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.

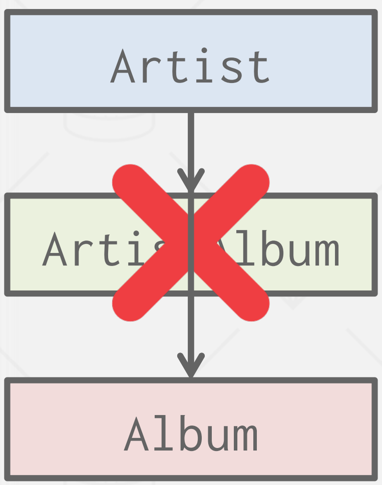
# ASIDE: OTHER DATA MODELS

Relational
Key/Value
Graph
Document / Object ← Leading Alternative
Wide-Column / Column-family
Array / Matrix / Vectors
Hierarchical
Network
Multi-Value

mongoDB
Couchbase
elastic
fauna
splunk

# ASIDE: DOCUMENT DATA MODEL

Embed data hierarchy into a single object.




***Application Code***

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

```
{
 "name": "GZA",
 "year": 1990,
 "albums": [
  {
   "name": "Liquid Swords",
   "year": 1995
  },
  {
   "name": "Beneath the Surface",
   "year": 1999
  }
 ]
}
```

# RELATIONAL MODEL: QUERIES

The relational model is independent of any query language implementation.

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

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

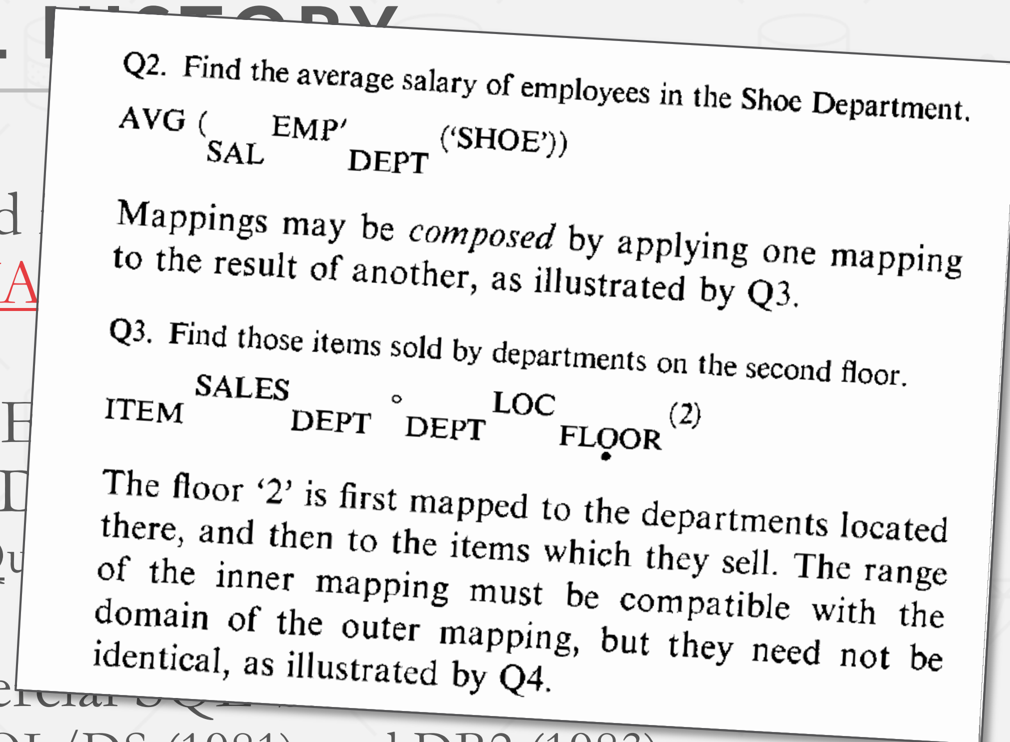
```
SELECT year FROM artists
 WHERE name = 'GZA';
```

# SQL

In 1971, IBM created
language called SQUA

IBM then created "SE
System R prototype D
→ Structured English Qu

IBM releases commer
→ System/38 (1979), SQL/DS (1981), and DB2 (1983).


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

$AVG\,({}_{SAL}\,EMP'_{DEPT}\,(\text{'SHOE'}))$

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.

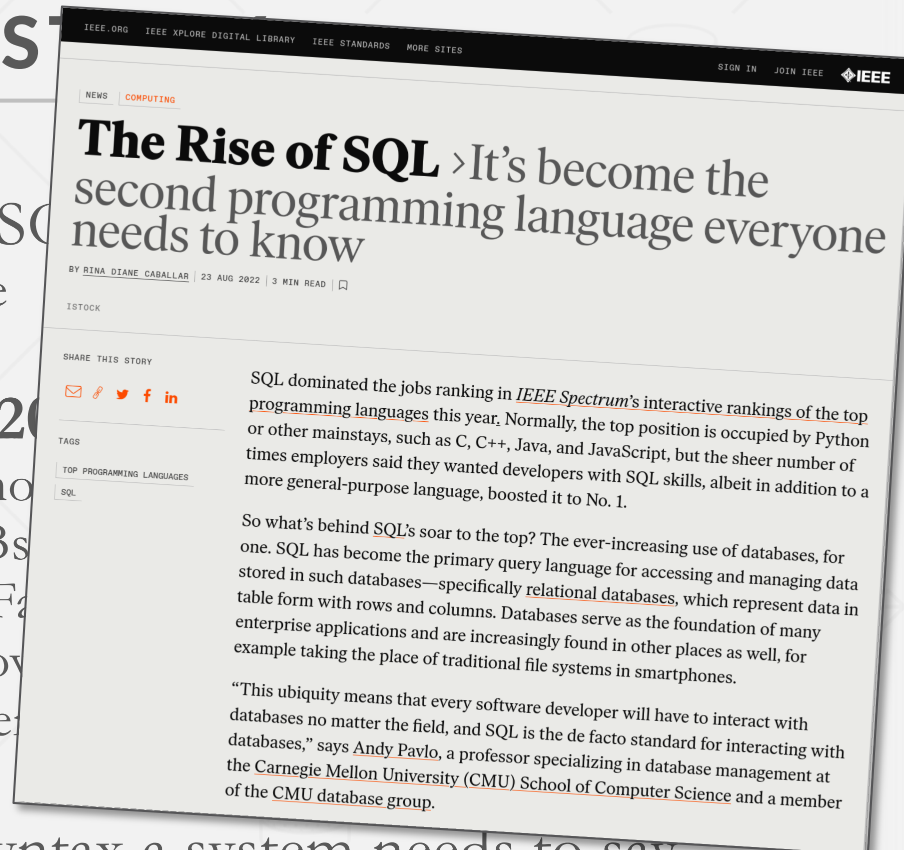
$ITEM\,{}^{SALES}{}_{DEPT} \circ DEPT\,{}^{LOC}{}_{FLOOR}\,(2)$

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 HIST

ANSI Standard in 1986. ISO
→ Structured Query Language

Current standard is **SQL:20**
→ **SQL:2016** → JSON, Polymo
→ **SQL:2011** → Temporal DBs
→ **SQL:2008** → Truncation, Fa
→ **SQL:2003** → XML, Windov
→ **SQL:1999** → Regex, Trigge



IEEE.ORG IEEE XPLORE DIGITAL LIBRARY IEEE STANDARDS MORE SITES SIGN IN JOIN IEEE IEEE

NEWS COMPUTING

**The Rise of SQL** › It's become the second programming language everyone needs to know

BY RINA DIANE CABALLAR | 23 AUG 2022 | 3 MIN READ

ISTOCK

SHARE THIS STORY

TAGS

TOP PROGRAMMING LANGUAGES

SQL

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.

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)

| sid | name | login | age | gpa |
|---|---|---|---|---|
| 53666 | Kanye | kanye@cs | 44 | 4.0 |
| 53688 | Bieber | jbieber@cs | 27 | 3.9 |
| 53655 | Tupac | shakur@cs | 25 | 3.5 |

course(cid,name)

| cid | name |
|---|---|
| 15-445 | Database Systems |
| 15-721 | Advanced Database Systems |
| 15-826 | Data Mining |
| 15-799 | Special Topics in Databases |

enrolled(sid,cid,grade)

| sid | cid | grade |
|---|---|---|
| 53666 | 15-445 | C |
| 53688 | 15-721 | A |
| 53688 | 15-826 | B |
| 53655 | 15-445 | B |
| 53666 | 15-721 | C |

# 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 SELECT output list.

*Get # of students with a "@cs" login:*

```
SELECT COUNT(login) AS cnt
```

```
SELECT COUNT(*) AS cnt
```

```
SELECT COUNT(1) AS cnt
```

```
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'
```

| AVG(gpa) | COUNT(sid) |
|---|---|
| 3.8 | 3 |

# 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
```

| AVG(s.gpa) | e.cid |
|---|---|
| 3.86 | ??? |

# 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
```

| e.sid | s.sid | s.gpa | e.cid |
|---|---|---|---|
| 53435 | 53435 | 2.25 | 15-721 |
| 53439 | 53439 | 2.70 | 15-721 |
| 56023 | 56023 | 2.75 | 15-826 |
| 59439 | 59439 | 3.90 | 15-826 |
| 53961 | 53961 | 3.50 | 15-826 |
| 58345 | 58345 | 1.89 | 15-445 |

| AVG(s.gpa) | e.cid |
|---|---|
| 2.46 | 15-721 |
| 3.39 | 15-826 |
| 1.89 | 15-445 |

# 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**

```
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;
```

| AVG(s.gpa) | e.cid |
|---|---|
| 3.75 | 15-415 |
| 3.950000 | 15-721 |
| 3.900000 | 15-826 |

→

| avg_gpa | e.cid |
|---|---|
| 3.950000 | 15-721 |

# STRING OPERATIONS

| | String Case | String Quotes |
|---|---|---|
| SQL-92 | Sensitive | Single Only |
| Postgres | Sensitive | Single Only |
| MySQL | Insensitive | Single/Double |
| SQLite | Sensitive | Single/Double |
| MSSQL | Sensitive | Single Only |
| Oracle | Sensitive | Single Only |

```
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

```
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:

```
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-92*

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

*MSSQL*

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

*MySQL*

```
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-92*

```
SELECT DISTINCT cid INTO CourseIds
  FROM
```

*Postgres*

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

```
CREATE
  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 integrity violations (e.g., invalid duplicates).

SQL-92

```
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
 WH
 OR
```

| sid | grade |
|---|---|

```
SELECT sid, grade FROM enrolled
 WHERE cid = '15-721'
 ORDER BY 1
```

```
SELECT sid FROM enrolled
 WH
 OR
```

| sid |
|---|

```
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'
```

*MSSQL*

```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.

**Outer Query**

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

**Inner Query**

# 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'
 )
```

45

# 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
 WHER
   SE
)
```

| sid | name |
|---|---|

```
SELECT sid, name FROM student
 WHERE sid IN (
   SE
    (
)
```

```
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
)
```

| cid | |
|---|---|
| 15-445 | Database Systems |
| 15-721 | Adva |
| 15-826 | Data |
| 15-799 | Special Topics in Databases |

| | | |
|---|---|---|
| 53666 | 15-445 | C |
| | | A |
| | | B |
| | | B |
| 53666 | 15-721 | C |

| cid | name |
|---|---|
| 15-799 | Special Topics in Databases |

# 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.

*How to "slice" up data*
*Can also sort*

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

*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.

| sid | cid | grade | row_num |
|---|---|---|---|
| 53666 | 15-445 | C | 1 |
| 53688 | 15-721 | A | 2 |
| 53688 | 15-826 | B | 3 |
| 53655 | 15-445 | B | 4 |
| 53666 | 15-721 | C | 5 |

```
SELECT *, ROW_NUMBER() OVER () AS row_num
  FROM enrolled
```

# WINDOW FUNCTIONS

The **OVER** keyword specifies how to group together tuples when computing the window function.

Use **PARTITION BY** to specify group.

| cid | sid | row_number |
|---|---|---|
| 15-445 | 53666 | 1 |
| 15-445 | 53655 | 2 |
| 15-721 | 53688 | 1 |
| 15-721 | 53666 | 2 |
| 15-826 | 53688 | 1 |

```
SELECT cid, sid,
    ROW_NUMBER() OVER (PARTITION BY cid)
  FROM enrolled
 ORDER BY cid
```

# WINDOW FUNCTIONS

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.*

**Group tuples by cid**
**Then sort by grade**

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

# 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.

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

```
WITH cteName (colXXX, colXXX) AS (
   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
```

# 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: CTEs!**

# 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.
→ 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