Advanced SQL
DATABASE RESEARCH

Database Group Meetings
→ Mondays @ 4:30pm (GHC 8102)
→ http://db.cs.cmu.edu

Database Seminar Series
→ http://db.cs.cmu.edu/seminar2017
DATABASE RESEARCH

Carnegie Mellon University

TIMESERIES
DATABASE LECTURES
Fall 2017 • Thursdays @ 12:00pm • CIC 4th Floor
RELATIONAL LANGUAGES

User only needs to specify the answer that they want, not how to compute it.

The DBMS is responsible for efficient evaluation of the query.
→ Query optimizer: re-orders operations and generates query plan
RELATIONAL LANGUAGES

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

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

Important: SQL is based on bags (duplicates) not sets (no duplicates).
HISTORY

Originally “SEQUEL” from IBM’s System R prototype.
- Structured English Query Language
- Adopted by Oracle in the 1970s.

IBM releases DB2 in 1983.

- Structured Query Language
HISTORY

Current standard is **SQL:2016**
- **SQL:2016** → JSON, Polymorphic tables
- **SQL:2011** → Temporal DBs, Pipelined DML
- **SQL:2008** → TRUNCATE, Fancy ORDER
- **SQL:2003** → XML, windows, sequences, auto-generated IDs.
- **SQL:1999** → Regex, triggers, OO

Most DBMSs at least support **SQL-92**
- System Comparison: [http://troels.arvin.dk/db/rdbms/](http://troels.arvin.dk/db/rdbms/)

CMU 15-445/645 (Fall 2017)
TODAY’S AGENDA

Aggregations + Group By
String / Date / Time Operations
Output Control + Redirection
Nested Queries
Common Table Expressions
Window Functions
### Example Database

The database contains three tables:

- **student(sid, name, login, gpa)**
- **enrolled(sid, cid, grade)**
- **course(cid, name)**

#### student Table

<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>kayne@cs</td>
<td>39</td>
<td>4.0</td>
</tr>
<tr>
<td>53688</td>
<td>Bieber</td>
<td>jbieber@cs</td>
<td>22</td>
<td>3.9</td>
</tr>
<tr>
<td>53655</td>
<td>Tupac</td>
<td>shakur@cs</td>
<td>26</td>
<td>3.5</td>
</tr>
</tbody>
</table>

#### enrolled 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>

#### course 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>15-823</td>
<td>Advanced Topics in Databases</td>
</tr>
</tbody>
</table>
FUNCTIONS THAT RETURN A SINGLE VALUE FROM A BAG OF TUPLES:

- \texttt{AVG(col)} ➞ Return the average \texttt{col} value.
- \texttt{MIN(col)} ➞ Return minimum \texttt{col} value.
- \texttt{MAX(col)} ➞ Return maximum \texttt{col} value.
- \texttt{SUM(col)} ➞ Return sum of values in \texttt{col}.
- \texttt{COUNT(col)} ➞ Return # of values for \texttt{col}.
AGGREGATES

Aggregate functions can 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'
```
AGGREGATES

Aggregate functions can 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 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'

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

SELECT COUNT(1) AS cnt
FROM student
WHERE login LIKE '%@cs'
```
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>3.25</td>
<td>12</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)
10
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, student AS s
WHERE e.sid = s.sid
```
### GROUP BY

Project tuples into subsets and calculate aggregates against each subset.

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

<table>
<thead>
<tr>
<th>e.sid</th>
<th>s.sid</th>
<th>s.gpa</th>
<th>e.cid</th>
</tr>
</thead>
<tbody>
<tr>
<td>53435</td>
<td>53435</td>
<td>2.25</td>
<td>15-721</td>
</tr>
<tr>
<td>53439</td>
<td>53439</td>
<td>2.70</td>
<td>15-721</td>
</tr>
<tr>
<td>56023</td>
<td>56023</td>
<td>2.75</td>
<td>15-826</td>
</tr>
<tr>
<td>59439</td>
<td>59439</td>
<td>3.90</td>
<td>15-826</td>
</tr>
<tr>
<td>53961</td>
<td>53961</td>
<td>3.50</td>
<td>15-826</td>
</tr>
<tr>
<td>58345</td>
<td>58345</td>
<td>1.89</td>
<td>15-445</td>
</tr>
</tbody>
</table>
GROUP BY

Project tuples into subsets and calculate aggregates against each subset.

```
SELECT AVG(s.gpa), e.cid
FROM enrolled AS e, student AS s
WHERE e.sid = s.sid
GROUP BY e.cid
```
Non-aggregated values in `SELECT` output clause must appear in `GROUP BY` clause.
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, s.name
```
HAVING

Filters output results
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
    AND avg_gpa > 3.9
    GROUP BY e.cid
```
HAVING

Filters output results
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_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>DB2</td>
<td>Sensitive</td>
<td>Single Only</td>
</tr>
<tr>
<td>Oracle</td>
<td>Sensitive</td>
<td>Single Only</td>
</tr>
</tbody>
</table>

```sql
WHERE UPPER(name) = UPPER('KaNyE')  SQL-92
WHERE name = "KaNyE"  MySQL
```
STRING OPERATIONS

**LIKE** is used for string matching.

String-matching operators

→ "%" Matches any substring (including empty strings).
→ "_" Match any one character

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

```sql
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:

```
SELECT SUBSTRING(name,0,5) AS abbrv_name
FROM student WHERE sid = 53688
```

```
SELECT * FROM student AS s
WHERE UPPER(e.name) LIKE 'KAN%'
```
SQL standard says to use `||` operator to concatenate two or more strings together.

- **SQL-92**
  ```sql
  SELECT name FROM student
  WHERE login = LOWER(name) || '@cs'
  ```

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

- **MySQL**
  ```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 either output and predicates.
Support/syntax varies wildly...

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.

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

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

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

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

```sql
INSERT INTO CourseIds
(SELECT DISTINCT cid FROM enrolled);
```
ORDER BY `<column*>` [ASC|DESC]

→ Order the output tuples by the values in one or more of their columns.

<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>

```
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
```
OUT PUT C O N T R O L

LIMIT <count> [offset]
→ Limit the # of tuples returned in output.
→ Can set an offset to return a “range”

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

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

Think of this as a function that returns the result of the inner query
NESTED QUERIES

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'

```
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

**ALL** ➔ Must satisfy expression for all rows in sub-query

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

**IN** ➔ Equivalent to '=*ANY()'.

**EXISTS** ➔ At least one row is returned.
**NESTED QUERIES**

Get the names of students in ‘15-445’

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

Get the names of students in ‘15-445’

```
SELECT (SELECT S.name FROM student AS S
    WHERE S.sid = E.sid) AS sname
    FROM enrolled AS E
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;
```

Won't work in SQL-92.
This runs in SQLite, but not Postgres or MySQL (as of v5.7).
NESTED QUERIES

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

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

"Is greater than every other sid"
NESTED QUERIES

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

```
SELECT sid, name FROM student
WHERE sid is greater than every
SELECT sid FROM enrolled
```
NESTED QUERIES

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

```
SELECT sid, name FROM student
WHERE sid => ALL(
    SELECT sid FROM enrolled
);
```

<table>
<thead>
<tr>
<th>sid</th>
<th>name</th>
</tr>
</thead>
<tbody>
<tr>
<td>53688</td>
<td>Bieber</td>
</tr>
</tbody>
</table>
NESTED QUERIES

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

```
SELECT sid, name FROM student
WHERE sid IN (SELECT MAX(sid) FROM enrolled)
```
NESTED QUERIES

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

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

Find all courses that has 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>15-823</td>
<td>Advanced 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 has no students enrolled in it.

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

Find all courses that has 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>
</tr>
</thead>
<tbody>
<tr>
<td>15-823</td>
<td>Advanced Topics in Databases</td>
</tr>
</tbody>
</table>
WINDOW FUNCTIONS

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

```
SELECT ... FUNC-NAME(...) OVER (...
FROM tableName
```
Aggregation functions:
→ Anything that we discussed earlier

Special window functions:
→ **ROW_NUMBER()** → Number of the current row
→ **RANK()** → Order position of the current row.

```sql
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
```

<table>
<thead>
<tr>
<th>cid</th>
<th>sid</th>
<th>row_number</th>
</tr>
</thead>
<tbody>
<tr>
<td>15-445</td>
<td>53666</td>
<td>1</td>
</tr>
<tr>
<td>15-445</td>
<td>53655</td>
<td>2</td>
</tr>
<tr>
<td>15-721</td>
<td>53688</td>
<td>1</td>
</tr>
<tr>
<td>15-721</td>
<td>53666</td>
<td>2</td>
</tr>
<tr>
<td>15-826</td>
<td>53688</td>
<td>1</td>
</tr>
</tbody>
</table>
You can also include an **ORDER BY** in the window grouping.

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

Find the student with the 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 = 1
```
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 output columns to names before the AS keyword.

```sql
WITH cteName (col1, col2) AS (  
  SELECT 1, 2  
)  
SELECT col1 + col2 FROM cteName
```
WITH cteSource (maxId) AS (
SELECT MAX(sid) FROM enrolled
)
SELECT name FROM student, cteSource
WHERE student.sid = cteSource.maxId

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

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: Postgres CTE!
CONCLUSION

SQL is not a dead language.

You should (almost) always strive to compute your answer as a single SQL statement.
NEXT CLASS

Normal Forms
Functional Dependencies
Schema Refinement