

M1

SQL Sets and Set Operations

Set operators combine two query results.

Four set operations on two query results

Combine A and B four ways. Color = membership: teal both, blue only A, violet only B.

Query A
users averaging above 4.0

name
Mickey
Minnie

Query B
users who played Evermore

name
Mickey
Daffy

combine the two results four ways:

UNION
either side, deduped

name
Mickey
Minnie
Daffy

UNION ALL
either side, keep dups

name
Mickey
Minnie
Mickey
Daffy

Mickey appears twice

INTERSECT
in BOTH A and B

name
Mickey

EXCEPT
in A, not in B

name
Minnie

M1

JOINS: Combining Tables

A JOIN pairs rows from two tables when a predicate in ON evaluates to true.

Users joined to Listens on user_id

```
-- Keep matched pairs; users without listens drop.  
SELECT u.user_id, u.name, l.song_id, l.rating  
FROM Users u  
INNER JOIN Listens l ON u.user_id = l.user_id  
ORDER BY u.user_id
```

Users 4 rows

user_id	name
1	Mickey
2	Minnie
3	Daffy
4	Pluto

on

-- no match

Listens 9 rows

user_id	song_id	rating
1	1	4.5
1	2	4.2
1	6	3.9
2	2	4.7
2	7	4.6
2	8	3.9
3	1	2.9
3	2	4.9
3	6	NULL

JOIN
ON user_id

Users ⋈ Listens

9 rows

user_id	name	song_id	rating
1	Mickey	1	4.5
1	Mickey	2	4.2
1	Mickey	6	3.9
2	Minnie	2	4.7
2	Minnie	7	4.6
2	Minnie	8	3.9
3	Daffy	1	2.9
3	Daffy	2	4.9
3	Daffy	6	NULL

INNER JOIN → matched pairs only · 9 rows

Multiple JOINS: users, listens, and songs

```
-- Get user + song details for high-rated listens (rating > 4.0)
SELECT u.name, s.title, s.artist, l.rating
FROM Users u
INNER JOIN Listens l ON u.user_id = l.user_id
INNER JOIN Songs s ON l.song_id = s.song_id
WHERE l.rating > 4.0
ORDER BY u.name, l.rating DESC;
```

Multiple JOINS: five listens rated above 4.0

name	title	artist	rating
Daffy	Willow	Taylor Swift	4.9
Mickey	Evermore	Taylor Swift	4.5
Mickey	Willow	Taylor Swift	4.2
Minnie	Willow	Taylor Swift	4.7
Minnie	Yellow Sub	Beatles	4.6

Users joined to Listens on user_id; LEFT JOIN keeps the unmatched Pluto

```
-- Keep every user; missing listen columns are NULL.  
SELECT u.user_id, u.name, l.song_id, l.rating  
FROM Users u  
LEFT JOIN Listens l ON u.user_id = l.user_id  
ORDER BY u.user_id
```

Users 4 rows

user_id	name
1	Mickey
2	Minnie
3	Daffy
4	Pluto

on

-- no match

Listens 9 rows

user_id	song_id	rating
1	1	4.5
1	2	4.2
1	6	3.9
2	2	4.7
2	7	4.6
2	8	3.9
3	1	2.9
3	2	4.9
3	6	NULL

JOIN
ON user_id**Users ⋈ Listens**

INNER 9 · LEFT 10

user_id	name	song_id	rating
1	Mickey	1	4.5
1	Mickey	2	4.2
1	Mickey	6	3.9
2	Minnie	2	4.7
2	Minnie	7	4.6
2	Minnie	8	3.9
3	Daffy	1	2.9
3	Daffy	2	4.9
3	Daffy	6	NULL
4	Pluto	NULL	NULL

INNER JOIN → matched pairs only · 9 rows

LEFT JOIN → keeps Pluto, song_id / rating NULL-filled · 10 rows

Listens RIGHT JOIN Songs keeps the five unplayed songs

```
-- Keep every song; missing listen columns are NULL.  
SELECT s.song_id, s.title, l.rating  
FROM Listens l  
RIGHT JOIN Songs s ON l.song_id = s.song_id
```

Listens 9 rows

song_id	rating
1	4.5
1	2.9
2	4.2
2	4.7
2	4.9
6	3.9
6	NULL
7	4.6
8	3.9

on

Songs 10 rows

song_id	title	
1	Evermore	
2	Willow	
3	Shape of You	no match
4	Photograph	no match
5	Shivers	no match
6	Yesterday	
7	Yellow Submarine	
8	Hey Jude	
9	Bad Blood	no match
10	DJ Mix	no match

RIGHT
JOIN
ON song_id

Listens RIGHT JOIN Songs

INNER 9 · RIGHT 14

song_id	title	rating
1	Evermore	4.5
1	Evermore	2.9
2	Willow	4.2
2	Willow	4.7
2	Willow	4.9
6	Yesterday	3.9
6	Yesterday	NULL
7	Yellow Submarine	4.6
8	Hey Jude	3.9
3	Shape of You	NULL
4	Photograph	NULL
5	Shivers	NULL
9	Bad Blood	NULL
10	DJ Mix	NULL

INNER JOIN → only songs with a listen · 9 rows

RIGHT JOIN → keeps the 5 songs nobody played, NULL-filled · 14 rows

A foreign key guarantees one direction

Listens.user_id is a non-NULL foreign key to Users.user_id.

Every listen has a user.

The foreign key guarantees a match.

A user can have no listens.

Pluto exists, but has never listened.

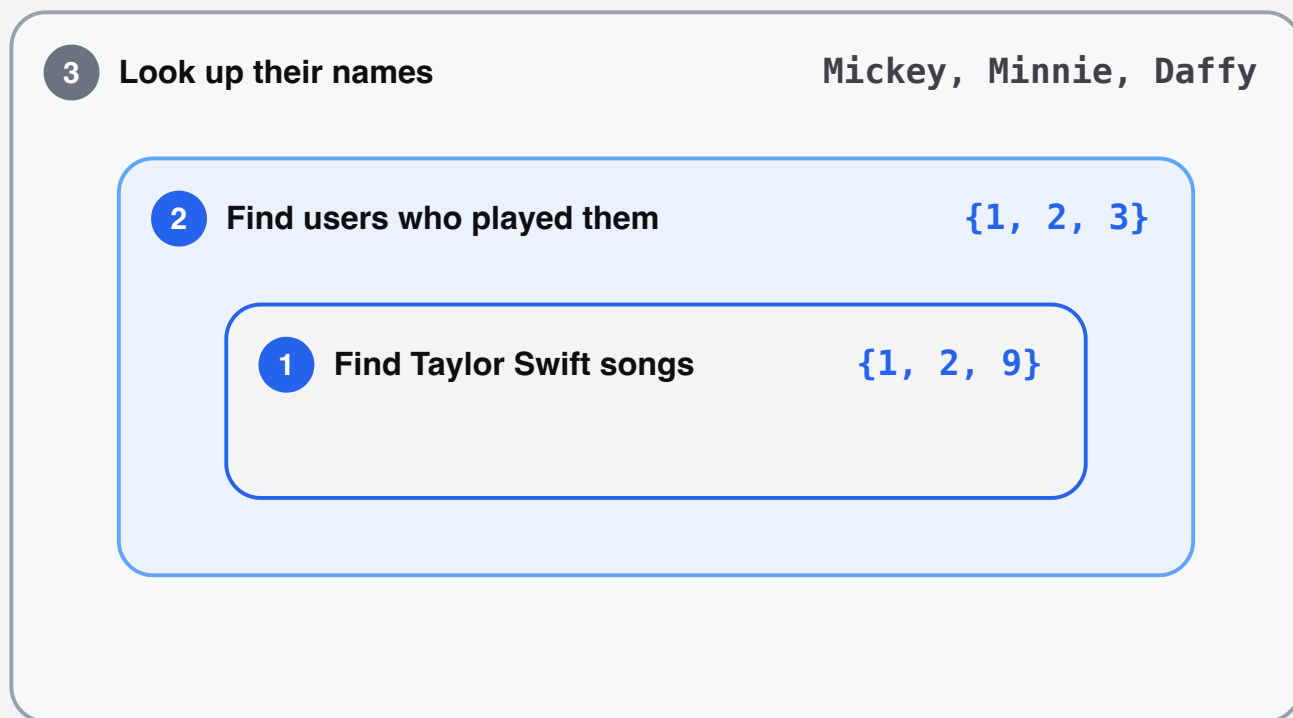
M1

Subqueries: Basic Patterns

A subquery is a SELECT nested inside another query.

Subqueries: queries inside queries

Read inside-out: the innermost query runs first and feeds the next, layer by layer.



Three flavors

IN / NOT IN

is a value in the inner list?

EXISTS

does the inner query return any row?

Scalar

compare against one inner value

Read the query inside-out

```
-- Step 3 (LAST): pick names for the users found in Step 2.
SELECT name
FROM Users
WHERE user_id IN (
    -- Step 2: distinct users who listened to those songs.
    SELECT DISTINCT user_id
    FROM Listens
    WHERE song_id IN (
        -- Step 1 (FIRST): all Taylor Swift song IDs. Returns {1, 2, 9}.
        SELECT song_id
        FROM Songs
        WHERE artist = 'Taylor Swift'
    )
)
```

IN subquery: walk the data inside-out

1. Songs WHERE artist = 'Taylor Swift'

song_id	artist
1	Taylor Swift
2	Taylor Swift
3	Ed Sheeran
4	Ed Sheeran
5	Ed Sheeran
6	Beatles
7	Beatles
8	Beatles
9	Taylor Swift
10	DD

{1, 2, 9}

2. Listens on those songs distinct

user_id	song_id
1	1
1	2
2	2
3	1
3	2

{1, 2, 3}

3. Users for {1, 2, 3} 3 rows

name	user_id
Mickey	1
Minnie	2
Daffy	3

Result: Mickey, Minnie, Daffy

The NULL-in-NOT-IN Trap

Including NULL

Daffy's ratings form the list

```
SELECT listen_id, rating
FROM Listens
WHERE rating NOT IN (
    SELECT rating FROM Listens
    WHERE user_id = 3
)
```

Test rating: 4.5 | List: (2.9, 4.9, NULL)

4.5 != 2.9	TRUE
4.5 != 4.9	TRUE
4.5 != NULL	UNKNOWN

Combined with AND: UNKNOWN

Query result: 0 rows

Filtering out NULL

add AND rating IS NOT NULL

```
SELECT listen_id, rating
FROM Listens
WHERE rating NOT IN (
    SELECT rating FROM Listens
    WHERE user_id = 3
    AND rating IS NOT NULL
)
```

Test rating: 4.5 | List: (2.9, 4.9)

4.5 != 2.9	TRUE
4.5 != 4.9	TRUE

Combined with AND: TRUE

Query result: 6 rows

Scalar subquery: one value, reused for every row

```
SELECT u.name, l.rating
FROM Users u
JOIN Listens l ON u.user_id = l.user_id
WHERE l.rating > (
    SELECT AVG(rating) FROM Listens
)
```

Inner subquery, runs once

AVG(rating) over all Listens

the one NULL rating is skipped (8 ratings)

= 4.2

one value,
→
every row

Outer query: keep l.rating > 4.2

name	rating	> 4.2 ?
Mickey	4.5	✓
~~Mickey~~	~~4.2~~	~~x~~
~~Mickey~~	~~3.9~~	~~x~~
Minnie	4.7	✓
Minnie	4.6	✓
~~Minnie~~	~~3.9~~	~~x~~
~~Daffy~~	~~2.9~~	~~x~~
Daffy	4.9	✓
~~Daffy~~	~~NULL~~	~~? UNKNOWN~~

4 rows kept

Basic: one global average. Correlated: each user's own.

Which users have a rating above their own average?

Listens 9 rows

name	rating
Mickey	4.5
Mickey	4.2
Mickey	3.9
Minnie	4.7
Minnie	4.6
Minnie	3.9
Daffy	2.9
Daffy	4.9
Daffy	NULL

Basic subquery: one global average

AVG over all ratings

4.2

every user is compared to the same 4.2

Correlated subquery: each user's own average

user	AVG of THEIR ratings
Mickey	4.2
Minnie	4.4
Daffy	3.9

each colored arrow is the correlation: WHERE user_id = u.user_id keeps only that user's rows

Intuition: a double-loop model

Conceptual pseudocode. The optimizer may use a different, equivalent execution plan.

the inner reads
the outer's u:
the correlation

```
for u in Users:
    u_avg = (          # u's own average
        SELECT AVG(rating)
        FROM Listens l2
        WHERE l2.user_id = u.user_id
    )
    keep u when u has a rating above u_avg
```

Two scopes, cross-referenced

- u** outer scope:
the current user (outer loop)
- l2** inner scope:
this query's own Listens

The inner WHERE reaches OUT to u, the outer loop's variable. That cross-reference is the correlation.

In this conceptual model, each user supplies the rows for their own average:

u = Mickey

AVG = 4.2

u = Minnie

AVG = 4.4

u = Daffy

AVG = 3.9

The inner query follows the outer user

```
SELECT DISTINCT u.name
FROM Users u
JOIN Listens l ON l.user_id = u.user_id
WHERE l.rating > (
    SELECT AVG(rating)
    FROM Listens l2
    WHERE l2.user_id = u.user_id
);
```

User ratings above their average

user	rating	their average	above their average?
Mickey	4.5	4.2	TRUE
Mickey	4.2	4.2	FALSE
Mickey	3.9	4.2	FALSE
Minnie	4.7	4.4	TRUE
Minnie	4.6	4.4	TRUE
Minnie	3.9	4.4	FALSE
Daffy	2.9	3.9	FALSE
Daffy	4.9	3.9	TRUE
Daffy	NULL	3.9	UNKNOWN

Pluto has no listens: the INNER JOIN excludes him.

A user is kept when any of their rows survives.

Output: Mickey, Minnie, Daffy

NOT EXISTS: users with no rating above 4.5

```
SELECT u.name FROM Users u
WHERE NOT EXISTS (
  SELECT 1 FROM Listens l
  WHERE l.user_id = u.user_id AND l.rating > 4.5
)
```

user	rating
Mickey	4.5
Mickey	4.2
Mickey	3.9
Minnie	4.7
Minnie	4.6
Minnie	3.9
Daffy	2.9
Daffy	4.9
Daffy	NULL

Pluto has no listens, so NOT EXISTS keeps him.

NOT EXISTS result

name
Mickey
Pluto

CTEs: name the steps of a query

Goal: find the users who listen to Classic songs.

Nested subqueries

Read inside-out · hard to debug

SELECT name FROM Users

WHERE user_id IN (...)

Step 3 (outer)

SELECT user_id FROM Listens

WHERE song_id IN (...)

Step 2 (middle)

SELECT song_id FROM Songs

WHERE genre = 'Classic'

Step 1 (innermost)

transform

CTE approach

Linear · clear · maintainable

WITH ClassicSongs

Step 1: find Classic songs

SELECT song_id FROM Songs WHERE genre = 'Classic'

→ {6, 7, 8}



ClassicListeners

Step 2: find users who listened

SELECT DISTINCT user_id FROM Listens
WHERE song_id IN (SELECT song_id FROM ClassicSongs)

→ {1, 2, 3}



Final query

Step 3: get user names

SELECT name FROM Users
WHERE user_id IN (SELECT user_id FROM ClassicListeners)

→ {Mickey, Minnie, Daffy}

Problems

Must read inside-out

Hard to debug a single step

Cannot reuse a piece

Benefits

Named, self-documenting steps

Test each intermediate result on its own

Reuse a CTE

Name a result, then query it

```
-- Step 1: WITH names a result you can reuse
WITH HighRatedSongs AS (
    SELECT song_id, AVG(rating) AS avg_rating
    FROM Listens
    GROUP BY song_id
    HAVING AVG(rating) > 4.0
)
-- Step 2: the outer query reads it by name
SELECT s.title, h.avg_rating
FROM HighRatedSongs h
JOIN Songs s ON h.song_id = s.song_id
```

From song IDs to song titles

HighRatedSongs

song_id	avg_rating
2	4.6
7	4.6

Final Output

title	avg_rating
Willow	4.6
Yellow Submarine	4.6

The CTE keeps the two songs averaging above 4.0. Joining to Songs adds their titles.

Combine two named results

```
-- Step 1: count every user's listens
WITH UserListenCounts AS (
    SELECT user_id, COUNT(*) AS listen_count
    FROM Listens
    GROUP BY user_id
),
-- Step 2: average only their 4+ ratings
UserHighRatings AS (
    SELECT user_id, AVG(rating) AS high_avg, COUNT(*) AS high_count
    FROM Listens
    WHERE rating >= 4.0
    GROUP BY user_id
)
-- Step 3: join both CTEs back to Users
SELECT u.name, lc.listen_count, hr.high_avg, hr.high_count
FROM Users u
LEFT JOIN UserListenCounts lc ON u.user_id = lc.user_id
LEFT JOIN UserHighRatings hr ON u.user_id = hr.user_id
```

Two CTEs, two intermediate tables

UserListenCounts: all listens

user_id	listen_count
1	3
2	3
3	3

UserHighRatings: ratings $\geq$ 4.0

user_id	high_avg	high_count
1	4.35	2
2	4.65	2
3	4.9	1

UserHighRatings filters before averaging. Pluto has no listens, so neither CTE contains his user ID.

LEFT JOIN keeps Pluto in the final output

Final Output

name	listen_count	high_avg	high_count
Mickey	3	4.35	2
Minnie	3	4.65	2
Daffy	3	4.9	1
Pluto	NULL	NULL	NULL

Both LEFT JOINs preserve every row of Users. Pluto has no matching CTE row, so the joined columns are NULL, including listen_count.

GROUP BY collapses the rows; a window keeps them and adds a column

Listens

user_id	song_id	rating
1	1	4.5
1	2	4.2
1	6	3.9
2	2	4.7
2	7	4.6
2	8	3.9
3	1	2.9
3	2	4.9
3	6	NULL

GROUP BY user_id

user_id	avg_rating
1	4.2
2	4.4
3	3.9

GROUP BY



9 rows → 3, detail gone

AVG(rating) OVER (PARTITION BY user_id)

user_id	song_id	rating	user_avg
1	1	4.5	4.2
1	2	4.2	4.2
1	6	3.9	4.2
2	2	4.7	4.4
2	7	4.6	4.4
2	8	3.9	4.4
3	1	2.9	3.9
3	2	4.9	3.9
3	6	NULL	3.9

or keep
every row

the same 9 Listens rows, plus a user_avg column

GROUP BY collapses rows

```
-- One row per user; detail rows are gone.  
SELECT user_id, AVG(rating) AS avg_rating  
FROM Listens  
GROUP BY user_id;
```

OVER attaches the average to every listen

```
-- Keep all 9 rows; attach each user's average.  
SELECT user_id, song_id, rating,  
       AVG(rating) OVER (PARTITION BY user_id) AS user_avg  
FROM Listens  
ORDER BY user_id, song_id;
```

RANK in three steps: partition, order, number

```
-- Rank each user's listens by rating, highest first.
SELECT user_id, song_id, rating,
       RANK() OVER (PARTITION BY user_id ORDER BY rating DESC NULLS LAST) AS rank
FROM Listens
```

Step 1: PARTITION BY user_id

user_id	rating
1	4.5
1	4.2
1	3.9
2	4.7
2	4.6
2	3.9
3	2.9
3	4.9
3	NULL

Without PARTITION BY: one global ranking

Step 2: ORDER BY rating DESC NULLS LAST

user_id	rating
1	4.5
1	4.2
1	3.9
2	4.7
2	4.6
2	3.9
3	4.9
3	2.9
3	NULL

ordered: user 1

ordered: user 2

ordered: user 3

DESC: highest first; NULLS LAST: unrated last

Step 3: assign the rank

user_id	rating	rank
1	4.5	1
1	4.2	2
1	3.9	3
2	4.7	1
2	4.6	2
2	3.9	3
3	4.9	1
3	2.9	2
3	NULL	3

top = 1, then 2, then 3 per lane

Three ways to rank, when there is a tie

Mickey's listens by rating, plus a hypothetical second 4.5. The functions agree until the tie.

rating	ROW_NUMBER	RANK	DENSE_RANK
4.5	1	1	1
4.5	2	1	1
4.2	3	3	2
3.9	4	4	3

never ties

shares 1, then SKIPS
to 3 (a gap)

shares 1, stays PACKED
at 2 (no gap)

M1

SQL Problem Solving: Reading and Debugging Queries

Read and debug a query by tracing its input rows through each clause.

What does this query return?

Trace FROM/JOIN, GROUP BY, and HAVING to check the result.

```
-- Find user-genre pairs with at least two listens in that genre.  
SELECT u.user_id, s.genre, COUNT(*) AS song_count  
FROM Users u  
JOIN Listens l ON l.user_id= u.user_id  
JOIN Songs s ON s.song_id= l.song_id  
GROUP BY u.user_id, s.genre  
HAVING COUNT(*) >= 2
```

Step 1 · FROM + JOIN

9 rows, every listen with its genre

user	genre	rating
1	Pop	4.5
1	Pop	4.2
1	Classic	3.9
2	Pop	4.7
2	Classic	4.6
2	Classic	3.9
3	Pop	2.9
3	Pop	4.9
3	Classic	NULL

GROUP BY

Step 2 · GROUP BY

9 rows -> 6 user-genre groups

user	genre	count
1	Pop	2
1	Classic	1
2	Pop	1
2	Classic	2
3	Pop	2
3	Classic	1

HAVING

Step 3 · HAVING COUNT(*) >= 2

6 groups -> 3 survive

user	genre	count	
1	Pop	2	✓
1	Classic	1	
2	Pop	1	
2	Classic	2	✓
3	Pop	2	✓
3	Classic	1	

Result: 3 user-genre pairs.

Read the Query as a Sequence of Tables

Start with the inputs, not the first line of SQL.

1. **Name one row.** After FROM and JOIN, is it a listen, a user, or a user–genre pair?
2. **Read each named result.** For each CTE, identify its columns and what one output row represents before reading the query that uses it.
3. **Track what changes.** WHERE and HAVING remove rows or groups. GROUP BY combines rows; a window adds a value while keeping the rows.
4. **Describe the answer.** Check SELECT, DISTINCT, ORDER BY, and LIMIT: which columns, which duplicates, which order, and how many rows?

Next example: join listens to users and songs → group by user and genre → keep groups with at least two listens.

Reading Example 1: Finding Top Genres Per User

*-- Goal: For each user, find genres with 2+ listens.
-- Read in logical order: FROM/JOIN → WHERE → GROUP BY → HAVING → SELECT → ORDER BY*

SELECT u.name, s.genre,	<i>-- Step 5: project columns</i>
COUNT (*) AS song_count,	
AVG (l.rating) AS avg_rating	
FROM Listens l	<i>-- Step 1a: read Listens</i>
JOIN Songs s ON l.song_id = s.song_id	<i>-- Step 1b: join Songs</i>
JOIN Users u ON u.user_id = l.user_id	<i>-- Step 1c: join Users for the name</i>
WHERE s.genre IS NOT NULL	<i>-- Step 2: drop NULL genres</i>
GROUP BY u.user_id, u.name, s.genre	<i>-- Step 3: form groups</i>
HAVING COUNT (*) >= 2	<i>-- Step 4: keep groups with 2+ listens</i>
ORDER BY u.user_id, avg_rating DESC ;	<i>-- Step 6: sort output</i>

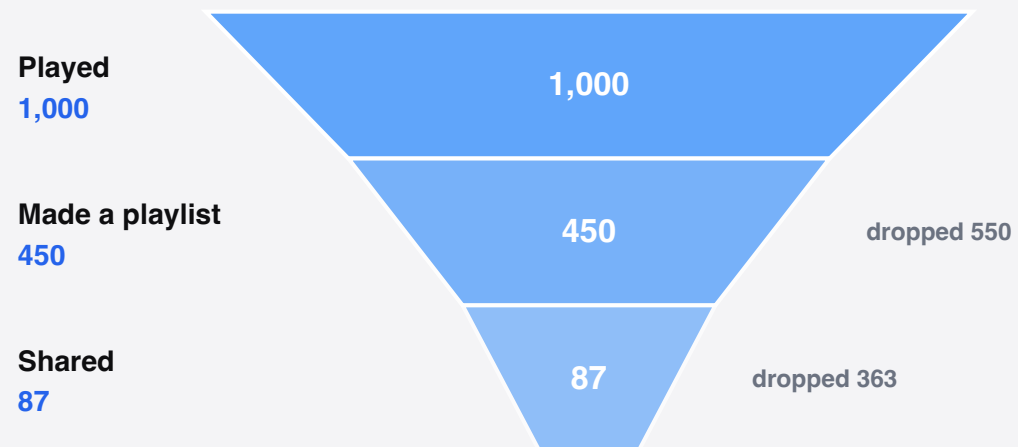
M1

SQL Problem Solving: Writing Queries

Most queries you write are one of five shapes: filter to a subset, rank within groups, accumulate over time, compare to a group, or flag outliers.

The Funnel: who did A but not B

Keep listeners outside the shares set; NOT IN assumes no NULL share user IDs.

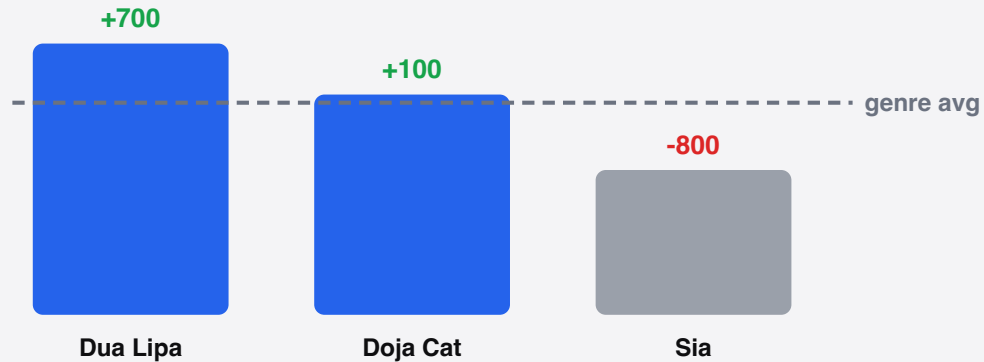


the query returns the 913 who played but never shared

```
SELECT user_id, name
FROM users
-- Keep users who have listened.
WHERE user_id IN
      (SELECT user_id FROM listens)
-- Exclude users who have shared.
-- Assumes shares.user_id contains no NULLs.
AND user_id NOT IN
      (SELECT user_id FROM shares)
```

The Comparison: a row against its group average

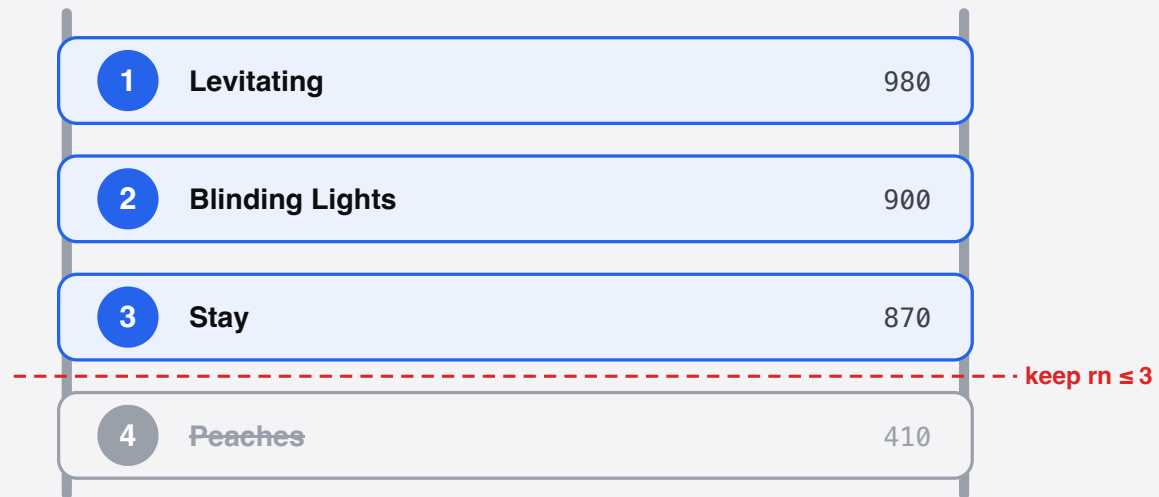
Subtract the group's AVG, computed over the partition, from each row.



```
SELECT artist_name, genre, plays,  
       -- Subtract this genre's average.  
       plays - AVG(plays) OVER (  
         PARTITION BY genre  
       ) AS vs_avg  
       -- Positive: above average; negative: below.  
FROM artist_stats
```

The Ladder: top N within each group

ROW_NUMBER restarts per genre; the outer WHERE keeps rows 1 through 3.

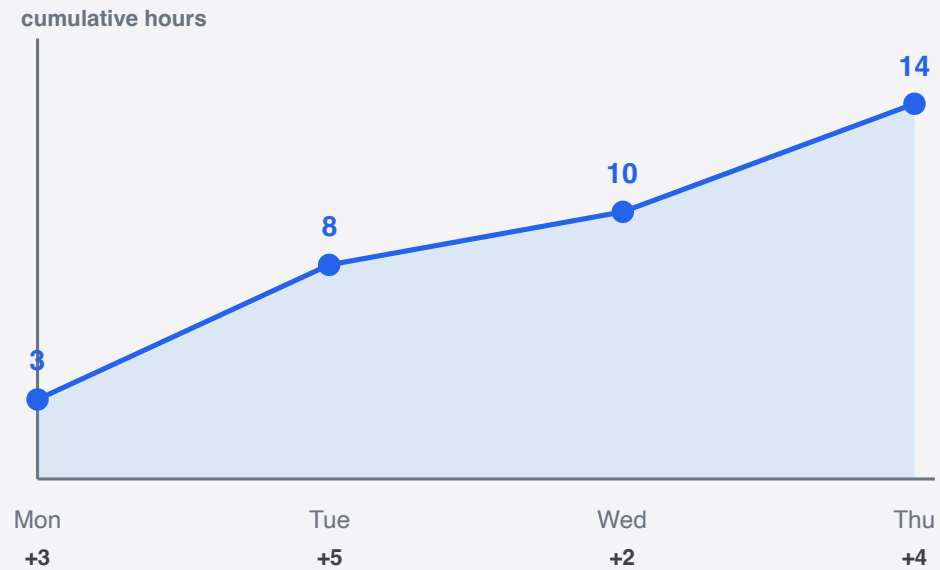


1	Levitating	980
2	Blinding Lights	900
3	Stay	870
4	Peaches	410

```
SELECT * FROM (  
  SELECT song, genre, plays,  
    ROW_NUMBER() OVER (  
      -- Number songs separately in each genre.  
      PARTITION BY genre  
      -- Most plays first.  
      ORDER BY plays DESC  
    ) AS rn  
  FROM songs  
) ranked  
-- Keep up to three songs per genre.  
WHERE rn <= 3
```

The Timeline: a running total over time

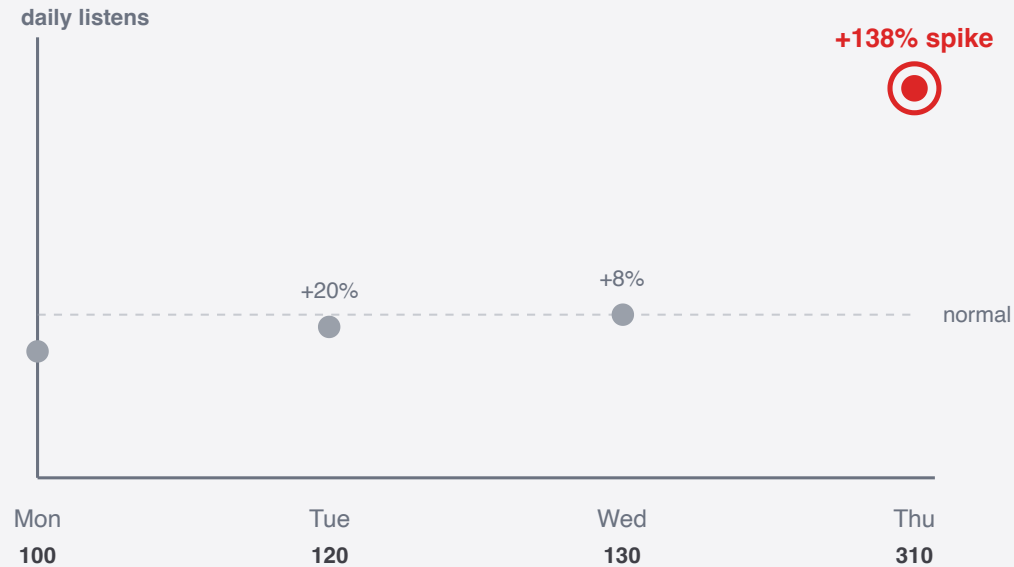
Keep one row per day; SUM adds the hours up through that date.



```
SELECT date, hours,  
       SUM(hours) OVER (  
         -- Add hours from earlier dates and today.  
         ORDER BY date  
       ) AS cumulative_hours  
FROM daily_listening
```

Optional: find the outlier with LAG

LAG reads the previous row for each song, in date order; the first has no previous value.



```
WITH changes AS (  
    SELECT song_id, date, listens,  
           -- Previous count for this song.  
           LAG(listens) OVER (  
               PARTITION BY song_id ORDER BY date  
           ) AS prev_listens  
    FROM daily_listens  
)  
with_pct AS (  
    SELECT *,  
           -- Percentage change; NULL for a zero baseline.  
           (listens - prev_listens) * 100.0  
           / NULLIF(prev_listens, 0) AS pct_change  
    FROM changes  
)  
SELECT * FROM with_pct  
-- Keep increases over 50%.  
WHERE pct_change > 50
```