

Group By

The GROUP BY clause in SQL is used to group rows that have the same values in specified columns into summary rows, like finding the total sales per region, average salary per department, etc.

It's often used with aggregate functions like:

- SUM() – total
- AVG() – average
- COUNT() – number of items
- MIN() / MAX() – min or max value

For each year in the movies table, count how many movies were released.

```
mysql> select year, count(year) from movies GROUP BY year;
```

year	count(year)
2002	12056
2000	11643
1971	4072
1913	3690
1915	3722
1923	1759
1920	2308
1921	2041
2001	11690
1939	1996
1941	1829
1912	3232
1999	10976
1996	8362
1918	2122
1914	3441
2004	8718
1980	4673
1989	5697
1975	4384
1924	1847
1911	2026
1986	5284

This query is used to count how many movies were released each year, and then list the results sorted by year.

```
mysql> select year, count(year) from movies GROUP BY year ORDER BY year;
```

year	count(year)
1888	2
1890	3
1891	6
1892	9
1893	2
1894	59
1895	72
1896	410
1897	688
1898	1004
1899	845
1900	759
1901	837
1902	788
1903	831
1904	176
1905	232
1906	384
1907	332
1908	498
1909	812
1910	1276
1911	2026
1912	3232
1913	3690
1914	3441

It gives you a list of years and how many movies were released in each year, but this time, it sorts the result by the count, not by the year.

```
mysql> select year, count(year) year_count from movies GROUP BY year ORDER BY year_count;
```

year	year_count
2008	1
1893	2
1888	2
1890	3
1891	6
2007	7
1892	9
1894	59
1895	72
1904	176
2006	195
1905	232
1907	332
1906	384
1896	410
1908	498
1897	688
1900	759
1902	788
1909	812
1903	831
1901	837
1899	845

1979	4428	
1982	4597	
1983	4641	
1968	4668	
1980	4673	
1984	4937	
1985	5180	
1986	5284	
1987	5465	
1989	5697	
1988	5702	
1991	6068	
1990	6098	
1992	6580	
1993	6900	
1994	7502	
1995	7919	
1996	8362	
2004	8718	
1997	9023	
1998	10067	
1999	10976	
2000	11643	
2001	11690	
2003	11890	
2002	12056	
+-----+-----+		

Having

The HAVING clause is like WHERE but for groups.

- WHERE filters rows *before* grouping.
- HAVING filters groups *after* grouping.

You use HAVING with GROUP BY, usually when you want to filter based on an aggregate function (like COUNT(), SUM(), AVG(), etc.).

Syntax:

SELECT column, AGG_FUNC(column)

FROM table

GROUP BY column

HAVING AGG_FUNC(column) condition;

It retrieves a list of years from the movies table **where more than 2000 movies were released in that year.**

```
mysql> select year, count(year) year_count from movies GROUP BY year HAVING year_count>2000;
```

year	year_count
2002	12056
2000	11643
1971	4072
1913	3690
1915	3722
1920	2308
1921	2041
2001	11690
1912	3232
1999	10976
1996	8362
1918	2122
1914	3441
2004	8718
1980	4673
1989	5697
1975	4384
1911	2026
1986	5284
1968	4668
1987	5465
1951	2267
1917	2796
1988	5702
1993	6900
1994	7502

Order of Execution:

- 1) GROUP BY to create groups.
- 2) Apply the AGGREGATE function.
- 3) Apply HAVING condition.

Having often used along with GROUP BY, Not mandatory.

Select name, year from movies HAVING year>2000;

Note:

- 1) HAVING without GROUP BY is same as WHERE
- 2) WHERE is applied on individual rows while HAVING is applied on groups.

```
mysql> select year, count(year) year_count from movies WHERE rankscore>9 GROUP BY year HAVING year_count>20;
+-----+-----+
| year | year_count |
+-----+-----+
| 1996 |          34 |
| 2003 |          87 |
| 2004 |          48 |
| 2002 |          77 |
| 2000 |          45 |
| 1999 |          33 |
| 1982 |          24 |
| 2001 |          38 |
| 1997 |          23 |
| 1998 |          21 |
+-----+-----+
10 rows in set (0.12 sec)
```

The **order of SQL keywords** (also called the logical query processing order) is super important, especially when you're writing more complex queries.

Here's the typical **logical order** in which SQL processes the keywords (which is *different* from how we write them!):

SELECT column1, column2

FROM table_name

[JOIN another_table ON condition]

WHERE condition

GROUP BY column

HAVING condition

ORDER BY column

LIMIT number;

Join and Natural Join

A **JOIN** is used in SQL to combine rows from two or more tables, based on a **related column** between them.

```
mysql> Select m.name, g.genre from movies m JOIN movies_genres g ON m.id = g.movie_id LIMIT 20;
```

name	genre
#7 Train: An Immigrant Journey, The	Documentary
#7 Train: An Immigrant Journey, The	Short
\$	Comedy
\$	Crime
\$1,000 Reward	Western
\$1,000,000 Duck	Comedy
\$1,000,000 Duck	Family
\$10,000 Under a Pillow	Animation
\$10,000 Under a Pillow	Comedy
\$10,000 Under a Pillow	Short
\$100,000	Drama
\$100,000 Pyramid, The	Family
\$1000 a Touchdown	Comedy
\$20,000 Carat, The	Crime
\$20,000 Carat, The	Drama
\$20,000 Carat, The	Short
\$21 a Day Once a Month	Animation
\$21 a Day Once a Month	Short
\$2500 Bride, The	Drama
\$2500 Bride, The	Romance

20 rows in set (0.02 sec)

```
mysql> Select * from movies m JOIN movies_genres g ON m.id = g.movie_id LIMIT 20;
```

id	name	year	rankscore	movie_id	genre
1	#7 Train: An Immigrant Journey, The	2000	NULL	1	Documentary
1	#7 Train: An Immigrant Journey, The	2000	NULL	1	Short
2	\$	1971	6.4	2	Comedy
2	\$	1971	6.4	2	Crime
5	\$1,000 Reward	1923	NULL	5	Western
6	\$1,000,000 Duck	1971	5	6	Comedy
6	\$1,000,000 Duck	1971	5	6	Family
8	\$10,000 Under a Pillow	1921	NULL	8	Animation
8	\$10,000 Under a Pillow	1921	NULL	8	Comedy
8	\$10,000 Under a Pillow	1921	NULL	8	Short
9	\$100,000	1915	NULL	9	Drama
10	\$100,000 Pyramid, The	2001	NULL	10	Family
11	\$1000 a Touchdown	1939	6.7	11	Comedy
12	\$20,000 Carat, The	1913	NULL	12	Crime
12	\$20,000 Carat, The	1913	NULL	12	Drama
12	\$20,000 Carat, The	1913	NULL	12	Short
13	\$21 a Day Once a Month	1941	NULL	13	Animation
13	\$21 a Day Once a Month	1941	NULL	13	Short
14	\$2500 Bride, The	1912	NULL	14	Drama
14	\$2500 Bride, The	1912	NULL	14	Romance

20 rows in set (0.01 sec)

Table aliases: **m** and **g**

A **NATURAL JOIN** is a type of join that automatically matches columns with the **same name and data type** in both tables.

```
mysql> select name, genre from movies natural join movies_genres limit 20;
```

name	genre
"Girl in a Black Bikini"	Documentary
"Girl from U.N.C.L.E., The"	Documentary
"Girl From Tomorrow, The"	Documentary
"Girl From Tomorrow Part Two: Tomorrow's End, The"	Documentary
"Girl Friday"	Documentary
"Girl About Town"	Documentary
"Girasoles para Luca"	Documentary
"Gioved della signora Giulia, I"	Documentary
"Giovani, carini ma disoccupati"	Documentary
"Giovane Garibaldi, Il"	Documentary
"Giorni da Leone"	Documentary
"Giornalisti"	Documentary
"Giocando a golf una mattina"	Documentary
"Gintberg Show Off"	Documentary
"Gintberg - var det det?"	Documentary
"Gintberg - men nok om mig"	Documentary
"Ginr kaiki fairu"	Documentary
"Gino"	Documentary
"Gingerbread Girl, The"	Documentary
"Ginger Tree, The"	Documentary

```
20 rows in set (0.00 sec)
```

Inner, Left, Right and Outer joins

LEFT JOIN (or LEFT OUTER JOIN):

Returns all rows from the left table and the matched rows from the right table.

If no match, NULLs are returned from the right.

```
mysql> Select * from movies m LEFT JOIN movies_genres g ON m.id = g.movie_id LIMIT 20;
```

id	name	year	rankscore	movie_id	genre
0	#28	2002	NULL	NULL	NULL
1	#7 Train: An Immigrant Journey, The	2000	NULL	1	Documentary
1	#7 Train: An Immigrant Journey, The	2000	NULL	1	Short
2	\$	1971	6.4	2	Comedy
2	\$	1971	6.4	2	Crime
3	\$1,000 Reward	1913	NULL	NULL	NULL
4	\$1,000 Reward	1915	NULL	NULL	NULL
5	\$1,000 Reward	1923	NULL	5	Western
6	\$1,000,000 Duck	1971	5	6	Comedy
6	\$1,000,000 Duck	1971	5	6	Family
7	\$1,000,000 Reward, The	1920	NULL	NULL	NULL
8	\$10,000 Under a Pillow	1921	NULL	8	Animation
8	\$10,000 Under a Pillow	1921	NULL	8	Comedy
8	\$10,000 Under a Pillow	1921	NULL	8	Short
9	\$100,000	1915	NULL	9	Drama
10	\$100,000 Pyramid, The	2001	NULL	10	Family
11	\$1000 a Touchdown	1939	6.7	11	Comedy
12	\$20,000 Carat, The	1913	NULL	12	Crime
12	\$20,000 Carat, The	1913	NULL	12	Drama
12	\$20,000 Carat, The	1913	NULL	12	Short

20 rows in set (0.01 sec)

RIGHT JOIN (or RIGHT OUTER JOIN):

Opposite of LEFT JOIN — returns all rows from the right table and matched rows from the left.

```
mysql> Select * from movies m RIGHT JOIN movies_genres g ON m.id = g.movie_id LIMIT 20;
```

id	name	year	rankscore	movie_id	genre
1	#7 Train: An Immigrant Journey, The	2000	NULL	1	Documentary
1	#7 Train: An Immigrant Journey, The	2000	NULL	1	Short
2	\$	1971	6.4	2	Comedy
2	\$	1971	6.4	2	Crime
5	\$1,000 Reward	1923	NULL	5	Western
6	\$1,000,000 Duck	1971	5	6	Comedy
6	\$1,000,000 Duck	1971	5	6	Family
8	\$10,000 Under a Pillow	1921	NULL	8	Animation
8	\$10,000 Under a Pillow	1921	NULL	8	Comedy
8	\$10,000 Under a Pillow	1921	NULL	8	Short
9	\$100,000	1915	NULL	9	Drama
10	\$100,000 Pyramid, The	2001	NULL	10	Family
11	\$1000 a Touchdown	1939	6.7	11	Comedy
12	\$20,000 Carat, The	1913	NULL	12	Crime
12	\$20,000 Carat, The	1913	NULL	12	Drama
12	\$20,000 Carat, The	1913	NULL	12	Short
13	\$21 a Day Once a Month	1941	NULL	13	Animation
13	\$21 a Day Once a Month	1941	NULL	13	Short
14	\$2500 Bride, The	1912	NULL	14	Drama
14	\$2500 Bride, The	1912	NULL	14	Romance

20 rows in set (0.00 sec)

FULL OUTER JOIN:

Returns all rows from both tables.

Non-matching rows get NULLs.

NOTE: MySQL **does not support** FULL OUTER JOIN directly (unlike some other databases like PostgreSQL or SQL Server). So if you're trying to use:

```
ERROR 1064 (42000): You have an error in your SQL syntax; check the manual that corresponds to your MySQL server version for the right syntax to use near 'FULL OUTER JOIN movies_genres ON movies.id = movie_genres.movie_id' at line 1
```

If you want to simulate a FULL OUTER JOIN in MySQL, you can **combine a LEFT JOIN and a RIGHT JOIN using UNION**:

```
mysql> Select * From movies m LEFT JOIN movies_genres g ON m.id = g.movie_id UNION Select * From movies m RIGHT JOIN movies_genres g ON m.id = g.movie_id LIMIT 30;
```

id	name	year	rankscore	movie_id	genre
0	#28	2002	NULL	NULL	NULL
1	#7 Train: An Immigrant Journey, The	2000	NULL	1	Documentary
1	#7 Train: An Immigrant Journey, The	2000	NULL	1	Short
2	\$	1971	6.4	2	Comedy
2	\$	1971	6.4	2	Crime
3	\$1,000 Reward	1913	NULL	NULL	NULL
4	\$1,000 Reward	1915	NULL	NULL	NULL
5	\$1,000 Reward	1923	NULL	5	Western
6	\$1,000,000 Duck	1971	5	6	Comedy
6	\$1,000,000 Duck	1971	5	6	Family
7	\$1,000,000 Reward, The	1920	NULL	NULL	NULL
8	\$10,000 Under a Pillow	1921	NULL	8	Animation
8	\$10,000 Under a Pillow	1921	NULL	8	Comedy
8	\$10,000 Under a Pillow	1921	NULL	8	Short
9	\$100,000	1915	NULL	9	Drama
10	\$100,000 Pyramid, The	2001	NULL	10	Family
11	\$1000 a Touchdown	1939	6.7	11	Comedy
12	\$20,000 Carat, The	1913	NULL	12	Crime
12	\$20,000 Carat, The	1913	NULL	12	Drama
12	\$20,000 Carat, The	1913	NULL	12	Short
13	\$21 a Day Once a Month	1941	NULL	13	Animation
13	\$21 a Day Once a Month	1941	NULL	13	Short
14	\$2500 Bride, The	1912	NULL	14	Drama
14	\$2500 Bride, The	1912	NULL	14	Romance
14	\$2500 Bride, The	1912	NULL	14	Short
15	\$30	1999	7.5	15	Comedy
15	\$30	1999	7.5	15	Short
16	\$30,000	1920	NULL	NULL	NULL
17	\$300 y tickets	2002	NULL	17	Short
18	\$40,000	1996	9.6	18	Comedy

30 rows in set (0.00 sec)

3 way joins and k way joins

```
mysql> select a.first_name, a.last_name from actors a JOIN roles r ON a.id = r.actor_id JOIN movies m on m.id = r.movie_id AND m.name='Officer 444';
```

first_name	last_name
Frank (I)	Baker
Arthur	Beckel
Al	Ferguson
Francis	Ford
Philip	Ford
Harry	McDonald
Lafe	McKee
Jack	Mower
August	Vollmer
Ben F.	Wilson
Neva	Gerber
Margaret	Mann
Ruth	Royce

13 rows in set (0.03 sec)

Sub Queries

A **subquery** (also called an **inner query** or **nested query**) is a query **inside another query**. It's used to help the outer (main) query by providing a value or set of values.

Syntax:

SELECT column_name

FROM table_name

WHERE column_name **OPERATOR** (

SELECT column_name

FROM another_table

WHERE condition

);

Where Can You Use Subqueries?

Subqueries can be used in:

1. SELECT clause
2. FROM clause
3. WHERE clause
4. HAVING clause

Notes on Subqueries

- Subqueries must return the right number of rows and columns for the context (e.g., scalar vs. list).
- You can use operators like IN, =, ANY, ALL, EXISTS with subqueries.

The SQL command retrieves the **first and last names of actors** who appeared in the movie *"Schindler's List"*.

```
mysql> select first_name, last_name from actors Where id IN
-> (select actor_id from roles Where movie_id IN
-> (select id from movies where name='Schindler\'s List')
-> );
```

first_name	last_name
Peter	Appiano
Joachim Paul	Assböck
Hans-Jörg	Assmann
Uri	Avrahami
Joseph	Bau
Sigurd	Bemme
Dirk	Bender
Martin S.	Bergmann
Henryk	Bista
Tadeusz	Bradecki
Stanislaw	Brejdygant
Alexander	Buczolich
Haymon Maria	Buttinger
Piotr	Cyrwus
Ezra	Dagan
Grzegorz	Damiecki
Tomasz	Dedek
Daniel	Del Ponte
Pawel	Delag
Janek	Dresner
Ralph	Fiennes
Peter	Flechtner
Jeremy (I)	Flynn
Marian	Glinka
Michael (II)	Gordon
Gerald Alexander	Held
Rami	Heuberger
Michael Z.	Hoffmann
Slawomir	Holland
Ryszard	Horowitz
Tadeusz	Huk
Mark	Ivanir
Jan	Jurewicz
Piotr	Kadlcik
Georges	Kern
Ben	Kingsley
Wojciech	Klata
Stanislaw	Koczanowicz
Wieslaw	Komasa

Explanation:

1. Innermost Query:

select id from movies where name='Schindler\'s List';

- This part identifies the ID of the movie titled *"Schindler's List"* by searching the movies table.

2. Middle Query:

select actor_id from roles where movie_id IN (result of the innermost query);

- This takes the movie ID obtained in the innermost query and looks up the roles table to find the actor_id of all actors who appeared in the movie.

3. Outer Query:

select first_name, last_name from actors where id IN (result of the middle query);

- Finally, this query takes the actor IDs from the middle query and retrieves the first and last names of these actors from the actors table.

```
mysql> select * from movies where rankscore >= ALL (select MAX(rankscore) from movies);
```

id	name	year	rankscore
23608	Atunci i-am condamnat pe toti la moarte	1971	9.9
41537	Blow Job	2002	9.9
65522	Clearing, The	2001	9.9
68770	Complex Sessions, The	1994	9.9
79678	Dawn of the Friend	2004	9.9
84983	Devil's Circus, The	1926	9.9
87894	Distinto amanecer	1943	9.9
91477	Dosti	1964	9.9
94558	Duck Soup	1942	9.9
95123	Duminica la ora 6	1965	9.9
125616	Genet parle d'Angela Davis	1970	9.9
131247	Gong fu qi jie	1979	9.9
145429	Himala	1982	9.9
153301	Huttyn	1996	9.9
163898	Ivan Groznyj III	1988	9.9
171541	Jnos vitz	1973	9.9
205227	Marche des femmes Hendaye, La	1975	9.9
227596	Napolon Bonaparte	1934	9.9
230568	New Clear Farm	1998	9.9
230864	New World, The	1982	9.9
246768	Pair of Boots, A	1962	9.9
264084	Prince Solitaire	2003	9.9
266013	Prostitues de Lyon parlent, Les	1975	9.9
288249	Sargam	1995	9.9
289646	Scarmour	1997	9.9

The SQL query identify movies from the movies table that have the highest rank score.

Explanation:

1. Inner Query:

select MAX(rankscore) from movies;

- This part retrieves the highest rankscore value from the movies table. It ensures you're working with the maximum score available.

2. Outer Query:

select * from movies where rankscore >= ALL (result of inner query);

- The outer query filters all rows in the movies table where the rankscore is greater than or equal to the maximum rankscore identified by the inner query.