Carnegie Mellon University

Advanced SQL

Lecture #02

Database Systems
15-445/15-645
Fall 2018

AP

Andy Pavlo
Computer Science
Carnegie Mellon Univ.

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

SQL HISTORY

Originally "SEQUEL" from IBM's **System R** prototype.

- → Structured English Query Language
- → Adopted by Oracle in the 1970s.

Original name SEQUEL

IBM releases DB2 in 1983.

ANSI Standard in 1986. ISO in 1987

- → Structured Query Language

SQL 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

Even though there is a standard, nobody actually follows it

Most DBMSs at least support **SQL-92**

→ System Comparison: http://troels.arvin.dk/db/rdbms/

RELATIONAL LANGUAGES

Data Manipulation Language (DML)

Data Definition Language (DDL)

Data Control Language (DCL)

security authorization

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

Nested Queries

Common Table Expressions

Window Functions

EXAMPLE DATABASE

student(sid,name,login,gpa)

sid	name	login	age	gpa
53666	Kanye	kayne@cs	39	4.0
53688	Bieber	jbieber@cs	22	3.9
53655	Tupac	shakur@cs	26	3.5

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

course(cid,name)

cid	name
15-445	Database Systems
15-721	Advanced Database Systems
15-826	Data Mining
15-823	Advanced Topics in Databases

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 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'
```

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)
3.25	12

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

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, student AS s
 WHERE e.sid = s.sid
```

AVG(s.gpa)	e.cid
3.5	???

The way they fix this

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
```

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

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
```

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

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
```

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, 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, 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
   AND avg_gpa > 3.9
 GROUP BY e.cid
```

This doesn’t work because we can’t access anything in our aggregations in our where clause because we don’t have them yet in our where clause.

The where clause is filtering tuples as we go along and after we do our filtering then we can actually compute aggregation

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_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
DB2	Sensitive	Single Only
Oracle	Sensitive	Single Only

SQL-92

```
WHERE UPPER(name) = UPPER('KaNyE')
```

MySQL

```
WHERE name = "KaNyE"
```

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,0,5) AS abbrv_name
  FROM student WHERE sid = 53688
```

```
SELECT * FROM student AS s
 WHERE UPPER(e.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 either output and predicates.

Support/syntax varies wildly…

Demo: Get the # of days since the beginning of the year.

There is no standard way to do this even through there is a standard specification

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 enrolled;
```

MySQL

```
CREATE TABLE CourseIds (
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-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
 WHERE cid = '15-721'
 ORDER BY grade
```

sid	grade
53123	A
53334	A
53650	B
53666	D

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
```

sid	grade
53123	A
53334	A
53650	B
53666	D

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

sid
53666
53650
53123
53334

OUTPUT CONTROL

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
```

Search results

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 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'
```

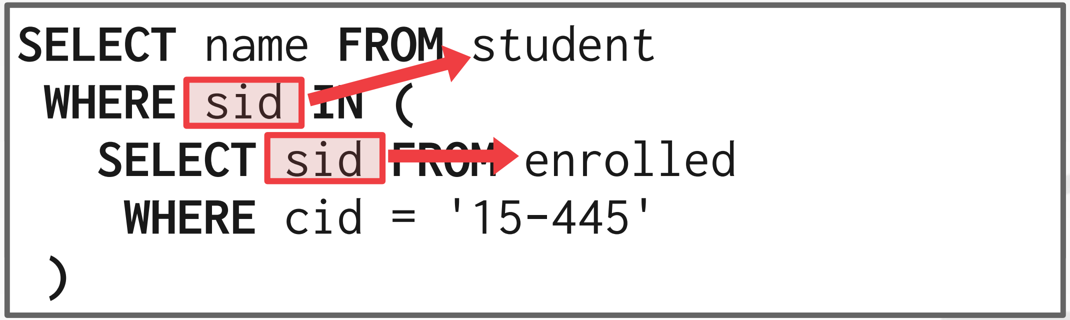
NESTED QUERIES

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

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'

```
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 (v5.7 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 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
)
```

sid	name
53688	Bieber

NESTED QUERIES

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

```
SELECT sid, name FROM student
 WHE
   SE
)
```

```
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
 WHER
   SE
)
```

```
SELECT sid, name FROM student
 WHER
   SE
)
```

```
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”

cid	name
15-445	Database Systems
15-721	Advanced Database Systems
15-826	Data Mining
15-823	Advanced Topics in Databases

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

NESTED QUERIES

Find all courses that has no students enrolled in it.

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

Can you think of a nested queries as nested for loop?

Ans: Yes but no

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
)
```

cid	name
15-823	Advanced Topics in Databases

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
)
```

cid	name
15-823	Advanced Topics in Databases

WINDOW FUNCTIONS

Performs a calculation across a set of tuples that related to a single row.

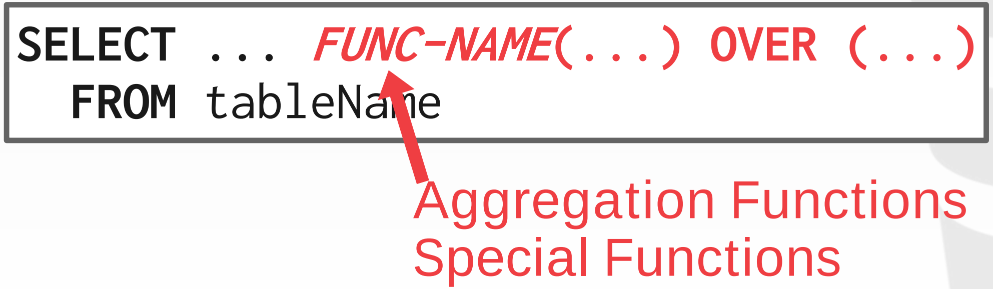
Like an aggregation but tuples are not grouped into a single output tuples.

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

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.



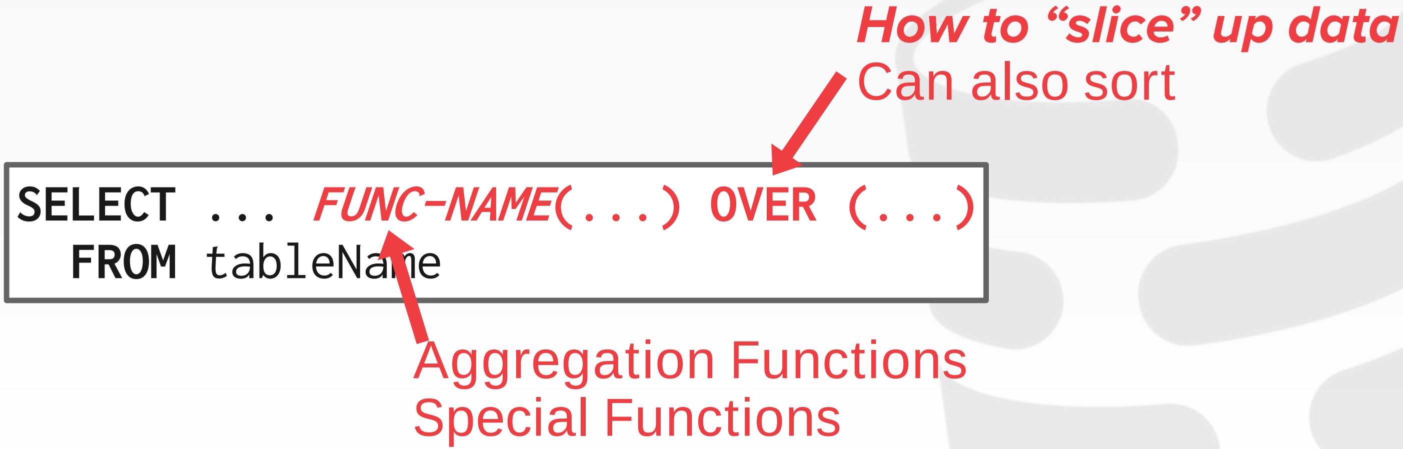
```
SELECT ... FUNC-NAME(...) OVER (...)
  FROM tableName
```

Aggregation Functions
Special Functions

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
```

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
```

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

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

```sql
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.

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

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

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 with the 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 = 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.

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

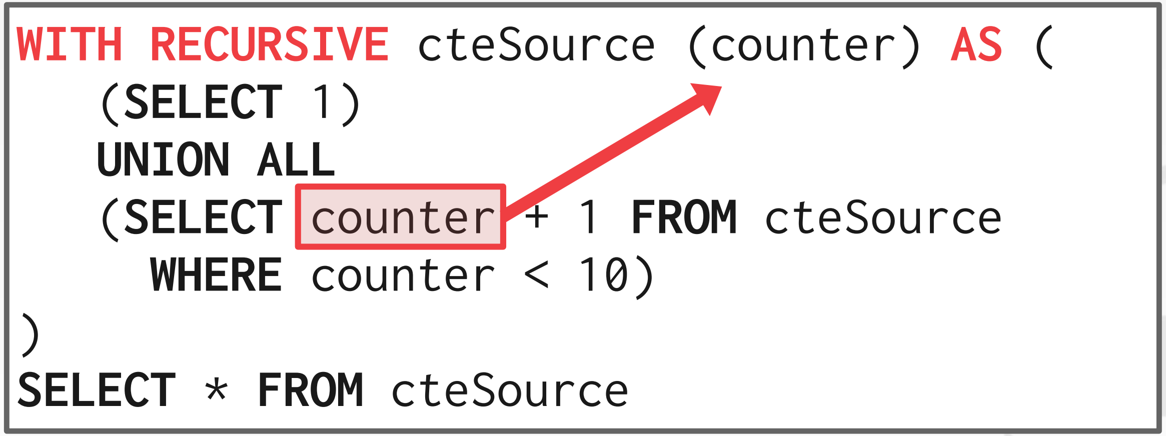
COMMON TABLE EXPRESSIONS

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: Postgres CTE!

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 on bike-sharing data from SFO.

→ Write the queries locally using SQLite.

→ Submit them to Gradescope

→ You can submit multiple times. We track your best score.

Due: Monday Sept 10th @ 11:59pm

https://15445.courses.cs.cmu.edu/fall2018/homework1/

NEXT CLASS

Storage Management