Writing Queries involving more than one Relation

CSC365
Spring 2019

Products and Joins in SQL

- To couple relations in one query
    - List relations separated by comma in FROM clause
    - SELECT * FROM Movies, Genre;

Disambiguating Attributes

- SELECT mid, title, year, genre FROM Movies, Genre WHERE mid=mid;
  - The column name mid is ambiguous.
- To disambiguate, we can prepend table name to column names.

```
SELECT Movies.mid, title, year, genre

FROM Movies, Genre

WHERE Movies.mid=Genre.mid;
```

Tuple Variables (Renaming Relations ρ)

- Sometimes, you want to alias the name of a relation to a short string

```
SELECT m.mid, title, year, genre

FROM Movies as m, Genre as g

WHERE m.mid=g.mid;
```

Eliminating Duplicates

DISTINCT

```
SELECT DISTINCT m.mid, title, year, genre
 FROM Movies as m, Genre as g
WHERE m.mid=g.mid;
```

Cartesian Product

R x S

There are two ways to express cartesian product in SQL.

Implicitly:

```
SELECT *
FROM Movies, Genre;
```

Explicitly:

```
SELECT *
FROM Movies CROSS JOIN Genre;
```

Theta Joins

We can use the following simple form to express theta joins:

Relational algebra expression:

$U \bowtie_{\text{A < V.C AND U.B != V.B}} V$

```
SELECT *
FROM U, V
WHERE A < V.C AND U.B <> V.B
```

Theta Joins

```
SELECT *

FROM Movies m, Genre g

WHERE m.mid = g.mid AND g.genre = 'Sci-Fi';
```

Equi Joins

The SQL standard defines JOIN … USING as a shortcut for natural equijoins. This allows explicit control over the join columns (vs. NATURAL JOIN)

```
SELECT *
FROM Movies INNER JOIN Genre USING (mid);
```

This syntax requires columns with the same name in joined tables.

Natural Joins

There are multiple ways to express a natural join in SQL.

Explicit projection and equijoin on common column(s):

```
SELECT DISTINCT Movies.*, Genre.genre
FROM Movies, Genre
WHERE Movies.mid = Genre.mid;
```

Implicitly, using NATURAL JOIN:

```
SELECT DISTINCT *
FROM Movies NATURAL JOIN Genre;
```

Theta Joins with JOIN

As an alternative, we can use the following to express theta joins. Note the lack of a WHERE clause. Instead, we use the ANSI ***infix*** operator, [INNER] JOIN … ON, to specify our join condition. The INNER keyword is *optional*.

```
SELECT *
FROM U INNER JOIN V ON A < V.C AND U.B <> V.B
```

Relational algebra expression (same as previous slide):

$U \bowtie_{A < V.C \ \text{AND} \ U.B \, != \, V.B} V$

Theta Joins with JOIN

```
SELECT m.*, g.genre

FROM Movies m

JOIN Genre g ON m.mid = g.mid AND g.genre = 'Sci-Fi';
```

SQL SELECT - Theta Join - WHERE vs JOIN … ON

Two ways to express theta join:

```
SELECT * FROM U, V WHERE A < V.C AND U.B <> V.B
```

```
SELECT * FROM U INNER JOIN V ON A < V.C AND U.B <> V.B
```

- What's the difference?
  - JOIN ON is the ANSI standard
  - Readability, especially with more than one join
  - Other types of joins we'll see in SQL (OUTER)

Recommendation: Use `JOIN` … `ON` for joins, `WHERE` clause for non-join conditions

Multiple Joins

A single SELECT statement may include any number of JOINs:

```
SELECT m.*, g.genre
FROM Movies m
JOIN Genre g ON g.mid = m.mid AND g.genre = 'Sci-Fi'
JOIN StarsIn s ON s.mid = m.mid AND sname = 'Will Smith';
```

For *inner joins*, the join order has no impact on results.

JOINs & WHERE

A SELECT statement may combine JOIN and WHERE:

```
SELECT m.*, g.genre
FROM Movies m
JOIN Genre g ON g.mid = m.mid AND g.genre = 'Sci-Fi'
WHERE m.year < 2000;
```

Corresponding Relational Algebra expression tree?

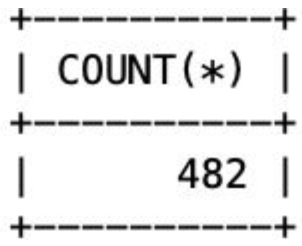
Aggregation Operators

SQL SELECT also supports **Aggregate Functions,** which summarize multiple rows.

- COUNT()
- MIN()
- MAX()
- SUM()
- AVG()

COUNT()

```
SELECT COUNT(*) FROM Movies;
```

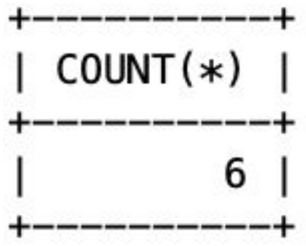


```
+----------+
| COUNT(*) |
+----------+
|      482 |
+----------+
```

```
SELECT COUNT(*)

FROM Movies

WHERE director = 'Steven Spielberg';
```



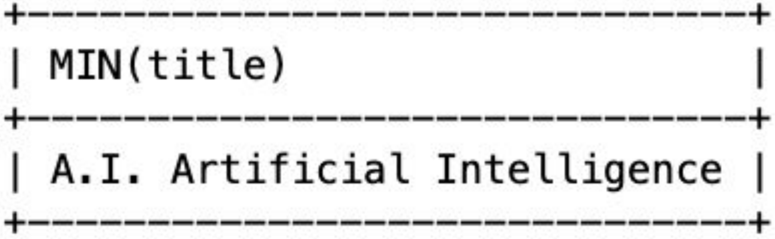
```
+----------+
| COUNT(*) |
+----------+
|        6 |
+----------+
```

MIN()

SELECT MIN(imdb) FROM Movies;

SELECT MIN(imdb) FROM Movies WHERE director = 'Steven Spielberg';

SELECT MIN(title) FROM Movies WHERE director = 'Steven Spielberg';



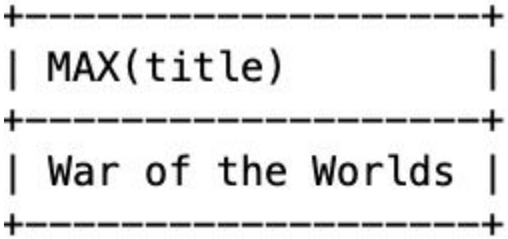
```
+------------------------------+
| MIN(title)                   |
+------------------------------+
| A.I. Artificial Intelligence |
+------------------------------+
```

MAX()

SELECT MAX(imdb) FROM Movies;

SELECT MAX(imdb) FROM Movies WHERE director = 'Steven Spielberg';

SELECT MAX(title) FROM Movies WHERE director = 'Steven Spielberg';



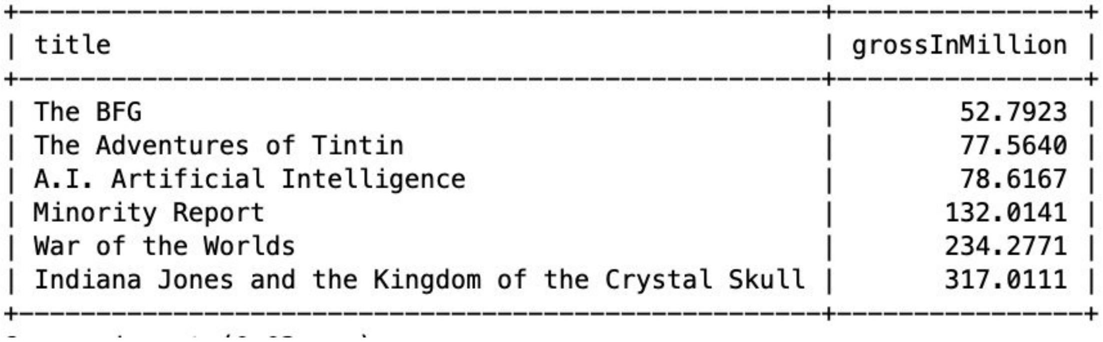
```
+-------------------+
| MAX(title)        |
+-------------------+
| War of the Worlds |
+-------------------+
```

SUM()

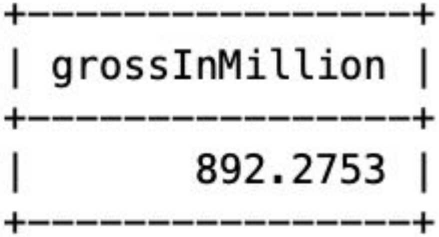
```
SELECT SUM(gross)/1000000 as grossInMillion

FROM Movies

WHERE director = 'Steven Spielberg';
```



```
+----------------------------------------------+----------------+
| title                                        | grossInMillion |
+----------------------------------------------+----------------+
| The BFG                                      |        52.7923 |
| The Adventures of Tintin                     |        77.5640 |
| A.I. Artificial Intelligence                 |        78.6167 |
| Minority Report                              |       132.0141 |
| War of the Worlds                            |       234.2771 |
| Indiana Jones and the Kingdom of the Crystal Skull |   317.0111 |
+----------------------------------------------+----------------+
```



```
+----------------+
| grossInMillion |
+----------------+
|       892.2753 |
+----------------+
```

AVG()

```
SELECT AVG(imdb) FROM Movies;

SELECT AVG(imdb) FROM Movies WHERE director = 'Steven Spielberg';

SELECT AVG(gross)/1000000 as grossInMillion FROM Movies;
```

```
SELECT AVG(gross)/1000000 as grossInMillion

FROM Movies

WHERE director = 'Steven Spielberg';
```

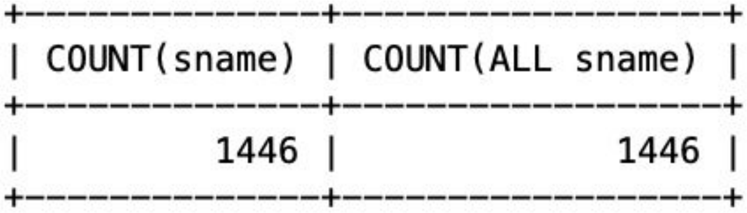
Aggregate Functions: ALL vs DISTINCT

By default, SQL aggregate functions operate in "ALL" mode:

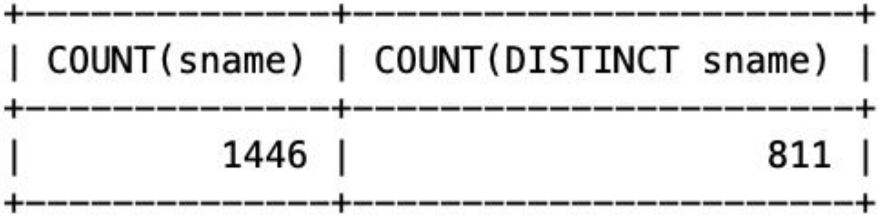
```
SELECT COUNT(sname), COUNT(ALL sname) FROM StarsIn;
```

DISTINCT causes the aggregate function to consider only unique values:

```
SELECT COUNT(sname), COUNT(DISTINCT sname) FROM StarsIn;
```



```
+--------------+------------------+
| COUNT(sname) | COUNT(ALL sname) |
+--------------+------------------+
|         1446 |             1446 |
+--------------+------------------+
```



```
+--------------+-----------------------+
| COUNT(sname) | COUNT(DISTINCT sname) |
+--------------+-----------------------+
|         1446 |                   811 |
+--------------+-----------------------+
```

The functions AVG(), COUNT(), and SUM() support DISTINCT mode

SQL SELECT - Aggregate Functions

- Used alone, an aggregate function collapses all rows into *one* summarized row.
- What if we want to include other column values in our result set?

```
SELECT director, AVG(imdb)
FROM Movies;
```

Grouping - GROUP BY

The GROUP BY clause, along with aggregate functions, allows us to identify which column(s) form the basis for our summary:

```
SELECT director, AVG(imdb)

FROM Movies

GROUP BY director;
```

GROUP BY

GROUP BY is permitted on both columns and the result of *scalar* functions:

```
SELECT
  FLOOR(DATEDIFF(CURRENT_DATE, birthdate) / 365) as age,
  COUNT(birthdate)
FROM Stars
GROUP BY FLOOR(DATEDIFF(CURRENT_DATE, birthdate) / 365);
```

GROUP BY with Conditions

Conditions on aggregate values require special treatment.

List all directors and average gross in million dollars per director of Movies made in or after year 2000 sorted in the order of the average gross.

```
SELECT director, AVG(gross)/1000000 as grossInMillion
FROM Movies
WHERE year >= 2000
GROUP BY director
ORDER BY grossInMillion;
```

GROUP BY with Conditions

The HAVING clause, used along with GROUP BY, allows us to apply conditions that **involve aggregate functions**.

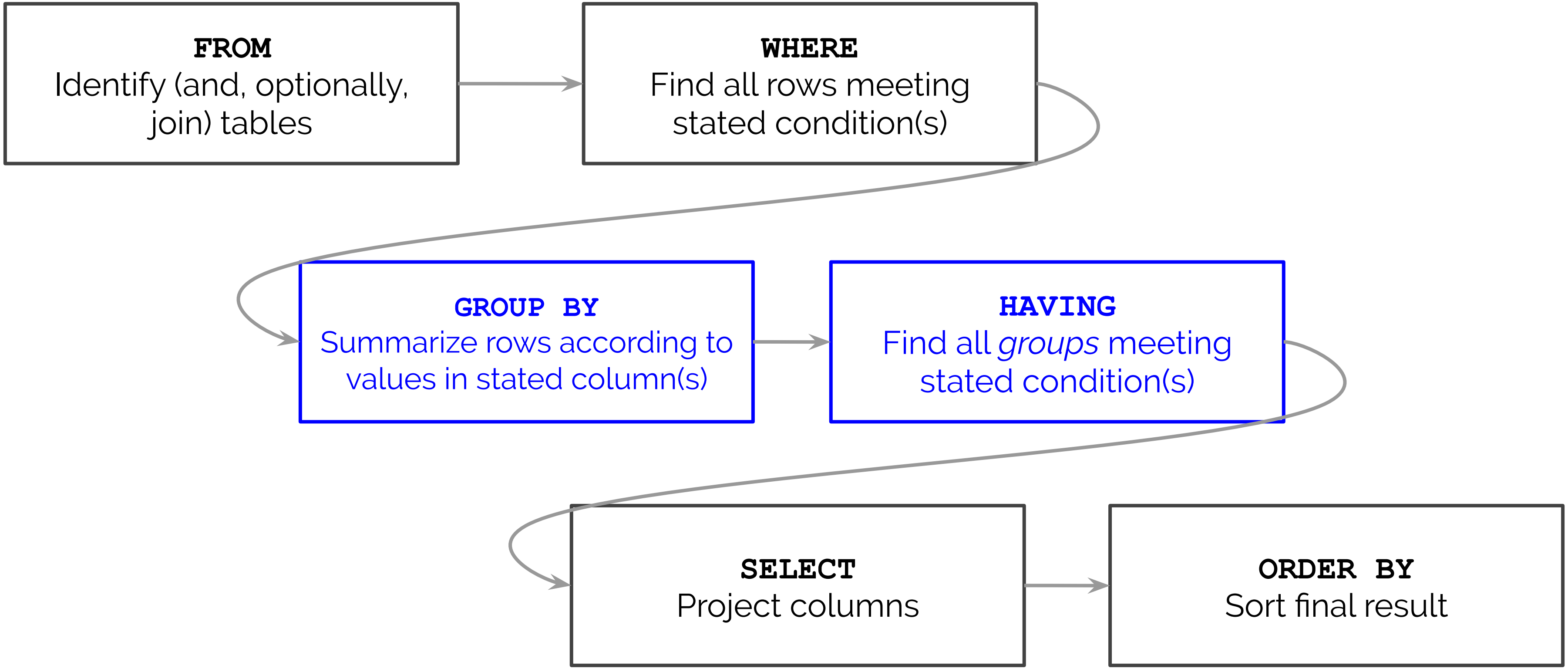
List directors and average gross in million dollars of Movies grouped by director whose average gross is greater than 100 million dollar sorted in the order of the average gross.

```
SELECT director, AVG(gross)/1000000 as grossInMillion
FROM Movies
GROUP BY director HAVING grossInMillion > 100
ORDER BY gorssInMillion;
```

Grouping, Aggregation, and Nulls

- The value NULL is ignored in any aggregation.
  - COUNT(A) where some values in A are NULL counts only non-NULL values in A.
- However, NULL is treated as an ordinary value when forming groups.
- The result of aggregation except for COUNT() over an empty bag of values is NULL. The COUNT of an empty bag is 0.

SQL SELECT Processing Order



FROM
Identify (and, optionally, join) tables

WHERE
Find all rows meeting stated condition(s)

GROUP BY
Summarize rows according to values in stated column(s)

HAVING
Find all *groups* meeting stated condition(s)

SELECT
Project columns

ORDER BY
Sort final result

Practice Writing Queries

Summary

SQL	Relational Algebra Operator	
`SELECT DISTINCT`	π	Projection
`FROM`	×	Cartesian product
`INNER JOIN ... ON`	$\bowtie_\theta$	Theta join
`WHERE`	σ	Selection
`AS`	ρ	Rename

Note: This is an oversimplified view. Treat it as a starting place, not the final story.