr>
<tr>
<td><strong>DATE_FORMAT()</strong></td>
<td>Displays date/time data in different formats</td>
</tr>
</tbody>
</table>
LIMIT

- Limit the number of rows returned
  - MySQL: \( \text{SELECT}\ column\_name(s) \)
    \( \text{FROM}\ table\_name \)
    \( \text{WHERE}\ condition \)
    \( \text{LIMIT}\ number; \)
  - SQL Server:
    \( \text{SELECT}\ \text{TOP}\ number|\text{percent}\ column\_name(s) \)
    \( \text{FROM}\ table\_name \)
    \( \text{WHERE}\ condition; \)

- can be a count or a percentage
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', 'F2020', NULL)
```
Bulk Insertion

- Insert the rows output by a SELECT

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

```sql
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 <> 'F2020'
```
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='F2020'
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', 'F2020')
```
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)

```
INSERT INTO CsReg (StudId, CrsCode, Semester) VALUES (1111, 'ECO105', 'F2020')
```
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)

```sql
CREATE VIEW ProfDept (PrName, DeName) AS
SELECT P.Name, D.Name
FROM Professor P, Department D
WHERE P.DeptId = D.DeptId
```
Tuple \(<\text{Smith, CS}>\) can be deleted from \text{ProfDept} by:

- Deleting row for Smith from \text{Professor} (but this is inappropriate if he is still at the University)
- Deleting row for CS from \text{Department} (not what is intended)
- Updating row for Smith in \text{Professor} by setting \text{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
```
Security Concerns

- Consider a Customer database
- Allows a user to query for their customer information by entering their SSN:
  
  $$\text{SELECT * FROM customers WHERE SSN = $input}$$
  
  - Expected input would be a 9 digit number (i.e. 123456789)
- What if user enters: 99 or 1=1
- Now, the query will be:
  
  $$\text{SELECT * FROM customers WHERE SSN = 99 or 1=1}$$
  
  - This will cause the DB software to return all records in the database!

- Solution: Scan the input data
  
  - Assure it meets the requirements and format (9 digit number, no special characters)
  - Return an error if it does not
  - Optional: modify the input to remove special characters, keywords, etc. Then run query