香港中文大學（深圳）
The Chinese University of Hong Kong, Shenzhen

SCHOOL OF DATA SCIENCE
數據科學學院

Ack: Prof. Jignesh Patel @ CMU
Dr. Chui Chun Kit @ HKU

CSC3170
3: SQL *part b*

Chenhao Ma
School of Data Science
The Chinese University of Hong Kong, Shenzhen

Last Lecture

- Basic usage of SQL
  - Define data
  - Insert, update, and delete
  - Query (SELECT)

This Lecture

More SQL usage (advanced)

- Aggregations + Group By
- String / Date / Time Operations
- Output Control + Redirection
- Window Functions
- Nested Queries
- Joins
- Common Table Expressions

Example Database

student(sid,name,login,gpa)

sid	name	login	age	gpa
53666	RZA	rza@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

Aggregations

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.
- ***Query***: *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.
- ***Query***: *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.
- ***Query***: *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'
```

Aggregates

- Aggregate functions can (almost) only be used in the SELECT output list.
- ***Query****: 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

- ***Query***: *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'
```

Multiple Aggregates

- ***Query****: 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** modifier.
  - Caveat: `COUNT(*)` does not support the `DISTINCT` modifier.

- ***Query****: Get the number of unique students that have an “@cs” login.*

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

Distinct Aggregates

- **COUNT**, **SUM**, **AVG** support **DISTINCT** modifier.
  - Caveat: `COUNT(*)` does not support the `DISTINCT` modifier.
- ***Query***: *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.
- ***Query***: *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
```

Aggregates

- Output of other columns outside of an aggregate is undefined.
- ***Query***: *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.
- ***Query**: Get the average GPA of students enrolled in each course.*

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

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

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
```

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

- 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, 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 + Date/Time Operations

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('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 abbrv_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
 WHERE login = LOWER(name) || '@cs'
```

SQL-92
Postgres
SQLite

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

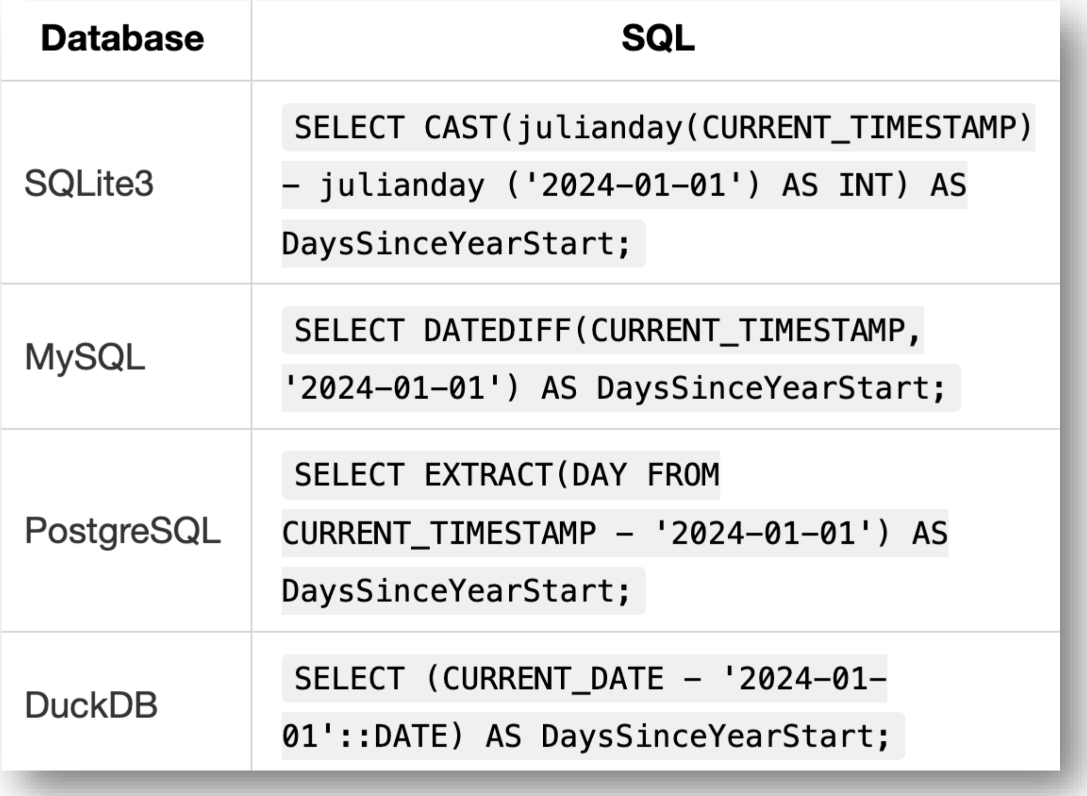
MSSQL

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

MySQL

Date/Time Operations

- Operations to manipulate and modify **DATE/TIME** attributes.
- Can be used in both output and predicates.
- Support/syntax varies wildly…



Database	SQL
SQLite3	`SELECT CAST(julianday(CURRENT_TIMESTAMP) - julianday ('2024-01-01') AS INT) AS DaysSinceYearStart;`
MySQL	`SELECT DATEDIFF(CURRENT_TIMESTAMP, '2024-01-01') AS DaysSinceYearStart;`
PostgreSQL	`SELECT EXTRACT(DAY FROM CURRENT_TIMESTAMP - '2024-01-01') AS DaysSinceYearStart;`
DuckDB	`SELECT (CURRENT_DATE - '2024-01-01'::DATE) AS DaysSinceYearStart;`

Output Control

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

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 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);
```

1. Throw an error on the first violation. All previously inserted tuples are removed.
2. Throw an error but keep any previously inserted tuples.
3. Ignore error and keep going.

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
```

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
 WH
 OR
```

```
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
 WH
 OR
```

```
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
 WH
 OR
```

```
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”

Postgres

```
SELECT sid, name FROM student
 WHERE login LIKE '%@cs'
FETCH FIRST 10 ROWS ONLY;
```

Postgres

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”

Postgres

```
SELECT sid, name FROM student
 WHERE login LIKE '%@cs'
FETCH FIRST 10 ROWS ONLY;
```

Postgres

```
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"

Postgres

```
SELECT sid, name FROM student
 WHERE login LIKE '%@cs'
FETCH FIRST 10 ROWS ONLY;
```

Postgres

```
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          Postgres
 WHERE login LIKE '%@cs'
FETCH FIRST 10 ROWS ONLY;
```

```
SELECT sid, name FROM student          Postgres
 WHERE login LIKE '%@cs'
 ORDER BY gpa
OFFSET 10 ROWS
FETCH FIRST 10 ROWS WITH TIES;
```

The `WITH TIES` clause ensures that if multiple rows have the same gpa value as the 10th row, all of them will be included in the result set.

Window Functions

Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.

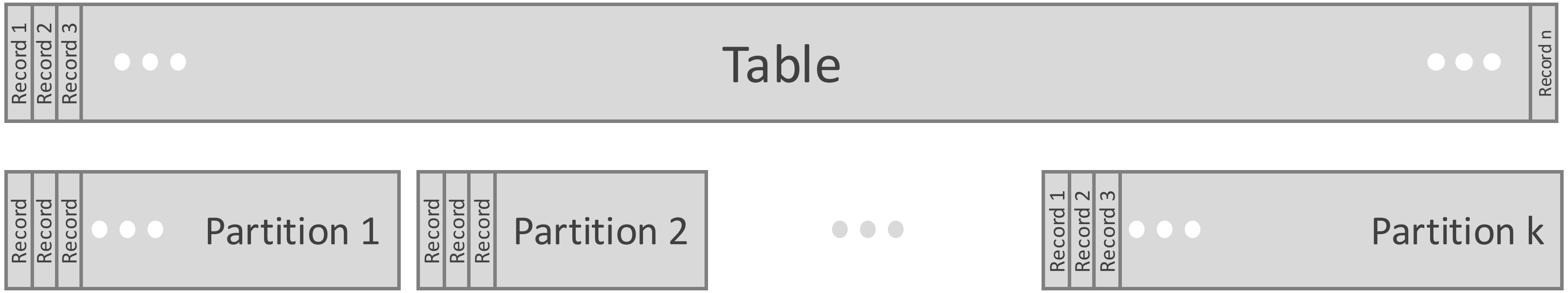
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.

Record 1
Record 2
Record 3
Table
Record n

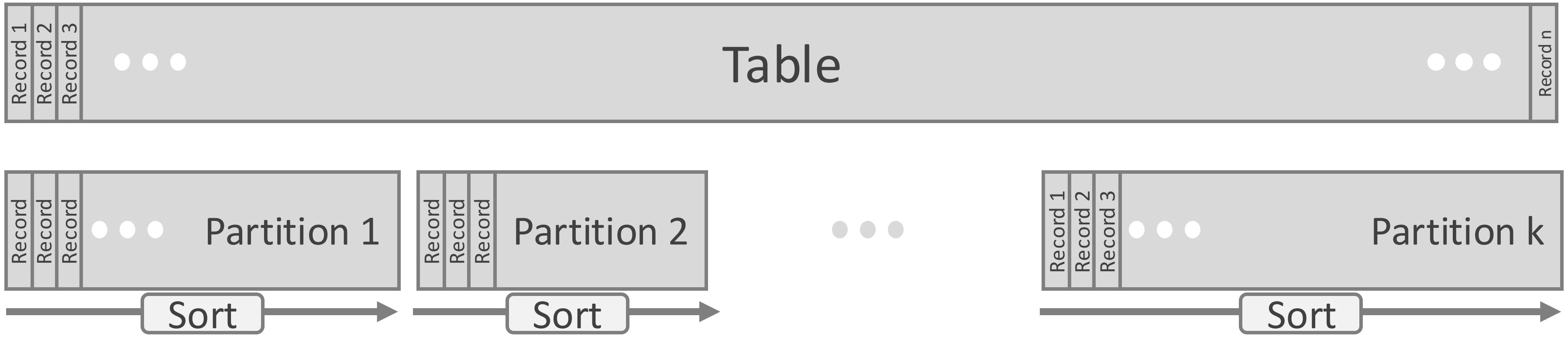
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



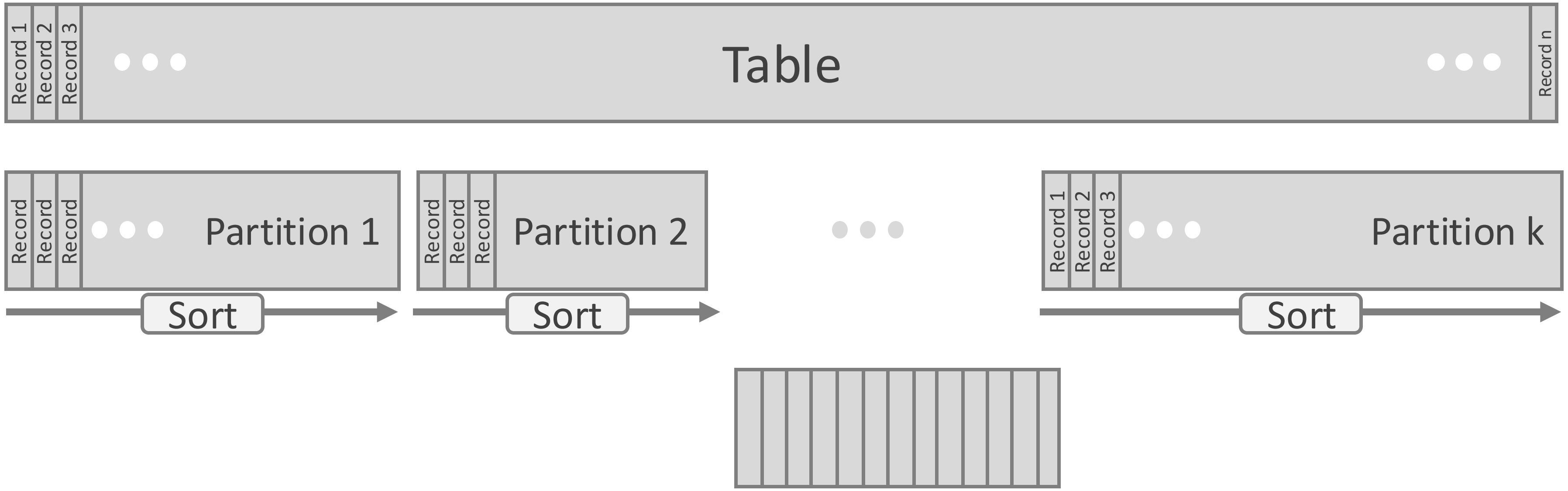
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



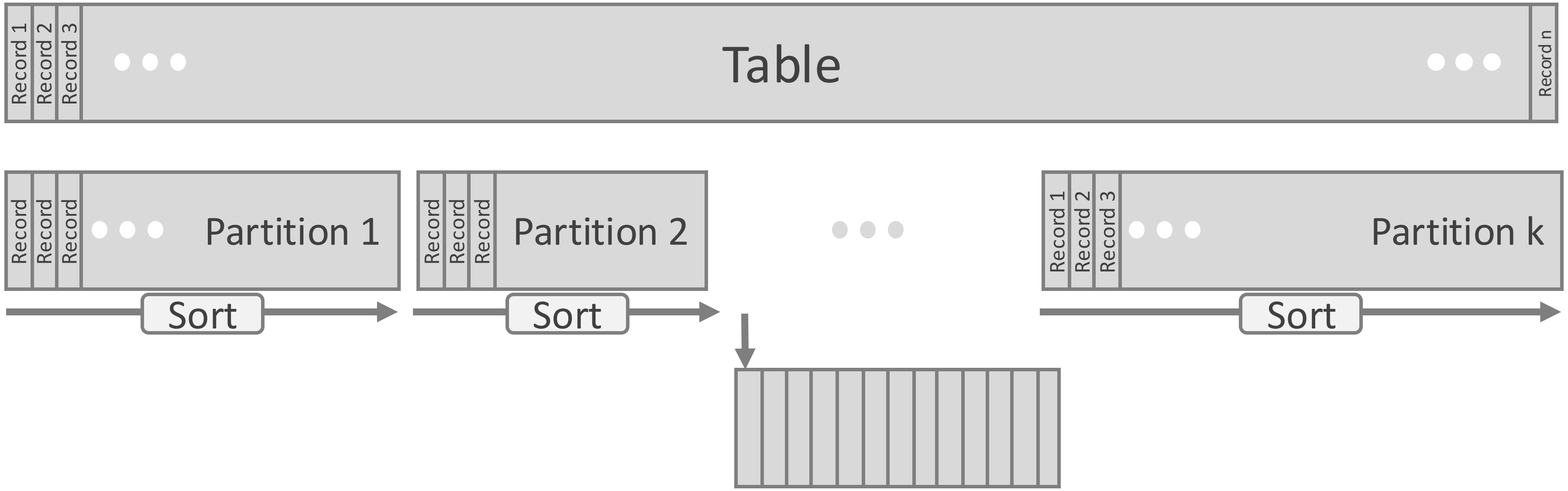
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



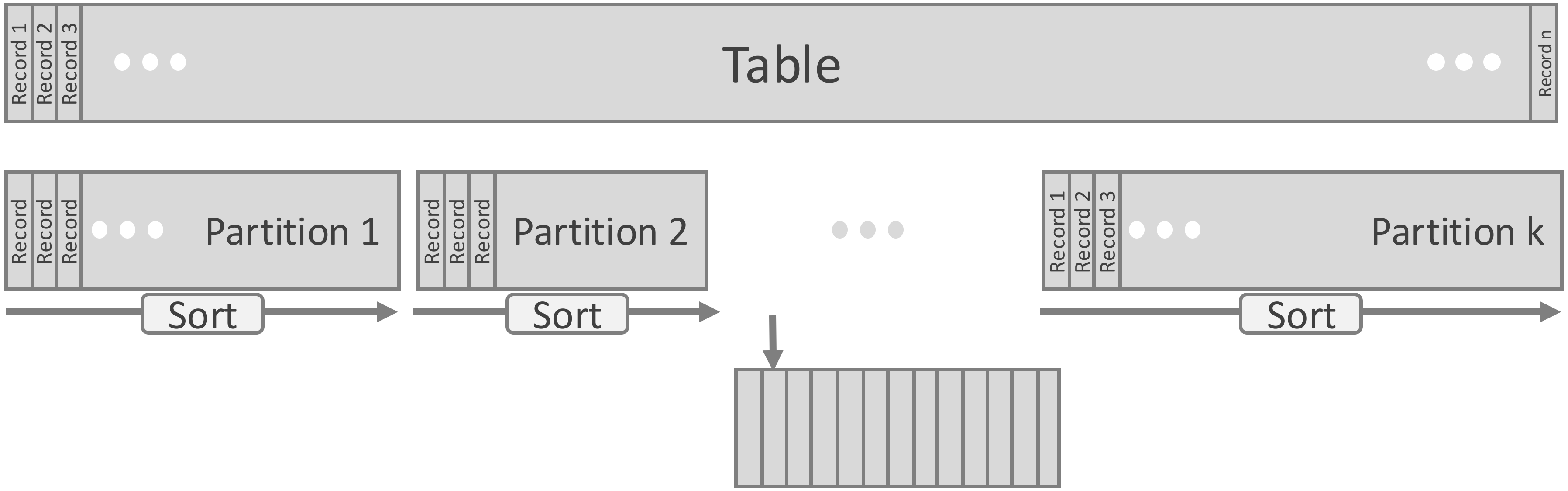
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



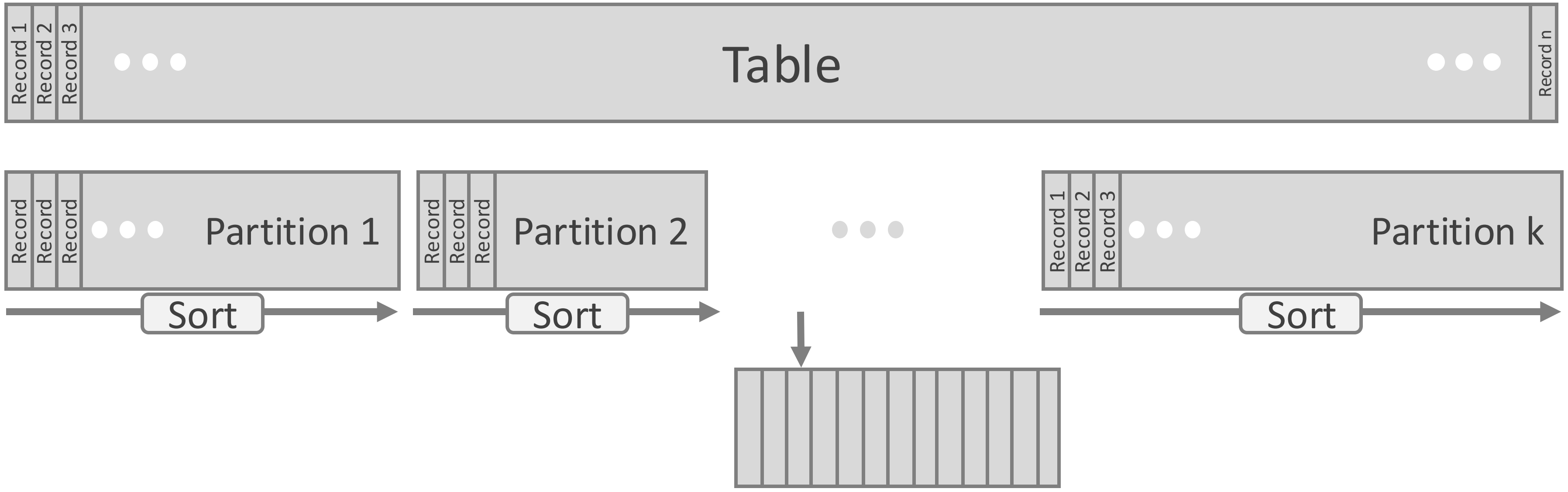
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



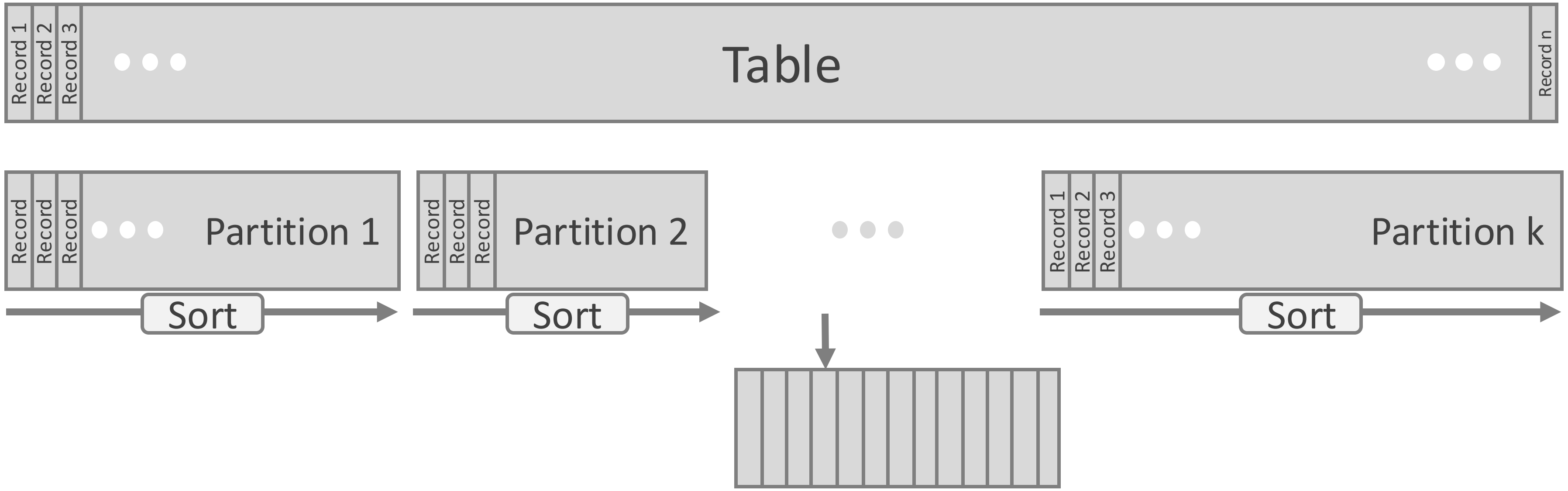
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



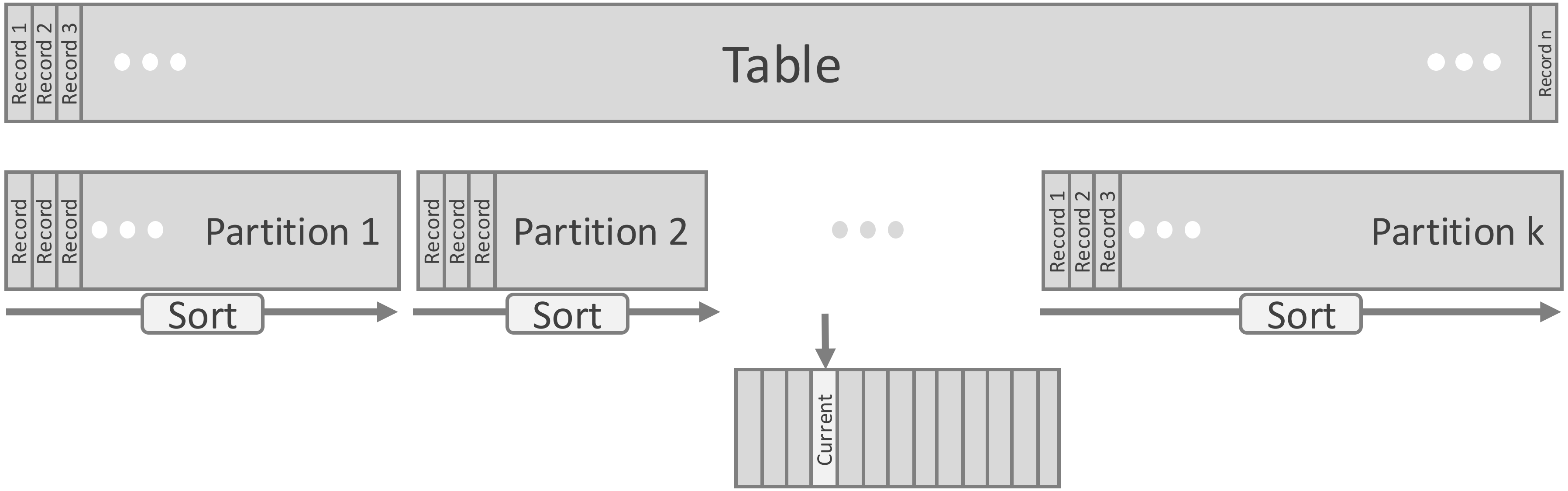
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



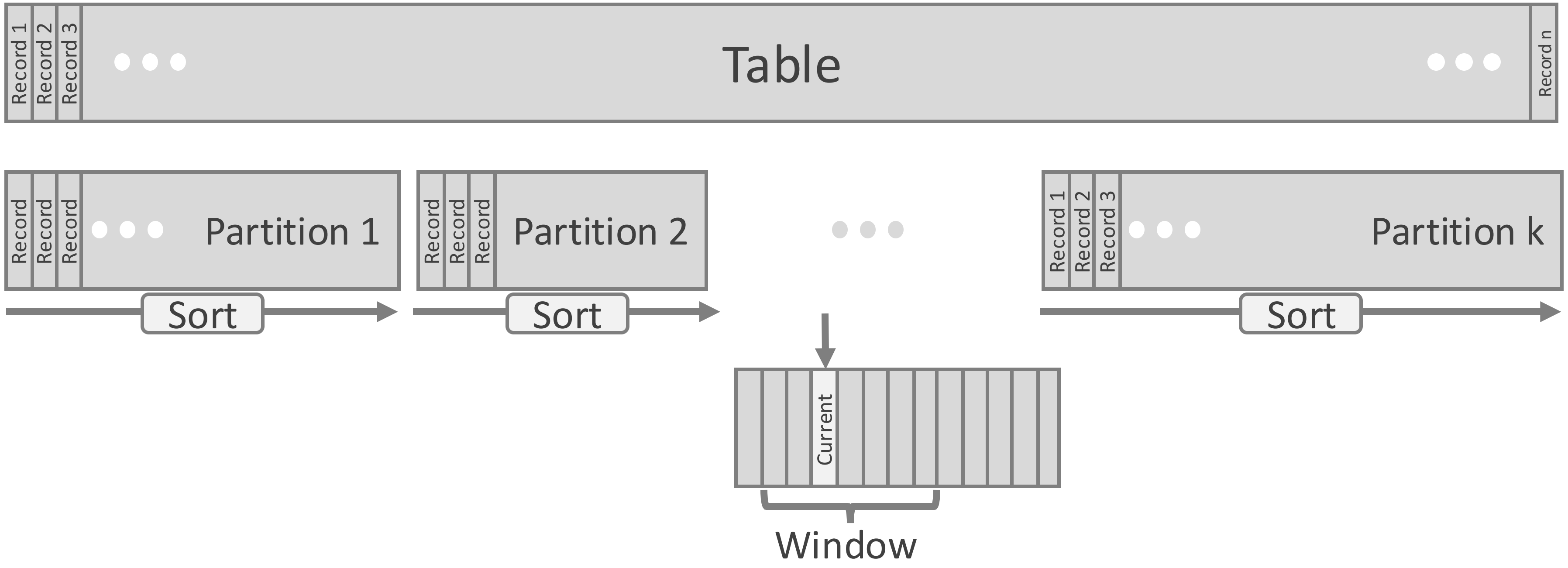
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



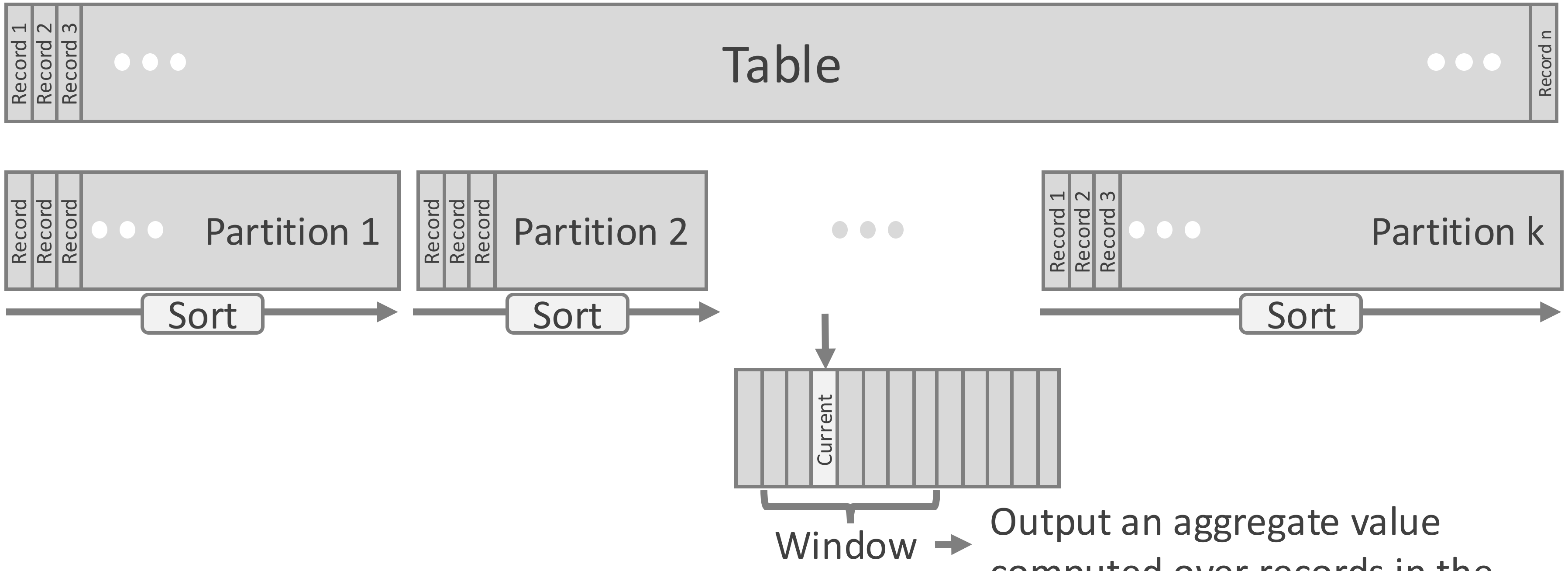
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



Window Functions

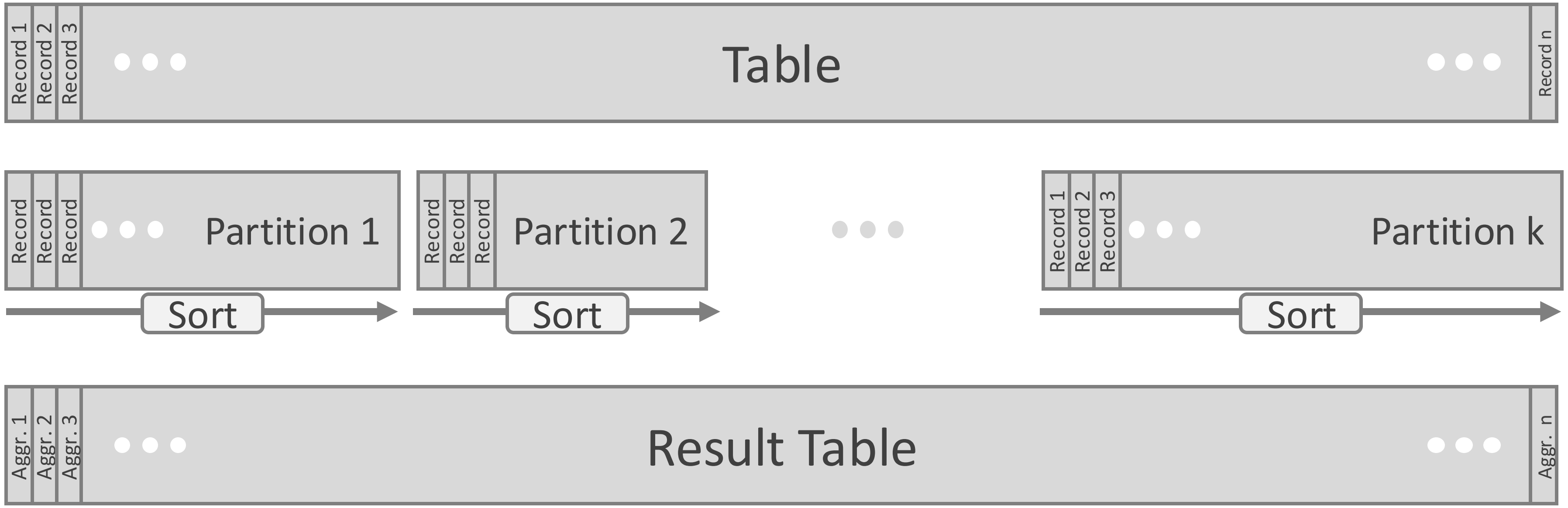
- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



Output an aggregate value computed over records in the window.

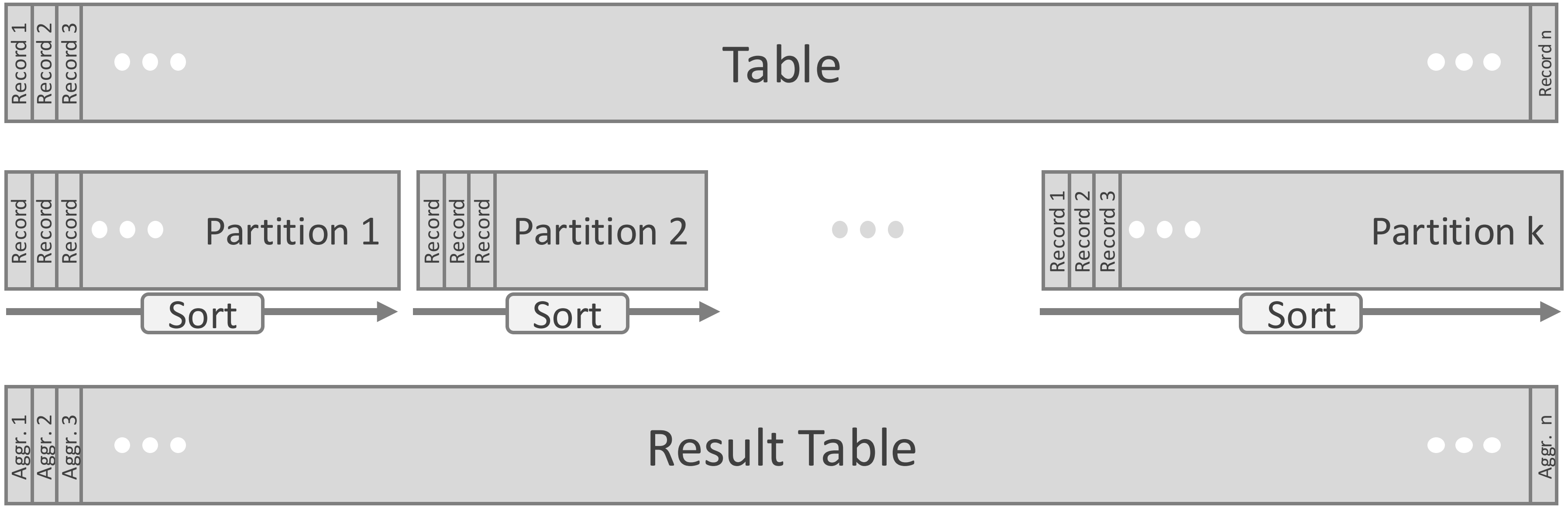
Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



Window Functions

- Conceptual execution: Partition data → sort each partition → for each record create a window → compute an answer for each window.



Window Functions

- Aggregation functions:
  - Anything that we discussed earlier
  - Will be re-visited a few slides later

- 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
  - Will be re-visited a few slides later

- 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

- ***Query***: *Find the student with the <u>second</u> 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

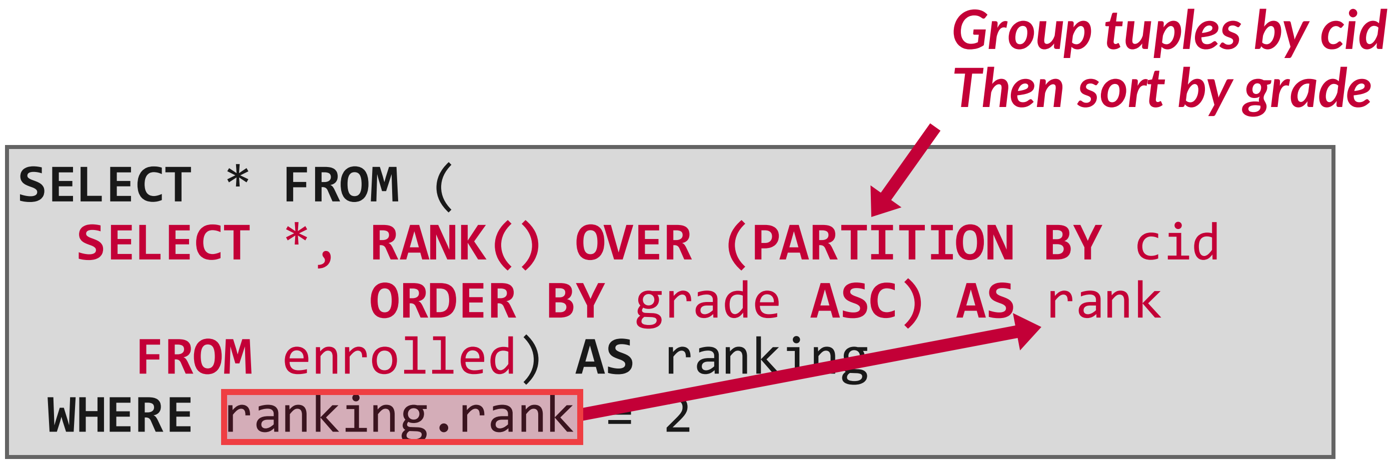
- ***Query**: Find the student with the <u>second</u> highest grade for each course.*

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

- ***Query***: *Find the student with the second highest grade for each course.*



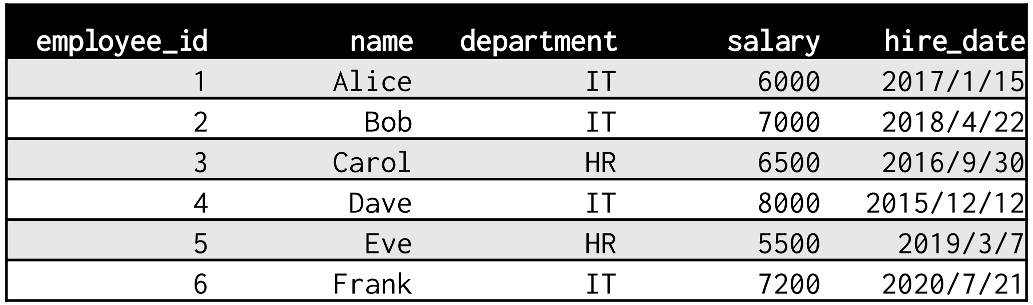
Group tuples by cid
Then sort by grade

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

Window Functions

Cumulative Aggregates

- When you use **aggregate functions** like `SUM()`, `AVG()`, `COUNT()`, etc., with the `OVER()` clause and an `ORDER BY`, you can get **cumulative results**.



employee_id	name	department	salary	hire_date
1	Alice	IT	6000	2017/1/15
2	Bob	IT	7000	2018/4/22
3	Carol	HR	6500	2016/9/30
4	Dave	IT	8000	2015/12/12
5	Eve	HR	5500	2019/3/7
6	Frank	IT	7200	2020/7/21

Window Functions

Cumulative Aggregates

- When you use **aggregate functions** like SUM(), AVG(), COUNT(), etc., with the OVER() clause and an ORDER BY, you can get **cumulative results**.

```
SELECT employee_id, name, salary,
SUM(salary) OVER (ORDER BY salary) AS
cumulative_salary
FROM employees;
```

employee_id	name	department	salary	hire_date
1	Alice	IT	6000	2017/1/15
2	Bob	IT	7000	2018/4/22
3	Carol	HR	6500	2016/9/30
4	Dave	IT	8000	2015/12/12
5	Eve	HR	5500	2019/3/7
6	Frank	IT	7200	2020/7/21

Window Functions

Cumulative Aggregates

- When you use **aggregate functions** like SUM(), AVG(), COUNT(), etc., with the OVER() clause and an ORDER BY, you can get **cumulative results**.

```
SELECT employee_id, name, salary,
SUM(salary) OVER (ORDER BY salary) AS
cumulative_salary
FROM employees;
```

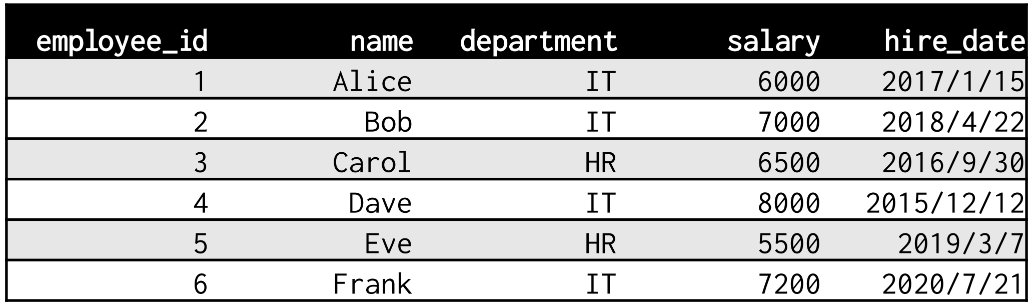
employee_id	name	department	salary	hire_date
1	Alice	IT	6000	2017/1/15
2	Bob	IT	7000	2018/4/22
3	Carol	HR	6500	2016/9/30
4	Dave	IT	8000	2015/12/12
5	Eve	HR	5500	2019/3/7
6	Frank	IT	7200	2020/7/21

employee_id	name	salary	cumulative_salary
5	Eve	5500	5500
1	Alice	6000	11500
3	Carol	6500	18000
2	Bob	7000	25000
6	Frank	7200	32200
4	Dave	8000	40200

Window Functions

Non-Cumulative Aggregates

- When you remove the `ORDER BY` clause from the `OVER()` function, the aggregate function is applied to **the entire window** (i.e., partition) without calculating cumulative values.
  - You just get the **same aggregate value** repeated for each row in the window.

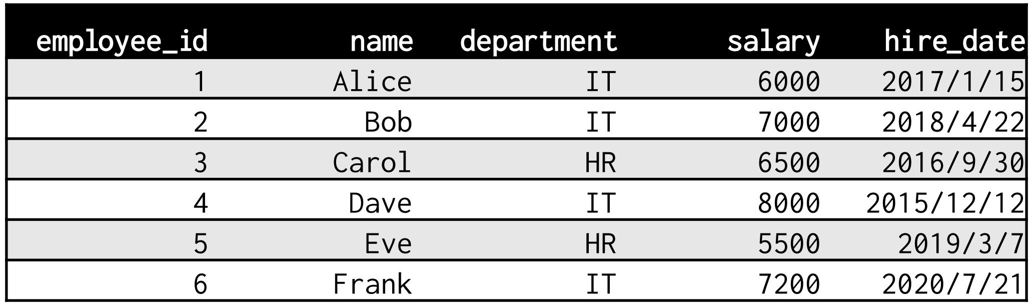


employee_id	name	department	salary	hire_date
1	Alice	IT	6000	2017/1/15
2	Bob	IT	7000	2018/4/22
3	Carol	HR	6500	2016/9/30
4	Dave	IT	8000	2015/12/12
5	Eve	HR	5500	2019/3/7
6	Frank	IT	7200	2020/7/21

Window Functions

Non-Cumulative Aggregates

- When you remove the ORDER BY clause from the OVER() function, the aggregate function is applied to **the entire window** (i.e., partition) without calculating cumulative values.
  - You just get the **same aggregate value** repeated for each row in the window.



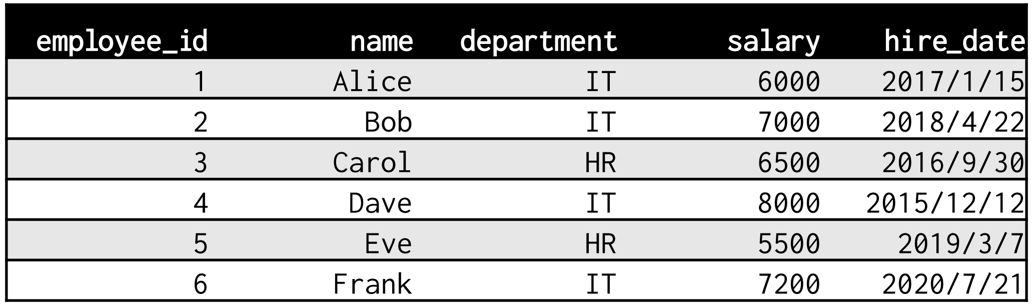
employee_id	name	department	salary	hire_date
1	Alice	IT	6000	2017/1/15
2	Bob	IT	7000	2018/4/22
3	Carol	HR	6500	2016/9/30
4	Dave	IT	8000	2015/12/12
5	Eve	HR	5500	2019/3/7
6	Frank	IT	7200	2020/7/21

```
SELECT employee_id, name, salary,
SUM(salary) OVER () AS total_salary
FROM employees;
```

Window Functions

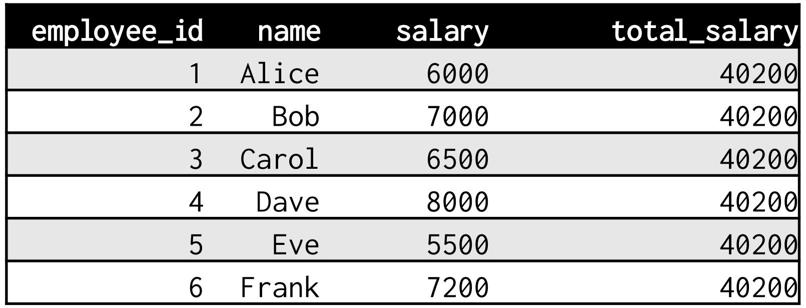
Non-Cumulative Aggregates

- When you remove the `ORDER BY` clause from the `OVER()` function, the aggregate function is applied to **the entire window** (i.e., partition) without calculating cumulative values.
  - You just get the **same aggregate value** repeated for each row in the window.



employee_id	name	department	salary	hire_date
1	Alice	IT	6000	2017/1/15
2	Bob	IT	7000	2018/4/22
3	Carol	HR	6500	2016/9/30
4	Dave	IT	8000	2015/12/12
5	Eve	HR	5500	2019/3/7
6	Frank	IT	7200	2020/7/21

```sql
SELECT employee_id, name, salary,
SUM(salary) OVER () AS total_salary
FROM employees;
```

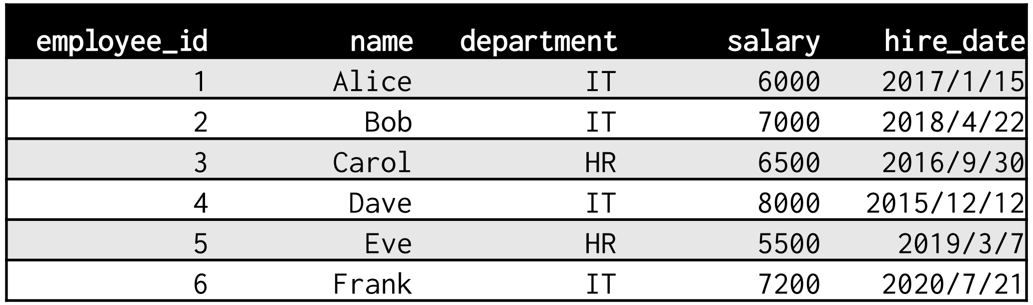


employee_id	name	salary	total_salary
1	Alice	6000	40200
2	Bob	7000	40200
3	Carol	6500	40200
4	Dave	8000	40200
5	Eve	5500	40200
6	Frank	7200	40200

Window Functions

Partitioned Aggregates

- You can also use `PARTITION BY` in the `OVER()` clause to compute aggregates over subsets (partitions) of the data.
    - Each partition will have its own aggregate value.

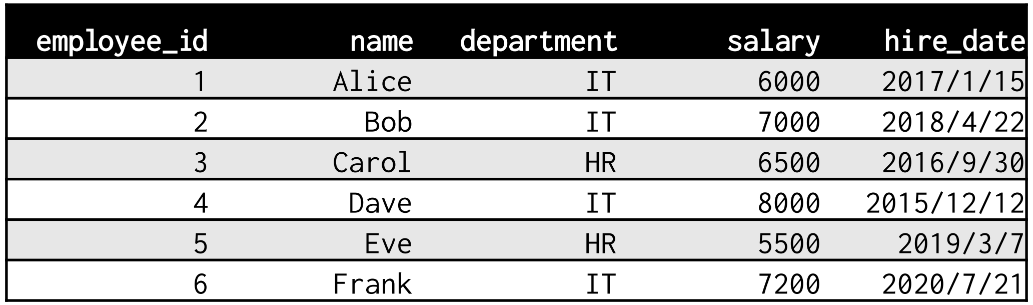


employee_id	name	department	salary	hire_date
1	Alice	IT	6000	2017/1/15
2	Bob	IT	7000	2018/4/22
3	Carol	HR	6500	2016/9/30
4	Dave	IT	8000	2015/12/12
5	Eve	HR	5500	2019/3/7
6	Frank	IT	7200	2020/7/21

Window Functions

Partitioned Aggregates

- You can also use PARTITION BY in the OVER() clause to compute aggregates over subsets (partitions) of the data.
  - Each partition will have its own aggregate value.



employee_id	name	department	salary	hire_date
1	Alice	IT	6000	2017/1/15
2	Bob	IT	7000	2018/4/22
3	Carol	HR	6500	2016/9/30
4	Dave	IT	8000	2015/12/12
5	Eve	HR	5500	2019/3/7
6	Frank	IT	7200	2020/7/21

```sql
SELECT employee_id, name, department, salary,
SUM(salary) OVER (PARTITION BY department)
AS department_salary
FROM employees;
```

Window Functions

Partitioned Aggregates

- You can also use `PARTITION BY` in the `OVER()` clause to compute aggregates over subsets (partitions) of the data.
  - Each partition will have its own aggregate value.

employee_id	name	department	salary	hire_date
1	Alice	IT	6000	2017/1/15
2	Bob	IT	7000	2018/4/22
3	Carol	HR	6500	2016/9/30
4	Dave	IT	8000	2015/12/12
5	Eve	HR	5500	2019/3/7
6	Frank	IT	7200	2020/7/21

```
SELECT employee_id, name, department, salary,
SUM(salary) OVER (PARTITION BY department)
AS department_salary
FROM employees;
```

employee_id	name	department	salary	department_salary
4	Dave	IT	8000	28200
1	Alice	IT	6000	28200
2	Bob	IT	7000	28200
6	Frank	IT	7200	28200
3	Carol	HR	6500	12000
5	Eve	HR	5500	12000

Window Functions

More general

3-day moving avg

Window Functions

More general

3-day moving avg

`sales(sid, sales_date, daily_sales)`

sid	sales_date	daily_sales
1	2024/9/1	100
2	2024/9/2	200
3	2024/9/3	150
4	2024/9/4	300
5	2024/9/5	250

Window Functions

More general

3-day moving avg

```
SELECT
    sales_date,
    daily_sales,
    AVG(daily_sales) OVER (
        ORDER BY sales_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg
FROM sales;
```

sales(sid, sales_date, daily_sales)

sid	sales_date	daily_sales
1	2024/9/1	100
2	2024/9/2	200
3	2024/9/3	150
4	2024/9/4	300
5	2024/9/5	250

Window Functions

More general

3-day moving avg

```
SELECT
    sales_date,
    daily_sales,
    AVG(daily_sales) OVER (
        ORDER BY sales_date
        ROWS BETWEEN 2 PRECEDING AND CURRENT ROW
    ) AS moving_avg
FROM sales;
```

sales(<u>sid</u>, sales_date, daily_sales)

sid	sales_date	daily_sales
1	2024/9/1	100
2	2024/9/2	200
3	2024/9/3	150
4	2024/9/4	300
5	2024/9/5	250

result

sales_date	daily_sales	moving_avg
2024/9/1	100	100
2024/9/2	200	150
2024/9/3	150	150
2024/9/4	300	216.67
2024/9/5	250	233.33

Window Functions - Summary

Case	Example SQL	Meaning / Effect
No PARTITION BY, no ORDER BY	`SELECT AVG(salary) OVER() FROM employees;`	Treats the **whole table** as one group; computes a single window aggregate for all rows (e.g., global average).
With PARTITION BY, no ORDER BY	`SELECT dept, AVG(salary) OVER(PARTITION BY dept) FROM employees;`	Groups rows by dept; computes aggregate within each partition; **no row ordering** inside partitions.
With ORDER BY, no PARTITION BY	`SELECT name, salary, SUM(salary) OVER(ORDER BY salary) FROM employees;`	One partition (whole table), but rows are ordered; allows cumulative, ranking, and frame-sensitive functions.
With both PARTITION BY and ORDER BY	`SELECT dept, name, salary, RANK() OVER(PARTITION BY dept ORDER BY salary DESC) FROM employees;`	Each partition is separately ordered; ranking/cumulative metrics are applied **per partition**.
With ROWS BETWEEN (frame clauses)	`SELECT name, salary, SUM(salary) OVER(ORDER BY hire_date ROWS BETWEEN 1 PRECEDING AND CURRENT ROW) FROM employees;`	Restricts the “window frame” within the ordered partition (e.g., rolling sum/average). Defaults are UNBOUNDED PRECEDING to CURRENT ROW if not specified.

Nested Queries

Nested Queries

- Invoke a query inside of another query to compose more complex computations.
  - They are often difficult to optimize for the DBMS due to correlations.
  - Inner queries can appear (almost) anywhere in query.

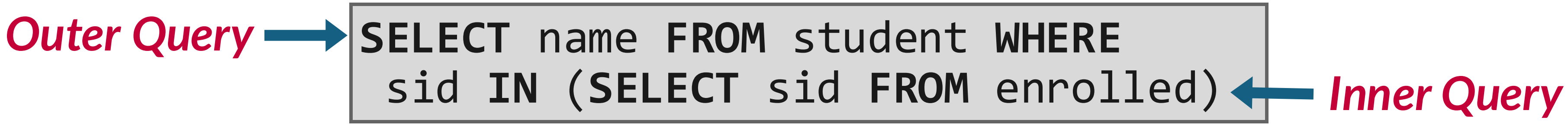
Nested Queries

- Invoke a query inside of another query to compose more complex computations.
  - They are often difficult to optimize for the DBMS due to correlations.
  - Inner queries can appear (almost) anywhere in query.

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

Nested Queries

- Invoke a query inside of another query to compose more complex computations.
  - They are often difficult to optimize for the DBMS due to correlations.
  - Inner queries can appear (almost) anywhere in query.



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

Nested Queries

- Invoke a query inside of another query to compose more complex computations.
  - They are often difficult to optimize for the DBMS due to correlations.
  - 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

Query: *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

Query: *Get the names of students in ‘15-445’*

```
SELECT name FROM student
 WHERE ...
   SELECT sid FROM enrolled
    WHERE cid = '15-445'
```

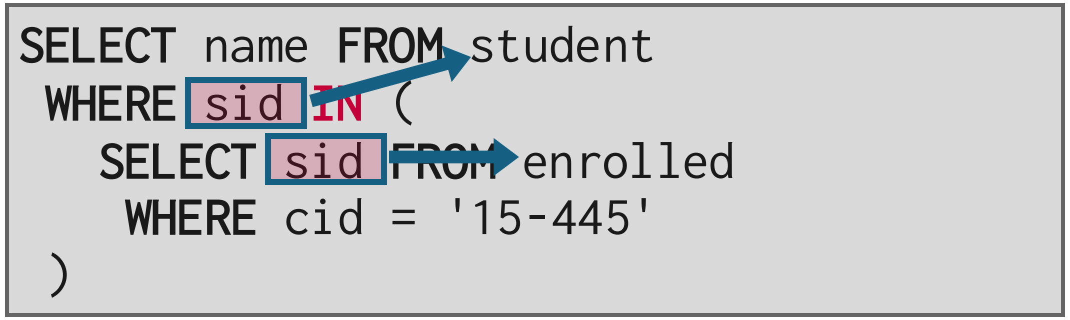
Nested Queries

Query: *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

Query: *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 the outer query.

Nested Queries

Query: *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

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

Nested Queries

- ***Query****: 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;
```

Nested Queries

- ***Query***: *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;
```

Nested Queries

- ***Query***: *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

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

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

Nested Queries

- ***Query****: 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

- ***Query***: *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

- ***Query***: *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

- ***Query:*** *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

- ***Query:*** *Find all courses that have no students enrolled in it.*

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

Nested Queries

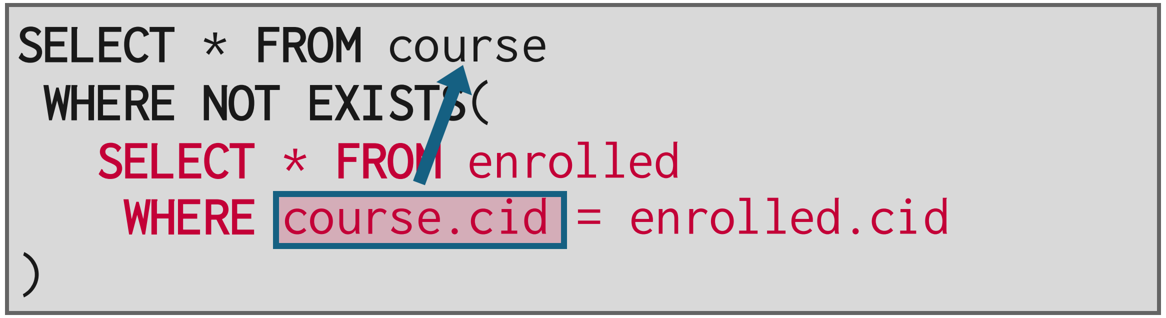
- ***Query:*** *Find all courses that have no students enrolled in it.*

```
SELECT * FROM course
 WHERE NOT EXISTS(
   SELECT * FROM enrolled
    WHERE course.cid = enrolled.cid
)
```

cid	name
15-799	Special Topics in Databases

Nested Queries

- ***Query:*** *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

Join

Inner Join, Outer Join, Natural Join, Lateral Join

Inner Join

- The **INNER JOIN** (often just called a **JOIN**) returns rows when there is a match in both tables being joined.
- If a row in one table does not have a matching row in the other table, that row will **not** be included in the result.

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	NULL

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

```
SELECT *
FROM Employee E JOIN Department D
ON E.department_id = D.department_id;
```

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME

Outer Join

- An **outer join** does not require each record in the two joined tables to have a matching record.

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	NULL

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

```
SELECT *
FROM Employee E LEFT OUTER JOIN Department D
ON E.department_id = D.department_id;
```

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME
David	NULL	NULL	NULL

Outer Join

- An **outer join** does not require each record in the two joined tables to have a matching record.

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	NULL

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

```
SELECT *
FROM Employee E LEFT OUTER JOIN Department D
ON E.department_id = D.department_id;
```

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME
David	NULL	NULL	NULL

Outer Join

- An **outer join** does not require each record in the two joined tables to have a matching record.

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	NULL

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

Even if the **LEFT table record does not have matching records in the RIGHT table**, we still output the tuple in the LEFT table (with null values for the columns of the RIGHT table).

```
SELECT *
FROM Employee E LEFT OUTER JOIN Department D
ON E.department_id = D.department_id;
```

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME
David	NULL	NULL	NULL

Outer Join

- An **outer join** does not require each record in the two joined tables to have a matching record.

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	NULL

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

```
SELECT *
FROM Employee E RIGHT OUTER JOIN Department D
ON E.department_id = D.department_id;
```

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME
NULL	NULL	35	EEE

Outer Join

- An **outer join** does not require each record in the two joined tables to have a matching record.

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	NULL

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME
NULL	NULL	35	EEE

```
SELECT *
FROM Employee E RIGHT OUTER JOIN Department D
ON E.department_id = D.department_id;
```

Outer Join

- An **outer join** does not require each record in the two joined tables to have a matching record.

e_name	department_id
Kit	31
Ben	33
John	33
Jolly	34
Yvonne	34
David	NULL

department_id	d_name
31	CS
33	Civil
34	ME
35	EEE

Even if the **RIGHT table record does not have matching records in the LEFT table**, we still output the tuple in the RIGHT table (with null values for the columns of the LEFT table).

e_name	E.department_id	D.department_id	d_name
Kit	31	31	CS
Ben	33	33	Civil
John	33	33	Civil
Jolly	34	34	ME
Yvonne	34	34	ME
NULL	NULL	35	EEE

```
SELECT *
FROM Employee E RIGHT OUTER JOIN Department D
ON E.department_id = D.department_id;
```

Lateral Join

- A `LATERAL JOIN` allows you to reference columns from the preceding tables in the `FROM` clause, especially in subqueries.
  - **Without `LATERAL`**, each subquery is evaluated independently and so **cannot cross-reference any other `FROM` item.**
  - It's useful when you want to apply a subquery that depends on the current row of the outer query.

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

t1.x	t2.y
1	2

More info:
https://stackoverflow.com/questions/28550679/what-is-the-difference-between-a-lateral-join-and-a-subquery-in-postgresql

Lateral Join

- A `LATERAL` join is more like a correlated subquery, not a plain subquery, in that ***expressions to the right*** of a `LATERAL` join are evaluated once ***for each row left of it*** - just like a *correlated* subquery
  - While a plain subquery (table expression) is evaluated *once* only.

Syntax

```
SELECT column_list
FROM table1
JOIN LATERAL (subquery) AS alias
ON condition;
```

Effectively, all of these do the same:

```
JOIN LATERAL ... ON true
```

```
, LATERAL ...
```

```
CROSS JOIN LATERAL ...
```

Lateral Join

- ***Query:*** *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

- ***Query:** 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
ORDER BY t1.cnt DESC;;
```

Lateral Join

- ***Query:*** *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
ORDER BY t1.cnt DESC;;
```

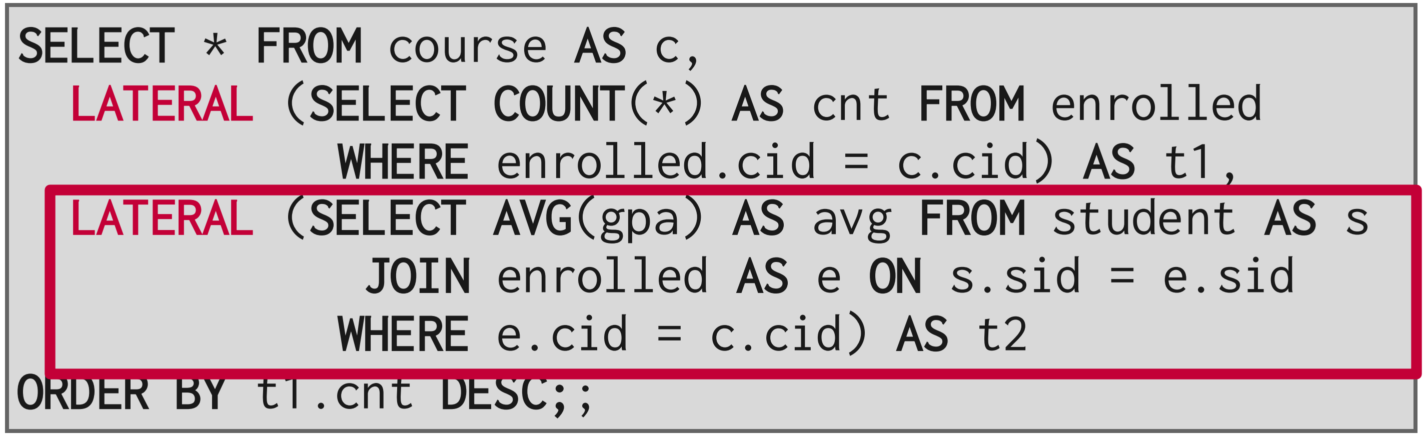
Lateral Join

- ***Query:*** *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
ORDER BY t1.cnt DESC;;
```

Lateral Join

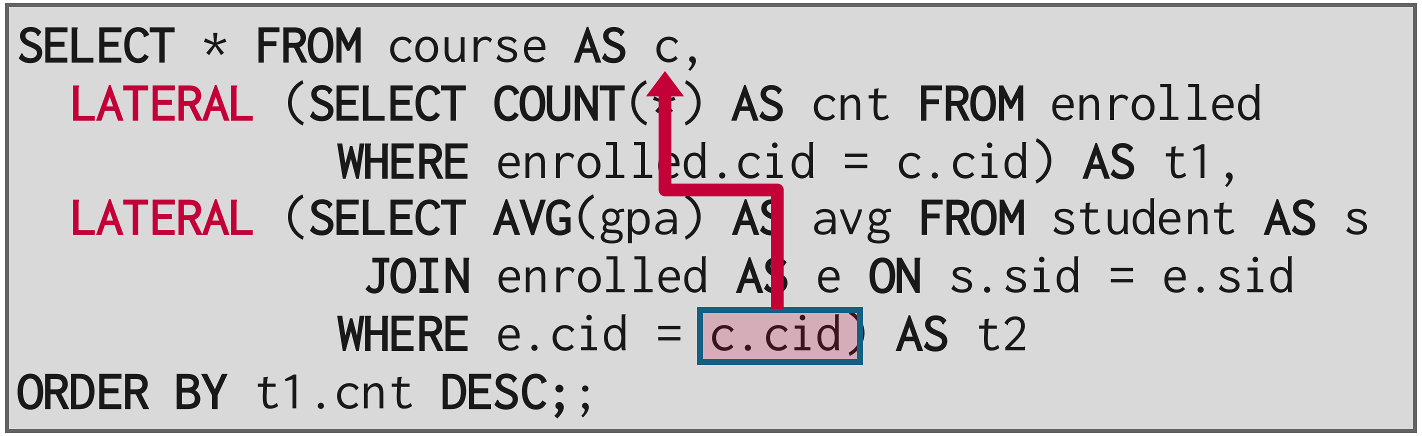
- ***Query:*** *Calculate the number of students enrolled in each course and the average GPA. Sort by enrollment count in descending order.*



```
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
ORDER BY t1.cnt DESC;;
```

Lateral Join

- ***Query:*** *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
ORDER BY t1.cnt DESC;;
```

Lateral Join

- ***Query:*** *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(*) A
             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
ORDER BY t1.cnt DESC;;
```

Lateral Join

- ***Query:*** *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(*) A
           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
ORDER BY t1.cnt DESC;;
```

You can think of it like a `for` loop that allows you to invoke another query for each tuple in a table.

Common Table Expressions

Common Table Expressions

- Provides a way to write auxiliary statements for use in a larger query.
  - A table variable with the lifespan for just that query.
- Alternative to nested queries and views.
  - Makes long queries modular

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

Common Table Expressions

- Provides a way to write auxiliary statements for use in a larger query.
    - A table variable with the lifespan for just that query.
- Alternative to nested queries and views.
    - Makes long queries modular

```
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
```

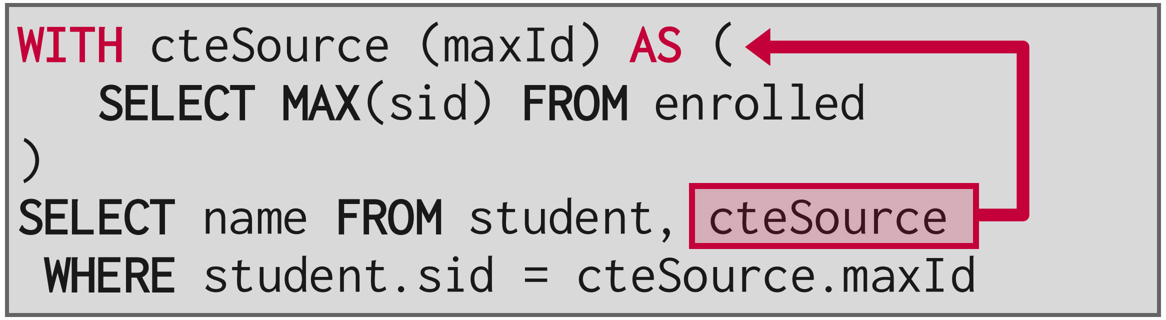
Common Table Expressions

- ***Query:*** *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
```

Common Table Expressions

- ***Query:*** *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 Topics In SQL

- Views
- Triggers
- More on foreign key constraints: `ON DELETE CASCADE`
- Data Control Language (DCL)
- Transaction Control Language (TCL)

Other Notes About SQL

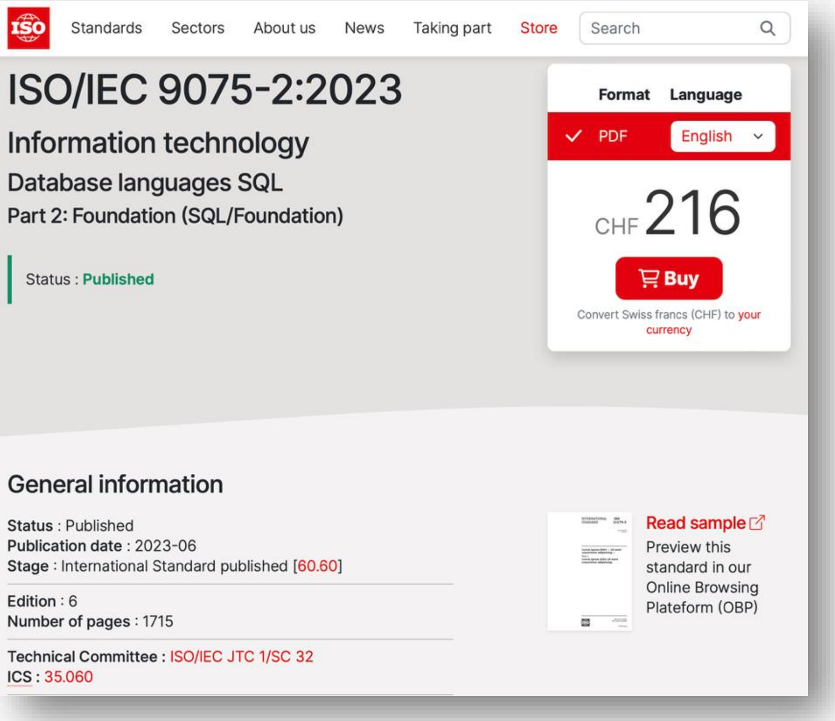
- Identifiers (e.g. table and column names) are case-insensitivity. Makes it harder for applications that care about case (e.g. use CamelCased names).
  - One often sees quotes around names, e.g. `SELECT “ArtistList.firstName”.` Ugly!

- The standard itself is behind a paywall ☹

Other Notes About SQL

- Identifiers (e.g. table and column names) are case-insensitivity. Makes it harder for applications that care about case (e.g. use CamelCased names).
  - One often sees quotes around names, e.g. `SELECT “ArtistList.firstName”.` Ugly!

- The standard itself is behind a paywall ☹



ISO/IEC 9075-2:2023

Information technology

Database languages SQL

Part 2: Foundation (SQL/Foundation)

Status : Published

Format: PDF
Language: English

CHF 216

Convert Swiss francs (CHF) to your currency

General information

Status : Published
Publication date : 2023-06
Stage : International Standard published [60.60]

Edition : 6
Number of pages : 1715

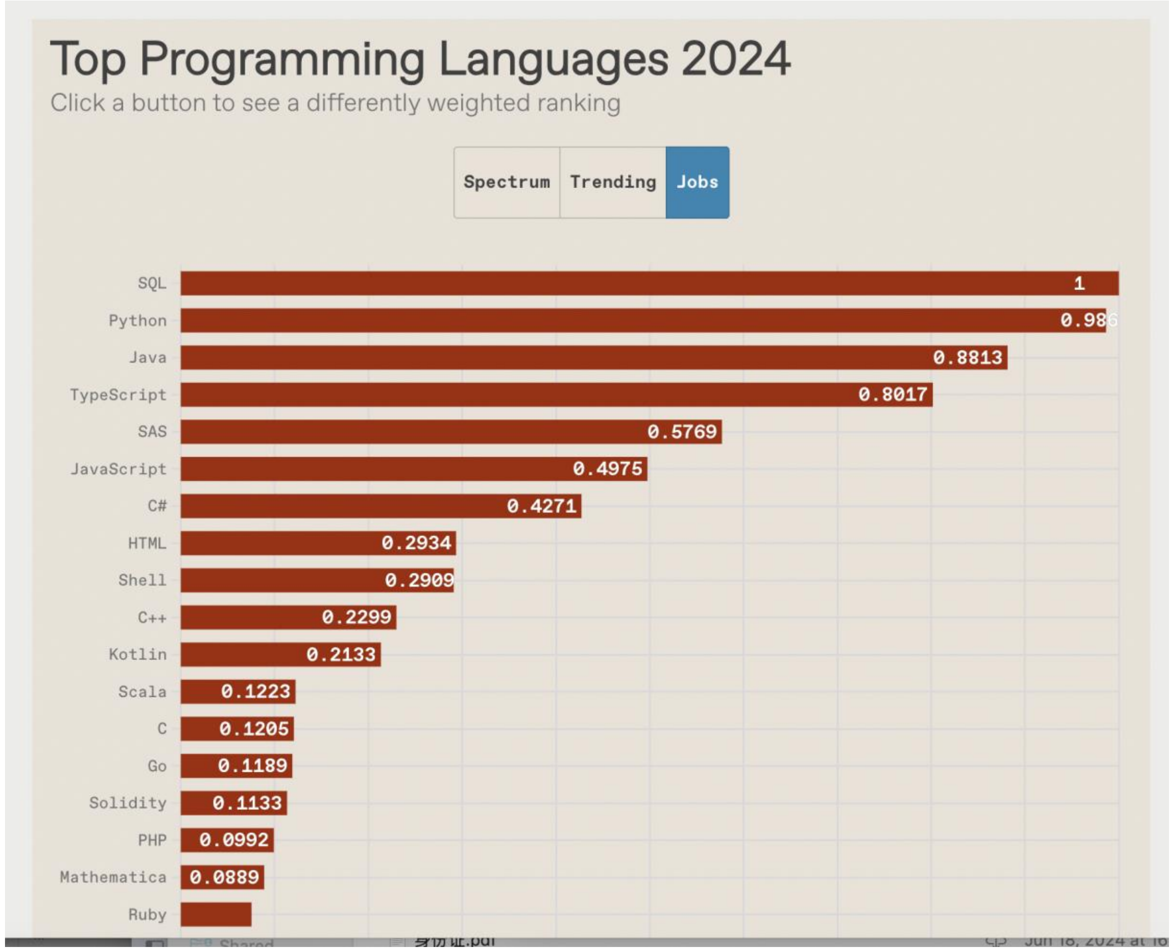
Technical Committee : ISO/IEC JTC 1/SC 32
ICS : 35.060

Read sample
Preview this standard in our Online Browsing Plateform (OBP)

Conclusion

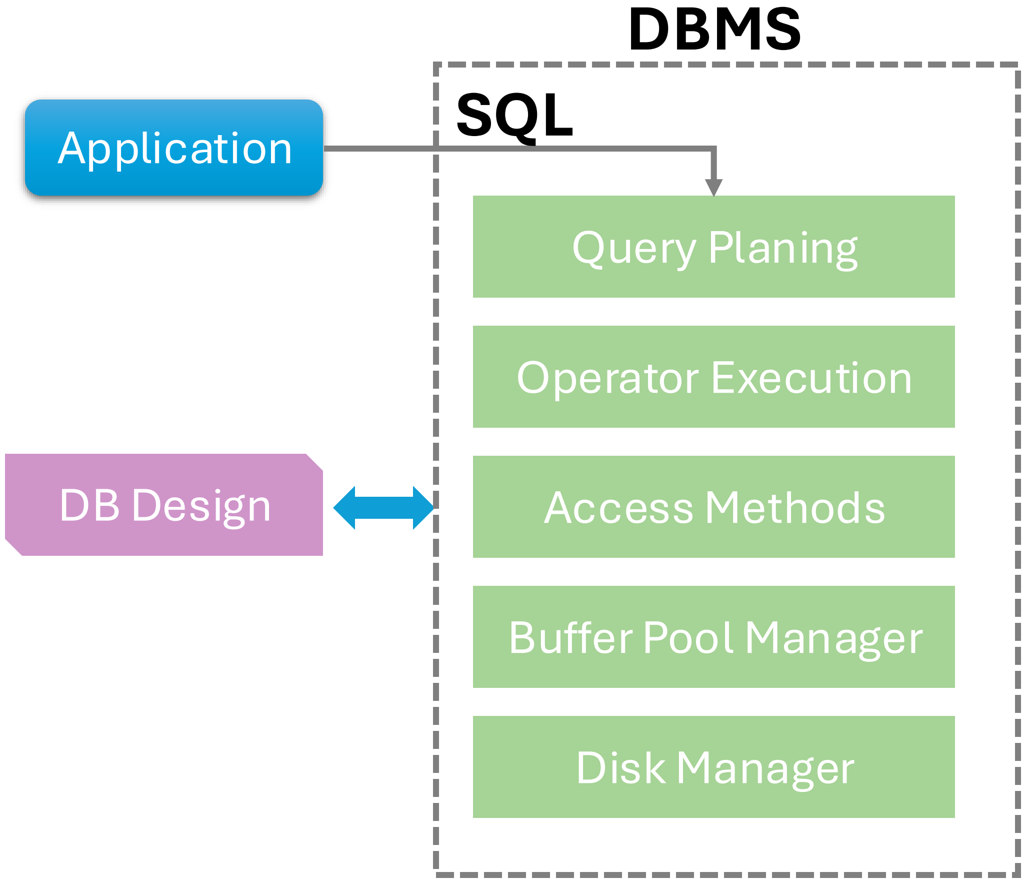
- SQL is “hot” language.
  - Lots of NL2SQL tools, but writing SQL is not going away, but these tools can complement writing SQL.

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



Next Lecture

- Storage


DBMS
SQL
Application
Query Planing
Operator Execution
DB Design
Access Methods
Buffer Pool Manager
Disk Manager