Relational Languages:

Recap of ICS 324
Lecture objectives

- Briefly mention relational languages covered in ICS 334
  - Relational Algebra
  - SQL
- Relational Languages

- Relational Algebra
  - Relational Operators
  - Selection
  - Projection
  - Cross product
  - Join
  - Natural Join
  - Union
  - Difference
  - Intersection
  - Renaming
  - Summary

- SQL
## Input relations

### Student

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Mustafa</td>
<td>17</td>
<td>3.2</td>
</tr>
<tr>
<td>222</td>
<td>Ali</td>
<td>17</td>
<td>2.8</td>
</tr>
<tr>
<td>333</td>
<td>Ahmed</td>
<td>22</td>
<td>2.5</td>
</tr>
<tr>
<td>444</td>
<td>Lutfi</td>
<td>20</td>
<td>3.5</td>
</tr>
</tbody>
</table>

### Course

<table>
<thead>
<tr>
<th>CID</th>
<th>Title</th>
</tr>
</thead>
<tbody>
<tr>
<td>ICS 102</td>
<td>Java</td>
</tr>
<tr>
<td>ICS 202</td>
<td>Data structures</td>
</tr>
<tr>
<td>ICS 434</td>
<td>Advanced Databases</td>
</tr>
<tr>
<td>ICS 334</td>
<td>Databases</td>
</tr>
<tr>
<td>ICS 431</td>
<td>Operating Systems</td>
</tr>
</tbody>
</table>

### Enroll

<table>
<thead>
<tr>
<th>SID</th>
<th>CID</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>ICS 102</td>
</tr>
<tr>
<td>222</td>
<td>ICS 434</td>
</tr>
<tr>
<td>222</td>
<td>ICS 431</td>
</tr>
<tr>
<td>333</td>
<td>ICS 334</td>
</tr>
<tr>
<td>333</td>
<td>ICS 431</td>
</tr>
<tr>
<td>444</td>
<td>ICS 102</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
<tr>
<td>......</td>
<td>......</td>
</tr>
</tbody>
</table>
Relational algebra is notation for operations on relations, like constructing new relations and defining queries on relations.

Very important for query optimization.

Core set of operators:
- Selection, projection, cross product, union, difference, and, renaming

Additional, derived operators:
- Join, natural join, intersection, etc.
--- Selection ...

- **Input**: a table $R$

- **Notation**: $\sigma_p(R)$
  - $p$ is called a selection condition/predicate

- **Purpose**: filter rows according to some criteria

- **Output**: same columns as $R$, but only rows of $R$ that satisfy $p$
--- Selection ---

**Example**

- Students with GPA higher than 3.0

\[ \sigma_{GPA} > 3.0 \ (Student) \]

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Mustafa</td>
<td>17</td>
<td>3.2</td>
</tr>
<tr>
<td>222</td>
<td>Ali</td>
<td>17</td>
<td>2.8</td>
</tr>
<tr>
<td>333</td>
<td>Ahmed</td>
<td>22</td>
<td>2.5</td>
</tr>
<tr>
<td>444</td>
<td>Lutfi</td>
<td>20</td>
<td>3.5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>

\[ \sigma_{GPA} > 3.0 \]
Selection

- Selection predicate in general can include any column of $R$, constants, comparisons such as $=$, $\leq$, etc., and Boolean connectives $\lor$, $\land$, and $\neg$

- Example: List all A students under 18 or over 20

$$\sigma_{\text{GPA} \geq 4.0 \land (\text{age} < 18 \lor \text{age} > 20)} (\text{Student})$$

- But you must be able to evaluate the predicate over a single row

- Example: student with the highest GPA

$$\sigma_{\text{GPA} \geq \text{all GPA}} (\text{Student})$$
--- Projection ...

- **Input**: a table $R$
- **Notation**: $\Pi_L ( R )$
  - $L$ is a list of columns in $R$
- **Purpose**: select columns to output
- **Output**: same rows, but only the columns in $L$. Duplicate output rows are removed.
... --- Projection

- Example:
  - ID’s and names of all students
    \[ \Pi_{SID, \text{name}}(\text{Student}) \]

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Mustafa</td>
<td>17</td>
<td>3.2</td>
</tr>
<tr>
<td>222</td>
<td>Ali</td>
<td>17</td>
<td>2.8</td>
</tr>
<tr>
<td>333</td>
<td>Ahmed</td>
<td>22</td>
<td>2.5</td>
</tr>
<tr>
<td>444</td>
<td>Lutfi</td>
<td>20</td>
<td>3.5</td>
</tr>
<tr>
<td>...</td>
<td>...</td>
<td>...</td>
<td>...</td>
</tr>
</tbody>
</table>
--- Cross product ...

- **Input**: two tables \( R \) and \( S \)
- **Notation**: \( R \times S \)
- **Purpose**: pairs rows from two tables
- **Output**: for each row \( r \) in \( R \) and each row \( s \) in \( S \), output a row \( rs \) (concatenation of \( r \) and \( s \))
  - The ordering of columns in a table is considered unimportant (as is the ordering of rows)
  - That means cross product is commutative, i.e., \( R \times S = S \times R \) for any \( R \) and \( S \)
... --- Cross product

- Example: \textit{Student} $\times$ \textit{Enroll}

\begin{itemize}
  \item \textbf{Student}
  \begin{tabular}{|c|c|c|c|}
    \hline
    SID & name & Age & GPA \\
    \hline
    111 & Mustafa & 17 & 3.2 \\
    222 & Ali & 17 & 2.8 \\
    333 & Ahmed & 22 & 2.5 \\
    444 & Lutfi & 20 & 3.5 \\
    \hline
  \end{tabular}

  \item \textbf{Enroll}
  \begin{tabular}{|c|c|}
    \hline
    SID & CID \\
    \hline
    111 & ICS 102 \\
    222 & ICS 434 \\
    222 & ICS 431 \\
    333 & ICS 334 \\
    333 & ICS 431 \\
    444 & ICS 102 \\
    \hline
  \end{tabular}
\end{itemize}
--- Join

- **Input:** two tables $R$ and $S$
- **Notation:** $R \bowtie p S$
  - $p$ is called a join condition/predicate

- **Purpose:** relate rows from two tables according to some criteria

- **Output:** for each row $r$ in $R$ and each row $s$ in $S$, output a row $rs$ if $r$ and $s$ satisfy

- **Shorthand for** $\sigma_p ( R \times S )$
... --- Join

- Example: Info about students, plus CID’s of their courses

\[
\text{Student} \Join \text{Student.SID} = \text{Enroll.SID} \text{ Enroll}
\]

### Student

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Mustafa</td>
<td>17</td>
<td>3.2</td>
</tr>
<tr>
<td>222</td>
<td>Ali</td>
<td>17</td>
<td>2.8</td>
</tr>
<tr>
<td>333</td>
<td>Ahmed</td>
<td>22</td>
<td>2.5</td>
</tr>
<tr>
<td>444</td>
<td>Lutfi</td>
<td>20</td>
<td>3.5</td>
</tr>
</tbody>
</table>

### Enroll

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Age</th>
<th>GPA</th>
<th>SID</th>
<th>CID</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Mustafa</td>
<td>17</td>
<td>3.2</td>
<td>111</td>
<td>ICS 102</td>
</tr>
<tr>
<td>222</td>
<td>Ali</td>
<td>17</td>
<td>2.8</td>
<td>222</td>
<td>ICS 434</td>
</tr>
<tr>
<td>222</td>
<td>Ali</td>
<td>17</td>
<td>2.8</td>
<td>222</td>
<td>ICS 431</td>
</tr>
<tr>
<td>333</td>
<td>Ahmed</td>
<td>22</td>
<td>2.5</td>
<td>333</td>
<td>ICS 334</td>
</tr>
<tr>
<td>333</td>
<td>Ahmed</td>
<td>22</td>
<td>2.5</td>
<td>333</td>
<td>ICS 431</td>
</tr>
<tr>
<td>444</td>
<td>ICS 102</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
<tr>
<td>.....</td>
<td>....</td>
<td>.....</td>
<td>...</td>
<td></td>
<td></td>
</tr>
</tbody>
</table>
--- Natural Join

- **Input**: two tables $R$ and $S$
- **Notation**: $R \bowtie S$
- **Purpose**: relate rows from two tables, and
  - Enforce equality on all common attributes
  - Eliminate one copy of common attributes
- **Shorthand for** $\Pi_L \left( R \bowtie_p S \right)$
  - $L$ is the union of all attributes from $R$ and $S$, with duplicates removed
  - $p$ equates all attributes common to $R$ and $S$
--- Natural Join

- **Example:** Info about students, plus CID’s of their courses

  \[ \text{Student} \Join \text{Enroll} \]

<table>
<thead>
<tr>
<th>SID</th>
<th>name</th>
<th>Age</th>
<th>GPA</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Mustafa</td>
<td>17</td>
<td>3.2</td>
</tr>
<tr>
<td>222</td>
<td>Ali</td>
<td>17</td>
<td>2.8</td>
</tr>
<tr>
<td>333</td>
<td>Ahmed</td>
<td>22</td>
<td>2.5</td>
</tr>
<tr>
<td>444</td>
<td>Lutfi</td>
<td>20</td>
<td>3.5</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SID</th>
<th>CID</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>ICS 102</td>
</tr>
<tr>
<td>222</td>
<td>ICS 434</td>
</tr>
<tr>
<td>222</td>
<td>ICS 431</td>
</tr>
<tr>
<td>333</td>
<td>ICS 334</td>
</tr>
<tr>
<td>333</td>
<td>ICS 431</td>
</tr>
<tr>
<td>444</td>
<td>ICS 102</td>
</tr>
<tr>
<td></td>
<td></td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>SID</th>
<th>Name</th>
<th>Age</th>
<th>GPA</th>
<th>CID</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>Mustafa</td>
<td>17</td>
<td>3.2</td>
<td>ICS 102</td>
</tr>
<tr>
<td>222</td>
<td>Ali</td>
<td>17</td>
<td>2.8</td>
<td>ICS 434</td>
</tr>
<tr>
<td>222</td>
<td>Ali</td>
<td>17</td>
<td>2.8</td>
<td>ICS 431</td>
</tr>
<tr>
<td>333</td>
<td>Ahmed</td>
<td>22</td>
<td>2.5</td>
<td>ICS 334</td>
</tr>
<tr>
<td></td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

---
--- Union

- **Input**: two tables $R$ and $S$
- **Notation**: $R \cup S$
  - $R$ and $S$ must have identical schema
- **Output**:
  - Has the same schema as $R$ and $S$
  - Contains all rows in $R$ and all rows in $S$, with duplicates eliminated
--- Difference

- **Input**: two tables $R$ and $S$
- **Notation**: $R - S$
  - $R$ and $S$ must have identical schema
- **Output**:  
  - Has the same schema as $R$ and $S$
  - Contains all rows in $R$ that are not found in $S$
--- Intersection

- **Input**: two tables $R$ and $S$
- **Notation**: $R \cap S$
  - $R$ and $S$ must have identical schema
- **Output**:
  - Has the same schema as $R$ and $S$
  - Contains all rows that are in both $R$ and $S$
- Shorthand for $R - (R - S)$
- Also equivalent to $S - (S - R)$
- And to $R \bowtie S$
--- Renaming

- **Input:** a table $R$
- **Notation:** $\rho_S(R)$, or $\rho_S(A_1, A_2, \ldots)(R)$
- **Purpose:** rename a table and/or its columns
- **Output:** a renamed table with the same rows as $R$
- **Used to**
  - Avoid confusion caused by identical column names
  - Create identical columns names for natural joins
--- Summary of Relational Algebra Operators

- **Core**
  - Selection: \( \sigma_p ( R ) \)
  - Projection: \( \Pi_L ( R ) \)
  - Cross product: \( R \times S \)
  - Union: \( R \cup S \)
  - Difference: \( R - S \)
  - Renaming: \( \rho_{S(A_1, A_2, \ldots)} ( R ) \)

- **Derived**
  - Join: \( R \bowtie p S \)
  - Natural join: \( R \bowtie S \)
  - Intersection: \( R \cap S \)
  - Many more: Semijoin, anti-semijoin, quotient, aggregation, ...
-- SQL

- Definition
- Basic CREATE/DROP TABLE
- INSERT
- DELETE
- UPDATE
- SELECT
- Set and bag operations
- Aggregation and grouping
- NULL’s
- SQL Constraints
- Others
--- Definition

- SQL: Structured Query Language
- Pronounced “S-Q-L” or “sequel”
- The standard query language support by most commercial DBMS
Creating and dropping tables

- CREATE TABLE table_name (... column_name column_type, ...);
- DROP TABLE table_name;
- Examples
  - create table Student (SID integer, name varchar(30), email varchar(30), age integer, GPA float);
  - create table Course (CID char(10), title varchar(100));
  - create table Enroll (SID integer, CID char(10));
  - drop table Student;
  - drop table Course;
  - drop table Enroll;
- everything from -- to the end of the line is ignored.
- SQL is insensitive to white space.
- SQL is case insensitive (e.g., ...Course... is equivalent to ...COURSE...).
--- INSERT

- Insert one row
  
  `INSERT INTO Enroll VALUES (111, 'ICS334');`
  
  - Student 111 takes ICS 334

- Insert the result of a query
  
  `INSERT INTO Enroll
  (SELECT SID, 'ICS334' FROM Student
  WHERE SID NOT IN (SELECT SID FROM Enroll
  WHERE CID = 'ICS334'));
  
  - Force everybody to take ICS 334
--- DELETE

- Delete everything
  
  `DELETE FROM Enroll;`

- Delete according to a WHERE condition
  
  Example: Student 111 drops ICS 334
  
  `DELETE FROM Enroll
  WHERE SID = 111 AND CID = 'ICS334';`

- Example: Drop students with GPA lower than 1.0 from all ICS classes
  
  `DELETE FROM Enroll
  WHERE SID IN (SELECT SID FROM Student
  WHERE GPA < 1.0)
  AND CID LIKE 'ICS%';`
--- Update

- Example: Student 111 changes name to “Hazem” and GPA to 3.0
  
  UPDATE Student
  SET name = 'Hazem', GPA = 3.0
  WHERE SID = 111;

- Example: Assign every student average GPA
  
  UPDATE Student
  SET GPA = (SELECT AVG(GPA) FROM Student);

  - But update of every row causes average GPA to change!
  - Average GPA is computed over the old Student table
--- Select

- SELECT * FROM Student;
- Single-table query
- WHERE clause is optional
- * is a short hand for “all columns”
- Equivalent to: $\sigma_{SID, name, age, GPA \ (Student)}$
--- Selection and Projection

- Name of students under 18
  
  ```
  SELECT name
  FROM Student
  WHERE age < 18;
  ```

- When was Mustafa born?
  
  ```
  SELECT 2006 - age
  FROM Student
  WHERE name = 'Mustafa';
  ```

- SELECT list can contain expressions
  - Can also use built-in functions such as SUBSTR, ABS, etc.

- String literals (case sensitive) are enclosed in single quotes
--- Join

- SID’s and name’s of students taking courses with the word “Database” in their titles

```sql
SELECT Student.SID, Student.name
FROM Student, Enroll, Course
WHERE Student.SID = Enroll.SID
AND Enroll.CID = Course.CID
AND title LIKE 'Database%';
```

- LIKE matches a string against a pattern
  - % matches any sequence of 0 or more characters

- Okay to omit `table_name` in `table_name.column_name` if `column_name` is unique
--- rename

- SID’s of students who take at least two courses

```sql
SELECT e1.SID AS SID
FROM Enroll AS e1, Enroll AS e2
WHERE e1.SID = e2.SID
AND e1.CID <> e2.CID;
```

- AS keyword is completely optional
--- A more complicated example

- Titles of all courses that Ali and Mustafa are taking together

```sql
SELECT c.title
FROM Student sb, Student sl, Enroll eb, Enroll el, Course c
WHERE sb.name = 'Ali' AND sl.name = 'Mustafa'
AND eb.SID = sb.SID AND el.SID = sl.SID
AND eb.CID = el.CID
AND eb.CID = c.CID;
```

- Tip: Write the FROM clause first, then WHERE, and then SELECT
--- Set versus bag semantics

- **Set versus bag semantics**

- **Set**
  - No duplicates
  - Relational model and algebra use set semantics

- **Bag**
  - Duplicates allowed
  - Number of duplicates is significant
  - SQL uses bag semantics by default
--- Set verses bag example

\[ \Pi_{SID} (\text{Enroll}) \]

**Set**

<table>
<thead>
<tr>
<th>SID</th>
<th>CID</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>ICS 102</td>
</tr>
<tr>
<td>222</td>
<td>ICS 434</td>
</tr>
<tr>
<td>222</td>
<td>ICS 431</td>
</tr>
<tr>
<td>333</td>
<td>ICS 334</td>
</tr>
<tr>
<td>333</td>
<td>ICS 431</td>
</tr>
<tr>
<td>444</td>
<td>ICS 102</td>
</tr>
<tr>
<td>.....</td>
<td>....</td>
</tr>
</tbody>
</table>

**Enroll**

<table>
<thead>
<tr>
<th>SID</th>
<th>CID</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
<td>ICS 102</td>
</tr>
<tr>
<td>222</td>
<td>ICS 434</td>
</tr>
<tr>
<td>222</td>
<td>ICS 431</td>
</tr>
<tr>
<td>333</td>
<td>ICS 334</td>
</tr>
<tr>
<td>333</td>
<td>ICS 431</td>
</tr>
<tr>
<td>444</td>
<td>ICS 102</td>
</tr>
<tr>
<td>.....</td>
<td>....</td>
</tr>
</tbody>
</table>

**Select SID From Enroll**

<table>
<thead>
<tr>
<th>SID</th>
</tr>
</thead>
<tbody>
<tr>
<td>111</td>
</tr>
<tr>
<td>222</td>
</tr>
<tr>
<td>222</td>
</tr>
<tr>
<td>333</td>
</tr>
<tr>
<td>333</td>
</tr>
<tr>
<td>444</td>
</tr>
<tr>
<td>.....</td>
</tr>
</tbody>
</table>

**Bag**
--- A case for bag semantics

- Efficiency
  - Saves time of eliminating duplicates

- Which one is more useful?
  - $\pi_{GPA} (Student)$
  - `SELECT GPA FROM Student;`
  - The first query just returns all possible GPA’s
  - The second query returns the actual GPA distribution

- Besides, SQL provides the option of set semantics with `DISTINCT` keyword
--- Operational semantics of SELECT

- **SELECT [DISTINCT]** \( E_1, E_2, \ldots, E_n \)
  FROM \( R_1, R_2, \ldots, R_m \)
  WHERE \( \text{condition} \);

- For each \( t_1 \) in \( R_1 \):
  - For each \( t_2 \) in \( R_2 \): \( \ldots \ldots \)
  - For each \( t_m \) in \( R_m \):
    - If \( \text{condition} \) is true over \( t_1, t_2, \ldots, t_m \):
      Compute and output \( E_1, E_2, \ldots, E_n \)

- If **DISTINCT** is present eliminate duplicate rows in output

- \( t_1, t_2, \ldots, t_m \) are often called tuple variables
--- SQL set and bag operations

- **UNION, EXCEPT, INTERSECT**
  - Set semantics
  - Exactly like set $U$, $-$, and $\cap$ in relational algebra

- **UNION ALL, EXCEPT ALL, INTERSECT ALL**
  - Bag semantics
  - Think of each row as having an implicit count (the number of times it appears in the table)
  - Bag union: sum up the counts from two tables
  - Bag difference: proper-subtract the two counts
  - Bag intersection: take the minimum of the two counts
--- Examples of bag operations

**Bag1**

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Apple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Apple</td>
<td></td>
</tr>
<tr>
<td>Apple</td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td></td>
</tr>
</tbody>
</table>

**Bag2**

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Apple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Apple</td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td></td>
</tr>
</tbody>
</table>

**Bag1 INTERSECT ALL Bag2**

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Apple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Apple</td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td></td>
</tr>
</tbody>
</table>

**Bag1 UNION ALL Bag2**

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Apple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Apple</td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td></td>
</tr>
</tbody>
</table>

**Bag1 EXCEPT ALL Bag2**

<table>
<thead>
<tr>
<th>Fruit</th>
<th>Apple</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td></td>
</tr>
<tr>
<td>Apple</td>
<td></td>
</tr>
<tr>
<td>Orange</td>
<td></td>
</tr>
</tbody>
</table>
--- Aggregates

- Standard SQL aggregate functions: COUNT, SUM, AVG, MIN, MAX

- Example: number of students under 18, and their average GPA

  ```sql
  SELECT COUNT(*), AVG(GPA)
  FROM Student
  WHERE age < 18;
  
  COUNT(*) counts the number of rows
  ```
----- GROUP BY

- SELECT ... FROM ... WHERE ...
  GROUP BY list_of_columns;

- Example: find the average GPA for each age group

- SELECT age, AVG(GPA)
  FROM Student
  GROUP BY age;
----- Operational semantics of GROUP BY

- **SELECT ... FROM ... WHERE ... GROUP BY ...;**
  - Compute FROM ($\times$)
  - Compute WHERE ($\sigma$)
  - Compute GROUP BY: group rows according to the values of GROUP BY columns
  - Compute SELECT for each group ($\pi$)
    - One output row per group in the final output
  - An aggregate with no GROUP BY clause represent a special case where all rows go into one group
--- Restriction on SELECT

- If a query uses aggregation/GROUP BY, then every column referenced in SELECT must be either
  - Aggregated, or
  - A GROUP BY column
- This restriction ensures that any SELECT expression produces only one value for each group
--- Examples of invalid queries

- SELECT SID, age FROM Student GROUP BY age;
  - Recall there is one output row per group
  - There can be multiple SID values per group

- SELECT SID, MAX(GPA) FROM Student;
  - Recall there is only one group for an aggregate query
  - with no GROUP BY clause
  - There can be multiple SID values
  - Wishful thinking (that the output SID value is the one associated with the highest GPA) does NOT work
- HAVING

- Used to filter groups based on the group properties (e.g., aggregate values, GROUP BY column values)

- `SELECT ... FROM ... WHERE ... GROUP BY ... HAVING condition;`
  - Compute FROM (×)
  - Compute WHERE (σ)
  - Compute GROUP BY: group rows according to the values of GROUP BY columns
  - Compute HAVING (another σ over the groups)
  - Compute SELECT (π) for each group that passes HAVING
---- HAVING example

- Find the average GPA for each age group over 10
  - SELECT age, AVG(GPA)
    FROM Student
    GROUP BY age
    HAVING age > 10;

- Can be written using WHERE without table expressions

- List the average GPA for each age group with more than a hundred students
  - SELECT age, AVG(GPA)
    FROM Student
    GROUP BY age
    HAVING COUNT(*) > 100;

- Can be written using WHERE and table expressions
--- Rules for NULL’s

- When we operate on a NULL and another value (including another NULL) using +, −, etc., the result is NULL
- Aggregate functions ignore NULL, except COUNT(*) (since it counts rows)
- When we compare a NULL with another value (including another NULL) using =, >, etc., the result is UNKNOWN.
- WHERE and HAVING clauses only select rows for output if the condition evaluates to TRUE. UNKNOWN is insufficient
--- Unfortunate consequences

- `SELECT AVG(GPA) FROM Student;`
- `SELECT SUM(GPA)/COUNT(*) FROM Student;`
  - Not equivalent
  - Although \( \text{AVG}(GPA) = \frac{\text{SUM}(GPA)}{\text{COUNT}(GPA)} \), still
- `SELECT * FROM Student;`
- `SELECT * FROM Student WHERE GPA = GPA;`
  - Not equivalent
- Be careful: NULL breaks many equivalences
Another problem

Example: Who has NULL GPA values?

SELECT * FROM Student WHERE GPA = NULL;
- Does not work; never returns anything

(SELECT * FROM Student)
EXCEPT ALL
(SELECT * FROM Student WHERE GPA = GPA)
- Works, but ugly

Introduced built-in predicates IS NULL and IS NOT NULL

SELECT * FROM Student WHERE GPA IS NULL;
--- SQL constraints

- NOT NULL
- Key
- Referential integrity (foreign key)
- CHECK
--- Example

- CREATE TABLE Student
  (SID INTEGER PRIMARY KEY,
   name VARCHAR(30) NOT NULL,
   email VARCHAR(30) UNIQUE,
   age INTEGER,
   GPA FLOAT);

- CREATE TABLE Course
  (CID CHAR(10) PRIMARY KEY,
   title VARCHAR(100) NOT NULL);

- CREATE TABLE Enroll
  (SID INTEGER NOT NULL,
   CID CHAR(10) NOT NULL,
   PRIMARY KEY(SID, CID)):
--- Others

- Subqueries
  - Simple:
    - IN
  - Quantified
    - ALL
    - ANY
  - Coorelated
    - EXISTS

- Views
- Triggers
- Indexes
END of ICS 334