Database Design

```
philip306apply@cloudshell:~ (cinetravel)$ gcloud sql connect cinetravel --user=root
Allowlisting your IP for incoming connection for 5 minutes...done.
Connecting to database with SQL user [root].Enter password:
Welcome to the MySQL monitor.  Commands end with ; or \g.
Your MySQL connection id is 24
Server version: 8.0.37-google (Google)

Copyright (c) 2000, 2025, Oracle and/or its affiliates.

Oracle is a registered trademark of Oracle Corporation and/or its
affiliates. Other names may be trademarks of their respective
owners.

Type 'help;' or '\h' for help. Type '\c' to clear the current input statement.

mysql>
```

Part 1: 5 Tables Creation:

Table 1:

```
create table hotel (
        Id int primary key auto_increment,
        Hotel_Address varchar(255),
        Average_Score real,
        Hotel_Name varchar(255),
        Hotel_country varchar(255),
        City varchar(255)
);
```

```
mysql> select count(*) from hotel;
+----------+
| count(*) |
+----------+
|     1283 |
+----------+
1 row in set (0.02 sec)
```

Table 2:

```
create table movie_location (
        Continent varchar(255),
        Country varchar(255),
        Movie  varchar(255),
        Director varchar(255),
        Locations varchar(255),
        primary key(Continent, Country, Movie),
        foreign key (Movie) references movie(Title) on delete set null
);
```

```
mysql> select count(*) from movie_location;
+----------+
| count(*) |
+----------+
|     1348 |
+----------+
1 row in set (0.03 sec)
```

Table 3:

```
create table flight(
   Sched_dep_time varchar(255) not null,
   Sched_arr_time varchar(255) not null,
   Tailnum      varchar(50) primary key,
   Origin       varchar(10) not null,
   Dest varchar(10) not null,
   Time varchar(255) not null,
   Name varchar(255) not null,
   foreign key (Dest) references airport(Code) on delete cascade,
   foreign key (Origin) references airport(Code) on delete cascade
);
```

```
mysql> select count(*) from flight;
+----------+
| count(*) |
+----------+
|     4045 |
+----------+
1 row in set (0.02 sec)
```

Table 4:

```
create table airport(
   Code varchar(10) primary key,
   Name varchar(255) not null,
   Continent varchar(255),
   City varchar(255)
);
```

```
mysql> select count(code) from airport;
+-------------+
| count(code) |
+-------------+
|        9806 |
+-------------+
1 row in set (0.04 sec)
```

Table 5:

```
create table movie(
```

```
    Id_movies int primary key,
    Title varchar(255) not null,
    Vote_average real,
    Director_id  int,
    Year int,
    Month varchar(10),
    Day  varchar(20),
    Director_name varchar(100)
);
```

```
mysql> select count(id_movies) from movie;
+------------------+
| count(id_movies) |
+------------------+
|             1466 |
+------------------+
1 row in set (0.00 sec)
```

Part 2:

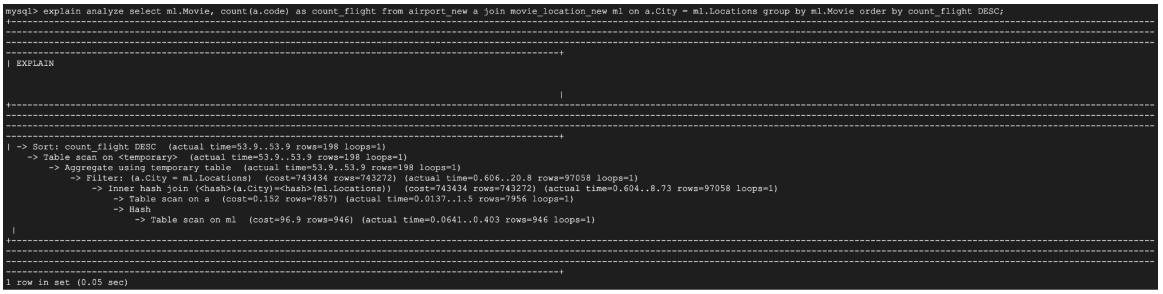
Advanced SQL queries 1

```
select ml.Movie, count(a.code) as count_flight
from airport_new a
join movie_location_new ml
        on a.City = ml.Locations
group by ml.Movie
order by count_flight DESC
limit 15;
```

Baseline performance:

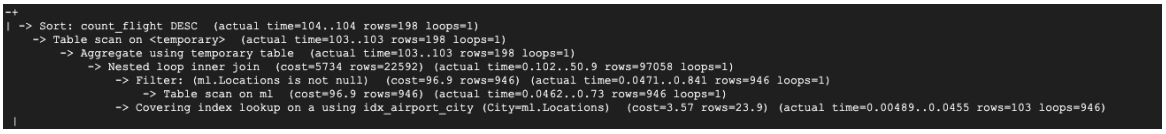
Total execution time: ~53.9 mseconds Estimated cost: 743434

```
mysql> select ml.Movie, count(a.code) as count_flight  from airport_new a  join
movie_location_new ml  on a.City = ml.Locations  group by ml.Movie  order by count_flight DESC limit 15;
+--------------------------+--------------+
| Movie                    | count_flight |
+--------------------------+--------------+
| region                   |         1326 |
| can case                 |         1326 |
| energy give treat        |         1326 |
| south opportunity health |         1326 |
| item                     |         1326 |
| cost item create         |         1326 |
| seat                     |         1326 |
| possible                 |          844 |
| tonight poor write       |          844 |
| through                  |          844 |
| by                       |          844 |
| away represent           |          844 |
| site actually            |          844 |
| town                     |          844 |
| or degree bag            |          844 |
+--------------------------+--------------+
15 rows in set (0.10 sec)
```



Index 1:

CREATE INDEX idx_airport_city ON airport_new(City);

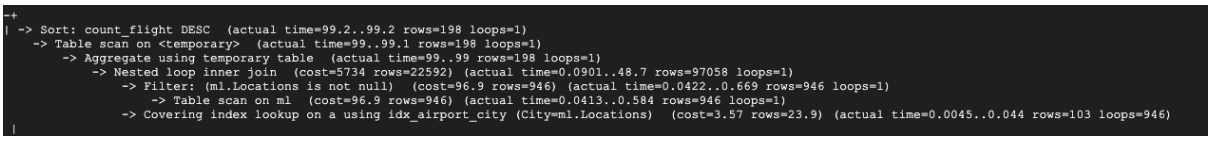


Total execution time: 104 mseconds Estimated cost: 5734

The query optimizer switched from a hash join to a **nested loop join with a covering index lookup** on the airport_new table. This transformation occurred after adding an index on the airport_new.City column (idx_airport_city), enabling more efficient row matching without scanning the entire table.

Index 2:

CREATE INDEX idx_location_city ON movie_location_new(Locations);



Total execution time: 99.2 mseconds Estimated cost: 5734

After adding an index on movie_location_new.Locations (idx_location_city), the query optimizer continued to use a **nested loop join**, but with greater efficiency. The database performed a **covering index lookup** on airport_new using the join condition City = Locations, as it did in the previous run with Index 1.

Index 3:

CREATE INDEX idx_movie_title ON movie_location_new(Movie);

```
> Sort: count_flight DESC  (actual time=101..101 rows=198 loops=1)
 -> Table scan on <temporary>  (actual time=101..101 rows=198 loops=1)
     -> Aggregate using temporary table  (actual time=101..101 rows=198 loops=1)
         -> Nested loop inner join  (cost=5734 rows=22592) (actual time=0.113..50.8 rows=97058 loops=1)
             -> Filter: (ml.Locations is not null)  (cost=96.9 rows=946) (actual time=0.0463..0.829 rows=946 loops=1)
                 -> Table scan on ml  (cost=96.9 rows=946) (actual time=0.0453..0.727 rows=946 loops=1)
             -> Covering index lookup on a using idx_airport_city (City=ml.Locations)  (cost=3.57 rows=23.9) (actual time=0.00485..0.045 rows=103 loops=946)
```

Total execution time:101 mseconds Estimated cost: 5734

After creating an index on movie_location_new.Movie (idx_movie_title), the query optimizer retained the **nested loop join** strategy and continued to use a **covering index lookup** on airport_new.City. However, there was **no major change in the join method or overall execution plan** compared to the previous configuration.

Therefore, we would use index 2.

Advanced SQL queries 2

```
SELECT a.city, COUNT(*) AS hotel_airport_count
FROM hotel_new h JOIN airport_new a
        ON h.City = a.city GROUP BY a.city
        ORDER BY hotel_airport_count DESC LIMIT 15;
```

```
\Database changed
mysql> SELECT a.city, COUNT(*) AS hotel_airport_count FROM hotel_new h JOIN airport_new a ON h.City = a.city GROUP BY a.city ORDER BY hotel_airport_count DESC LIMIT 15;
+---------+---------------------+
| city    | hotel_airport_count |
+---------+---------------------+
| london  |                7874 |
| paris   |                7812 |
| tokyo   |                7350 |
| berlin  |                7154 |
| toronto |                4588 |
| sydney  |                4582 |
| madrid  |                3864 |
| bangkok |                3650 |
| rome    |                3640 |
| nairobi |                2795 |
| cairo   |                1876 |
+---------+---------------------+
11 rows in set (0.06 sec)
```

output is less than 15 rows; 11 rows in set.

```
| EXPLAIN
        |
+----------------------------------------------------------------------------------------
-----------+
| -> Limit: 15 row(s)  (actual time=31.3..31.3 rows=11 loops=1)
    -> Sort: hotel_airport_count DESC, limit input to 15 row(s) per chunk  (actual time=31.3..31.3 rows=11 loops=1)
        -> Table scan on <temporary>  (actual time=31.3..31.3 rows=11 loops=1)
            -> Aggregate using temporary table  (actual time=31.3..31.3 rows=11 loops=1)
                -> Filter: (a.City = h.City)  (cost=785840 rows=785700) (actual time=0.531..12.8 rows=55185 loops=1)
                    -> Inner hash join (<hash>(a.City)=<hash>(h.City))  (cost=785840 rows=785700) (actual time=0.528..5.8 rows=55185 loops=1)
                        -> Table scan on a  (cost=0.117 rows=7857) (actual time=0.0162..1.68 rows=7956 loops=1)
                        -> Hash
                            -> Table scan on h  (cost=102 rows=1000) (actual time=0.0273..0.348 rows=1000 loops=1)
 |
+----------------------------------------------------------------------------------------
-----------+
1 row in set (0.04 sec)
```

index1:

```
CREATE INDEX idx_airport_city ON a(City);
```

```
mysql> EXPLAIN ANALYZE SELECT h.City, COUNT(*) AS hotel_airport_count FROM hotel h JOIN airport a ON h.City = a.City GROUP BY h.City ORDER BY hotel_airport_count DESC LIMIT 15;
| EXPLAIN
| -> Limit: 15 row(s)  (actual time=114..114 rows=5 loops=1)
    -> Sort: hotel_airport_count DESC, limit input to 15 row(s) per chunk  (actual time=114..114 rows=5 loops=1)
        -> Table scan on <temporary>  (actual time=114..114 rows=5 loops=1)
            -> Aggregate using temporary table  (actual time=114..114 rows=5 loops=1)
                -> Nested loop inner join  (cost=8211 rows=32243) (actual time=0.0639..57.5 rows=111117 loops=1)
                    -> Filter: (h.City is not null)  (cost=132 rows=1283) (actual time=0.0389..0.748 rows=1283 loops=1)
                        -> Table scan on h  (cost=132 rows=1283) (actual time=0.0379..0.625 rows=1283 loops=1)
                    -> Covering index lookup on a using idx_airport_city (city=h.City)  (cost=3.79 rows=25.1) (actual time=0.00922..0.0385 rows=86.6 loops=1283)
 |
1 row in set (0.12 sec)
```

Total execution time: 114 mseconds Estimated cost: 8211

Without idx_airport_city, the query would require full table scans on both tables, resulting in much higher I/O costs and execution time. After adding idx_airport_city, the query benefits from efficient index lookups when matching cities from the hotel table, dramatically reducing the number of rows that need to be examined in the airport table for each hotel city.

index 2:

CREATE INDEX idx_hotel_city ON h(City);

```
| EXPLAIN
| -> Limit: 15 row(s)  (actual time=130..130 rows=5 loops=1)
    -> Sort: hotel_airport_count DESC, limit input to 15 row(s) per chunk  (actual time=130..130 rows=5 loops=1)
        -> Table scan on <temporary>  (actual time=130..130 rows=5 loops=1)
            -> Aggregate using temporary table  (actual time=130..130 rows=5 loops=1)
                -> Nested loop inner join  (cost=313793 rows=1.73e+6) (actual time=0.204..76 rows=111117 loops=1)
                    -> Filter: (a.city is not null)  (cost=967 rows=9424) (actual time=0.038..3.97 rows=9806 loops=1)
                        -> Table scan on a  (cost=967 rows=9424) (actual time=0.0369..3.22 rows=9806 loops=1)
                    -> Covering index lookup on h using idx_hotel_city (City=a.city)  (cost=14.9 rows=183) (actual time=0.00231..0.00651 rows=11.3 loops=9806)
 |
1 row in set (0.13 sec)
```

Total execution time: 130 mseconds Estimated cost: 313793

reduced cost

With only idx_hotel_city but no index on airport.city, the query would benefit from efficient scanning of hotel cities but would still require a full table scan on the airport table for each matched city, potentially causing higher I/O costs when joining.

index 3: with both index

```
| EXPLAIN
| -> Limit: 15 row(s)  (actual time=75.3..75.3 rows=5 loops=1)
    -> Sort: hotel_airport_count DESC, limit input to 15 row(s) per chunk  (actual time=75.3..75.3 rows=5 loops=1)
        -> Stream results  (cost=11436 rows=7) (actual time=1.49..75.3 rows=5 loops=1)
            -> Group aggregate: count(0)  (cost=11436 rows=7) (actual time=1.49..75.3 rows=5 loops=1)
                -> Nested loop inner join  (cost=8211 rows=32243) (actual time=0.0648..55.7 rows=111117 loops=1)
                    -> Filter: (h.City is not null)  (cost=132 rows=1283) (actual time=0.0358..0.557 rows=1283 loops=1)
                        -> Covering index scan on h using idx_hotel_city  (cost=132 rows=1283) (actual time=0.0348..0.446 rows=1283 loops=1)
                    -> Covering index lookup on a using idx_airport_city (city=h.City)  (cost=3.79 rows=25.1) (actual time=0.00892..0.0374 rows=86.6 loops=1283)
 |
1 row in set (0.08 sec)
```

Total execution time: 75.3 mseconds Estimated cost: 8211

With both idx_airport_city and idx_hotel_city provide an additional time performance improvement by enabling a covering index scan on the hotel table rather than a full table scan, though interestingly, the optimizer's cost estimate remains unchanged at 8211 across all configurations.

Thus, we would select index 3 for advanced query 3 because it has the execution time and estimated cost.

Advanced SQL queries 3

```
SELECT f1.*
FROM flight f1
WHERE STR_TO_DATE(f1.sched_dep_time, '%H:%i') < (
  SELECT AVG(STR_TO_DATE(f2.sched_dep_time, '%H:%i'))
  FROM flight f2
  WHERE f2.origin = f1.origin
)
ORDER BY STR_TO_DATE(f1.sched_dep_time, '%H:%i')
LIMIT 15;
```

sched_dep_time	sched_arr_time	tailnum	origin	dest	time	name
	06:50	N186US	EWR	CLT	05:00	US Airways Inc.
	06:48	N435US	EWR	CLT	05:00	US Airways Inc.
	06:48	N566UW	EWR	CLT	05:00	US Airways Inc.
	06:50	N172US	EWR	CLT	05:00	US Airways Inc.
	06:48	N152UW	EWR	CLT	05:00	US Airways Inc.
05:15	08:08	N69806	EWR	IAH	05:15	United Air Lines Inc.
05:15	08:19	N14228	EWR	IAH	05:15	United Air Lines Inc.
05:15	08:08	N68802	EWR	IAH	05:15	United Air Lines Inc.
05:15	07:55	N653UA	EWR	IAH	05:15	United Air Lines Inc.
05:17	07:57	N556UA	EWR	IAH	05:17	United Air Lines Inc.
05:25	08:20	N78506	EWR	IAH	05:25	United Air Lines Inc.
05:25	08:20	N37277	EWR	IAH	05:25	United Air Lines Inc.
05:25	08:20	N33264	EWR	IAH	05:25	United Air Lines Inc.
05:29	08:30	N24211	LGA	IAH	05:29	United Air Lines Inc.
05:29	08:28	N493UA	LGA	IAH	05:29	United Air Lines Inc.

```
| -> Limit: 15 row(s)  (cost=437 rows=15) (actual time=4752..4752 rows=15 loops=1)
    -> Sort: str_to_date(f1.sched_dep_time,'%H:%i'), limit input to 15 row(s) per chunk  (cost=437 rows=4296) (actual time=4752..4752 rows=15 loops=1)
        -> Filter: (str_to_date(f1.sched_dep_time,'%H:%i') < (select #2))  (cost=437 rows=4296) (actual time=5.14..4751 rows=1948 loops=1)
            -> Table scan on f1  (cost=437 rows=4296) (actual time=0.0424..4.22 rows=4045 loops=1)
            -> Select #2 (subquery in condition; dependent)
                -> Aggregate: avg(str_to_date(f2.sched_dep_time,'%H:%i'))  (cost=93.7 rows=1) (actual time=1.17..1.17 rows=1 loops=4044)
                    -> Filter: (f2.origin = f1.origin)  (cost=50.7 rows=430) (actual time=0.00835..1.01 rows=1371 loops=4044)
                        -> Table scan on f2  (cost=50.7 rows=4296) (actual time=0.0079..0.701 rows=4045 loops=4044)
 |
```

Index 1:

CREATE INDEX idx_origin ON flight(origin);

Baseline Performance:

Estimated cost 437

This index on the origin column speeds up filtering in the subquery but doesn't lower the overall cost due to the overhead of time conversion

```
| -> Limit: 15 row(s)  (cost=437 rows=15) (actual time=5343..5343 rows=15 loops=1)
    -> Sort: str_to_date(f1.sched_dep_time,'%H:%i'), limit input to 15 row(s) per chunk  (cost=437 rows=4296) (actual time=5343..5343 rows=15 loops=1)
        -> Filter: (str_to_date(f1.sched_dep_time,'%H:%i') < (select #2))  (cost=437 rows=4296) (actual time=6.41..5341 rows=1948 loops=1)
            -> Table scan on f1  (cost=437 rows=4296) (actual time=0.0394..4.57 rows=4045 loops=1)
            -> Select #2 (subquery in condition; dependent)
                -> Aggregate: avg(str_to_date(f2.sched_dep_time,'%H:%i'))  (cost=238 rows=1) (actual time=1.32..1.32 rows=1 loops=4044)
                    -> Index lookup on f2 using idx_origin (origin=f1.origin)  (cost=131 rows=1074) (actual time=0.0475..1.15 rows=1371 loops=4044)
 |
```

Index 2:

CREATE INDEX idx_sched_dep_time ON flight(sched_dep_time);

Baseline Performance:

Estimated cost 437

Creating an index on sched_dep_time has no effect since the query's use of STR_TO_DATE() prevents the index from being utilized

```
| -> Limit: 15 row(s)  (cost=437 rows=15) (actual time=4785..4785 rows=15 loops=1)
    -> Sort: str_to_date(f1.sched_dep_time,'%H:%i'), limit input to 15 row(s) per chunk  (cost=437 rows=4296) (actual time=4785..4785 rows=15 loops=1)
        -> Filter: (str_to_date(f1.sched_dep_time,'%H:%i') < (select #2))  (cost=437 rows=4296) (actual time=5.18..4783 rows=1948 loops=1)
            -> Table scan on f1  (cost=437 rows=4296) (actual time=0.0669..4.29 rows=4045 loops=1)
            -> Select #2 (subquery in condition; dependent)
                -> Aggregate: avg(str_to_date(f2.sched_dep_time,'%H:%i'))  (cost=93.7 rows=1) (actual time=1.18..1.18 rows=1 loops=4044)
                    -> Filter: (f2.origin = f1.origin)  (cost=50.7 rows=430) (actual time=0.00839..1.01 rows=1371 loops=4044)
                        -> Table scan on f2  (cost=50.7 rows=4296) (actual time=0.00795..0.707 rows=4045 loops=4044)
|
```

Index 3:

```
CREATE INDEX idx_dep_time_conv ON flight (
  (IF(sched_dep_time COLLATE utf8mb4_0900_ai_ci REGEXP '^[0-9]{1,2}:[0-9]{2}$',
     STR_TO_DATE(sched_dep_time, '%H:%i'),
     NULL))
);
```

Baseline Performance:

Estimated cost 415

The functional index on the converted time (skipping invalid formats) effectively lowers the query cost by enabling index usage on valid time values

```
| -> Limit: 15 row(s)  (cost=415 rows=15) (actual time=7827..7827 rows=15 loops=1)
    -> Sort: str_to_date(f1.sched_dep_time,'%H:%i'), limit input to 15 row(s) per chunk  (cost=415 rows=4077) (actual time=7827..7827 rows=15 loops=1)
        -> Filter: (str_to_date(f1.sched_dep_time,'%H:%i') < (select #2))  (cost=415 rows=4077) (actual time=7.71..7823 rows=1948 loops=1)
            -> Table scan on f1  (cost=415 rows=4077) (actual time=0.0522..7.14 rows=4045 loops=1)
            -> Select #2 (subquery in condition; dependent)
                -> Aggregate: avg(str_to_date(f2.sched_dep_time,'%H:%i'))  (cost=89.3 rows=1) (actual time=1.93..1.93 rows=1 loops=4044)
                    -> Filter: (f2.origin = f1.origin)  (cost=48.5 rows=408) (actual time=0.0155..1.66 rows=1371 loops=4044)
                        -> Table scan on f2  (cost=48.5 rows=4077) (actual time=0.0148..1.2 rows=4045 loops=4044)
|
```

Thus, I would choose index 3 for advanced query 3 because it has the least estimated cost.

Advanced Query 4:

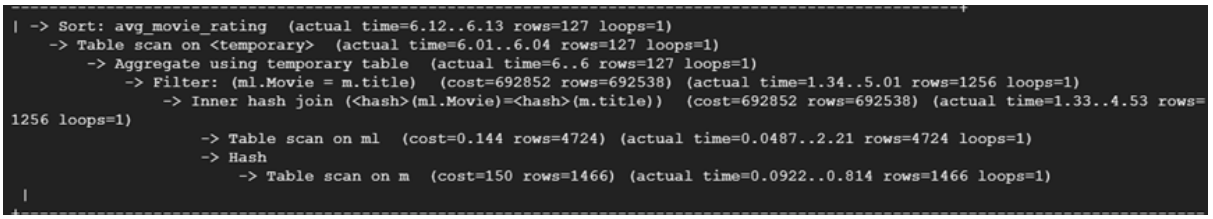
SQL:

```
SELECT LOWER(ml.Country) AS Country, ROUND(AVG(m.vote_average), 2) AS
avg_movie_rating FROM movie m JOIN movie_location_global ml ON m.title = ml.Movie
GROUP BY LOWER(ml.Country) ORDER BY avg_movie_rating ASC;
```

```
mysql> SELECT      LOWER(ml.Country) AS Country,      ROUND(AVG(m.vote_average), 2) AS avg_movie_rating FROM movie m JOIN movie
_location_global ml      ON m.title = ml.Movie GROUP BY LOWER(ml.Country) ORDER BY avg_movie_rating ASC limit 15;
+---------------+------------------+
| Country       | avg_movie_rating |
+---------------+------------------+
| peru          |              5.7 |
| chile         |              5.8 |
| cambodia      |              5.9 |
| brazil        |              5.9 |
| azerbaijan    |                6 |
| bulgaria      |              6.1 |
| panama        |              6.1 |
| tahiti        |             6.15 |
| greece        |             6.15 |
| gibraltar     |              6.2 |
| queensland    |              6.2 |
| kenya         |             6.25 |
| slovenia      |              6.3 |
| west virginia |              6.3 |
| austria       |             6.33 |
+---------------+------------------+
15 rows in set (0.02 sec)
```

Baseline performance:

Total execution time: ~6.12 mseconds Estimated cost: 692,852



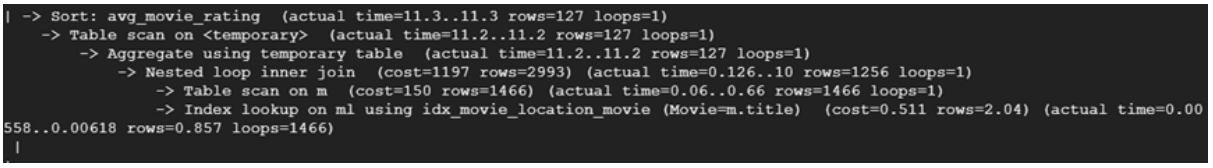
Index 1:

CREATE INDEX idx_movie_title ON movie(title);

CREATE INDEX idx_movie_location_movie ON movie_location_global(Movie);

Total execution time: ~11.3 mseconds Estimated cost: 1,197

The query optimizer switched from a hash join to a nested loop join with an index lookup on the movie_location_global table. The estimated cost decreased dramatically (from 692,852 to 1,197).

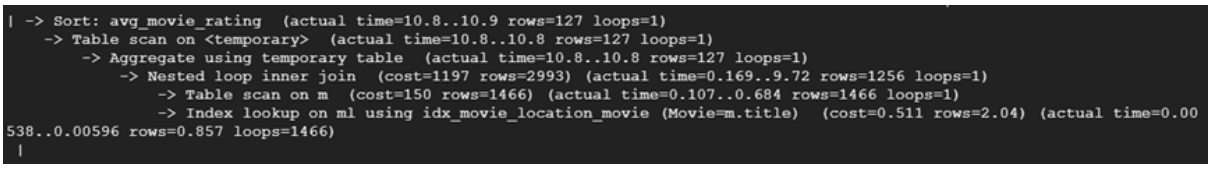


Index 2:

CREATE INDEX idx_movie_location_country ON movie_location_global(Country);

Total execution time: ~10.8 mseconds Estimated cost: 1197

Adding an index on the GROUP BY column (Country) didn't significantly change the execution plan compared to Option 1. The query optimizer still used the index on the Movie column for the JOIN operation and didn't utilize the Country index for the GROUP BY operation. This is likely because the grouping happens after the join, and the optimizer determined that using the Movie index for the join was more beneficial.



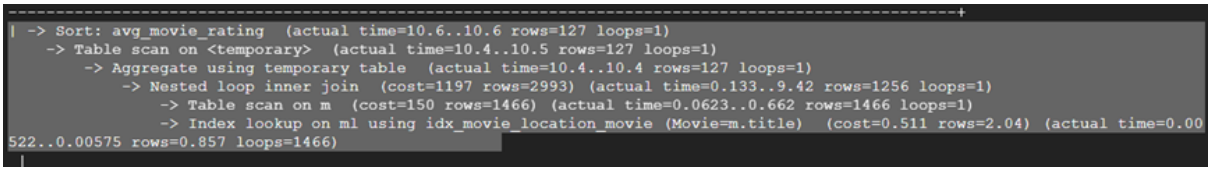
Index 3:

CREATE INDEX idx_movie_location_composite ON movie_location_global(Movie, Country);

CREATE INDEX idx_movie_title ON movie(title);

Total execution time: ~10.6 mseconds Estimated cost: 1197

The execution plan for the composite index strategy looks very similar to the previous options. The optimizer is still using the index on Movie for the JOIN operation and not taking advantage of the composite index for the GROUP BY operation.



We will use index1 as the final index selection since they look quite the same and the logic of the first one is quite simpler.