

M1

Spotify Database

The course runs on three tables: Users, Songs, and Listens.

One schema: three tables, two keys tie them together

Every query in the course is a question over these. The IDs are how they connect.

Users (4 rows)

user_id	PK	name	email
1		Mickey	mickey@ex.com
2		Minnie	minnie@ex.com
3		Daffy	daffy@ex.com
4		Pluto	pluto@ex.com

PK = primary key

user_id is one unique value per user, like a student ID. No two users share it.

Songs (10 rows)

song_id	PK	title	artist	genre
1		Evermore	Taylor Swift	Pop
2		Willow	Taylor Swift	Pop
3		Shape of You	Ed Sheeran	Rock
4		Photograph	Ed Sheeran	Rock
5		Shivers	Ed Sheeran	Rock
6		Yesterday	Beatles	Classic
7		Yellow Sub	Beatles	Classic
8		Hey Jude	Beatles	Classic
9		Bad Blood	Taylor Swift	Rock
10		DJ Mix	DJ	NULL

user_id = Users.user_id

song_id = Songs.song_id

Listens (9 rows)

listen_id	PK	user_id	FK	song_id	FK	rating	listen_time
1		1		1		4.5	2024-08-30
2		1		2		4.2	NULL
3		1		6		3.9	2024-08-29
4		2		2		4.7	NULL
5		2		7		4.6	2024-08-28
6		2		8		3.9	2024-08-27
7		3		1		2.9	NULL
8		3		2		4.9	2024-08-26
9		3		6		NULL	NULL

FK = foreign key

user_id and song_id are foreign keys. Each must match a real id, so a listen can't point at nobody.

Reading one row

listen 8: user_id 3, song_id 2, rating 4.9

Daffy played Willow, and rated it 4.9.

Reading the DDL: how the Listens table is defined

DDL, the Data Definition Language: CREATE TABLE names each column, types it, and states its rules.

```
CREATE TABLE listens (  
  listen_id    INT          PRIMARY KEY,  
  user_id     INT          NOT NULL,  
  song_id     INT          NOT NULL,  
  rating      DECIMAL(2,1),  
  listen_time  TIMESTAMP,  
  FOREIGN KEY (user_id) REFERENCES users(user_id),  
  FOREIGN KEY (song_id) REFERENCES songs(song_id)  
);
```

PRIMARY KEY

listen_id is the unique id for the row.
NOT NULL = required; rating / listen_time omit it, so those may be NULL.

the column's type

DECIMAL(2,1) holds a rating like 4.5.
INT, TIMESTAMP - every column names one.

FOREIGN KEY ... REFERENCES

every user_id must be a real Users row,
every song_id a real Songs row. The DB rejects a listen that points at nobody.

Users and Songs are defined the same way, just without the foreign keys.

Teaching Team



Shiva
Instructor · CS145 '94 · Teaching since 2018



Jason Hu
Head CA · CS145 '24



Ananya Ganapathi



Aryan Katiyar



Ganesh Venu
CS145 '25



Jason Shin



Owen Jung
CS145 '25



Qirui Zheng



Yixiao Zhang
CS145 '25

M1

SELECT-FROM-WHERE: The SQL Foundation

A basic query has three clauses.

SELECT-FROM-WHERE: keep some rows, then some columns

Execution order: FROM loads rows, WHERE filters rows, SELECT picks columns.

FROM Listens

9 rows

listen_id	user_id	song_id	rating
1	1	1	4.5
2	1	2	4.2
3	1	6	3.9
4	2	2	4.7
5	2	7	4.6
6	2	8	3.9
7	3	1	2.9
8	3	2	4.9
9	3	6	NULL

keep 5 rows,
→
then 3 columns

WHERE rating > 4.0

SELECT user_id, song_id, rating

5 rows

user_id	song_id	rating
1	1	4.5
1	2	4.2
2	2	4.7
2	7	4.6
3	2	4.9

rows survive WHERE, then SELECT drops listen_id

WHERE with AND and OR: same two conditions, opposite strictness

Keep a row only when BOTH hold

```
SELECT user_id, rating
FROM Listens
WHERE user_id = 1 AND rating > 4.0
```

user_id	rating	
1	4.5	✓ keep
1	4.2	✓ keep
1	3.9	fails WHERE
2	4.7	fails WHERE
2	4.6	fails WHERE
2	3.9	fails WHERE
3	2.9	fails WHERE
3	4.9	fails WHERE
3	NULL	fails WHERE

AND keeps 2 rows

Keep a row when EITHER holds

```
SELECT user_id, rating
FROM Listens
WHERE user_id = 1 OR rating > 4.0
```

user_id	rating	
1	4.5	✓ keep
1	4.2	✓ keep
1	3.9	✓ keep
2	4.7	✓ keep
2	4.6	✓ keep
2	3.9	fails WHERE
3	2.9	fails WHERE
3	4.9	✓ keep
3	NULL	fails WHERE

OR keeps 6 rows

Aliases (AS): rename a column, but WHERE can't use the new name

AS renames the output column

rating	rating * 2 (= double_rating)
4.5	9.0
4.2	8.4
3.9	7.8

FAILS

```
SELECT rating * 2 AS double_rating  
FROM Listens  
WHERE double_rating > 8
```

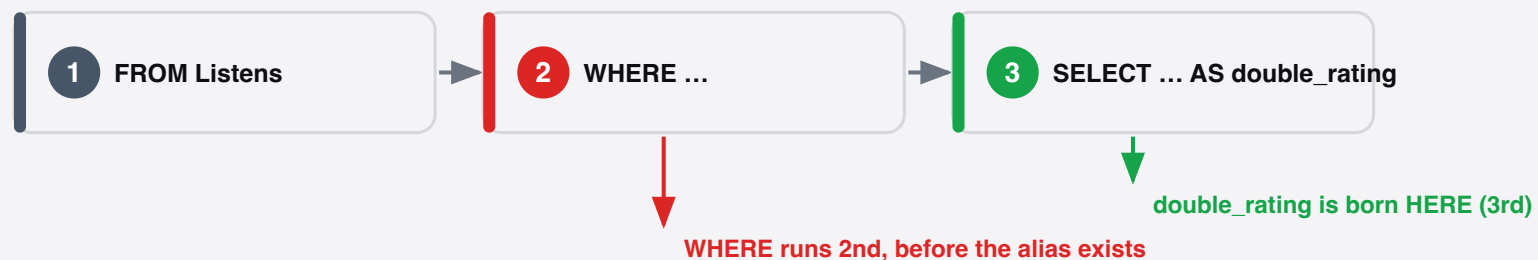
✗ ERROR: no column named double_rating yet

WORKS

```
SELECT rating * 2 AS double_rating  
FROM Listens  
WHERE rating * 2 > 8
```

✓ WHERE uses the expression, which already exists

Execution order: the alias is created last



M1

NULL Values

NULL means "unknown," not 0 or empty.

NULL adds a third truth value: UNKNOWN

NULL is not zero or empty. It means unknown, and comparisons propagate it.

Binary logic: every test is TRUE or FALSE

expression	result
4.5 > 4.0	✓ TRUE
4.5 > 5.0	✗ FALSE
4.5 > NULL	? UNKNOWN
NULL = NULL	? UNKNOWN
rating = NULL	? UNKNOWN

Add NULL, a missing value: every comparison with it is UNKNOWN, a third truth value.

WHERE keeps only TRUE rows.

FALSE and UNKNOWN rows drop silently.

Use rating IS NULL to find NULL rows.

Two-valued logic: every test is TRUE or FALSE

The refresher before NULL: AND needs both, OR needs one, NOT flips.

a AND b

a	b	AND
T	T	T
T	F	F
F	T	F
F	F	F

a OR b

a	b	OR
T	T	T
T	F	T
F	T	T
F	F	F

NOT a

a	NOT
T	F
F	T

How AND, OR, and NOT handle UNKNOWN

FALSE decides AND. TRUE decides OR. Otherwise, UNKNOWN remains.

Apply the complete expression

expression	result
FALSE AND NULL	FALSE
TRUE AND NULL	NULL
TRUE OR NULL	TRUE
FALSE OR NULL	NULL
NOT NULL	NULL

— false wins

— true wins

— NULL stays

Test missingness explicitly

expression	result
rating IS NULL	TRUE / FALSE

— the explicit test

WHERE applies AND/OR to every row

The whole predicate decides each row's fate. NULL changes row 9.

AND: both tests must pass

```
SELECT *  
FROM Listens  
WHERE user_id = 3  
      AND rating > 4
```

listen_id	user_id	rating	predicate
~~1~~	1	4.5	~~FALSE~~
~~2~~	1	4.2	~~FALSE~~
~~3~~	1	3.9	~~FALSE~~
~~4~~	2	4.7	~~FALSE~~
~~5~~	2	4.6	~~FALSE~~
~~6~~	2	3.9	~~FALSE~~
~~7~~	3	2.9	~~FALSE~~
8	3	4.9	TRUE
~~9~~	3	NULL	~~UNKNOWN~~

WHERE keeps 1 of 9 rows

OR: either test can pass

```
SELECT *  
FROM Listens  
WHERE user_id = 3  
      OR rating > 4
```

listen_id	user_id	rating	predicate
1	1	4.5	TRUE
2	1	4.2	TRUE
~~3~~	1	3.9	~~FALSE~~
4	2	4.7	TRUE
5	2	4.6	TRUE
~~6~~	2	3.9	~~FALSE~~
7	3	2.9	TRUE
8	3	4.9	TRUE
9	3	NULL	TRUE

WHERE keeps 7 of 9 rows

IS NULL finds the NULL row; comparisons drop it

Query 1

```
SELECT listen_id, rating
FROM Listens
WHERE rating IS NULL
```

listen_id	rating
9	NULL

1 row: IS NULL keeps it

Query 2

```
SELECT listen_id, rating
FROM Listens
WHERE rating > 4.0
```

listen_id	rating
1	4.5
2	4.2
4	4.7
5	4.6
8	4.9

5 rows: NULL dropped

Query 3

```
SELECT listen_id, rating
FROM Listens
WHERE rating <= 4.0
```

listen_id	rating
3	3.9
6	3.9
7	2.9

3 rows: NULL dropped again

Row 9 is NULL. Only IS NULL returns TRUE; > and <= return UNKNOWN, which WHERE drops.

COUNT(*) counts every row; COUNT(rating) and AVG skip the NULL

Listens.rating

rating
4.5
4.2
3.9
4.7
4.6
3.9
2.9
4.9
NULL

COUNT(*)

counts every row, including the NULL

9**COUNT(rating)**

skips the NULL: 8 ratings, not 9 rows

8**AVG(rating)** $33.6 \div 8 = 4.2$ (8 ratings, not 9 rows)**4.2**

M1

GROUP BY: Grouping and Aggregating Data

GROUP BY puts rows with the same values in the named columns into one group and produces one output row per group.

GROUP BY collapses a nine-row Listens table into a three-row grouped table

Listens

9 rows

GROUP BY user_id

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

Grouped by user_id

3 rows

user_id	COUNT(*)	AVG
1	3	4.2
2	3	4.4
3	3	3.9

GROUP BY user_id: same user_id falls in one group

Nine listens collapse into three groups, one row each.

```
-- Count each user's listens
SELECT user_id, COUNT(*) AS listen_count
FROM Listens
GROUP BY user_id -- one group per unique user_id
```

Listens (9 rows, grouped by user)

the input

listen_id	user_id	song_id	rating
1	1	1	4.5
2	1	2	4.2
3	1	6	3.9
4	2	2	4.7
5	2	7	4.6
6	2	8	3.9
7	3	1	2.9
8	3	2	4.9
9	3	6	NULL

GROUP BY

Grouped by user_id (3 rows)

one row per group

user_id	listen_count
1	3
2	3
3	3

Pluto: 0 listens, so no group
and no row.

GROUP BY user_id with four aggregates per group: COUNT, AVG, MAX, MIN

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

GROUP BY user_id

Grouped by user_id

3 rows

user_id	COUNT	AVG	MAX	MIN
1	3	4.2	4.5	3.9
2	3	4.4	4.7	3.9
3	3	3.9	4.9	2.9

More columns, finer groups

Mickey's three listens: one group by `user_id`; add `song_id` and the group splits into three.

GROUP BY `user_id`

Mickey <code>user_id = 1</code>	
<code>song_id</code>	<code>rating</code>
1	4.5
2	4.2
6	3.9
1 group <code>COUNT(*) = 3</code>	

+ `song_id`
→

GROUP BY `user_id, song_id`

(1, 1) rating 4.5	<code>COUNT(*) = 1</code>
(1, 2) rating 4.2	<code>COUNT(*) = 1</code>
(1, 6) rating 3.9	<code>COUNT(*) = 1</code>

3 groups

Ask for a column you did not group by, and the row cannot answer

GROUP BY user_id makes one row per user. But song_id had three values in that group.

```
-- Unsafe: song_id is in SELECT but not in GROUP BY
SELECT user_id, song_id, COUNT(*)
FROM Listens
GROUP BY user_id
```

After GROUP BY user_id: one row per user

song_id was never grouped, so it has no single value

user_id	COUNT(*)	song_id
1	3	?
2	3	?
3	3	?

collapsed songs 1, 2, 6 - which one?

collapsed songs 2, 7, 8 - which one?

collapsed songs 1, 2, 6 - which one?

Every column you SELECT must be grouped or aggregated.

M1

HAVING: Filtering Group Aggregates

HAVING filters groups after GROUP BY by evaluating aggregate expressions such as COUNT and AVG.

HAVING filters GROUPS (after GROUP BY)

```
SELECT user_id, COUNT(*), AVG(rating)
FROM Listens
GROUP BY user_id
HAVING AVG(rating) > 4.0
```

GROUP BY user_id -> one row per user

3 groups

user_id	COUNT(*)	AVG(rating)	
1	3	4.2	4.2 > 4.0 keep
2	3	4.4	4.4 > 4.0 keep
3	3	3.9	3.9 > 4.0 fails

Output: Mickey, Minnie (Daffy's 3.9 is below 4.0)

HAVING with COUNT: keep only songs played twice or more

```
-- Find songs played at least twice
SELECT song_id, COUNT(*), AVG(rating)
FROM Listens
GROUP BY song_id      -- Group by each song
HAVING COUNT(*) >= 2  -- Keep only groups with 2+ rows
```

GROUP BY song_id -> one row per song

5 groups

song_id	COUNT(*)	AVG(rating)	
1	2	3.7	2 >= 2 keep
2	3	4.6	3 >= 2 keep
6	2	3.9	2 >= 2 keep
7	1	4.6	1 >= 2 fails
8	1	3.9	1 >= 2 fails

Output: songs 1, 2, 6 (played at least twice)

HAVING with AND: a group must pass BOTH conditions

```
-- Find active users (3+ listens) with high average ratings (>4.0)
SELECT user_id, COUNT(*), AVG(rating), MIN(rating)
FROM Listens
GROUP BY user_id
HAVING COUNT(*) >= 3      -- One HAVING clause, two conditions joined by AND
      AND AVG(rating) > 4.0 -- Both must be true for group to survive
```

GROUP BY user_id -> one row per user

3 groups

user_id	COUNT(*)	AVG(rating)	MIN(rating)
1	3	4.2	3.9
2	3	4.4	3.9
3	3	3.9	2.9

both pass keep

both pass keep

AVG 3.9 not > 4.0 fails

Output: Mickey, Minnie (Daffy's 3.9 average fails the second test)

WHERE filters rows, then HAVING filters groups

```
-- Find users with 2+ ratings above 3.5
SELECT user_id, COUNT(*), AVG(rating)
FROM Listens
WHERE rating > 3.5      -- 1st: WHERE filters individual rows
GROUP BY user_id        -- 2nd: GROUP BY forms groups from survivors
HAVING COUNT(*) >= 2    -- 3rd: HAVING filters entire groups
```

WHERE rating > 3.5

9 -> 7 rows

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

GROUP
→

GROUP BY -> HAVING COUNT(*) >= 2

3 -> 2

user_id	COUNT(*)	AVG(rating)
1	3	4.2
2	3	4.4
3	1	4.9

3 >= 2 keep

3 >= 2 keep

1 >= 2 fails

Output: users 1 and 2 (Mickey, Minnie)

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