

Carnegie Mellon University

# Database Systems

## Modern SQL

15-445/645 SPRING 2024 » PROF. JIGNESH PATEL

# TODAY'S AGENDA

---

Database Systems Background

Relational Model

Relational Algebra

Alternative Data Models

Q&A Session

# 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:

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

# SQL HISTORY

---

ANSI Standard in 1986. ISO in 1987

→ 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:2003** → XML, Windows, Sequences, Auto-Gen IDs.

→ **SQL:1999** → Regex, Triggers, OO

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

# SQL HISTORY



s, Muti-Dim. Arrays  
 es  
 d DML  
 ng  
 nces, Auto-Gen IDs.

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

# SQL HISTORY



987



The minimum language to say that it supports

# 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.



Jo Kristian Bergum @jobergum

Tensor and vector da decade. A disruption neural representatio

Natural query langu (SQL).

2:35 AM · Apr 27, 2022

39 Retweets 32 Qu

ious. r company's data and chatbot for the data and

marks



# 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)**

sid	name	login	age	gpa
53666	RZA	rza@cs	55	4.0
53688	Taylor	swift@cs	27	3.9
53655	Tupac	shakur@cs	25	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
15-799	Special 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 (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'
```

# 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'
```

# 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'
```

# 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'
```



# 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

# 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	???

# 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	???

# 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	???

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

AVG(s.gpa)	e.cid
3.86	15-445

# 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

# 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

# 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.

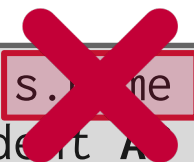
```
SELECT AVG(s.gpa), e.cid, s.name
  FROM enrolled AS e, student AS s
 WHERE e.sid = s.sid
GROUP BY e.cid
```

# 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**

```
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**

```
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**

```
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**

```
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**

```
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**

```
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;
```

# 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
<b>SQL-92</b>	<b>Sensitive</b>	<b>Single Only</b>
Postgres	Sensitive	Single Only
MySQL	Insensitive	Single/Double
SQLite	Sensitive	Single/Double
MSSQL	Sensitive	Single Only
Oracle	Sensitive	Single Only

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

**WHERE name = "TuPaC"** **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 abbrev_name  
FROM student WHERE sid = 53688
```

```
SELECT * FROM student AS s  
WHERE UPPER(s.name) LIKE 'KAN%'
```

# STRING OPERATIONS

---

SQL standard defines the **||** operator for concatenating two or more strings together.

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

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

```
SELECT name FROM student MySQL  
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...

**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.

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

*SQL-92*

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

*Postgres*

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

*MySQL*



# 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 SQL-92  
(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 2
```

# 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 2
```

```
SELECT sid FROM enrolled
```

```
WHERE cid = '15-721'
```

```
ORDER BY grade DESC, sid ASC
```

# 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 2
```

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

sid
53666
53650
53123
53334

# 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;
```

# 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”

```
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

---

Conceptual execution: (optional) Partition data → (optional) sort each partition  
→ for each record, creates window → compute an answer for each window.

# WINDOW FUNCTIONS

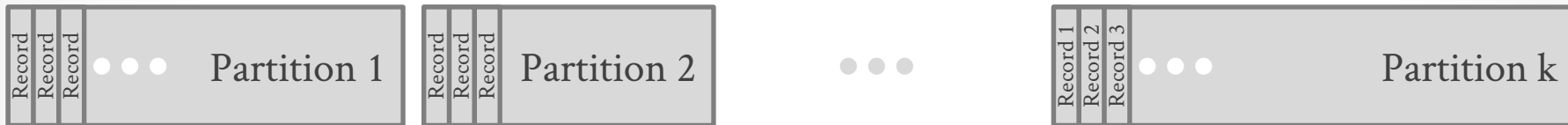
---

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



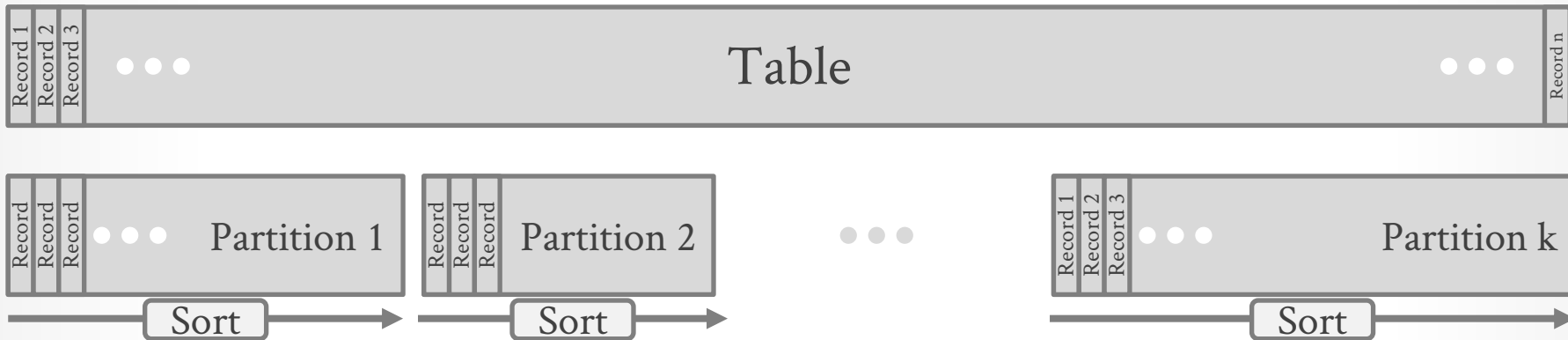
# WINDOW FUNCTIONS

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



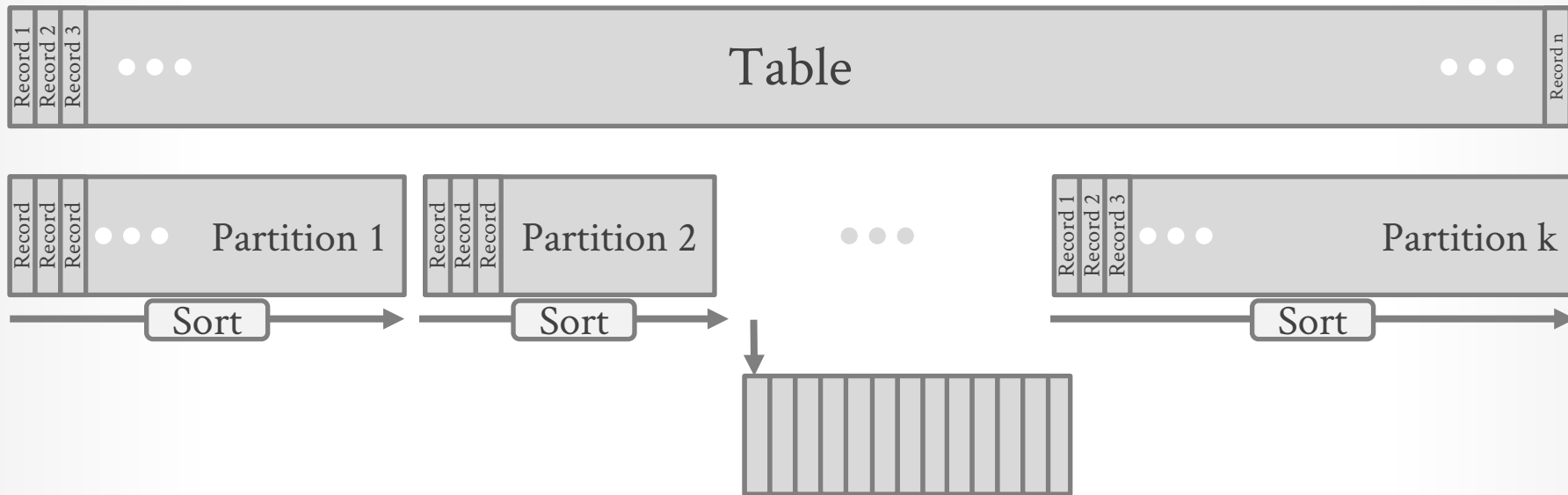
# WINDOW FUNCTIONS

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



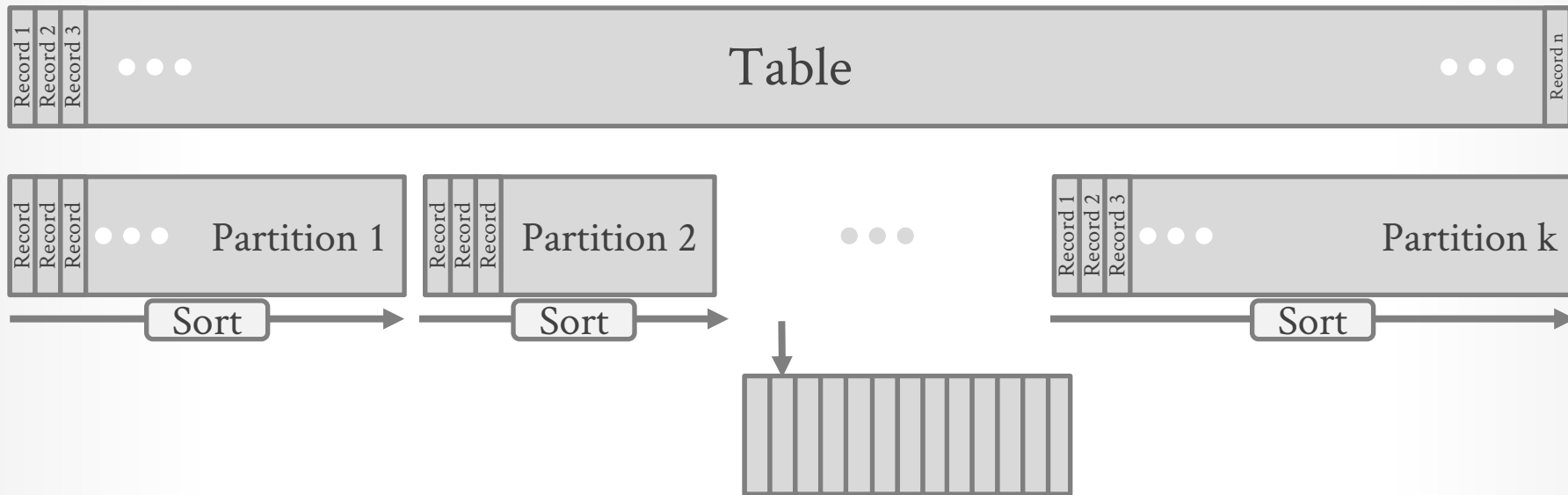
# WINDOW FUNCTIONS

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



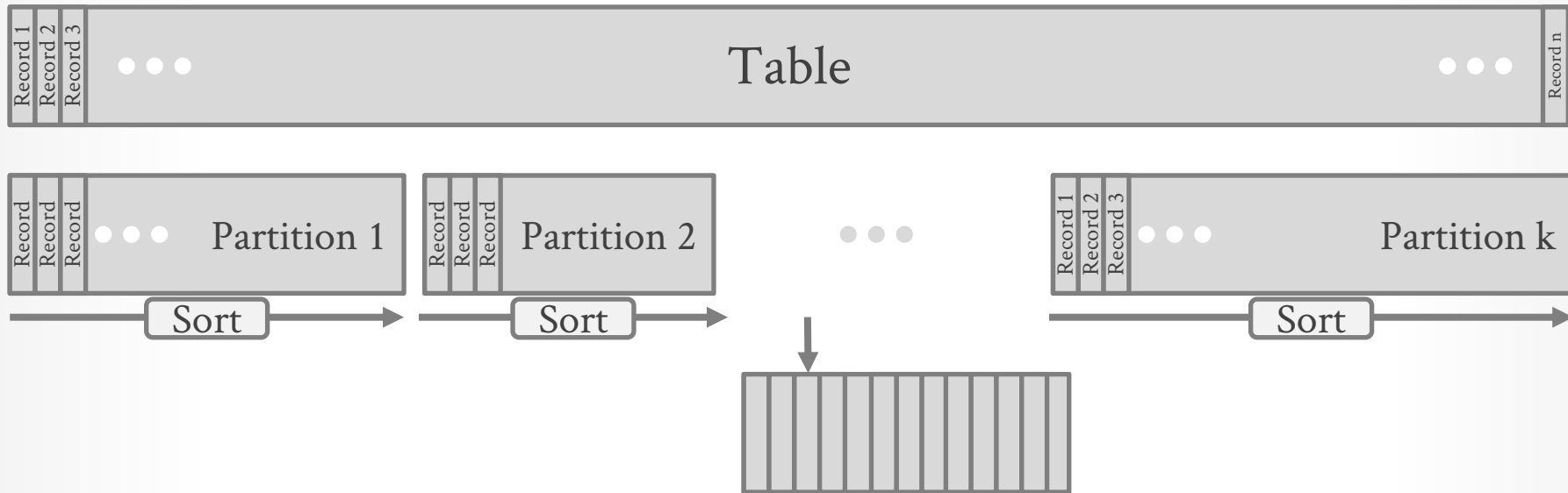
# WINDOW FUNCTIONS

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



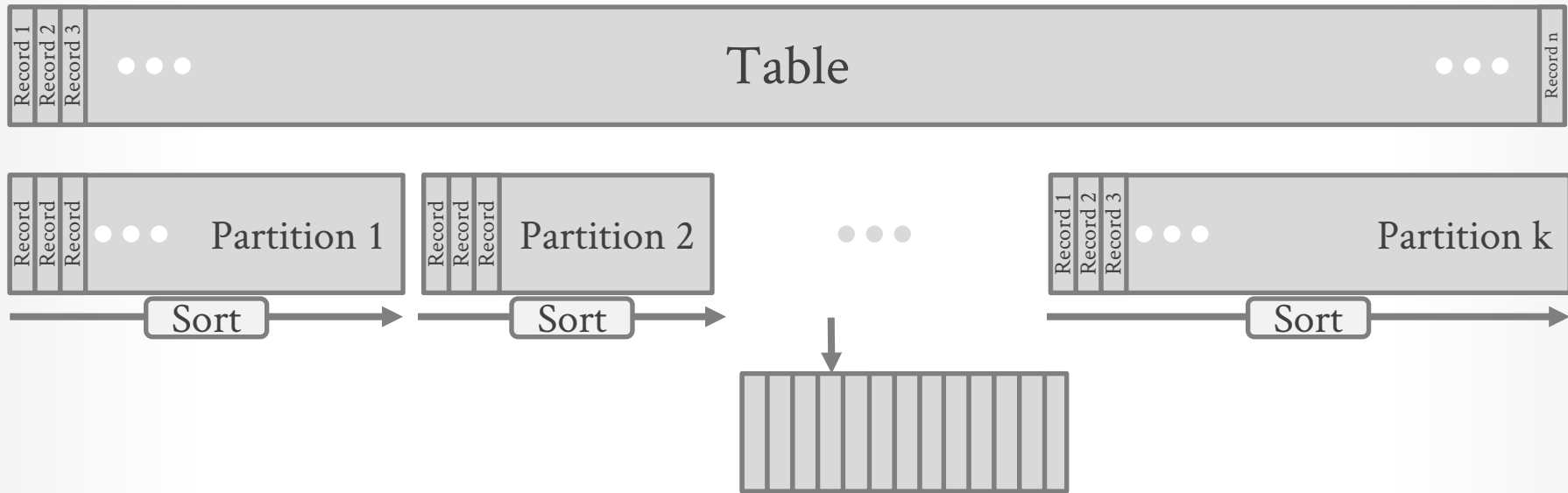
# WINDOW FUNCTIONS

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



# WINDOW FUNCTIONS

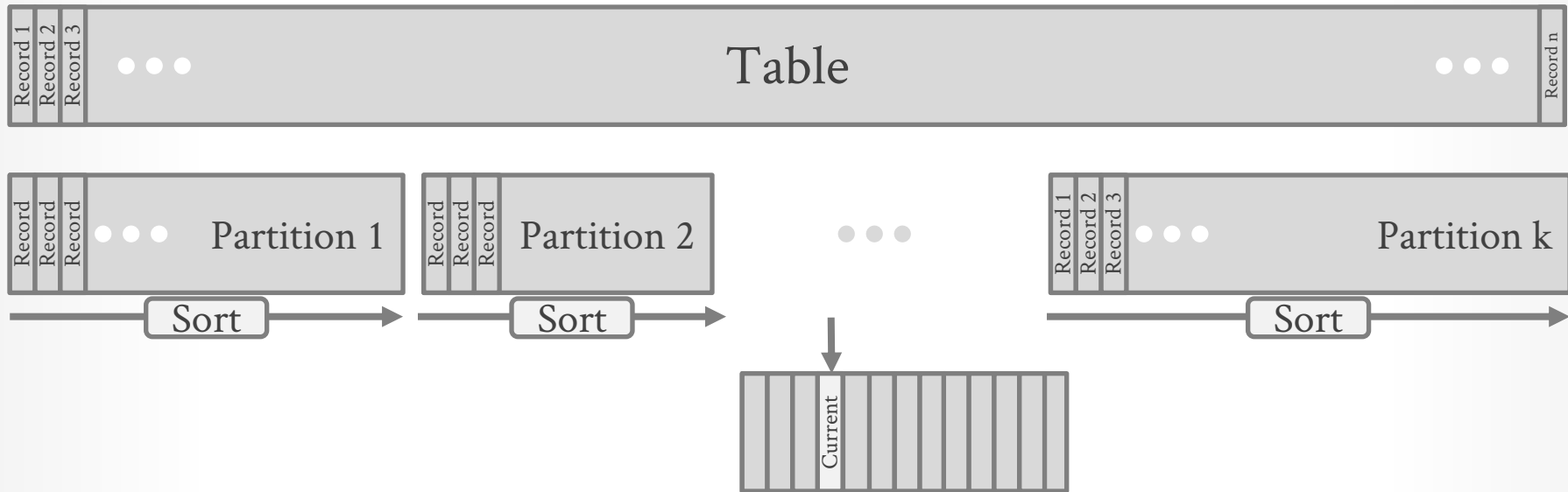
Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.





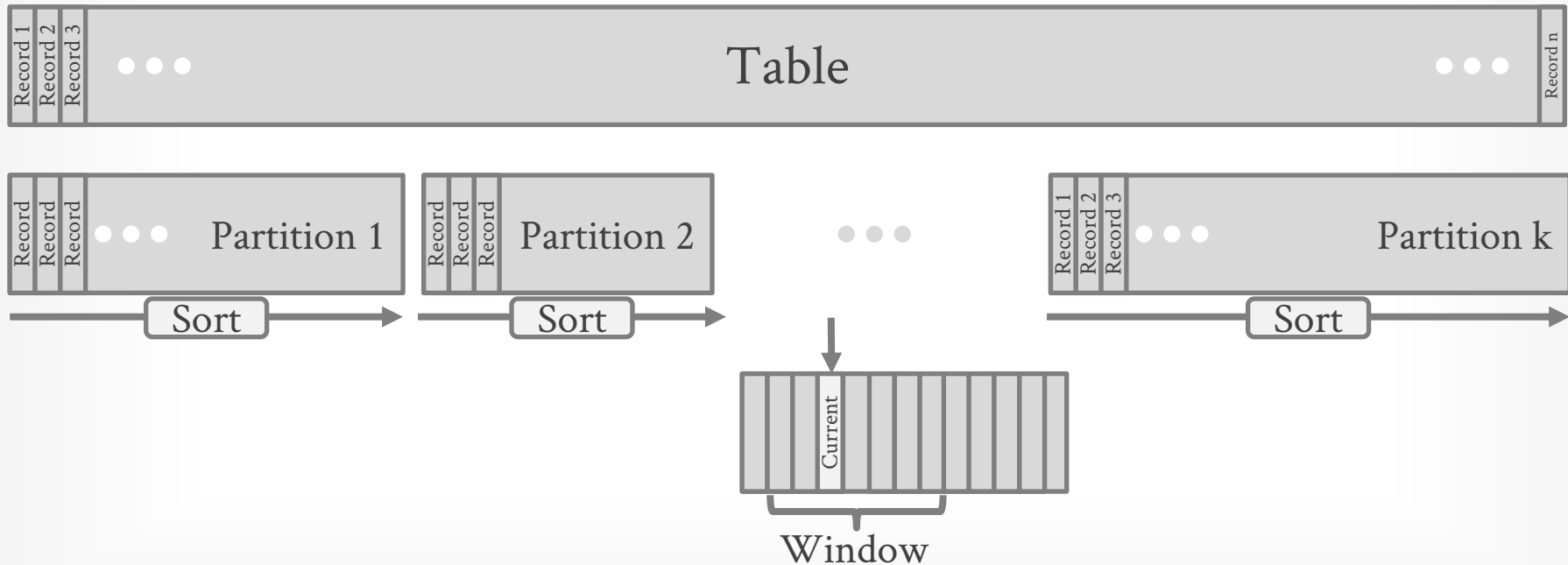
# WINDOW FUNCTIONS

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



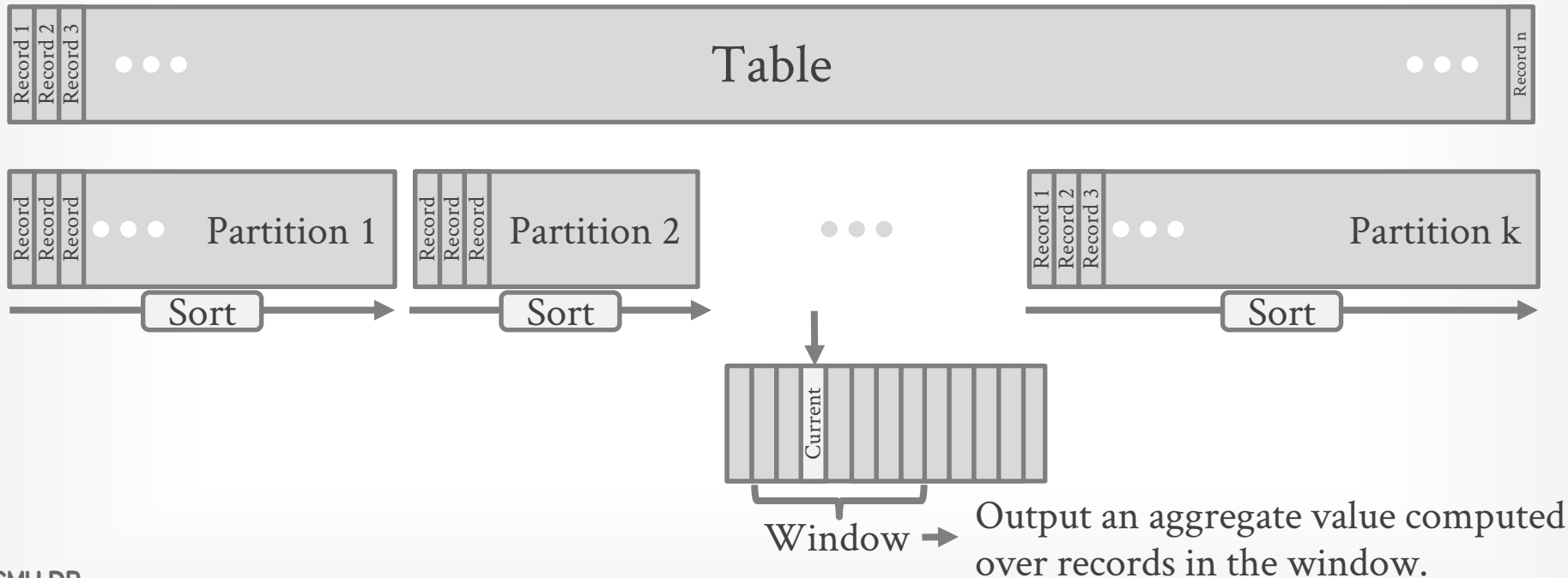
# WINDOW FUNCTIONS

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



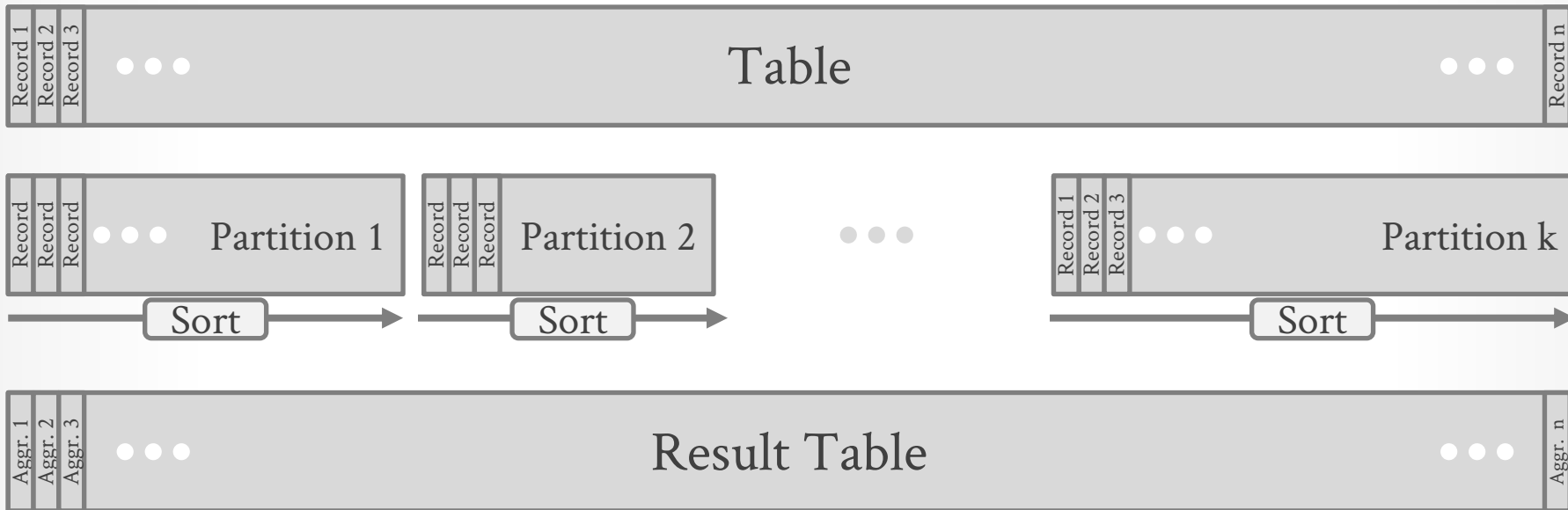
# WINDOW FUNCTIONS

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



# WINDOW FUNCTIONS

Conceptual execution: (optional) Partition data  $\rightarrow$  (optional) sort each partition  
 $\rightarrow$  for each record, creates window  $\rightarrow$  compute an answer for each window.



# WINDOW FUNCTIONS

---

Performs a calculation across a set of tuples that are related to the current tuple, without collapsing them into a single output tuple, to support running totals, ranks, and moving averages.

→ 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 are related to the current tuple, without collapsing them into a single output tuple, to support running totals, ranks, and moving averages.

→ 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 tuples*

*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

---

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 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
```

# WINDOW FUNCTIONS

---

*Find the student 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
```

# WINDOW FUNCTIONS

---

*Find the student 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
```

# NESTED QUERIES

---

Invoke a query inside of another query to compose more complex computations.

→ 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

---

Invoke a query inside of another query to compose more complex computations.

→ Inner queries can appear (almost) anywhere in query.

*Outer Query*



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

*Inner Query*



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

# NESTED QUERIES

---

Invoke a query inside of another query to compose more complex computations.

→ Inner queries can appear (almost) anywhere in query.

*Outer Query*



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

*Inner Query*



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

# 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 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 . . .
```

# 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.*

```
SELECT sid, name FROM student
WHERE sid =
    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 =
    SELECT MAX(sid) FROM enrolled
```

sid	name
53688	Bieber

# NESTED QUERIES

---

*Find all courses that have 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
15-799	Special 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 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
)
```

cid	name
15-799	Special Topics in Databases



# 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	name
15-799	Special Topics in Databases

# LATERAL JOINS

---

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.

```
SELECT * FROM  
  (SELECT 1 AS x) AS t1,  
  LATERAL (SELECT t1.x+1 AS y) AS t2;
```

t1.x	t2.y
1	2

# 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.*

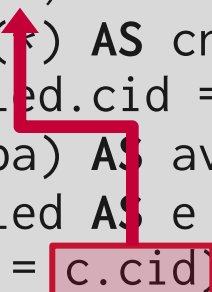
```
SELECT * FROM course AS c,  
  LATERAL (SELECT COUNT(*) AS cnt FROM enrolled  
           WHERE enrolled.cid = c.cid) AS t1,  
  LATERAL (SELECT AVG(gpa) AS avg FROM student AS s  
           JOIN enrolled AS e ON s.sid = e.sid  
           WHERE e.cid = c.cid) AS t2;
```

# 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;
```





# LATERAL JOIN

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

cid	name	cnt	avg
15-445	Database Systems	2	3.75
15-721	Advanced Database Systems	2	3.95
15-826	Data Mining	1	3.9
15-799	Special Topics in Databases	0	null

```

SELECT * FROM course AS c,
  LATERAL (SELECT COUNT(*)
            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;
  
```

# COMMON TABLE EXPRESSIONS

---

Specify a temporary result set that can then be referenced by another part of that query.

→ Think of it like a temp table just for one query.

Alternative to nested queries, views, and explicit temp tables.

```
WITH cteName AS (  
    SELECT 1  
)  
SELECT * FROM cteName
```

# COMMON TABLE EXPRESSIONS

---

Specify a temporary result set that can then be referenced by another part of that query.

→ Think of it like a temp table just for one query.

Alternative to nested queries, views, and explicit temp tables.

```
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
```

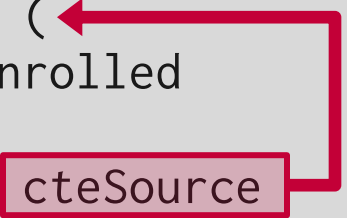
```
WITH cteName (colXXX, colXXX) AS ( Postgres  
    SELECT 1, 2  
)  
SELECT * FROM cteName
```

# 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
```



# OTHER THINGS TO NOTE

Identifiers (e.g. table and column names) are case-insensitive.

→ Makes it harder for applications that care about case (e.g., use CamelCased names).

One often sees quotes around names:

→ **SELECT "ArtistList.firstName"**

You have to pay cash money to get the standard documents.

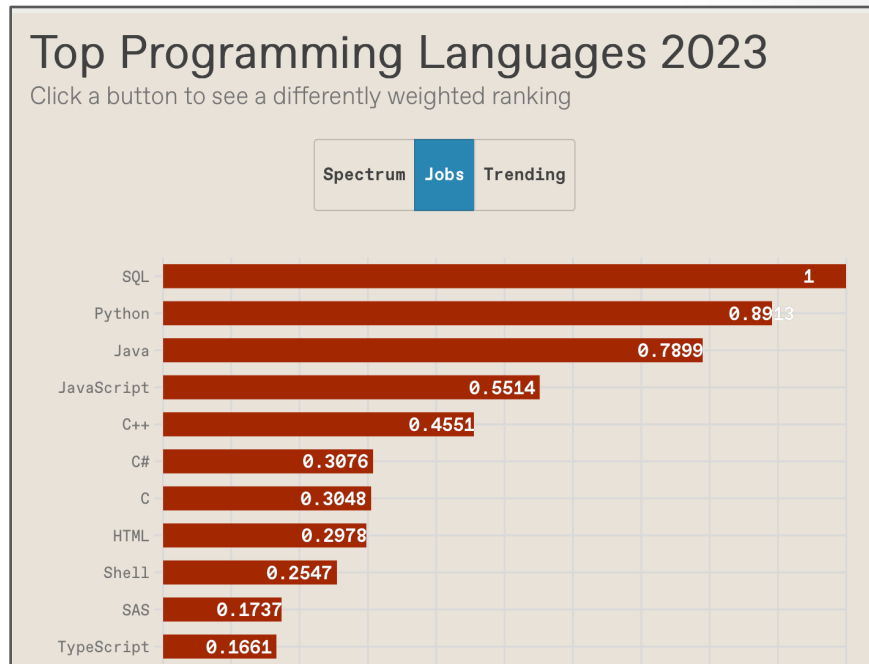
The screenshot shows the ISO website interface for the standard ISO/IEC 9075-2:2023. The page includes a navigation bar with links for Standards, Sectors, About us, News, Taking part, and Store, along with a search bar. The main content area displays the standard's title, 'Information technology Database languages SQL Part 2: Foundation (SQL/Foundation)', and its status as 'Published'. A price tag indicates the cost is CHF 216, with a 'Buy' button and a note to convert Swiss francs (CHF) to the user's currency. A 'General information' section provides details such as the publication date (2023-06), stage (International Standard published [60.60]), edition (6), and number of pages (1715). A 'Read sample' button is also visible, linking to a preview of the standard in the Online Browsing Platform (OBP).

# CONCLUSION

SQL is a hot language.

→ Lots of NL2SQL tools, but writing SQL is not going away.

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: Wednesday Jan. 29<sup>th</sup> @ 11:59pm**

<https://15445.courses.cs.cmu.edu/spring2025/homework1>



# NEXT CLASS

---

We will begin our journey to understanding the internals of database systems starting with Storage!