Intro to Database Systems (15-445/645)

02 Modern SQL

Carnegie Mellon University

SPRING 2023

Charlie Garrod
ADMINISTRIVIA

Project 0 due Sunday. Quick adaptation to C++ is a prerequisite for this course!

Homework 1 released today, due Friday Feb 3rd.
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.
ASIDE: OTHER DATA MODELS

- Relational
- Key/Value
- Graph
- **Document / Object** ← Leading Alternative
- Wide-Column / Column-family
- Array / Matrix / Vectors
- Hierarchical
- Network
- Multi-Value
ASIDE: DOCUMENT DATA MODEL

Embed data hierarchy into a single object.

**Application Code**

```java
class Artist {
    int id;
    String name;
    int year;
    Album albums[];
}

class Album {
    int id;
    String name;
    int year;
}
```

```json
{
    "name": "GZA",
    "year": 1990,
    "albums": [
        {
            "name": "Liquid Swords",
            "year": 1995
        },
        {
            "name": "Beneath the Surface",
            "year": 1999
        }
    ]
}
```
The relational model is independent of any query language implementation.

**SQL** is the *de facto* standard (many dialects).

```python
for line in file.readlines():
    record = parse(line)
    if record[0] == "GZA":
        print(int(record[1]))
```

```sql
SELECT year FROM artists
WHERE name = 'GZA';
```
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 releases:

**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.

\[
\text{SALES | LOC = '2' \rightarrow DEPT | DEPT | FLOOR (2)} \]

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.
SQL HISTORY

→ Structured Query Language

Current standard is SQL:2016
→ SQL:2016 → JSON, Polymorphic tables
→ SQL:2011 → Temporal DBs
→ 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.
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 multisets (a.k.a. bags, with duplicates), not sets (no duplicates).
TODAY'S AGENDA

- Aggregations + Group By
- String / Date / Time Operations
- Output Control + Redirection
- Nested Queries
- Window Functions
- 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>Kanye</td>
<td>kanye@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>

**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>

**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>
AGGREGATES

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

→ \texttt{AVG(col)} \rightarrow \text{Return the average col value.}
→ \texttt{MIN(col)} \rightarrow \text{Return minimum col value.}
→ \texttt{MAX(col)} \rightarrow \text{Return maximum col value.}
→ \texttt{SUM(col)} \rightarrow \text{Return sum of values in col.}
→ \texttt{COUNT(col)} \rightarrow \text{Return \# of values for col.}
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>
DISTINCT AGGREGATES

COUNT, SUM, AVG support DISTINCT

Get the number of unique students that have an “@cs” login.

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

COUNT(DISTINCT login) 3
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
```

<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>
Project tuples into subsets and calculate aggregates against each subset.

```sql
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 \texttt{SELECT} output clause must appear in \texttt{GROUP BY} clause.

\begin{verbatim}
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
\end{verbatim}
HAVING

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

```
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>SQL-92</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>

### Examples

- **SQL-92**
  
  ```sql
  WHERE UPPER(name) = UPPER('KaNyE')
  ```

- **MySQL**
  
  ```sql
  WHERE name = "KaNyE"
  ```
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%'
```
STRING OPERATIONS

SQL standard says to use `||` operator to concatenate 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 DATE/TIME attributes.
Can be used in both output and predicates.
Support/syntax varies wildly…
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;
```

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

`SQL-92`

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

`Postgres`
OUTPUT REDIRECTION

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

SELECT sid FROM enrolled
WHERE cid = '15-721'
ORDER BY grade DESC, 1 ASC
OUTPUT CONTROL

**LIMIT** `<count> [offset]`
- 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'
LIMIT 10
```

```sql
SELECT TOP 10 sid, name FROM student
WHERE login LIKE '%@cs'
```

```sql
SELECT sid, name FROM student
WHERE login LIKE '%@cs'
LIMIT 10 OFFSET 20
```
NESTED QUERIES

Queries containing other queries.
They are often difficult to optimize.

Inner queries can appear (almost) anywhere in query.

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

```sql
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.

```sql
SELECT sid, name FROM student WHERE sid IN (SELECT sid FROM enrolled ORDER BY sid DESC LIMIT 1)
```

```sql
SELECT student.sid, name FROM student JOIN (SELECT MAX(sid) AS 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 NOT EXISTS(
    SELECT * FROM enrolled
    WHERE course.cid = enrolled.cid
)
```

<table>
<thead>
<tr>
<th>cid</th>
<th>name</th>
<th>sid</th>
<th>course cid</th>
<th>grade</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-721</td>
<td>Advanced Database Systems</td>
<td>53688</td>
<td>15-721</td>
<td>A</td>
</tr>
<tr>
<td>15-826</td>
<td>Data Mining</td>
<td></td>
<td>15-826</td>
<td>B</td>
</tr>
<tr>
<td>15-799</td>
<td>Special Topics in Databases</td>
<td></td>
<td>15-799</td>
<td>B</td>
</tr>
<tr>
<td>15-799</td>
<td>Special Topics in Databases</td>
<td>53666</td>
<td>15-721</td>
<td>C</td>
</tr>
</tbody>
</table>
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
WINDOW FUNCTIONS

Aggregation functions:
→ Anything that we discussed earlier

Special window functions:
→ `ROW_NUMBER()` → # of the current row
→ `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.

```
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.

```
SELECT *,
    ROW_NUMBER() OVER (ORDER BY cid)
FROM enrolled
ORDER BY cid
```
WINDOW FUNCTIONS

Find the student(s) 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
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.

```sql
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
```

```sql
WITH cteName (colXXX, colXXX) AS (  
    SELECT 1, 2  
)  
SELECT * FROM cteName
```
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
Print the sequence of numbers from 1 to 10.

WITH RECURSIVE cteSource (counter) AS (  
  (SELECT 1)  
  UNION ALL  
  (SELECT counter + 1 FROM cteSource  
   WHERE counter < 10)  
)  
SELECT * FROM cteSource

Demo: CTEs!
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.
→ Submit them to Gradescope
→ You can submit multiple times and use your best score.

Due: Friday, Feb 3rd @ 11:59pm

https://15445.courses.cs.cmu.edu/spring2023/homework1
NEXT CLASS

Storage Management