# More interesting queries (milestone 3)

In this chapter, we present the SQL queries developed for the third milestone (called “more interesting queries”).

## SQL code

Here are the SQL codes for the more interesting queries.

### Query (a)

Original assignment was as follows.

*“Find the actors and actresses (and report the productions) who played in a production where they were 55 or more year older than the youngest actor/actress playing”*

Here is the corresponding SQL query we developed.

XXX

### Query (b)

Original assignment was as follows.

*“Given an actor, compute his most productive year”*

Here is the corresponding SQL query we developed.

XXX

### Query (c)

Original assignment was as follows.

*“Given a year, list the company with the highest number of productions in each genre. Only production company, a company per gender”*

Here is the corresponding SQL query we developed.

XXX

### Query (d)

Original assignment was as follow.

*“Compute who worked with spouses/children/potential relatives on the same production. You can assume that the same real surname (last name) implies a relation”*

Here is the corresponding SQL query we developed.

XXX

### Query (e)

Original assignment was as follow.

*“Compute the of average number of actors per production per yea”*

Here is the corresponding SQL query we developed.

XXX

### Query (f)

Original assignment was as follow.

*“Compute the average number of episodes per season”*

Here is the corresponding SQL query we developed.

XXX

### Query (g)

Original assignment was as follow.

*“Compute the average number of seasons per series”*

Here is the corresponding SQL query we developed.

XXX

### Query (h)

Original assignment was as follow.

*“Compute the top ten tv-series (by number of seasons)”*

Here is the corresponding SQL query we developed.

XXX

### Query (i)

Original assignment was as follow.

*“Compute the top ten tv-series (by number of episodes per season)”*

Here is the corresponding SQL query we developed.

XXX

### Query (j)

Original assignment was as follow.

*“Find actors, actresses and directors who have movies (including tv movies and video movies) released after their death”*

Here is the corresponding SQL query we developed.

XXX

### Query (k)

Original assignment was as follow.

*“For each year, show three companies that released the most movies”*

Here is the corresponding SQL query we developed.

XXX

### Query (l)

Original assignment was as follow.

*“List all living people who are opera singers ordered from youngest to oldest”*

Here is the corresponding SQL query we developed.

XXX

### Query (m)

Original assignment was as follow.

*“L* *ist 10 most ambiguous credits (pairs of people and productions) ordered by the degree of ambiguity. A credit is ambiguous if either a person has multiple alternative names or a production has multiple alternative titles. The degree of ambiguity is a product of the number of possible names (real name + all alternatives) and the number of possible titles (real + alternatives)*

*”*

Here is the corresponding SQL query we developed.

XXX

### Query (n)

Original assignment was as follow.

*“For each country, list the most frequent character name that appears in the productions of a production company (not a distributor) from that country”*

Here is the corresponding SQL query we developed.

XXX

## Necessity of indexes

We are going to explain the necessity of indexes by choosing 3 queries of the milestone 3 and comparing the gain obtained using indexes.

Unfortunately *MySQL* doesn’t propose a tool like *Oracle* to show the exact execution plan where we can see exactly which can of join is used and others interesting details. However, we have used the command *EXPLAIN EXTENDED* and the visualized of *MySQLWorkbench*. More informations about those outputs can be found on <https://dev.mysql.com/doc/refman/5.5/en/explain-output.html#explain-join-types> and <https://dev.mysql.com/doc/refman/5.5/en/explain-extended.html> .

The running time obtained for those 3 queries were measured using a *MacBook Pro mid-2010*.

### Query(e)

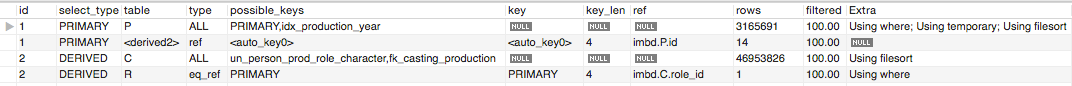
First, we recall the Query(e) :

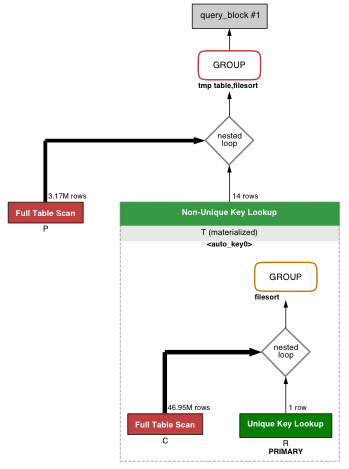
XXX

We can see that we have a temporary table (which is called a derived in *MySQL*) containing a join between the table *casting* and *role*. Once the temporary table is created, we join it with the table *production*.

Withou any indexes

Let’s have a check of the execution plan without any indexes except the primary keys.

**

**

For the table *role*, we have to do a look-up to find the *id* corresponding to an actor. We can create an index on the filed *role.name* to reduce the time to know which *id* is an actor. However the gain won’t be so much comparing to the other indexes.

For the table *casting*, we have to do a full-scan because the rows aren’t clustered by *role\_id*. The join operation will be high due to the lack of index on this field because we have to check every row (~45 millions) to proceed the join.

We can constat that for the table *production*, we have to do a full-scan because we don’t have any information about the location of the production which have a year. Moreover, the group by year is costly because we have to create a temporary table to sort the production by year.

With those observation, we can create indexes to solve those problems. Without any indexes, we obtains a time of **212 seconds**.

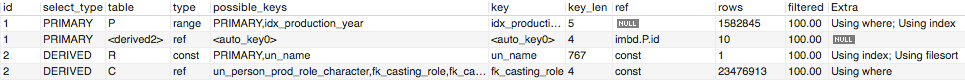
For the rest, using the primary key helps, because otherwise the time would be incredibly high !

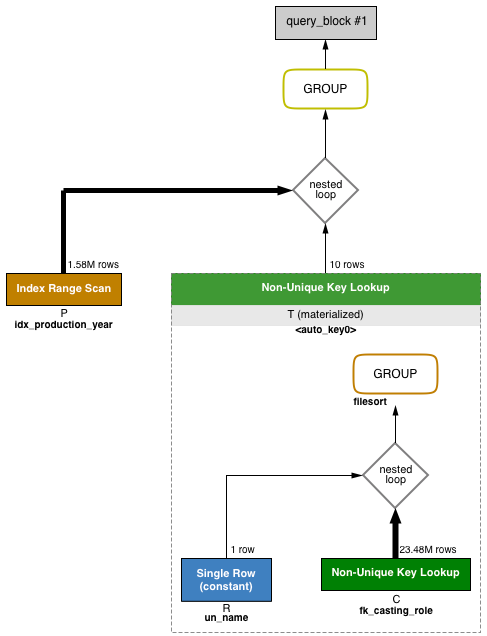
With indexes

We have create the following indexes on :

* *casting.role\_id* to improve the join operation
* *role.name* to improve the look-up to see which *id* corresponds to an actor
* *production.year* to improve the group by part

With those indexes, we obtain the below execution plan :





What have changed comparing to the other one ?

Let’s first talk about the index *production.year*. Before, we had an full-scan, filesort and temporary table (for the group by). Now, we have a range-scan (which is more efficient) and don’t need to use a filesort with a temporary table for the group by because the field *year* is indexed. Using the primary key wouldn’t have been a good idea because we still need a full scan to obtain the production which have a year !

For the temporary table (derived2), it doesn’t change anything because we didn’t have indexes before (because this table doesn’t exists) and it doesn’t worthwile to create a table with indexes during the query processing.

If we compare the row for *role*, we can observe we passed from *eq\_ref* (which means that we use and equality operator to find the corresponding row, which was a good join) to *const* (because of the index on *role.name*) which is much better because there is only one row, it can be considered as constant by the optimized and so reducing the accessing time to read this value.

Finally, for the table *casting*, the index *casting.role\_id* helps a lot for the join operation, passing from a full-scan to a ref, which is used for the operator “=” or “< = >”, which is much better than a full-scan. Remember that casting has 45 millions of rows so we gain a lot in performances.

Another intesting details is the number of rows checked ! You can see the differences between the case without and with the indexes. For *production* table, we seek 2 times less rows and 2 times less rows for *casting*, wich correspond to ~25millions of rows in total !

With all those indexes, we have reach a time of **102 seconds**, wich is a **speed up of 2.08** of the initial time,. We might ask ourself if it is better to use an index on *casting.production\_id* rather *casting.role\_id*, improving the group-by but not the join part. We measure the time and we obtains didn’t wait until it finished because 10 minutes was already left, which is catastrophic comparing to the index on the *role\_id* field !

Finally, we can ask ourself whether other indexes could help. With this manner of writing query, we don’t think we can find better indexes (to convince ourself, it is enough to see the query plan). Maybe it is better to obtain better performances without using a temporary table, we have tried and have obtained worse performances.

### Query(j)

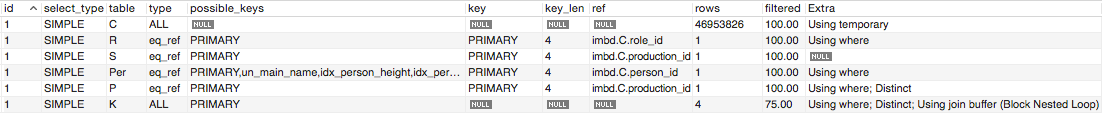
First, we recall the Query(j) :

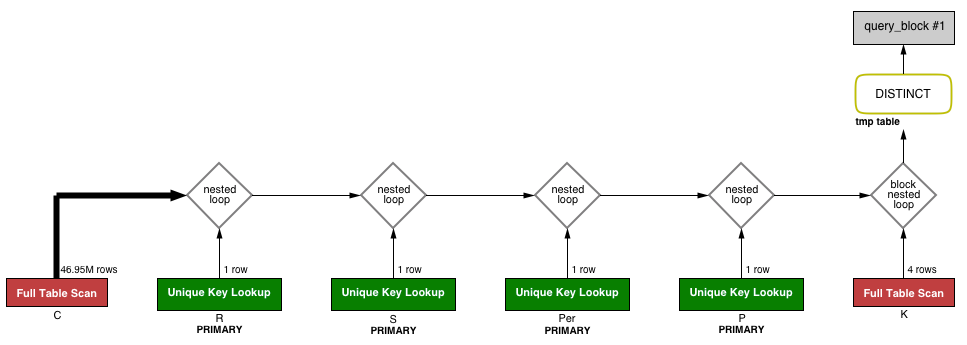
XXX

Because we have normalized everything, we have a lot of joins. However, the primary keys are used which help a lot for the efficiency. Finally, we have 4 “where conditions” which help reducing the running time of the query.

Without indexes

Let’s have a check of the execution plan without any indexes except the primary keys.





For the table *singleproduction, person* and *production*, the primary keys are used and so we cannot improve this part in the join. However, for the table *casting*, we don’t have any information about the foreign keys for *casting*. Moreover, as before, it is more efficient to use and index on *role.name* rather its primary key. For the table *kind*, we will add an index on the field *name*, because we can observe that only 75% of the data are returned using the primary key.

To improve this part, it is necessary to use an index on the field *role.name* (rather than the primary key) and we should choose the index on the field *casting.role\_id* for the same reason as before, rather choosing an index on *casting.production\_id.*

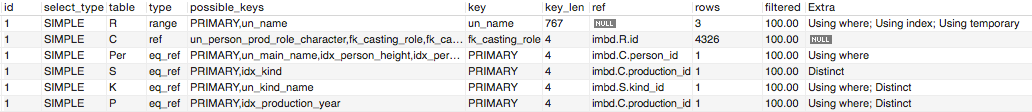
With those observation, we can create indexes to solve those problems. Without any indexes, we obtains a time of **30 seconds**.

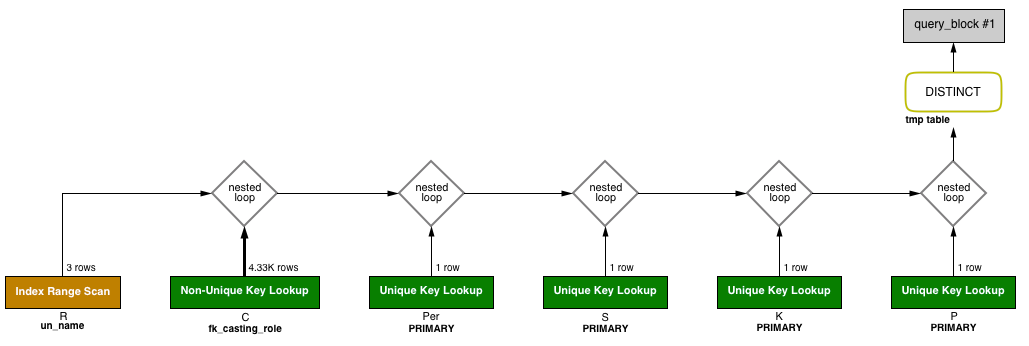
With indexes

We have create the following indexes on :

* *casting.role\_id* to improve the join operation
* *role.name* to improve the look-up to see which *id* corresponds to an actor/actress/director
* *kind.name* to improve the look-up to see which *id* corresponds to the specified kind of movies

With those indexes, we obtain the below execution plan :





What are the differences ?

We can see we avoid the full-scan of *casting* and only 4326 rows are analyzed (reduced by a factor 10’000) which is a very big gain !

For the table *role* we don’t use anymore the primary key and use the index on *role.name* which leads to a range-scan. Because *role* is very small (we checked before 1 row and now 3 rows, which is very small), it won’t be worse to use it and it will be better for the same reason as said before !

The table *person* and *production* hasn’t changed their execution. The table *singleproduction* kept the same index but has to use a distinct now. We didn’t find an explanation for this latter.

Finally, the table *kind* didn’t use our index. The reason is surely the order of the join and the optimized one has seen that only 1 row was possible at this step. Maybe with different data in the table *kind* (in fact we have only 4 entries and we look up for 3), the index on *kind.name* may be used for the same reason as *role.name* (where *role* has 11 entries).

With all those indexes, we have reach a time of **9 seconds**, wich is a **speed up of 3.33.** Those indexes were quite trivial because a lot of primary keys are used. However with those 2 indexes (and 1 not used), we obtain a more efficient query,

### Query(m)

First, we recall the Query(m) :

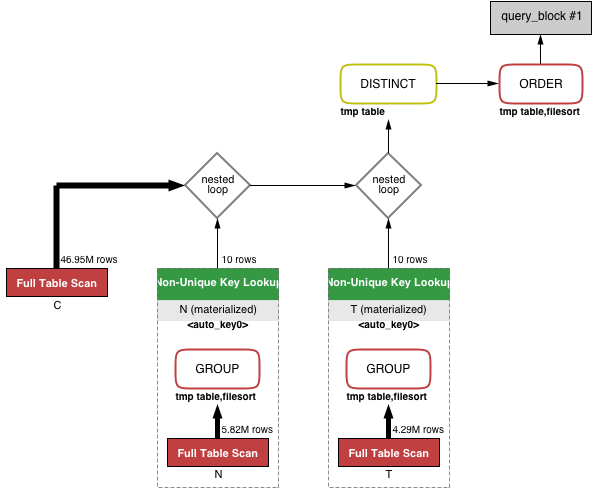
XXX

Because tables *name*, *title* and *casting* are big (6 millions, 45 millions, 3.5 millions), we cannot do directly a join on those 3 titles (as again due to our massive normalization). So, it is necessary to create temporary tables which will have less rows !

Without indexes

Let’s have a check of the execution plan without any indexes except the primary keys.





The first thing we can observe is that we have a full-scan for the 3 tables, which is enormous (55 millions in total). No keys at all are used in this query !

Let’s begin with the table *casting*. We join directly from *casting.person\_id* to *name.person\_id* and we do the same for the titles. It should be very interesting to add a double index on the fields (*casting.person\_id, casting.production\_id*) ! It will help a lot for the join operation which is very big.

For the temporary tables, we won’t create any indexes because it will be faster to use them directly rather to take time to create indexes. The number of rows for the temporary table of *name* and *title* are respectly 500’000 and 200’00.

For the table *name, title*, their behavior is similar. To count the number of rows, we have to use a temporary table which is costly. Interesting indexes should be on *name.person\_id* and *title.production\_id*. This will help a lot for the grouping part (that’s need a temporary table with a filesort).

Unfortunately, for the last distinct and order, we cannot use indexes to improve this part.

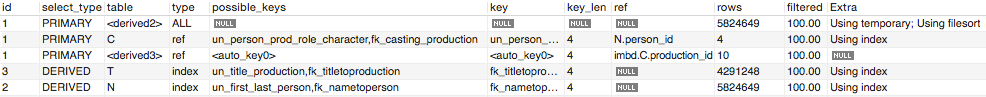
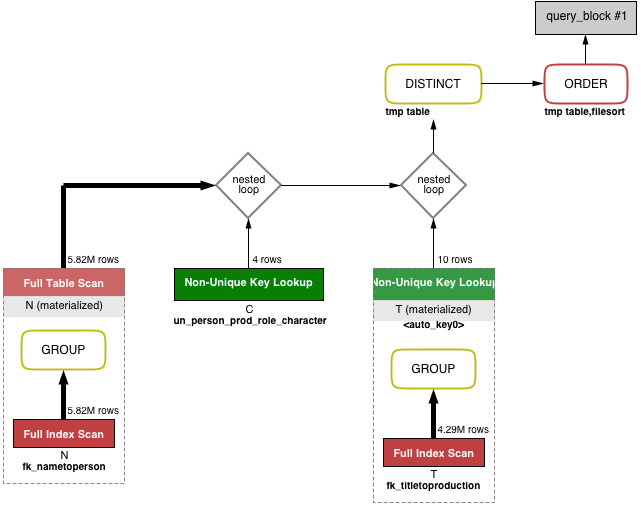
With those observation, we can create indexes to solve those problems. Without any indexes, we obtains a time of **301 seconds**.

With indexes

We have create the following indexes on :

* *casting.(person\_id, production\_id)* to improve the join operation
* *name.person\_id* to improve the group by part
* *title.production\_id* to improve the group by part

With those indexes, we obtain the below execution plan :

First thing we can see is that we have avoid a full-scan to the table *casting*, which will be a big gain in the execution time ! Moreover the number of rows has been a lot reduced to reach the number 4.

For the tables *name, title* with their index, we have improved the group-by part which doesn’t need anymore a temporary table with a filesort, which is also a good thing for the running time.

The table *derived3* hasn’t changed at all. However, table *derived2* now need a full-scan (and still the temporary table and the filesort to do the last distinct and order). We didn’t find an explanation but if we analyze the total number of row, it is ~15millions comparing to the 55millions before.

With all those indexes, we have reach a time of **42 seconds**, wich is a **speed up of 7.17.**

## Running time of all queries

All queries have been executed on a *MacBook Pro Retina late 2013*. 2 kinds of runs have been used : the first one is an execution without any cache and the second one with cache (which is more realistic in a website due to the number of requests done by visitors). For the rest of details, please refer to chapter **XX (config MySQL ….)**.

Milestone 2

|  |  |  |
| --- | --- | --- |
| Query / Time [sec] | Without cache | With cache |
| Query a |  | 2 |
| Query b |  | 5 |
| Query c |  | 360 |
| Query d |  | 88 |
| Query e |  | 0.7 |
| Query f |  | 0.7 |
| Query g |  | 5.7 |

Queries c and d have been improved to use materialized views which are update every day one time, in order to have a execution time (simply a fetch in the materialized views) of few seconds. However, we kept the running time of the whole query. In the case of materialized views, it is a bit longer because we have to store the results in a new table.

Milestone 3

|  |  |  |
| --- | --- | --- |
| Query / Time [sec] | Without cache | With cache |
| Query a |  | 45 |
| Query b |  | 0.001 |
| Query c |  | 2.1 |
| Query d |  | 3.6 |
| Query e |  | 52 |
| Query f |  | 0.5 |
| Query g |  | 0.07 |
| Query h |  | 0.038 |
| Query i |  | 0.6 |
| Query j |  | 6 |
| Query k |  | 75 |
| Query l |  | 4 |
| Query m |  | 25 |
| Query n |  | 934 |

Queries k and n have been improved to use materialized views in the same manner as for the milestones 2.

## Distribution of the cost

After improving the 3 specified queries in this chapter, we are going to try giving a rational explanation about the distribution of the cost. Because *MySQL* doesn’t propose a tool like *Oracle* to show the exact execution plan with the time needed for each part.

We won’t put the SQL code and the query execution plan again, but we will refer to the same as put earlier in this chapter.

When we talk about times in this chapter, we reference about the time obtained where we described the necessity of indexes.

### Query(e)

Let’s start with the sub-query. The final temporary table has 2.3 millions rows. It is much better to have this temporary table rather to directly do a join between *casting, role* and *production*. Creating indexes for this temporary isn’t worthiwile to its size.

Due to the index on *casting.production\_id*, the grouping part is quite fast alone but because we have a join on the table *role*, we need to do a filesort. The other index on *role.name* is very useful because the *id* of the role actor is considered as constant by the optimizer. This sub-query (if executed alone) is very fast (less than few seconds).

The place where we loose most of the time (we would say around 90%) is writing the result of this subquery inside a temporary table and then doing a join with the table *production*. The reason is that the last clause where is not enough strong to reduce the number of join to do. The production which hasn’t a year represent only 4.23% of the production. After the join, it represents only 1.6% over all the production in the join which is clearly nothing. However, if the percentage was higher (~20-40%) the time will be a lot reduced (remind, the index *production.year*).

Finally, we still have the group by but it is optimized because we don’t need to use a temporary table nor a filesort (due to the index on *production.year*).

### Query(j)

This query is 5 joins which is quite big (especially with tables like *casting, production, person*). However, we have 4 where conditions which will limit a lot the number of final output (10’000).

Due to primary keys for tables *person, singleproduction, kind, production* and the indexes on *casting.role\_id* and *role.name*, the time of the join is a lot reduced.

We still have the distinct at the end which need a temporary table to eliminate the duplicate, which takes also times but less than the join of course. We have tried to replaced it with a group by but it was more efficient to use distinct rather group by because the group by need a temporary table and a file sort

### Query(m)

First thing we can see which takes a lot of time are our 2 temporary tables (so no indexes) due to the sub-queries. Remember they contains 500’000 and 200’000 rows with which we have to do a join operation with *casting*. Those steps are costly but less than the distinct/order part, it takes approximately 10% of the time (4.6 seconds).

Hopefully, those temporary table use indexes on *name.person\_id* and *production.title\_id* which reduces their execution time (as said earlier in this chapter) especially for the group by part.

Finally, we have to dinstinct them (which need a temporary table) and last step is to sort them which needs another temporary table and a filesort. The bottleneck is clearly here because have 3 millions entries for the distinct (which takes few seconds), which becomes 2.7 millions and finally the order part which is the bottleneck which takes the rest of the time.

Unfortunately, to improve the order part, we cannot make any assumption about the temporary tables (such that the person/production with the highest score has a high score for the name and the title).