بسم الله الرحمن الرحيم

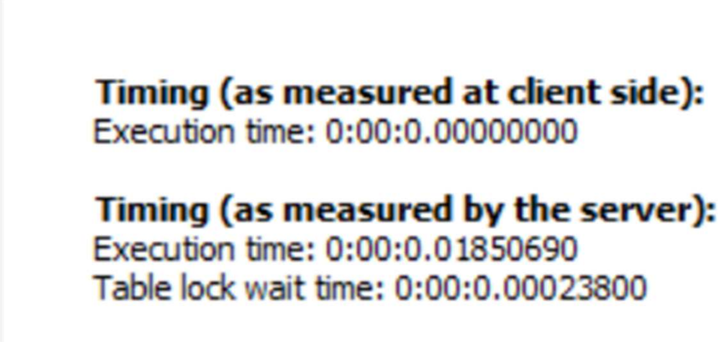
فارس الدباسي

تركي المهيني

IMDB data was chosen, and these queries has used (before indexing and portioning), the time and execution plan for each query below it:

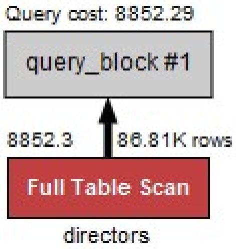
These are queries on directors table before portioning:

select * from directors where id>=100;



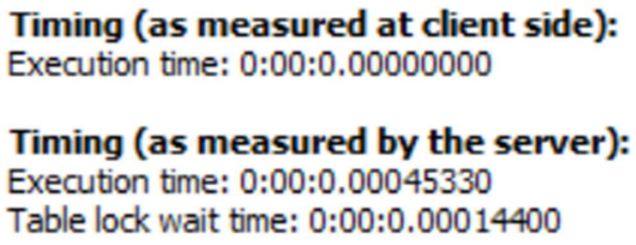
Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.01850690
Table lock wait time: 0:00:0.00023800

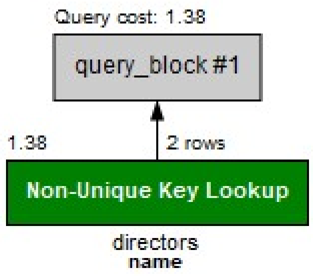


insert into directors (first_name) VALUES ("OMARALOBAID");

select * from directors where first_name="OMARALOBAID;"

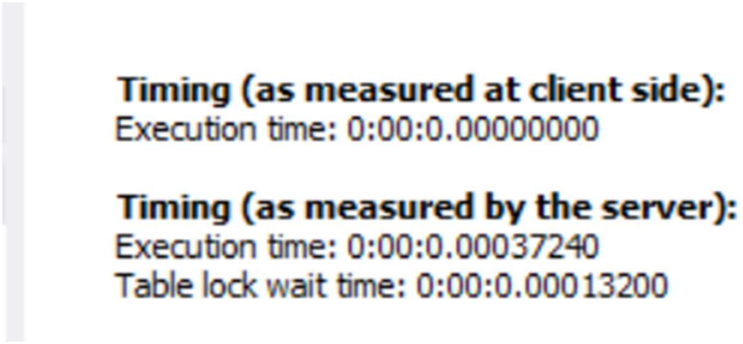


```
Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00045330
Table lock wait time: 0:00:0.00014400
```

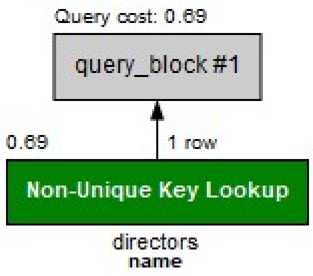


delete from directors where first_name="OMARALOBAID;"

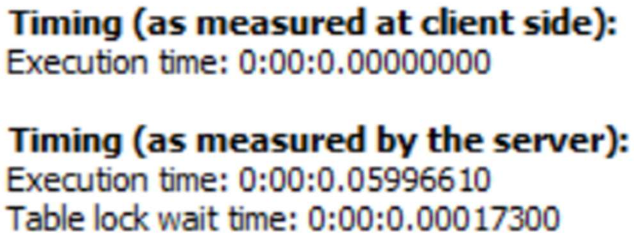
select * from directors where first_name="OMARALOBAID;"



```
Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00037240
Table lock wait time: 0:00:0.00013200
```

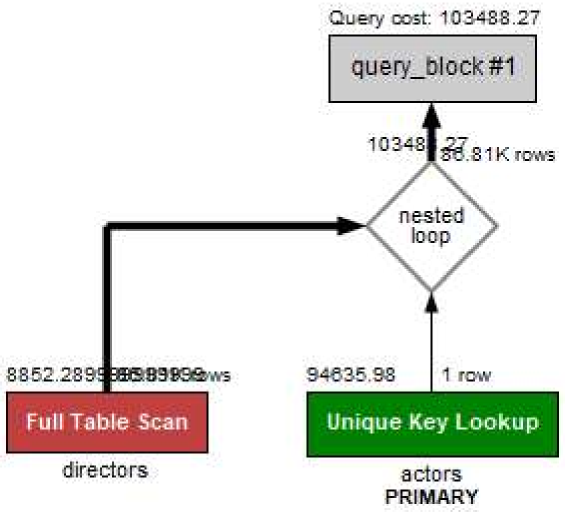


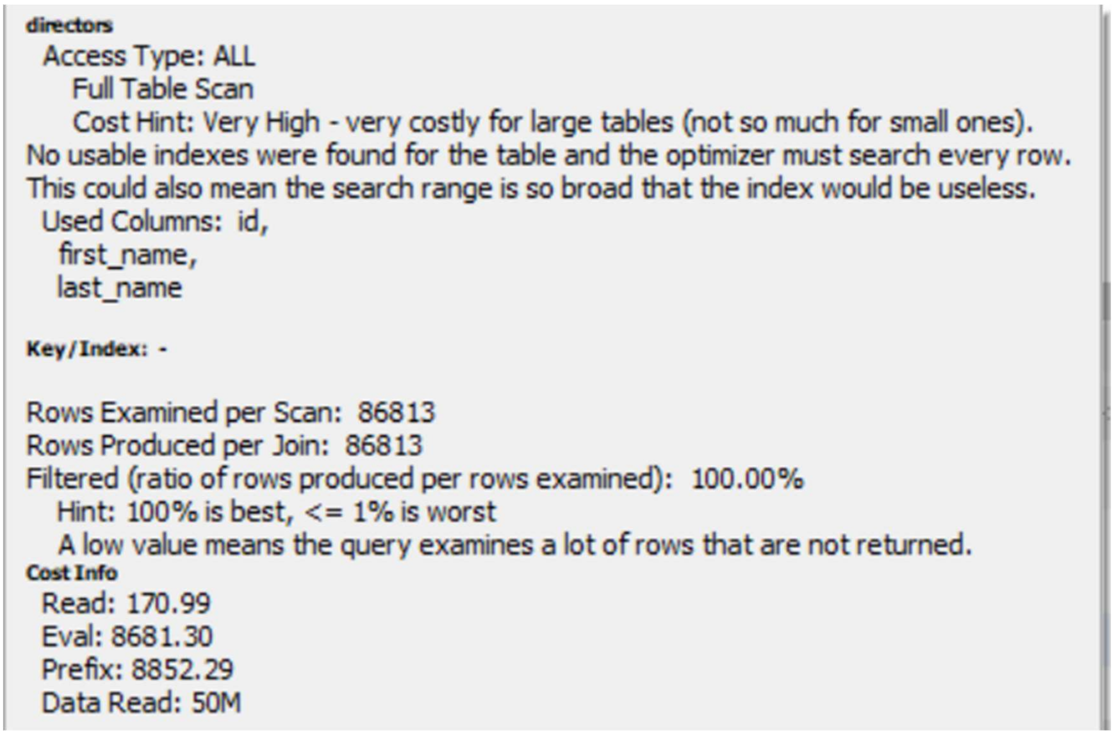
select * from directors INNER JOIN actors on actors.id=directors.id;



Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.05996610
Table lock wait time: 0:00:0.00017300





directors
Access Type: ALL
Full Table Scan
Cost Hint: Very High - very costly for large tables (not so much for small ones).
No usable indexes were found for the table and the optimizer must search every row.
This could also mean the search range is so broad that the index would be useless.
Used Columns: id,
first_name,
last_name

Key/Index: -

Rows Examined per Scan: 86813
Rows Produced per Join: 86813
Filtered (ratio of rows produced per rows examined): 100.00%
Hint: 100% is best, <= 1% is worst
A low value means the query examines a lot of rows that are not returned.
Cost Info
Read: 170.99
Eval: 8681.30
Prefix: 8852.29
Data Read: 50M

This is queries on directors table after portioning:

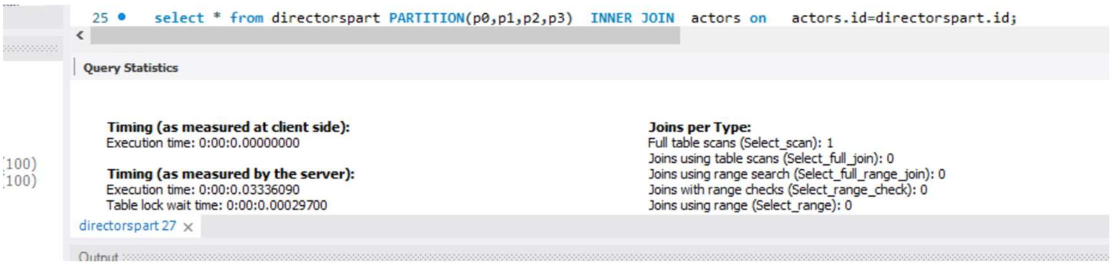
(The new table was created from directors table and takes all its data to use portioning and indexing to compare it with the pervious results, since we had problem to do it on the original):

create table directorspart like directors;

insert into directorspart select * from directors;

```
ALTER TABLE directorspart
      PARTITION BY RANGE(id)  (
       PARTITION p0 VALUES LESS THAN (100),
       PARTITION p1 VALUES LESS THAN (200),
       PARTITION p2 VALUES LESS THAN (500),
       PARTITION p3 VALUES LESS THAN MAXVALUE
);
```

select * from directorspart PARTITION(p0,p1,p2,p3) where id>=100;



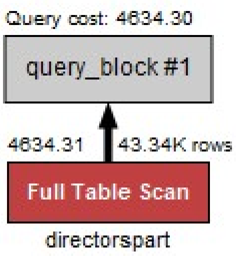
```
select * from directorspart PARTITION(p0,p1,p2,p3)  INNER JOIN  actors on   actors.id=directorspart.id;
```

Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.03336090
Table lock wait time: 0:00:0.00029700

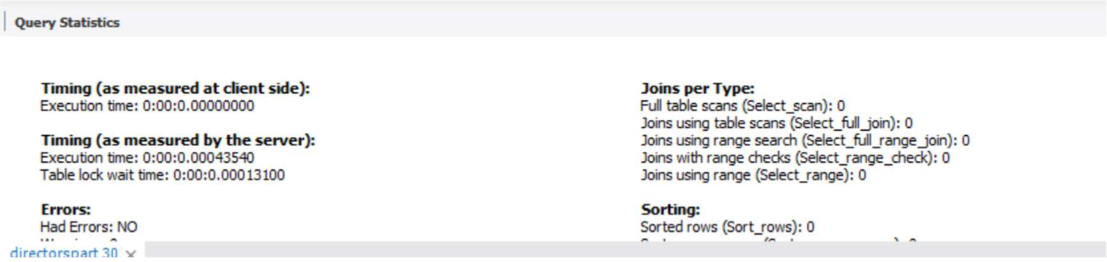
Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0



# Decrease the cost to nearly half.

insert into directorspart (first_name) VALUES ("OMARALOBAID");

select * from directorspart PARTITION(p0,p1,p2,p3) where first_name="OMARALOBAID";



Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00043540
Table lock wait time: 0:00:0.00013100

Errors:
Had Errors: NO

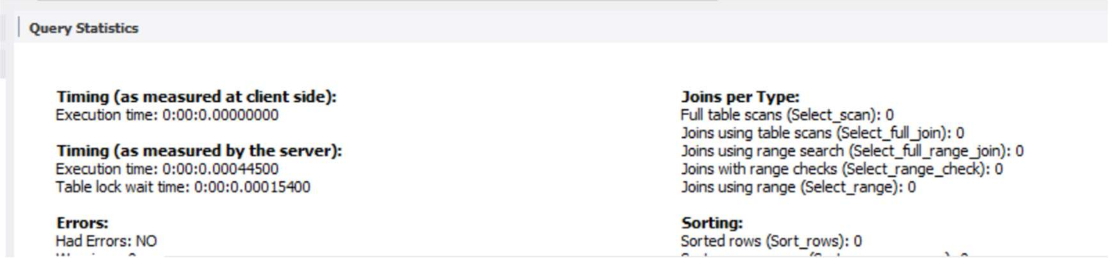
Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0

# There is not a big difference except time increases.

delete from directorspart where first_name="OMARALOBAID";

select * from directorspart PARTITION(p0,p1,p2,p3) where first_name="OMARALOBAID";



Timing (as measured at client side):
Execution time: 0:00:0.00000000

Timing (as measured by the server):
Execution time: 0:00:0.00044500
Table lock wait time: 0:00:0.00015400

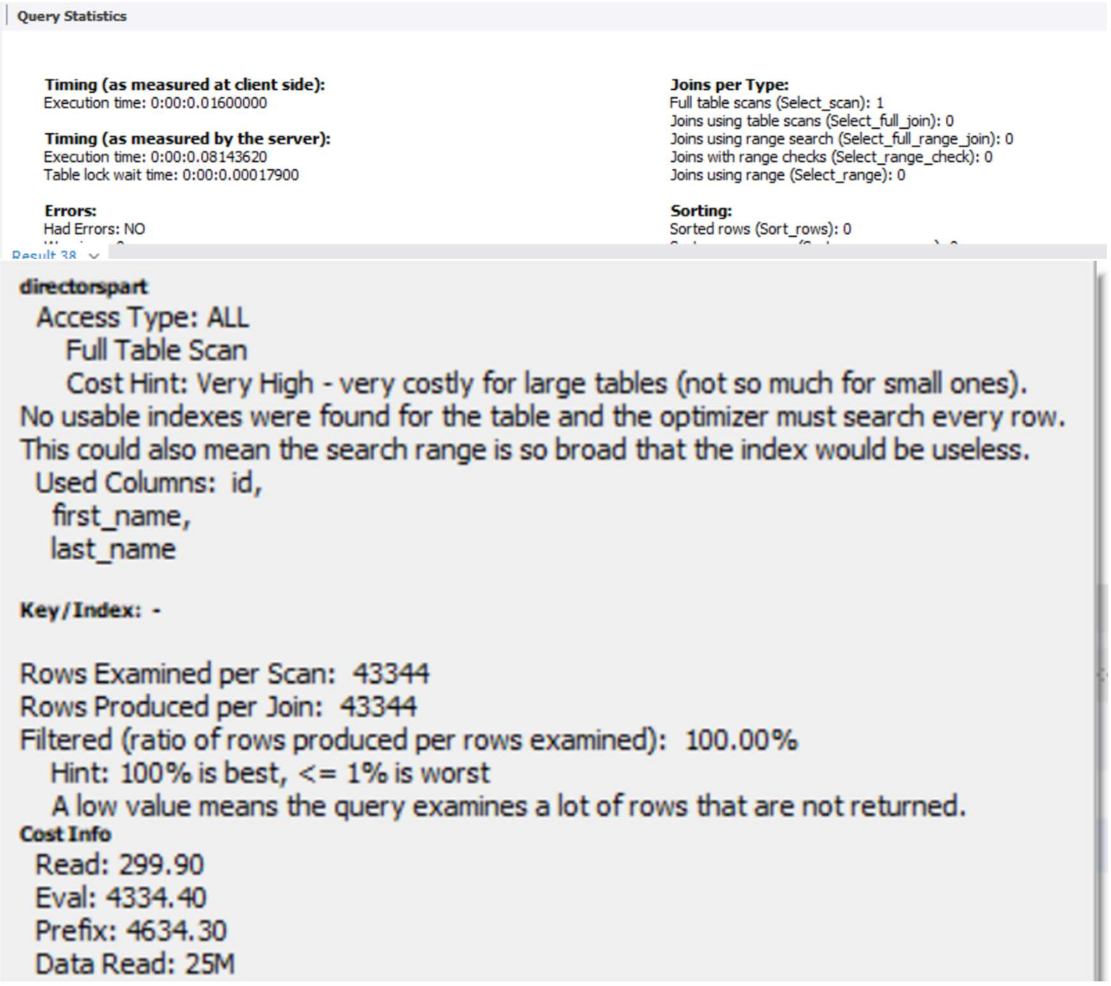
Errors:
Had Errors: NO

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0

# There is not a big difference except time increases.

select * from directorspart PARTITION(p0,p1,p2,p3) INNER JOIN actors on actors.id=directorspart.id;



Query Statistics

Timing (as measured at client side):
Execution time: 0:00:0.01600000

Timing (as measured by the server):
Execution time: 0:00:0.08143620
Table lock wait time: 0:00:0.00017900

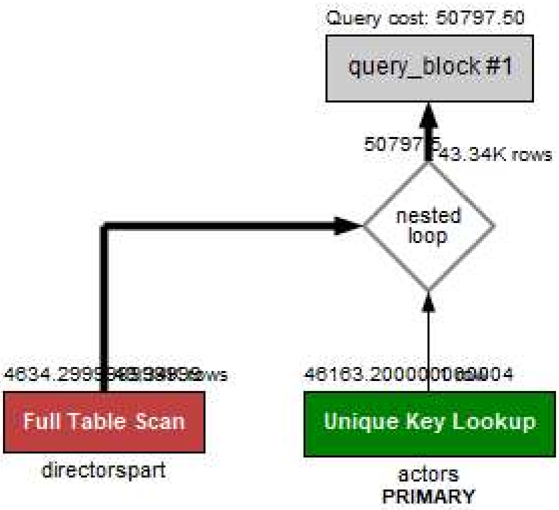
Errors:
Had Errors: NO

Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0

Result 38

```
directorspart
  Access Type: ALL
     Full Table Scan
     Cost Hint: Very High - very costly for large tables (not so much for small ones).
No usable indexes were found for the table and the optimizer must search every row.
This could also mean the search range is so broad that the index would be useless.
  Used Columns:  id,
    first_name,
    last_name

Key/Index:  -

Rows Examined per Scan:  43344
Rows Produced per Join:  43344
Filtered (ratio of rows produced per rows examined):  100.00%
    Hint:  100% is best, <= 1% is worst
    A low value means the query examines a lot of rows that are not returned.
Cost Info
  Read: 299.90
  Eval: 4334.40
  Prefix: 4634.30
  Data Read: 25M
```



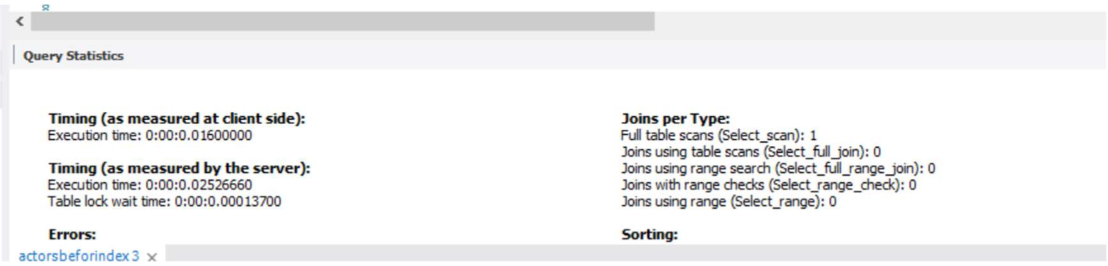
# Decrease the cost and data read to nearly half, but time increases from 0.05 to 0.08.

In conclusion:

Portioning decreases the cost and data that read to nearly half, but the overhead is in increasing the time.

This is queries on actors table before indexing:

select * from actorsbeforindex where gender="m";



Query Statistics

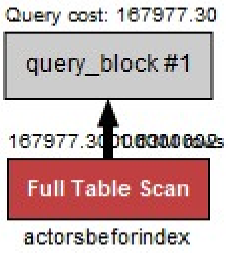
Timing (as measured at client side):
Execution time: 0:00:0.01600000

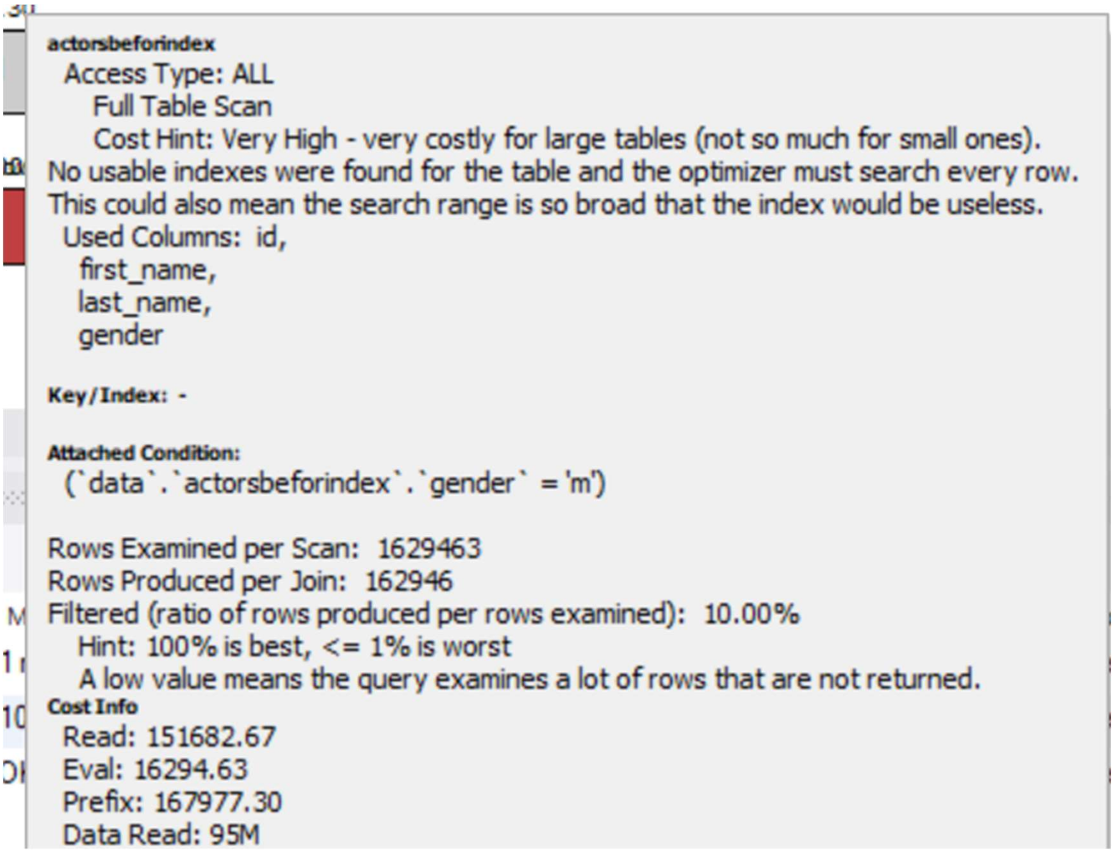
Timing (as measured by the server):
Execution time: 0:00:0.02526660
Table lock wait time: 0:00:0.00013700

Errors:

Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:

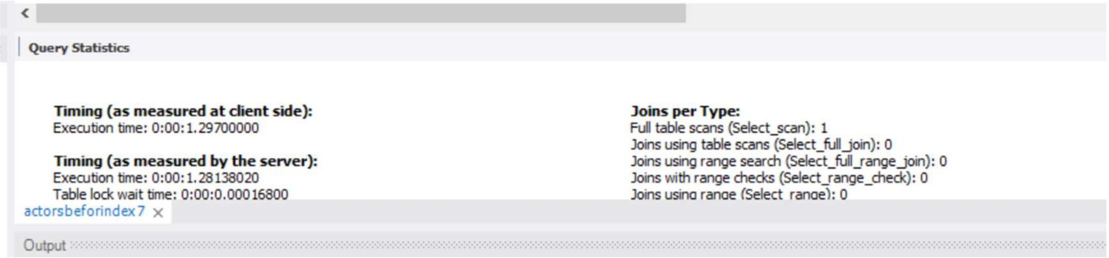




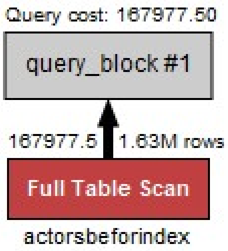
```
actorsbeforindex
  Access Type: ALL
    Full Table Scan
    Cost Hint: Very High - very costly for large tables (not so much for small ones).
No usable indexes were found for the table and the optimizer must search every row.
This could also mean the search range is so broad that the index would be useless.
  Used Columns: id,
    first_name,
    last_name,
    gender

Key/Index: -

Attached Condition:
  (`data`.`actorsbeforindex`.`gender` = 'm')

Rows Examined per Scan:  1629463
Rows Produced per Join:  162946
Filtered (ratio of rows produced per rows examined):  10.00%
    Hint: 100% is best, <= 1% is worst
    A low value means the query examines a lot of rows that are not returned.
Cost Info
  Read: 151682.67
  Eval: 16294.63
  Prefix: 167977.30
  Data Read: 95M
```

insert into actorsbeforindex (last_name) VALUES ("faresaldebasi");

select * from actorsbeforindex where last_name="faresaldebasi";

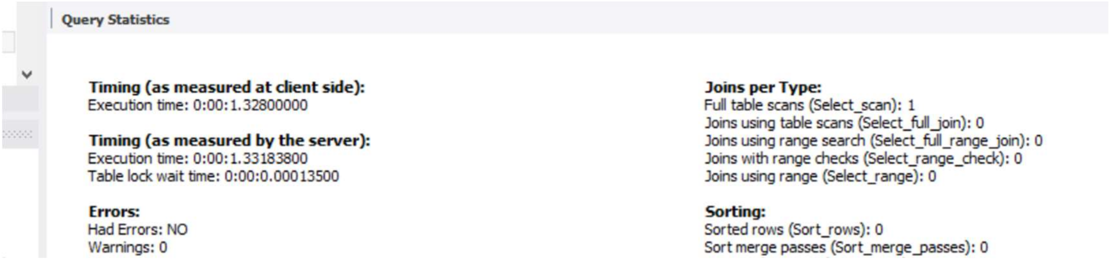


```
Query Statistics

Timing (as measured at client side):                Joins per Type:
Execution time: 0:00:1.29700000                     Full table scans (Select_scan): 1
                                                    Joins using table scans (Select_full_join): 0
Timing (as measured by the server):                 Joins using range search (Select_full_range_join): 0
Execution time: 0:00:1.28138020                     Joins with range checks (Select_range_check): 0
Table lock wait time: 0:00:0.00016800               Joins using range (Select_range): 0
```



delete from actorsbeforindex where last_name="faresaldebasi";

select * from actorsbeforindex where last_name="faresaldebasi";



Query Statistics

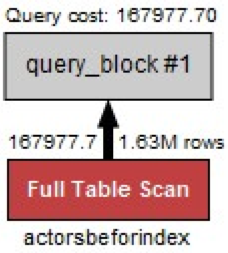
Timing (as measured at client side):
Execution time: 0:00:1.32800000

Timing (as measured by the server):
Execution time: 0:00:1.33183800
Table lock wait time: 0:00:0.00013500

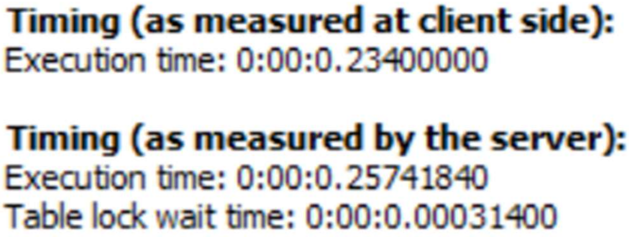
Errors:
Had Errors: NO
Warnings: 0

Joins per Type:
Full table scans (Select_scan): 1
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0

Sorting:
Sorted rows (Sort_rows): 0
Sort merge passes (Sort_merge_passes): 0

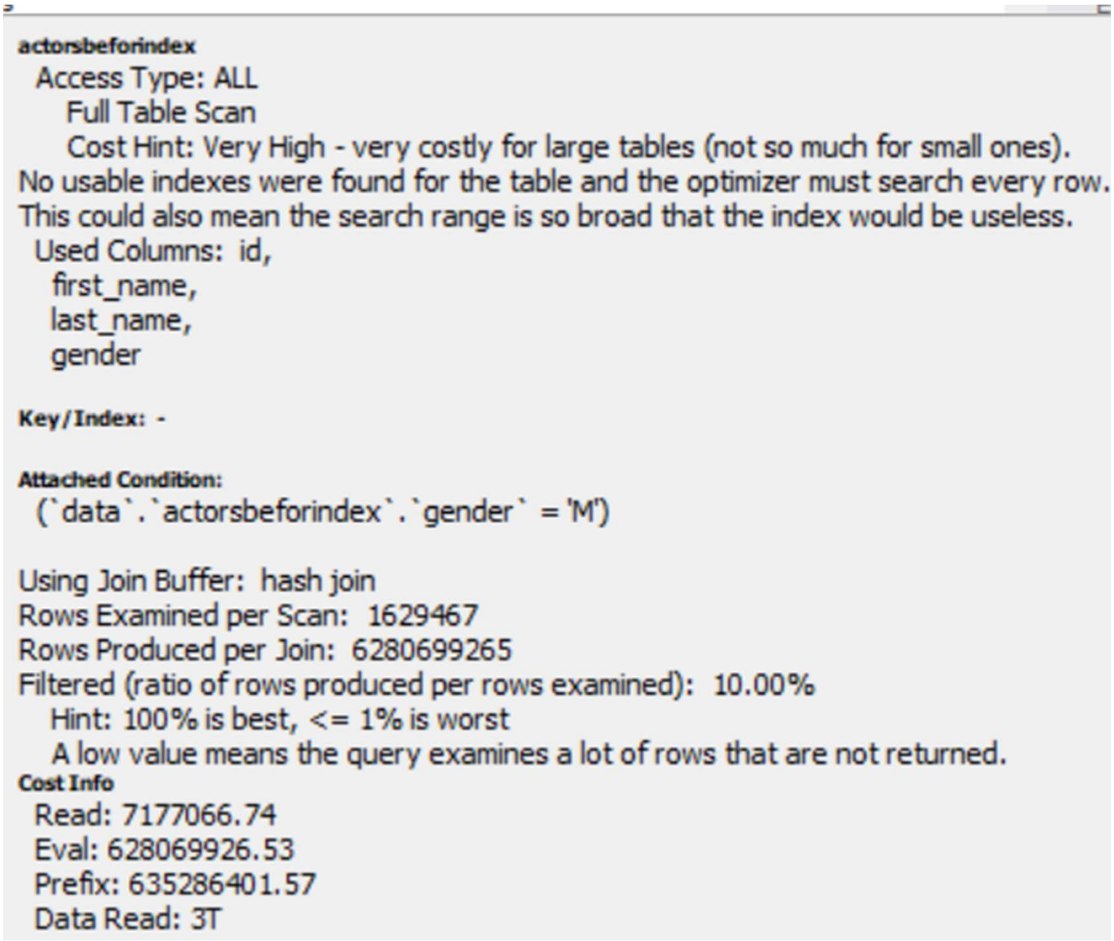


select * from actorsbeforindex INNER JOIN movies on movies.year=2000 and actorsbeforindex.gender='M';

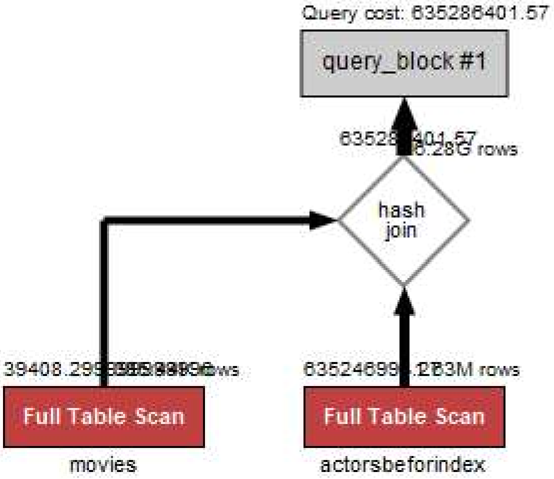


Timing (as measured at client side):
Execution time: 0:00:0.23400000

Timing (as measured by the server):
Execution time: 0:00:0.25741840
Table lock wait time: 0:00:0.00031400



```
actorsbeforindex
  Access Type: ALL
     Full Table Scan
     Cost Hint: Very High - very costly for large tables (not so much for small ones).
No usable indexes were found for the table and the optimizer must search every row.
This could also mean the search range is so broad that the index would be useless.
  Used Columns:  id,
    first_name,
    last_name,
    gender

Key/Index: -

Attached Condition:
  (`data`.`actorsbeforindex`.`gender` = 'M')

Using Join Buffer:  hash join
Rows Examined per Scan:  1629467
Rows Produced per Join:  6280699265
Filtered (ratio of rows produced per rows examined):  10.00%
    Hint:  100% is best, <= 1% is worst
    A low value means the query examines a lot of rows that are not returned.
Cost Info
  Read: 7177066.74
  Eval: 628069926.53
  Prefix: 635286401.57
  Data Read: 3T
```

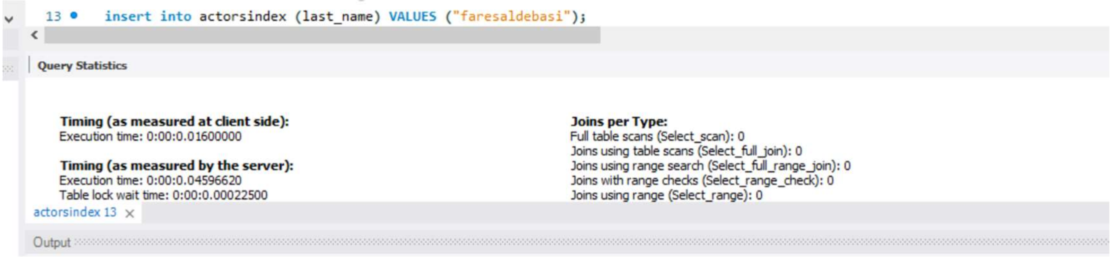


After indexing:

(new table named actorsindex was created with the same actorsbeforeindex data to make the compare clearer):

CREATE INDEX idindex ON actorsindex (id);

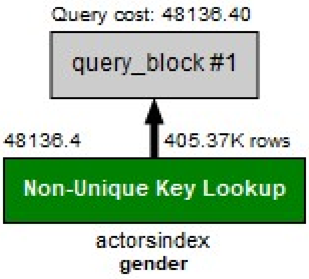
select * from actorsindex where gender="m";



```
insert into actorsindex (last_name) VALUES ("faresaldebasi");
```

Query Statistics

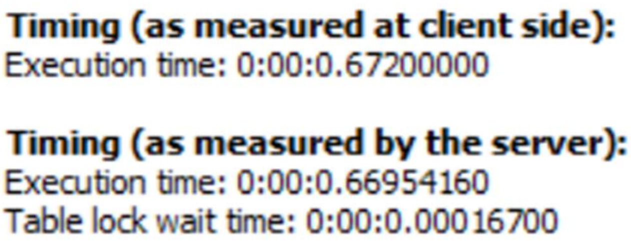
```
Timing (as measured at client side):
Execution time: 0:00:0.01600000

Timing (as measured by the server):
Execution time: 0:00:0.04596620
Table lock wait time: 0:00:0.00022500

Joins per Type:
Full table scans (Select_scan): 0
Joins using table scans (Select_full_join): 0
Joins using range search (Select_full_range_join): 0
Joins with range checks (Select_range_check): 0
Joins using range (Select_range): 0
```



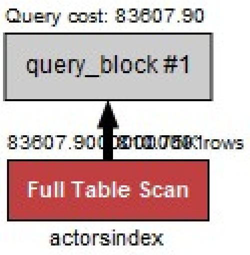
# Decrease the cost from 167977 to 48136

insert into actorsindex (last_name) VALUES ("faresaldebasi");

select * from actorsindex where last_name="faresaldebasi";



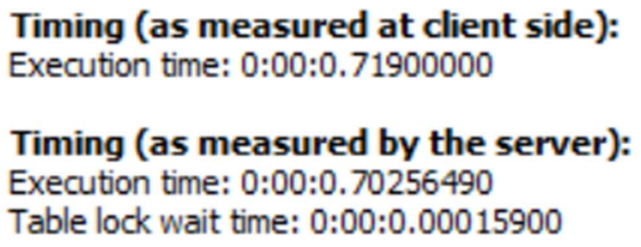
```
Timing (as measured at client side):
Execution time: 0:00:0.67200000

Timing (as measured by the server):
Execution time: 0:00:0.66954160
Table lock wait time: 0:00:0.00016700
```



# Decrease the cost from 169779 to 83608 and the time from 1.2 to 0.6

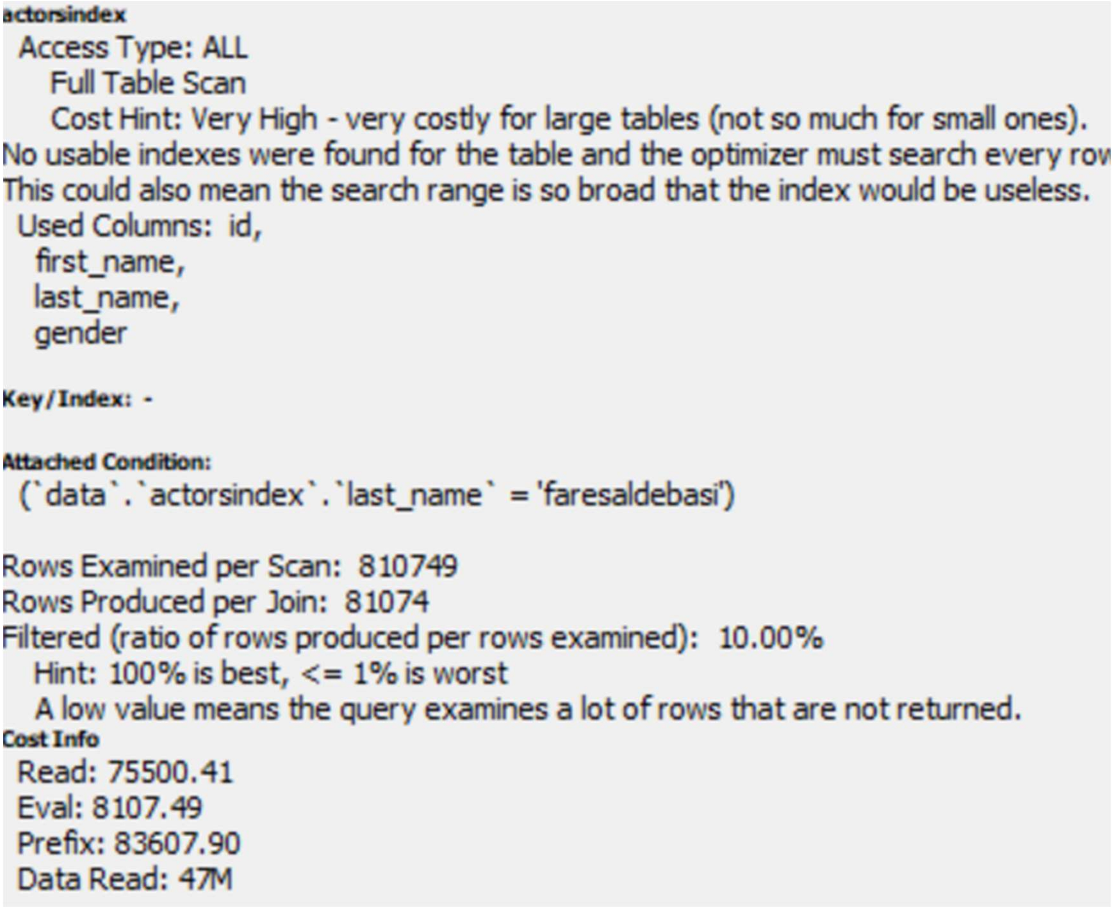
delete from actorsindex where last_name="faresaldebasi";

select * from actorsindex where last_name="faresaldebasi";

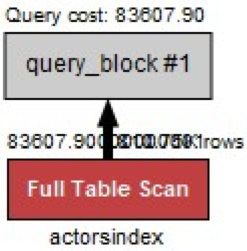


Timing (as measured at client side):
Execution time: 0:00:0.71900000

Timing (as measured by the server):
Execution time: 0:00:0.70256490
Table lock wait time: 0:00:0.00015900

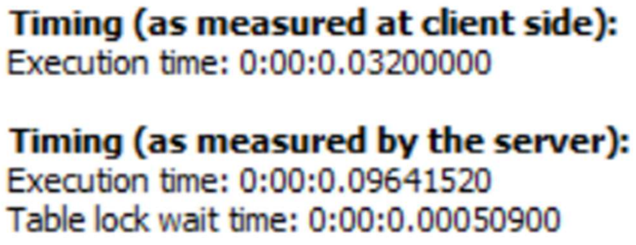


```
actorsindex
 Access Type: ALL
    Full Table Scan
    Cost Hint: Very High - very costly for large tables (not so much for small ones).
No usable indexes were found for the table and the optimizer must search every ro
This could also mean the search range is so broad that the index would be useless.
 Used Columns:  id,
   first_name,
   last_name,
   gender

Key/Index: -

Attached Condition:
 (`data`.`actorsindex`.`last_name` = 'faresaldebasi')

Rows Examined per Scan:  810749
Rows Produced per Join:  81074
Filtered (ratio of rows produced per rows examined):  10.00%
   Hint: 100% is best, <= 1% is worst
   A low value means the query examines a lot of rows that are not returned.
Cost Info
 Read: 75500.41
 Eval: 8107.49
 Prefix: 83607.90
 Data Read: 47M
```



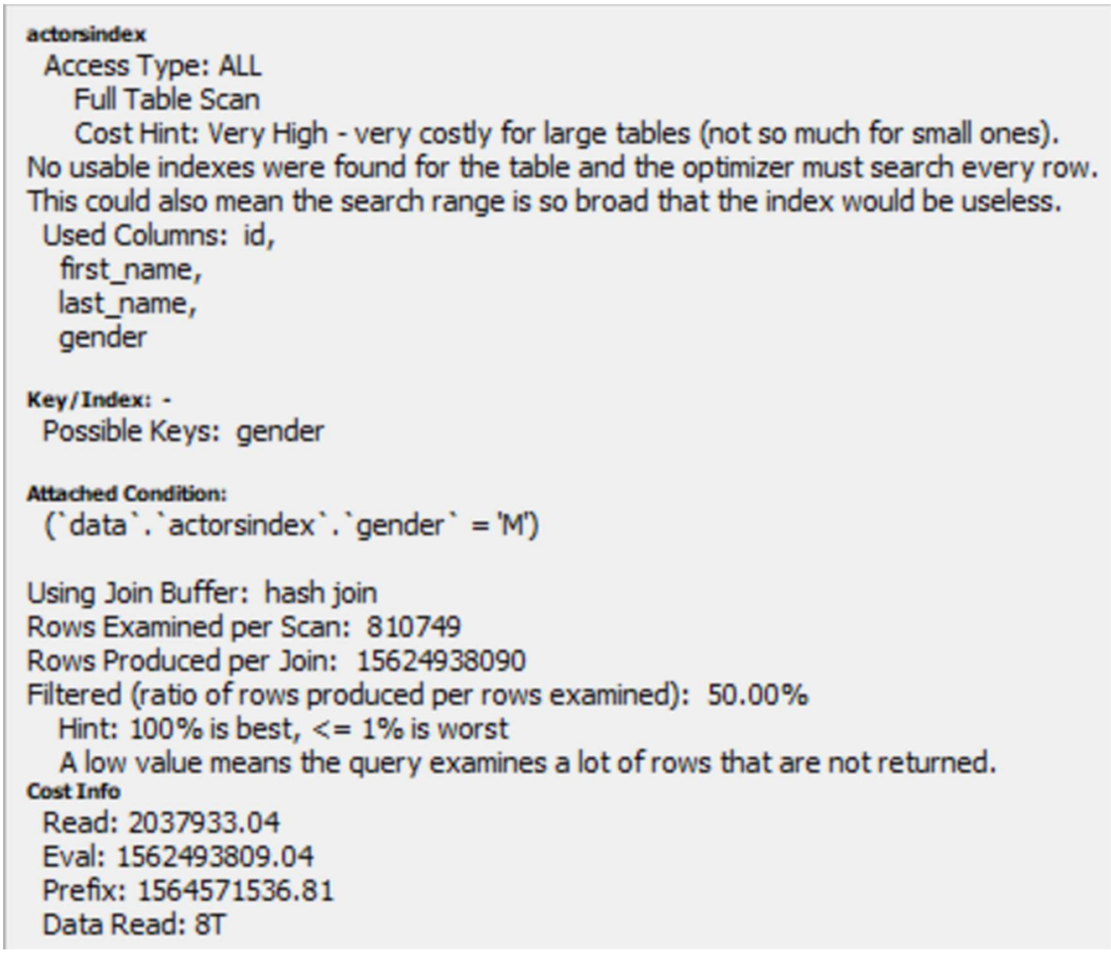
# Decrease the cost from 169779 to 83607 and the time from 1.3 to 0.7

select * from actorsindex INNER JOIN movies on movies.year=2000 and actorsindex.gender='M';



Timing (as measured at client side):
Execution time: 0:00:0.03200000

Timing (as measured by the server):
Execution time: 0:00:0.09641520
Table lock wait time: 0:00:0.00050900

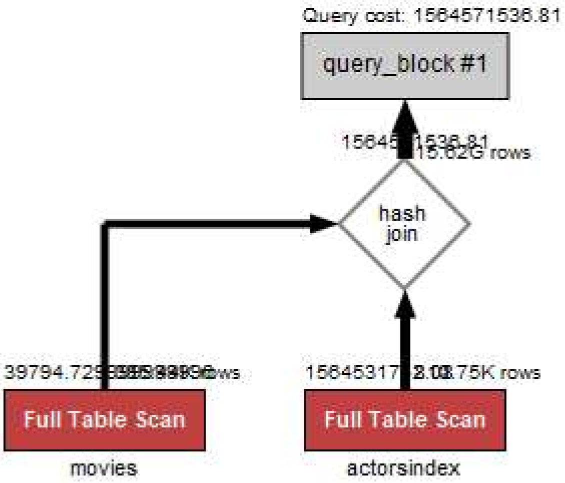


actorsindex
Access Type: ALL
Full Table Scan
Cost Hint: Very High - very costly for large tables (not so much for small ones).
No usable indexes were found for the table and the optimizer must search every row.
This could also mean the search range is so broad that the index would be useless.
Used Columns: id,
first_name,
last_name,
gender

Key/Index: -
Possible Keys: gender

Attached Condition:
(`data`.`actorsindex`.`gender` = 'M')

Using Join Buffer: hash join
Rows Examined per Scan: 810749
Rows Produced per Join: 15624938090
Filtered (ratio of rows produced per rows examined): 50.00%
Hint: 100% is best, <= 1% is worst
A low value means the query examines a lot of rows that are not returned.
Cost Info
Read: 2037933.04
Eval: 1562493809.04
Prefix: 1564571536.81
Data Read: 8T



# Increases the cost from 635286401 To 1564571536 (double) and the data read rise from 3T to 8T, but the time decreases from 0.2 to 0.03.

In conclusion:

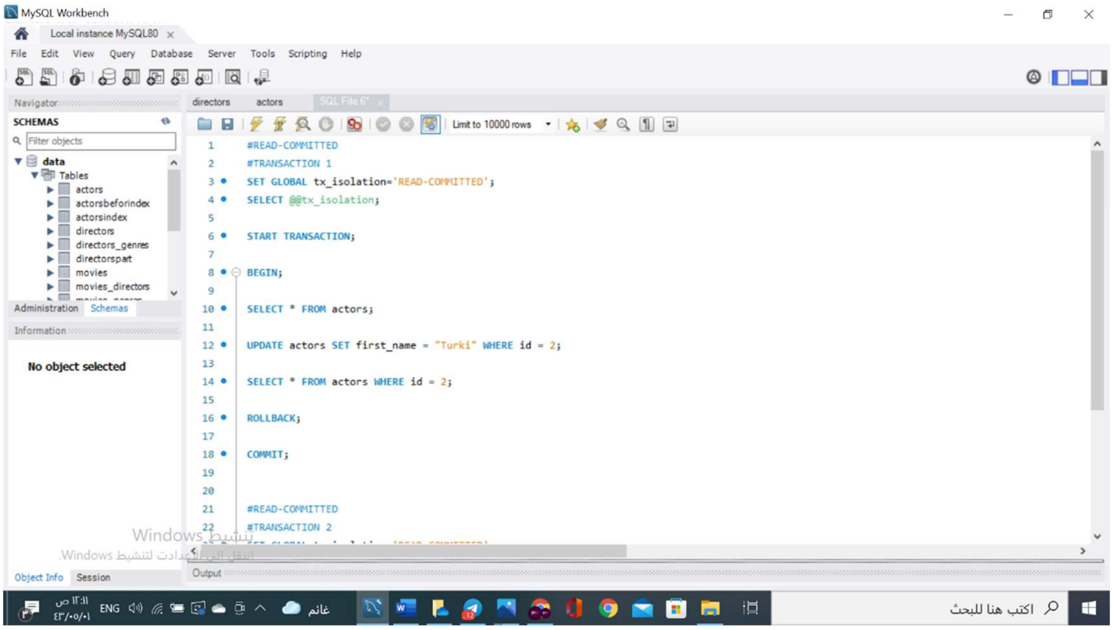
Indexing decreases the time and cost, but the overhead is any change of data on the table cause change to index. In general, indexing makes insert more complex.

transactions with isolation level:

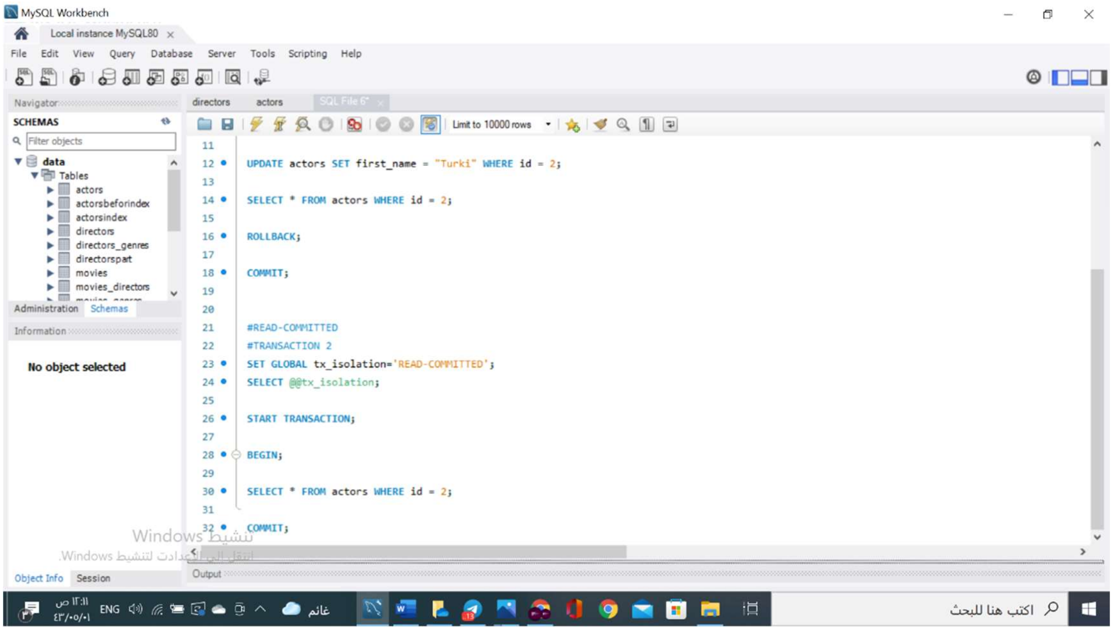
this isolation level was chosen, and with these transactions, the transaction 2 reads the original value before transaction 1 because we add ROLLBACK in the

transaction 1 and the commit in all of them, because of that we prevent dirty reads.

Challenge faced us was how to implement the transactions and isolation level in MYSQL.



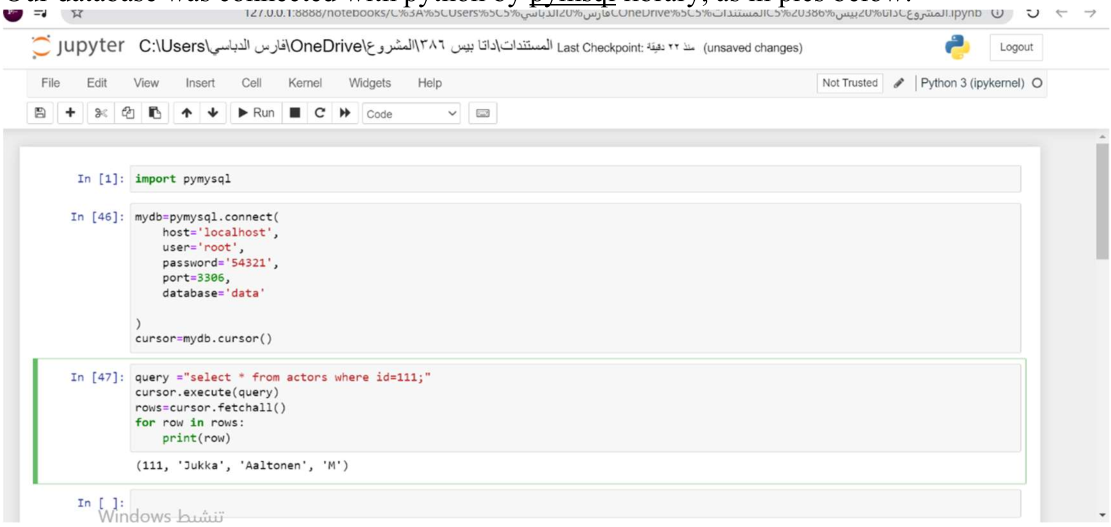
```
#READ-COMMITTED
#TRANSACTION 1
SET GLOBAL tx_isolation='READ-COMMITTED';
SELECT @@tx_isolation;

START TRANSACTION;

BEGIN;

SELECT * FROM actors;

UPDATE actors SET first_name = "Turki" WHERE id = 2;

SELECT * FROM actors WHERE id = 2;

ROLLBACK;

COMMIT;


#READ-COMMITTED
#TRANSACTION 2
```



```

UPDATE actors SET first_name = "Turki" WHERE id = 2;

SELECT * FROM actors WHERE id = 2;

ROLLBACK;

COMMIT;


#READ-COMMITTED
#TRANSACTION 2
SET GLOBAL tx_isolation='READ-COMMITTED';
SELECT @@tx_isolation;

START TRANSACTION;

BEGIN;

SELECT * FROM actors WHERE id = 2;

COMMIT;
```

Connecting with an external application:

Our database was connected with python by pymsql library, as in pics below:



jupyter
C:\Users\فارس الدباسي\OneDrive\المشروع\٣٨٦\داتا بيس المستندات
Last Checkpoint: منذ ٢٢ دقيقة (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run
Code

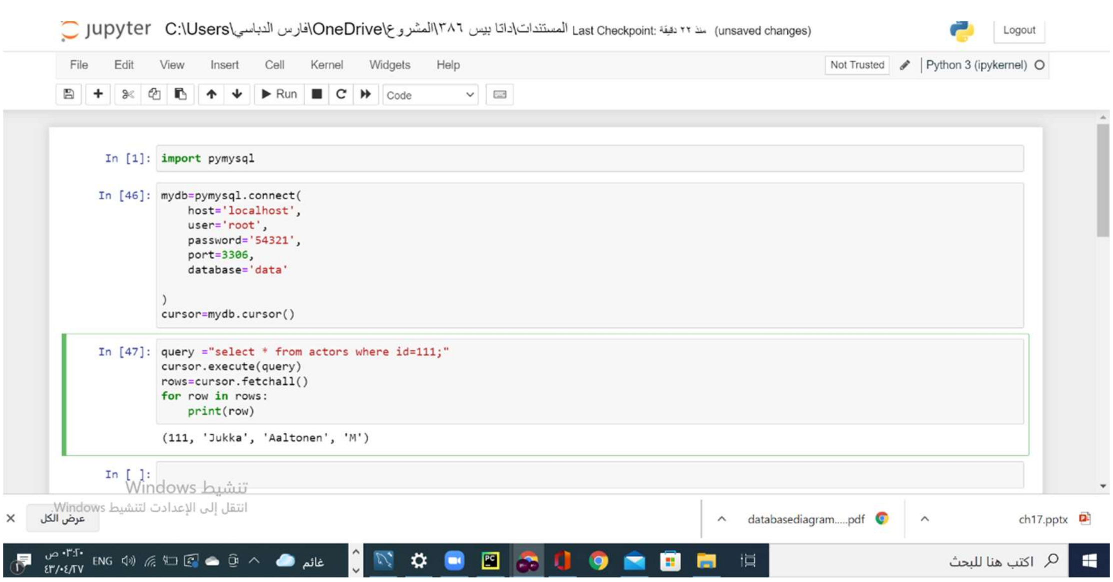
In [1]:
```
import pymysql
```

In [46]:
```
mydb=pymysql.connect(
    host='localhost',
    user='root',
    password='54321',
    port=3306,
    database='data'

)
cursor=mydb.cursor()
```

In [47]:
```
query ="select * from actors where id=111;"
cursor.execute(query)
rows=cursor.fetchall()
for row in rows:
    print(row)
```

```
(111, 'Jukka', 'Aaltonen', 'M')
```

In [ ]:



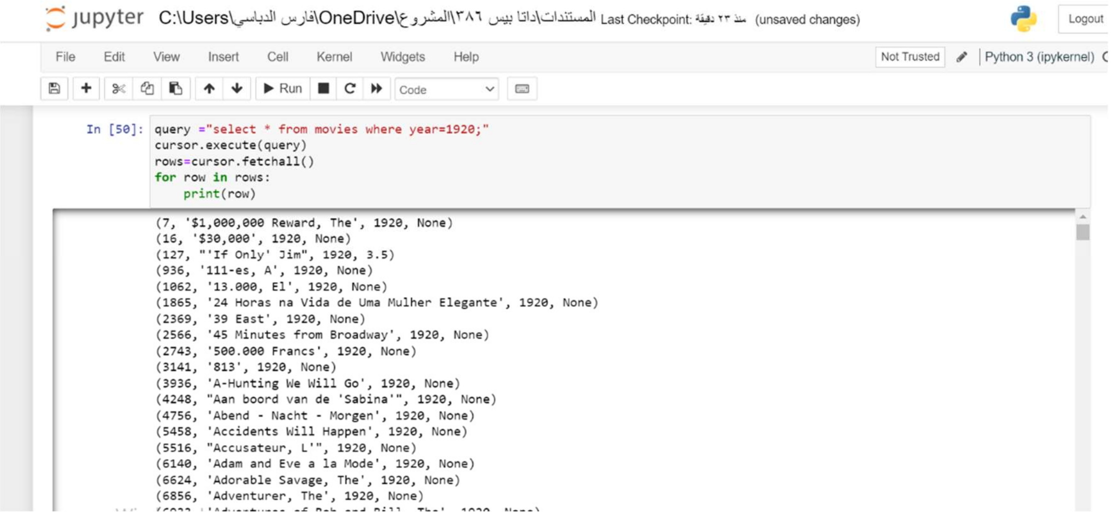
In [1]:
```
import pymysql
```

In [46]:
```
mydb=pymysql.connect(
    host='localhost',
    user='root',
    password='54321',
    port=3306,
    database='data'

)
cursor=mydb.cursor()
```

In [47]:
```
query ="select * from actors where id=111;"
cursor.execute(query)
rows=cursor.fetchall()
for row in rows:
    print(row)
```

```
(111, 'Jukka', 'Aaltonen', 'M')
```

In [ ]:


jupyter
C:\Users\فارس الدباسي\OneDrive\المشروع\٣٨٦ داتا بيس\المستندات Last Checkpoint: منذ ٢٣ دقيقة (unsaved changes)
Logout
File
Edit
View
Insert
Cell
Kernel
Widgets
Help
Not Trusted
Python 3 (ipykernel)
Run
Code

In [50]:
```
query ="select * from movies where year=1920;"
cursor.execute(query)
rows=cursor.fetchall()
for row in rows:
    print(row)
```

```
(7, '$1,000,000 Reward, The', 1920, None)
(16, '$30,000', 1920, None)
(127, "'If Only' Jim", 1920, 3.5)
(936, '111-es, A', 1920, None)
(1062, '13.000, El', 1920, None)
(1865, '24 Horas na Vida de Uma Mulher Elegante', 1920, None)
(2369, '39 East', 1920, None)
(2566, '45 Minutes from Broadway', 1920, None)
(2743, '500.000 Francs', 1920, None)
(3141, '813', 1920, None)
(3936, 'A-Hunting We Will Go', 1920, None)
(4248, "Aan boord van de 'Sabina'", 1920, None)
(4756, 'Abend - Nacht - Morgen', 1920, None)
(5458, 'Accidents Will Happen', 1920, None)
(5516, "Accusateur, L'", 1920, None)
(6140, 'Adam and Eve a la Mode', 1920, None)
(6624, 'Adorable Savage, The', 1920, None)
(6856, 'Adventurer, The', 1920, None)
(6933, 'Adventures of Bob and Bill, The', 1920, None)
```