

## Part 1 (All AI Prompts using Claude, Sonnet 4.6):

### Prompt 1:

Given the following PostgreSQL schema: create table classroom

```
(building varchar(15),
room_number varchar(7),
capacity numeric(4,0),
primary key (building, room_number)
);
create table department
(dept_name varchar(20),
building varchar(15),
budget numeric(12,2) check (budget > 0),
primary key (dept_name)
);
create table course
(course_id varchar(8),
title varchar(50),
dept_name varchar(20),
credits numeric(2,0) check (credits > 0),
primary key (course_id),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table instructor
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
salary numeric(8,2) check (salary > 29000),
primary key (ID),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table section
(course_id varchar(8),
sec_id varchar(8),
semester varchar(6)
check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
year numeric(4,0) check (year > 1701 and year <
2100),
building varchar(15),
room_number varchar(7),
time_slot_id varchar(4),
primary key (course_id, sec_id, semester, year),
foreign key (course_id) references course (course_id)
on delete cascade,
foreign key (building, room_number) references
classroom (building, room_number)
on delete set null
);
create table teaches
(ID varchar(5),
course_id varchar(8),
sec_id varchar(8),
semester varchar(6),
year numeric(4,0),
primary key (ID, course_id, sec_id, semester, year),
foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
```

```
on delete cascade,
foreign key (ID) references instructor (ID)
on delete cascade
);
create table student
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
tot_cred numeric(3,0) check (tot_cred >= 0),
primary key (ID),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table takes
(ID varchar(5),
course_id varchar(8),
sec_id varchar(8),
semester varchar(6),
year numeric(4,0),
grade varchar(2),
primary key (ID, course_id, sec_id, semester, year),
foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
on delete cascade,
foreign key (ID) references student (ID)
on delete cascade
);
create table advisor
(s_ID varchar(5),
i_ID varchar(5),
primary key (s_ID),
foreign key (i_ID) references instructor (ID)
on delete set null,
foreign key (s_ID) references student (ID)
on delete cascade
);
create table time_slot
(time_slot_id varchar(4),
day varchar(1),
start_hr numeric(2) check (start_hr >= 0 and start_hr <
24),
start_min numeric(2) check (start_min >= 0 and
start_min < 60),
end_hr numeric(2) check (end_hr >= 0 and end_hr <
24),
end_min numeric(2) check (end_min >= 0 and
end_min < 60),
primary key (time_slot_id, day, start_hr, start_min)
);
create table prereq
(course_id varchar(8),
prereq_id varchar(8),
primary key (course_id, prereq_id),
foreign key (course_id) references course (course_id)
on delete cascade,
foreign key (prereq_id) references course (course_id)
);
```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50 instructors, and 20 departments. From this, write an SQL

query to find the total number of course enrollments and the average total credits (tot\_cred) of students in each department. Rank departments from highest enrollment count to lowest.

## Claude AI Response:

```
SELECT
  d.dept_name,
  COUNT(t.ID) AS total_enrollments,
  ROUND(AVG(s.tot_cred), 2) AS avg_tot_cred
FROM department d
LEFT JOIN student s ON s.dept_name = d.dept_name
LEFT JOIN takes t ON t.ID = s.ID
GROUP BY d.dept_name
ORDER BY total_enrollments DESC;
```

How it works:

The query chains three LEFT JOINS so that every department appears in the result, even those with no students or no enrollment records.

- `department` → `student` links each department to its students
- `student` → `takes` links each student to their enrollment rows
- `COUNT(t.ID)` counts non-NULL rows in `takes`, so departments with no enrollments correctly get 0
- `AVG(s.tot_cred)` averages the total credits across all students in that department (NULL if no students exist)
- `ROUND(..., 2)` keeps the average tidy to two decimal places
- `ORDER BY total_enrollments DESC` ranks from most-enrolled to least

Why LEFT JOIN instead of INNER JOIN?

With ~20 departments but potentially sparse enrollment, an INNER JOIN would silently drop any department that has no students or no takes rows. LEFT JOIN ensures every department is represented, which is usually the desired behavior for reporting queries like this.

Performance note:

With 30,000 rows in `takes` and 2,000 students, indexes on `takes (ID)`, `student (dept_name)`, and `student (ID)` will keep this fast. These are already covered by the primary key on `takes (ID, course_id, sec_id, semester, year)` and the primary key on `student (ID)`, but a separate index on `student (dept_name)` would help the join to `department`.

## Execution Plan Results:

"Sort (cost=981.54..981.59 rows=20 width=49) (actual time=24.018..24.025 rows=20.00 loops=1)"

" Sort Key: (count(t.id)) DESC"

" Sort Method: quicksort Memory: 25kB"

" Buffers: shared hit=236"

" -> HashAggregate (cost=980.80..981.10 rows=20 width=49) (actual time=23.980..23.998 rows=20.00 loops=1)"

" Group Key: d.dept\_name"

```

"    Batches: 1 Memory Usage: 32kB"
"    Buffers: shared hit=236"
"    -> Hash Right Join (cost=61.45..755.80 rows=30000 width=18) (actual
time=0.522..16.983 rows=30000.00 loops=1)"
"        Hash Cond: ((s.dept_name)::text = (d.dept_name)::text)"
"        Buffers: shared hit=236"
"        -> Hash Right Join (cost=60.00..658.92 rows=30000 width=18) (actual
time=0.478..10.160 rows=30000.00 loops=1)"
"            Hash Cond: ((t.id)::text = (s.id)::text)"
"            Buffers: shared hit=235"
"            -> Seq Scan on takes t (cost=0.00..520.00 rows=30000 width=5) (actual
time=0.012..1.858 rows=30000.00 loops=1)"
"                Buffers: shared hit=220"
"                -> Hash (cost=35.00..35.00 rows=2000 width=18) (actual time=0.458..0.460
rows=2000.00 loops=1)"
"                    Buckets: 2048 Batches: 1 Memory Usage: 119kB"
"                    Buffers: shared hit=15"
"                    -> Seq Scan on student s (cost=0.00..35.00 rows=2000 width=18) (actual
time=0.009..0.205 rows=2000.00 loops=1)"
"                        Buffers: shared hit=15"
"                        -> Hash (cost=1.20..1.20 rows=20 width=9) (actual time=0.030..0.030 rows=20.00
loops=1)"
"                            Buckets: 1024 Batches: 1 Memory Usage: 9kB"
"                            Buffers: shared hit=1"
"                            -> Seq Scan on department d (cost=0.00..1.20 rows=20 width=9) (actual
time=0.020..0.022 rows=20.00 loops=1)"
"                                Buffers: shared hit=1"
"Planning:"
" Buffers: shared hit=22"
"Planning Time: 0.856 ms"
"Execution Time: 24.107 ms"

```

## New Query:

```

SELECT
    s.dept_name,
    COUNT(t.ID)                AS total_enrollments,
    AVG(s.tot_cred)            AS avg_tot_cred
FROM student s
LEFT JOIN takes t ON t.ID = s.ID
GROUP BY s.dept_name

```

```
ORDER BY total_enrollments DESC;
```

## New Execution Plan:

```
"Sort (cost=884.60..884.65 rows=20 width=49) (actual time=15.549..15.554 rows=20.00
loops=1)"
" Sort Key: (count(t.id)) DESC"
" Sort Method: quicksort Memory: 26kB"
" Buffers: shared hit=235"
" -> HashAggregate (cost=883.92..884.17 rows=20 width=49) (actual time=15.523..15.534
rows=20.00 loops=1)"
"   Group Key: s.dept_name"
"   Batches: 1 Memory Usage: 32kB"
"   Buffers: shared hit=235"
"   -> Hash Join (cost=60.00..658.92 rows=30000 width=18) (actual time=0.496..9.012
rows=30000.00 loops=1)"
"     Hash Cond: ((t.id)::text = (s.id)::text)"
"     Buffers: shared hit=235"
"     -> Seq Scan on takes t (cost=0.00..520.00 rows=30000 width=5) (actual
time=0.026..1.575 rows=30000.00 loops=1)"
"       Buffers: shared hit=220"
"       -> Hash (cost=35.00..35.00 rows=2000 width=18) (actual time=0.458..0.460
rows=2000.00 loops=1)"
"         Buckets: 2048 Batches: 1 Memory Usage: 119kB"
"         Buffers: shared hit=15"
"         -> Seq Scan on student s (cost=0.00..35.00 rows=2000 width=18) (actual
time=0.008..0.212 rows=2000.00 loops=1)"
"           Buffers: shared hit=15"
"Planning:"
" Buffers: shared hit=12"
"Planning Time: 0.495 ms"
"Execution Time: 15.628 ms"
```

## Optimization:

### AI Notes:

- Round() was used, although not requested.
- Because of the presence of a 'department' table, this was used despite requiring additional joins.

### Optimizations Made:

- Removed additional join with department table.

- Removed unnecessary rounding.

Metric	Before	After	Change
Scan Type	Seq Scan	Seq Scan	No Change
Rows Scanned	30,000, 3 scans	30,000, 2 scans	No Change, -33% fewer scans
Buffer Pages Read	22	12	-45%
Execution Time	24.107 ms	15.628 ms	-35%

## Prompt 2:

Given the following PostgreSQL schema: create table classroom

```
(building varchar(15),
 room_number varchar(7),
 capacity numeric(4,0),
 primary key (building, room_number)
);
create table department
(dept_name varchar(20),
 building varchar(15),
 budget numeric(12,2) check (budget > 0),
 primary key (dept_name)
);
create table course
(course_id varchar(8),
 title varchar(50),
 dept_name varchar(20),
 credits numeric(2,0) check (credits > 0),
 primary key (course_id),
 foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table instructor
(ID varchar(5),
 name varchar(20) not null,
 dept_name varchar(20),
 salary numeric(8,2) check (salary > 29000),
 primary key (ID),
 foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table section
(course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6)
check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
 year numeric(4,0) check (year > 1701 and year <
2100),
 building varchar(15),
```

```
room_number varchar(7),
 time_slot_id varchar(4),
 primary key (course_id, sec_id, semester, year),
 foreign key (course_id) references course (course_id)
on delete cascade,
 foreign key (building, room_number) references
classroom (building, room_number)
on delete set null
);
create table teaches
(ID varchar(5),
 course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6),
 year numeric(4,0),
 primary key (ID, course_id, sec_id, semester, year),
 foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
on delete cascade,
 foreign key (ID) references instructor (ID)
on delete cascade
);
create table student
(ID varchar(5),
 name varchar(20) not null,
 dept_name varchar(20),
 tot_cred numeric(3,0) check (tot_cred >= 0),
 primary key (ID),
 foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table takes
(ID varchar(5),
 course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6),
 year numeric(4,0),
 grade varchar(2),
 primary key (ID, course_id, sec_id, semester, year),
```

```

foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
on delete cascade,
foreign key (ID) references student (ID)
on delete cascade
);
create table advisor
(s_ID varchar(5),
i_ID varchar(5),
primary key (s_ID),
foreign key (i_ID) references instructor (ID)
on delete set null,
foreign key (s_ID) references student (ID)
on delete cascade
);
create table time_slot
(time_slot_id varchar(4),
day varchar(1),
start_hr numeric(2) check (start_hr >= 0 and start_hr <
24),
start_min numeric(2) check (start_min >= 0 and
start_min < 60),

```

```

end_hr numeric(2) check (end_hr >= 0 and end_hr <
24),
end_min numeric(2) check (end_min >= 0 and
end_min < 60),
primary key (time_slot_id, day, start_hr, start_min)
);
create table prereq
(course_id varchar(8),
prereq_id varchar(8),
primary key (course_id, prereq_id),
foreign key (course_id) references course (course_id)
on delete cascade,
foreign key (prereq_id) references course (course_id)
);

```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50 instructors, and 20 departments. From this, write an SQL query that for each instructor, lists their name, department, the courses they teach, the semester and year, and the number of students enrolled in each of their sections. Include only sections with at least 10 students.

## Claude AI Response:

```

SELECT
    i.name AS instructor_name,
    i.dept_name,
    c.title AS course_title,
    te.course_id,
    te.sec_id,
    te.semester,
    te.year,
    COUNT (ta.ID) AS students_enrolled
FROM instructor i
JOIN teaches te ON te.ID = i.ID
JOIN course c ON c.course_id = te.course_id
JOIN takes ta ON ta.course_id = te.course_id
AND ta.sec_id = te.sec_id
AND ta.semester = te.semester
AND ta.year = te.year

GROUP BY
    i.ID,
    i.name,
    i.dept_name,
    c.title,
    te.course_id,
    te.sec_id,
    te.semester,
    te.year
HAVING COUNT (ta.ID) >= 10
ORDER BY
    i.name,
    te.year DESC,
    te.semester,
    te.course_id;

```

How it works:

The query builds up in four logical layers:

Joins

- `instructor` → `teaches` maps each instructor to the sections they teach
- `teaches` → `course` pulls in the human-readable course title
- `teaches` → `takes` (on all four composite key columns) matches every student enrollment row to the correct section — joining on all four columns is critical here, since `(course_id, sec_id, semester, year)` together form the section's composite primary key; matching on fewer columns would fan out rows incorrectly

### Grouping

`GROUP BY` includes `i.ID` in addition to `i.name` to correctly handle two instructors who might share the same name. All non-aggregated `SELECT` columns must appear here.

### HAVING vs WHERE

`HAVING COUNT(ta.ID) >= 10` filters *after* aggregation, which is required when the filter condition depends on an aggregate value. A `WHERE` clause cannot reference `COUNT(ta.ID)` because row-level filtering happens before grouping.

### Why INNER JOINS here (not LEFT JOINS)

Unlike a department reporting query where you want zero-enrollment rows, the requirement here is specifically sections *with enrollments*, so INNER JOINS are correct and also more efficient — the planner can prune non-matching rows earlier.

### Ordering

Results are sorted by instructor name, then most recent year first, then semester, then course — a natural reading order for a teaching schedule report.

### Performance note:

The most important index for this query is a composite one on `takes(course_id, sec_id, semester, year, ID)`, which covers the four-column join and lets `COUNT(ta.ID)` be resolved from the index alone. The existing primary key on `takes` already starts with `ID`, so a separate index with `course_id` leading would complement it well at this data scale.

## Execution Plan:

```
"Sort (cost=1015.89..1016.39 rows=198 width=63) (actual time=46.275..46.292
  rows=100.00 loops=1)"
"  Sort Key: i.name, te.year DESC, te.semester, te.course_id"
"  Sort Method: quicksort  Memory: 33kB"
"  Buffers: shared hit=224"
" -> HashAggregate (cost=1000.92..1008.34 rows=198 width=63) (actual
  time=45.899..45.950 rows=100.00 loops=1)"
"   Group Key: te.year, te.semester, te.course_id, i.id, c.title, te.sec_id"
"   Filter: (count(ta.id) >= 10)"
"   Batches: 1  Memory Usage: 57kB"
"   Buffers: shared hit=224"
" -> Hash Join (cost=12.89..990.52 rows=594 width=60) (actual time=0.258..27.837
  rows=30000.00 loops=1)"
"   Hash Cond: ((te.id)::text = (i.id)::text)"
"   Buffers: shared hit=224"
" -> Hash Join (cost=10.76..986.70 rows=594 width=43) (actual
  time=0.196..19.519 rows=30000.00 loops=1)"
```



```

c.title                               AS course_title,
te.course_id,
te.sec_id,
te.semester,
te.year,
COUNT (ta.ID)                       AS students_enrolled
FROM instructor i
JOIN teaches te                       ON te.ID = i.ID
JOIN course c                         ON c.course_id = te.course_id
JOIN takes ta                         ON ta.course_id = te.course_id
                                     AND ta.sec_id = te.sec_id
                                     AND ta.semester = te.semester
                                     AND ta.year = te.year

GROUP BY
i.ID,
i.name,
i.dept_name,
c.title,
te.course_id,
te.sec_id,
te.semester,
te.year
HAVING COUNT (ta.ID) >= 10
ORDER BY
i.name

```

## New Execution Plan:

```

"Sort (cost=1116.43..1116.92 rows=198 width=63) (actual time=43.317..43.322
  rows=100.00 loops=1)"
"  Sort Key: i.name"
"  Sort Method: quicksort  Memory: 33kB"
"  Buffers: shared hit=16890"
"-> HashAggregate (cost=1101.45..1108.87 rows=198 width=63) (actual
  time=43.204..43.231 rows=100.00 loops=1)"
"  Group Key: i.id, c.title, te.course_id, te.sec_id, te.semester, te.year"
"  Filter: (count(ta.id) >= 10)"
"  Batches: 1  Memory Usage: 57kB"
"  Buffers: shared hit=16890"
"-> Nested Loop (cost=0.83..1091.05 rows=594 width=60) (actual time=0.079..29.097
  rows=30000.00 loops=1)"
"  Join Filter: ((c.course_id)::text = (ta.course_id)::text)"
"  Buffers: shared hit=16890"
"  -> Merge Join (cost=0.53..45.71 rows=100 width=59) (actual time=0.035..0.584
  rows=100.00 loops=1)"
"    Merge Cond: ((te.course_id)::text = (c.course_id)::text)"
"    Buffers: shared hit=160"

```

```
"      -> Nested Loop (cost=0.29..24.82 rows=100 width=38) (actual
time=0.026..0.305 rows=100.00 loops=1)"
"          Buffers: shared hit=64"
"      -> Index Scan using idx_teaches_section on teaches te (cost=0.14..13.64
rows=100 width=21) (actual time=0.013..0.044 rows=100.00 loops=1)"
"          Index Searches: 1"
"          Buffers: shared hit=2"
"      -> Memoize (cost=0.15..0.29 rows=1 width=22) (actual time=0.002..0.002
rows=1.00 loops=100)"
"          Cache Key: te.id"
"          Cache Mode: logical"
"          Hits: 69 Misses: 31 Evictions: 0 Overflows: 0 Memory Usage: 4kB"
"          Buffers: shared hit=62"
"      -> Index Scan using instructor_pkey on instructor i (cost=0.14..0.28
rows=1 width=22) (actual time=0.004..0.004 rows=1.00 loops=31)"
"          Index Cond: ((id)::text = (te.id)::text)"
"          Index Searches: 31"
"          Buffers: shared hit=62"
"      -> Index Scan using course_pkey on course c (cost=0.14..19.14 rows=200
width=21) (actual time=0.006..0.090 rows=199.00 loops=1)"
"          Index Searches: 1"
"          Buffers: shared hit=96"
"      -> Memoize (cost=0.30..12.17 rows=6 width=21) (actual time=0.022..0.237
rows=300.00 loops=100)"
"          Cache Key: te.course_id, te.sec_id, te.semester, te.year"
"          Cache Mode: logical"
"          Hits: 0 Misses: 100 Evictions: 0 Overflows: 0 Memory Usage: 1616kB"
"          Buffers: shared hit=16730"
"      -> Index Scan using idx_takes_section on takes ta (cost=0.29..12.16 rows=6
width=21) (actual time=0.021..0.164 rows=300.00 loops=100)"
"          Index Cond: (((course_id)::text = (te.course_id)::text) AND ((sec_id)::text =
(te.sec_id)::text) AND ((semester)::text = (te.semester)::text) AND (year = te.year))"
"          Index Searches: 100"
"          Buffers: shared hit=16730"
"Planning:"
" Buffers: shared hit=24"
"Planning Time: 0.955 ms"
"Execution Time: 43.773 ms"
```

## Optimization:

### AI Notes:

- Correctly uses a Join with GroupBy and Having.
- Joins across all four columns for sections between takes and teaches
- Sorts unnecessarily using Order.

### Optimizations Made:

- Created Index: CREATE INDEX idx\_takes\_section ON takes(course\_id, sec\_id, semester, year) and for teaches CREATE INDEX idx\_teaches\_section ON teaches(course\_id, sec\_id, semester, year)
- Removed unnecessary ordering that is now resolved from index.

Metric	Before	After	Change
Scan Type	Seq Scan	Index Scan	Changed
Buffer Pages Read	24	24	No Change
Execution Time	46.433 ms	43.733 ms	-6%

## Prompt 3:

```
Given the following PostgreSQL schema: create table
classroom
(building varchar(15),
 room_number varchar(7),
 capacity numeric(4,0),
 primary key (building, room_number)
);
create table department
(dept_name varchar(20),
 building varchar(15),
 budget numeric(12,2) check (budget > 0),
 primary key (dept_name)
);
create table course
(course_id varchar(8),
 title varchar(50),
 dept_name varchar(20),
 credits numeric(2,0) check (credits > 0),
 primary key (course_id),
 foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table instructor
(ID varchar(5),
 name varchar(20) not null,
 dept_name varchar(20),
 salary numeric(8,2) check (salary > 29000),
 primary key (ID),
```

```
foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table section
(course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6)
check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
 year numeric(4,0) check (year > 1701 and year <
2100),
 building varchar(15),
 room_number varchar(7),
 time_slot_id varchar(4),
 primary key (course_id, sec_id, semester, year),
 foreign key (course_id) references course (course_id)
on delete cascade,
 foreign key (building, room_number) references
classroom (building, room_number)
on delete set null
);
create table teaches
(ID varchar(5),
 course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6),
 year numeric(4,0),
 primary key (ID, course_id, sec_id, semester, year),
 foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
```

```

        on delete cascade,
        foreign key (ID) references instructor (ID)
        on delete cascade
    );
create table student
(ID      varchar(5),
 name    varchar(20) not null,
 dept_name  varchar(20),
 tot_cred  numeric(3,0) check (tot_cred >= 0),
 primary key (ID),
 foreign key (dept_name) references department
(dept_name)
        on delete set null
    );
create table takes
(ID      varchar(5),
 course_id  varchar(8),
 sec_id    varchar(8),
 semester  varchar(6),
 year      numeric(4,0),
 grade     varchar(2),
 primary key (ID, course_id, sec_id, semester, year),
 foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
        on delete cascade,
 foreign key (ID) references student (ID)
        on delete cascade
    );
create table advisor
(s_ID     varchar(5),
 i_ID     varchar(5),

```

```

        primary key (s_ID),
        foreign key (i_ID) references instructor (ID)
        on delete set null,
        foreign key (s_ID) references student (ID)
        on delete cascade
    );
create table time_slot
(time_slot_id  varchar(4),
 day          varchar(1),
 start_hr     numeric(2) check (start_hr >= 0 and start_hr <
24),
 start_min    numeric(2) check (start_min >= 0 and
start_min < 60),
 end_hr      numeric(2) check (end_hr >= 0 and end_hr <
24),
 end_min     numeric(2) check (end_min >= 0 and
end_min < 60),
 primary key (time_slot_id, day, start_hr, start_min)
    );
create table prereq
(course_id  varchar(8),
 prereq_id  varchar(8),
 primary key (course_id, prereq_id),
 foreign key (course_id) references course (course_id)
        on delete cascade,
 foreign key (prereq_id) references course (course_id)
    );

```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50 instructors, and 20 departments. From this, write an SQL query to find all courses where every section ever offered had at least one student who received an 'A' grade.

## Claude AI Response:

This is a classic "for all" problem in SQL, which has no direct universal quantifier — instead it's solved by double negation: *"courses where there does NOT exist a section that has NO 'A' grade student."*

sql

```

SELECT
    c.course_id,
    c.title,
    c.dept_name
FROM course c
WHERE
    -- Course must have been offered at least once
    EXISTS (
        SELECT 1
        FROM section s
        WHERE s.course_id = c.course_id
    )
    AND NOT EXISTS (
        SELECT 1
        FROM section s
        WHERE s.course_id = c.course_id
            AND NOT EXISTS (
                SELECT 1
                FROM takes ta
                WHERE ta.course_id = s.course_id
            )
    )

```

```

                AND ta.sec_id      = s.sec_id
                AND ta.semester   = s.semester
                AND ta.year       = s.year
                AND ta.grade      = 'A'
            )
        )
    ORDER BY
        c.dept_name,
        c.course_id;

```

The logic, layer by layer:

Outer **EXISTS** — guards against courses that exist in the `course` table but were never actually scheduled as a section. Without this, a course with no sections at all would vacuously satisfy the **NOT EXISTS** condition and appear in results, which would be misleading.

Outer **NOT EXISTS** — eliminates any course for which we can find even one offending section. It asks: "does a bad section exist?" If yes, the course is excluded.

Inner **NOT EXISTS** — defines what makes a section "bad": it has no row in `takes` where `grade = 'A'`. The four-column join (`course_id, sec_id, semester, year`) is necessary to pinpoint the exact section, since the same course can have multiple sections across different semesters and years.

The full double-negation chain reads as:

```

Keep course C where:
    C was offered at least once
    AND there is no section S of C
    such that no student in S received an 'A'

```

Why not use **GROUP BY / HAVING** here?

A **HAVING MIN(...)** `>= threshold` approach works well for counting problems, but the predicate here is existential per section ("at least one A exists"), not a simple aggregate. Trying to flatten this into a single **GROUP BY** would conflate sections with each other and make it impossible to check the condition independently per section. The correlated **NOT EXISTS** nesting is the cleaner and more correct fit.

A note on `grade = 'A'`

If your grading scheme includes 'A+' or 'A-' and those should also qualify, replace the equality with:

```

sql
AND ta.grade LIKE 'A%'

```

or use an explicit list:

```

sql
AND ta.grade IN ('A+', 'A', 'A-')

```

Performance note

With 30,000 rows in `takes` and 100 sections, the correlated subqueries are evaluated per course (200 iterations) and per section. An index on `takes(course_id, sec_id, semester, year, grade)` lets the innermost **NOT EXISTS** resolve with an index-only scan, keeping this efficient despite the nesting.

## Execution Plan:

```
"QUERY PLAN"
```

```
"Sort (cost=614.08..614.20 rows=49 width=30) (actual time=1.396..1.397 rows=0.00
  loops=1)"
```

```
" Sort Key: c.dept_name, c.course_id"
```

```
" Sort Method: quicksort Memory: 25kB"
```

```
" Buffers: shared hit=226"
```

```
" -> Hash Semi Join (cost=10.75..612.70 rows=49 width=30) (actual time=1.389..1.390
  rows=0.00 loops=1)"
```

```

" Hash Cond: ((c.course_id)::text = (s.course_id)::text)"
" Buffers: shared hit=226"
" -> Hash Right Anti Join (cost=6.50..607.48 rows=115 width=30) (actual
time=1.340..1.350 rows=115.00 loops=1)"
" Hash Cond: ((s_1.course_id)::text = (c.course_id)::text)"
" Buffers: shared hit=224"
" -> Nested Loop Anti Join (cost=0.00..600.25 rows=100 width=4) (actual
time=1.279..1.293 rows=100.00 loops=1)"
" Join Filter: (((ta.course_id)::text = (s_1.course_id)::text) AND ((ta.sec_id)::text
= (s_1.sec_id)::text) AND ((ta.semester)::text = (s_1.semester)::text) AND (ta.year =
s_1.year))"
" Buffers: shared hit=222"
" -> Seq Scan on section s_1 (cost=0.00..3.00 rows=100 width=16) (actual
time=0.003..0.004 rows=100.00 loops=1)"
" Buffers: shared hit=2"
" -> Materialize (cost=0.00..595.00 rows=1 width=16) (actual
time=0.013..0.013 rows=0.00 loops=100)"
" Storage: Memory Maximum Storage: 17kB"
" Buffers: shared hit=220"
" -> Seq Scan on takes ta (cost=0.00..595.00 rows=1 width=16) (actual
time=1.273..1.273 rows=0.00 loops=1)"
" Filter: ((grade)::text = 'A'::text)"
" Rows Removed by Filter: 30000"
" Buffers: shared hit=220"
" -> Hash (cost=4.00..4.00 rows=200 width=30) (actual time=0.036..0.036
rows=200.00 loops=1)"
" Buckets: 1024 Batches: 1 Memory Usage: 21kB"
" Buffers: shared hit=2"
" -> Seq Scan on course c (cost=0.00..4.00 rows=200 width=30) (actual
time=0.008..0.019 rows=200.00 loops=1)"
" Buffers: shared hit=2"
" -> Hash (cost=3.00..3.00 rows=100 width=4) (actual time=0.032..0.032 rows=100.00
loops=1)"
" Buckets: 1024 Batches: 1 Memory Usage: 12kB"
" Buffers: shared hit=2"
" -> Seq Scan on section s (cost=0.00..3.00 rows=100 width=4) (actual
time=0.016..0.021 rows=100.00 loops=1)"
" Buffers: shared hit=2"
"Planning:"
" Buffers: shared hit=47 dirtied=2"

```

"Planning Time: 10.607 ms"

"Execution Time: 1.428 ms"

## New Query:

```

SELECT
  c.course_id,
  c.title,
  c.dept_name
FROM course c
JOIN section s ON s.course_id = c.course_id
LEFT JOIN takes ta
  ON ta.course_id = s.course_id
  AND ta.sec_id = s.sec_id
  AND ta.semester = s.semester
  AND ta.year = s.year
  AND ta.grade = 'A '
GROUP BY
  c.course_id,
  c.title,
  c.dept_name
HAVING
  COUNT(s.sec_id) > 0
  AND COUNT(s.sec_id) = COUNT(ta.id)
ORDER BY
  c.dept_name,
  c.course_id;

```

## New Execution Plan:

```

"Sort (cost=678.16..678.17 rows=1 width=30) (actual time=3.803..3.806 rows=85.00
  loops=1)"
" Sort Key: c.dept_name, c.course_id"
" Sort Method: quicksort Memory: 29kB"
" Buffers: shared hit=224"
" -> HashAggregate (cost=675.15..678.15 rows=1 width=30) (actual time=3.692..3.706
  rows=85.00 loops=1)"
" Group Key: c.course_id"
" Filter: ((count(s.sec_id) > 0) AND (count(s.sec_id) = count(ta.id)))"
" Batches: 1 Memory Usage: 40kB"

```

```

"   Buffers: shared hit=224"
"   -> Hash Join (cost=11.50..650.27 rows=3318 width=37) (actual time=0.237..3.288
rows=3318.00 loops=1)"
"       Hash Cond: ((s.course_id)::text = (c.course_id)::text)"
"       Buffers: shared hit=224"
"       -> Hash Right Join (cost=5.00..634.87 rows=3318 width=11) (actual
time=0.088..2.711 rows=3318.00 loops=1)"
"           Hash Cond: (((ta.course_id)::text = (s.course_id)::text) AND ((ta.sec_id)::text =
(s.sec_id)::text) AND ((ta.semester)::text = (s.semester)::text) AND (ta.year = s.year))"
"           Buffers: shared hit=222"
"           -> Seq Scan on takes ta (cost=0.00..595.00 rows=3318 width=21) (actual
time=0.010..1.793 rows=3318.00 loops=1)"
"               Filter: ((grade)::text = 'A '::text)"
"               Rows Removed by Filter: 26682"
"               Buffers: shared hit=220"
"           -> Hash (cost=3.00..3.00 rows=100 width=16) (actual time=0.067..0.067
rows=100.00 loops=1)"
"               Buckets: 1024 Batches: 1 Memory Usage: 13kB"
"               Buffers: shared hit=2"
"           -> Seq Scan on section s (cost=0.00..3.00 rows=100 width=16) (actual
time=0.014..0.036 rows=100.00 loops=1)"
"               Buffers: shared hit=2"
"           -> Hash (cost=4.00..4.00 rows=200 width=30) (actual time=0.139..0.139
rows=200.00 loops=1)"
"               Buckets: 1024 Batches: 1 Memory Usage: 21kB"
"               Buffers: shared hit=2"
"           -> Seq Scan on course c (cost=0.00..4.00 rows=200 width=30) (actual
time=0.031..0.081 rows=200.00 loops=1)"
"               Buffers: shared hit=2"
"Planning:"
" Buffers: shared hit=12"
"Planning Time: 0.662 ms"
"Execution Time: 3.906 ms"

```

## Optimization:

### AI Notes:

- Uses a Where Exists instead of a join with group by/having.
- Does not correctly filter by grade; returning 0 rows of data because all A grades have trailing space

### Optimizations Made:

- Used a Join with Group by to optimize

- Fixed to include trailing space after 'A' values in grade filter
- Overall decrease in time and buffers

Metric	Before	After	Change
Buffer Pages Read	47, dirtied 2	12	-74%
Planning Time	10.607 ms	0.662 ms	-94%
Execution Time	1.428 ms	3.906 ms	173%

## Prompt 4:

Given the following PostgreSQL schema: create table classroom

```
(building varchar(15),
room_number varchar(7),
capacity numeric(4,0),
primary key (building, room_number)
);
create table department
(dept_name varchar(20),
building varchar(15),
budget numeric(12,2) check (budget > 0),
primary key (dept_name)
);
create table course
(course_id varchar(8),
title varchar(50),
dept_name varchar(20),
credits numeric(2,0) check (credits > 0),
primary key (course_id),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table instructor
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
salary numeric(8,2) check (salary > 29000),
primary key (ID),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table section
(course_id varchar(8),
sec_id varchar(8),
semester varchar(6)
check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
year numeric(4,0) check (year > 1701 and year <
2100),
building varchar(15),
room_number varchar(7),
time_slot_id varchar(4),
primary key (course_id, sec_id, semester, year),
```

```
foreign key (course_id) references course (course_id)
on delete cascade,
foreign key (building, room_number) references
classroom (building, room_number)
on delete set null
);
create table teaches
(ID varchar(5),
course_id varchar(8),
sec_id varchar(8),
semester varchar(6),
year numeric(4,0),
primary key (ID, course_id, sec_id, semester, year),
foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
on delete cascade,
foreign key (ID) references instructor (ID)
on delete cascade
);
create table student
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
tot_cred numeric(3,0) check (tot_cred >= 0),
primary key (ID),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table takes
(ID varchar(5),
course_id varchar(8),
sec_id varchar(8),
semester varchar(6),
year numeric(4,0),
grade varchar(2),
primary key (ID, course_id, sec_id, semester, year),
foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
on delete cascade,
foreign key (ID) references student (ID)
on delete cascade
);
create table advisor
```

```

(s_ID    varchar(5),
i_ID    varchar(5),
primary key (s_ID),
foreign key (i_ID) references instructor (ID)
on delete set null,
foreign key (s_ID) references student (ID)
on delete cascade
);
create table time_slot
(time_slot_id  varchar(4),
day           varchar(1),
start_hr      numeric(2) check (start_hr >= 0 and start_hr <
24),
start_min     numeric(2) check (start_min >= 0 and
start_min < 60),
end_hr        numeric(2) check (end_hr >= 0 and end_hr <
24),

```

```

end_min       numeric(2) check (end_min >= 0 and
end_min < 60),
primary key (time_slot_id, day, start_hr, start_min)
);
create table prereq
(course_id  varchar(8),
prereq_id  varchar(8),
primary key (course_id, prereq_id),
foreign key (course_id) references course (course_id)
on delete cascade,
foreign key (prereq_id) references course (course_id)
);

```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50 instructors, and 20 departments. From this, write an SQL query for each department, ranking instructors by salary from highest to lowest. Show the instructor's name, department, salary, their rank within the department, and what percentile they fall into.

## Claude AI Response:

```

SELECT
    i.name AS instructor_name,
    i.dept_name,
    i.salary,
    RANK() OVER (
        PARTITION BY i.dept_name
        ORDER BY i.salary DESC
    ) AS dept_salary_rank,
    DENSE_RANK() OVER (
        PARTITION BY i.dept_name
        ORDER BY i.salary DESC
    ) AS
dept_salary_dense_rank,
    COUNT(*) OVER (
        PARTITION BY i.dept_name
    ) AS
dept_instructor_count,
    ROUND(
        PERCENT_RANK() OVER (
            PARTITION BY i.dept_name
            ORDER BY i.salary ASC
        ) * 100, 1
    ) AS salary_percentile,
    NTILE(4) OVER (
        PARTITION BY i.dept_name
        ORDER BY i.salary DESC
    ) AS salary_quartile,
    ROUND(AVG(i.salary) OVER (
        PARTITION BY i.dept_name
    ), 2) AS dept_avg_salary,
    ROUND(
        (i.salary - AVG(i.salary) OVER (PARTITION BY i.dept_name))
        / NULLIF(STDDEV(i.salary) OVER (PARTITION BY i.dept_name), 0)
    , 2) AS salary_zscore
FROM instructor i
WHERE i.dept_name IS NOT NULL
ORDER BY

```

```
i.dept_name,
dept_salary_rank;
```

Column-by-column breakdown:

`RANK()` assigns salary position within the department, with gaps after ties. If two instructors share rank 2, the next rank assigned is 4. This mirrors how athletic competitions are typically scored.

`DENSE_RANK()` also ranks by salary but without gaps — two tied instructors at rank 2 are followed by rank 3, not 4. Useful when you want to know how many distinct salary levels exist above someone.

`COUNT(*) OVER (PARTITION BY dept_name)` is a window aggregate, not a grouping — it attaches the department's total instructor headcount to every row without collapsing them, which is what makes window functions so powerful for reporting.

`PERCENT_RANK()` uses `ORDER BY salary ASC` intentionally. The formula is  $(rank - 1) / (total\ rows - 1)$ , producing a value from 0.0 to 1.0. With ascending order, the lowest earner sits at the 0th percentile and the highest at the 100th — the conventional interpretation of "what percentage of peers earn less than or equal to this salary."

`NTILE(4)` divides the department's instructors into four equal salary buckets (quartiles). Quartile 1 = top earners, quartile 4 = bottom earners, since the window orders by salary `DESC`. With small departments this is approximate — `NTILE` distributes remainder rows into the earliest buckets.

`dept_avg_salary` and `salary_zscore` give distributional context. The Z-score measures how many standard deviations an instructor's salary sits above or below their department mean. `NULLIF(..., 0)` guards against division by zero in single-instructor departments where `STDDEV` returns 0.

Why `WHERE dept_name IS NOT NULL`

The `instructor.dept_name` column allows NULLs (via `ON DELETE SET NULL`). Without this filter, instructors with no department would all be grouped into a single NULL partition, which is technically valid SQL but almost certainly not meaningful for a per-department ranking report.

Key window function concept — `PARTITION BY` vs `GROUP BY`

	<code>GROUP BY</code>	<code>PARTITION BY</code>
Output rows	One per group	Same as input
Aggregates	Collapses rows	Attached to each row
Can mix row + aggregate	No	Yes

All five window functions here share the same logical scan of the `instructor` table — PostgreSQL is smart enough to compute them in a single pass rather than rescanning per function.

## Execution Plan:

One of the instances of rounding in here doesn't actually work, fixed in order to run initial execution plan.

"Incremental Sort (cost=3.58..9.49 rows=50 width=130) (actual time=0.954..0.993 rows=50.00 loops=1)"

```

" Sort Key: dept_name, (rank() OVER w1)"
" Presorted Key: dept_name"
" Full-sort Groups: 2 Sort Method: quicksort Average Memory: 28kB Peak Memory:
  28kB"
" Buffers: shared hit=4"
" -> WindowAgg (cost=3.27..8.14 rows=50 width=130) (actual time=0.790..0.928
  rows=50.00 loops=1)"
"   Window: w3 AS (PARTITION BY dept_name)"
"   Storage: Memory Maximum Storage: 17kB"
"   Buffers: shared hit=1"
"   -> WindowAgg (cost=3.10..6.39 rows=50 width=54) (actual time=0.216..0.274
  rows=50.00 loops=1)"
"     Window: w2 AS (PARTITION BY dept_name ORDER BY salary ROWS
  UNBOUNDED PRECEDING)"
"     Storage: Memory Maximum Storage: 17kB"
"     Buffers: shared hit=1"
"     -> Incremental Sort (cost=3.03..5.52 rows=50 width=46) (actual
  time=0.209..0.239 rows=50.00 loops=1)"
"       Sort Key: dept_name, salary"
"       Presorted Key: dept_name"
"       Full-sort Groups: 2 Sort Method: quicksort Average Memory: 27kB Peak
  Memory: 27kB"
"       Buffers: shared hit=1"
"       -> WindowAgg (cost=2.94..4.16 rows=50 width=46) (actual
  time=0.155..0.192 rows=50.00 loops=1)"
"         Window: w1 AS (PARTITION BY dept_name ORDER BY salary ROWS
  UNBOUNDED PRECEDING)"
"         Storage: Memory Maximum Storage: 17kB"
"         Buffers: shared hit=1"
"         -> Sort (cost=2.91..3.04 rows=50 width=26) (actual time=0.143..0.146
  rows=50.00 loops=1)"
"           Sort Key: dept_name, salary DESC"
"           Sort Method: quicksort Memory: 27kB"
"           Buffers: shared hit=1"
"           -> Seq Scan on instructor i (cost=0.00..1.50 rows=50 width=26)
  (actual time=0.059..0.065 rows=50.00 loops=1)"
"             Filter: (dept_name IS NOT NULL)"
"             Buffers: shared hit=1"
"Planning:"
" Buffers: shared hit=40"

```

"Planning Time: 22.341 ms"

"Execution Time: 1.520 ms"

## New Query:

```

SELECT
  i.name                               AS instructor_name,
  i.dept_name,
  i.salary,
  RANK() OVER (
    PARTITION BY i.dept_name
    ORDER BY i.salary DESC
  )                                     AS dept_salary_rank,
  PERCENT_RANK() OVER (
    PARTITION BY i.dept_name
    ORDER BY i.salary ASC
  ) * 100                              AS salary_percentile
FROM instructor i
WHERE i.dept_name IS NOT NULL
ORDER BY
  i.dept_name,
  dept_salary_rank;

```

## New Execution Plan:

```

"Incremental Sort (cost=2.42..17.60 rows=50 width=42) (actual time=0.469..0.510 rows=50.00
loops=1)"
" Sort Key: dept_name, (rank() OVER w1)"
" Presorted Key: dept_name"
" Full-sort Groups: 2 Sort Method: quicksort Average Memory: 27kB Peak Memory: 27kB"
" Buffers: shared hit=1 read=1"
" -> WindowAgg (cost=1.53..16.24 rows=50 width=42) (actual time=0.315..0.424 rows=50.00
loops=1)"
" Window: w2 AS (PARTITION BY dept_name ORDER BY salary ROWS UNBOUNDED
PRECEDING)"
" Storage: Memory Maximum Storage: 17kB"
" Buffers: shared hit=1 read=1"
" -> Incremental Sort (cost=1.23..15.24 rows=50 width=34) (actual time=0.297..0.350
rows=50.00 loops=1)"

```

```

"      Sort Key: dept_name, salary"
"      Presorted Key: dept_name"
"      Full-sort Groups: 2  Sort Method: quicksort  Average Memory: 26kB  Peak Memory:
26kB"
"      Buffers: shared hit=1 read=1"
"      -> WindowAgg (cost=0.42..13.89 rows=50 width=34) (actual time=0.157..0.250
rows=50.00 loops=1)"
"          Window: w1 AS (PARTITION BY dept_name ORDER BY salary ROWS
UNBOUNDED PRECEDING)"
"          Storage: Memory  Maximum Storage: 17kB"
"          Buffers: shared hit=1 read=1"
"          -> Index Scan using idx_instructor_dept_salary on instructor i (cost=0.14..13.02
rows=50 width=26) (actual time=0.145..0.170 rows=50.00 loops=1)"
"              Index Cond: (dept_name IS NOT NULL)"
"              Index Searches: 1"
"              Buffers: shared hit=1 read=1"
"Planning Time: 0.255 ms"
"Execution Time: 0.566 ms"

```

## Optimization:

### AI Notes:

- Includes many unnecessary values returned
- Missing index so does many unnecessary seq scans.

### Optimizations Made:

- Created Index: CREATE INDEX idx\_instructor\_dept\_salary ON instructor(dept\_name, salary DESC)
- Removed unnecessary values
- Removed unnecessary rounding that caused error

Metric	Before	After	Change
Scan Type	Seq Scan	Index Scan	Changed
Buffer Pages Read	40	1	-97.5%
Planning Time	22.341 ms	0.255 ms	-99%
Execution Time	1.520 ms	0.566 ms	-63%

## Prompt 5:

Given the following PostgreSQL schema: create table classroom

```
(building varchar(15),
 room_number varchar(7),
 capacity numeric(4,0),
 primary key (building, room_number)
);
create table department
(dept_name varchar(20),
 building varchar(15),
 budget numeric(12,2) check (budget > 0),
 primary key (dept_name)
);
create table course
(course_id varchar(8),
 title varchar(50),
 dept_name varchar(20),
 credits numeric(2,0) check (credits > 0),
 primary key (course_id),
 foreign key (dept_name) references department
(dept_name)
 on delete set null
);
create table instructor
(ID varchar(5),
 name varchar(20) not null,
 dept_name varchar(20),
 salary numeric(8,2) check (salary > 29000),
 primary key (ID),
 foreign key (dept_name) references department
(dept_name)
 on delete set null
);
create table section
(course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6)
 check (semester in ('Fall', 'Winter', 'Spring', 'Summer')),
 year numeric(4,0) check (year > 1701 and year <
2100),
 building varchar(15),
 room_number varchar(7),
 time_slot_id varchar(4),
 primary key (course_id, sec_id, semester, year),
 foreign key (course_id) references course (course_id)
 on delete cascade,
 foreign key (building, room_number) references
classroom (building, room_number)
 on delete set null
);
create table teaches
(ID varchar(5),
 course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6),
 year numeric(4,0),
 primary key (ID, course_id, sec_id, semester, year),
 foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
 on delete cascade,
 foreign key (ID) references instructor (ID)
 on delete cascade
```

```
);
create table student
(ID varchar(5),
 name varchar(20) not null,
 dept_name varchar(20),
 tot_cred numeric(3,0) check (tot_cred >= 0),
 primary key (ID),
 foreign key (dept_name) references department
(dept_name)
 on delete set null
);
create table takes
(ID varchar(5),
 course_id varchar(8),
 sec_id varchar(8),
 semester varchar(6),
 year numeric(4,0),
 grade varchar(2),
 primary key (ID, course_id, sec_id, semester, year),
 foreign key (course_id, sec_id, semester, year) references
section (course_id, sec_id, semester, year)
 on delete cascade,
 foreign key (ID) references student (ID)
 on delete cascade
);
create table advisor
(s_ID varchar(5),
 i_ID varchar(5),
 primary key (s_ID),
 foreign key (i_ID) references instructor (ID)
 on delete set null,
 foreign key (s_ID) references student (ID)
 on delete cascade
);
create table time_slot
(time_slot_id varchar(4),
 day varchar(1),
 start_hr numeric(2) check (start_hr >= 0 and start_hr <
24),
 start_min numeric(2) check (start_min >= 0 and
start_min < 60),
 end_hr numeric(2) check (end_hr >= 0 and end_hr <
24),
 end_min numeric(2) check (end_min >= 0 and
end_min < 60),
 primary key (time_slot_id, day, start_hr, start_min)
);
create table prereq
(course_id varchar(8),
 prereq_id varchar(8),
 primary key (course_id, prereq_id),
 foreign key (course_id) references course (course_id)
 on delete cascade,
 foreign key (prereq_id) references course (course_id)
);
```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50 instructors, and 20 departments. From this, write an SQL query to find students who have taken courses offered by at least 3 different departments but have never taken a course taught by their own advisor.

## Claude AI Response:

This query has two independent conditions that must both be true, so it's cleanest to solve each part separately in CTEs and then intersect them.

sql

```

WITH
-----
-- 1. Students who have taken courses from at least 3 departments
-----
multi_dept_students AS (
    SELECT
        ta.ID
    FROM takes ta
    JOIN course c ON c.course_id = ta.course_id
    WHERE c.dept_name IS NOT NULL
    GROUP BY ta.ID
    HAVING COUNT(DISTINCT c.dept_name) >= 3
),
-----
-- 2. All (student, instructor) pairs where the student has taken
--    at least one section the instructor taught
-----
student_took_from_instructor AS (
    SELECT DISTINCT
        ta.ID          AS student_id,
        te.ID          AS instructor_id
    FROM takes ta
    JOIN teaches te
        ON te.course_id = ta.course_id
        AND te.sec_id   = ta.sec_id
        AND te.semester = ta.semester
        AND te.year     = ta.year
),
-----
-- 3. Students who have taken a course taught by their own advisor
-----
took_from_advisor AS (
    SELECT
        a.s_ID      AS student_id
    FROM advisor a
    JOIN student_took_from_instructor stfi
        ON stfi.student_id = a.s_ID
        AND stfi.instructor_id = a.i_ID
)
-----
-- Final: multi-department students who never took from advisor
-----
SELECT
    s.ID,
    s.name                AS student_name,
    s.dept_name          AS student_dept,
    s.tot_cred,
    COUNT(DISTINCT c.dept_name) AS distinct_depts_taken,
    COUNT(DISTINCT ta.course_id) AS total_courses_taken,
    i.name                AS advisor_name,
    i.dept_name          AS advisor_dept

```

```

FROM student s
JOIN multi_dept_students mds ON mds.ID = s.ID
JOIN takes ta ON ta.ID = s.ID
JOIN course c ON c.course_id = ta.course_id
LEFT JOIN advisor a ON a.s_ID = s.ID
LEFT JOIN instructor i ON i.ID = a.i_ID
WHERE s.ID NOT IN (
    SELECT student_id FROM took_from_advisor
)
GROUP BY
    s.ID,
    s.name,
    s.dept_name,
    s.tot_cred,
    i.name,
    i.dept_name
ORDER BY
    distinct_depts_taken DESC,
    s.name;

```

---

#### CTE breakdown:

`multi_dept_students` groups `takes` by student and counts distinct `dept_name` values from the joined `course` table. The `HAVING` filter keeps only students meeting the  $\geq 3$  threshold. `WHERE c.dept_name IS NOT NULL` excludes orphaned courses whose department was deleted, since those carry no meaningful departmental identity.

`student_took_from_instructor` resolves the many-to-many relationship between students and the instructors whose sections they actually sat in. The four-column join on (`course_id`, `sec_id`, `semester`, `year`) is essential — without all four, a student who took *any* section of a course would incorrectly match an instructor who taught a *different* section of the same course in a different semester. `took_from_advisor` applies the advisor constraint by joining `advisor` to the previous CTE. It only produces a row when the student's recorded advisor is also one of the instructors they learned from.

Final `SELECT` — why `NOT IN` over `NOT EXISTS` here

Both are logically equivalent, but `NOT IN` reads naturally when the subquery is already a clean, pre-built CTE with no NULLs. Since `took_from_advisor.student_id` is derived from `advisor.s_ID` which references `student.ID` (a `NOT NULL` primary key), there is no `NULL` risk that would make `NOT IN` behave unexpectedly.

Why `LEFT JOIN` on `advisor` and `instructor` in the final `SELECT`

Students are not required to have an advisor — the `advisor` table has no mandatory entry per student. Using `LEFT JOIN` ensures students without advisors still appear in results. Their `advisor_name` and `advisor_dept` columns will show `NULL`, which is informative rather than silently dropping the row.

---

#### The `NULL` trap to be aware of

If a student has no advisor (`a.i_ID IS NULL`), the `took_from_advisor` CTE will never produce a row for them, so they naturally satisfy the "never took from advisor" condition and will appear in results — which is the correct interpretation, since the constraint is vacuously satisfied.

If you want to exclude advisorless students from results, add this to the final `WHERE`:

```

sql
AND a.s_ID IS NOT NULL

```

---

#### Performance note

The most expensive operation is the four-way join in `student_took_from_instructor` across 30,000 `takes` rows. A composite index on `teaches(course_id, sec_id, semester, year, ID)` allows the join to be resolved efficiently. The `DISTINCT` in that CTE then deduplicates what is typically a small result set ( $\sim 50$  instructors  $\times$  bounded student overlap).

## Execution Plan:

```

"Sort (cost=5943.25..5955.78 rows=5010 width=57) (actual time=375.408..375.548
  rows=1580.00 loops=1)"
"  Sort Key: (count(DISTINCT c.dept_name)) DESC, s.name"
"  Sort Method: quicksort  Memory: 179kB"
"  Buffers: shared hit=17213"
"  -> GroupAggregate (cost=5510.13..5635.38 rows=5010 width=57) (actual
  time=338.007..371.552 rows=1580.00 loops=1)"
"    Group Key: s.id, i.name, i.dept_name"
"    Buffers: shared hit=17213"
"    -> Sort (cost=5510.13..5522.65 rows=5010 width=54) (actual time=337.926..344.527
  rows=23486.00 loops=1)"
"      Sort Key: s.id, i.name, i.dept_name, c.dept_name"
"      Sort Method: quicksort  Memory: 2566kB"
"      Buffers: shared hit=17213"
"      -> Hash Left Join (cost=4536.13..5202.25 rows=5010 width=54) (actual
  time=189.038..227.836 rows=23486.00 loops=1)"
"        Hash Cond: ((a.i_id)::text = (i.id)::text)"
"        Buffers: shared hit=17213"
"        -> Hash Left Join (cost=4534.01..5185.86 rows=5010 width=42) (actual
  time=188.992..221.560 rows=23486.00 loops=1)"
"          Hash Cond: ((s.id)::text = (a.s_id)::text)"
"          Buffers: shared hit=17212"
"          -> Hash Join (cost=4478.01..5116.67 rows=5010 width=37) (actual
  time=188.285..213.452 rows=23486.00 loops=1)"
"            Hash Cond: ((ta.course_id)::text = (c.course_id)::text)"
"            Buffers: shared hit=17201"
"            -> Hash Join (cost=4471.51..5096.74 rows=5010 width=28) (actual
  time=188.212..207.325 rows=23486.00 loops=1)"
"              Hash Cond: ((ta_1.id)::text = (s.id)::text)"
"              Buffers: shared hit=17199"
"              -> Hash Join (cost=3096.16..3695.07 rows=10005 width=14)
  (actual time=132.743..143.737 rows=30000.00 loops=1)"
"                Hash Cond: ((ta.id)::text = (ta_1.id)::text)"
"                Buffers: shared hit=442"
"                -> Seq Scan on takes ta (cost=0.00..520.00 rows=30000
  width=9) (actual time=0.035..2.305 rows=30000.00 loops=1)"

```

```

"           Buffers: shared hit=220"
"           -> Hash (cost=3087.82..3087.82 rows=667 width=5) (actual
time=132.693..132.702 rows=2000.00 loops=1)"
"           Buckets: 2048 (originally 1024) Batches: 1 (originally 1)
Memory Usage: 90kB"
"           Buffers: shared hit=222"
"           -> GroupAggregate (cost=2837.82..3087.82 rows=667
width=5) (actual time=118.627..132.078 rows=2000.00 loops=1)"
"           Group Key: ta_1.id"
"           Filter: (count(DISTINCT c_1.dept_name) >= 3)"
"           Buffers: shared hit=222"
"           -> Sort (cost=2837.82..2912.82 rows=30000 width=14)
(actual time=118.593..125.578 rows=30000.00 loops=1)"
"           Sort Key: ta_1.id, c_1.dept_name"
"           Sort Method: quicksort Memory: 1803kB"
"           Buffers: shared hit=222"
"           -> Hash Join (cost=6.50..606.92 rows=30000
width=14) (actual time=0.096..11.682 rows=30000.00 loops=1)"
"           Hash Cond: ((ta_1.course_id)::text =
(c_1.course_id)::text)"
"           Buffers: shared hit=222"
"           -> Seq Scan on takes ta_1 (cost=0.00..520.00
rows=30000 width=9) (actual time=0.008..3.005 rows=30000.00 loops=1)"
"           Buffers: shared hit=220"
"           -> Hash (cost=4.00..4.00 rows=200 width=13)
(actual time=0.081..0.084 rows=200.00 loops=1)"
"           Buckets: 1024 Batches: 1 Memory Usage:
17kB"
"           Buffers: shared hit=2"
"           -> Seq Scan on course c_1 (cost=0.00..4.00
rows=200 width=13) (actual time=0.016..0.043 rows=200.00 loops=1)"
"           Filter: (dept_name IS NOT NULL)"
"           Buffers: shared hit=2"
"           -> Hash (cost=1362.85..1362.85 rows=1000 width=24) (actual
time=55.462..55.475 rows=1580.00 loops=1)"
"           Buckets: 2048 (originally 1024) Batches: 1 (originally 1)
Memory Usage: 108kB"
"           Buffers: shared hit=16757"
"           -> Seq Scan on student s (cost=1322.85..1362.85 rows=1000
width=24) (actual time=54.772..55.257 rows=1580.00 loops=1)"

```

```

"                               Filter: (NOT (ANY ((id)::text = ((hashed SubPlan
1).col1)::text)))"
"                               Rows Removed by Filter: 420"
"                               Buffers: shared hit=16757"
"                               SubPlan 1"
"                               -> Hash Join (cost=1281.33..1322.83 rows=8 width=5)
(actual time=54.301..54.677 rows=420.00 loops=1)"
"                               Hash Cond: (((a_1.s_id)::text = (ta_2.id)::text) AND
((a_1.i_id)::text = (te.id)::text))"
"                               Buffers: shared hit=16742"
"                               -> Seq Scan on advisor a_1 (cost=0.00..31.00
rows=2000 width=10) (actual time=0.009..0.104 rows=2000.00 loops=1)"
"                               Buffers: shared hit=11"
"                               -> Hash (cost=1257.75..1257.75 rows=1572
width=10) (actual time=54.272..54.279 rows=21838.00 loops=1)"
"                               Buckets: 32768 (originally 2048) Batches: 1
(originally 1) Memory Usage: 1189kB"
"                               Buffers: shared hit=16731"
"                               -> HashAggregate (cost=1242.03..1257.75
rows=1572 width=10) (actual time=46.638..50.505 rows=21838.00 loops=1)"
"                               Group Key: ta_2.id, te.id"
"                               Batches: 1 Memory Usage: 1305kB"
"                               Buffers: shared hit=16731"
"                               -> Nested Loop (cost=3.31..1234.17
rows=1572 width=10) (actual time=0.178..36.933 rows=30000.00 loops=1)"
"                               Buffers: shared hit=16731"
"                               -> Seq Scan on teaches te (cost=0.00..2.00
rows=100 width=21) (actual time=0.057..0.074 rows=100.00 loops=1)"
"                               Buffers: shared hit=1"
"                               -> Memoize (cost=3.31..14.38 rows=6
width=21) (actual time=0.086..0.338 rows=300.00 loops=100)"
"                               Cache Key: te.course_id, te.sec_id,
te.semester, te.year"
"                               Cache Mode: logical"
"                               Hits: 0 Misses: 100 Evictions: 0
Overflows: 0 Memory Usage: 1616kB"
"                               Buffers: shared hit=16730"
"                               -> Bitmap Heap Scan on takes ta_2
(cost=3.30..14.37 rows=6 width=21) (actual time=0.081..0.240 rows=300.00 loops=100)"

```

```

"                               Recheck Cond: (((course_id)::text =
(te.course_id)::text) AND ((sec_id)::text = (te.sec_id)::text) AND ((semester)::text =
(te.semester)::text) AND (year = te.year))"
"                               Heap Blocks: exact=16401"
"                               Buffers: shared hit=16730"
"                               -> Bitmap Index Scan on
idx_takes_section (cost=0.00..3.30 rows=6 width=0) (actual time=0.067..0.067
rows=300.00 loops=100)"
"                               Index Cond: (((course_id)::text =
(te.course_id)::text) AND ((sec_id)::text = (te.sec_id)::text) AND ((semester)::text =
(te.semester)::text) AND (year = te.year))"
"                               Index Searches: 100"
"                               Buffers: shared hit=329"
"                               -> Hash (cost=4.00..4.00 rows=200 width=13) (actual
time=0.065..0.066 rows=200.00 loops=1)"
"                               Buckets: 1024 Batches: 1 Memory Usage: 17kB"
"                               Buffers: shared hit=2"
"                               -> Seq Scan on course c (cost=0.00..4.00 rows=200 width=13)
(actual time=0.021..0.036 rows=200.00 loops=1)"
"                               Buffers: shared hit=2"
"                               -> Hash (cost=31.00..31.00 rows=2000 width=10) (actual
time=0.695..0.697 rows=2000.00 loops=1)"
"                               Buckets: 2048 Batches: 1 Memory Usage: 102kB"
"                               Buffers: shared hit=11"
"                               -> Seq Scan on advisor a (cost=0.00..31.00 rows=2000 width=10)
(actual time=0.081..0.237 rows=2000.00 loops=1)"
"                               Buffers: shared hit=11"
"                               -> Hash (cost=1.50..1.50 rows=50 width=22) (actual time=0.036..0.038
rows=50.00 loops=1)"
"                               Buckets: 1024 Batches: 1 Memory Usage: 11kB"
"                               Buffers: shared hit=1"
"                               -> Seq Scan on instructor i (cost=0.00..1.50 rows=50 width=22) (actual
time=0.021..0.026 rows=50.00 loops=1)"
"                               Buffers: shared hit=1"
"Planning:"
" Buffers: shared hit=44"
"Planning Time: 2.071 ms"
"Execution Time: 377.760 ms"

```

## New Query:

```

WITH multi_dept_students AS (
  SELECT ta.ID
  FROM takes ta
  JOIN course c ON c.course_id = ta.course_id
  WHERE c.dept_name IS NOT NULL
  GROUP BY ta.ID
  HAVING COUNT(DISTINCT c.dept_name) >= 3
),
took_from_advisor AS (
  SELECT DISTINCT ta.ID
  FROM takes ta
  JOIN teaches te
    ON te.course_id = ta.course_id
    AND te.sec_id = ta.sec_id
    AND te.semester = ta.semester
    AND te.year = ta.year
  JOIN advisor a
    ON a.s_ID = ta.ID
    AND a.i_ID = te.ID
)
SELECT s.ID, s.name
FROM student s
JOIN multi_dept_students mds ON mds.ID = s.ID

EXCEPT

SELECT s.ID, s.name
FROM student s
JOIN took_from_advisor tfa ON tfa.ID = s.ID

ORDER BY name;

```

## New Execution Plan:

```

"Sort (cost=4314.30..4315.97 rows=667 width=82) (actual time=150.816..150.920
  rows=1580.00 loops=1)"
" Sort Key: s.name"
" Sort Method: quicksort Memory: 98kB"
" Buffers: shared hit=484"
" -> SetOp Except (cost=4276.11..4283.02 rows=667 width=82) (actual
  time=145.322..146.738 rows=1580.00 loops=1)"

```

```
" Buffers: shared hit=484"
" -> Sort (cost=3180.86..3182.53 rows=667 width=11) (actual time=122.740..122.925
rows=2000.00 loops=1)"
"   Sort Key: s.id, s.name"
"   Sort Method: quicksort Memory: 112kB"
"   Buffers: shared hit=237"
" -> Hash Join (cost=2897.82..3149.57 rows=667 width=11) (actual
time=111.547..121.758 rows=2000.00 loops=1)"
"   Hash Cond: ((ta.id)::text = (s.id)::text)"
"   Buffers: shared hit=237"
" -> GroupAggregate (cost=2837.82..3087.82 rows=667 width=5) (actual
time=110.689..119.988 rows=2000.00 loops=1)"
"   Group Key: ta.id"
"   Filter: (count(DISTINCT c.dept_name) >= 3)"
"   Buffers: shared hit=222"
" -> Sort (cost=2837.82..2912.82 rows=30000 width=14) (actual
time=110.660..114.285 rows=30000.00 loops=1)"
"   Sort Key: ta.id, c.dept_name"
"   Sort Method: quicksort Memory: 1803kB"
"   Buffers: shared hit=222"
" -> Hash Join (cost=6.50..606.92 rows=30000 width=14) (actual
time=0.204..10.651 rows=30000.00 loops=1)"
"   Hash Cond: ((ta.course_id)::text = (c.course_id)::text)"
"   Buffers: shared hit=222"
" -> Seq Scan on takes ta (cost=0.00..520.00 rows=30000 width=9)
(actual time=0.016..1.928 rows=30000.00 loops=1)"
"   Buffers: shared hit=220"
" -> Hash (cost=4.00..4.00 rows=200 width=13) (actual
time=0.115..0.117 rows=200.00 loops=1)"
"   Buckets: 1024 Batches: 1 Memory Usage: 17kB"
"   Buffers: shared hit=2"
" -> Seq Scan on course c (cost=0.00..4.00 rows=200 width=13)
(actual time=0.016..0.065 rows=200.00 loops=1)"
"   Filter: (dept_name IS NOT NULL)"
"   Buffers: shared hit=2"
" -> Hash (cost=35.00..35.00 rows=2000 width=11) (actual time=0.834..0.836
rows=2000.00 loops=1)"
"   Buckets: 2048 Batches: 1 Memory Usage: 104kB"
"   Buffers: shared hit=15"
```

```

"         -> Seq Scan on student s (cost=0.00..35.00 rows=2000 width=11) (actual
time=0.032..0.384 rows=2000.00 loops=1)"
"             Buffers: shared hit=15"
"         -> Sort (cost=1095.26..1095.33 rows=31 width=11) (actual time=22.570..22.595
rows=420.00 loops=1)"
"             Sort Key: s_1.id, s_1.name"
"             Sort Method: quicksort Memory: 38kB"
"             Buffers: shared hit=247"
"         -> Hash Join (cost=1054.23..1094.49 rows=31 width=11) (actual
time=21.259..21.783 rows=420.00 loops=1)"
"             Hash Cond: ((s_1.id)::text = (ta_1.id)::text)"
"             Buffers: shared hit=247"
"         -> Seq Scan on student s_1 (cost=0.00..35.00 rows=2000 width=11) (actual
time=0.066..0.190 rows=2000.00 loops=1)"
"             Buffers: shared hit=15"
"         -> Hash (cost=1053.84..1053.84 rows=31 width=5) (actual
time=21.171..21.175 rows=420.00 loops=1)"
"             Buckets: 1024 Batches: 1 Memory Usage: 24kB"
"             Buffers: shared hit=232"
"         -> Unique (cost=1053.68..1053.84 rows=31 width=5) (actual
time=20.988..21.109 rows=420.00 loops=1)"
"             Buffers: shared hit=232"
"         -> Sort (cost=1053.68..1053.76 rows=31 width=5) (actual
time=20.984..21.016 rows=613.00 loops=1)"
"             Sort Key: ta_1.id"
"             Sort Method: quicksort Memory: 25kB"
"             Buffers: shared hit=232"
"         -> Hash Join (cost=60.25..1052.92 rows=31 width=5) (actual
time=0.517..20.079 rows=613.00 loops=1)"
"             Hash Cond: (((ta_1.course_id)::text = (te.course_id)::text) AND
(((ta_1.sec_id)::text = (te.sec_id)::text) AND ((ta_1.semester)::text = (te.semester)::text)
AND (ta_1.year = te.year) AND ((a.i_id)::text = (te.id)::text))"
"             Buffers: shared hit=232"
"         -> Hash Join (cost=56.00..654.92 rows=30000 width=26)
(actual time=0.340..13.871 rows=30000.00 loops=1)"
"             Hash Cond: ((ta_1.id)::text = (a.s_id)::text)"
"             Buffers: shared hit=231"
"         -> Seq Scan on takes ta_1 (cost=0.00..520.00 rows=30000
width=21) (actual time=0.015..1.983 rows=30000.00 loops=1)"
"             Buffers: shared hit=220"

```

```

"          -> Hash (cost=31.00..31.00 rows=2000 width=10) (actual
time=0.307..0.308 rows=2000.00 loops=1)"
"          Buckets: 2048 Batches: 1 Memory Usage: 102kB"
"          Buffers: shared hit=11"
"          -> Seq Scan on advisor a (cost=0.00..31.00 rows=2000
width=10) (actual time=0.015..0.109 rows=2000.00 loops=1)"
"          Buffers: shared hit=11"
"          -> Hash (cost=2.00..2.00 rows=100 width=21) (actual
time=0.076..0.077 rows=100.00 loops=1)"
"          Buckets: 1024 Batches: 1 Memory Usage: 14kB"
"          Buffers: shared hit=1"
"          -> Seq Scan on teaches te (cost=0.00..2.00 rows=100
width=21) (actual time=0.019..0.031 rows=100.00 loops=1)"
"          Buffers: shared hit=1"
"Planning:"
" Buffers: shared hit=24"
"Planning Time: 2.333 ms"
"Execution Time: 151.665 ms"

```

## Optimization:

### AI Notes:

- Has unnecessary third CTE to address students taking from their advisor that does not improve speed
- Returns many unnecessary columns
- Does not use indexes

### Optimizations Made:

- Created Index: CREATE INDEX idx\_course\_dept ON course (course\_id, dept\_name)
- Removed unnecessary values returned
- Removed unnecessary CTE

Metric	Before	After	Change
Scan Type	Bitmap Index Scan	Seq Scan	Changed
Buffer Pages Read	44	24	-45.5%
Planning Time	2.071 ms	2.333 ms	13%
Execution Time	377.760 ms	151.665 ms	-60%

## Part 2: Reflection

### **Patterns in AI SQL:**

Some repeated errors were noticed across functions. For example, in both Query 5 and Query 4, unnecessary and unrequested values were returned, adding additional computations for every row in the instance of Query 4. Additionally, consistent issues were found with the formatting of rounding numbers. This suggests that the AI is trying to provide useful information, but not necessarily the information being requested.

### **Limitations Observed:**

The use of indexes was never suggested by the AI throughout the process. Moreover, multiple of the queries used more complicated where clauses rather than joining tables, such as in Query 2 and Query 5. This shows that while the AI may be generating function SQL, it may not be scalable when individual row execution could be more costly.

### **Practical Implications:**

One thing I have learned from previous AI use is to be extremely specific in requests. Here I went with the default prompting, but in future applications I might also prompt to ask for indexes to be included or to focus on optimizing to reduce functions within 'Where' clauses, for example. An additional check to me would be to look at whether the outputs being returned are necessary, given the commonality of that issue in my experience.

### **Learning Impact:**

My main takeaway from this was how useful indexes could be. While the parallel processing on my device means that for this dataset, sequential scans were not incredibly costly, I gained a large appreciation for the benefits of them through this exercise.

## Part 3 (Appendix):

Given the following PostgreSQL schema: create table

```

classroom
  (building varchar(15),
  room_number varchar(7),
  capacity numeric(4,0),
  primary key (building, room_number)
  );
create table department
  (dept_name varchar(20),
  building varchar(15),
  budget numeric(12,2) check (budget > 0),
  primary key (dept_name)
  );
create table course
  (course_id varchar(8),
  title varchar(50),
  dept_name varchar(20),
  credits numeric(2,0) check (credits > 0),
  primary key (course_id),
  foreign key (dept_name) references department
    (dept_name)
    on delete set null
  );
create table instructor
  (ID varchar(5),
  name varchar(20) not null,
  dept_name varchar(20),
  salary numeric(8,2) check (salary > 29000),
  primary key (ID),
  foreign key (dept_name) references department
    (dept_name)
    on delete set null
  );
create table section
  (course_id varchar(8),
  sec_id varchar(8),
  semester varchar(6)
  check (semester in ('Fall', 'Winter', 'Spring',
  'Summer')),
  year numeric(4,0) check (year > 1701 and year
  < 2100),
  building varchar(15),
  room_number varchar(7),
  time_slot_id varchar(4),
  primary key (course_id, sec_id, semester, year),
  foreign key (course_id) references course
    (course_id)
    on delete cascade,
  foreign key (building, room_number) references
  classroom (building, room_number)
  on delete set null
  );
create table teaches
  (ID varchar(5),
  course_id varchar(8),
  sec_id varchar(8),
  semester varchar(6),
  year numeric(4,0),
  primary key (ID, course_id, sec_id, semester, year),
  foreign key (course_id, sec_id, semester, year)
  references section (course_id, sec_id, semester,
  year)
  on delete cascade,
  foreign key (ID) references instructor (ID)
  on delete cascade
  );
create table student
  (ID varchar(5),
  name varchar(20) not null,
  dept_name varchar(20),
  tot_cred numeric(3,0) check (tot_cred >= 0),
  primary key (ID),
  foreign key (dept_name) references department
    (dept_name)
    on delete set null
  );
create table takes
  (ID varchar(5),
  course_id varchar(8),
  sec_id varchar(8),
  semester varchar(6),
  year numeric(4,0),
  grade varchar(2),
  primary key (ID, course_id, sec_id, semester, year),
  foreign key (course_id, sec_id, semester, year)
  references section (course_id, sec_id, semester,
  year)
  on delete cascade,
  foreign key (ID) references student (ID)
  on delete cascade
  );
create table advisor
  (s_ID varchar(5),
  i_ID varchar(5),
  primary key (s_ID),
  foreign key (i_ID) references instructor (ID)
  on delete set null,
  foreign key (s_ID) references student (ID)
  on delete cascade
  );
create table time_slot
  (time_slot_id varchar(4),
  day varchar(1),
  start_hr numeric(2) check (start_hr >= 0 and
  start_hr < 24),
  start_min numeric(2) check (start_min >= 0 and
  start_min < 60),
  end_hr numeric(2) check (end_hr >= 0 and
  end_hr < 24),
  end_min numeric(2) check (end_min >= 0 and
  end_min < 60),
  primary key (time_slot_id, day, start_hr, start_min)
  );
create table prereq
  (course_id varchar(8),
  prereq_id varchar(8),
  primary key (course_id, prereq_id),
  foreign key (course_id) references course
    (course_id)
    on delete cascade,
  foreign key (prereq_id) references course (course_id)
  );

```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50

instructors, and 20 departments. From this, write an SQL query to find the total number of course enrollments and the average total credits (tot\_cred) of students in each department. Rank departments from highest enrollment count to lowest.

Show more

Apr 30

**Claude responded: How it works:**

sql

```
SELECT
    d.dept_name,
    COUNT(t.ID) AS
    total_enrollments,
    ROUND(AVG(s.tot_cred), 2) AS
    avg_tot_cred
FROM department d
LEFT JOIN student s ON s.dept_name =
    d.dept_name
LEFT JOIN takes t ON t.ID = s.ID
GROUP BY d.dept_name
ORDER BY total_enrollments DESC;
```

How it works:

The query chains three LEFT JOINS so that every department appears in the result, even those with no students or no enrollment records.

- `department` → `student` links each department to its students
- `student` → `takes` links each student to their enrollment rows
- `COUNT(t.ID)` counts non-NULL rows in `takes`, so departments with no enrollments correctly get 0
- `AVG(s.tot_cred)` averages the total credits across all students in that department (NULL if no students exist)
- `ROUND(..., 2)` keeps the average tidy to two decimal places
- `ORDER BY total_enrollments DESC` ranks from most-enrolled to least

Why LEFT JOIN instead of INNER JOIN?

With ~20 departments but potentially sparse enrollment, an INNER JOIN would silently drop any department that has no students or no takes rows. LEFT JOIN ensures every department is represented, which is usually the desired behavior for reporting queries like this.

Performance note:

With 30,000 rows in `takes` and 2,000 students, indexes on `takes(ID)`, `student(dept_name)`, and `student(ID)` will keep this fast. These are already covered by the primary key on `takes(ID, course_id, sec_id, semester, year)` and the primary key on `student(ID)`, but a separate index on `student(dept_name)` would help the join to `department`.

**You said: Given the following PostgreSQL schema: create table classroom**

Given the following PostgreSQL schema: create table classroom

```
(building varchar(15),
room_number varchar(7),
capacity numeric(4,0),
primary key (building, room_number)
);
```

create table department

```
(dept_name varchar(20),
building varchar(15),
budget numeric(12,2) check (budget > 0),
primary key (dept_name)
);
```

create table course

```
(course_id varchar(8),
title varchar(50),
dept_name varchar(20),
credits numeric(2,0) check (credits > 0),
primary key (course_id),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
```

create table instructor

```
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
salary numeric(8,2) check (salary > 29000),
primary key (ID),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
```

create table section

```
(course_id varchar(8),
sec_id varchar(8),
semester varchar(6)
check (semester in ('Fall', 'Winter', 'Spring',
'Summer')),
year numeric(4,0) check (year > 1701 and year
< 2100),
building varchar(15),
room_number varchar(7),
time_slot_id varchar(4),
primary key (course_id, sec_id, semester, year),
foreign key (course_id) references course
(course_id)
on delete cascade,
foreign key (building, room_number) references
classroom (building, room_number)
on delete set null
);
```

create table teaches

```
(ID varchar(5),
course_id varchar(8),
sec_id varchar(8),
semester varchar(6),
year numeric(4,0),
primary key (ID, course_id, sec_id, semester, year),
foreign key (course_id, sec_id, semester, year)
references section (course_id, sec_id, semester,
year)
on delete cascade,
foreign key (ID) references instructor (ID)
on delete cascade
);
```

create table student

```
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
tot_cred numeric(3,0) check (tot_cred >= 0),
primary key (ID),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
```

create table takes

```
(ID varchar(5),
course_id varchar(8),
sec_id varchar(8),
```

```

semester    varchar(6),
year        numeric(4,0),
grade       varchar(2),
primary key (ID, course_id, sec_id, semester, year),
foreign key (course_id, sec_id, semester, year)
  references section (course_id, sec_id, semester,
  year)
  on delete cascade,
foreign key (ID) references student (ID)
  on delete cascade
);
create table advisor
(s_ID       varchar(5),
i_ID       varchar(5),
primary key (s_ID),
foreign key (i_ID) references instructor (ID)
  on delete set null,
foreign key (s_ID) references student (ID)
  on delete cascade
);
create table time_slot
(time_slot_id varchar(4),
day         varchar(1),
start_hr    numeric(2) check (start_hr >= 0 and
start_hr < 24),
start_min   numeric(2) check (start_min >= 0 and
start_min < 60),
end_hr      numeric(2) check (end_hr >= 0 and
end_hr < 24),
end_min     numeric(2) check (end_min >= 0 and
end_min < 60),
primary key (time_slot_id, day, start_hr, start_min)
);
create table prereq
(course_id   varchar(8),
prereq_id   varchar(8),
primary key (course_id, prereq_id),
foreign key (course_id) references course
(course_id)
  on delete cascade,
foreign key (prereq_id) references course (course_id)
);

```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50 instructors, and 20 departments. From this, write an SQL query for each instructor, list their name, department, the courses they teach, the semester and year, and the number of students enrolled in each of their sections. Include only sections with at least 10 students.

Show more

Apr 30

**Claude responded: How it works:**

sql

```

SELECT
  i.name           AS
  instructor_name,
  i.dept_name,
  c.title         AS
  course_title,
  te.course_id,
  te.sec_id,
  te.semester,
  te.year,
  COUNT (ta.ID)  AS
  students_enrolled
FROM instructor i
JOIN teaches te  ON te.ID = i.ID

```

```

JOIN course c      ON c.course_id =
te.course_id
JOIN takes ta     ON ta.course_id =
te.course_id
AND ta.sec_id     =
te.sec_id
AND ta.semester   =
te.semester
AND ta.year       =
te.year
GROUP BY
  i.ID,
  i.name,
  i.dept_name,
  c.title,
  te.course_id,
  te.sec_id,
  te.semester,
  te.year
HAVING COUNT (ta.ID) >= 10
ORDER BY
  i.name,
  te.year DESC,
  te.semester,
  te.course_id;

```

How it works:

The query builds up in four logical layers:

Joins

- `instructor` → `teaches` maps each instructor to the sections they teach
- `teaches` → `course` pulls in the human-readable course title
- `teaches` → `takes` (on all four composite key columns) matches every student enrollment row to the correct section — joining on all four columns is critical here, since `(course_id, sec_id, semester, year)` together form the section's composite primary key; matching on fewer columns would fan out rows incorrectly

Grouping

`GROUP BY` includes `i.ID` in addition to `i.name` to correctly handle two instructors who might share the same name. All non-aggregated `SELECT` columns must appear here.

`HAVING` vs `WHERE`

`HAVING COUNT (ta.ID) >= 10` filters *after* aggregation, which is required when the filter condition depends on an aggregate value. A `WHERE` clause cannot reference `COUNT (ta.ID)` because row-level filtering happens before grouping.

Why `INNER` JOINS here (not `LEFT` JOINS)

Unlike a department reporting query where you want zero-enrollment rows, the requirement here is specifically sections *with enrollments*, so `INNER` JOINS are correct and also more efficient — the planner can prune non-matching rows earlier.

Ordering

Results are sorted by instructor name, then most recent year first, then semester, then course — a natural reading order for a teaching schedule report.

Performance note:

The most important index for this query is a composite one on `takes(course_id, sec_id, semester, year, ID)`, which covers the four-column join and lets `COUNT (ta.ID)` be resolved from the index alone. The existing primary key on `takes` already starts with `ID`, so a separate index with `course_id` leading would complement it well at this data scale.

**You said: Given the following PostgreSQL schema: create table classroom**

Given the following PostgreSQL schema: create table

```

classroom
  (building varchar(15),
  room_number varchar(7),
  capacity numeric(4,0),
  primary key (building, room_number)
);
create table department
  (dept_name varchar(20),
  building varchar(15),
  budget numeric(12,2) check (budget > 0),
  primary key (dept_name)
);
create table course
  (course_id varchar(8),
  title varchar(50),
  dept_name varchar(20),
  credits numeric(2,0) check (credits > 0),
  primary key (course_id),
  foreign key (dept_name) references department
  (dept_name)
  on delete set null
);
create table instructor
  (ID varchar(5),
  name varchar(20) not null,
  dept_name varchar(20),
  salary numeric(8,2) check (salary > 29000),
  primary key (ID),
  foreign key (dept_name) references department
  (dept_name)
  on delete set null
);
create table section
  (course_id varchar(8),
  sec_id varchar(8),
  semester varchar(6)
  check (semester in ('Fall', 'Winter', 'Spring',
  'Summer')),
  year numeric(4,0) check (year > 1701 and year
  < 2100),
  building varchar(15),
  room_number varchar(7),
  time_slot_id varchar(4),
  primary key (course_id, sec_id, semester, year),
  foreign key (course_id) references course
  (course_id)
  on delete cascade,
  foreign key (building, room_number) references
  classroom (building, room_number)
  on delete set null
);
create table teaches
  (ID varchar(5),
  course_id varchar(8),
  sec_id varchar(8),
  semester varchar(6),
  year numeric(4,0),
  primary key (ID, course_id, sec_id, semester, year),
  foreign key (course_id, sec_id, semester, year)
  references section (course_id, sec_id, semester,
  year)
  on delete cascade,
  foreign key (ID) references instructor (ID)
  on delete cascade
);
create table student

```

```

  (ID varchar(5),
  name varchar(20) not null,
  dept_name varchar(20),
  tot_cred numeric(3,0) check (tot_cred >= 0),
  primary key (ID),
  foreign key (dept_name) references department
  (dept_name)
  on delete set null
);
create table takes
  (ID varchar(5),
  course_id varchar(8),
  sec_id varchar(8),
  semester varchar(6),
  year numeric(4,0),
  grade varchar(2),
  primary key (ID, course_id, sec_id, semester, year),
  foreign key (course_id, sec_id, semester, year)
  references section (course_id, sec_id, semester,
  year)
  on delete cascade,
  foreign key (ID) references student (ID)
  on delete cascade
);
create table advisor
  (s_ID varchar(5),
  i_ID varchar(5),
  primary key (s_ID),
  foreign key (i_ID) references instructor (ID)
  on delete set null,
  foreign key (s_ID) references student (ID)
  on delete cascade
);
create table time_slot
  (time_slot_id varchar(4),
  day varchar(1),
  start_hr numeric(2) check (start_hr >= 0 and
  start_hr < 24),
  start_min numeric(2) check (start_min >= 0 and
  start_min < 60),
  end_hr numeric(2) check (end_hr >= 0 and
  end_hr < 24),
  end_min numeric(2) check (end_min >= 0 and
  end_min < 60),
  primary key (time_slot_id, day, start_hr, start_min)
);
create table prereq
  (course_id varchar(8),
  prereq_id varchar(8),
  primary key (course_id, prereq_id),
  foreign key (course_id) references course
  (course_id)
  on delete cascade,
  foreign key (prereq_id) references course (course_id)
);

```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50 instructors, and 20 departments. From this, write an SQL query to find all courses where every section ever offered had at least one student who received an 'A' grade.

Show more

Apr 30

**Claude responded: This is a classic "for all" problem in SQL, which has no direct universal quantifier — instead it's solved by double negation: "courses where there does NOT ex...**

This is a classic "for all" problem in SQL, which has no direct universal quantifier — instead it's solved by double negation:

"courses where there does NOT exist a section that has NO 'A' grade student."

```
sql
SELECT
  c.course_id,
  c.title,
  c.dept_name
FROM course c
WHERE
  -- Course must have been offered at
  -- least once
  EXISTS (
    SELECT 1
    FROM section s
    WHERE s.course_id = c.course_id
  )
  -- There must be no section of this
  -- course that lacks an 'A' student
  AND NOT EXISTS (
    SELECT 1
    FROM section s
    WHERE s.course_id = c.course_id
      AND NOT EXISTS (
        SELECT 1
        FROM takes ta
        WHERE ta.course_id =
          s.course_id
          AND ta.sec_id =
            s.sec_id
          AND ta.semester =
            s.semester
          AND ta.year =
            s.year
          AND ta.grade = 'A'
      )
    )
  )
ORDER BY
  c.dept_name,
  c.course_id;
```

The logic, layer by layer:

Outer **EXISTS** — guards against courses that exist in the **course** table but were never actually scheduled as a section. Without this, a course with no sections at all would vacuously satisfy the **NOT EXISTS** condition and appear in results, which would be misleading.

Outer **NOT EXISTS** — eliminates any course for which we can find even one offending section. It asks: "does a bad section exist?" If yes, the course is excluded.

Inner **NOT EXISTS** — defines what makes a section "bad": it has no row in **takes** where **grade = 'A'**. The four-column join (**course\_id**, **sec\_id**, **semester**, **year**) is necessary to pinpoint the exact section, since the same course can have multiple sections across different semesters and years.

The full double-negation chain reads as:

```
Keep course C where:
  C was offered at least once
  AND there is no section S of C
  such that no student in S received an
'A'
```

Why not use **GROUP BY / HAVING** here?

A **HAVING MIN(...)**  $\geq$  **threshold** approach works well for counting problems, but the predicate here is existential per section ("at least one A exists"), not a simple aggregate. Trying to flatten this into a single **GROUP BY** would conflate sections with each other and make it impossible to check the condition independently per section.

The correlated **NOT EXISTS** nesting is the cleaner and more correct fit.

A note on **grade = 'A'**

If your grading scheme includes 'A+' or 'A-' and those should also qualify, replace the equality with:

```
sql
AND ta.grade LIKE 'A%'
```

or use an explicit list:

```
sql
AND ta.grade IN ('A+', 'A', 'A-')
```

Performance note

With 30,000 rows in **takes** and 100 sections, the correlated subqueries are evaluated per course (200 iterations) and per section. An index on **takes(course\_id, sec\_id, semester, year, grade)** lets the innermost **NOT EXISTS** resolve with an index-only scan, keeping this efficient despite the nesting.

**You said: Given the following PostgreSQL schema: create table classroom**

Given the following PostgreSQL schema: create table classroom

```
(building varchar(15),
room_number varchar(7),
capacity numeric(4,0),
primary key (building, room_number)
);
```

create table department

```
(dept_name varchar(20),
building varchar(15),
budget numeric(12,2) check (budget > 0),
primary key (dept_name)
);
```

create table course

```
(course_id varchar(8),
title varchar(50),
dept_name varchar(20),
credits numeric(2,0) check (credits > 0),
primary key (course_id),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
```

create table instructor

```
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
salary numeric(8,2) check (salary > 29000),
primary key (ID),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
```

create table section

```
(course_id varchar(8),
sec_id varchar(8),
semester varchar(6)
check (semester in ('Fall', 'Winter', 'Spring',
'Summer')),
year numeric(4,0) check (year > 1701 and year
< 2100),
building varchar(15),
room_number varchar(7),
time_slot_id varchar(4),
primary key (course_id, sec_id, semester, year),
foreign key (course_id) references course
(course_id)
on delete cascade,
```

```

foreign key (building, room_number) references
  classroom (building, room_number)
  on delete set null
);
create table teaches
(ID      varchar(5),
course_id  varchar(8),
sec_id    varchar(8),
semester  varchar(6),
year      numeric(4,0),
primary key (ID, course_id, sec_id, semester, year),
foreign key (course_id, sec_id, semester, year)
  references section (course_id, sec_id, semester,
  year)
  on delete cascade,
foreign key (ID) references instructor (ID)
  on delete cascade
);
create table student
(ID      varchar(5),
name    varchar(20) not null,
dept_name  varchar(20),
tot_cred  numeric(3,0) check (tot_cred >= 0),
primary key (ID),
foreign key (dept_name) references department
  (dept_name)
  on delete set null
);
create table takes
(ID      varchar(5),
course_id  varchar(8),
sec_id    varchar(8),
semester  varchar(6),
year      numeric(4,0),
grade     varchar(2),
primary key (ID, course_id, sec_id, semester, year),
foreign key (course_id, sec_id, semester, year)
  references section (course_id, sec_id, semester,
  year)
  on delete cascade,
foreign key (ID) references student (ID)
  on delete cascade
);
create table advisor
(s_ID     varchar(5),
i_ID     varchar(5),
primary key (s_ID),
foreign key (i_ID) references instructor (ID)
  on delete set null,
foreign key (s_ID) references student (ID)
  on delete cascade
);
create table time_slot
(time_slot_id  varchar(4),
day           varchar(1),
start_hr      numeric(2) check (start_hr >= 0 and
start_hr < 24),
start_min     numeric(2) check (start_min >= 0 and
start_min < 60),
end_hr        numeric(2) check (end_hr >= 0 and
end_hr < 24),
end_min       numeric(2) check (end_min >= 0 and
end_min < 60),
primary key (time_slot_id, day, start_hr, start_min)
);
create table prereq
(course_id    varchar(8),

```

```

prereq_id    varchar(8),
primary key (course_id, prereq_id),
foreign key (course_id) references course
  (course_id)
  on delete cascade,
foreign key (prereq_id) references course (course_id)
);

```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50 instructors, and 20 departments. From this, write an SQL query for each department, ranking instructors by salary from highest to lowest. Show the instructor's name, department, salary, their rank within the department, and what percentile they fall into.

Show more

Apr 30

**Claude responded: ---**

sql

```

SELECT
  i.name
  AS instructor_name,
  i.dept_name,
  i.salary,
  RANK() OVER (
    PARTITION BY i.dept_name
    ORDER BY i.salary DESC
  )
  AS dept_salary_rank,
  DENSE_RANK() OVER (
    PARTITION BY i.dept_name
    ORDER BY i.salary DESC
  )
  AS dept_salary_dense_rank,
  COUNT(*) OVER (
    PARTITION BY i.dept_name
  )
  AS dept_instructor_count,
  ROUND(
    PERCENT_RANK() OVER (
      PARTITION BY i.dept_name
      ORDER BY i.salary ASC
    ) * 100, 1
  )
  AS salary_percentile,
  NTILE(4) OVER (
    PARTITION BY i.dept_name
    ORDER BY i.salary DESC
  )
  AS salary_quartile,
  ROUND(AVG(i.salary) OVER (
    PARTITION BY i.dept_name
  ), 2)
  AS dept_avg_salary,
  ROUND(
    (i.salary - AVG(i.salary) OVER
    (PARTITION BY i.dept_name))
    / NULLIF(STDDEV(i.salary) OVER
    (PARTITION BY i.dept_name), 0)
    , 2)
  AS salary_zscore
FROM instructor i
WHERE i.dept_name IS NOT NULL
ORDER BY
  i.dept_name,
  dept_salary_rank;

```

Column-by-column breakdown:

**RANK()** assigns salary position within the department, with gaps after ties. If two instructors share rank 2, the next rank

assigned is 4. This mirrors how athletic competitions are typically scored.

`DENSE_RANK()` also ranks by salary but without gaps — two tied instructors at rank 2 are followed by rank 3, not 4. Useful when you want to know how many distinct salary levels exist above someone.

`COUNT(*) OVER (PARTITION BY dept_name)` is a window aggregate, not a grouping — it attaches the department's total instructor headcount to every row without collapsing them, which is what makes window functions so powerful for reporting.

`PERCENT_RANK()` uses `ORDER BY salary ASC` intentionally. The formula is  $(rank - 1) / (total\ rows - 1)$ , producing a value from 0.0 to 1.0. With ascending order, the lowest earner sits at the 0th percentile and the highest at the 100th — the conventional interpretation of "what percentage of peers earn less than or equal to this salary."

`NTILE(4)` divides the department's instructors into four equal salary buckets (quartiles). Quartile 1 = top earners, quartile 4 = bottom earners, since the window orders by salary DESC. With small departments this is approximate — `NTILE` distributes remainder rows into the earliest buckets. `dept_avg_salary` and `salary_zscore` give distributional context. The Z-score measures how many standard deviations an instructor's salary sits above or below their department mean. `NULLIF(..., 0)` guards against division by zero in single-instructor departments where `STDDEV` returns 0.

Why `WHERE dept_name IS NOT NULL`

The `instructor.dept_name` column allows NULLs (via `ON DELETE SET NULL`). Without this filter, instructors with no department would all be grouped into a single NULL partition, which is technically valid SQL but almost certainly not meaningful for a per-department ranking report.

Key window function concept — `PARTITION BY` vs `GROUP BY`

	GROUP BY	PARTITION BY
Output rows	One per group	Same as input
Aggregates	Collapses rows	Attached to each row
Can mix row + aggregate	No	Yes

All five window functions here share the same logical scan of the `instructor` table — PostgreSQL is smart enough to compute them in a single pass rather than rescanning per function.

**You said: Given the following PostgreSQL schema: create table classroom**

```
Given the following PostgreSQL schema: create table
classroom
(building varchar(15),
room_number varchar(7),
capacity numeric(4,0),
```

```
primary key (building, room_number)
);
create table department
(dept_name varchar(20),
building varchar(15),
budget numeric(12,2) check (budget > 0),
primary key (dept_name)
);
create table course
(course_id varchar(8),
title varchar(50),
dept_name varchar(20),
credits numeric(2,0) check (credits > 0),
primary key (course_id),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table instructor
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
salary numeric(8,2) check (salary > 29000),
primary key (ID),
foreign key (dept_name) references department
(dept_name)
on delete set null
);
create table section
(course_id varchar(8),
sec_id varchar(8),
semester varchar(6)
check (semester in ('Fall', 'Winter', 'Spring',
'Summer')),
year numeric(4,0) check (year > 1701 and year
< 2100),
building varchar(15),
room_number varchar(7),
time_slot_id varchar(4),
primary key (course_id, sec_id, semester, year),
foreign key (course_id) references course
(course_id)
on delete cascade,
foreign key (building, room_number) references
classroom (building, room_number)
on delete set null
);
create table teaches
(ID varchar(5),
course_id varchar(8),
sec_id varchar(8),
semester varchar(6),
year numeric(4,0),
primary key (ID, course_id, sec_id, semester, year),
foreign key (course_id, sec_id, semester, year)
references section (course_id, sec_id, semester,
year)
on delete cascade,
foreign key (ID) references instructor (ID)
on delete cascade
);
create table student
(ID varchar(5),
name varchar(20) not null,
dept_name varchar(20),
tot_cred numeric(3,0) check (tot_cred >= 0),
primary key (ID),
```

```

foreign key (dept_name) references department
  (dept_name)
  on delete set null
);
create table takes
  (ID      varchar(5),
   course_id  varchar(8),
   sec_id    varchar(8),
   semester  varchar(6),
   year      numeric(4,0),
   grade     varchar(2),
   primary key (ID, course_id, sec_id, semester, year),
   foreign key (course_id, sec_id, semester, year)
     references section (course_id, sec_id, semester,
       year)
     on delete cascade,
   foreign key (ID) references student (ID)
     on delete cascade
);
create table advisor
  (s_ID     varchar(5),
   i_ID     varchar(5),
   primary key (s_ID),
   foreign key (i_ID) references instructor (ID)
     on delete set null,
   foreign key (s_ID) references student (ID)
     on delete cascade
);
create table time_slot
  (time_slot_id  varchar(4),
   day           varchar(1),
   start_hr      numeric(2) check (start_hr >= 0 and
     start_hr < 24),
   start_min     numeric(2) check (start_min >= 0 and
     start_min < 60),
   end_hr        numeric(2) check (end_hr >= 0 and
     end_hr < 24),
   end_min       numeric(2) check (end_min >= 0 and
     end_min < 60),
   primary key (time_slot_id, day, start_hr, start_min)
);
create table prereq
  (course_id  varchar(8),
   prereq_id  varchar(8),
   primary key (course_id, prereq_id),
   foreign key (course_id) references course
     (course_id)
     on delete cascade,
   foreign key (prereq_id) references course (course_id)
);

```

The database contains approximately 2000 students, 30,000 enrollment records in 'takes', 200 courses, 100 sections, 50 instructors, and 20 departments. From this, write an SQL query to find students who have taken courses offered by at least 3 different departments but have never taken a course taught by their own advisor.

Show more

Apr 30

**Claude responded: This query has two independent conditions that must both be true, so it's cleanest to solve each part separately in CTEs and then intersect them.**

This query has two independent conditions that must both be true, so it's cleanest to solve each part separately in CTEs and then intersect them.

sql

WITH

```

--
-----
-----
-- 1. Students who have taken courses
   from at least 3 departments
--
-----
-----
multi_dept_students AS (
  SELECT
    ta.ID
  FROM takes ta
  JOIN course c ON c.course_id =
    ta.course_id
  WHERE c.dept_name IS NOT NULL
  GROUP BY ta.ID
  HAVING COUNT(DISTINCT c.dept_name)
    >= 3
),
--
-----
-----
-- 2. All (student, instructor) pairs
   where the student has taken
   at least one section the
   instructor taught
--
-----
-----
student_took_from_instructor AS (
  SELECT DISTINCT
    ta.ID      AS student_id,
    te.ID      AS instructor_id
  FROM takes ta
  JOIN teaches te
    ON te.course_id =
    ta.course_id
    AND te.sec_id   = ta.sec_id
    AND te.semester =
    ta.semester
    AND te.year     = ta.year
),
--
-----
-----
-- 3. Students who have taken a course
   taught by their own advisor
--
-----
-----
took_from_advisor AS (
  SELECT
    a.s_ID     AS student_id
  FROM advisor a
  JOIN student_took_from_instructor
    stfi
    ON stfi.student_id =
    a.s_ID
    AND stfi.instructor_id =
    a.i_ID
)
--
-----
-----
-- Final: multi-department students who
   never took from advisor

```

```

--
-----
-----
SELECT
  s.ID,
  s.name
  AS student_name,
  s.dept_name
  AS student_dept,
  s.tot_cred,
  COUNT(DISTINCT c.dept_name)
  AS distinct_depts_taken,
  COUNT(DISTINCT ta.course_id)
  AS total_courses_taken,
  i.name
  AS advisor_name,
  i.dept_name
  AS advisor_dept
FROM student s
JOIN multi_dept_students mds ON
  mds.ID = s.ID
JOIN takes ta ON
  ta.ID = s.ID
JOIN course c ON
  c.course_id = ta.course_id
LEFT JOIN advisor a ON
  a.s_ID = s.ID
LEFT JOIN instructor i ON i.ID
  = a.i_ID
WHERE s.ID NOT IN (
  SELECT student_id FROM
  took_from_advisor
)
GROUP BY
  s.ID,
  s.name,
  s.dept_name,
  s.tot_cred,
  i.name,
  i.dept_name
ORDER BY
  distinct_depts_taken DESC,
  s.name;

```

#### CTE breakdown:

`multi_dept_students` groups `takes` by student and counts distinct `dept_name` values from the joined `course` table. The `HAVING` filter keeps only students meeting the  $\geq 3$  threshold. `WHERE c.dept_name IS NOT NULL` excludes orphaned courses whose department was deleted, since those carry no meaningful departmental identity.

`student_took_from_instructor` resolves the many-to-many relationship between students and the instructors whose sections they actually sat in. The four-column join on `(course_id, sec_id, semester, year)` is essential — without all four, a student who took *any* section of a course would incorrectly match an instructor who taught a *different* section of the same course in a different semester.

`took_from_advisor` applies the advisor constraint by joining `advisor` to the previous CTE. It only produces a row when the student's recorded advisor is also one of the instructors they learned from.

Final `SELECT` — why `NOT IN` over `NOT EXISTS` here  
Both are logically equivalent, but `NOT IN` reads naturally when the subquery is already a clean, pre-built CTE with no NULLs. Since `took_from_advisor.student_id` is derived from `advisor.s_ID` which references `student.ID` (a NOT NULL primary key), there is no NULL risk that would make `NOT IN` behave unexpectedly.

Why `LEFT JOIN` on `advisor` and `instructor` in the final `SELECT`

Students are not required to have an advisor — the `advisor` table has no mandatory entry per student. Using `LEFT JOIN` ensures students without advisors still appear in results. Their `advisor_name` and `advisor_dept` columns will show NULL, which is informative rather than silently dropping the row.

The NULL trap to be aware of

If a student has no advisor (`a.i_ID IS NULL`), the `took_from_advisor` CTE will never produce a row for them, so they naturally satisfy the "never took from advisor" condition and will appear in results — which is the correct interpretation, since the constraint is vacuously satisfied. If you want to exclude advisorless students from results, add this to the final `WHERE`:

```

sql
AND a.s_ID IS NOT NULL

```

Performance note

The most expensive operation is the four-way join in `student_took_from_instructor` across 30,000 `takes` rows. A composite index on `teaches(course_id, sec_id, semester, year, ID)` allows the join to be resolved efficiently. The `DISTINCT` in that CTE then deduplicates what is typically a small result set (~50 instructors  $\times$  bounded student overlap).