Lecture #02

Modern SQL
LAST CLASS

We introduced the Relational Model as the superior data model for databases.

We then showed how Relational Algebra is the building blocks that will allow us to query and modify a relational database.
SQL HISTORY

In 1971, IBM created its first relational query language called **SQUARE**.

IBM then created "SEQUEL" in 1972 for IBM **System R** prototype DBMS.
→ Structured English Query Language

IBM releases commercial SQL-based DBMSs:
In 1971, IBM created its first relational query language called **SQUARE**.

IBM then created "SEQUEL" in 1972 for **System R** prototype DBMS.

IBM releases commercial SQL-based DBMSs: **System/38** (1979), **SQL/DS** (1981), and **DB2** (1983).

---

**Q2.** Find the average salary of employees in the Shoe Department.

\[
\text{AVG (SAL | DEPT = "SHOE")}
\]

Mappings may be composed by applying one mapping to the result of another, as illustrated by Q3.

**Q3.** Find those items sold by departments on the second floor.

\[
\begin{align*}
\text{SALES} & \rightarrow \text{ITEM} \times \text{DEPT} \times \text{LOC} \\
\text{DEPT} & \rightarrow \text{FLOOR}
\end{align*}
\]

The floor ‘2’ is first mapped to the departments located there, and then to the items which they sell. The range of the inner mapping must be compatible with the domain of the outer mapping, but they need not be identical, as illustrated by Q4.
In 1971, IBM created its first relational query language called **SQUARE**.

IBM then created "SEQUEL" in 1972 for **IBM System R** prototype DBMS.
→ Structured English Query Language

IBM releases commercial SQL-based DBMSs:
SQL HISTORY

→ Structured Query Language

Current standard is SQL:2023
→ SQL:2023 → Property Graph Queries, Muti-Dim. Arrays
→ SQL:2016 → JSON, Polymorphic tables
→ SQL:2011 → Temporal DBs, Pipelined DML
→ SQL:2008 → Truncation, Fancy Sorting
→ SQL:1999 → Regex, Triggers, OO

The minimum language syntax a system needs to say that it supports SQL is SQL-92.
SQL HISTORY

Tensor and vector databases will replace most legacy systems in the next decade. A disruption fueled by natural language interfaces - new neural representations. In other words:

Natural query languages (NQL) replace the Instruction Set Architecture (ISA), i.e., (SQL).

The minimum language syntax a system needs to say that it supports SQL is SQL-92.
SQL dominated the jobs ranking in *IEEE Spectrum*’s interactive rankings of the top programming languages this year. Normally, the top position is occupied by Python or other mainstays, such as C, C++, Java, and JavaScript, but the sheer number of times employers said they wanted developers with SQL skills, albeit in addition to a more general-purpose language, boosted it to No. 1.

So what’s behind SQL’s soar to the top? The ever-increasing use of databases, for one. SQL has become the primary query language for accessing and managing data stored in such databases—specifically relational databases, which represent data in table form with rows and columns. Databases serve as the foundation of many enterprise applications and are increasingly found in other places as well, for example taking the place of traditional file systems in smartphones.

“This ubiquity means that every software developer will have to interact with databases no matter the field, and SQL is the de facto standard for interacting with databases,” says Andy Pavlo, a professor specializing in database management at the Carnegie Mellon University (CMU) School of Computer Science and a member of the CMU database group.
RELATIONAL LANGUAGES

Data Manipulation Language (DML)
Data Definition Language (DDL)
Data Control Language (DCL)

Also includes:
→ View definition
→ Integrity & Referential Constraints
→ Transactions

Important: SQL is based on bags (duplicates) not sets (no duplicates).
TODAY'S AGENDA

Aggregations + Group By
String / Date / Time Operations
Output Control + Redirection
Window Functions
Nested Queries
Lateral Joins
Common Table Expressions
**EXAMPLE DATABASE**

**student(sid,name,login,gpa)**

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
<th>login</th>
<th>age</th>
<th>gpa</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>RZA</td>
<td>rza@cs</td>
<td>44</td>
<td>4.0</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>27</td>
<td>3.9</td>
</tr>
<tr>
<td>53655</td>
<td>Tupac</td>
<td>shakur@cs</td>
<td>25</td>
<td>3.5</td>
</tr>
</tbody>
</table>

**course(cid,name)**

<table>
<thead>
<tr>
<th>cid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
<td>Database Systems</td>
</tr>
<tr>
<td>15-721</td>
<td>Advanced Database Systems</td>
</tr>
<tr>
<td>15-826</td>
<td>Data Mining</td>
</tr>
<tr>
<td>15-799</td>
<td>Special Topics in Databases</td>
</tr>
</tbody>
</table>

**enrolled(sid,cid,grade)**

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
</tbody>
</table>
AGGREGATES

Functions that return a single value from a bag of tuples:

→ **AVG**(col) → Return the average col value.
→ **MIN**(col) → Return minimum col value.
→ **MAX**(col) → Return maximum col value.
→ **SUM**(col) → Return sum of values in col.
→ **COUNT**(col) → Return # of values for col.
AGGREGATES

Aggregate functions can (almost) only be used in the \textbf{SELECT} output list.

\textit{Get \# of students with a “@cs” login:}

\begin{verbatim}
SELECT \textbf{COUNT}(login) AS cnt 
FROM student WHERE login LIKE '%@cs'
\end{verbatim}
AGGREGATES

Aggregate functions can (almost) only be used in the SELECT output list.

Get # of students with a “@cs” login:

```
SELECT COUNT(login) AS cnt
FROM student WHERE login LIKE '%@cs'
```
Aggregate functions can (almost) only be used in the **SELECT** output list.

*Get # of students with a “@cs” login:*

```
SELECT COUNT(login) AS cnt
FROM student
WHERE login LIKE '%@cs'
```

```
SELECT COUNT(*) AS cnt
FROM student
WHERE login LIKE '%@cs'
```
AGGREGATES

Aggregate functions can (almost) only be used in the **SELECT** output list.

*Get # of students with a “@cs” login:*

```sql
SELECT COUNT(login) AS cnt
FROM student
WHERE login LIKE '%@cs'
```

```sql
SELECT COUNT(*) AS cnt
FROM student
WHERE login LIKE '%@cs'
```

```sql
SELECT COUNT(1) AS cnt
FROM student
WHERE login LIKE '%@cs'
```
AGGREGATES

Aggregate functions can (almost) only be used in the **SELECT** output list.

*Get # of students with a “@cs” login:*

```
SELECT COUNT(login) AS cnt
FROM student
WHERE login LIKE '%@cs'
```

```
SELECT COUNT(*) AS cnt
FROM student
WHERE login LIKE '%@cs'
```

```
SELECT COUNT(1) AS cnt
FROM student
WHERE login LIKE '%@cs'
```

```
SELECT COUNT(1+1+1) AS cnt
FROM student
WHERE login LIKE '%@cs'
```
MULTIPLE AGGREGATES

Get the number of students and their average GPA that have a “@cs” login.

```
SELECT AVG(gpa), COUNT(sid)
FROM student
WHERE login LIKE '%@cs'
```

<table>
<thead>
<tr>
<th>AVG(gpa)</th>
<th>COUNT(sid)</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.8</td>
<td>3</td>
</tr>
</tbody>
</table>
AGGREGATES

Output of other columns outside of an aggregate is undefined.

Get the average GPA of students enrolled in each course.

```sql
SELECT AVG(s.gpa), e.cid
FROM enrolled AS e JOIN student AS s
ON e.sid = s.sid
```

<table>
<thead>
<tr>
<th>AVG(s.gpa)</th>
<th>e.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.86</td>
<td>???</td>
</tr>
</tbody>
</table>
AGGREGATES

Output of other columns outside of an aggregate is undefined.

Get the average GPA of students enrolled in each course.

```
SELECT AVG(s.gpa), e.cid
FROM enrolled AS e
JOIN student AS s
ON e.sid = s.sid
```
GROUP BY

Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
FROM enrolled AS e JOIN student AS s
ON e.sid = s.sid
GROUP BY e.cid
```
GROUP BY

Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

```
SELECT AVG(s.gpa), e.cid, s.name
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid
```
GROUP BY

Non-aggregated values in **SELECT** output clause must appear in **GROUP BY** clause.

```
SELECT AVG(s.gpa), e.cid, s.name
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid
```
GROUP BY

Non-aggregated values in SELECT output clause must appear in GROUP BY clause.

```
SELECT AVG(s.gpa), e.cid, s.name
FROM enrolled AS e JOIN student AS s
ON e.sid = s.sid
GROUP BY e.cid, s.name
```
HAVING

Filters results based on aggregation computation.
Like a **WHERE** clause for a **GROUP BY**

```sql
SELECT AVG(s.gpa) AS avg_gpa, e.cid
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
AND avg_gpa > 3.9
GROUP BY e.cid
```
HAVING

Filters results based on aggregation computation.
Like a **WHERE** clause for a **GROUP BY**

```sql
SELECT AVG(s.gpa) AS avg_gpa, e.cid
  FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid
HAVING avg_gpa > 3.9;
```
HAVING

Filters results based on aggregation computation.
Like a **WHERE** clause for a **GROUP BY**

```sql
SELECT AVG(s.gpa) AS avg_gpa, e.cid
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid
HAVING AVG(s.gpa) > 3.9;
```

<table>
<thead>
<tr>
<th>AVG(s.gpa)</th>
<th>e.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.75</td>
<td>15-415</td>
</tr>
<tr>
<td>3.950000</td>
<td>15-721</td>
</tr>
<tr>
<td>3.900000</td>
<td>15-826</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>avg_gpa</th>
<th>e.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.950000</td>
<td>15-721</td>
</tr>
</tbody>
</table>
## STRING OPERATIONS

<table>
<thead>
<tr>
<th></th>
<th>String Case</th>
<th>String Quotes</th>
</tr>
</thead>
<tbody>
<tr>
<td><strong>SQL-92</strong></td>
<td>Sensitive</td>
<td>Single Only</td>
</tr>
<tr>
<td>Postgres</td>
<td>Sensitive</td>
<td>Single Only</td>
</tr>
<tr>
<td>MySQL</td>
<td>Insensitive</td>
<td>Single/Double</td>
</tr>
<tr>
<td>SQLite</td>
<td>Sensitive</td>
<td>Single/Double</td>
</tr>
<tr>
<td>MSSQL</td>
<td>Sensitive</td>
<td>Single Only</td>
</tr>
<tr>
<td>Oracle</td>
<td>Sensitive</td>
<td>Single Only</td>
</tr>
</tbody>
</table>

WHERE `UPPER(name) = UPPER('TuPaC')`  \___**SQL-92**

WHERE `name = "TuPaC"`  \___**MySQL**
LIKE is used for string matching.

String-matching operators

→ '％' Matches any substring (including empty strings).

→ '__' Match any one character

SELECT * FROM enrolled AS e
WHERE e.cid LIKE '15-%'

SELECT * FROM student AS s
WHERE s.login LIKE '%@c_'
STRING OPERATIONS

SQL-92 defines string functions.
→ Many DBMSs also have their own unique functions
Can be used in either output and predicates:

```sql
SELECT SUBSTRING(name,1,5) AS abbrv_name
FROM student WHERE sid = 53688

SELECT * FROM student AS s
WHERE UPPER(s.name) LIKE 'KAN%'
```
SQL standard defines the `||` operator for concatenating two or more strings together.

```sql
SELECT name FROM student
WHERE login = LOWER(name) || '@cs'
```

```sql
SELECT name FROM student
WHERE login = LOWER(name) + '@cs'
```

```sql
SELECT name FROM student
WHERE login = CONCAT(LOWER(name), '@cs')
```
DATE/TIME OPERATIONS

Operations to manipulate and modify \texttt{DATE/TIME} attributes.
Can be used in both output and predicates.
Support/syntax varies wildly…

\textbf{Demo: Get the \# of days since the beginning of the year.}
OUTPUT REDIRECTION

Store query results in another table:
→ Table must not already be defined.
→ Table will have the same # of columns with the same types as the input.

```
CREATE TABLE CourseIds (
    SELECT DISTINCT cid INTO CourseIds
    FROM enrolled;

CREATE TABLE CourseIds (  
    SELECT DISTINCT cid FROM enrolled);
```

**SQL-92**

**MySQL**
OUTPUT REDIRECTION

Store query results in another table:
→ Table must not already be defined.
→ Table will have the same # of columns with the same types as the input.

```sql
CREATE TABLE CourseIds
(SELECT DISTINCT cid
FROM enrolled);
```

MySQL
```
SELECT DISTINCT cid
INTO CourseIds
FROM enrolled;
```

SQL-92

Postgres
```
SELECT DISTINCT cid
INTO TEMPORARY CourseIds
FROM enrolled;

CREATE TABLE CourseIds
(SELECT DISTINCT cid FROM enrolled);
```
Insert tuples from query into another table:
→ Inner **SELECT** must generate the same columns as the target table.
→ DBMSs have different options/syntax on what to do with integrity violations (e.g., invalid duplicates).

```
INSERT INTO CourseIds
(SELECT DISTINCT cid FROM enrolled);
```
OUTPUT CONTROL

ORDER BY <column*> [ASC|DESC]
→ Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
WHERE cid = '15-721'
ORDER BY grade
```

<table>
<thead>
<tr>
<th>sid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53123</td>
<td>A</td>
</tr>
<tr>
<td>53334</td>
<td>A</td>
</tr>
<tr>
<td>53650</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>D</td>
</tr>
</tbody>
</table>
OUTPUT CONTROL

ORDER BY <column*> [ASC|DESC]
→ Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
WHERE cid = '15-721'
ORDER BY grade
```
OUTPUT CONTROL

ORDER BY <column*> [ASC|DESC]
→ Order the output tuples by the values in one or more of their columns.

```
SELECT sid, grade FROM enrolled
WHERE cid = '15-721'
ORDER BY grade
```

```
SELECT sid FROM enrolled
WHERE cid = '15-721'
ORDER BY grade DESC, sid ASC
```

<table>
<thead>
<tr>
<th>sid</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
</tr>
<tr>
<td>53650</td>
</tr>
<tr>
<td>53123</td>
</tr>
<tr>
<td>53334</td>
</tr>
</tbody>
</table>
OUTPUT CONTROL

FETCH \{FIRST|NEXT\} <count> ROWS
OFFSET <count> ROWS
→ Limit the # of tuples returned in output.
→ Can set an offset to return a “range”

```
SELECT sid, name FROM student
  WHERE login LIKE '%@cs'
FETCH FIRST 10 ROWS ONLY;
```

```
SELECT sid, name FROM student
  WHERE login LIKE '%@cs'
  ORDER BY gpa
OFFSET 10 ROWS
FETCH FIRST 10 ROWS WITH TIES;
```
OUTPUT CONTROL

FETCH {FIRST|NEXT} <count> ROWS
OFFSET <count> ROWS
→ Limit the # of tuples returned in output.
→ Can set an offset to return a “range”

```sql
SELECT sid, name FROM student
    WHERE login LIKE '%@cs'
    FETCH FIRST 10 ROWS ONLY;

SELECT sid, name FROM student
    WHERE login LIKE '%@cs'
    ORDER BY gpa
    OFFSET 10 ROWS
    FETCH FIRST 10 ROWS WITH TIES;
```
WINDOW FUNCTIONS

Performs a "sliding" calculation across a set of tuples that are related.
Like an aggregation but tuples are not grouped into a single output tuples.

SELECT ... FUNC-NAME(...) OVER (...) FROM tableName

How to “slice” up data
Can also sort

Aggregation Functions
Special Functions
WINDBOW FUNCTIONS

Aggregation functions:
→ Anything that we discussed earlier

Special window functions:
→ \texttt{ROW\_NUMBER()} → # of the current row
→ \texttt{RANK()} → Order position of the current row.

```
SELECT *, ROW\_NUMBER() OVER () AS row\_num
FROM enrolled
```
The **OVER** keyword specifies how to group together tuples when computing the window function. Use **PARTITION BY** to specify group.

```sql
SELECT cid, sid,
       ROW_NUMBER() OVER (PARTITION BY cid)
FROM enrolled
ORDER BY cid
```
You can also include an **ORDER BY** in the window grouping to sort entries in each group.

```sql
SELECT *,
    ROW_NUMBER() OVER (ORDER BY cid)
FROM enrolled
ORDER BY cid
```
Find the student with the **second** highest grade for each course.

```
SELECT * FROM (
    SELECT *, RANK() OVER (PARTITION BY cid ORDER BY grade ASC) AS rank
    FROM enrolled) AS ranking
WHERE ranking.rank = 2
```

Group tuples by cid
Then sort by grade
NESTED QUERIES

Invoke a query inside of another query to compose more complex computations.
→ They are often difficult to optimize for the DBMS to optimize due to correlations.
→ Inner queries can appear (almost) anywhere in query.

Outer Query

```
SELECT name FROM student WHERE sid IN (SELECT sid FROM enrolled)
```

Inner Query
Get the names of students in '15-445'

```
SELECT name FROM student
WHERE ... 
```

*sid in the set of people that take 15-445*
NESTED QUERIES

Get the names of students in '15-445'

```sql
SELECT name FROM student
WHERE ...
    SELECT sid FROM enrolled
    WHERE cid = '15-445'
```
NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student
WHERE sid IN (  
    SELECT sid FROM enrolled  
    WHERE cid = '15-445'
  )
```
NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student
WHERE sid IN (
  SELECT sid FROM enrolled
  WHERE cid = '15-445'
)
```
NESTED QUERIES

**ALL** → Must satisfy expression for all rows in the sub-query.

**ANY** → Must satisfy expression for at least one row in the sub-query.

**IN** → Equivalent to '=`ANY()`'.

**EXISTS** → At least one row is returned without comparing it to an attribute in outer query.
NESTED QUERIES

Get the names of students in '15-445'

```
SELECT name FROM student
WHERE sid = ANY(
    SELECT sid FROM enrolled
    WHERE cid = '15-445'
)
```
NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```
SELECT MAX(e.sid), s.name
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid;
```

This won't work in SQL-92. It runs in SQLite, but not Postgres or MySQL (v8 with strict mode).
NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```
SELECT sid, name FROM student
WHERE ...
```

"Is the highest enrolled sid"
NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```sql
SELECT sid, name FROM student
WHERE sid IN (
    SELECT MAX(sid) FROM enrolled
)
```
Find student record with the highest id that is enrolled in at least one course.

```sql
SELECT sid, name FROM student
WHERE sid IN (
    SELECT sid FROM enrolled
    ORDER BY sid DESC FETCH FIRST 1 ROW ONLY
)
```
NESTED QUERIES

Find student record with the highest id that is enrolled in at least one course.

```sql
SELECT sid, name FROM student
WHERE sid IN (SELECT MAX(sid) FROM enrolled) AS max_e
ON student.sid = max_e.sid;
```
NESTED QUERIES

Find all courses that have no students enrolled in it.

```
SELECT * FROM course
WHERE ... “with no tuples in the enrolled table”
```

<table>
<thead>
<tr>
<th>cid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
<td>Database Systems</td>
</tr>
<tr>
<td>15-721</td>
<td>Advanced Database Systems</td>
</tr>
<tr>
<td>15-826</td>
<td>Data Mining</td>
</tr>
<tr>
<td>15-799</td>
<td>Special Topics in Databases</td>
</tr>
</tbody>
</table>

<table>
<thead>
<tr>
<th>sid</th>
<th>cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>53666</td>
<td>15-445</td>
<td>C</td>
</tr>
<tr>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>53688</td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>53655</td>
<td>15-445</td>
<td>B</td>
</tr>
<tr>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
</tbody>
</table>
NESTED QUERIES

Find all courses that have no students enrolled in it.

```
SELECT * FROM course
WHERE NOT EXISTS (  
tuples in the enrolled table
 )
```
NESTED QUERIES

Find all courses that have no students enrolled in it.

```
SELECT * FROM course
WHERE NOT EXISTS(
  SELECT * FROM enrolled
  WHERE course.cid = enrolled.cid
)
```
The **LATERAL** operator allows a nested query to reference attributes in other nested queries that precede it.

→ You can think of it like a `for` loop that allows you to invoke another query for each tuple in a table.
LATERAL JOIN

Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order.

```
SELECT * FROM course AS c,
   
   For each course:
     ➔ Compute the # of enrolled students

   For each course:
     ➔ Compute the average gpa of enrolled students
```
LATERAL JOIN

Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order.

```
SELECT * FROM course AS c,
  LATERAL (SELECT COUNT(*) AS cnt FROM enrolled
            WHERE enrolled.cid = c.cid) AS t1,
  LATERAL (SELECT AVG(gpa) AS avg FROM student AS s
           JOIN enrolled AS e ON s.sid = e.sid
           WHERE e.cid = c.cid) AS t2;
```
LATERAL JOIN

Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order.

```
SELECT * FROM course AS c,
    LATERAL (SELECT COUNT(*) AS cnt FROM enrolled
                WHERE enrolled.cid = c.cid) AS t1,
    LATERAL (SELECT AVG(gpa) AS avg FROM student AS s
                JOIN enrolled AS e ON s.sid = e.sid
                WHERE e.cid = c.cid) AS t2;
```
LATERAL JOIN

Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order.

```
SELECT * FROM course AS c,
LATERAL (SELECT COUNT(*) AS cnt FROM enrolled
WHERE enrolled.cid = c.cid) AS t1,
LATERAL (SELECT AVG(gpa) AS avg FROM student AS s
JOIN enrolled AS e ON s.sid = e.sid
WHERE e.cid = c.cid) AS t2;
```
LATERAL JOIN

Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order.

```
SELECT * FROM course AS c,
    LATERAL (SELECT COUNT(*) AS cnt FROM enrolled
         WHERE enrolled.cid = c.cid) AS t1,
    LATERAL (SELECT AVG(gpa) AS avg FROM student AS s
         JOIN enrolled AS e ON s.sid = e.sid
         WHERE e.cid = c.cid) AS t2;
```
LATERAL JOIN

Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order.

```
SELECT * 
FROM course AS c,
LATERAL (SELECT COUNT(*) AS cnt 
           FROM enrolled 
           WHERE enrolled.cid = c.cid) AS t1,
LATERAL (SELECT AVG(gpa) AS avg 
           FROM student AS s 
           JOIN enrolled AS e ON s.sid = e.sid 
           WHERE e.cid = c.cid) AS t2;
```

<table>
<thead>
<tr>
<th>cid</th>
<th>name</th>
<th>cnt</th>
<th>avg</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
<td>Database Systems</td>
<td>2</td>
<td>3.75</td>
</tr>
<tr>
<td>15-721</td>
<td>Advanced Database Systems</td>
<td>2</td>
<td>3.95</td>
</tr>
<tr>
<td>15-826</td>
<td>Data Mining</td>
<td>1</td>
<td>3.9</td>
</tr>
<tr>
<td>15-799</td>
<td>Special Topics in Databases</td>
<td>0</td>
<td>null</td>
</tr>
</tbody>
</table>
COMMON TABLE EXPRESSIONS

Provides a way to write auxiliary statements for use in a larger query.
→ Think of it like a temp table just for one query.
Alternative to nested queries and views.

WITH cteName AS (  
    SELECT 1  
  )  
SELECT * FROM cteName
COMMON TABLE EXPRESSIONS

You can bind/alias output columns to names before the `AS` keyword.

```sql
WITH cteName (col1, col2) AS (
    SELECT 1, 2
) 
SELECT col1 + col2 FROM cteName

WITH cteName (colXXX, colXXX) AS (
    SELECT 1, 2
) 
SELECT colXXX + colXXX FROM cteName
```
You can bind/alias output columns to names before the **AS** keyword.

```sql
WITH cteName (col1, col2) AS (
    SELECT 1, 2
)
SELECT col1 + col2 FROM cteName
```

```sql
WITH cteName (colXXX, colXXX) AS (
    SELECT 1, 2
)
SELECT colXXX + colXXX FROM cteName
```
COMMON TABLE EXPRESSIONS

You can bind/alias output columns to names before the AS keyword.

WITH cteName (col1, col2) AS (  
    SELECT 1, 2  
)  
SELECT col1 + col2 FROM cteName

WITH cteName (colXXX, colXXX) AS (Postgres  
    SELECT 1, 2  
)  
SELECT * FROM cteName
COMMON TABLE EXPRESSIONS

Find student record with the highest id that is enrolled in at least one course.

WITH cteSource (maxId) AS (
    SELECT MAX(sid) FROM enrolled
)
SELECT name FROM student, cteSource
WHERE student.sid = cteSource.maxId
CONCLUSION

SQL is not a dead language.

You should (almost) always strive to compute your answer as a single SQL statement.
HOMEWORK #1

Write SQL queries to perform basic data analysis.
→ Write the queries locally using SQLite + DuckDB.
→ Submit them to Gradescope
→ You can submit multiple times and use your best score.

Due: Sunday Sept 10th @ 11:59pm

https://15445.courses.cs.cmu.edu/fall2023/homework1
NEXT CLASS

Storage Management